

Helmut Herold

4. Auflage

Linux/Unix- Systemprogrammierung



ADDISON-WESLEY

LINUX / UNIX und seine Werkzeuge

bisher erschienen:

Helmut Herold: LINUX-UNIX-Grundlagen

Helmut Herold: LINUX-UNIX-Profitools

Helmut Herold: LINUX-UNIX-Shells

Helmut Herold: LINUX-UNIX-Systemprogrammierung

Helmut Herold: LINUX-UNIX-Kurzreferenz

Helmut Herold

LINUX-UNIX-System- programmierung

4., überarbeitete Auflage



An imprint of Addison Wesley Longman, Inc.

Bonn • Reading, Massachusetts • Menlo Park, California
New York • Harlow, England • Don Mills, Ontario
Sydney • Mexico City • Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Herold, Helmut:

Linux-Unix-Systemprogrammierung : Helmut Herold. –

4., überarb. Aufl. – Bonn ; Rending, Mass. [u. a.] : Addison-Wesley-Longman, 2002.

(Linux / Unix und seine Werkzeuge)

ISBN 3-8273-1512-3

Buch: GB

© 2002 Addison-Wesley (Deutschland) GmbH, A Pearson Education Company

4., überarbeitete Auflage 2002

Lektorat: Susanne Spitzer und Andrea Stumpf, München

Satz: Reemers EDV-Satz, Krefeld. Gesetzt aus der Palatino 9,5 Punkt

Belichtung, Druck und Bindung: Kösel GmbH, Kempten

Produktion: TYPisch Müller, München

Umschlaggestaltung: Hommer Grafik-Design, Haar bei München

Das verwendete Papier ist aus chlorfrei gebleichten Rohstoffen hergestellt und alterungsbeständig. Die Produktion erfolgt mit Hilfe umweltschonender Technologien und unter strengsten Auflagen in einem geschlossenen Wasserkreislauf unter Wiederverwertung unbedruckter, zurückgeführter Papiere.

Text, Abbildungen und Programme wurden mit größter Sorgfalt erarbeitet. Verlag, Übersetzer und Autoren können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Kein Teil dieses Buches darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form durch Fotokopie, Mikrofilm oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere Datenverarbeitungsanlagen, verwendbare Sprache übertragen werden. Auch die Rechte der Wiedergabe durch Vortrag, Funk und Fernsehen sind vorbehalten.

Die in diesem Buch erwähnten Software- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

Inhaltsverzeichnis

Einleitung	1
Gliederung dieses Buches	1
Unix-Standards und -Implementierungen	7
Beispiele und Übungen	7
Hinweis zur Buchreihe: Unix und seine Werkzeuge	7
1 Überblick über die Unix-Systemprogrammierung	9
1.1 Anmelden am Unix-System	9
1.2 Dateien und Directories	11
1.3 Ein- und Ausgabe	17
1.4 Prozesse unter Unix	21
1.5 Ausgabe von System-Fehlermeldungen	26
1.6 Benutzerkennungen	28
1.7 Signale	29
1.8 Zeiten in Unix	32
1.9 Unterschiede zwischen Systemaufrufen und Bibliotheksfunktionen	33
1.10 Unix-Standardisierungen und -Implementierungen	35
1.11 Limits	39
1.12 Erste Einblicke in den Linux-Systemkern	52
1.13 Übung	99
2 Überblick über ANSI C	101
2.1 Allgemeines	101
2.2 Der Präprozessor	106
2.3 Die Sprache ANSI C	114
2.4 Die ANSI-C-Bibliothek	124
2.5 Übung	160
3 Standard-E/A-Funktionen	167
3.1 Der Datentyp FILE	167
3.2 stdin, stdout und stderr	168

3.3	Öffnen und Schließen von Dateien	168
3.4	Lesen und Schreiben in Dateien	172
3.5	Pufferung	200
3.6	Positionieren in Dateien	204
3.7	Temporäre Dateien	207
3.8	Löschen und Umbenennen von Dateien	212
3.9	Ausgabe von Systemfehlermeldungen	214
3.10	Übung	216
4	Elementare E/A-Funktionen	221
4.1	Filedeskriptoren	221
4.2	Öffnen und Schließen von Dateien	222
4.3	Lesen und Schreiben in Dateien	229
4.4	Positionieren in Dateien	233
4.5	Effizienz von E/A-Operationen	237
4.6	Kerntabellen für offene Dateien	240
4.7	File Sharing und atomare Operationen	241
4.8	Duplizieren von Filedeskriptoren	245
4.9	Ändern oder Abfragen der Eigenschaften einer offenen Datei	247
4.10	Filedeskriptoren und der Datentyp FILE	253
4.11	Das Directory /dev/fd	259
4.12	Übung	260
5	Dateien, Directories und ihre Attribute	263
5.1	Dateiattribute	263
5.2	Dateiarten	265
5.3	Zugriffsrechte einer Datei	267
5.4	Eigentümer und Gruppe einer Datei	281
5.5	Partitionen, Filesysteme und i-nodes	282
5.6	Symbolische Links	297
5.7	Größe einer Datei	303
5.8	Zeiten einer Datei	307
5.9	Directories	311
5.10	Geräte-dateien	325
5.11	Der Puffercache	327
5.12	Realisierung von Filesystemen unter Linux	329
5.13	Übung	364

6	Informationen zum System und seinen Benutzern	369
6.1	Informationen aus der Paßwortdatei	369
6.2	Informationen aus der Gruppendatei	374
6.3	Informationen aus Netzwerkdateien	377
6.4	Informationen zum lokalen System	378
6.5	Informationen zu Systemanmeldungen	380
6.6	Übung	381
7	Datums- und Zeitfunktionen	385
7.1	Datentypen und Konstanten	385
7.2	Datums- und Zeitfunktionen	386
7.3	Übung	401
8	Nicht-lokale Sprünge	403
8.1	Die Headerdatei <setjmp.h>	403
8.2	Übung	416
9	Der Unix-Prozeß	419
9.1	Start eines Unix-Prozesses	419
9.2	Beendigung eines Unix-Prozesses	421
9.3	Environment eines Unix-Prozesses	427
9.4	Speicherbelegung eines Unix-Prozesses	431
9.5	Ressourcenlimits eines Unix-Prozesses	439
9.6	Ressourcenbenutzung eines Unix-Prozesses	443
9.7	Die Speicherverwaltung unter Linux	445
9.8	Übung	477
10	Die Prozeßsteuerung	483
10.1	Prozeßkennungen und die Unix-Prozeßhierarchie	483
10.2	Kreieren von neuen Prozessen	486
10.3	Warten auf Beendigung von Prozessen	502
10.4	Synchronisationsprobleme zwischen Eltern- und Kindprozessen	515
10.5	Die exec-Funktionen	520
10.6	Die Funktion system	527
10.7	Ändern der User-ID und Group-ID eines Prozesses	532
10.8	Informationen zu Prozessen	537
10.9	Übung	545

11 Attribute eines Prozesses (Kontrollterminal, Prozeßgruppe und Session)	549
11.1 Loginprozesse	549
11.2 Prozeßgruppen	554
11.3 Session	556
11.4 Kontrollterminals, Sessions und Prozeßgruppen	557
11.5 Jobkontrolle und Programmausführung durch die Shell	559
11.6 Verwaiste Prozeßgruppen	565
11.7 Übung	566
12 Blockierungen und Sperren von Dateien	567
12.1 Blockierende und nichtblockierende E/ A-Operationen	567
12.2 Sperren von Dateien (record locking)	568
12.3 Übung (Multiuser-Datenbankbibliothek)	583
13 Signale	599
13.1 Das Signalkonzept und die Funktion signal	599
13.2 Signalnamen und Signalnummern	607
13.3 Probleme mit der signal-Funktion	616
13.4 Das neue Signalkonzept	618
13.5 Senden von Signalen mit den Funktionen kill und raise	628
13.6 Einrichten einer Zeitschaltuhr und Suspendieren eines Prozesses ..	630
13.7 Anormale Beendigung mit Funktion abort	648
13.8 Zusätzliche Argumente für Signalhandler	650
13.9 Übung	651
14 STREAMS in System V	655
14.1 Allgemeines zu STREAMS	655
14.2 STREAM-Messages	657
14.3 Übung	669
15 Fortgeschrittene Ein- und Ausgabe	671
15.1 E/ A-Multiplexing	671
15.2 Asynchrone E/ A	681
15.3 Memory Mapped I/O	683
15.4 Weitere read- und write-Funktionen	695
15.5 Übung	699

16 Dämonprozesse	703
16.1 Typische Unix-Dämonen	703
16.2 Besonderheiten von Dämonen	704
16.3 Schreiben von eigenen Dämonen	705
16.4 Fehlermeldungen von Dämonen	707
16.5 Übung	714
17 Pipes und FIFOs	717
17.1 Überblick über die unterschiedlichen Arten der Interprozeßkommunikation	717
17.2 Pipes	718
17.3 Benannte Pipes (FIFOs)	744
17.4 Übung	749
18 Message-Queues, Semaphore und Shared Memory	753
18.1 Allgemeine Strukturen und Eigenschaften	753
18.2 Message-Queues	756
18.3 Semaphore	770
18.4 Shared Memory	780
18.5 Übung	800
19 Stream Pipes, Client-Server-Realisierungen und Netzwerkprogrammierung	805
19.1 Client-Server-Eigenschaften der klassischen IPC-Methoden	805
19.2 Stream Pipes	807
19.3 Austausch von Filedeskriptoren zwischen Prozessen	811
19.4 Client-Server-Realisierung mit verwandten Prozessen	823
19.5 Benannte Stream Pipes	828
19.6 Client-Server-Realisierung mit nicht verwandten Prozessen	845
19.7 Netzwerkprogrammierung mit TCP/IP	856
19.8 Übung	877
20 Terminal-E/A	879
20.1 Charakteristika eines Terminals im Überblick	879
20.2 Terminalattribute und Terminalidentifizierung	887
20.3 Spezielle Eingabezeichen	896
20.4 Terminalflags	900
20.5 Baudraten von Terminals	908
20.6 Zeilensteuerung bei Terminals	910

20.7	Kanonischer und nicht-kanonischer Modus	912
20.8	Terminalfenstergrößen	919
20.9	termcap, terminfo und curses	921
20.10	S-Lang – Eine Alternative zu curses unter Linux	936
20.11	Die Linux-Konsole	953
20.12	Die Programmierung von virtuellen Konsolen unter Linux	985
20.13	Übung	994
21	Weitere nützliche Funktionen und Techniken	1007
21.1	Expandierung von Dateinamen	1007
21.2	String-Vergleiche mit regulären Ausdrücken	1013
21.3	Abarbeiten von Optionen auf der Kommandozeile	1023
22	Wichtige Entwicklungswerkzeuge	1055
22.1	gcc – Der GNU-C-Compiler	1055
22.2	ld – Der Linux / Unix-Linker	1060
22.3	gdb – Der GNU-Debugger	1061
22.4	strace – Mitprotokollieren aller Systemaufrufe	1067
22.5	Tools zum Auffinden von Speicherüberschreibungen und -lücken ..	1073
22.6	ar – Erstellen und Verwalten von statischen Bibliotheken	1082
22.7	Dynamische Bibliotheken	1087
22.8	make – Ein Werkzeug zur automatischen Programmgenerierung ..	1100
A	Headerdatei eighdr.h und Modul fehler.c	1123
A.1	Headerdatei eighdr.h	1123
A.2	Zentrales Fehlermeldungsmodul fehler.c	1124
B	Ausgewählte Lösungen zu den Übungen	1129
B.1	Ausgewählte Lösungen zu Kapitel 4 (Elementare E / A-Funktionen)	1129
B.2	Ausgewählte Lösungen zu Kapitel 5 (Dateien, Directories und ihre Attribute)	1130
B.3	Ausgewählte Lösungen zu Kapitel 7 (Datums- und Zeitfunktionen)	1133
B.4	Ausgewählte Lösungen zu Kapitel 8 (Nicht-lokale Sprünge)	1133
B.5	Ausgewählte Lösungen zu Kapitel 9 (Der Unix-Prozeß)	1134
B.6	Ausgewählte Lösungen zu Kapitel 10 (Die Prozeßsteuerung)	1135
B.7	Ausgewählte Lösungen zu Kapitel 11 (Attribute eines Prozesses) ..	1137
B.8	Ausgewählte Lösungen zu Kapitel 13 (Signale)	1139
B.9	Ausgewählte Lösungen zu Kapitel 14 (STREAMS in System V)	1141

B.10	Ausgewählte Lösungen zu Kapitel 15 (Fortgeschrittene Ein- und Ausgabe)	1141
B.11	Ausgewählte Lösungen zu Kapitel 16 (Dämonprozesse)	1142
B.12	Ausgewählte Lösungen zu Kapitel 17 (Pipes und FIFOs)	1142
B.13	Ausgewählte Lösungen zu Kapitel 18 (Message-Queues, Semaphore und Shared Memory)	1144
Literaturverzeichnis		1145
Stichwortverzeichnis		1149

Einleitung

*In die Tiefe mußt du steigen,
soll sich dir das Wesen zeigen.*

Schiller

Dieses Buch beschreibt die Systemprogrammierung unter Linux/Unix. Unix bietet wie jedes Betriebssystem sogenannte Systemaufrufe an, die von den Benutzerprogrammen aus aufgerufen werden können, wenn diese bestimmte Dienste vom System benötigen. Typische von einem Betriebssystem angebotene Dienste sind z.B. Öffnen einer Datei, Schreiben auf eine Datei, Bereitstellen von freiem Speicherplatz oder Kommunizieren mit anderen Programmen.

Diese Systemaufrufe werden ebenso wie andere wichtige Funktionen aus der C-Standardbibliothek in diesem Buch anhand von zahlreichen anschaulichen Beispielen ausführlich beschrieben. Praxisnahe Übungen am Ende jedes Kapitels ermöglichen dem Leser das Anwenden und Vertiefen der jeweils erworbenen Kenntnisse.

An entsprechenden Stellen wird in diesem Buch die Umsetzung von wichtigen Betriebssystemkonzepten und -algorithmen am System Linux gezeigt. Dieses System wurde nicht nur aufgrund seiner großen Beliebtheit ausgewählt, sondern auch, weil Linux alle seine Quellprogramme der Öffentlichkeit zur Verfügung stellt.

Gliederung dieses Buches

Der Inhalt dieses Buch untergliedert sich in zehn Themengebiete sowie in einen Anhang.

Einführung in die Unix-Systemprogrammierung (Kapitel 1 - 2)

Überblick über die Unix-Systemprogrammierung (Kapitel 1)

In diesem Kapitel wird zunächst ein kurzer Einblick in die Unix-Konzepte und -Begriffe gegeben, bevor ein kleiner Ausflug in die wichtigsten Gebiete der Systemprogrammierung erfolgt, um in den späteren Kapiteln auf diese Grundbegriffe Bezug nehmen zu können, ohne daß ständig eine Erklärung eines erst später behandelten Begriffes eingeschoben werden muß.

In diesem Kapitel wird darüber hinaus ein kurzer Überblick über wichtige Unix-Standards und -Systeme gegeben. Zum Abschluß bekommen Sie erste Einblicke in den Linux-Systemkern. Dieser Linux-spezifische Abschnitt ist nur für Leser gedacht, die an der Umsetzung von Betriebssystemkonzepten und -algorithmen interessiert sind oder die

selbst Kernroutinen oder systemnahe Funktionen programmieren möchten. Dieser umfangreiche Abschnitt zeigt den grundlegenden Aufbau des Linux-Systemkerns, klärt wichtige Begriffe und stellt wesentliche Kernalgorithmen und -konzepte vor, die für das Verständnis der späteren Linux-spezifischen Kapitel vorausgesetzt werden.

Überblick über ANSI C (Kapitel 2)

Da zur Linux/Unix-Systemprogrammierung die Programmiersprache C verwendet wird, wird hier ein kurzer Überblick über das heute gültige Standard-C (auch ANSI C genannt) gegeben. Dazu werden in diesem Kapitel zunächst allgemein geltende ANSI-C-Begriffe und -Konstrukte behandelt, bevor näher auf den Präprozessor und die Sprache ANSI C eingegangen wird. Am Ende dieses Kapitels wird ein Überblick über die nun standardisierten Headerdateien gegeben. Dabei werden alle von ANSI C vorgeschriebenen Konstanten, Datentypen, Makros, globale Variablen und Funktionen kurz vorgestellt, soweit diese nicht in späteren Kapiteln ausführlich behandelt werden.

Ein- und Ausgabe (Kapitel 3 - 5)

Standard-E/A-Funktionen (Kapitel 3)

Hier werden die Funktionen beschrieben, die sich in der C-Standardbibliothek befinden und in der Headerdatei `<stdio.h>` definiert sind. Die in dieser Headerdatei definierten Datentypen und Funktionen dienen der Ein- und Ausgabe auf das Terminal oder auf Dateien. Die hier vorgestellten Funktionen arbeiten mit optimal eingestellten Puffern, so daß sich der Benutzer vollständig auf seine Ein- und Ausgabe konzentrieren kann, ohne sich um solche Details kümmern zu müssen.

Elementare E/A-Funktionen (Kapitel 4)

Die hier beschriebenen elementaren E/A-Funktionen leisten ähnliches wie die Standard-E/A-Funktionen, nur daß sie als systemnahe Funktionen nicht Bestandteil von ANSI C sind und nicht den Komfort der Standard-E/A-Funktionen bieten, dafür aber schneller ablaufen und dem Benutzer mehr Einflußmöglichkeiten auf seine Ein- und Ausgabe geben.

Dateien, Directories und ihre Attribute (Kapitel 5)

Dieses Kapitel beschreibt die Attribute, die zu jeder Datei und jedem Directory im sogenannten i-node gespeichert sind, und stellt die Funktionen vor, mit denen diese Attribute erfragt oder modifiziert werden können. Außerdem wird die grundlegende Struktur eines Unix-Dateisystems vorgestellt, und es werden Begriffe wie i-nodes und symbolische Links geklärt, bevor auf die konkrete Realisierung von Dateisystemen unter Linux eingegangen wird, wobei hier insbesondere das meist unter Linux verwendete ext2-Dateisystem detaillierter beschrieben wird. Auch stellt dieses Kapitel Funktionen vor, mit denen man Directories anlegen, deren Inhalt lesen oder aber in andere Directories wechseln kann.

Systeminformationen (Kapitel 6 - 7)

Informationen zum System und seinen Benutzern (Kapitel 6)

Dieses Kapitel stellt Funktionen vor, mit denen Informationen aus der Paßwortdatei, aus der Gruppendatei, aus Netzwerkdateien und Informationen zum lokalen System und seinen Benutzern erfragt werden können.

Datums- und Zeitfunktionen (Kapitel 7)

Hier werden Konstanten, Datentypen und Funktionen beschrieben, mit denen das Setzen und Erfragen von Datums- und Zeitwerten möglich ist.

Nicht-lokale Sprünge (Kapitel 8)

Dieses Kapitel beschreibt die beiden ANSI-C-Funktionen *setjmp* und *longjmp*, mit denen ein Springen über Funktionsgrenzen hinweg möglich ist.

Prozesse (Kapitel 9 - 13)

Der Unix-Prozeß (Kapitel 9)

Dieses Kapitel beschäftigt sich mit Unix-Prozessen im allgemeinen. Dazu beschreibt es zunächst die Aktivitäten seitens des Systems, die beim Start und der Beendigung eines Unix-Prozesses ablaufen, bevor es auf die Umgebung (*Environment*) und die Speicherbelegung eines Unix-Prozesses genauer eingeht. Es wird auch auf die Ressourcenlimits eingegangen, die einem Unix-Prozeß auferlegt sind. Zum Abschluß dieses Kapitels wird ein Einblick in die Speicherverwaltung und das Abbilden von Dateien in den Speicher (*Memory Mapping*) unter Linux gegeben. Dieses Kapitel ist nur für Leser von Interesse, die mehr über die interne Speicherverwaltung eines realen Systems wissen möchten.

Die Prozeßsteuerung (Kapitel 10)

Dieses Kapitel stellt die Kennungen eines Prozesses und die Unix-Prozeßhierarchie vor, bevor es auf das Kreieren von neuen Prozessen und dabei insbesondere auf die Beziehungen von Eltern- und Kind-Prozessen näher eingeht. Ebenso beschäftigt sich dieses Kapitel mit dem Warten von Prozessen auf die Beendigung von anderen Prozessen, bevor es mögliche Probleme der Synchronisation von Eltern- und Kindprozessen beschreibt. Des weiteren stellt dieses Kapitel die *exec*-Funktionen vor, mit denen sich ein Prozeß durch ein anderes Programm überlagern kann. Der Rest dieses Kapitels beschäftigt sich mit dem Ändern von Prozeßkennungen und dem Erfragen von Informationen zu einem Prozeß.

Attribute eines Prozesses (Kapitel 11)

Hier werden zunächst die bei einem Login ablaufenden Prozesse beschrieben, wobei zwischen Terminal- und Netzwerk-Logins unterschieden wird. Des weiteren werden in diesem Kapitel die Begriffe Prozeßgruppe, Kontrollterminal und Session (Sitzung) näher erläutert. Auch wird hier ein detaillierter Einblick in die von vielen Shells angebotene Jobkontrolle und die dabei ablaufenden Mechanismen gegeben.

Sperren von Dateien (Kapitel 12)

Dieses Kapitel stellt zunächst blockierende und nicht blockierende E / A-Operationen vor, bevor es sich ausführlich mit dem Sperren von Dateien und den dabei möglichen Problemen beschäftigt. In der Übung wird ein umfangreicheres Projekt vorgestellt, in dem eine einfache *Mehrbenutzer*-Datenbank entwickelt werden soll.

Signale (Kapitel 13)

Signale sind asynchrone Ereignisse, die von der Hard- oder Software erzeugt werden, wenn während einer Programmausführung besondere Ausnahmesituationen auftreten. In diesem Kapitel wird zunächst das Unix-Signalkonzept und die wichtige Funktion *signal* vorgestellt, bevor ein Überblick über die verschiedenen Arten von Signalen gegeben wird. Nachfolgend werden weitere Funktionen vorgestellt, mit denen z.B. das explizite Senden von Signalen, das Einrichten einer Zeitschaltuhr, das Suspendieren oder das anormale Beenden eines Prozesses möglich ist.

Besondere Arten von E/A (Kapitel 14 - 16)

STREAMS in SVR4 (Kapitel 14)

Die in diesem Kapitel beschriebenen STREAMS werden von System V Release 4 (SVR4) vollständig unterstützt und sind dort die allgemeine Schnittstelle zu Kommunikationstreibern.

Fortgeschrittene E/A (Kapitel 15)

Dieses Kapitel beschäftigt sich mit den folgenden Formen der Ein- und Ausgabe: E / A-Multiplexing, asynchrone E / A, gleichzeitiges Lesen und Schreiben aus mehreren nicht zusammenhängenden Puffern und das sogenannte Memory Mapped I/O. Die Kenntnis dieser Formen der Ein- und Ausgabe ist Voraussetzung für das Verständnis der Kapitel 17, 18 und 19, die sich mit der Interprozeßkommunikation beschäftigen.

Dämonprozesse (Kapitel 16)

Dämonprozesse sind Prozesse, die ständig im Hintergrund ablaufen. Sie werden üblicherweise beim Booten des Systems gestartet und laufen dann so lange, bis das System ordnungsgemäß heruntergefahren wird oder aber zusammenbricht. Dämonprozesse sind für ständig anfallende Aufgaben zuständig. Dieses Kapitel gibt zunächst einen Überblick

über typische Unix-Dämonen und deren Besonderheiten und zeigt dann, wie ein eigener Dämonprozeß zu erstellen ist. Da ein Dämonprozeß im Hintergrund läuft und somit auch kein Kontrollterminal besitzt, wird zusätzlich noch gezeigt, wie ein Dämonprozeß dennoch das Auftreten von Fehlern melden kann.

Interprozeßkommunikation (Kapitel 17 - 19)

Pipes und FIFOs (Kapitel 17)

In diesem Kapitel werden Techniken der Kommunikation zwischen unterschiedlichen Prozessen, der sogenannten Interprozeßkommunikation, vorgestellt. Als Kommunikationsmittel werden Pipes und FIFOs (benannte Pipes), die beide zunächst ausführlich beschrieben werden, verwendet. Auch wird in einem Beispiel eine erste Client-Server-Kommunikation vorgestellt, die mittels FIFOs verwirklicht ist.

Message-Queues, Semaphore und Shared Memory (Kapitel 18)

In diesem Kapitel werden drei Methoden der Interprozeßkommunikation vorgestellt:

- ▶ Austausch von Nachrichten (*Message-Queues* = Nachrichten-Warteschlangen)
- ▶ Synchronisation über Semaphore
- ▶ Austausch von Daten über gemeinsame Speicherbereiche (*Shared Memory*).

Bevor in diesem Kapitel auf die Methoden und die zugehörigen Funktionen im einzelnen eingegangen wird, werden zunächst die allen drei Methoden zugrundeliegenden Strukturen und Eigenschaften vorgestellt.

Stream Pipes, Client-Server-Realisierungen und Netzwerkprogrammierung (Kapitel 19)

In diesem Kapitel werden neuere Formen der Interprozeßkommunikation vorgestellt: Stream Pipes und benannte Stream Pipes. Diese beiden Methoden erlauben z.B. den Austausch von Filedeskriptoren zwischen verschiedenen Prozessen oder die Kommunikation von Clients mit einem Server, der als Dämonprozeß abläuft. Hierzu werden jeweils Beispiele gegeben. Auch geht dieses Kapitel auf die Grundlagen der Socket- und Netzwerkprogrammierung mit TCP/IP ein, wozu es u.a. ein Beispielprogramm zur Kommunikation zwischen zwei Rechnern in einem Netzwerk vorstellt.

Terminal-E/A (Kapitel 20)

Der Begriff Terminal-E/A umfaßt alle Funktionen zur Steuerung und Programmierung der seriellen Schnittstellen (seriellen Ports) eines Rechners. An den seriellen Ports können neben Terminals auch Modems, Drucker usw. angeschlossen werden. In diesem Kapitel werden alle von POSIX.1 vorgeschriebenen Terminalfunktionen und einige zusätzliche Funktionen vorgestellt, die von System V Release 4 und BSD-Unix angeboten werden. Zudem stellt dieses Kapitel die Bibliotheken *curses* und *S-Lang* vor, mit denen Semigra-

phikprogrammierung unter Linux/Unix möglich ist. Des weiteren werden hier die Eigenschaften einer Linux-Konsole detaillierter vorgestellt, bevor am Ende dieses Kapitels noch auf die Programmierung von virtuellen Konsolen unter Linux eingegangen wird.

Nützliche Funktionen und Techniken (Kapitel 21)

Hier werden weitere Funktionen vorgestellt, die sehr wertvolle Dienste bei der Systemprogrammierung leisten können. Es werden dabei zunächst Funktionen zur Dateinamenexpansion vorgestellt, bevor dann wichtige Funktionen beschrieben werden, die man zum Arbeiten mit regulären Ausdrücken innerhalb von Programmen benötigt. Am Ende des Kapitels werden dann Funktionen und Techniken vorgestellt, mit denen man Optionen auf der Kommandozeile abarbeiten kann.

Wichtige Entwicklungswerkzeuge (Kapitel 22)

Dieses Kapitel stellt kurz wichtige Entwicklungswerkzeuge vor, die bei der Systemprogrammierung unter Linux/Unix benötigt werden: den GNU-C-Compiler *gcc*, den Linux/Unix-Linker *ld*, den GNU-Debugger *gdb*, das Programm *strace* zum Mitprotokollieren von Systemaufrufen, Werkzeuge zum Auffinden von Speicherüberschreibungen (*Electric Fence*, *checkgcc* und *mpr*), das Programm *ar* zum Erstellen und Verwalten von statischen Bibliotheken, das Erstellen von und Arbeiten mit dynamischen Bibliotheken und sogenannten *shared objects* und das Werkzeug *make* zur automatischen Programmgenerierung.

Anhang

Im Anhang befinden sich neben der eigenen Headerdatei *eihdr.h* und dem Programm *fehler.c*, die beide in fast allen Beispielen dieses Buches benutzt werden, ausgewählte Lösungen zu den Übungen der einzelnen Kapitel.

Literaturhinweise

Als Vorbild zu diesem Buch diente das Buch *Advanced Programming in the UNIX Environment* von W. Richard Stevens. Dieses Standardwerk von Stevens gab viele Hinweise, Anregungen und Tips.

Zu dem vorliegenden Buch existiert ein begleitendes Buch *Linux-Unix Kurzreferenz*, das neben der Beschreibung anderer wichtiger Linux/Unix-Tools auch eine Kurzfassung zu allen typischen Aufrufformen der hier behandelten Funktionen, wichtige Konstanten, Datentypen, Strukturen oder Limitvorgaben enthält. Die Kurzreferenz soll neben den Manpages dem Programmierer nützliche und schnelle Informationen beim täglichen Programmieren seines Linux/Unix-Systems geben.

Unix-Standards und -Implementierungen

Die Vielzahl der verschiedenen Unix-Versionen führte in den achtziger Jahren dazu, daß große Anstrengungen unternommen wurden, Standards zu schaffen, an die sich die einzelnen Unix-Varianten halten sollten.

So wurde mit ANSI C ein Standard für die Programmiersprache C geschaffen, an den sich heute die meisten C-Compiler halten. Für das Betriebssystem Unix selbst ist der IEEE-POSIX-Standard und der X/Open Portability Guide (XPG) von Bedeutung. Dieses Buch beschreibt diese Standards, wobei es allerdings immer wieder auf die heute weit verbreiteten Implementierungen System V Release 4 (SVR4), BSD-Unix (BSD) und Linux eingeht.

Beispiele und Übungen

In diesem Buch befinden sich viele Programmbeispiele und Übungen. Alle Programm Listings, die Lösungen zu den einzelnen Übungen sind, können ebenso wie alle Beispielprogramme von der WWW-Adresse <http://www.addison-wesley.de/service/herold/sysprog.tgz> heruntergeladen werden.

Test der Beispiele unter SOLARIS und Linux

Die meisten der in diesem Buch angegebenen Programmbeispiele wurden sowohl unter SOLARIS wie unter Linux getestet. Da teilweise auch implementierungsspezifische Eigenschaften in den Programmen verwendet werden, konnten jedoch einige wenige Programmbeispiele nicht auf beiden Systemen zum Laufen gebracht werden.

Übungen am Ende jedes Kapitels

Am Ende jedes der nachfolgenden Kapitel befinden sich Übungen, die dem Leser die Möglichkeit geben, das Verständnis der zuvor beschriebenen Funktionen und Konstrukte zu vertiefen. Ausgewählte Lösungen zu diesen Aufgabenstellungen befinden sich in Anhang B.

Hinweis zur Buchreihe: Unix und seine Werkzeuge

Diese Buchreihe soll

- ▶ den Unix-Anfänger systematisch vom Unix-Basiswissen über die leistungstarken Unix- Werkzeuge bis hin zu den fortgeschrittenen Techniken der Systemprogrammierung führen.
- ▶ dem bereits erfahrenen Unix-Anwender – durch ihren modularen Aufbau – eine Vertiefung bzw. Ergänzung seines Unix-Wissens ermöglichen.

Linux-Unix Kurzreferenz Nachschlagewerk zu Kommandos und Systemfunktionen	Teil 4 - Linux-Unix Systemprogrammierung Dateien, Prozesse und Signale Fortgeschrittene E/A, Dämonen und Prozeßkommunikation
	Teil 3 - Linux-Unix Profitools awk, sed, lex, yacc und make
	Teil 2 - Linux-Unix Shells Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh
	Teil 1 - Linux-Unix Grundlagen Kommandos und Konzepte

Die Buchreihe »Unix und seine Werkzeuge«

1 Überblick über die Unix-Systemprogrammierung

*Hat der Fuchs die Nase erst hinein,
so weiß er bald den Leib auch nachzubringen.*

Shakespeare

Jedes Betriebssystem bietet sogenannte Systemroutinen an, die von den Benutzerprogrammen aufgerufen werden können, wenn diese gewisse Dienste vom System benötigen. Typische von einem Betriebssystem angebotene Dienste sind z.B. Öffnen einer Datei, Schreiben auf eine Datei, Bereitstellen von freiem Speicherplatz oder Kommunikation mit anderen Programmen.

In diesem Kapitel wird anhand von kurzen Beschreibungen und Beispielen ein grober Überblick über grundlegende Unix-Eigenschaften und die wichtigsten Gebiete der Systemprogrammierung gegeben, um den Leser bereits zu Beginn mit den wichtigsten Grundbegriffen und Konzepten vertraut zu machen. Bei den detaillierteren Beschreibungen der einzelnen Systemfunktionen in den späteren Kapiteln verfügt der Leser dann über das entsprechende Grundwissen, und es muß nicht ständig eine Erklärung eines erst später genau behandelten Begriffes eingeschoben werden. Auch wird in diesem Kapitel noch ein kurzer Überblick über wichtige Unix-Standardisierungen und Unix-Systeme gegeben.

Zum Abschluß werden erste Einblicke in den Linux-Systemkern gegeben. Dieser Linux-spezifische Abschnitt ist nur für Leser gedacht, die an der Verwirklichung von Betriebssystemkonzepten und -algorithmen interessiert sind oder die selbst Kernroutinen oder systemnahe Funktionen programmieren möchten. Dieser umfangreichere Abschnitt zeigt den grundlegenden Aufbau des Linux-Systemkerns, klärt wichtige Begriffe und stellt wesentliche Kernalgorithmen und -konzepte vor, die für das Verständnis der späteren Linux-spezifischen Kapitel vorausgesetzt werden.

1.1 Anmelden am Unix-System

Um sich am Unix-System anzumelden, muß der Benutzer zunächst seinen Loginnamen und sein Paßwort eingeben. Das System sucht den Loginnamen zunächst in der Datei `/etc/passwd`.

Weitere sehr beliebte Shells, die z.B. bei Linux schon standardgemäß mitgeliefert werden, sind die

- ▶ Bourne-Again-Shell (**/bin/bash**) und die
- ▶ TC-Shell (**/bin/tcsh**).

Diese beiden letzten Shells sind als Freeware erhältlich und sind verbesserte Versionen der Bourne- (**bash**) bzw. der C-Shell (**tcsh**).

Welche Shell das System nach dem Anmelden für den betreffenden Benutzer starten soll, erfährt es aus dem 7. Feld der entsprechenden Benutzerzeile in `/etc/passwd`.

1.2 Dateien und Directories

1.2.1 Dateistruktur

Unter Unix gibt es eigentlich keine Struktur für Dateien². Eine Datei ist für das System nur eine Folge von Bytes (*featureless byte stream*), und ihrem Inhalt wird vom System keine Bedeutung beigemessen. Unix kennt nur sequentielle Dateien und keine sonstigen Datei-Organisationen, welche in anderen Betriebssystemen üblich sind, wie z.B. indexsequentielle Dateien. Die einzigen Ausnahmen sind die Dateiarten, die für die Dateihierarchie und die Identifizierung der Geräte benötigt werden.

1.2.2 Länge von Dateien

Dateien sind stets in Blöcken von Bytes gespeichert. Damit ergeben sich zwei mögliche Größen für Dateien:

- ▶ Länge in Byte
- ▶ Länge in Blöcken (übliche Blockgrößen sind z.B. 512 oder 1024 Byte)

Unix legt keine Begrenzung bezüglich einer maximalen Dateigröße fest. Somit können zumindest theoretisch Dateien beliebig lang sein.

1.2.3 Dateiarten

Es werden mehrere Arten von Dateien unterschieden:

- ▶ *Regular Files* (reguläre Dateien, einfache Dateien, gewöhnliche Dateien)
Eine solche Datei ist eine Sammlung von Zeichen, die unter den entsprechenden Dateinamen gespeichert sind. Diese Dateien können beliebigen Text, Programme oder aber den Binärcode eines Programms enthalten.

2. Das Unix-Dateisystem, die Dateien und Directories sind ausführlich im Band »Linux-Unix-Grundlagen« dieser Reihe beschrieben.

► *Special Files* (spezielle Dateien, Gerätedateien)

Gerätedateien repräsentieren die logische Beschreibung von physikalischen Geräten wie z.B. Bildschirmen, Druckern oder Festplatten. Das Besondere am Unix-System ist, daß es von solchen Gerätedateien in der gleichen Weise liest oder auf sie schreibt, wie es dies bei gewöhnlichen Dateien tut. Jedoch wird hierbei nicht der normale Dateizugriff aktiviert, sondern der entsprechende Gerätetreiber (*device driver*). Es werden zwei Klassen von Geräten unterschieden:

- zeichenorientierte Geräte (Datentransfer erfolgt zeichenweise, wie z.B. Terminal)
- blockorientierte Geräte (Datentransfer erfolgt nicht byteweise, sondern in Blöcken, wie z.B. bei Festplatten)

► *Directory* (Dateiverzeichnis)

Ein Directory enthält wieder Dateien. Es kann neben einfachen Dateien auch andere Dateiarten (wie z.B. Gerätedateien) oder aber auch wiederum Directories (sogenannte *Subdirectories* bzw. *Unterverzeichnisse*) enthalten. Zu jedem in einem Directory enthaltenen Dateinamen existiert Information über dessen Attribute. Diese Dateiattribute informieren z.B. über die Art, Größe, Eigentümer, Zugriffsrechte einer Datei. Die in einem späteren Kapitel vorgestellten Systemfunktionen *stat* und *fstat* liefern dem Aufrufer eine Struktur, in der er alle Attribute zu der entsprechenden Datei findet. Beim Anlegen eines neuen Directories werden immer die folgenden beiden Dateinamen automatisch dort angelegt:

```
.   Name für dieses Directory
..  Name für das sogenannte Parent-Directory (siehe unten).
```

► *FIFO* (first in first out, Named Pipes)

FIFOS – auch Named Pipes genannt – dienen der Kommunikation und Synchronisation verschiedener Prozesse. Prinzipiell können sie wie einfache Dateien benutzt werden, mit dem wesentlichen Unterschied, daß Daten nur einmal gelesen werden können. Zudem können sie nur in der Reihenfolge gelesen werden, wie sie geschrieben wurden.

► *Sockets*

Sockets dienen zur Kommunikation von Prozessen in einem Netzwerk, können aber auch zur Kommunikation von Prozessen auf einem lokalen Rechner benutzt werden.

► *Symbolic Links* (symbolische Verweise)

Symbolische Links sind Dateien, die lediglich auf andere Dateien zeigen.

1.2.4 Zugriffsrechte

Jeder Datei (reguläre Datei, Directory ...) ist unter Unix ein aus 9 Bits bestehendes Zugriffsrechte-Muster zugeordnet. Jeweils 3 Bit geben dabei die Zugriffsrechte (*read*, *write*, *execute*) der entsprechenden Benutzerklasse (*owner*, *group*, *others*) an. Diese Zugriffsrechte von Dateien kann man sich mit der Angabe der Option **-l** beim **ls**-Kommando anzeigen lassen, wie z.B.:


```
$ ls -l kopier
-rwxr-x--x  1 hh      grafik      867 May  17  1995 kopier
$
```

An dieser Ausgabe läßt sich erkennen, daß der Eigentümer der Datei (hier *hh*) die Datei *kopier* lesen, beschreiben oder ausführen darf, während alle Mitglieder der *grafik*-Gruppe die Datei *kopier* nur lesen oder ausführen dürfen. Alle anderen Benutzer (*others*) dürfen die *kopier*-Datei nur ausführen, aber nicht lesen oder beschreiben.

1.2.5 Dateinamen

In einem Dateinamen sind außer dem Slash (/) und dem NUL-Zeichen alle Zeichen erlaubt. Trotzdem ist es empfehlenswert, folgende Zeichen nicht in Dateinamen zu verwenden, um Konflikte mit den Metazeichen der Shells zu vermeiden:

? @ # \$ % ^ & * () ` [] \ | ' " < > Leerzeichen Tabulatorzeichen

Auch sollte als erstes Zeichen eines Dateinamens nicht +, - oder . benutzt werden. Während auf älteren Unix-Systemen die Länge von Dateinamen auf 14 Zeichen begrenzt war, wurde in neueren Unix-Systemen diese Grenze erheblich hochgesetzt (z.B. auf 255 Zeichen).

1.2.6 Dateisystem

Das Unix-Dateisystem (*file system*) ist hierarchisch in Form eines nach unten wachsenden Baumes aufgebaut. Die Wurzel dieses Baums ist das sogenannte *Root-Directory*, das einen Slash (/) als Namen hat. Bei jedem Arbeiten unter Unix befindet man sich an einem bestimmten Ort im Dateibaum. Jeder Benutzer wird nach dem Anmelden an einer ganz bestimmten Stelle innerhalb des Dateibaums positioniert. Von dieser Ausgangsposition kann er sich nun durch den Dateibaum »hangeln«, solange er nicht durch Zugriffsrechte vom Betreten bestimmter Äste abgehalten wird. Nachfolgend sind die gebräuchlichsten Begriffe aus dem Dateisystem-Vokabular aufgezählt.

1.2.7 Root-Directory

Das *Root-Directory* (Root-Verzeichnis) ist die Wurzel des Dateisystems und enthält kein übergeordnetes Directory mehr. Im Root-Directory entspricht der Name »..« (Punkt, Punkt) dem Namen ».« (Punkt), so daß das Parent-Directory zum Root-Directory wieder das Root-Directory selbst ist.

1.2.8 Working-Directory

Das *Working-Directory* (Arbeitsverzeichnis) ist der momentane Aufenthaltsort im Dateibaum. Mit dem Kommando **pwd** kann der aktuelle Aufenthaltsort (Working-Directory) am Bildschirm ausgegeben, und mit dem Kommando **cd** gewechselt werden in ein neues Working-Directory.

1.2.9 Home-Directory

Jeder eingetragene Systembenutzer hat einen eindeutigen und von ihm allein verwaltbaren Platz im Dateisystem: sein *Home-Directory* (Home-Verzeichnis). Der Pfadname des Home-Directorys steht in der betreffenden Benutzerzeile in der Datei `/etc/passwd`. Wird das Kommando `cd` ohne Angabe eines Directory-Namens abgegeben, so wird immer zum Home-Directory gewechselt.

1.2.10 Parent-Directory

Das *Parent-Directory* (Elternverzeichnis) ist das Directory, das in der Dateihierarchie unmittelbar über einem Directory angeordnet ist. Zum Beispiel ist `/user1` das Parent-Directory zum Directory `/user1/fritz`. Eine Ausnahme gibt es dabei: Das Parent-Directory zum Root-Directory ist das Root-Directory selbst.

1.2.11 Pfadnamen

Jede Datei und jedes Directory im Dateisystem ist durch einen eindeutigen Pfadnamen gekennzeichnet. Man unterscheidet zwei Arten von Pfadnamen:

- ▶ **absoluter Pfadname**
Hierbei wird, beginnend mit dem Root-Directory, ein Pfad durch den Dateibaum zum entsprechenden Directory oder zur Datei angegeben. Ein absoluter Pfadname ist dadurch gekennzeichnet, daß er mit einem Slash (`/`) beginnt. Der erste *Slash* ist die Wurzel des Dateibaums, alle weiteren stellen die Trennzeichen bei jeden »Abstieg um eine Ebene im Dateibaum« dar.
- ▶ **relativer Pfadname**
Die Angabe eines solchen Pfadnamens beginnt nicht in der Wurzel des Dateibaums, sondern im Working-Directory. Anders als beim absoluten Pfadnamen ist das erste Zeichen hier kein Slash: Hier erfolgt also die Orientierung relativ zum momentanen Aufenthaltsort (*Working-Directory*).
Ein relativer Pfadname beginnt immer mit einer der folgenden Angaben:
 - ▶ einem Directory- oder Dateinamen
 - ▶ ».« (Punkt): Kurzform für das *Working-directory*
 - ▶ »..« (Punkt,Punkt): Kurzform für das *Parent-Directory*

Beispiel

Absolute und relative Pfadnamen

- ▶ Angenommen, das Working-Directory sei `/user1/herbert`, dann würde der relative Pfadname `briefe/finanzamt` dem absoluten Pfadnamen `/user1/herbert/briefe/finanzamt` entsprechen.

- Angenommen, das Working-Directory sei /user1/herbert, dann würde der relative Pfadname ./briefe/finanzamt dem absoluten Pfadnamen /user1/herbert/briefe/finanzamt entsprechen.
- Angenommen, das Working-Directory sei /user1/herbert, dann würde der relative Pfadname ../../bin/sort dem absoluten Pfadnamen /bin/sort entsprechen.

Beispiel*Ausgeben der Dateien eines Directorys*

```
#include <sys/types.h>
#include <dirent.h>
#include <string.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    char        dir_name[MAX_ZEICHEN]; /* MAX_ZEICHEN ist in eighdr.h def. */
    DIR         *dir;
    struct dirent *dir_info;

    if (argc > 2)
        fehler_meld(FATAL, "Es ist nur ein Argument (Directory-Name) erlaubt");
    else if (argc==2)
        strcpy(dir_name, argv[1]);
    else
        strcpy(dir_name, "."); /* working directory */

    if ( (dir = opendir(dir_name)) == NULL)
        fehler_meld(FATAL_SYS, "kann %s nicht eroeffnen", dir_name);

    while ( (dir_info = readdir(dir)) != NULL)
        printf("%s\n", dir_info->d_name);

    closedir(dir);
    exit(0);
}
```

Programm 1.1 (meinls.c): Alle Dateien eines Directorys ausgeben

Wenn wir dieses Programm 1.1 (meinls.c) wie folgt kompilieren und linken:

```
cc -o meinls meinls.c fehler.c      [unter Linux eventuell: gcc -o ...]
```

dann liefert es beim Aufruf z.B. folgende Ausgaben:

```
$ meinls /usr/include
.
..
alloca.h
ctype.h
```

```

curses.h
dirent.h
errno.h
.....
.....
fcntl.h
ftw.h
getopt.h
stdio.h
signal.h
stdlib.h
string.h
$ meinls /dev/console
kann /dev/console nicht eroeffnen: Not a directory
$ meinls /usr /tmp
Es ist nur ein Argument (Directory-Name) erlaubt
$ meinls /ect
kann /ect nicht eroeffnen: No such file or directory
$ meinls          [Ausgeben der Dateien des Working-Directory]
.
..
copy1.c
copy2.c
meinls.c
numer1.c
procid.c
zaehlen.c
eighdr.h
fehler.c
meinls
$

```

In diesem Programm 1.1 (meinls.c) wird mit

```
#include "eighdr.h"
```

unsere eigene Headerdatei eighdr.h zum Bestandteil dieses Programms gemacht. Diese Headerdatei wird in nahezu jedes Programm der späteren Kapitel eingefügt, also »*included*«. Die Headerdatei eighdr.h »*included*« zum einen einige für die Systemprogrammierung häufig benötigte Headerdateien, zum anderen definiert sie zahlreiche Konstanten und Prototypen von eigenen Funktionen (wie Fehlerrouninen), die in den Beispielen dieses und späterer Kapitel benutzt werden. Das Listing zu der Headerdatei eighdr.h befindet sich im Anhang.

Falls beim Programm 1.1 (meinls.c) auf der Kommandozeile ein Directory-Name angegeben wurde, so befindet sich dieser in `argv[1]`. Wurde auf der Kommandozeile keinerlei Argument angegeben, so nimmt das Programm als Default (Voreinstellung) das Working-Directory (.) an. Für den Fall, daß dieses Programm mit mehr als einem Argument aufgerufen wird, ruft es die Fehlerroutine `fehler_meld` auf.

Bei `fehler_meld` handelt es sich um eine eigene Fehlerroutine aus dem Modul `fehler.c`, dessen Listing sich ebenfalls im Anhang befindet. Das erste Argument legt dabei fest, wie

der entsprechende Fehler zu behandeln ist. Es sind die folgenden in `error.h` definierten Konstanten als erstes Argument erlaubt:

```
WARNUNG
WARNUNG_SYS
FATAL
FATAL_SYS
DUMP
```

Es wurde dabei die folgende Regelung bei der Vergabe der Konstantennamen gewählt:

- ▶ Die Endung `SYS` bedeutet, daß zusätzlich zur eigenen Meldung noch die zum entsprechenden Fehler gehörige System-Fehlermeldung auszugeben ist.
- ▶ Nur bei den `WARNUNG`-Konstanten bewirkt die Fehlerroutine nicht die Beendigung des gesamten Programms.
- ▶ Bei Angabe der `FATAL`- und `DUMP`-Konstanten bewirkt die Fehlerroutine einen Programmabbruch. Nur bei der `DUMP`-Konstante wird mittels **`abort`** das Programm beendet und ein *core dump* (Speicherabzug) erzeugt. Bei `FATAL` und `FATAL_SYS` wird das Programm mit `exit(1)` beendet.

Die weiteren Argumente zu `fehler_meld` entsprechen denen eines **`printf`**-Aufrufs.

Der Aufruf von **`opendir`** bewirkt das Öffnen des betreffenden Directorys und liefert einen `DIR`-Zeiger zurück. Unter Verwendung dieses `DIR`-Zeigers liest nun **`readdir`** in einer Schleife jeden Eintrag im entsprechenden Directory, wobei es entweder einen Zeiger auf die `dirent`-Struktur oder einen `NULL`-Zeiger (am Ende) liefert. Die `dirent`-Struktur enthält für jeden Directory-Eintrag in der Komponente `d_name` dessen Name. **`closedir`** schließt dann wieder das geöffnete Directory.

Um das Programm zu beenden, wird die Funktion **`exit`** aufgerufen. Der Wert 0 zeigt an, daß das Programm fehlerfrei ausgeführt wurde. Liefert dagegen ein Programm als Exit-Status einen Wert zwischen 1 und 255, so deutet dies üblicherweise auf das Auftreten eines Fehlers bei der Ausführung dieses Programms hin.

Es ist anzumerken, daß das Programm `meinls` die Namen in einem Directory nicht (wie **`ls`**) alphabetisch auflistet, sondern entsprechend der Reihenfolge, in der sie in der Directory-Datei eingetragen sind.

1.3 Ein- und Ausgabe

1.3.1 Filedeskriptoren

Wenn eine Datei geöffnet wird, dann wird dieser Datei vom Betriebssystemkern eine nichtnegative ganze Zahl (0, 1, 2, 3 ...), der sogenannte *Filedeskriptor* zugewiesen. Unter Angabe dieses Filedeskriptors kann das Benutzerprogramm unter Verwendung der entsprechenden Systemroutinen in die geöffnete Datei schreiben oder aus ihr lesen.

1.3.2 Standardeingabe, Standardausgabe, Standardfehlerausgabe

Wird ein Programm gestartet, so öffnet die Shell für dieses Programm immer automatisch drei Filedeskriptoren:

Standardeingabe (standard input)
Standardausgabe (standard output)
Standardfehlerausgabe (standard error)

Die Filedeskriptor-Nummern für diese drei »Dateien« sind üblicherweise 0, 1 und 2. Anstelle dieser Nummern sollte man allerdings in Systemen, die den POSIX-Standard erfüllen, folgende Konstanten aus der Headerdatei `<unistd.h>` benutzen:

STDIN_FILENO (üblicherweise 0)
STDOUT_FILENO (üblicherweise 1)
STDERR_FILENO (üblicherweise 2)

Normalerweise sind alle diese drei Filedeskriptoren auf das Terminal eingestellt. So erwartet z.B. der einfache Aufruf

```
cat
```

Eingaben von der Tastatur (bis Strg-D für EOF), welche er wieder am Bildschirm ausgibt. Lenkt man dagegen die Standardausgabe um, wie z.B.

```
cat >x.txt
```

dann werden alle von der Tastatur eingegebenen Zeilen nicht auf den Bildschirm, sondern in die Datei `x.txt` geschrieben.

1.3.3 Standard-E/A-Funktionen (aus `<stdio.h>`)

Die Standard-E/A-Funktionen sind in der Headerdatei `<stdio.h>` definiert. Im Gegensatz zu den nachfolgend vorgestellten elementaren E/A-Funktionen arbeiten diese Funktionen mit eigenen Puffern, so daß sich der Aufrufer darum (Definition eines eigenen Puffers mit selbstgewählter Puffergröße) nicht eigens kümmern muß. Auch bieten die Standard-E/A-Funktionen dem Benutzer mehr Komfort an, wie z.B. Formatierung der Ausgabe bei ***printf*** oder zeilenweises Einlesen bei ***fgets***.

Beispiel

Kopieren von Standardeingabe auf Standardausgabe

```
#include "eighdr.h"

int
main(void)
{
    int    zeich;
```

```

while ( (zeich=getc(stdin)) != EOF)
    if (putc(zeich, stdout) == EOF)
        fehler_meld(FATAL_SYS, "Fehler bei putc");

    if (ferror(stdin))
        fehler_meld(FATAL_SYS, "Fehler bei getc");

    exit(0);
}

```

Programm 1.2 (copy1.c): Standardeingabe auf Standardausgabe kopieren

Die Funktion **getc** liest immer ein Zeichen von der Standardeingabe (`stdin`), das dann mit **putc** auf die Standardausgabe (`stdout`) geschrieben wird. Wenn das letzte Byte gelesen wird oder ein Fehler beim Lesen auftritt, liefert **getc** als Rückgabewert die Konstante `EOF`. Um festzustellen, ob ein Fehler beim Lesen aufgetreten ist, wird die Funktion **ferror** aufgerufen.

Anders als die elementaren E/A-Funktionen wird beim Öffnen einer Datei mit den Standard-E/A-Funktionen nicht ein Filedeskriptor, sondern ein `FILE`-Zeiger zurückgeliefert. Der Datentyp `FILE` ist eine Struktur, die alle Informationen enthält, die von den entsprechenden Standard-E/A-Routinen beim Umgang mit der betreffenden Datei benötigt werden. Wird ein Programm gestartet, so werden für dieses Programm immer automatisch drei `FILE`-Zeiger geöffnet:

```

stdin (Standardeingabe)
stdout (Standardausgabe)
stderr (Standardfehlerausgabe)

```

Wenn wir dieses Programm 1.2 (copy1.c) nun kompilieren und linken

```
cc -o copy1 copy1.c fehler.c
```

und dann aufrufen, so liest es immer aus der Standardeingabe (bis `EOF` bzw. `Strg-D`) und schreibt die gelesenen Zeichen wieder auf die Standardausgabe. Es ist allerdings auch möglich, die Standardeingabe und/oder Standardausgabe umzulenken, wie z.B.:

```

copy1 <liste           [gibt Datei liste am Bildschirm aus]
copy1 >a.c             [schreibt alle über Tastatur eingegeb. Daten in Datei a.c]
copy1 <datei1 >datei2  [kopiert datei1 nach datei2]

```

Um weitere Dateien zu öffnen, steht die Funktion **fopen** zur Verfügung, der als erstes Argument der Name der zu öffnenden Datei zu übergeben ist. Als zweites Argument ist bei dieser Funktion anzugeben, was man nach dem Öffnen mit dieser Datei zu tun wünscht, wie z.B. `»r«` für Lesen oder `»w«` für Schreiben.

Beispiel*Ausgeben einer Datei mit Zeilennummerierung*

```
#include "eighdr.h"

#define MAX_ZEILLAENG 200

int
main(int argc, char *argv[])
{
    FILE    *fz;
    char    zeile[MAX_ZEILLAENG];
    int     zeilnr=0;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s dateiname", argv[0]);

    if ( (fz=fopen(argv[1], "r")) == NULL)
        fehler_meld(FATAL_SYS, "Kann %s nicht eroeffnen", argv[1]);

    while (fgets(zeile, MAX_ZEILLAENG, fz) != NULL)
        fprintf(stdout, "%5d %s", ++zeilnr, zeile);

    if (ferror(fz))
        fehler_meld(FATAL_SYS, "Fehler beim Lesen aus %s", argv[1]);

    fclose(fz);

    exit(0);
}
```

Programm 1.3 (*numer1.c*): Datei mit Zeilennummerierung auf Standardausgabe ausgeben

Dieses Programm 1.3 (*numer1.c*) liest mit ***fgets*** Zeile für Zeile ein, wobei vorausgesetzt wird, daß eine Zeile maximal 200 Zeichen lang ist. Jede gelesene Zeile wird mit Zeilennummer mittels ***fprintf*** auf die Standardausgabe (*stdout*) ausgegeben.

1.3.4 Elementare E/A-Funktionen (aus <unistd.h>)

Elementare E/A-Funktionen sind in der Headerdatei <unistd.h> deklariert. Wichtige elementare E/A-Funktionen sind z.B.:

<i>open</i>	(Öffnen einer Datei; liefert entsprechenden Filedeskriptor)
<i>read</i>	(Lesen aus einer geöffneten Datei)
<i>write</i>	(Schreiben in eine geöffnete Datei)
<i>lseek</i>	(Positionieren des Schreib-/Lesezeigers in geöffneter Datei)
<i>close</i>	(Schließen einer geöffneten Datei)

Alle diese elementaren E/A-Funktionen benutzen den von ***open*** gelieferten Filedeskriptor.

Beispiel*Kopieren von Standardeingabe auf Standardausgabe*

```
#include "eighdr.h"

#define PUFF_GROESSE 1024

int
main(void)
{
    int    n;
    char   puffer[PUFF_GROESSE];

    while ( (n=read(STDIN_FILENO, puffer, PUFF_GROESSE)) > 0)
        if (write(STDOUT_FILENO, puffer, n) != n)
            fehler_meld(FATAL_SYS, "Fehler bei write");

    if (n<0)
        fehler_meld(FATAL_SYS, "Fehler bei read");

    exit(0);
}
```

Programm 1.4 (copy2.c): Standardeingabe auf Standardausgabe kopieren

Die Funktion **read** versucht bei jedem Aufruf aus einer Datei, deren Filedeskriptor als erstes Argument anzugeben ist (hier Standardeingabe), maximal so viele Bytes zu lesen, wie mit dem dritten Argument festgelegt wird (hier PUFF_GROESSE). Die gelesenen Zeichen werden dann im Speicher an der Adresse abgelegt, die als zweites Argument (hier puffer) angegeben ist. Wie viele Bytes wirklich gelesen werden konnten, liefert **read** als Rückgabewert. Dieser Rückgabewert wird hier in *n* abgelegt.

Diese Anzahl *n* von Bytes (drittes Argument) wird mit **write** wieder aus dem puffer (zweites Argument) ausgelesen und dann in die Datei geschrieben, deren Filedeskriptor als erstes Argument anzugeben ist (hier Standardausgabe).

Falls beim Lesen das Dateiende erreicht wurde, liefert **read** den Wert 0. Ist beim Lesen ein Fehler aufgetreten, liefert **read** den Wert -1, was im übrigen für die meisten Systemfunktionen gilt.

1.4 Prozesse unter Unix

1.4.1 Der Begriff Prozeß

Von der Vielzahl von möglichen Prozeßdefinitionen scheint die Definition

Prozeß = ein Programm während der Ausführung

die einfachste und verständlichste zu sein. In manchen Systemen wird anstelle des Begriffes *Prozeß* auch der Begriff *Task* verwendet.

Wird ein Programm (Benutzerprogramm oder Unix-Kommando) aufgerufen, so wird der zugehörige Programmcode, der sich in einer Datei befindet, in den Hauptspeicher geladen und dann gestartet. Das ablaufende Programm wird als Prozeß bezeichnet.

Wird das gleiche Programm (wie z.B. das Kommando **ls**) von unterschiedlichen Benutzern gestartet, so handelt es sich dabei um zwei verschiedene Prozesse, obwohl beide das gleiche Programm ausführen.

1.4.2 Prozeß-ID

Jedem Prozeß wird vom Betriebssystem eine eindeutige Kennung in Form einer nicht-negativen ganzen Zahl zugewiesen: die sogenannte *Prozeß-ID* (*process identification*). Meist verwendet man die Abkürzung *PID*.

Will ein Prozeß seine PID erfahren, so muß er nur die Systemfunktion **getpid** aufrufen, welche die PID des aufrufenden Prozesses als Rückgabewert liefert.

Beispiel

Erfragen der eigenen Prozeß-ID

```
#include "eighdr.h"

int
main(void)
{
    printf("Meine PID ist ---%d---\n", getpid());
    exit(0);
}
```

Programm 1.5 (procid.c): Ausgeben der eigenen PID

Wenn wir das Programm 1.5 (procid.c) kompilieren und linken mit

```
cc -o procid procid.c fehler.c
```

und dann aufrufen, so können wir erkennen, daß es bei jedem Aufruf eine andere PID liefert, da immer ein neuer Prozeß gestartet wird.

```
$ procid
Meine PID ist ---783---
$ procid
Meine PID ist ---812---
$
```

1.4.3 Systemfunktionen zur Prozeßsteuerung

Die zur Steuerung von Prozessen angebotenen Systemfunktionen bieten die unterschiedlichsten Dienste an, wie z.B.

Kreieren von neuen Prozessen (**fork**)
 Prozesse mit anderen Programmcode überlagern (**exec**, ...)
 Kommunikation zwischen verschiedenen Prozessen (**pipe**, **popen**, ...)
 Warten auf die Beendigung von Prozessen (**waitpid**, ...)

In späteren Kapiteln werden alle zur Prozeßsteuerung angebotenen Systemfunktionen ausführlich besprochen. Ein einfaches Beispiel soll jedoch bereits hier einen kleinen Einblick in die Prozeßsteuerung geben.

Beispiel

Gleichzeitiges Zählen durch Eltern- und Kindprozeß

Im folgenden Programm 1.6 (zaehlen.c) zählen parallel ein Eltern- und ein Kindprozeß um die Wette³. Der Elternprozeß meldet dabei seinen Zwischenstand in 200000er und der Kindprozeß in 100000er Schritten:

```
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    long int z=1;
    pid_t pid;

    printf("Eltern- und Kindprozess zaehlen um die Wette:\n\n");

    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei fork");
    else if (pid == 0) {
        printf("%75s\n", "Kind: Ich beginne zu zaehlen");
        while (z<=1000000) {
            if ((z%100000) == 0)
                printf("%70s %d\n", "Kind: Ich bin schon bei", z);
            z++;
        }
        printf("%65s %d\n", "z(Kind) = ", z);
    } else if (pid > 0) {
        printf("Vater: Ich beginne zu zaehlen\n");
        while (z<=1200000) {
            if ((z%200000) == 0)
                printf("Vater: %d und rede nicht soviel!\n", z);
        }
    }
}
```

3. Statt Elternprozeß spricht man oft auch von Vaterprozeß.

```

        z++;
    }
    printf("z(Vater) = %d\n", z);

    printf(" ----> z = %d\n", z); /* wird von Vater und Kind ausgefuehrt */
}
/* Eltern- */
/* prozesses*/
/*- - - - */

```

Programm 1.6 (zaehlen.c): Eltern- und Kindprozeß zählen um die Wette

Hier wird der Systemaufruf **fork** benutzt, um einen neuen Prozeß zu kreieren. Der neue Prozeß ist eine exakte Kopie des aufrufenden Prozesses, was heißt, daß sowohl das gesamte Code- wie das Datensegment dieses Prozesses (Programm 1.6) dupliziert wird, wobei der Befehlszähler (*program counter*) im Eltern- wie im Kindprozeß auf dieselbe Programmstelle zeigt. Mit Elternprozeß bezeichnet man den aufrufenden und mit Kindprozeß den neu kreierten Prozeß. Die Funktion **fork** gibt für den Elternprozeß die nichtnegative PID des neuen Kindprozesses und für den Kindprozeß den Wert 0 zurück. Da **fork** einen neuen Prozeß kreiert, sagt man auch, daß es zwar nur einmal (vom Elternprozeß) aufgerufen wird, aber zweimal einen Rückgabewert liefert, nämlich einen für den Eltern- und einen für den Kindprozeß.

Es ist somit der Rückgabewert beim Aufruf

```
pid=fork()
```

entscheidend. Es gilt dabei folgendes:

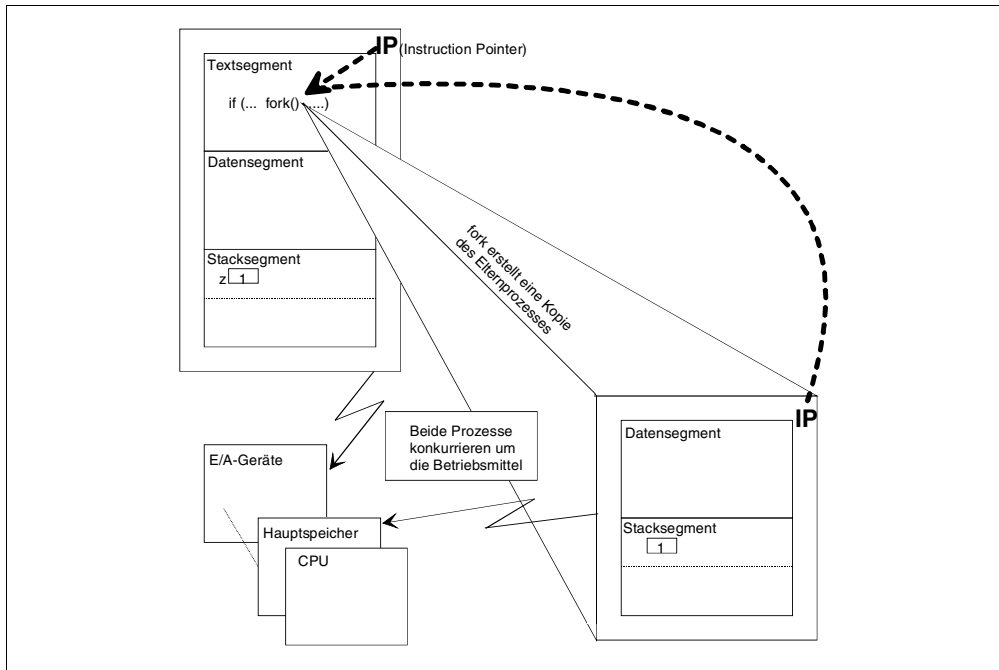
```

pid=0 (im Kindprozeß)
pid>0 (im Elternprozeß; pid ist dann die PID des Kindprozesses)
pid=-1 (fork war nicht erfolgreich)

```

Da ein Kindprozeß in der Regel einen anderen Programmteil ausführen soll als der Elternprozeß, kann über diesen Rückgabewert gesteuert werden, welcher Programmteil vom Kind- und welcher vom Elternprozeß auszuführen ist.

Im obigen Programm 1.6 (zaehlen.c) wird mit **fork** ein Kindprozeß gestartet, der eine Kopie des Code-, Daten- und Stacksegmentes des Elternprozesses enthält; d.h., daß er z.B. den momentanen Wert der Variablen *z* erbt. Auch übernimmt dieser Kindprozeß den Wert des Befehlszählers vom Elternprozeß. Somit fährt er zwar an der gleichen Programmstelle (nach **fork**-Aufruf) fort, an der er aufgerufen wurde, aber – und das ist wichtig – mit seinem eigenem Befehlszähler (*instruction pointer*) für das Codesegment und mit seinem eigenen Daten- und Stacksegment (siehe Abbildung 1.1).

Abbildung 1.1: Kreieren eines Kindprozesses mit `fork`

Beide Prozesse konkurrieren nun um die Betriebsmittel (CPU, Hauptspeicher usw.). Um die Ausgabe des Kindprozesses von der des Elternprozesses unterscheiden zu können, erfolgen in `zaehlen.c` die Ausgaben des Elternprozesses am linken und die des Kindprozesses am rechten Bildschirmrand. Nachdem man das Programm 1.6 (`zaehlen.c`) kompiliert und gelinkt hat

```
cc -o zaehlen zaehlen.c fehler.c
```

kann ein Aufruf von `zaehlen` z.B. die folgende Ausgabe liefern.

```
$ zaehlen
```

```
Eltern- und Kindprozess zaehlen um die Wette:
```

```
Vater: Ich beginne zu zaehlen
```

```
Kind: Ich beginne zu zaehlen
```

```
Kind: Ich bin schon bei 100000
```

```
Vater: 200000 und rede nicht soviel!
```

```
Kind: Ich bin schon bei 200000
```

```
Kind: Ich bin schon bei 300000
```

```
Vater: 400000 und rede nicht soviel!
```

```
Kind: Ich bin schon bei 400000
```

```
Kind: Ich bin schon bei 500000
```

```
Vater: 600000 und rede nicht soviel!
```

```
Kind: Ich bin schon bei 600000
```

```
Kind: Ich bin schon bei 700000
```

```

Vater: 800000 und rede nicht soviel!
Kind: Ich bin schon bei 800000
Kind: Ich bin schon bei 900000

Vater: 1000000 und rede nicht soviel!
Kind: Ich bin schon bei 1000000
      z(Kind) = 1000001

----> z = 1000001
Vater: 1200000 und rede nicht soviel!
z(Vater) = 1200001
----> z = 1200001
$

```

Bei dieser Ausgabe ist zu erkennen, daß beiden Prozessen abwechselnd die Betriebsmittel (CPU, E/A-Geräte usw.), um die sie konkurrieren, zugeteilt werden.

Auch ist an der Ausgabe zu erkennen, daß der Kindprozeß bei seiner Erzeugung die Variable *z* (und ihren Wert) erbt. Da diese lokale Variable allerdings in sein eigenes Stacksegment kopiert wird, ist *z* ab diesem Zeitpunkt eine eigene Variable des Kindprozesses, d.h., daß ein Verändern von *z* durch den Kindprozeß keinerlei Einfluß auf das *z* des Elternprozesses hat.

Ein weiterer interessanter Aspekt, der an dieser Ausgabe zu erkennen ist, ist die Tatsache, daß beide Prozesse nach Beendigung ihres entsprechenden Programmteils (in der **if**-Anweisung) mit dem Programm nach der **if**-Anweisung fortfahren. In diesem Programmteil wird nur noch der jeweilige Wert von *z* ausgegeben:

```

----> z = 1000001      (Kindprozeß)
----> z = 1200001      (Elternprozeß)

```

1.5 Ausgabe von System-Fehlermeldungen

Wenn bei der Ausführung einer Systemfunktion ein Fehler auftritt, so liefern viele Systemfunktionen `-1` als Rückgabewert und setzen zusätzlich die Variable `errno` auf einen von 0 verschiedenen Wert. Diese Variable `errno` ist in `<errno.h>` mit

```
extern int errno;
```

definiert. Zusätzlich zu dieser Definition der Variablen `errno` definiert `<errno.h>` Konstanten für jeden Wert, der `errno` von den Systemfunktionen zugewiesen werden kann. Jede dieser Konstanten beginnt mit dem Buchstaben **E** (für **Error**). In den Unix-Manpages sind unter `intro(2)` alle in `<errno.h>` definierten Konstanten zusammengefaßt.

Bezüglich der Verwendung der Variablen `errno` ist folgendes zu beachten.

- ANSI C garantiert nur für den Programmstart, daß die Variable `errno` auf 0 gesetzt wird. Die Systemfunktionen setzen niemals diese Variable zurück auf 0, und es gibt in `<errno.h>` keine Fehlerkonstante mit dem Wert 0.

- Deshalb ist es gängige Praxis, daß man `errno` vor dem Aufruf einer Systemfunktion explizit auf 0 setzt und nach dem Aufruf dieser Funktion den Wert von `errno` überprüft, um festzustellen, ob während der Ausführung dieser Funktion ein Fehler aufgetreten ist.

Um die Fehlermeldung zu erhalten, die zu einem in `errno` stehenden Fehlercode gehört, schreibt ANSI C die beiden Funktionen **`perror`** und **`strerror`** vor.

1.5.1 `perror` – Ausgabe der zu `errno` gehörenden Fehlermeldung

Die Funktion **`perror`** gibt auf `stderr` die zum momentan in `errno` stehenden Fehlercode gehörende Fehlermeldung aus.

```
#include <stdio.h>

void perror(const char *meldung);
```

Diese `errno`-Fehlermeldung entspricht genau dem Rückgabewert der nachfolgend beschriebenen Funktion **`strerror`**, falls diese mit dem gleichen `errno`-Wert als Argument aufgerufen wird.

1.5.2 `strerror` – Erfragen der zu einer Fehlernummer gehörigen Meldung

Die Funktion **`strerror`** (in `<string.h>` definiert) liefert die zu einer Fehlernummer (üblicherweise der `errno`-Wert) gehörende Meldung als Rückgabewert.

```
#include <string.h>

char *strerror(int fehler_nr);
```

gibt zurück: Zeiger auf die entsprechende Fehlermeldung

Die beiden folgenden Anweisungen liefern das gleiche Ergebnis:

```
perror("testausgabe")
fprintf(stderr, "testausgabe: %s\n", strerror(errno));
```

Beispiel

Demonstrationsbeispiel zu `perror` und `strerror`

```
#include <string.h>
#include <errno.h> /* da globale Variable errno verwendet wird */
#include "eighdr.h"

int
main(void)
{
```

```
int    fehler_nr=0;

for (fehler_nr=0 ; fehler_nr<5 ; fehler_nr++) {
    fprintf(stderr, "%3d -> strerror: %s\n", fehler_nr, strerror(fehler_nr));
    errno = fehler_nr;
    perror("      perror  ");
}
exit(0);
}
```

Programm 1.7 (**errodemo.c**): Demonstrationsbeispiel zu **perror** und **strerror**

Nachdem man dieses Programm 1.7 (errodemo.c) kompiliert und gelinkt hat

```
cc -o errodemo errodemo.c
```

kann sich z.B. folgender Ablauf ergeben:

```
$ errodemo
0 -> strerror: Unknown error
    perror  : Unknown error
1 -> strerror: Operation not permitted
    perror  : Operation not permitted
2 -> strerror: No such file or directory
    perror  : No such file or directory
3 -> strerror: No such process
    perror  : No such process
4 -> strerror: Interrupted system call
    perror  : Interrupted system call
$
```

In den späteren Beispielprogrammen dieses Buches wird jedoch weder **perror** noch **strerror** direkt aufgerufen. Statt dessen wird dort die eigene Fehlerroutine **fehler_meld** aus dem Programm **fehler.c**, dessen Listing sich im Anhang befindet, aufgerufen.

1.6 Benutzerkennungen

1.6.1 User-ID

Zu jedem Benutzer existiert in der Paßwortdatei eine eindeutige Kennung in Form einer Nummer. Diese Nummer, die dem Benutzer vom Systemadministrator beim Einrichten seines Loginnamens zugeteilt wird, bezeichnet man als *User-ID*. 0 ist die User-ID des besonders privilegierten Superusers, dessen Loginname meist *root* ist. Ein Superuser hat alle Rechte im System, während die Rechte von normalen Benutzern meist sehr eingeschränkt sind.

1.6.2 Group-ID

Jeder Benutzer ist einer Gruppe und jeder Gruppe ist eine eindeutige Kennung in Form einer Nummer zugeordnet. Diese Nummer, die dem Benutzer vom Systemadministrator ebenfalls beim Einrichten seines Loginnamens zugeteilt wird, bezeichnet man als *Group-ID*. Die Group-ID eines Benutzers befindet sich auch im entsprechenden Paßwortdatei-Eintrag eines Benutzers. Da mehrere Benutzer zu einer Gruppe gehören können, was der Normalfall ist, können natürlich auch mehrere Benutzer die gleiche Group-ID besitzen. Die Zuordnung von Gruppennamen zu Group-IDs befindet sich in der Datei `/etc/group`.

Beispiel

Ausgeben der User-ID und Group-ID eines Benutzers

Das folgende Programm 1.8 (`usergrup.c`) gibt unter Verwendung der beiden Funktionen ***getuid*** und ***getgid*** die User- und Group-ID des aufrufenden Benutzers aus.

```
#include "eighdr.h"

int
main(void)
{
    printf("uid = %d\n"
           "gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Programm 1.8 (usergrup.c): Ausgeben der User-ID und Group-ID

Nachdem man das Programm 1.8 (`usergrup.c`) kompiliert und gelinkt hat

```
cc -o usergrup usergrup.c
```

kann sich z.B. folgender Ablauf ergeben:

```
$ usergrup
uid = 2021
gid = 5
$
```

1.7 Signale

Signale sind asynchrone Ereignisse, die erzeugt werden, wenn während einer Programmausführung besondere Ereignisse eintreten. So wird z.B. bei einer Division durch 0 dem entsprechenden Prozeß das Signal `SIGFPE` (FPE=*floating point error*) geschickt. Ein Prozeß hat drei verschiedene Möglichkeiten, auf das Eintreffen eines Signals zu reagieren:

1. Ignorieren des Signals

Dies ist nicht für Signale empfehlenswert, die einen Hardwarefehler (wie Division durch 0 oder Zugriff auf unerlaubte Speicherbereiche) anzeigen, da der weitere Ablauf eines solchen Prozesses zu nicht vorhersagbaren Ergebnissen führen kann.

2. Voreingestellte Reaktion

Für jedes mögliche Signal ist eine bestimmte Reaktion festgelegt. So ist z.B. die voreingestellte Reaktion auf das Signal `SIGFPE` die Beendigung des entsprechenden Prozesses. Trifft ein Benutzer keine besonderen Vorrichtungen für das Eintreffen eines Signals, so ist die voreingestellte Reaktion (meist Beendigung des Prozesses) für dieses Signals eingerichtet.

3. Ausführen einer eigenen Funktion

Für jedes Signal kann ein Prozeß auch seine eigene Reaktion festlegen. Dazu muß er mit der Funktion ***signal*** sogenannte Signalhandler (Funktionen) einrichten. Bei Eintreffen der entsprechenden Signale werden dann automatisch diese eingerichteten Signalhandler ausgeführt. Mit solchen Funktionen kann somit der Prozeß seine eigene Reaktion auf das Eintreffen eines bestimmten Signals festlegen.

Beispiel

Einrichten eines eigenen Signalhandlers

Das folgende Programm 1.9 (`sighandl.c`) demonstriert, wie man sich mit der Funktion ***signal*** einen eigenen Signalhandler einrichten kann.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include "eighdr.h"

static int  intr_aufgetreten = 0;

/*----- sig_intr -----*/
void sig_intr(int signr)
{
    printf("Du willst das Programm abbrechen?\n");
    printf("Noch nicht ganz, du must noch ein bisschen warten\n");
    sleep(5); /* 5 Sekunden warten, bevor Programm fortgesetzt wird */
    intr_aufgetreten = 1;
}

/*----- main -----*/
int
main(void)
{
    int      a = 0;
```

```

printf("Programmstart\n");
if (signal(SIGINT, sig_intr) == SIG_ERR)
    fehler_meld(FATAL_SYS, "kann Signalhandler sig_intr nicht einrichten");

while (intr_aufgetreten == 0) /* Endlosschleife: Warten auf das */
    ;                        /* Eintreffen des interrupt-Signals */
printf("Schleife verlassen\n");
printf("%d\n", 2/a);
printf("----- Fertig -----\\n");

exit(0);
}

```

Programm 1.9 (sig handl.c): Einrichten eines eigenen Signalhandlers

Nachdem man das Programm 1.9 (sig handl.c) kompiliert und gelinkt hat

```
cc -o sig handl sig handl.c fehler.c
```

kann sich z.B. folgender Ablauf ergeben:

```

$ sig handl
Programmstart
Strg-C      [Drücken der Interrupt-Taste]
Du willst das Programm abbrechen?
Noch nicht ganz, du must noch ein bisschen warten
Schleife verlassen
Floating exception
$

```

In dem Programm 1.9 (sig handl.c) wird ein Signalhandler sig_intr zum Signal SIGINT eingerichtet. Das Signal SIGINT wird geschickt, wenn der Benutzer die *Interrupt*-Taste (meist *Strg-C* oder *DELETE*) drückt. Das Programm 1.9 (sig handl.c) begibt sich nach dem Einrichten des Signalhandlers in eine Endlosschleife. Drückt der Aufrufer dann irgendwann die *Interrupt*-Taste, so wird die Funktion sig_intr angesprungen, die zunächst etwas Text ausgibt, bevor sie mit sleep(5) die Ausführung des Programms für fünf Sekunden anhält. Danach setzt sie die globale Variable intr_aufgetreten auf 1, was dazu führt, daß nach Beendigung der Funktion sig_intr die Schleife beendet und das durch Ausgabe eines entsprechenden Textes dem Benutzer mitteilt. Die darauffolgende Division durch 0 (Signal SIGFPE) bewirkt allerdings, daß die voreingestellte Reaktion auf das Signal SIGFPE aktiviert wird, da für dieses Signal kein eigener Signalhandler eingerichtet wurde. Die voreingestellte Reaktion auf das Signal SIGFPE ist die Beendigung des Programms, so daß die letzte **printf**-Anweisung (printf("----- Fertig -----\\n")) nicht mehr ausgeführt wird, sondern das Programm vorzeitig mit der Meldung *Floating exception* vom System beendet wird.

1.8 Zeiten in Unix

1.8.1 Kalenderzeit und CPU-Zeit

Unix unterscheidet zwischen zwei Zeiten:

1. *Kalenderzeit*

Diese Zeit wird im Systemkern als die Anzahl der Sekunden dargestellt, die seit *00:00:00 Uhr des 1. Januars 1970 (UTC⁴)* vergangen sind. Diese Kalenderzeit, die immer im Datentyp `time_t` dargestellt wird, benutzt z.B. das Kommando **date** zur Ausgabe der aktuellen Datums- und Zeitwerte. Ebenso wird diese Zeit für die Einträge der Zeitmarken bei Dateien (z.B. letzte Zugriffs- oder Modifikationszeit) verwendet.

2. *CPU-Zeit*

Diese Zeit gibt an, wie lange ein bestimmter Prozeß die CPU benutzte. Die CPU-Zeit wird anders als die Kalenderzeit nicht in Sekunden, sondern in sogenannten *clock ticks* ("Uhr-Ticks") pro Sekunde gemessen. Ein typischer Wert für *clock ticks* pro Sekunde ist z.B. 50 oder 100. Seit ANSI C ist dieser Wert in der Konstante `CLOCKS_PER_SEC` in der Headerdatei `<time.h>` definiert, während früher die Konstante `CLK_TCK`; diesen Wert definierte. Die CPU-Zeit wird immer im Datentyp `clock_t` dargestellt.

1.8.2 Prozeßzeiten

Für einen Prozeß unterhält der Kern drei Zeitwerte:

- ▶ abgelaufene Uhrzeit seit Start
- ▶ Benutzer-CPU-Zeit
- ▶ System-CPU-Zeit

Die *abgelaufene Uhrzeit* ist die Zeit, die seit dem Start eines Prozesses vergangen ist. Je mehr Prozesse gleichzeitig im System ablaufen, um so länger dauert die Ausführung eines Prozesses und um so größer wird dieser Wert sein.

Die *Benutzer-CPU-Zeit* ist die Zeit, die ein Prozeß die CPU zur Ausführung von Benutzeranweisungen beansprucht. Die *System-CPU-Zeit* ist die Zeit, die ein Prozeß die CPU zur Ausführung von Kernroutinen beansprucht. Die Summe aus Benutzer-CPU- und System-CPU-Zeit bezeichnet man üblicherweise als *CPU-Zeit*.

Um die von einem Programm verbrauchte Uhrzeit, Benutzer-CPU- und System-CPU-Zeit zu erfahren, muß man der entsprechenden Kommandozeile nur das Kommando **time** voranstellen, wie z.B.:

4. Abkürzung für Universal Time Coordinated, die der GMT (Greenwich Mean Time) entspricht.

```
$ time find / -name "*.h" -print
.....
.....
1.54user 9.42system 1:06.34elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+0minor)pagefaults 0swaps
$
```

Das Ausgabeformat des **time**-Kommandos ist von der benutzten Shell abhängig.

1.9 Unterschiede zwischen Systemaufrufen und Bibliotheksfunktionen

Obwohl in den späteren Kapiteln immer nur von *Funktionen* gesprochen wird, soll hier darauf hingewiesen werden, daß es zwei verschiedene Arten von Funktionen gibt: *Systemaufrufe* und *Bibliotheksfunktionen*. Nachfolgend werden die Unterschiede zwischen diesen beiden Arten von Funktionen vorgestellt.

1.9.1 Systemaufrufe sind Systemkern-Schnittstellen

Die Systemaufrufe sind die Schnittstellen zum Kern. Sie sind in *Section 2* des *Unix Programmer's Manual* beschrieben, wo sie in Form von C-Funktionsdeklarationen angegeben sind. Alle diese Systemfunktionen befinden sich ebenso wie die nachfolgend beschriebenen Bibliotheksfunktionen in der C-Standardbibliothek, so daß aus Benutzersicht kein Unterschied zwischen diesen beiden Funktionsarten besteht. Beim Aufruf von Systemfunktionen wird aber anders als bei den Bibliotheksfunktionen Systemkern-Code ausgeführt.

1.9.2 Bibliotheksfunktionen sind keine Schnittstellen zum Kern

Die Bibliotheksfunktionen, die in *Section 3* des *Unix Programmer's Manual* beschrieben sind, stellen anders als die Systemfunktionen keine Schnittstellen zum Systemkern dar, wenn auch einige Bibliotheksfunktionen eine oder mehrere Systemfunktionen ihrerseits aufrufen. So ruft z.B. die Bibliotheksfunktion **printf** zur Ausgabe die Systemfunktion **write** auf. Andere Bibliotheksfunktionen dagegen, wie z.B. **strlen** (ermittelt Länge eines Strings) oder **sqrt** (berechnet Quadratwurzel), kommen ohne jeglichen Aufruf einer Systemfunktion aus.

Während Bibliotheksfunktionen leicht durch neue ersetzt werden können, können Systemfunktionen nicht so einfach ausgetauscht werden. Im letzteren Fall wäre eine Änderung des Kerns notwendig.

Abbildung 1.2 verdeutlicht noch einmal, daß ein Benutzerprogramm sowohl Systemfunktionen als auch Bibliotheksfunktionen aufrufen kann. Zudem zeigt Abbildung 1.2, daß einige Bibliotheksfunktionen ihrerseits Systemfunktionen aufrufen.

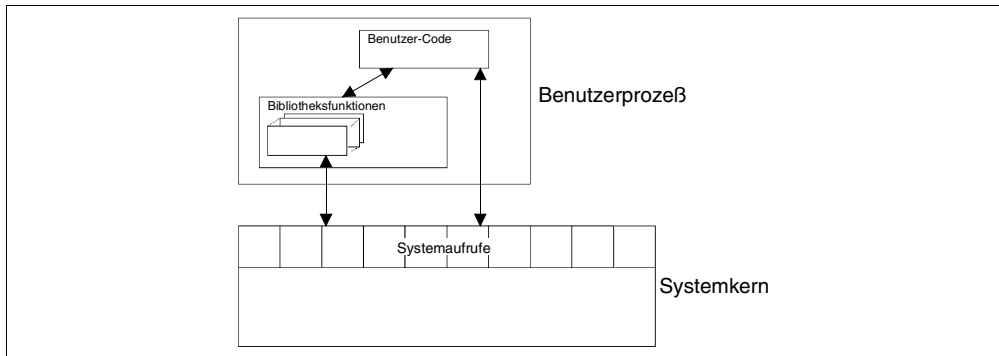


Abbildung 1.2: Beziehungen zwischen Anwenderprogrammen, Bibliotheksfunktionen und Systemaufrufen

Beispiel

Systemaufruf *time* und Bibliotheksfunktionen aus `<time.h>`

Die Headerdatei `<time.h>` enthält Funktionen, die sich für das Erfragen von Datums- und Zeitwerten eignen. Von diesen Funktionen ist die Funktion ***time*** ein Systemaufruf, während alle anderen Bibliotheksfunktionen sind.

Die Systemfunktion ***time*** erfragt vom Kern die aktuelle Zeit und liefert diese als die Anzahl von Sekunden, die seit 00:00:00 Uhr am 1. Januar 1970 verstrichen sind. Die Interpretation der zurückgelieferten Sekundenzahl, wie z.B. die Konvertierung in ein verständliches Datumsformat (wie z.B. Mon Jun 05 03:57:12 1995), ist Sache des Benutzerprozesses. Aber auch in `<time.h>` sind Bibliotheksfunktionen vorhanden, die eine solche Konvertierung ermöglichen, wie z.B. ***ctime*** (siehe auch Kapitel 7).

Während also ***time*** ein Systemaufruf ist, der die Zeit direkt vom Kern erfragt, sind alle anderen Zeitfunktionen aus `<time.h>` Bibliotheksfunktionen, die keinerlei Dienste vom Kern anfordern, sondern lediglich mit dem von ***time*** zurückgelieferten Wert arbeiten (siehe Abbildung 1.3).

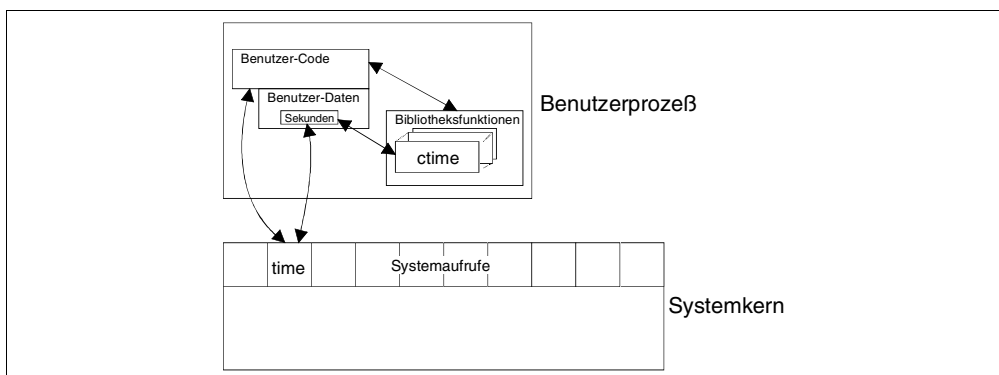


Abbildung 1.3: Systemaufruf ***time*** und Bibliotheksfunktionen zur Interpretation des Zeitwertes

1.10 Unix-Standardisierungen und -Implementierungen

Während der achtziger Jahre wurden große Anstrengungen unternommen, Unix zu standardisieren. Im Laufe der Jahre hatte sich nämlich eine Vielzahl von unterschiedlichen Unix-Versionen herausgebildet. Um dieser »Wucherung« von Unix-Versionen mit ihren vielen kleinen Unterschieden Einhalt zu gebieten, wurde der Ruf nach einem Unix-Standard immer lauter.

Hier werden die Standardisierungen und Implementierungen vorgestellt, auf die dieses Buch ausgerichtet ist.

1.10.1 Unix-Standardisierungen

POSIX

Die Standardisierungsbestrebungen der amerikanischen Unix-Benutzergemeinde wurden 1986 vom amerikanischen *Institute for Electrical and Electronic Engineers* (IEEE) unter dem Namen POSIX (*Portable Operating System Interface*) aufgegriffen.

POSIX ist nicht nur ein Standard, sondern eine ganze Familie von Standards. Der Standard IEEE POSIX 1003.1 für die Betriebssystem-Schnittstellen wurde bereits 1988 verabschiedet.

Weitere Standards, wie IEEE POSIX 1003.2 (Shells und Utilities), wurden im wesentlichen 1991/1992 abgeschlossen. An zahlreichen weiteren Standards wird momentan noch gearbeitet.

Für das vorliegende Buch ist insbesondere der Standard 1003.1 (System-Schnittstellen) von Wichtigkeit. Dieser Standard definiert die Dienste, die jedes Betriebssystem anbieten muß, wenn es vorgibt, die POSIX-1003.1-Forderungen zu erfüllen. Die meisten heutigen Unix-Systeme genügen diesem POSIX.1-Standard. Der POSIX-Standard basiert zwar auf Unix, ist jedoch nicht nur auf Unix-Systeme begrenzt. Es existieren auch andere Betriebssysteme, die den POSIX-Standard erfüllen.

Ende 1990 wurde eine Revision des POSIX-1003.1-Standards durchgeführt. Den dabei verabschiedeten Standard bezeichnet man allgemein als POSIX.1. 1992 wurden einige Erweiterungen dem 1990 verabschiedeten Standard hinzugefügt, woraus die Version 1003.1a von POSIX.1 resultierte.

X/Open XPG

1984 gründeten 13 führende Computerhersteller, darunter AT&T, BULL, DEC, Ericson, Hewlett Packard, ICL, Nixdorf, Olivetti, Phillips, Siemens und Unisys, die sogenannte *X/Open-Gruppe* mit dem Ziel, Industriestandards für offene Systeme zu schaffen.

Ein wesentliches Ergebnis der Arbeit der X/Open-Gruppe war der sogenannte *X/Open Portability Guide (XPG)*, dessen erste Ausgabe 1985 (XPG1) erschien. Die meisten heutigen Unix-Implementierungen unterstützen die 3. Ausgabe des XPG (XPG3), die 1988 herauskam, obwohl zwischenzeitlich eine neue Ausgabe (XPG4) existiert, die Mitte 1992 verabschiedet wurde. XPG4 wurde notwendig, da XPG3 nur auf einen Entwurf des ANSI-C-Standards basierte, der erst 1989 mit einigen Änderungen verabschiedet wurde.

ANSI C

Ende 1989 wurde der ANSI⁵-Standard X3.159-1989 für die Programmiersprache C verabschiedet. Dieser Standard wurde im Jahre 1990 auch als internationaler Standard ISO / IEC 9899:1990 für die Sprache C anerkannt. Der ANSI-C-Standard wird in Kapitel 2 ausführlicher beschrieben.

1.10.2 Unix-Implementierungen

Während Standardisierungen wie IEEE POSIX, X/Open XPG4, ANSI C von unabhängigen Organisationen durchgeführt werden, werden die eigentlichen Unix-Implementierungen, die diesen gesetzten Standards mehr oder weniger genügen, von speziellen Computerfirmen vorgenommen. In diesem Buch wird auf drei wichtige Unix-Implementierungen eingegangen, die sich heute auf dem Markt befinden.

System V Release 4 (SVR4)

System V Release 4 (SVR4) ist ein Produkt von USL (Unix System Laboratories) der Firma AT&T. SVR4 erfüllt die beiden Standards POSIX 1003.1 und X/Open XPG3.

AT&T veröffentlichte 1984 ebenfalls die *System V Interface Definition (SVID)*. 1986 brachte AT&T eine überarbeitete System V Interface Definition, Issue 2 (*SVID-2*) heraus, die im wesentlichen XPG3 prägte. SVID-2 lag System V Release 3 (SVR3) zugrunde.

Die 3. Ausgabe des SVID (*SVID-3*), die die Kompatibilität mit POSIX herstellte, war die Grundlage für die Implementierung von SVR4.

SVR4 enthält auch eine sogenannte *Berkeley Compatibility Library*, die Funktionen und Kommandos enthält, die sich wie unter 4.3BSD-Unix verhalten, was jedoch nicht immer dem POSIX-Standard entspricht. Deshalb sollte man bei neuen Applikationen von diesen Funktionen und Kommandos keinen Gebrauch machen.

BSD-Unix

BSD (*Berkeley Software Distribution*) ist eine Unix-Linie, die an der UCB (*University of California at Berkeley*) entstanden ist und dort auch weiterentwickelt wird. Die Version 4.2BSD wurde 1983 und die Version 4.3BSD wurde 1986 freigegeben. Beide Versionen liefen auf einem VAX-Minicomputer. Inzwischen ist die Version 4.4BSD erschienen.

5. American National Standards Institute

Linux

Linux ist ein frei verfügbares Unix-System für PCs, das sich heute sehr großer Beliebtheit erfreut. Es umfaßt Teile der Funktionalität von SVR4, des POSIX-Standards und der BSD-Linie. Wesentliche Teile des Unix-Kerns wurden von Linus Torvalds, einem finnischen Informatik-Studenten, entwickelt. Er stellte die Programmquellen des Kerns unter die *GNU Public License*. Somit hat jeder das Recht, sie zu kopieren.

Die erste Version des Linux-Kerns war Ende 1991 im Internet verfügbar. Es bildete sich schnell eine Gruppe von Linux-Entwicklern, die die Entwicklung dieses Systems vorantrieben. Die Linux-Software wird unter offenen und verteilten Bedingungen entwickelt, was bedeutet, daß jeder, der dazu in der Lage ist, sich beteiligen kann. Das Kommunikationsmedium der Linux-Entwickler ist das Internet.

An entsprechenden Stellen wird in diesem Buch die Umsetzung von wichtigen Betriebssystemkonzepten und -algorithmen am System Linux gezeigt. Dieses System wurde nicht nur aufgrund seiner großen Beliebtheit hierfür ausgewählt, sondern eben auch, weil Linux alle seine Quellprogramme der Öffentlichkeit zur Verfügung stellt.

1.10.3 Headerdateien

Die Tabelle 1.1 gibt einen Überblick darüber, welche Headerdateien von den einzelnen Standards gefordert bzw. in den einzelnen Implementierungen angeboten werden. Bei der Kurzbeschreibung ist dabei in Klammern das Kapitel angegeben, in dem diese Headerdateien näher beschrieben werden.

Headerdatei	Standards			Implementierung		Kurzbeschreibung
	ANSI C	POSIX.1	XPG	SVR4	BSD	
<assert.h>	x			x	x	Testmöglichkeiten in einem Programm (2.4)
<cpio.h>		x		x		cpio-Archivwerte
<ctype.h>	x			x	x	Umwandlung/Klassifikation von Zeichen (2.4)
<dirent.h>		x	x	x	x	Directory-Einträge (5.9)
<errno.h>	x			x	x	Fehlerkonstanten (1.5)
<fcntl.h>		x	x	x	x	Elementare E / A-Operationen (4.2)
<float.h>	x			x	x	Limits/Eigenheiten für Gleitpunkt-Typen (2.4)
<ftw.h>			x	x		Rekursives Durchlaufen eines Dir.-Baums (5.9)
<grp.h>		x	x	x	x	Gruppendatei (6.2)

Tabelle 1.1: Headerdateien in den einzelnen Standards und Implementierungen

Headerdatei	Standards			Implementierung		Kurzbeschreibung
	ANSI C	POSIX.1	XPG	SVR4	BSD	
<langinfo.h>			x	x		Sprachenspezifische Konstanten
<limits.h>	x			x	x	Implementierungskonstanten (1.11 und 2.4)
<locale.h>	x			x	x	Länderspezifische Gegebenheiten (2.4)
<math.h>	x			x	x	Mathemat. Konstanten / Funktionen (2.4)
<nl_types.h>			x	x		message-Kataloge
<pwd.h>		x	x	x	x	Paßwortdatei (6.1)
<regex.h>			x	x	x	Reguläre Ausdrücke
<search.h>			x	x		Suchtabellen
<setjmp.h>	x			x	x	Nichtlokale Sprünge (8)
<signal.h>	x			x	x	Signale (13)
<stdarg.h>	x			x	x	Variabel lange Argumentlisten (2.3)
<stddef.h>	x			x	x	Standarddefinitionen (2.4)
<stdio.h>	x			x	x	Standard-E / A-Bibliothek (3)
<stdlib.h>	x			x	x	Allgemein nützliche Funktionen (2.4)
<string.h>	x			x	x	String-Bearbeitung (2.4)
<tar.h>		x		x		tar-Archivwerte
<termios.h>		x	x	x	x	Terminal-E / A (20)
<time.h>	x			x	x	Datum und Zeit (7)
<ulimit.h>			x	x		Benutzerlimits
<unistd.h>		x	x	x	x	Symbolische Konstanten
<utime.h>		x	x	x	x	Dateizeiten (5.8)
<sys / ipc.h>			x	x	x	Interprozeßkommunikation (18.1)
<sys / msg.h>			x	x		message queues (18.2)
<sys / sem.h>			x	x		Semaphore (18.3)
<sys / shm.h>			x	x	x	shared memory (18.4)
<sys / stat.h>		x	x	x	x	Dateistatus (5)
<sys / times.h>		x	x	x	x	Prozeßzeiten (10.8)
<sys / types.h>		x	x	x	x	Primitive Systemdatentypen (1.12)

Tabelle 1.1: Headerdateien in den einzelnen Standards und Implementierungen

Headerdatei	Standards			Implementierung		Kurzbeschreibung
	ANSI C	POSIX.1	XPG	SVR4	BSD	
<sys/utsname.h>		x	x	x		Systemname (6.4)
<sys/wait.h>		x	x	x	x	Prozeßsteuerung (10.3)

Tabelle 1.1: Headerdateien in den einzelnen Standards und Implementierungen

1.11 Limits

Die einzelnen Implementierungen legen über Konstantendefinitionen in den Headerdateien ihre eigene Limits fest, wie z.B. die Anzahl von Dateien, die ein Prozeß zu einem Zeitpunkt maximal geöffnet haben darf. Man unterscheidet zwei Arten von Limits: Limits zur Kompilierungszeit und Laufzeitlimits.

1.11.1 Optionen und Limits zur Kompilierungszeit (compile-time options and limits)

Optionen und Limits zur Kompilierungszeit werden während der Kompilierung eines Programmes festgelegt. Dies sind üblicherweise Konstanten, die in Headerdateien definiert sind, wie z.B. die Konstante `LONG_MAX` (aus `<limits.h>`), die den maximalen Wert für den Datentyp **long** festlegt, oder die Konstante `_POSIX_JOB_CONTROL` (aus `<unistd.h>`), die angibt, ob das jeweilige System Jobkontrolle unterstützt oder nicht. Bei letzterer Konstante handelt es sich um eine Option, da diese Konstante entweder definiert ist oder nicht. Ob diese Konstante definiert ist, kann mit der Präprozessor-Direktive `#ifdef _POSIX_JOB_CONTROL` erfragt werden.

1.11.2 Laufzeitlimits (run-time limits)

Dies sind Limits, die zum Kompilierungszeitpunkt noch nicht bekannt sind, sondern erst während der Laufzeit eines Programms erfragt werden können. So ist z.B. die maximale Anzahl von Zeichen für einen Dateinamen vom Filesystem abhängig, in dem man sich gerade befindet. Im System V waren früher nur maximal 14 Zeichen, während in BSD-Unix schon seit längerem bis zu 255 Zeichen für einen Dateinamen möglich sind. Da sich in einem System unterschiedliche Filesysteme befinden können, ist die maximal erlaubte Länge eines Dateinamens davon abhängig, in welchem Filesystem sich ein Prozeß gerade befindet. Um die aktuell erlaubte maximale Dateinamenlänge zu erfragen, muß deshalb der Prozeß zur Laufzeit eine Funktion aufrufen, die ihm das entsprechende Limit liefert.

1.11.3 ANSI C-Limits

Alle von ANSI C definierten Limits sind Kompilierungszeit-Limits (*compile-time limits*), die in Headerdateien (wie z.B. `<limits.h>`, `<float.h>` oder `<stdio.h>`) als Konstanten definiert sind. Alle diese ANSI-C-Limits werden in Kapitel 2.4 bei der Vorstellung der von ANSI C vorgeschriebenen Bibliotheksfunktionen vorgestellt.

1.11.4 POSIX-Limits

POSIX.1 kennt 33 verschiedene Limits und Konstanten. Diese sind in folgende Kategorien aufgeteilt:

Invariante Minimalwerte

Tabelle 1.2 zeigt 13 Konstanten, die invariante Minimalwerte festlegen.

Name	maximaler Wert für	Wert
_POSIX_ARG_MAX	Länge der Argumente bei den exec-Aufrufen	4096
_POSIX_CHILD_MAX	Anzahl von Kindprozessen für eine reale User-ID	6
_POSIX_LINK_MAX	Anzahl von Links auf eine Datei	8
_POSIX_MAX_CANON	Anzahl von Bytes in der kanonischen Eingabe-Warteschlange eines Terminals	255
_POSIX_MAX_INPUT	Anzahl von verfügbarer Speicherplatz in der Eingabe-Warteschlange eines Terminals	255
_POSIX_NAME_MAX	Anzahl von Bytes für einen Dateinamen	14
_POSIX_NGROUPS_MAX	Anzahl von Zusatz-Group-IDs pro Prozeß	0
_POSIX_OPEN_MAX	Anzahl von offenen Datei pro Prozeß	16
_POSIX_PATH_MAX	Anzahl von Bytes für einen Dateinamen	255
_POSIX_PIPE_BUF	Anzahl von Bytes, die in einer atomaren Operation in eine Pipe geschrieben werden können	512
_POSIX_SSIZE_MAX	Datentyp ssize_t	32767
_POSIX_STREAM_MAX	Anzahl von Standard-E/A-Dateien (Streams), die ein Prozeß gleichzeitig geöffnet haben darf	8
_POSIX_TZNAME_MAX	Anzahl der Bytes für den Zeitzonen-Namen	3

Tabelle 1.2: Invariante POSIX.1-Minimalwerte aus <limits.h>

Diese 13 invarianten Konstanten haben auf allen Systemen, die sich an den POSIX.1-Standard halten, den gleichen Wert. Die von diesen Konstanten festgelegten Werte sind Minimalwerte, die auf jeder POSIX.1-Implementierung eingehalten werden müssen (die Endung MAX ist etwas irreführend). Ein Programm, das sich POSIX.1 konform nennt, darf diese Minimalwerte nicht überschreiten.

Leider sind einige dieser Minimalwerte für die Praxis zu klein, wie z.B. _POSIX_OPEN_MAX=16 oder _POSIX_PATH_MAX=255. Deswegen ließ der POSIX.1-Standard ein Schlupfloch zu, indem er der jeweiligen Implementierung erlaubt, eigene höhere Limits zu definieren. Diese höheren Limits müssen in Namen definiert sein, die identisch mit

den Namen in Tabelle 1.2 sind, aber ohne das Präfix `_POSIX_` (siehe auch weiter unten). Leider ist nicht garantiert, daß jede Implementierung diese 13 implementierungsspezifischen Konstanten (ohne Präfix `_POSIX_`), die – wenn vorhanden – in der Headerdatei `<limits.h>` definiert sind, anbietet. Der Grund hierfür ist, daß manche Werte von dem am jeweiligen Rechner verfügbaren Speicherplatz abhängig sind. Wenn gewisse Konstantennamen nicht in der Headerdatei `<limits.h>` definiert sind, können sie nicht als obere Grenze bei Array-Definitionen verwendet werden.

Das heißt jedoch nicht, daß diese Limits nicht vorhanden sind. Sie sind lediglich nicht zur Kompilierungszeit, wohl aber zur Laufzeit des Programms verfügbar. Deswegen schrieb POSIX.1 die drei Funktionen ***sysconf***, ***pathconf*** und ***fpathconf*** vor, mit denen sich der aktuelle Implementierungswert zur Laufzeit des Programms erfragen läßt (siehe auch weiter unten).

SSIZE_MAX – Maximaler Wert für den Datentyp `ssize_t`

Diese Konstante legt den maximalen nicht veränderbaren Wert für den Datentyp `ssize_t` fest.

NGROUPS_MAX – Maximale Anzahl von Zusatz-Group-IDs pro Prozeß

Diese Laufzeitkonstante legt die maximale Anzahl von *Zusatz-Group-IDs* fest, die pro Prozeß existieren können. Dieser Wert kann niemals erhöht werden.

Invariante Laufzeitkonstanten

Hierzu zählen die folgenden Konstanten:

`ARG_MAX`

maximale Länge der Argumente bei den *exec*-Funktionen

`CHILD_MAX`

maximale Anzahl von Kindprozessen für eine reale User-ID

`OPEN_MAX`

maximale Anzahl der offenen Dateien pro Prozeß

`STREAM_MAX`

maximale Anzahl von Standard-E/A-Dateien (*Streams*), die ein Prozeß gleichzeitig geöffnet haben darf

`TZNAME_MAX`

maximale Anzahl von Bytes für den Zeitzonennamen

Werte für Pfadnamen und Puffer

LINK_MAX

maximale Anzahl von Links für eine Datei

MAX_CANON

maximale Anzahl von Bytes in der kanonischen Eingabewarteschlange eines Terminals

MAX_INPUT

maximal verfügbarer Speicherplatz in der Eingabewarteschlange eines Terminals

NAME_MAX

maximale Anzahl von Bytes für einen Dateinamen

PATH_MAX

maximale Anzahl von Bytes für einen Pfadnamen

PIPE_BUF

maximale Anzahl von Bytes, die atomar in eine Pipe geschrieben werden können

Optionen und POSIX.1-Version

_POSIX_JOB_CONTROL

wenn definiert, so unterstützt das System Jobkontrolle

_POSIX_SAVED_IDS

wenn definiert, so unterstützt das System saved set-user-IDs und saved set-group-IDs

_POSIX_VERSION

zeigt die POSIX.1-Version an

Konstanten, die zur Ausführungszeit ausgewertet werden

_POSIX_CHOWN_RESTRICTED

wenn definiert, so ist **chown** nur bestimmten Benutzern erlaubt

_POSIX_NO_TRUNC

wenn definiert, so führt die Verwendung von Pfadnamen, die länger als NAME_MAX sind, zu einem Fehler

_POSIX_VDISABLE

wenn definiert, so können spezielle Terminalzeichen durch dieses Zeichen ausgeschaltet werden

Anzahl der Ticks pro Sekunde

CLK_TCK

Diese Konstante enthält die Anzahl der Uhrticks pro Sekunde der auf dem jeweiligen System vorhandenen Uhr

Von den hier angegebenen Konstanten sind 15 immer definiert. Abhängig von bestimmten Voraussetzungen sind die restlichen auf dem jeweiligen System definiert oder auch nicht. Darauf wird nun bei der Vorstellung der Funktionen **sysconf**, **pathconf** und **fpathconf** genauer eingegangen.

1.11.5 sysconf, pathconf und fpathconf – Erfragen von Laufzeitlimits

Um Laufzeitlimits zu erfragen, stehen die drei Funktionen **sysconf**, **pathconf** und **fpathconf** zur Verfügung.

#include <unistd.h>

long **sysconf**(int name);
long **pathconf**(const char *pfadname, int name);
long **fpathconf**(in fd, int name);

alle drei geben zurück: entsprechender Wert (bei Erfolg); -1 bei Fehler

Die Funktionen **pathconf** und **fpathconf** unterscheiden sich nur darin, daß bei **pathconf** ein Pfadname und bei **fpathconf** ein Filedeskriptor einer bereits geöffneten Datei anzugeben ist. Die möglichen Angaben für das bei allen drei vorhandene Argument *name* sind in Tabelle 1.3 angegeben. Die für **sysconf** anzugebenden Konstanten beginnen mit **_SC_**, und die für **pathconf** oder **fpathconf** anzugebenden Konstanten beginnen mit **_PC_**.

Limitname	Beschreibung	name-Argument
ARG_MAX	maximale Länge der Argumente bei den exec-Funktionen	_SC_ARG_MAX
CHILD_MAX	maximale Anzahl von Kindprozessen für eine reale User-ID	_SC_CHILD_MAX
Uhrticks / Sek.	Anzahl der Uhrticks pro Sekunde	_SC_CLK_TCK
NGROUPS_MAX	maximale Anzahl von Zusatz-Group-IDs pro Prozeß	_SC_NGROUPS_MAX
OPEN_MAX	Anzahl von offenen Dateien pro Prozeß	_SC_OPEN_MAX
PASS_MAX	maximale Anzahl von signifikanten Zeichen in einem Paßwort (nicht POSIX.1)	_SC_PASS_MAX
STREAM_MAX	maximale Anzahl von Standard-E / A-Dateien (Streams), die ein Prozeß gleichzeitig geöffnet haben darf (muß gleich FOPEN_MAX sein)	_SC_STREAM_MAX

Tabelle 1.3: Limits und *name*-Argument für die Funktionen **sysconf**, **pathconf** und **fpathconf**

<i>Limitname</i>	<i>Beschreibung</i>	<i>name-Argument</i>
TZNAME_MAX	maximale Anzahl der Bytes für den Zeitzonen-Namen	_SC_TZNAME_MAX
_POSIX_JOB_CONTROL	zeigt an, ob die entsprechende Implementierung Jobkontrolle unterstützt	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	zeigt an, ob die entsprechende Implementierung saved Set-User-IDs und saved Set-Group-IDs unterstützt	_SC_SAVED_IDS
_POSIX_VERSION	zeigt die entsprechende POSIX.1-Version an	_SC_VERSION
XOPEN_VERSION	zeigt die entsprechende XPG-Version an	_SC_XOPEN_VERSION
LINK_MAX	maximale Anzahl von Links auf eine Datei	_PC_LINK_MAX
MAX_CANON	maximale Anzahl von Bytes in der kanonischen Eingabewarteschlange eines Terminals	_PC_MAX_CANON
MAX_INPUT	maximal verfügbarer Speicherplatz in der Eingabewarteschlange eines Terminals	_PC_MAX_INPUT
NAME_MAX	maximale Anzahl von Bytes für einen Dateinamen	_PC_NAME_MAX
PATH_MAX	maximale Anzahl von Bytes in einem relativen Pfadnamen	_PC_PATH_MAX
PIPE_BUF	maximale Anzahl von Bytes, die in einer atomaren Operation in eine Pipe geschrieben werden können	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	zeigt an, ob die Verwendung von chown nur bestimmten Benutzern erlaubt ist	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	zeigt an, ob Pfadnamen, die länger als NAME_MAX Zeichen sind, zu einem Fehler führen	_PC_NO_TRUNC
_POSIX_VDISABLE	wenn definiert, so kann Sonderbedeutung von speziellen Terminalzeichen mit diesem Wert ausgeschaltet werden	_PC_VDISABLE

Tabelle 1.3: Limits und *name-Argument* für die Funktionen **sysconf**, **pathconf** und **fpathconf**

Rückgabewerte

Bei den Rückgabewerten der drei Funktionen sind folgende Fälle zu unterscheiden:

1. Alle drei Funktionen geben -1 zurück und setzen `errno` auf `EINVAL`, wenn `name` nicht einer der in der dritten Spalte der Tabelle 1.3 angegebenen Namen ist.
2. Bei Angabe von Namen aus Tabelle 1.3, die `MAX` enthalten oder den Namen `_PC_PIPE_BUF`, wird entweder der Wert der entsprechenden Variable (≥ 0) oder -1 (für unbestimmte Werte) zurückgegeben. Im letzteren Fall wird `errno` nicht gesetzt.
3. Der für `_SC_CLK_TCK` zurückgegebene Wert ist die Anzahl von Uhrticks pro Sekunde. Dieser Wert wird verwendet, um den von **times** zurückgegebenen Wert (siehe Kapitel 10.8) in einen Sekundenwert umzurechnen.
4. Der für `_SC_VERSION` zurückgegebene Wert enthält das Jahr (vierstellig) und den Monat der entsprechenden Version, wie z.B. 199207L für Juli 1992.
5. Die bei `_SC_XOPEN_VERSION` zurückgegebene Zahl zeigt die Version von XPG (wie z.B. 4 für XPG4) an, der das aktuelle System entspricht.
6. Wenn **sysconf** bei `_SC_JOB_CONTROL` oder `_SC_SAVED_IDS` den Wert -1 zurückgibt (ohne `errno` zu setzen), so werden Jobkontrolle bzw. saved Set-User-/Group-IDs nicht unterstützt. Beide Konstanten können auch zur Kompilierungszeit mit den entsprechenden Konstanten aus der Headerdatei `<unistd.h>` erfragt werden.
7. Bei den Namen `_PC_CHOWN_RESTRICTED` und `_PC_NO_TRUNC` wird -1 zurückgegeben (ohne Setzen von `errno`), wenn diese Konstanten nicht für `pathname` oder `fd` gesetzt sind.
8. Bei dem Namen `_PC_VDISABLE` wird -1 zurückgegeben (ohne Setzen von `errno`), wenn diese Konstante nicht für `pathname` oder `fd` gesetzt ist. Falls diese Konstante gesetzt ist, ist der Rückgabewert das Zeichen, mit dem spezielle Terminaleingabezeichen ausgeschaltet werden können.

Einschränkungen für `pathconf` und `fpathconf`

1. Die bei `_PC_LINK_MAX` angegebene Datei kann entweder eine Datei oder ein Directory sein. Der Rückgabewert bei einem Directory gilt dabei für das Directory und nicht für die Dateien in diesem Directory.
2. Die bei `_PC_NAME_MAX` und `_PC_NO_TRUNC` angegebene Datei muß ein Directory sein. Der Rückgabewert gilt dabei für die Dateien in diesem Directory.
3. Die bei `_PC_PATH_MAX` angegebene Datei muß ein Directory sein. Der zurückgegebene Wert ist die maximale Länge von relativen Pfadnamen, wenn das angegebene Directory das Working-Directory ist. Dies ist jedoch nicht die wirkliche maximale Länge eines absoluten Pfadnamens (siehe auch das Programm 1.11, `pathmax.c`).
4. Die bei `_PC_PIPE_BUF` angegebene Datei muß entweder eine Pipe, eine FIFO oder ein Directory sein. Wenn ein Directory angegeben wurde, so wird das Limit für eine FIFO in diesem Directory zurückgegeben.

5. Die bei `_PC_MAX_CANON`, `_PC_MAX_INPUT` und `_PC_VDISABLE` angegebene Datei muß eine Terminaldatei sein.
6. Die bei `_PC_CHOWN_RESTRICTED` angegebene Datei muß entweder eine Datei oder ein Directory sein. Bei Angabe eines Directorys zeigt der Rückgabewert an, ob diese Option für Dateien in diesem Directory eingeschaltet ist.

Das folgende Programm 1.10 (`syslimit.c`) gibt alle Limits aus Tabelle 1.3 aus.

```
#include <errno.h>
#include "eighdr.h"

static void sysconf_limits(char *name, int kwert);
static void pathconf_limits(char *name, int kwert, char *pfad);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        fehler_meld(FATAL, "%s directory", argv[0]);

    printf("-----\n");
    sysconf_limits("ARG_MAX", _SC_ARG_MAX);
    sysconf_limits("CHILD_MAX", _SC_CHILD_MAX);
    sysconf_limits("NGROUPS_MAX", _SC_NGROUPS_MAX);
    sysconf_limits("OPEN_MAX", _SC_OPEN_MAX);
#ifdef _SC_STREAM_MAX
    sysconf_limits("STREAM_MAX", _SC_STREAM_MAX);
#endif
#ifdef _SC_TZNAME_MAX
    sysconf_limits("TZNAME_MAX", _SC_TZNAME_MAX);
#endif
    sysconf_limits("_POSIX_JOB_CONTROL", _SC_JOB_CONTROL);
    sysconf_limits("_POSIX_SAVED_IDS", _SC_SAVED_IDS);
    sysconf_limits("_POSIX_VERSION", _SC_VERSION);
    sysconf_limits("Uhrticks pro Sekunde", _SC_CLK_TCK);
    printf("-----\n");
    pathconf_limits("MAX_CANON", _PC_MAX_CANON, "/dev/tty");
    pathconf_limits("MAX_INPUT", _PC_MAX_INPUT, "/dev/tty");
    pathconf_limits("_POSIX_VDISABLE", _PC_VDISABLE, "/dev/tty");
    pathconf_limits("LINK_MAX", _PC_LINK_MAX, argv[1]);
    pathconf_limits("NAME_MAX", _PC_NAME_MAX, argv[1]);
    pathconf_limits("PATH_MAX", _PC_PATH_MAX, argv[1]);
    pathconf_limits("PIPE_BUF", _PC_PIPE_BUF, argv[1]);
    pathconf_limits("_POSIX_NO_TRUNC", _PC_NO_TRUNC, argv[1]);
    pathconf_limits("_POSIX_CHOWN_RESTRICTED", _PC_CHOWN_RESTRICTED, argv[1]);
    printf("-----\n");

    exit(0);
}

static void sysconf_limits(char *name, int kwert)
{
    long wert;
```

```

printf("%30s = ", name);
errno = 0;
if ( (wert = sysconf(kwert)) < 0) {
    if (errno != 0)
        fehler_meld(WARNUNG_SYS, "sysconf-Fehler");
    printf("nicht definiert\n");
} else
    printf("%12ld\n", wert);
}

static void pathconf_limits(char *name, int kwert, char *pfad)
{
    long wert;

    printf("%30s = ", name);
    errno = 0;
    if ( (wert = pathconf(pfad, kwert)) < 0) {
        if (errno != 0)
            fehler_meld(WARNUNG_SYS, "pathconf-Fehler bei %s", pfad);
        printf("unlimitiert\n");
    } else
        printf("%12ld\n", wert);
}

```

Programm 1.10 (syslimit.c): Ausgabe aller möglichen **sysconf**- und **pathconf**-Werte

Nachdem man das Programm 1.10 (syslimit.c) kompiliert und gelinkt hat

```
cc -o syslimit syslimit.c fehler.c
```

kann es z.B. die folgende Ausgabe (unter Linux) liefern:

```
$ syslimit .
```

```

-----
ARG_MAX =          131072
CHILD_MAX =         999
NGROUPS_MAX =       32
OPEN_MAX =          256
_POSIX_JOB_CONTROL = 1
_POSIX_SAVED_IDS =  1
_POSIX_VERSION =    199009
Uhrticks pro Sekunde = 100
-----
MAX_CANON =         255
MAX_INPUT =         255
_POSIX_VDISABLE =    0
LINK_MAX =          127
NAME_MAX =          255
PATH_MAX =          1024
PIPE_BUF =          4096
_POSIX_NO_TRUNC =    1
_POSIX_CHOWN_RESTRICTED = 1
-----
$

```

1.11.6 Überblick über die Limits

Tabelle 1.4 faßt noch einmal alle zuvor besprochenen Limits alphabetisch zusammen. Es werden dabei folgende Abkürzungen in der Spalte für Kompilierungszeitkonstanten verwendet:

l <limits.h>
s <stdio.h>
u <unistd.h>

* optional. Ist kein * angegeben, so muß Konstante in entsprechender Headerdatei definiert sein.

Konstante	Kompilierungszeit (Header)	Laufzeitname	Minimalwert
ARG_MAX	l*	_SC_ARG_MAX	_POSIX_ARG_MAX=4096
CHAR_BIT	l		8
CHAR_MAX	l		127
CHAR_MIN	l		0
CHILD_MAX	l	_SC_CHILD_MAX	_POSIX_CHILD_MAX=6
FOPEN_MAX	s		8
INT_MAX	l		32767
INT_MIN	l		-32768
LINK_MAX	l*	_PC_LINK_MAX	_POSIX_LINK_MAX=8
LONG_MAX	l		2147483647
LONG_MIN	l		-2147483648
MAX_CANON	l*	_PC_MAX_CANON	_POSIX_MAX_CANON=255
MAX_INPUT	l*	_PC_MAX_INPUT	_POSIX_MAX_INPUT=255
MB_LEN_MAX	l		
NAME_MAX	l*	_PC_NAME_MAX	_POSIX_NAME_MAX=14
NGROUPS_MAX	l	_SC_NGROUPS_MAX	_POSIX_NGROUPS_MAX=0
NL_ARGMAX	l		9
NL_LANGMAX	l		14
NL_MSGMAX	l		32767
NL_NMAX	l		
NL_SETMAX	l		255
NL_TEXTMAX	l		255

Tabelle 1.4: Zusammenfassung der Kompilierungszeit- und Laufzeitkonstanten

Konstante	Kompilierungszeit (Header)	Laufzeitname	Minimalwert
NZERO	l		20
OPEN_MAX	l*	_SC_OPEN_MAX	_POSIX_OPEN_MAX=16
PASS_MAX	l*	_SC_PASS_MAX	8
PATH_MAX	l*	_PC_PATH_MAX	_POSIX_PATH_MAX=255
PIPE_BUF	l*	_PC_PIPE_BUF	_POSIX_PIPE_BUF=512
SCHAR_MAX	l		127
SCHAR_MIN	l		-127
SHRT_MAX	l		32767
SHRT_MIN	l		-32768
SSIZE_MAX	l		_POSIX_SSIZE_MAX=32767
STREAM_MAX	l*	_SC_STREAM_MAX	_POSIX_STREAM_MAX=8
TMP_MAX	s		10000
TZNAME_MAX	l*	_SC_TZNAME_MAX	_POSIX_TZNAME_MAX=3
UCHAR_MAX	l		255
Uhrticks / Sekunde		_SC_CLK_TCK	
UINT_MAX	l		65535
ULONG_MAX	l		4294967295
USHRT_MAX	l		65535
_POSIX_CHOWN_RESTRICTED	u*	_PC_CHOWN_RESTRICTED	
_POSIX_JOB_CONTROL	u*	_SC_JOB_CONTROL	
_POSIX_NO_TRUNC	u*	_PC_NO_TRUNC	
_POSIX_SAVED_IDS	u*	_PC_SAVED_IDS	
_POSIX_VDISABLE	u*	_PC_VDISABLE	
_POSIX_VERSION	u	_SC_VERSION	
_XOPEN_VERSION	u	_SC_XOPEN_VERSION	

Tabelle 1.4: Zusammenfassung der Kompilierungszeit- und Laufzeitkonstanten (Forts.)

Laufzeitnamen in Tabelle 1.4, die mit `_SC_` beginnen, sind Argumente für die Funktion **sysconf**, und Laufzeitnamen, die mit `_PC_` beginnen, sind Argumente für die Funktionen **pathconf** und **fpathconf**.

1.11.7 Unbestimmte Laufzeitlimits

Die in Tabelle 1.4 mit einem »*« gekennzeichneten optionalen Konstanten, deren Name MAX enthält, und die Konstante PIPE_BUF können unbestimmte Werte haben. Für Programme, die mit diesen eventuell unbestimmten Konstanten arbeiten, besteht nun das Problem, daß die Konstanten eventuell nicht in `<limits.h>` definiert sind, so daß sie nicht zur Kompilierungszeit verwendet werden können. Zur Laufzeit können sie aber auch nicht verwendet werden, da ihr Wert unbestimmt, also nicht festgelegt ist.

Das folgende Programm 1.11 (`pathmax.c`) zeigt, wie man dieses Problem beheben kann. Es enthält eine Funktion `pathmax`, die als Rückgabewert die maximale Länge eines Pfadnamens im jeweiligen System liefert. Der Aufrufer dieser Routine müßte dann mit **malloc** einen Speicherplatz dieser Größe plus 1 (wegen abschließendes `\0`) allokieren, um dann z.B. Funktionen wie **getcwd** aufzurufen. Die Funktion **getcwd** schreibt den Pfadnamen des Working-Directories in den Puffer, dessen Adresse ihm als erstes Argument übergeben wird.

```
#include <errno.h>
#include <limits.h>
#include "eighdr.h"

#ifdef PATH_MAX
static int maxpfad = PATH_MAX; /* zur Kompilierungszeit festgelegt */
#else
static int maxpfad = 0;        /* muss zur Laufzeit bestimmt werden */
#endif

int pathmax(void)
{
    if (maxpfad == 0) {
        errno = 0;

        /* maximalen Pfad relativ zum Root-Directory bestimmen */
        if ( (maxpfad = pathconf("/", _PC_PATH_MAX)) < 0 ) {
            if (errno == 0)
                maxpfad = 1024; /* unbestimmt; also wird einfach 1024 angenommen */
            else
                fehler_meld(FATAL_SYS, "pathconf-Fehler bei _PC_PATH_MAX");
        } else {
            maxpfad++; /* +1 wegen "relativ zum root-Directory" */
        }
    }
    return(maxpfad);
}

#ifdef TEST
int
main(void)
{
    int    pfadlaenge;
    char   *pfad;
```

```

pfadlaenge = pathmax();
printf("Maximale Pfadlaenge: %d\n", pfadlaenge);

if ( (pfad = malloc(pfadlaenge+1)) == NULL)
    fehler_meld(FATAL_SYS, "Speicherplatzmangel");

if (getcwd(pfad, pfadlaenge+1) == NULL)
    fehler_meld(FATAL_SYS, "getcwd-Fehler");
printf("Working Directory: %s\n", pfad);

exit(0);
}
#endif

```

Programm 1.11 (pathmax.c): Erfragen der maximalen Pfadlänge, selbst wenn unbestimmt

Nachdem man das Programm 1.11 (pathmax.c) kompiliert und gelinkt hat.

```
cc -o pathmax pathmax.c fehler.c -DTEST
```

liefert es z.B. die folgende Ausgabe:

```

$ pathmax
Maximale Pfadlaenge: 1024
Working Directory: /home/hh/sysprog/kap1
$

```

Die hier gezeigte Technik kann auch in ähnlicher Form für die anderen eventuell unbestimmten Werte in Tabelle 1.4 verwendet werden.

1.11.8 Konstante `_POSIX_SOURCE`

Neben den durch POSIX.1 standardisierten Konstanten kann jede Implementierung noch weitere implementierungsspezifische Konstanten definieren. Wenn ein Programm absolut POSIX.1-konform sein soll und keine implementierungsspezifischen Konstanten verwendet, so kann dies dem Compiler mit der Definition der Konstante `_POSIX_SOURCE` mitgeteilt werden, wie z.B.:

```

cc -o prog .... -D_POSIX_SOURCE    (auf der Kommandozeile) oder
#define _POSIX_SOURCE                (in der 1. Zeile des Quellprogramms)

```

1.11.9 Primitive Systemdatentypen

Die Headerdatei `<sys/types.h>` definiert (mit **typedef**) implementierungsabhängige Datentypen, die sogenannten primitiven Systemdatentypen. Durch die Definition dieser Datentypen, die auch in anderen Headerdateien definiert sein können, können implementierungsunabhängige Programme erstellt werden.

Nehmen wir als Beispiel den Datentyp `ino_t`, der für die Speicherung von sogenannten *inodes* vorgesehen ist. Während hierfür ein System z.B. **unsigned int** vorsieht, kann ein anderes System, das mehr *inodes* zulässt, hierfür **unsigned long** festlegen. Bei der Kompi-

lierung des Programms wird in jedem Fall der für das entsprechende System geeignete Datentyp verwendet, ohne daß irgendwelche Änderungen am jeweiligen Programm notwendig sind. Tabelle 1.5 zeigt die Systemdatentypen, die in diesem Buch vorkommen.

<i>Datentyp</i>	<i>Kurzbeschreibung</i>
caddr_t	Speicheradresse (15.3)
clock_t	Uhrticks (7.1)
dev_t	Gerätenummern (5.10)
fd_set	Filedeskriptor-Mengen (15.1)
fpos_t	Schreib / Lesezeiger-Position in Datei (3.6)
gid_t	Gruppen-IDs (5)
ino_t	inode-Nummern (5)
mode_t	Eröffnungsmodus für Dateien (5)
nlink_t	Linkzähler (5)
off_t	Dateigrößen und Offsets (4.4)
pid_t	Prozeß-IDs und Prozeßgruppen-IDs (10.1 und 11.1)
ptrdiff_t	Ergebnis bei Zeigersubtraktion (2.4)
rlim_t	Ressourcenlimits (9.5)
sig_atomic_t	Datentyp, der atomare Zugriffe ermöglicht (13.6)
sigset_t	Signalmengen (13.4)
size_t	Größe von Objekten (4.3)
ssize_t	Rückgabetyt von Funktionen, die eine Byteanzahl liefern (4.3)
time_t	Zähler für die Kalenderzeitsekunden (7.1)
uid_t	User-IDs (7.1)
wchar_t	Vielbyte-Zeichen (2.4)

Tabelle 1.5: Primitive Systemdatentypen

1.12 Erste Einblicke in den Linux-Systemkern



Dieses Kapitel ist nur für die Leser gedacht, die an Interna des Linux-Kerns interessiert sind. Es kann übergangen werden, wenn man nur die Programmierung des jeweiligen Unix-Systems unter Zuhilfenahme der angebotenen Systemfunktionen kennenlernen möchte. Lesern dagegen, die an der Umsetzung von Betriebssystemkonzepten und -algorithmen interessiert sind oder die selbst Kernroutinen oder systemnahe Funktionen (wie z.B. Gerätetreiber) programmieren möchten, gibt es erste wesentliche Einblicke in den Linux-Systemkern.

In diesem Kapitel wird zunächst ein Überblick über die wichtigsten Directories gegeben, in denen sich die Quellprogramme und die zugehörigen Headerdateien des Linux-Kerns befinden, bevor kurz auf die Übersetzung und die Konfigurationsmöglichkeiten des Linux-Kerns eingegangen wird.

Ein weiteres umfangreicheres Kapitel zeigt dann den grundlegenden Aufbau des Linux-Systemkerns, klärt wichtige Begriffe und stellt wesentliche Kernalgorithmen und -konzepte vor, die für das Verständnis der späteren Linux-spezifischen Kapitel vorausgesetzt werden.

1.12.1 Directories der Quellprogramme des Linux-Kerns

Die Quellen des Linux-Kerns befinden sich normalerweise im Directory `/usr/src/linux`. Alle entsprechenden Pfadangaben auf den restlichen Seiten dieses Buches werden relativ zu diesem Pfad angegeben. Da Linux zur Zeit vorwiegend auf Intel-x86-Prozessoren eingesetzt wird, konzentriert sich dieses Buch beim Vorstellen von Linux-Konzepten meist auf diese Intel-Architektur.

Nachfolgend ist ein Überblick über die wichtigsten Directories der Linux-Kernquellen gegeben, wobei bei architekturabhängigen Quellen nur die Intel-Architektur detaillierter gezeigt wird:

```
/usr/src/linux/
|----arch/      Architekturabhängige Quellen
|  |----alpha/  Alphaprozessoren
|  |----i386/   Intel-Prozessoren
|  |  |----boot/
|  |  |----kernel/  zentraler (architekturabhängiger)
|  |  |                      Teil des Kerns
|  |  |----lib/
|  |  |----math-emu/
|  |  |----mm/    architekturspezifische Speicherverwaltung
|  |----m68k/   Motorola-Prozessoren
|  |----mips/   MIPS-Architektur
|  |----ppc/    Power-PC
|  |----sparc/  Sparc-Workstations
|----drivers/   Treiber für
|  |----block/  blockorientierte Geräte
|  |----cdrom/  CDROM-Laufwerke (keine SCSI oder IDE)
|  |----char/   zeichenorientierte Geräte
|  |----isdn/   ISDN
|  |----net/    Netzwerkkarten
|  |----pci/    Ansteuerung des PCI-Busses
|  |----sbus/   Ansteuerung des S-Busses von Sparc-Rechnern
|  |----scsi/   SCSI-Interface
|  |----sound/  Soundkarten
|----fs/        Filesysteme (VFS und filesystemspezifische Quellen)
|  |----affs/
|  |----autofs/
|  |----ext/
|  |----ext2/
```

```

| |----fat/
| |----hpfs/
| |----isofs/
| |----minix/
| |----msdos/
| |----ncpfs/
| |----nfs/
| |----proc/
| |----smbfs/
| |----sysv/
| |----ufs/
| |----umsdos/
| |----vfat/
| |----xiafs/
|----include/   kernspezifische Headerdateien
| |----asm@     Link auf das entsprechende Directory
| |              der aktuellen Architektur (in diesem Directory)
| |----asm-alpha/
| |----asm-generic/
| |----asm-i386/
| |----asm-m68k/
| |----asm-mips/
| |----asm-ppc/
| |----asm-sparc/
| |----linux/
| |----net/
| |----scsi/
|----init/      Start des Kerns
|----ipc/       klassische Interprozeßkommunikation (IPC) von System V
|               (Semaphore, Shared Memory und Message Queues)
|----kernel/    zentraler (architekturunabhängiger) Teil des Kerns
|----lib/       C-Standardbibliotheken
|----mm/        (architekturunabhängige) Speicherverwaltung
|----modules/   Module, die bei der Kompilierung des Kerns erzeugt wurden;
|               können dem Linux-Kern später zur Laufzeit mit dem
|               Kommando insmod hinzugefügt werden.
|----net/       Netzwerkprotokolle (TCP, ARP, ...) sowie Sockets
|----vmlinux

```

Der Kern von Linux besteht im wesentlichen nur aus C-Programmen, die sich in zwei Punkten von sonstigen C-Programmen unterscheiden:

- ▶ Beim Linux-Kern ist die Startfunktion nicht `int main(int argc, char *argv[])`, sondern `start_kernel(void)`.
- ▶ Es existiert noch kein Programm-Environment.

Dies bedeutet, daß vor dem Aufruf der ersten C-Funktion zunächst einige architekturspezifische Aktionen, wie z.B. das Konfigurieren der Hardware, das Laden des Kerns, Installation von Interruptservice-Routinen usw. notwendig sind. Die dafür verantwortlichen Assemblerprogramme befinden sich in architekturspezifischen Directories (z.B. `arch/i386/boot` oder `arch/i386/kernel`).

Die dann für den Start des Kerns zuständigen Funktionen sind im Directory `init`. Hier befindet sich z.B. auch die Funktion **`start_kernel`** (in `init/main.c`), deren Aufgabe die Initialisierung des Kerns entsprechend der übergebenen Bootparameter ist. Hierzu gehört auch die Erzeugung des Urprozesses, was ohne Zuhilfenahme der Funktion **`fork`** erfolgen muß.

Hervorzuheben ist an dieser Stelle noch das Subdirectory `include`, das alle kernspezifischen Headerdateien enthält.

Dabei ist `include/asm` immer ein symbolischer Link auf die für die aktuelle Architektur gültigen Headerdateien, wie z.B. bei Intel-PCs:

```
/usr/src/linux/include/asm -> asm-i386/
```

Im Directory `/usr/include` befinden sich dann ebenso Links auf die beiden Subdirectories `include/linux` und `include/asm`:

```
/usr/include/linux -> ../src/linux/include/linux/  
/usr/include/asm -> ../src/linux/include/asm-i386/
```

Diese Links ermöglichen ein leichtes Austauschen der Headerdateien, wenn diese sich in einer neueren Version geändert haben. `/usr/include` enthält somit immer automatisch die aktuell gültigen Headerdateien.

1.12.2 Generieren und Installieren eines neuen Linux-Kerns

Das Erzeugen eines neuen Linux-Kerns erfolgt im Directory `/usr/src/linux` in den folgenden Schritten⁶:

Konfigurieren des Kerns

Dazu muß der Superuser folgendes aufrufen:

make config

Dabei wird das Shellskript `scripts/Configure` ausgeführt. Es liest die architekturabhängige Konfigurationsdatei `config.in` (z.B. `arch/i386/config.in`), in der sich die entsprechenden Konfigurationsangaben für den Kern befinden, und fragt den Aufrufer, welche Komponenten in den Kern aufzunehmen sind. Diese Datei `config.in` liest ihrerseits die Dateien `Config.in` in den Directories der jeweiligen Subsysteme des Kerns, wie z.B. `source fs/Config.in` oder `source drivers/char/Config.in`.

Möchte man menügesteuert auf einem textbasierten Terminal installieren, muß man folgendes aufrufen:

make menuconfig

6. Hier wird die Generierung des Kerns unter S.u.S.E.Linux beschrieben. Die dabei angegebenen Schritte gelten aber auch für die meisten anderen Linux-Distributionen.

Für eine menügeführte Installation unter X Windows ist folgendes aufzurufen:

make xconfig

Das Shellskript `scripts/Configure` erstellt sowohl die Datei `<linux/autoconf.h>`, die für die bedingte Kompilierung innerhalb der Kern-Quellen sorgt, und die Datei `.config`, die bei einem erneuten Aufruf von `Configure` verwendet wird, um die Antworten von einer vorherigen Konfiguration als Standardantworten anzubieten. Ruft man bei einer erneuten Konfiguration

make oldconfig

auf, werden alle Standardwerte ohne jegliche Rückfragen als Antworten auf die einzelnen Fragen genommen. Dieser Aufruf ermöglicht es, eine früher erstellte Konfiguration für eine neue Linux-Version wiederzuverwenden, so daß der neue Kern mit der gleichen Konfiguration generiert wird.

Erweiterungen für den Linux-Kern müssen in der Datei `config.in` bzw. in der Datei `Config.in` eingetragen werden. Die dabei zu verwendenden Angaben sind an zwei Einträgen in der Datei `/usr/src/linux/drivers/block/Config.in` gezeigt:

```
bool 'Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support' CONFIG_BLK_DEV_IDE
tristate 'Normal floppy disk support' CONFIG_BLK_DEV_FD
```

Die Angabe `bool` bedeutet, daß hier bei der Konfiguration des Kerns nur **y(es)** oder **n(o)** eingegeben werden kann. Bei der Angabe `tristate` sind drei Antworten möglich: **y(es)**, **n(o)** oder **m(odule)**; **m** bedeutet, daß die entsprechende Komponente als Modul zu erstellen ist, das zur Laufzeit mit dem Kommando ***insmod*** installiert werden kann.

Generieren des Kerns und der Module

Um die Abhängigkeiten der Quellprogramme untereinander neu zu erstellen, muß folgendes aufgerufen werden:

make dep

Diese Abhängigkeiten werden in die Dateien `.depend` in den einzelnen Subdirectories hinterlegt und später in den entsprechenden Makefiles eingefügt. Danach sollten eventuell von früheren Generierungen vorhandene Restbestände beseitigt werden, was sich mit folgendem Aufruf erreichen läßt:

make clean

Die eigentliche Generierung des Kerns erfolgt dann mit:

make zImage

Diese drei Aufrufe lassen sich zu einem Aufruf zusammenfassen:

make dep clean zImage

Nach einer erfolgreichen Kerngenerierung befindet sich der komprimierte, bootfähige Kern in der Datei `arch/i386/boot/zImage`.

Wenn Teile des Kerns als ladbare Module konfiguriert wurden, muß man anschließend noch das Übersetzen dieser Module veranlassen:

make modules

Wurden die entsprechenden Module erfolgreich erzeugt, muß man sie mit dem folgenden Aufruf installieren:

make modules_install

Dieser Aufruf bewirkt, daß die Module in die entsprechenden Subdirectories `block`, `cdrom`, `net`, `scsi`, `fs`, `misc` usw. des Directorys `/lib/modules/kernversion` kopiert werden.

Installieren des Kerns

Nachdem der Kern generiert wurde, muß man noch dafür sorgen, daß er in Zukunft gebootet wird. Möchte man den Bootmanager LILO (**LinuxLoader**) verwenden, so ist dieser neu zu installieren, was sich mit den beiden folgenden Aufrufen erreichen läßt:

cp arch/i386/boot/zImage /vmlinuz

lilo

Vor diesen Schritten empfiehlt sich jedoch ein Sichern des alten Kerns, um notfalls – wenn etwas schief lief – immer noch booten zu können. Dazu ist zunächst der folgende Aufruf notwendig

cp /vmlinuz /vmlinuz.old

Danach sollte man noch den Eintrag in `/etc/lilo.conf` entsprechend ändern (`vmlinuz` à `vmlinuz.old`). So stellt man sicher, daß man immer noch mit dem alten Kern booten kann.

Die Installation des Kerns kann auch mit dem folgenden Aufruf erreicht werden, der automatisch die zuvor beschriebenen Schritte durchführt.

make zlilo

Dieser Aufruf kopiert den generierten Kern nach `/vmlinuz`, der alte Kern wird in `/vmlinuz.old` umbenannt. Danach erfolgt die Installation des Linux-Kerns durch den Aufruf von **lilo**. Auch bei diesem Aufruf sollte zuvor die Datei `/etc/lilo.conf` entsprechend angepaßt werden.

Möchte man sich eine Bootdiskette mit dem neuen Kern erstellen, muß nur folgendes aufgerufen werden:

make zdisk

Aktualisieren von Teilen des Linux-Kerns

Ändert man Teile eines Linux-Kerns, wie z.B. in dem Fall, daß man einen neuen Treiber geschrieben hat, den man in den Kern aufnehmen möchte, so muß man nicht den ganzen Kern neu übersetzen, sondern man kann statt dessen nur das jeweilige Teil neu übersetzen lassen, wie z.B.

make drivers

Durch diesen Aufruf werden nur die Quellprogramme im Subdirectory `drivers`, wo sich die Treiber befinden, neu übersetzt. Durch diesen Aufruf wird allerdings noch kein neuer Kern generiert. Dazu müßte man den Kern mit dem folgenden Aufruf neu linkern:

make SUBDIRS=drivers

1.12.3 Konfigurieren des Kerns in den Quellprogrammen

In einigen wenigen Fällen kann es notwendig sein, die Quellprogramme selbst zu ändern, um entsprechende Einstellungen für den zu generierenden Kern vorzunehmen. Nachfolgend werden einige solche Fälle beschrieben.

Einstellen der Zielmaschine für den Kern (im Makefile)

Wenn man keinen Intel-PC mit einem x86-Prozessor hat, muß man im Makefile im Directory `/usr/src/include` die entsprechende Architektur einstellen. Hierzu ist dann die Zeile

```
ARCH = i386
```

in diesem Makefile entsprechend zu ändern, wie z.B. für einen Alpha-Prozessor:

```
ARCH = alpha
```

oder für einen SPARC-Rechner:

```
ARCH = sparc
```

Weitere Architekturen werden vorläufig nur teilweise unterstützt.

Weitere Konfigurationsmöglichkeiten im Makefile

Weitere Konfigurationsmöglichkeiten im Makefile sind nachfolgend kurz vorgestellt.

Möchte man einen Kern mit SMP-Unterstützung (SMP steht für *Symmetric Multi Processing*) generieren, muß man bei der Zeile `SMP = 1` das Kommentarzeichen `#` entfernen:

```
# For SMP kernels, set this. We don't want to have this in the config file
# because it makes re-config very ugly and too many fundamental files depend
# on "CONFIG_SMP"
#
# NOTE! SMP is experimental. See the file Documentation/SMP.txt
#
# SMP = 1 ← Hier das Kommentarzeichen # entfernen
```

```
#
# SMP profiling options
# SMP_PROF = 1    ← Eventuell auch hier das Kommentarzeichen # entfernen
```

Des weiteren könnten die nachfolgend fett gedruckten Zeilen in diesem Makefile den eigenen Bedürfnissen angepaßt werden:

```
#
# INSTALL_PATH specifies where to place the updated kernel and system map
# images. Uncomment if you want to place them anywhere other than root.

#INSTALL_PATH=/boot

#
# If you want to preset the SVGA mode, uncomment the next line and
# set SVGA_MODE to whatever number you want.
# Set it to -DSVGA_MODE=NORMAL_VGA if you just want the EGA/VGA mode.
# The number is the same as you would ordinarily press at bootup.

SVGA_MODE = -DSVGA_MODE=NORMAL_VGA

.....
#
# if you want the ram-disk device, define this to be the
# size in blocks.
#

#RAMDISK = -DRAMDISK=512
```

Natürlich können beliebig weitere Änderungen an dem Makefile vorgenommen werden, so lange man sich bewußt ist, welche Auswirkungen dies hat.

Einstellen der maximal möglichen Anzahl von Prozessen (in include/linux/tasks.h)

Die maximal mögliche Anzahl der Prozesse ist mit

```
#define NR_TASKS    512
```

in der Datei `include/linux/tasks.h` festgelegt. Soll diese erhöht oder erniedrigt werden, muß hier anstelle von 512 die neue gewünschte maximale Anzahl von Prozessen angegeben werden.

Einstellen der maximal möglichen Filesysteme (in include/linux/fs.h)

Die maximal mögliche Anzahl von Filesystemen, die der Kern unterstützt, ist mit

```
#define NR_SUPER 64
```

in der Datei `include/linux/fs.h` festgelegt. Soll diese erhöht oder erniedrigt werden, muß hier anstelle von 64 die neue gewünschte maximale Anzahl von Filesystemen angegeben werden.

Dies sind natürlich nicht alle Konfigurationsmöglichkeiten des Linux-Kerns, sondern nur ein kleiner Ausschnitt aus der Vielzahl der Einstellmöglichkeiten.

1.12.4 Einführung in wichtige Algorithmen und Konzepte des Linux-Kerns

Dieses Kapitel zeigt den grundlegenden Aufbau des Linux-Systemkerns, klärt Begriffe und stellt wesentliche Algorithmen, Konzepte und Datenstrukturen des Linux-Kerns vor.

Allgemeine Daten zum Linux-Kern

Der gesamte Linux-Kern der Version 2.0 für die Intel-Architektur umfaßt nahezu eine halbe Million Zeilen C-Code und etwa 8000 Zeilen Assembler-Code. Die Implementierung der Gerätetreiber nimmt bereits einen Großteil des C-Codes (fast 400.000 Zeilen) ein. Der Assembler-Code dagegen umfaßt vorwiegend die folgenden Implementierungen (fast 7000 Zeilen): Emulation des mathematischen Koprozessors, Ansteuerung der Hardware und Booten des Systems. Die zentralen Routinen des eigentlichen Kerns (Prozeß- und Speicherverwaltung) umfassen nur etwa fünf Prozent des Codes.

Da es inzwischen möglich ist, eine große Zahl von Treibern aus dem Kern auszulagern, die dann später als eigenständige, unabhängige Module bei Bedarf nachgeladen werden können, kann der eigentliche Linux-Kern klein gehalten werden, was große Vorteile mit sich bringt.

Prozesse, Tasks und Threads

Linux hat das Unix-Prozeßmodell übernommen und um einige neue Ideen erweitert, um eine wirklich schnelle Thread-Implementierung möglich zu machen. In den ersten Unix-Implementierungen war ein Prozeß ein gerade ablaufendes Programm. Für jedes Programm hat sich der Kern dabei z.B. folgende Informationen gehalten:

- ▶ aktuelles Working-Directory des Prozesses
- ▶ vom Prozeß geöffnete Dateien
- ▶ aktuelle Ausführungsposition, oft auch *Kontext* des Prozesses genannt
- ▶ Zugriffsrechte des Prozesses
- ▶ Speicherbereiche, auf die der Prozeß Zugriff hat

Ein Prozeß war somit auch die Basiseinheit für das Multitasking des Betriebssystems.

Auch in Linux gilt noch, daß Prozesse unabhängig nebeneinander existieren und sich nicht direkt gegenseitig beeinflussen können. Der eigene Speicherbereich eines Prozesses ist vor dem Zugriff anderer Prozesse geschützt.

Intern dagegen arbeitet der Linux-Kern mit einem Konzept, das man als *kooperatives Multitasking* bezeichnet. Hierbei entscheidet jede Task selbst, wann sie die Steuerung an eine andere Task abgibt. Im Unterschied zu einem Prozeß, der keinen Zugriff auf die Ressour-

cen anderer Prozesse hat, kann jede Task auf alle Ressourcen anderer Tasks zugreifen. Dies gilt jedoch nur für die Teile einer Task, die im privilegierten Systemmodus ablaufen, während die anderen Teile, die im nicht privilegierten Benutzermodus ablaufen, keinen Zugriff auf die Ressourcen anderer Tasks haben. Diese nicht privilegierten Teile einer Task stellen sich unter Linux nach außen hin als Prozesse dar. Für diese nicht privilegierten Tasks, die Prozesse also, findet somit ein echtes Multitasking statt.

Abbildung 1.4 zeigt die interne und externe Sicht von Prozessen unter Linux.

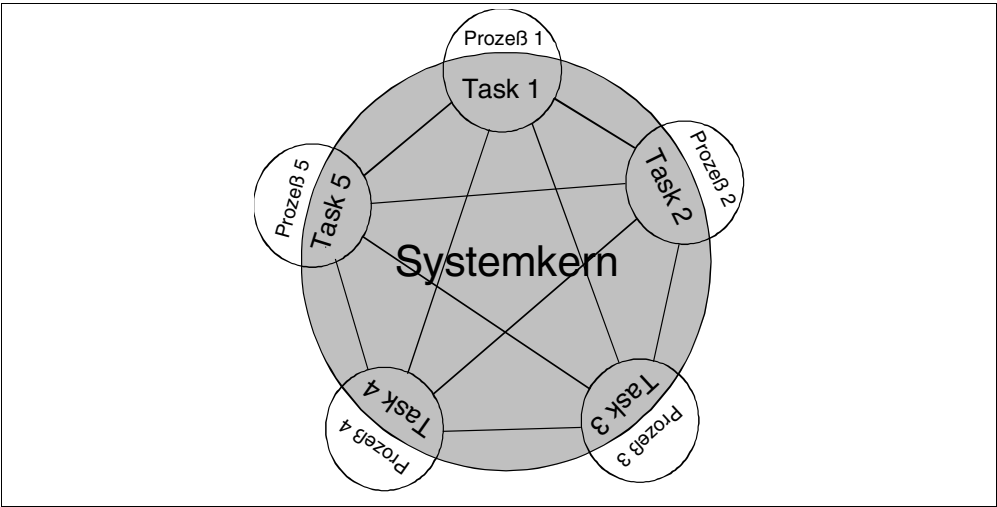


Abbildung 1.4: Interne und externe Sicht von Prozessen unter Linux

In diesem Buch wird jedoch auf diese Unterscheidung von Prozessen und Tasks verzichtet. Statt dessen wird immer der Begriff *Prozeß* verwendet, der auch Tasks miteinschließt.

Eine sich im privilegierten Systemmodus befindende Task kann unterschiedliche Zustände annehmen, wie dies in Abbildung 1.5 gezeigt ist.

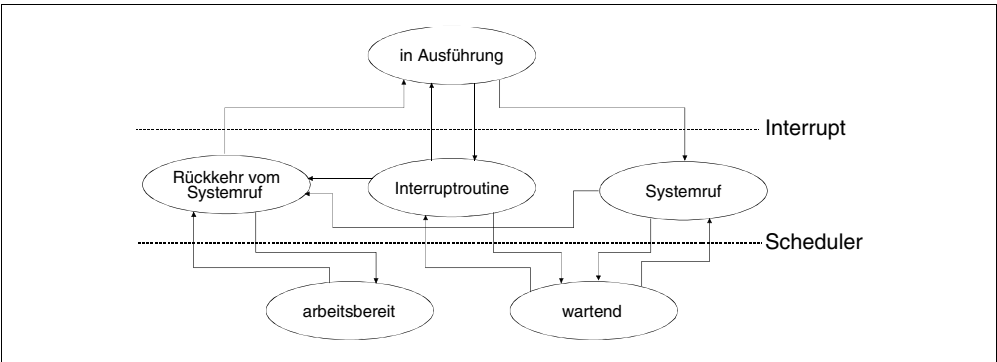


Abbildung 1.5: Zustandsdiagramm eines Prozesses (aus Linux-Kernel-Programmierung; M. Beck, u.a.)

Zustandsübergänge sind in diesem Diagramm durch Pfeile angegeben. Die einzelnen Zustände sind nachfolgend kurz erläutert:

In Ausführung bedeutet, daß die Task gerade aktiv ist und sich im nicht privilegierten Benutzermodus befindet. Ein Wechsel von diesem Zustand zu einem anderen Zustand (im privilegierten Systemmodus) ist nur durch einen Interrupt oder einem Systemruf möglich.

Eine *Interruptroutine* wird aktiv, wenn die Hardware ein Signal schickt, wie z.B. beim Ablauf der zugeordneten Zeitscheibe oder bei einer Tastatureingabe.

Systemrufe werden bei auftretenden Software-Interrupts gestartet.

Wartend bedeutet, daß ein Prozeß auf ein externes Ereignis wartet. Erst nach dem Auftreten dieses Ereignisses setzt der Prozeß seine Arbeit fort.

Im Zustand *Rückkehr vom Systemruf* wird geprüft, ob der Scheduler aufzurufen ist und ob Signale abzuarbeiten sind. Der Scheduler kann den Zustand des Prozesses auf *arbeitsbereit* setzen und einen anderen Prozeß aktivieren.

Arbeitsbereit bedeutet, daß der Prozeß zwar seine Ausführung fortsetzen könnte, aber warten muß, bis der Prozessor, der zur Zeit von einem anderen Prozeß belegt ist, ihm vom Scheduler zugeteilt wird.

Andere Betriebssysteme kennen sogenannte *Threads*. Threads ermöglichen es Programmen, an verschiedenen Stellen zugleich abzulaufen. Im Unterschied zu Prozessen, die sich nicht direkt gegenseitig beeinflussen können, teilen sich Threads, die von einem Programm erzeugt werden, mehrere Ressourcen, wie z.B. denselben Speicher, die Informationen über offene Dateien, das Working Directory usw., und können sich so gegenseitig beeinflussen. Ändert z.B. ein Thread eine globale Variable, steht dieser neue Wert auch sofort allen anderen Threads zur Verfügung. Viele Unix-Implementierungen (wie z.B. auch System-V) wurden überarbeitet, so daß Threads (und nicht mehr Prozesse) die fundamentalen Verwaltungseinheiten für das Multitasking sind; ein Prozeß ist dort nunmehr eine Sammlung von Threads, die sich bestimmte Ressourcen teilen. Dies erlaubt es dem Systemkern schneller zwischen den einzelnen Threads zu wechseln, als wenn er einen vollständigen Kontextwechsel machen müßte, um zu einem anderen Prozeß zu wechseln. Der Kern in solchen Unix-Systemen ist als ein zweistufiges Prozeßmodell aufgebaut, das zwischen Prozessen und Threads unterscheidet.

Da in Linux die Kontextwechsel schon immer sehr schnell waren, und in etwa der Geschwindigkeit von Thread-Wechseln, die mit dem zweistufigen Prozeßmodell eingeführt wurden, entsprachen, entschied man sich bei Linux für einen anderen Weg: Statt das Linux-Multitasking zu ändern, wurde es Prozessen (Tasks, die im privilegierten Systemmodus arbeiten) erlaubt, ihre Ressourcen untereinander zu teilen. Diese Vorgehensweise ermöglicht es den Linux-Entwicklern, die traditionelle Unix-Prozeßverwaltung beizubehalten, während die Thread-Schnittstelle außerhalb des Kerns aufgebaut werden kann.

Umsetzung von Tasks unter Linux

Die Informationen zu einem Prozeß werden in der Struktur `task_struct` gehalten, die in `<linux/sched.h>` definiert ist. Dabei ist zu beachten, daß auf die ersten Komponenten dieser Struktur auch aus AssemblerROUTINEN heraus zugegriffen wird, wobei hierbei der Zugriff nicht wie in C über den Namen der Komponente, sondern über deren Offset relativ zum Strukturanfang erfolgt. Dies ist auch der Grund, warum die Reihenfolge der ersten Komponenten nicht verändern werden darf, außer man würde auch die entsprechenden AssemblerROUTINEN anpassen. Die Struktur `task_struct` ist wie folgt definiert:

```
struct task_struct {
/* these are hardcoded - don't touch */
    volatile long state;
        /* aktueller Zustand des Prozesses:
            TASK_RUNNING:      gerade aktiv oder wartet auf CPU
            TASK_INTERRUPTIBLE: wartet auf bestimmte Ereignisse;
                               kann durch Signale wieder
                               aktiviert werden.
            TASK_UNINTERRUPTIBLE: wartet auf bestimmte Ereignisse;
                                   kann nur durch Hardware-
                                   bedingungen aktiviert werden.
            TASK_ZOMBIE:       ist ein Zombieprozess, der zwar
                               schon beendet ist, dessen
                               Taskstruktur sich aber noch
                               in der Prozeßtabelle befindet.
            TASK_STOPPED:      Prozeß wurde mit einem der Signale
                               SIGSTOP, SIGSTP, SIGTTIN, SIGTTOU
                               angehalten oder wird von anderen
                               Prozeß durch ptrace überwacht.
            TASK_SWAPPING:     in Version 2.0 ungenutzt          */
    long counter;
        /* Zeit in "Uhrticks", bevor zwangsweises Scheduling
           stattfindet. Da der Scheduler diesen Wert benutzt,
           um nächsten Prozeß auszuwählen, ist dies zugleich
           auch die dynamische Priorität eines Prozesses          */
    long priority;
        /* statische Priorität
           Scheduling-Algorithmus verwendet diesen Wert, um
           eventuell einen neuen counter-Wert zu ermitteln          */
    unsigned long signal;
        /* Bitmap für eingetroffene Signale          */
    unsigned long blocked;
        /* Bitmap der Signale, die später zu bearbeiten sind,
           also deren Bearbeitung zur Zeit blockiert ist          */
    unsigned long flags;
        /* Statusflags; Kombination aus
            PF_PTRACED: gesetzt, wenn Prozeß von anderen
                       Prozeß durch ptrace überwacht wird
            PF_TRACESYS: wie PF_PTRACED, nur bei Systemaufruf
            PF_STARTING: Prozeß wird gerade erzeugt
            PF_EXITING:  Prozeß wird gerade beendet
            ...:         weitere Flags (siehe auch <linux/sched.h>) */
```

```

int errno;
/* Fehlernummer des letzten fehlerhaften Systemaufrufs */
long debugreg[8];
/* Debuggingregister des 80x86-Prozessors */
struct exec_domain *exec_domain;
/* Beschreibung, welches Unix für diesen Prozeß emuliert
   wird; Linux kann nämlich Programme anderer
   Unix-Systeme auf i386-Basis, die dem iBCS2-Standard
   entsprechen, abarbeiten */
struct linux_binfmt *binfmt;
/* beschreibt Funktionen, die für das Laden des Programms
   zuständig sind */
struct task_struct *next_task,
                  *prev_task;
/* Nachfolger und Vorgänger in der doppelt verketteten
   Liste von Task-Strukturen. Auf Anfang und Ende dieser
   Liste zeigt die globale Variable init_task, die wie
   folgt in <linux/sched.h> deklariert ist:
   extern struct task_struct init_task; */
struct task_struct *next_run,
                  *prev_run;
/* Nachfolger und Vorgänger in der doppelt verketteten
   Liste von Prozessen, die auf Zuteilung der CPU
   warten; wird vom Scheduler benutzt;
   auf Anfang und Ende dieser Liste zeigt wieder
   die globale Variable init_task */
unsigned long kernel_stack_page;
/* Adresse des Stacks für den Prozeß,
   wenn er im Systemmodus läuft */
unsigned long saved_kernel_stack;
/* Bei MS-DOS-Emulator (Systemaufruf vm86) wird hier
   der alte Stackpointer gesichert */
int exit_code, exit_signal;
/* Exit-Status und Signal, das Prozeß beendete; kann vom
   Elternprozeß mit wait oder waitpid abgefragt werden */
unsigned long personality;
/* dient zusammen mit der obigen Komponente exec_domain
   der genauen Beschreibung des Unix-Systems, das
   emuliert wird. Für normale Linux-Programme auf
   PER_LINUX (in <linux/personality.h> definiert)
   gesetzt. */
int dumpable:1;
/* Flag zeigt an, ob beim Eintreffen bestimmter Signale ein
   core dump (Speicherabzug) zu erstellen ist oder nicht*/
int did_exec:1;
/* Flag zeigt an, ob Prozeß bereits mit execve durch ein
   neues Programm ersetzt wurde oder ob es sich noch
   um das ursprüngliche Programm handelt */
int pid;
/* Prozeßkennung (Prozeß-ID) */
int pgrp;
/* Prozeßgruppenkennung (Prozeßgruppen-ID) */
int tty_old_pgrp;

```

```

/* Kontrollterminal der alten Prozeßgruppe */
int session;
/* Sessionkennung (Session-ID) */
int leader;
/* zeigt an, ob Prozeß Session-Führer (session leader) ist */
int groups[NGROUPS];
/* enthält Zusatz-Group-IDs, denen der Prozeß noch angehört.
   Anders als bei der Komponente gid (siehe weiter unten)
   wird hier der Datentyp int verwendet, da nicht
   benutzte Einträge im Array groups den Wert NOGROUP
   (-1) haben. NGROUPS ist in <asm/param.h> definiert:
   #define NGROUPS 32 */
struct task_struct *p_opptr, /* ursprünglicher Elternprozeß */
                  *p_pptr, /* aktueller Elternprozeß */
                  *p_cptra, /* jüngster Kindprozeß */
                  *p_ysptra, /* nächst jüngerer Kindprozeß */
                  *p_osptr; /* nächst älterer Kindprozeß */
struct wait_queue *wait_chldexit;
/* Warteschlange für den Systemaufruf wait4
   Ein Prozeß, der wait4 aufruft, soll bis zur Beendigung
   seines Kindprozesses unterbrochen werden. Dazu trägt
   er sich in diese Warteschlange ein, setzt sein
   Statusflag auf TASK_INTERRUPTIBLE und gibt die
   Steuerung an den Scheduler ab. Grundsätzlich gilt, daß
   jeder Prozeß, der sich beendet, dies seinem
   Elternprozeß über diese Warteschlange signalisiert. */
unsigned short uid, /* User-ID des Prozesses */
              euid, /* effektive User-ID des Prozesses */
              suid, /* Set-User-ID des Prozesses */
              fsuid; /* Filesystem-User-ID des Prozesses */
/* Anmerkung: Für die Zugriffe wird nicht die wirkliche
   uid bzw. gid, sondern die effektive User-ID/Group-ID
   euid und egid verwendet. Neu in Linux ist die
   Komponente fsuid bzw. fsgid. Diese werden bei allen
   Filesystemzugriffen verwendet.
   Normalerweise sind alle drei Komponenten gleich
   (uid, euid, fsuid) bzw. (gid, egid, fsgid). Ist aber
   das Set-User-ID- bzw. das Set-Group-ID-Bit gesetzt,
   unterscheiden sich die uid und euid bzw. gid und egid.
   In diesem Fall ist dann normalerweise euid==fsuid bzw.
   egid==fsgid. Durch den Aufruf setfsuid bzw. setfsgid
   kann nun das fsuid bzw. fsgid geändert werden, ohne
   daß das euid bzw. das egid geändert wird.
   Grund für die Einführung von fsuid und fsgid war eine
   Sicherheitslücke im NFS-Dämon. Dieser mußte zum
   Einschränken seiner Rechte bei Filesystemzugriffen
   die euid bzw. egid auf die User-ID bzw. auf die
   Group-ID des anfragenden Benutzers setzen. Dadurch
   wurde es dem Benutzer ermöglicht, dem NFS-Dämon
   Signale zu schicken, wie z.B. auch ein SIGKILL.
   Mit dem neuen fsuid-/fsgid-Konzept ist diese
   Sicherheitslücke nun geschlossen */

```

```

unsigned short gid, /* Group-ID des Prozesses */
                egid, /* effektive Group-ID des Prozesses */
                sgid, /* Set-Group-ID des Prozesses */
                fguid; /* Filesystem-Group-ID des Prozesses */

unsigned long timeout; /* Zeitschaltuhr für Systemaufruf alarm */
unsigned long policy,
                rt_priority;
/* Verwendeter Schedulingalgorithmus für den Prozeß;
   policy kann mit einer der folgenden Konstanten gesetzt
   sein:
   SCHED_OTHER: klassisches Scheduling
   SCHED_RR: Round-Robin; Realtime-Scheduling;POSIX.4*/
   SCHED_FIFO: FIFO-Strategie;
                 Realtime-Scheduling;POSIX.4
   rt_priority enthält die Realtime-Priorität */
unsigned long it_real_value,
                it_prof_value,
                it_virt_value;
/* enthalten die Zeitspanne in Ticks, nach der der Timer
   abgelaufen ist */
unsigned long it_real_incr,
                it_prof_incr,
                it_virt_incr;
/* enthalten die entsprechenden Werte, um den Timer nach
   Ablauf wieder zu initialisieren */

struct timer_list real_timer;
/* wird zur Realisierung des Realtime-Intervalltimers
   benötigt */
long utime, /* Zeit, die Prozeß im Benutzermodus arbeitete */
      stime, /* Zeit, die Prozeß im Systemmodus arbeitete */
      cutime, /* Zeitsumme aller Kindprozesse im Benutzermodus */
      cstime, /* Zeitsumme aller Kindprozesse im Systemmodus */
      start_time; /* Zeitpunkt der Kreierung des Prozesses */

unsigned long min_flt,
                maj_flt,
                nswap,
                cmin_flt,
                cmaj_flt,
                cnswap;
int swappable:1;
unsigned long swap_address;
unsigned long old_maj_flt;
unsigned long dec_flt;
unsigned long swap_cnt;
/* Swap- und Page(Faults)-Informationen */

struct rlimit rlim[RLIM_NLIMITS];
/* Limits für die Systemressourcen des Prozesses;
   können mit den beiden Funktionen setrlimit bzw.
   getrlimit neu festgelegt bzw. erfragt werden. */

```

```

unsigned short used_math;
char comm[16];
    /* Name des vom Prozeß ausgeführten Programms;
       wird für Debugging benötigt */

int link_count;
struct tty_struct *tty; /* NULL if no tty */

struct sem_undo *semundo;
struct sem_queue *semsleeping;
    /* Linux unterstützt das Semaphor-Konzept von System V:
       Ein Prozeß kann ein Semaphor (in semsleeping) setzen
       und damit andere Prozesse blockieren, die auch
       dieses Semaphor setzen möchten. Die anderen Prozesse
       bleiben solange blockiert, bis das Semaphor
       (in semsleeping) wieder freigegeben wird.
       Beendet sich ein Prozeß, der Semaphore belegt hat,
       gibt der Systemkern alle von diesem Prozeß belegten
       Semaphore wieder frei. Die Komponente semundo
       enthält die dazu notwendigen Informationen. */

struct desc_struct *ldt;
    /* wurde speziell für den Windows-Emulator WINE eingeführt;
       bei ihm werden mehr Informationen und andere
       Funktionen zur Speicherverwaltung benötigt als für
       normale Linux-Programme */

struct thread_struct tss;
    /* Prozessorstatus beim letzten Wechsel vom Benutzermodus
       in den Systemmodus. Hier sind alle Prozessorregister
       enthalten, um diese bei der Rückkehr in Benutzermodus
       wiederherzustellen. Die Struktur thread_struct ist in
       <asm/processor.h> definiert. */

struct fs_struct *fs;
    /* enthält filesystemspezifische Informationen;
       Die Struktur fs_struct ist in <linux/sched.h> wie folgt
       definiert:
       struct fs_struct {
           int count; // Referenzzähler, da diese Struktur
                     // von mehreren Tasks benutzt
                     // werden kann.
           unsigned short umask; // Dateikreierungsmaske
                               // des Prozesses
           struct inode * root, // Root Directory
                               // des Prozesses
           * pwd; // Working Directory
                 // des Prozesses
       };

struct files_struct *files;
    /* Informationen zu den vom Prozeß geöffneten Dateien;
       Die Struktur files_struct ist in <linux/sched.h> wie

```

folgt definiert:

```
struct files_struct {
    int count; // Referenzzähler, da diese Struktur
              // von mehreren Tasks benutzt
              // werden kann.
    fd_set close_on_exec; // Bitmaske aller benutzt.
                        // Filedesktoren, die
                        // beim Systemruf exec
                        // zu schließen sind
    fd_set open_fds; // Bitmaske aller benutzter
                  // Filedesktoren
    struct file * fd[NR_OPEN]; // Index für dieses
                              // Array ist der
                              // entsprechende
                              // Filedesktor
};
```

struct mm_struct *mm;

/* Notwendige Daten zur Speicherverwaltung des Prozesses;
Die Struktur mm_struct ist in <linux/sched.h> wie folgt
definiert:

```
struct mm_struct {
    int count;
    pgd_t * pgd;
    unsigned long context;
    unsigned long start_code, end_code,
                  start_data, end_data;
    unsigned long start_brk, brk, start_stack,
                  start_mmap;
    unsigned long arg_start, arg_end,
                  env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    struct vm_area_struct * mmap;
    struct vm_area_struct * mmap_avl;
    struct semaphore mmap_sem;
};
```

Diese Struktur enthält unter anderem Informationen
über den Beginn und die Größe der Code- und Daten-
segmente für das gerade ablaufende Programm */

struct signal_struct *sig;

/* zeigt auf die Struktur signal_struct, die wie folgt in
<linux/sched.h> definiert ist:

```
struct signal_struct {
    int count;
    struct sigaction action[32];
};
```

Die Komponente action[32] gibt dabei für jedes Signal
an, wie der Prozeß auf das Eintreffen des jeweiligen
Signals reagieren soll; Index ist dabei die Nummer
des entsprechenden Signals */


```
#ifdef __SMP__
    int processor;
    int last_processor;
    int lock_depth;
#endif
    /* wird für Symmetric Multi Processing (SMP) benötigt;
       ist SMP aktiviert, muß der Systemkern für jede Task
       noch wissen, auf welchem Prozessor diese läuft.      */
};
```

Für jeden Prozeß, der gerade abläuft, befindet sich ein Eintrag in der sogenannten *Prozeß-tabelle*, die wie folgt in `<linux/sched.h>` deklariert ist:

```
extern struct task_struct *task[NR_TASKS];
```

Die Konstante `NR_TASKS` ist in `<linux/tasks.h>` wie folgt definiert:

```
#define NR_TASKS    512
```

Die einzelnen gerade ablaufenden Tasks sind dabei als doppelt verkettete Liste miteinander verbunden, in der man sich über die beiden Komponenten `next_task` und `prev_task` in der eben vorgestellten Struktur `task_struct` vorwärts und rückwärts bewegen kann. Die globale Variable `init_task`, die in `<linux/sched.h>` wie folgt deklariert ist, zeigt zugleich auf den Anfang und auf das Ende dieser Ringliste:

```
extern struct task_struct init_task;
```

Diese Variable wird beim Systemstart mit der Ur-Task `INIT_TASK` initialisiert. Nach dem Booten des Systems wird diese Ur-Task, die sich immer in `task[0]` befindet, eigentlich nicht mehr benötigt, weshalb sie dazu verwendet wird, nicht benötigte Systemzeit zu verbrauchen, also einen sogenannten *Idle-Prozeß* darzustellen. Dies ist auch der Grund, warum diese Task normalerweise beim Durchlaufen der einzelnen Tasks – was der Systemkern des öfteren tun muß – einfach übersprungen wird. Zum Durchlaufen aller Tasks wird das folgende in `<linux/sched.h>` definierte Makro verwendet:

```
#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

Auf die aktuell ablaufende Task läßt sich immer über das Makro `current` zugreifen, das inzwischen auch für Multiprozessoring (SMP) ausgelegt ist. Das Makro `current` ist in `<linux/sched.h>` über die folgenden Zeilen definiert:

```
extern struct task_struct *current_set[NR_CPUS];
/*
 * On a single processor system this comes out as current_set[0]
 * when cpp has finished with it, which gcc will optimise away.
 */
/* Current on this processor */
#define current (0+current_set[smp_processor_id()])
```

Das Warten von Prozessen auf das Eintreten von bestimmten Ereignissen – wie z.B. das Warten eines Elternprozesses auf das Ende eines Kindprozesses oder das Warten auf

Daten, die von der Festplatte gelesen werden – erfolgt in Linux mit Hilfe von Warteschlangen. Dabei ist eine Warteschlange nichts anderes als eine Ringliste, deren Element Zeiger in die Prozeßtafel sind. Die dazugehörige Struktur ist in `<linux/wait.h>` wie folgt definiert:

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

Um einen neuen Eintrag `wait` zu der Warteschlange `p` hinzuzufügen oder einen Eintrag `wait` aus der Warteschlange `p` zu entfernen, stehen die folgenden in `<linux/sched.h>` definierten Funktionen zur Verfügung:

```
extern inline void __add_wait_queue(struct wait_queue ** p,
                                   struct wait_queue * wait)
{
    struct wait_queue *head = *p;
    struct wait_queue *next = WAIT_QUEUE_HEAD(p);

    if (head)
        next = head;
    *p = wait;
    wait->next = next;
}

extern inline void add_wait_queue(struct wait_queue ** p,
                                  struct wait_queue * wait)
{
    unsigned long flags;

    save_flags(flags); /* aktuellen Prozessorstatus sichern */
    cli();             /* keine weiteren Interrupts zulassen */
    __add_wait_queue(p, wait);
    restore_flags(flags); /* ursprgl. Prozessorstatus wiederherstellen */
}

extern inline void __remove_wait_queue(struct wait_queue ** p,
                                       struct wait_queue * wait)
{
    struct wait_queue * next = wait->next;
    struct wait_queue * head = next;

    for (;;) {
        struct wait_queue * nextlist = head->next;
        if (nextlist == wait)
            break;
        head = nextlist;
    }
    head->next = next;
}
```

```
extern inline void remove_wait_queue(struct wait_queue ** p,
                                   struct wait_queue * wait)
{
    unsigned long flags;

    save_flags(flags); /* aktuellen Prozessorstatus sichern */
    cli();             /* keine weiteren Interrupts zulassen */
    __remove_wait_queue(p, wait);
    restore_flags(flags); /* ursprgl. Prozessorstatus wiederherstellen */
}
```

Ein Prozeß, der auf ein bestimmtes Ereignis warten will oder muß, trägt sich in die entsprechende Ereigniswarteschlange⁷ ein und gibt die Steuerung ab. Tritt das Ereignis ein, werden alle Prozesse in der betreffenden Warteschlange wieder aktiviert und können weiterarbeiten. Die Implementierung dazu sind die folgenden in `kernel/sched.c` definierten Funktionen:

```
static inline void __sleep_on(struct wait_queue **p, int state)
{
    unsigned long flags;
    struct wait_queue wait = { current, NULL };

    if (!p)
        return;
    if (current == task[0])
        panic("task[0] trying to sleep");
    current->state = state; /* setzt Status des Prozesses auf
                           state (TASK_INTERRUPTIBLE oder
                           TASK_UNINTERRUPTIBLE) */

    save_flags(flags);
    cli();             /* keine weiteren Interrupts zulassen */
    __add_wait_queue(p, &wait); /* trägt den Prozeß in die
                               Warteschlange ein */
    sti();             /* Weitere Interrupts wieder zulassen */
    schedule(); /* Prozeß gibt Steuerung an den Scheduler ab */
    cli();             /* keine weiteren Interrupts zulassen */
    __remove_wait_queue(p, &wait); /* entfernt Prozeß wieder
                                aus der Warteschlange */
    restore_flags(flags);
}

void interruptible_sleep_on(struct wait_queue **p)
{
    __sleep_on(p, TASK_INTERRUPTIBLE);
}

void sleep_on(struct wait_queue **p)
{
    __sleep_on(p, TASK_UNINTERRUPTIBLE);
}
```

7. Zu jedem möglichen Ereignistyp existiert eine eigene Warteschlange.

Ein Prozeß wird erst dann wieder aktiviert, wenn der Prozeßstatus sich in `TASK_RUNNING` ändert. Dies geschieht normalerweise dadurch, daß ein anderer Prozeß eine der beiden in `<linux/sched.h>` wie folgt deklarierten Funktionen aufruft:

```
extern void wake_up(struct wait_queue ** p);
extern void wake_up_interruptible(struct wait_queue ** p);
```

Diese beiden rufen ihrerseits die folgende, ebenfalls in `<linux/sched.h>` deklarierte Funktion auf:

```
extern void wake_up_process(struct task_struct * tsk);
```

Die Implementierungen zu diesen drei Funktionen befinden sich `kernel/sched.c`.

Zur Synchronisation von Zugriffen der Kernroutinen auf gemeinsam benutzte Datenstrukturen verwendet Linux sogenannte Semaphore, die nicht mit dem später in diesem Buch vorgestellten Semaphorkonzept (von Unix System V) auf Benutzerebene zu wechseln sind, sondern nur intern für die Kernsynchronisation benutzt werden.

Die dazu notwendige Struktur ist in `<asm/semaphore.h>` wie folgt definiert:

```
struct semaphore {
    int count;
    int waking;
    int lock ; /* to make waking testing atomic */
    struct wait_queue * wait;
};
```

Wenn `count` einen Wert kleiner oder gleich 0 hat, gilt das Semaphor als belegt. Ist das Semaphor belegt, tragen sich alle Prozesse, die das Semaphor ebenfalls belegen wollen, in eine Warteschlange ein. Wird das Semaphor von dem entsprechenden Prozeß freigegeben, werden die wartenden Prozesse benachrichtigt. Zum Belegen und Freigeben von Semaphoren werden die beiden folgenden Funktionen **down** und **up** angeboten:

```
extern inline void down(struct semaphore * sem);
extern inline void up(struct semaphore * sem);
```

down prüft, ob das Semaphor frei (größer 0) ist; wenn ja, erniedrigt diese Funktion das Semaphor (Komponente `count`). Ansonsten trägt sich der Prozeß in eine Warteschlange ein und wird blockiert, bis das Semaphor frei wird. **up** gibt das Semaphor wieder frei, indem es das Semaphor (Komponente `count`) um 1 inkrementiert und ein **wake_up** für die zum Semaphor gehörende Warteschlange ausführt.

Booten des Linux-Systems

Nachdem der LILO (**Linux Loader**) den Linux-Kern in den Speicher geladen hat, startet der Kern am Einsprungpunkt

```
start:
```

der sich im Assemblerprogramm `arch/i386/boot/setup.S` befindet. Nachdem in diesem Assemblerprogramm die Initialisierung der Hardware durchgeführt wurde und der Prozessor in den Protected Mode umgeschaltet wurde, wird mit folgender Assemblerzeile

```
jmp $ 0x1000 , KERNEL_CS
```

zur Startadresse des eigentlichen Systemkerns gesprungen. Diese Startadresse befindet sich bei der Marke

```
startup_32:
```

im Assemblerprogramm `arch/i386/kernel/head.S`. Dieses Programm ist für weitere Hardware-Initialisierungen zuständig, wie z.B. die Initialisierung der MMU für das Paging (an Marke `setup_paging`) oder die Initialisierung der Interruptdeskriptortabelle (an Marke `setup_idt`). Da zu diesem Zeitpunkt noch kein Programm-Environment (wie z.B. Stack, Umgebungsvariablen usw.) existiert, ist es auch die Aufgabe des Assemblerprogramms ein solches Environment einzurichten, wie es von den C-Kernroutinen, die nun zur Ausführung gebracht werden, benötigt wird.

Nachdem die erforderlichen Initialisierungen abgeschlossen sind, wird die erste C-Funktion **`start_kernel`** aufgerufen:

```
call _start_kernel
```

Die Funktion **`start_kernel`** ist in `init/main.c` wie folgt definiert:

```
asm__linkage void start_kernel(void)
{
    char * command_line;

#ifdef __SMP__
    static int first_cpu=1;

    if(!first_cpu)
        start_secondary();
    first_cpu=0;
#endif
    /*
     * Interrupts are still disabled. Do necessary setups, then
     * enable them
     */
    setup_arch(&command_line, &memory_start, &memory_end);
    memory_start = paging_init(memory_start, memory_end);
    trap_init();
    init_IRQ();
    sched_init();
    time_init();
    parse_options(command_line);
#ifdef CONFIG_MODULES
    init_modules();
#endif
}
```

```

#ifdef CONFIG_PROFILE
    if (!prof_shift)
#ifdef CONFIG_PROFILE_SHIFT
        prof_shift = CONFIG_PROFILE_SHIFT;
#else
        prof_shift = 2;
#endif
#endif
    if (prof_shift) {
        prof_buffer = (unsigned int *) memory_start;
        /* only text is profiled */
        prof_len = (unsigned long) &_etext - (unsigned long) &_stext;
        prof_len >>= prof_shift;
        memory_start += prof_len * sizeof(unsigned int);
        memset(prof_buffer, 0, prof_len * sizeof(unsigned int));
    }
    memory_start = console_init(memory_start, memory_end);
#ifdef CONFIG_PCI
    memory_start = pci_init(memory_start, memory_end);
#endif
    memory_start = kmalloc_init(memory_start, memory_end);
    sti();
    calibrate_delay();
    memory_start = inode_init(memory_start, memory_end);
    memory_start = file_table_init(memory_start, memory_end);
    memory_start = name_cache_init(memory_start, memory_end);
#ifdef CONFIG_BLK_DEV_INITRD
    if (initrd_start && initrd_start < memory_start) {
        printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
            "disabling it.\n", initrd_start, memory_start);
        initrd_start = 0;
    }
#endif
    mem_init(memory_start, memory_end);
    buffer_init();
    sock_init();
#ifdef CONFIG_SYSVIPC || defined(CONFIG_KERNELD)
    ipc_init();
#endif
    dquot_init();
    arch_syms_export();
    sti();
    check_bugs();

    printk(linux_banner);
#ifdef __SMP__
    smp_init();
#endif
    sysctl_init();
    /*
     * We count on the initial thread going ok
     * Like idlers init is an unlocked kernel thread, which will
     * make syscalls (and thus be locked).

```

```

    */
    kernel_thread(init, NULL, 0);
/*
 * task[0] is meant to be used as an "idle" task: it may not sleep, but
 * it might do some general things like count free pages or it could be
 * used to implement a reasonable LRU algorithm for the paging routines:
 * anything that can be useful, but shouldn't take time from the real
 * processes.
 *
 * Right now task[0] just does a infinite idle loop.
 */
    cpu_idle(NULL);
}

```

Nachdem zunächst mit der in `arch/i386/kernel/setup.c` definierten Funktion `setup_arch` alle von den vorherigen Assemblerprogramm ermittelten Daten gesichert wurden, werden alle Teile des Kerns initialisiert.

Der hier laufende Prozeß ist der Ur-Prozeß mit der Prozeß-ID 0. Mit dem Aufruf

```
kernel_thread(init, NULL, 0);
```

kreiert er schließlich einen Kern-Thread, der die Kernroutine ***init*** aufruft. Der Ur-Prozeß hat damit seine wichtigste Aufgabe erfüllt und übernimmt mit dem Aufruf

```
cpu_idle(NULL);
```

nun seine zweite Aufgabe: das Verbrauchen von nicht benötigter Rechenzeit. Die Funktion ***cpu_idle*** ist in `init/main.c` z.B. für den Fall, daß kein SMP stattfindet, wie folgt definiert:

```

int cpu_idle(void *unused)
{
    for(;;)
        idle();
}

```

Die hier aufgerufene Systemfunktion ***idle*** (eigentlicher Name ist ***sys_idle***) ist für Single- und Multiprozessorsysteme unterschiedlich in `arch/i386/kernel/process.c` definiert. Dieser Systemaufruf ***idle*** repräsentiert den Idle-Prozeß, von dem niemals zurückgekehrt wird.

Nun aber zurück zur ***init***-Funktion, die für die restliche Initialisierung zuständig ist, und von ***kernel_thread*** beim Aufruf

```
kernel_thread(init, NULL, 0);
```

aufgerufen wird. Die Funktion ***init*** ist in `init/main.c` definiert. Nachfolgend ein Auszug zu dieser Definition sowie der von zwei weiteren Routinen, die in ***init*** aufgerufen werden:

```

static int init(void * unused)
{
    int pid,i;
    .....

    /* Starten des Dämonprozesses bdf flush, der für die
       Synchronisation des Buffercaches mit dem Filesystem
       zuständig ist */
    kernel_thread(bdf flush, NULL, 0);

    /* Starten und Initialisieren des Dämonprozesses kswapd,
       der für das Swappen verantwortlich ist */
    kswapd_setup();
    kernel_thread(kswapd, NULL, 0);

    .....

    /* Die Aufgabe von setup ist das Initialisieren der
       Filesysteme und das Mounten des Root-Filesystems */
    setup();

    .....

    /* Nun wird versucht, eine Verbindung zur Konsole
       herzustellen und die Filedeskriptoren 0, 1 und 2
       zu öffnen */
    if ((open("/dev/tty1",O_RDWR,0) < 0) &&
        (open("/dev/ttyS0",O_RDWR,0) < 0))
        printk("Unable to open an initial console.\n");
    (void) dup(0);
    (void) dup(0);

    /* Nun wird versucht, eines der Programme /etc/init,
       /bin/init oder /sbin/init zu starten.
       Das entsprechende, zuerst gestartete Programm ist dann
       normalerweise der immer im Hintergrund laufende
       init-Prozeß mit der Prozeßnummer 1.
       Er wird oft auch als der Vater aller Prozesse bezeichnet,
       was unter Linux nicht ganz richtig ist, da dies
       eigentlich der Ur-Prozeß (nun Idle-Prozeß) mit der
       Prozeßnummer 0 ist.
       Die Aufgabe des init-Prozesses ist es nun unter anderem,
       die erforderlichen Dämonen zu starten und auf jedem
       angeschlossenen Terminal das getty-Programm ablaufen zu
       lassen, so daß neue Anmeldungen von Benutzern
       dort erkannt werden. */
    if (!execute_command) {
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);

        /* Sollte keiner dieser drei Aufrufe erfolgreich sein,
           wird versucht, zunächst die Datei /etc/rc abzuarbeiten

```



```

        und dann anschließend eine Shell zu starten (siehe
        unten bei XXX), um dem Superuser entsprechende
        Aktionen durchführen zu lassen, damit beim nächsten
        Booten des Systems einer der vorherigen drei Aufrufe
        erfolgreich ist.                                     */
pid = kernel_thread(do_rc, "/etc/rc", SIGCHLD);
if (pid>0)
    while (pid != wait(&i))
        /* nothing */;
}

while (1) { /* XXX*/
    pid = kernel_thread(do_shell,
        execute_command ? execute_command : "/bin/sh",
        SIGCHLD);
    if (pid < 0) {
        printf("Fork failed in init\n\r");
        continue;
    }
    while (1)
        if (pid == wait(&i))
            break;
    printf("\n\rchild %d died with code %04x\n\r",pid,i);
    sync();
}
return -1;
}

static int do_rc(void * rc)
{
    close(0);
    if (open(rc, O_RDONLY, 0))
        return -1;
    return execve("/bin/sh", argv_rc, envp_rc);
}

static int do_shell(void * shell)
{
    close(0);close(1);close(2);
    setsid();
    (void) open("/dev/tty1", O_RDWR, 0);
    (void) dup(0);
    (void) dup(0);
    return execve(shell, argv, envp);
}

```

Hier wurde nur ein Überblick über einige wichtige Aktionen gegeben, die beim Booten eines Systems ablaufen. Die Details sind natürlich komplexer, insbesondere wenn es um die Initialisierung der Hardware geht.

Hardware-Interrupts unter Linux

Interrupts werden vom Systemkern zur Kommunikation mit der Hardware benötigt. Hier wird ein kurzer Einblick über das Geschehen beim Aufruf eines Interrupts gegeben. Linux unterscheidet zwei Arten von Hardware-Interrupts: Langsame Interrupts (*slow interrupts*) und schnelle Interrupts (*fast interrupts*). Neben der Geschwindigkeit, die natürlich vom Umfang der durchzuführenden Aktionen abhängt, unterscheiden sich diese beiden Arten von Interrupts noch dadurch, daß während des Abarbeitens von langsamen Interrupts weitere Interrupts zugelassen sind, wogegen bei dem Abarbeiten von schnellen Interrupts alle anderen Interrupts gesperrt sind, außer die jeweilige Bearbeitungsroutine gibt diese explizit frei.

Beim Ablauf eines langsamen Interrupts werden üblicherweise folgende Aktionen durchgeführt:

```
IRQ(intr_nr, intr_controller, intr_mask) {

    SAVE_ALL    /* in <include/asm/irq.h> definiertes Makro zum
                  Sichern aller Prozessorregister                */

    ENTER_KERNEL /* in <include/asm/irq.h> definiertes Makro zur
                  Synchronisation der Prozessorzugriffe auf den
                  Kern (im Falle von symmetric multi processing)*/

    ACK(intr_controller, intr_mask) /* Bestätigen des Interrupt-
                                      Empfangs mit gleichzeitigem
                                      Sperren von Interrupts
                                      dieses Typs                */

    ++intr_count; /* Erhöhen der Verschachtelungstiefe der
                  Interrupts.                                */

    sti(); /* Weitere Interrupts wieder zulassen                */

    do_IRQ(intr_nr, regs); /* Aufruf des eigentlichen Interrupt-
                           handlers (in arch/i386/kernel/irq.c
                           definiert). Über die übergebenen
                           Register (regs) können einige
                           Interrupthandler - wenn dies nötig
                           ist - feststellen, ob der Interrupt
                           einen Benutzerprozeß oder den
                           Systemkern unterbrochen hat.        */

    cli(); /* Weitere Interrupts zunächst sperren */

    UNBLK(intr_controller, intr_mask) /* Interruptcontroller mit-
                                       teilen, daß nun wieder
                                       Interrupts dieses Typs
                                       akzeptiert werden.        */

    --intr_count; /* Interruptzähler wieder dekrementieren    */
}
```

```

ret_from_sys_call(); /* Diese Assembleroutine ist nach jedem
                        langsamen Interrupt und nach jedem
                        Systemaufruf für die hier nun durchzu-
                        führenden Aktionen verantwortlich.
                        Diese Routine, die nie zum Aufrufer
                        zurückkehrt, ist für das Wiederherstellen
                        der mit SAVE_ALL gesicherten Register
                        zuständig und führt das zur Beendigung
                        jeder Interrupt-Routine nötige iret aus.*/
}

```

Bei der Bearbeitung von schnellen Interrupts, die für kleine Aufgaben eingesetzt werden, werden alle anderen Interrupts gesperrt, außer die entsprechende Behandlungsroutine gibt diese explizit frei. Beim Ablauf eines schnellen Interrupts werden nun üblicherweise die folgenden Aktionen durchgeführt:

```

fast_IRQ(intr_nr, intr_controller, intr_mask) {

    SAVE MOST /* in <include/asm/irq.h> definiertes Makro zum
                Sichern der Prozessorregister, die von normalen
                C-Funktionen modifiziert werden können */

    ENTER_KERNEL /* in <include/asm/irq.h> definiertes Makro zur
                    Synchronisation der Prozessorzugriffe auf den
                    Kern (im Falle von symmetric multi processing)*/

    ACK(intr_controller, intr_mask) /* Bestätigen des Interrupt-
                                    Empfangs mit gleichzeitigem
                                    Sperren von Interrupts
                                    dieses Typs */

    ++intr_count; /* Erhöhen der Verschachtelungstiefe der
                    Interrupts. */

    /* Hier werden nicht wie bei den langsamen Interrupts mit
       sti() weitere Interrupts wieder zugelassen */

    do_fast_IRQ(intr_nr); /* Aufruf des eigentlichen Interrupt-
                           handlers (in arch/i386/kernel/irq.c
                           definiert). */

    UNBLK(intr_controller, intr_mask) /* Interruptcontroller mit-
                                       teilen, daß nun wieder
                                       Interrupts dieses Typs
                                       akzeptiert werden. */

    --intr_count; /* Interruptzähler wieder dekrementieren */

    LEAVE_KERNEL /* führt die nach jedem schnellen Interrupt
                  erforderlichen Aktionen (bei SMP) durch */
}

```

```

    RESTORE_MOST /* wie SAVE_MOST ist auch dieses Makro in
                  <include/asm/irq.h> definiert. Es stellt
                  die mit SAVE_MOST gesicherten Register
                  wieder her und führt das zur Beendigung
                  jeder Interrupt-Routine nötige iret aus.      */
}

```

Realisierung von Timerinterrupts unter Linux

In jedem Linux-System gibt es eine interne Uhr, die mit dem Start des Systems zu ticken beginnt. Ein Ticken entspricht dabei zehn Millisekunden, was bedeutet, daß diese Uhr in einer Sekunde hundertmal tickt. Bei jedem Ticken wird dabei ein sogenannter *Timerinterrupt* ausgelöst, der die entsprechende Zeit in der globalen Variable `jiffies`, die nur von ihm modifiziert werden kann, aktualisiert. Diese Variable ist in `kernel/sched.c` wie folgt definiert:

```
unsigned long volatile jiffies=0;
```

Neben dieser internen Zeit existiert noch die reale Zeit, die für den Anwender meist von größerem Interesse ist. Diese wird in der Variablen `xtime` gehalten, die ebenfalls vom Timerinterrupt ständig aktualisiert wird und in `kernel/sched.c` wie folgt definiert ist:

```
volatile struct timeval xtime;
```

Die Struktur `timeval` ist in `<linux/time.h>` wie folgt definiert:

```

struct timeval {
    int  tv_sec;    /* Sekunden      */
    int  tv_usec;   /* Mikrosekunden */
};

```

Die für Timerinterrupts zuständige Interruptroutine aktualisiert immer die Variable `jiffies` und kennzeichnet die sogenannte *Bottom-Half-Routine* (siehe weiter unten) als aktiv. Diese Routine, die eventuell erst später nach der Entgegennahme weiterer Interrupts durch das System von diesem aufgerufen wird, ist für die Restarbeiten zuständig. Durch diese Vorgehensweise kann es vorkommen, daß weitere Timerinterrupts ausgelöst werden, bevor die eigentliche Behandlungsroutinen aktiviert werden, weswegen in `kernel/sched.c` die folgenden beiden Variablen definiert sind.

```

static unsigned long lost_ticks = 0;
/* enthält die Anzahl der seit dem letzten Aufruf der
   Bottom-Half-Routine aufgetretenen Timerinterrupts      */

static unsigned long lost_ticks_system = 0;
/* enthält die Anzahl der seit dem letzten Aufruf der
   Bottom-Half-Routine aufgetretenen Timerinterrupts, bei
   deren Aufruf sich der Prozeß im Systemmodus befand      */

```

Ein Timerinterrupt inkrementiert diese beiden Variablen, um sie später in den *Bottom-Half-Routinen* auszuwerten. Die Timerinterrupt-Routine ist in `kernel/sched.c` z.B. wie folgt definiert:

```

void do_timer(struct pt_regs * regs)
{
    (*(unsigned long *)&jiffies)++;
    lost_ticks++;
    mark_bh(TIMER_BH);
    if (!user_mode(regs)) {
        lost_ticks_system++;
        .....
    }
    if (tq_timer)
        mark_bh(TQUEUE_BH);
}

```

Die ebenfalls in `kernel/sched.c` definierte *Bottom-Half*-Routine des Timerinterrupts hat das folgende Aussehen:

```

static void timer_bh(void)
{
    update_times();
    run_old_timers();
    run_timer_list();
}

```

Die Funktion ***update_times*** ist für das Aktualisieren der Zeiten zuständig und in `kernel/sched.c` wie folgt definiert:

```

static inline void update_times(void)
{
    unsigned long ticks;

    ticks = xchg(&lost_ticks, 0);

    if (ticks) {
        unsigned long system;

        system = xchg(&lost_ticks_system, 0);
        calc_load(ticks); /* berechnet die Systemauslastung */
        update_wall_time(ticks);
        update_process_times(ticks, system);
    }
}

```

xchg ist ein in `asm/system.h` definiertes Makro, das nicht zu unterbrechen ist. Es liest den Wert an der als erstes Argument angegebenen Adresse und liefert diesen als Rückgabewert. Bevor dieser Wert allerdings zurückgegeben wird, überschreibt es den alten Wert dieser Adresse mit dem als zweitem Argument angegebenen Wert. Da dieses Makro nicht unterbrochen werden kann, ist sichergestellt, daß eventuell neu ankommende Timerinterrupts während der Ausführung dieses Makros nicht verlorengehen, weil erst danach die entsprechende Variable (`lost_ticks` bzw. `lost_ticks_system`) inkrementiert wird.

Während **update_wall_time** (in kernel/sched.c definiert) für die Aktualisierung der realen Zeit in der Variablen xtime zuständig ist, ist die Funktion **update_process_times**, die ebenfalls in kernel/sched.c definiert ist, für die Aktualisierung der Zeiten des aktuellen Prozesses verantwortlich. Nachfolgend ist die Definition dieser Funktion für ein System mit einem Prozessor gezeigt:

```
static void update_process_times(unsigned long ticks,
                                unsigned long system)
{
    struct task_struct * p = current;
    unsigned long user = ticks - system;

    if (p->pid) {
        /* Aktualisierung der Komponente counter in der
           Struktur task_struct (siehe Seite #).
           Wird der Wert von counter kleiner als 0,
           so ist die Zeitscheibe des aktuellen Prozesses
           abgelaufen und es wird bei der nächsten
           Gelegenheit der Scheduler aktiviert
           (angezeigt durch need_resched=1).          */
        p->counter -= ticks;
        if (p->counter < 0) {
            p->counter = 0;
            need_resched = 1;
        }
        /* Priorität des Prozesses aktualisieren          */
        if (p->priority < DEF_PRIORITY)
            kstat.cpu_nice += user;
        else
            kstat.cpu_user += user;
        /* Systemzeit des Prozesses entsprechend anpassen */
        kstat.cpu_system += system;
    }
    update_one_process(p, ticks, user, system);
}
```

Die in dieser Funktion aufgerufene Funktion **update_one_process** ist ebenfalls in kernel/sched.c wie folgt definiert:

```
static void update_one_process(
    struct task_struct *p, unsigned long ticks,
    unsigned long user, unsigned long system)
{
    do_process_times(p, user, system);
    do_it_virt(p, user);
    do_it_prof(p, ticks);
}
```

Die hier aufgerufene Funktion **do_process_times** ist in kernel/sched.c wie folgt definiert:

```
static void do_process_times(
    struct task_struct *p,
    unsigned long user, unsigned long system)
```

```

{
    long psecs;

    p->utime += user;    /* wird für statische Zwecke */
    p->stime += system; /* benötigt */

    /* prüft, ob die mit der Systemfunktion
       setrlimit eingestellte maximale CPU-Zeit des
       Prozesses überschritten wurde.
       Wenn ja, wird der Prozeß mit dem Signal
       SIGXCPU darüber informiert und mit dem
       Signal SIGKILL abgebrochen. */
    psecs = (p->stime + p->utime) / HZ;
    if (psecs > p->rlim[RLIMIT_CPU].rlim_cur) {
        /* Send SIGXCPU every second.. */
        if (psecs * HZ == p->stime + p->utime)
            send_sig(SIGXCPU, p, 1);
        /* and SIGKILL when we go over max.. */
        if (psecs > p->rlim[RLIMIT_CPU].rlim_max)
            send_sig(SIGKILL, p, 1);
    }
}

```

Die beiden ebenfalls in **update_one_process** aufgerufenen Funktionen **do_it_virt** und **do_it_prof** sind für die Aktualisierung der Intervalltimer (virtuelle Zeitschaltuhren) zuständig, die mit der Funktion **setitimer** für den Prozeß durch den Benutzer eingerichtet wurden. Ist ein Intervalltimer abgelaufen, wird die Task durch ein entsprechendes Signal beendet. Diese beiden Funktionen sind in `kernel/sched.c` wie folgt definiert:

```

/* überprüft die Zeit, die der Prozeß aktiv ist,
   sich aber nicht im Systemmodus befindet.
   die entsprechende Zeitschaltuhr wurde mit
   setitimer(ITIMER_VIRTUAL, ...); eingerichtet */
static void do_it_virt(struct task_struct * p,
                      unsigned long ticks)
{
    unsigned long it_virt = p->it_virt_value;

    if (it_virt) {
        if (it_virt <= ticks) {
            it_virt = ticks + p->it_virt_incr;
            send_sig(SIGVTALRM, p, 1);
        }
        p->it_virt_value = it_virt - ticks;
    }
}

/* überprüft die gesamte Zeit, die der Prozeß läuft;
   Die entsprechende Zeitschaltuhr wurde mit
   setitimer(ITIMER_PROF, ...); eingerichtet.
   Zusammen mit dem vorherigen Timer (ITIMER_VIRTUAL)
   ermöglicht dies eine Unterscheidung zwischen der
   im Systemmodus und im Benutzermodus verbrachten Zeit */

```



```

        expires stehenden Zeit
        sortiert ist.                */
unsigned long expires; /* gibt Zeitpunkt an, an dem Funktion,
                        auf die die Komponente function
                        zeigt, mit dem Argument data
                        aufzurufen ist.                */
unsigned long data;    /* Argument für function                */
void (*function)(unsigned long); /* zeigt auf Funktion, die
                                zum Zeitpunkt expires
                                aufzurufen ist.                */
};

```

Zur Aktualisierung dieser Timer dient die Funktion ***run_timer_list***.

Realisierung des Scheduler unter Linux

Die Aufgabe des Schedulers ist die Zuteilung der CPU an die einzelnen Prozesse. Unter Linux werden verschiedene Schedulingstrategien (entsprechend dem POSIX-Standard 1003.4) angeboten. Die Festlegung der Schedulingstrategie erfolgt mit dem Systemaufruf ***sched_scheduler***, der seinerseits wieder die Funktion ***setscheduler*** aufruft. Beide Funktionen benötigen die folgende in `<linux/sched.h>` definierte Struktur und die ebenfalls dort definierten Konstante, die den Schedulingalgorithmus festlegen:

```

struct sched_param {
    int sched_priority;
};

/* Schedulingstrategien */
#define SCHED_OTHER 0
#define SCHED_FIFO 1
#define SCHED_RR 2

```

Diese Konstanten legen die folgenden Schedulingstrategien fest:

- ▶ **SCHED_OTHER**
Dies ist der klassische Unix-Schedulingalgorithmus. Jeder Echtzeitprozeß, der mit den folgenden Schedulingstrategien (**SCHED_FIFO** und **SCHED_RR**) arbeitet, hat nach POSIX 1003.4 eine höhere Priorität als ein Prozeß, der nach der Schedulingstrategie **SCHED_OTHER** behandelt wird. **SCHED_OTHER** ist die voreingestellte Schedulingstrategie für Prozesse unter Linux.
- ▶ **SCHED_FIFO**
Dies ist eine Echtzeitstrategie, bei der ein Prozeß so lange laufen kann, bis er die Steuerung freiwillig abgibt oder aber durch einen Prozeß mit höherer Realtime-Priorität verdrängt wird.
- ▶ **SCHED_RR**
Im Gegensatz zu **SCHED_FIFO** wird bei dieser Strategie ein Prozeß auch unterbrochen, wenn seine Zeitscheibe abgelaufen ist und es Prozesse mit derselben Echtzeitpriorität gibt. **RR** steht für *Round-Robin*.

Die beiden Echtzeitstrategien `SCHED_FIFO` und `SCHED_RR` garantieren nicht wie in wirklichen Echtzeitbetriebssystemen feste Reaktions- und Prozeßumschaltzeiten. Sie garantieren nur folgendes: Wenn ein Prozeß mit höherer Echtzeitpriorität (in Komponente `rt_priority` der Taskstruktur enthalten) auf der CPU ablaufen möchte, so werden alle Prozesse mit niedrigerer Priorität verdrängt.

Die beiden Funktionen ***sched_scheduler*** und ***setscheduler***, die zur Festlegung der Schedulingstrategie dienen, sind in `kernel/sched.c` definiert:

```
asm linkage int sys_sched_setscheduler(pid_t pid, int policy,
                                     struct sched_param *param)
{
    return setscheduler(pid, policy, param);
}

static int setscheduler(pid_t pid, int policy,
                       struct sched_param *param)
{
    int error;
    struct sched_param lp;
    struct task_struct *p;

    if (!param || pid < 0)
        return -EINVAL; /* ungültiges Argument param oder
                        oder ungültige Prozeß-ID */

    /* Folgende in mm/memory.c definierte Funktion prüft,
       ob ein Lesen an der Adresse param erlaubt ist */
    error = verify_area(VERIFY_READ, param,
                       sizeof(struct sched_param));
    if (error)
        return error;

    /* kopiert den Inhalt von param in die lokale Variable lp */
    memcpy_fromfs(&lp, param, sizeof(struct sched_param));

    /* Die in kernel/sched.c definierte Funktion
       find_process_by_pid sucht den Prozeß mit Prozeß-ID pid
       in der Task-Liste und liefert dessen Task-Struktur zurück. */
    p = find_process_by_pid(pid);
    if (!p)
        return -ESRCH; /* Prozeß mit Prozeß-Id pid konnte in
                       der Taskliste nicht gefunden werden. */

    if (policy < 0)
        policy = p->policy;
    else if (policy != SCHED_FIFO && policy != SCHED_RR &&
            policy != SCHED_OTHER)
        return -EINVAL; /* ungültige Schedulingstrategie */

    /* Erlaubte Prioritäten für SCHED_FIFO und SCHED_RR sind 1..99
       und für SCHED_OTHER ist nur 0 als Priorität erlaubt */
    if (lp.sched_priority < 0 || lp.sched_priority > 99)
        return -EINVAL; /* ungültige Priorität */
}
```

```

if ((policy == SCHED_OTHER) != (lp.sched_priority == 0))
    return -EINVAL; /* keine Priorität für SCHED_OTHER erlaubt */
if ((policy == SCHED_FIFO || policy == SCHED_RR) && !suser())
    return -EPERM; /* nur Superuser hat Rechte,
                    eine Realtime-Strategie festzulegen */
if ((current->euid != p->euid) && (current->euid != p->uid) &&
    !suser())
    return -EPERM; /* keine Rechte, um Strategie festzulegen */

p->policy = policy;
p->rt_priority = lp.sched_priority;
cli();
if (p->next_run)
    move_last_runqueue(p); /* siehe auch weiter unten */
sti();
need_resched = 1; /* Aufruf des Schedulers ist erforderlich */
return 0;
}

```

Mit der in **setscheduler** aufgerufenen Funktion **move_last_runqueue** (in `kernel/sched.c` definiert) wird die übergebene Task am Ende der Liste von ausführbaren Tasks angefügt:

```

static inline void move_last_runqueue(struct task_struct * p)
{
    struct task_struct *next = p->next_run;
    struct task_struct *prev = p->prev_run;

    /* Task p aus Liste entfernen */
    next->prev_run = prev; /* */
    prev->next_run = next;
    /* Task p am Ende (vor init_task) einfügen */
    p->next_run = &init_task;
    prev = init_task.prev_run;
    init_task.prev_run = p;
    p->prev_run = prev;
    prev->next_run = p;
}

```

Der Schedulingalgorithmus von Linux ist in der Funktion **schedule** (in `kernel/sched.c` definiert) implementiert. Diese Funktion **schedule** wird von bestimmten Systemfunktionen direkt oder aber durch die Funktion **sleep_on** indirekt aufgerufen. Daneben wird vor jeder Rückkehr aus einem Systemaufruf oder einem Interrupt von der Funktion **ret_from_sys_call** die Variable `need_resched` überprüft. Ist der Wert dieser Variablen ungleich 0, wird der Scheduler in diesem Fall auch aufgerufen. Da regelmäßig der Timerinterrupt aufgerufen und hierbei wenn notwendig die Variable `need_resched` gesetzt wird, ist sichergestellt, daß der Scheduler in regelmäßigen Abständen aufgerufen wird. Die nachfolgend gezeigte, etwas gekürzte Funktion **schedule** soll die prinzipiellen Schritte zeigen, die der Linux-Scheduler durchführt. Der Code für SMP (Symmetric Multi Processing) wurde hierbei aus Übersichtsgründen entfernt.

```

/* NOTE!! Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
asmlinkage void schedule(void)
{
    int c;
    struct task_struct * p;
    struct task_struct * prev, * next;
    unsigned long timeout = 0;
    int this_cpu=smp_processor_id();

    /* Wurde schedule während eines Interrupts (intr_count>0) */
    /* aufgerufen, beendet sich diese Funktion sofort wieder. */
    if (intr_count)
        goto scheduling_in_interrupt;

    /* Zuerst werden die Bottom-Halbs der Interruptroutinen
     * aufgerufen (zwecks besserer Performance nicht im
     * Interrupthandler, sondern hier durchgeführt). */
    if (bh_active & bh_mask) {
        intr_count = 1;
        do_bottom_half(); /* in kernel/softirq.c definiert */
        intr_count = 0;
    }

    /* Nun werden alle Routinen aufgerufen, die in der
     * Task-Queue für den Scheduler reserviert wurden
     * (zwecks besserer Performance nicht im
     * Interrupthandler, sondern hier durchgeführt). */
    run_task_queue(&rq_scheduler); /* in <linux/tqueue.h> definiert */

    need_resched = 0;
    prev = current; /* prev zeigt nun auf die gerade ablaufende Task,
                     * der momentan die CPU zugeteilt ist. */
    cli();

    /* Falls die aktuelle Task nach der Schedulingstrategie
     * SCHED_RR abgearbeitet wird und die Zeitscheibe für diese
     * Task abgelaufen ist, wird sie an letzter Stelle (hinter
     * allen auf CPU wartenden Tasks, die nach der Round-Robin-
     * Strategie bearbeitet werden) eingeordnet. */
    if (!prev->counter && prev->policy == SCHED_RR) {
        prev->counter = prev->priority;
        move_last_runqueue(prev);
    }
    switch (prev->state) {
        case TASK_INTERRUPTIBLE:
            if (prev->signal & ~prev->blocked)
                goto makerunnable;
            timeout = prev->timeout;
            if (timeout && (timeout <= jiffies)) {
                prev->timeout = 0;

```

```

        timeout = 0;
makerunnable:
    prev->state = TASK_RUNNING;
    break;
}
default:
    /* Falls schedule aufgerufen wurde, weil die
       aktuelle Task auf ein Ereignis warten muß,
       wird diese Task aus der Run-Queue entfernt.
       del_from_runqueue ist in kernel/sched.c definiert */
    del_from_runqueue(prev);
    case TASK_RUNNING:
}
p = init_task.next_run;
sti();

#define idle_task (&init_task)

/* Hier ist nun der eigentliche Scheduling-Algorithmus:
   Es wird die Task mit der höchsten Priorität in der Run-Queue
   gesucht. Realtime-Tasks haben dabei eine höhere Priorität
   als Tasks, die nach SCHED_OTHER abgearbeitet werden.
   Die Definition der Funktion goodness ist weiter unten gezeigt. */

c = -1000;
next = idle_task;
while (p != &init_task) {
    int weight = goodness(p, prev, this_cpu);
    if (weight > c)
        c = weight, next = p;
    p = p->next_run;
}

/* Ist c==0, existieren zwar lafbereite Tasks, aber
   deren dynamischen Prioritäten (Wert von counter) müssen
   neu berechnet werden. Dabei werden auch die counter-Werte
   aller anderen Tasks neu berechnet. */
if (!c) {
    for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
}

/* next zeigt in jedem Fall auf die zu aktivierende Task,
   eventuell auch auf idle_task, falls kein lauffähiger
   Prozeß gefunden wurde. Falls es sich bei der Task, der
   nun die CPU zusteht (next) um eine andere Task handelt
   als diejenige, die bisher die CPU benutzte (prev), wird
   der Task next (eventuell also auch der idle_task) die
   CPU zugeteilt. */
if (prev != next) {
    struct timer_list timer;

```

```

    kstat.context_swch++;
    if (timeout) {
        init_timer(&timer);
        timer.expires = timeout;
        timer.data = (unsigned long) prev;
        timer.function = process_timeout;
        add_timer(&timer);
    }
    get_mmu_context(next); /* CPU der Task next zuteilen */
    switch_to(prev,next);
    if (timeout)
        del_timer(&timer);
}
return;

scheduling_in_interrupt:
    printk("Aiee: scheduling in interrupt %p\n", /* Für Debugging */
        __builtin_return_address(0));
}

```

Die in kernel/sched.c definierte Funktion **goodness** hat das folgende Aussehen:

```

static inline int goodness(struct task_struct * p,
                          struct task_struct * prev, int this_cpu)
{
    int weight;

    /*
     * Realtime process, select the first one on the
     * runqueue (taking priorities within processes
     * into account).
     */
    if (p->policy != SCHED_OTHER)
        return 1000 + p->rt_priority;

    /*
     * Give the process a first-approximation goodness value
     * according to the number of clock-ticks it has left.
     *
     * Don't do any other calculations if the time slice is
     * over..
     */
    weight = p->counter;
    if (weight) {
        /* .. and a slight advantage to the current process */
        if (p == prev)
            weight += 1;
    }

    return weight;
}

```

Systemaufrufe unter Linux

Zu jedem Systemaufruf existiert in `<asm/unistd.h>` eine Konstante:

<code>#define __NR_setup</code>	0
<code>#define __NR_exit</code>	1
<code>#define __NR_fork</code>	2
<code>#define __NR_read</code>	3
<code>#define __NR_write</code>	4
<code>#define __NR_open</code>	5
<code>#define __NR_close</code>	6
<code>#define __NR_waitpid</code>	7
<code>#define __NR_creat</code>	8
<code>#define __NR_link</code>	9
<code>#define __NR_unlink</code>	10
<code>#define __NR_execve</code>	11
<code>#define __NR_chdir</code>	12
<code>#define __NR_time</code>	13
<code>#define __NR_mknod</code>	14
<code>#define __NR_chmod</code>	15
<code>#define __NR_chown</code>	16
<code>#define __NR_break</code>	17
<code>#define __NR_oldstat</code>	18
<code>#define __NR_lseek</code>	19
<code>#define __NR_getpid</code>	20
<code>#define __NR_mount</code>	21
<code>#define __NR_umount</code>	22
<code>#define __NR_setuid</code>	23
<code>#define __NR_getuid</code>	24
<code>#define __NR_stime</code>	25
<code>#define __NR_ptrace</code>	26
<code>#define __NR_alarm</code>	27
<code>#define __NR_oldfstat</code>	28
<code>#define __NR_pause</code>	29
<code>#define __NR_utime</code>	30
<code>#define __NR_stty</code>	31
<code>#define __NR_gtty</code>	32
<code>#define __NR_access</code>	33
<code>#define __NR_nice</code>	34
<code>#define __NR_ftime</code>	35
<code>#define __NR_sync</code>	36
<code>#define __NR_kill</code>	37
<code>#define __NR_rename</code>	38
<code>#define __NR_mkdir</code>	39
<code>#define __NR_rmdir</code>	40
<code>#define __NR_dup</code>	41
<code>#define __NR_pipe</code>	42
<code>#define __NR_times</code>	43
<code>#define __NR_prof</code>	44
<code>#define __NR_brk</code>	45
<code>#define __NR_setgid</code>	46
<code>#define __NR_getgid</code>	47
<code>#define __NR_signal</code>	48
<code>#define __NR_geteuid</code>	49

#define __NR_getegid	50
#define __NR_acct	51
#define __NR_phys	52
#define __NR_lock	53
#define __NR_ioctl	54
#define __NR_fcntl	55
#define __NR_mpx	56
#define __NR_setpgid	57
#define __NR_ulimit	58
#define __NR_oldolduname	59
#define __NR_umask	60
#define __NR_chroot	61
#define __NR_ustat	62
#define __NR_dup2	63
#define __NR_getppid	64
#define __NR_getpgrp	65
#define __NR_setsid	66
#define __NR_sigaction	67
#define __NR_sgetmask	68
#define __NR_ssetmask	69
#define __NR_setreuid	70
#define __NR_setregid	71
#define __NR_sigsuspend	72
#define __NR_sigpending	73
#define __NR_sethostname	74
#define __NR_setrlimit	75
#define __NR_getrlimit	76
#define __NR_getrusage	77
#define __NR_gettimeofday	78
#define __NR_settimeofday	79
#define __NR_getgroups	80
#define __NR_setgroups	81
#define __NR_select	82
#define __NR_symlink	83
#define __NR_oldlstat	84
#define __NR_readlink	85
#define __NR_uselib	86
#define __NR_swapon	87
#define __NR_reboot	88
#define __NR_readdir	89
#define __NR_mmap	90
#define __NR_munmap	91
#define __NR_truncate	92
#define __NR_ftruncate	93
#define __NR_fchmod	94
#define __NR_fchown	95
#define __NR_getpriority	96
#define __NR_setpriority	97
#define __NR_profil	98
#define __NR_statfs	99
#define __NR_fstatfs	100
#define __NR_ioperm	101
#define __NR_socketcall	102

#define __NR_syslog	103
#define __NR_setitimer	104
#define __NR_getitimer	105
#define __NR_stat	106
#define __NR_lstat	107
#define __NR_fstat	108
#define __NR_olduname	109
#define __NR_iopl	110
#define __NR_vhangup	111
#define __NR_idle	112
#define __NR_vm86	113
#define __NR_wait4	114
#define __NR_swapoff	115
#define __NR_sysinfo	116
#define __NR_ipc	117
#define __NR_fsync	118
#define __NR_sigreturn	119
#define __NR_clone	120
#define __NR_setdomainname	121
#define __NR_uname	122
#define __NR_modify_ldt	123
#define __NR_adjtimex	124
#define __NR_mprotect	125
#define __NR_sigprocmask	126
#define __NR_create_module	127
#define __NR_init_module	128
#define __NR_delete_module	129
#define __NR_get_kernel_syms	130
#define __NR_quotactl	131
#define __NR_getpgid	132
#define __NR_fchdir	133
#define __NR_bdflush	134
#define __NR_sysfs	135
#define __NR_personality	136
#define __NR_afs_syscall	137 /* Andrew File System */
#define __NR_setfsuid	138
#define __NR_setfsgid	139
#define __NR_llseek	140
#define __NR_getdents	141
#define __NR_newselect	142
#define __NR_flock	143
#define __NR_msync	144
#define __NR_readv	145
#define __NR_writev	146
#define __NR_getsid	147
#define __NR_fdatasync	148
#define __NR__sysctl	149
#define __NR_mlock	150
#define __NR_munlock	151
#define __NR_mlockall	152
#define __NR_munlockall	153
#define __NR_sched_setparam	154
#define __NR_sched_getparam	155

```
#define __NR_sched_setscheduler      156
#define __NR_sched_getscheduler      157
#define __NR_sched_yield             158
#define __NR_sched_get_priority_max  159
#define __NR_sched_get_priority_min  160
#define __NR_sched_rr_get_interval   161
#define __NR_nanosleep               162
#define __NR_mremap                  163
```

Implementiert man nun einen neuen Systemaufruf, wie z.B. **sys_rmtree**, muß man diesen in dieser Liste mit der nächsten freien Nummer hinzufügen:

```
#define __NR_rmtree                  164
```

Zudem enthält die Datei `arch/i386/kernel/entry.S` die zugehörige initialisierte Tabelle von Systemaufrufen:

```
.data
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_setup)           /* 0 */
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    .long SYMBOL_NAME(sys_open)           /* 5 */
    .long SYMBOL_NAME(sys_close)
    .long SYMBOL_NAME(sys_waitpid)
    .long SYMBOL_NAME(sys_creat)
    .long SYMBOL_NAME(sys_link)
    .long SYMBOL_NAME(sys_unlink)         /* 10 */
    .long SYMBOL_NAME(sys_execve)
    .....
    .....
    .long SYMBOL_NAME(sys_sched_get_priority_max)
    .long SYMBOL_NAME(sys_sched_get_priority_min) /* 160 */
    .long SYMBOL_NAME(sys_sched_rr_get_interval)
    .long SYMBOL_NAME(sys_nanosleep)
    .long SYMBOL_NAME(sys_mremap)
    .space (NR_syscalls-163)*4
```

Hier muß nun an der Position 164 ein Zeiger auf die Funktion, die den neuen Systemaufruf behandelt, eingefügt und die letzte Zeile entsprechend angepaßt werden:

```
    .long SYMBOL_NAME(sys_sched_get_priority_max)
    .long SYMBOL_NAME(sys_sched_get_priority_min) /* 160 */
    .long SYMBOL_NAME(sys_sched_rr_get_interval)
    .long SYMBOL_NAME(sys_nanosleep)
    .long SYMBOL_NAME(sys_mremap)
    .long SYMBOL_NAME(sys_rmtree)
    .space (NR_syscalls-164)*4
```

Das Makro `SYMBOL_NAME` ist im übrigen in `<linux/linkage.h>` wie folgt definiert:

```
#define SYMBOL_NAME(X) X
```

Das zu diesem neuen Systemaufruf gehörige Quellprogramm sollte man in der Datei `kernel/rmtree.c` speichern. Es ist ratsam, jeden neuen Systemaufruf in einer eigenen Datei zu speichern, da so eine Portierung auf eine neuere Kern-Version erheblich erleichtert wird. Nun muß noch in der Datei `kernel/Makefile` der folgende Eintrag:

```
0_OBJS = sched.o dma.o fork.o exec_domain.o panic.o printk.o sys.o \
        module.o exit.o signal.o itimer.o info.o time.o softirq.o \
        resource.o sysctl.o
```

um `rmtree.o` erweitert werden:

```
0_OBJS = sched.o dma.o fork.o exec_domain.o panic.o printk.o sys.o \
        module.o exit.o signal.o itimer.o info.o time.o softirq.o \
        resource.o sysctl.o rmtree.o
```

Jetzt kann ein neuer Kernel generiert und installiert werden (siehe Seite # und #). Um dem Benutzer eine Bibliotheksfunktion mit dem Namen **`rmtree`** (und nicht nur **`sys_rmtree`**) zur Verfügung zu stellen, empfiehlt es sich, das folgende C-Programm zu schreiben:

```
#include <linux/unistd.h>

_syscall1(int, rmtree, char *, pathname)
```

Kompiliert man dieses Programm, so wird der Aufruf des Makros **`_syscall1`** (in `<asm/unistd.h>` definiert) wie folgt expandiert:

```
int rmtree(char * pathname)
{
    long __res;
    __asm__ volatile ("int $0x80"
                     : "=a" (__res)
                     : "0" (__NR_rmtree), "b" ((long)(pathname)));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}
```

Die so erzeugte Objektdatei kann man nun mit dem Kommando **`ar`** in der C-Standardbibliothek `/usr/lib/libc.a` hinzufügen, damit Benutzer den neuen Systemaufruf **`rmtree`** verwenden können.

Wird ein Systemaufruf von einem Benutzer aufgerufen, gilt allgemein, daß dieser seine Argumente und die Nummer des Systemaufrufs in definierte Übergaberegister schreibt und anschließend den Interrupt 0x80 auslöst. Bei Rückkehr der zugehörigen Interruptserviceroutine wird der Rückgabewert aus dem entsprechenden Übergaberegister gelesen und der Systemaufruf ist beendet.

Die eigentliche Arbeit bei Systemaufrufen wird also von der Interruptroutine durchgeführt. Diese Interruptroutine, die sich in `arch/i386/kernel/entry.S` befindet, ist in Assembler geschrieben und beginnt ihre Arbeit am Einsprungpunkt:

```
ENTRY(system_call)
```

Der Einsprungpunkt wird für alle Systemaufrufe verwendet. Der dort angegebene Assemblercode ist unter anderem für folgendes zuständig:

- ▶ Sichern aller Register (mit dem Makro `SAVE_ALL` in `entry.S`)
- ▶ Überprüfung, ob es sich um einen erlaubten Systemaufruf handelt
- ▶ Ausführung des zu diesem Systemaufruf gehörenden Codes. Zum Auffinden dieses Codes wird die bei `entry(sys_call_table)` angegebene Nummer (siehe auch oben) verwendet.
- ▶ Nach der Beendigung des Systemaufruf-Codes muß an den Einsprungpunkt `ret_from_sys_call`: gesprungen werden. Dort wird noch geprüft, ob eventuell der Scheduler aufzurufen ist, was sich an dem Inhalt der Variablen `need_sched` erkennen läßt.
- ▶ Wiederherstellen aller Register (mit dem Makro `RESTOR_ALL` in `entry.S`)

Die Makros **`_syscallnr`** sind in `<asm/unistd.h>` definiert, wobei die Nummer *nr* angibt, wie viele Parameter die entsprechende Systemfunktion hat:

```
/* XXX - _foo needs to be __foo, while __NR_bar could be _NR_bar. */
#define _syscall0(type,name) \
type name(void) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}

#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

```

#define __syscall12(type,name,type1,arg1,type2,arg2) \
type name(type1 arg1,type2 arg2) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}

#define __syscall13(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
          "d" ((long)(arg3))); \
    if (__res>=0) \
        return (type) __res; \
    errno=-__res; \
    return -1; \
}

#define __syscall14(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
          "d" ((long)(arg3)),"S" ((long)(arg4))); \
    if (__res>=0) \
        return (type) __res; \
    errno=-__res; \
    return -1; \
}

#define __syscall15(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
    type5,arg5) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5))); \
    if (__res>=0) \
        return (type) __res; \
}

```

```
errno=-__res; \  
return -1; \  
}
```

Die Realisierungen der einzelnen Linux-Systemaufrufe befinden sich in den jeweiligen Subdirectories von `/usr/src/linux` und können dort nachgeschlagen werden. Teilweise lassen sich solche Systemaufrufe sehr einfach realisieren, wie der folgende Ausschnitt aus `kernel/sched.c` zeigt:

```
asmlinkage int sys_getpid(void)  
{  
    return current->pid;  
}  
  
asmlinkage int sys_getppid(void)  
{  
    return current->p_opptr->pid;  
}  
  
asmlinkage int sys_getuid(void)  
{  
    return current->uid;  
}  
asmlinkage int sys_geteuid(void)  
{  
    return current->euid;  
}  
  
asmlinkage int sys_getgid(void)  
{  
    return current->gid;  
}  
  
asmlinkage int sys_getegid(void)  
{  
    return current->egid;  
}
```

Andere Systemaufrufe dagegen sind komplexer. Es würde den Rahmen dieses Buches sprengen, alle Systemaufrufe von Linux näher zu erläutern. Hier sollte nur ein Einblick in den Systemkern von Linux gegeben werden. An entsprechenden Stellen wird noch genauer auf wichtige Konzepte des Linux-Kerns eingegangen.



1.13 Übung

1.13.1 Primitive Systemdatentypen am aktuellen System

Erstellen Sie ein Programm `primtyp.c`, das Ihnen zu den auf Ihrem System vorhandenen Systemdatentypen die Anzahl der Bytes ausgibt, die sie jeweils belegen. Ermitteln Sie dazu alle benötigten Headerdateien, in denen diese eventuell definiert sind, wenn die entsprechende Definition für einen Datentyp in `<sys/types.h>` auf ihrem System fehlt.

Nachdem man das Programm `primtyp.c` kompiliert und gelinkt hat

```
cc -o primtyp primtyp.c
```

kann sich z.B. der folgende Ablauf ergeben:

```
$ primtyp
  caddr_t :   4 Bytes
  clock_t :   4 Bytes
   dev_t :   4 Bytes
  fd_set : 128 Bytes
  fpos_t :   4 Bytes
   gid_t :   4 Bytes
   ino_t :   4 Bytes
  mode_t :   4 Bytes
  nlink_t :   4 Bytes
   off_t :   4 Bytes
   pid_t :   4 Bytes
 ptrdiff_t :   4 Bytes
   rlim_t :   4 Bytes
sig_atomic_t :   4 Bytes
  sigset_t :  16 Bytes
   size_t :   4 Bytes
  ssize_t :   4 Bytes
   time_t :   4 Bytes
   uid_t :   4 Bytes
  wchar_t :   4 Bytes

$
```


2

Überblick über ANSI C

*Die Gewalt einer Sprache ist nicht,
daß sie das Fremde abweist,
sondern daß sie es verschlingt.*

Goethe

Zur Programmierung des Unix-Systems verwendet man die Sprache C. Diese Sprache wurde im Jahr 1989 durch ein ANSI-Komitee standardisiert. Der dabei geschaffene Standard wird allgemein mit *ANSI C* bezeichnet.

In diesem Kapitel wird ein Überblick über ANSI C gegeben. Dabei werden zunächst Begriffe und allgemein geltende Konventionen vorgestellt, bevor detaillierter auf den Präprozessor und die Sprache ANSI C selbst eingegangen wird. Zum Abschluß dieses Kapitels wird ein Überblick über die nun standardisierten Headerdateien gegeben. Dabei werden alle von ANSI C vorgeschriebenen Konstanten, Datentypen, Makros, globale Variablen und Funktionen, soweit sie nicht in späteren Kapiteln ausführlich beschrieben werden, kurz vorgestellt.

2.1 Allgemeines

Das ANSI¹-Komitee X3J11 begann im Juni 1983 mit dem Vorhaben, die Sprache C zu standardisieren. Vorher galt die erste Ausgabe des Buches »The C Programming Language« von Kernighan und Ritchie (Prentice-Hall, 1978) als die Bibel für alle C-Fragen. Es ließ jedoch einige Fragen offen. So wurde bereits in den frühen achtziger Jahren die Notwendigkeit für einen wirklichen C-Standard erkannt.

Es sollten nun Standardvorgaben für alle möglichen C-Aspekte geschaffen werden. Bei dieser Untersuchung haben sich drei unterschiedliche Schwerpunkte herausgebildet, für die es galt, eine Standardisierung zu finden:

- ▶ Sprache
- ▶ Präprozessor
- ▶ Bibliothek

1. ANSI (American National Standards Institute) ist eine amerikanische Organisation, die ein Mitglied der International Standards Organisation (ISO) ist. 1985 entschied das Komitee X3J11, daß nur ein C-Standard geschaffen werden soll, der von beiden Organisationen ANSI und ISO verabschiedet wurde.

Mit der Einführung von ANSI C können nun portable C-Programme geschrieben werden. ANSI C kümmerte sich nicht nur um die *Portabilität von C-Programmen*, sondern hat auch einige Neuheiten in C einfließen lassen, wobei wohl die *Funktionsprototypen* die wichtigste Neuheit sind. Funktionsprototypen wurden von der Weiterentwicklung von C, der Sprache C++, übernommen. Dieses Kapitel stellt die wichtigsten Begriffe und Konventionen von ANSI C vor.

2.1.1 Begriffsklärung

Implementierung

Eine *Implementierung* ist ein bestimmtes Softwarepaket, das C-Programme übersetzt (kompiliert) und für ein bestimmtes Betriebssystem lauffähig macht. Beispiele für Implementierungen sind:

- ▶ GNU C Compiler für Unix
- ▶ Borland C für MSDOS
- ▶ Microsoft C für MSDOS

Objekt

Ein *Objekt* ist ein Speicherbereich, der Daten aufnehmen kann. Außer für Bitfelder sind Objekte aus einer zusammenhängenden² Folge von einem oder mehreren Bytes³ zusammengesetzt. Ein Beispiel für ein Objekt ist eine **float**-Variable.

Argument

Der Begriff *Argument* steht für die altbekannten Begriffe »aktuelles Argument« oder »aktueller Parameter«. In ANSI C werden Parameter, die beim Aufruf einer Funktion oder eines Makros angegeben werden, *Argumente* genannt.

Parameter

Der Begriff *Parameter* steht für die altbekannten Begriffe »formales Argument« oder »formaler Parameter«. ANSI C spricht beim Funktionsaufruf von Argumenten und bei Funktionsdeklarationen oder -definitionen von *Parametern*.

2. Die Betonung liegt hier auf zusammenhängend. Somit kann ein Objekt wie ein Array von char-Elementen betrachtet werden, was zur Folge hat, daß seine Größe mit dem sizeof-Operator bestimmt werden kann.

3. Für ein Byte schreibt ANSI C vor, daß es mindestens 8 Bit »breit« ist und daß der Datentyp **char** (vorzeichenbehaftet oder nicht) genau ein Byte belegt.

Unspezifiziertes Verhalten

Dies ist das Verhalten einer korrekten C-Konstruktion, für die ANSI C keine Vorschriften macht. Ein Beispiel dafür ist die Reihenfolge, in der Funktionsargumente ausgewertet werden. Wenn beispielsweise eine Funktion zwei **int**-Parameter besitzt, dann ist für das folgende Programmstück

```
a = 100;
funktion(a*=2, a+=500);
```

nicht festgelegt, ob *funktion* mit (200,700) oder (1200,600) aufgerufen wird.

Undefiniertes Verhalten

Es bezeichnet das Verhalten bei Angabe von fehlerhaften oder nicht ANSI C konformen Sprachkonstruktionen, für was ANSI C keine Vorschriften macht. Wenn undefiniertes Verhalten vorliegt, so ist ein C-Compiler nicht verpflichtet, es zu erkennen und zu melden⁴. Beispiele für undefiniertes Verhalten sind:

- ▶ Eine arithmetische Operation, die zu einer *Division durch 0* führt.
- ▶ Betrag eines Wertes wird während einer Berechnung größer als der maximale Betrag, den der dafür vorgesehene Speicherbereich aufnehmen kann (*Overflow* = Überlauf).

Implementierungsdefiniertes Verhalten

Dies ist das Verhalten einer korrekten C-Konstruktion, die von der Auslegung durch die entsprechende C-Realisierung (Compiler) abhängt. ANSI C schreibt für jedes implementierungsdefinierte Verhalten vor, daß es in der begleitenden Compiler-Beschreibung dokumentiert sein muß. Ein Beispiel hierfür ist das Verhalten bei der Anwendung der Bit-Schiebeoperation `>>` auf negative **int**-Werte. Hierbei ergeben sich zwei Möglichkeiten:

- ▶ linkes Nachziehen von Nullen (*logical shift*)
- ▶ linkes Nachziehen von Einsen (*arithmetic shift*)

Lokalspezifisches Verhalten

Dies ist das Verhalten, das von lokalen Eigenheiten (wie Nationalität, Kultur oder Sprache) abhängig ist. Ein Beispiel hierfür ist das Verhalten der Bibliotheksroutine *isupper*⁵, wenn diese auf Umlaute wie ä oder ü angewendet wird.

4. Wäre aber nett, wenn er es trotzdem tun würde.

5. Überprüft, ob es sich bei einem Zeichen um einen Großbuchstaben im anglo-amerikanischen Alphabet handelt.

2.1.2 Trigraphs

Andere Länder, andere Zeichen: So ist z.B. den Franzosen das ö aus der deutschen Sprache nicht bekannt. C wurde in den USA entwickelt und setzt den amerikanischen Zeichensatz voraus. ANSI C nun möchte sich gerne eine »Weltsprache« nennen. Damit alle Nicht-Amerikaner ebenso die Möglichkeit haben, den von C vorgegebenen Grundzeichensatz darstellen zu können, wurden die Trigraphs (siehe Tabelle 2.1) eingeführt:

Trigraph	Repräsentiertes Zeichen
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	
??>	}
??-	~

Tabelle 2.1: Trigraphs in ANSI C

Trigraphs sind 3-Zeichen-Sequenzen, die mit ?? beginnen. Trigraphs werden vom Compiler durch das entsprechende »repräsentierte Zeichen« ersetzt.

Es ist anzumerken, daß Trigraphs sogar innerhalb von Zeichenketten (Strings) durch ihr »repräsentiertes Zeichen« ersetzt werden, wie das nachfolgende Beispiel verdeutlicht:

```
printf("Was ist 3 * 4 ???/n");
printf("3 * 4 = ??=12, oder nicht ???");
```

wird als

```
printf("Was ist 3 * 4 ?\n");
printf("3 * 4 = #12, oder nicht ???");
```

interpretiert.

2.1.3 Allgemeine Konventionen

Namen, die mit Unterstrich (_) beginnen

Namen, die mit Unterstrich beginnen, sind für den Gebrauch in Bibliotheken reserviert und sollten nicht vom Benutzer verwendet werden. Eigentlich legt ANSI C diese Restriktion nur für globale Namen fest. Für andere vom Benutzer gewählte Namen gilt nur die Einschränkung, daß sie nicht mit __ oder _G (G steht für Großbuchstabe) beginnen sollten.

Minimal garantierte Größe für die unterschiedlichen Typen

char >= 8 Bits
short >= 16 Bits
int >= short
long >= 32 Bits

Vielbyte-Zeichen

Manche Sprachen benötigen mehr als 1 Byte, um ein Zeichen zu speichern. Solche Vielbyte-Zeichen sind in ANSI C erlaubt. Es wurde sogar ein eigener Datentyp `wchar_t` eingeführt, um Vielbyte-Zeichen aufzunehmen

Erweiterung der nichtdruckbaren Zeichen

ANSI C hat die Menge der »Fluchtsymbol«-Sequenzen (Folge von Zeichen, die mit *Backslash* starten) erweitert. Diese Fluchtsymbolsequenzen erlauben es, nichtdruckbare Zeichen (wie z.B. den Piepston `\a`) in Zeichenketten unterzubringen.

Tabelle 2.2 zeigt eine Zusammenfassung dieser ANSI-C-Fluchtsymbole.

Fluchtsymbol	Bedeutung
<code>\a</code>	(alert) akustisches oder visuelles Aufmerksamkeitssignal . (neu in ANSI C) (meist die Klingel); aktive Position ⁶ wird in diesem Fall nicht verändert.
<code>\b</code>	(backspace) Zurücksetzzeichen versetzt die aktive Position auf die vorherige Position in entsprechender Zeile. Wenn sich die aktive Position bereits am Zeilenanfang befand, dann liegt »unspezifiziertes Verhalten« vor.
<code>\f</code>	(form feed) Seitenvorschub versetzt die aktive Position auf den Anfang der nächsten Seite.
<code>\n</code>	(new line) Neue Zeile versetzt die aktive Position auf den Anfang der nächsten Zeile.
<code>\r</code>	(carriage return) Wagenrücklauf versetzt die aktive Position auf den Anfang der momentanen Zeile.
<code>\t</code>	(horizontal tab) Horizontales Tabulatorzeichen versetzt die aktive Position zur nächsten horizontalen Tabulatorposition in der momentanen Zeile. Falls sich die aktive Position bereits an der letzten horizontalen Tabulatorposition oder dahinter befindet, dann liegt »unspezifiziertes Verhalten« vor.
<code>\v</code>	(vertical tab) Vertikales Tabulatorzeichen (neu in ANSI C) versetzt die aktive Position zur nächsten vertikalen Tabulatorposition. Falls sich die aktive Position bereits an der letzten vertikalen Tabulatorposition oder dahinter befindet, dann liegt »unspezifiziertes Verhalten« vor.

Tabelle 2.2: »Fluchtsymbolsequenzen« in ANSI C

6. Die aktive Position ist die Stelle auf einem Aufzeichnungsgerät (z.B. Cursor auf dem Bildschirm), wo die nächste Ausgabe eines Zeichens erfolgen würde.

2.2 Der Präprozessor

Während im ursprünglichen C von Kernighan und Ritchie die Funktionsweise des Präprozessors am ungenauesten vom ganzen C-Sprachumfang beschrieben war, hat das ANSI-C-Komitee um so mehr Aufwand betrieben, die Rolle des Präprozessors genau festzulegen.

Der Präprozessor verarbeitet den Quelltext einer Programmdatei, wobei alle Präprozessorkommandos (Präprozessordirektiven) mit dem Zeichen **#** beginnen. Zwischenraumzeichen (whitespace: *Leerzeichen*, **\f**, **\n**, **\r**, **\t** oder **\v**) sind vor **#** zugelassen. Zwischen **#** und Anfang der restlichen Präprozessordirektive sind nur *Leerzeichen* oder **\t** zugelassen.

Üblicherweise ruft der Compiler automatisch den Präprozessor auf, bevor er mit der Übersetzung beginnt. ANSI C schreibt vor, daß der Präprozessor wie ein eigener Schritt vor dem eigentlichen Compilerlauf zu verstehen ist. Das heißt nicht, daß der Präprozessorlauf als eigener Durchgang (wie es in heutigen Compilern oft der Fall ist) realisiert sein muß, sondern sich nur so verhalten muß.

Der Präprozessor bietet die folgenden Leistungen an:

- ▶ **#define** (Ersetzen von Zeichenketten, Funktionsmakros, ...)
- ▶ **#include** (Einkopieren ganzer Dateien)
- ▶ Bedingte Kompilierung
- ▶ Restliche Präprozessordirektiven
- ▶ Von ANSI C vordefinierte Makros

2.2.1 #define – Definieren von Konstanten und Makros

Textersatz- und Funktion-Makros (Alt-C)

Meist wird **#define** verwendet, um die Lesbarkeit eines Programms zu erhöhen:

```
#define MEHRWERT_STEUER 0.15           /*Textersatz-Makro*/
#define MAXIMUM(a,b) ((a) > (b) ? (a) : (b)) /*Funktion-Makro */
```

Anweisungen wie

```
end_betrag = betrag + betrag * MEHRWERT_STEUER;
max = MAXIMUM(zahl1,zahl2);
```

werden vom Präprozessor durch

```
end_betrag = betrag + betrag * 0.15;
max = ((zahl1) > (zahl2) ? (zahl1) : (zahl2));
```

ersetzt.

Konkatenation von hintereinander angegebenen Zeichenketten

ANSI C legt fest, daß hintereinander angegebene Zeichenketten (Leer-, Tabulator- und Neuezeilezeichen dazwischen zählen nicht) zu einer Zeichenkette zusammengefaßt werden.

Beispiel

```
char adresse[100] = "Sascha " "Kimmel, "
                  "Lohestr. 10, "
                  "97535 Gressthal";
```

wird umgewandelt nach

```
char adresse[100]="Sascha Kimmel, Lohestr. 10, 97535 Gressthal";
```

Beispiel

```
#define geschichte(jahr,ereignis) \
    printf("Im Jahre " jahr " war " ereignis"\n");
```

Ein Aufruf

```
geschichte("1492", "Entdeckung Amerikas durch Kolumbus");
```

wird vom Präprozessor zunächst in

```
printf("Im Jahre " "1492" " war " "Entdeckung Amerikas durch Kolumbus"\n");
```

umgewandelt und dann wird die Zeichenketten-Konkatenation angewendet, was zu folgender Darstellung führt:

```
printf("Im Jahre 1492 war Entdeckung Amerikas durch Kolumbus\n");
```

Ersetzung von Makroparametern durch Zeichenketten-Konstanten (Operator #)

Oft ist es nützlich, wenn man den Wert von Variablen zu Testzwecken in bestimmten Programmphasen ausgibt. Für einen solchen Anwendungsfall eignet sich das folgende Makro:

```
#define wertvon(variable) printf("variable=%d\n", variable)
```

Ein späterer Aufruf `wertvon(steuer);` kann nun vom Präprozessor durch

```
(a) printf("variable=%d\n",steuer);      oder
(b) printf("steuer=%d\n",steuer);
```

ersetzt werden.

Wahrscheinlich ist (b) in neunzig Prozent der Fälle erwünscht, aber darauf konnte man sich in »Alt-C« nicht verlassen. ANSI C brachte nun Licht in diese etwas nebulöse Situation, indem es folgende Regel aufstellte:

Wenn bei einer Makrodefinition ein formaler Parameter im Ersetzungstext mit vorangestelltem # angegeben wird, dann wird beim nachfolgenden Aufruf dieses Makros das entsprechende aktuelle Argument als Zeichenkettenkonstante dargestellt.

So wird z.B. nach folgender Präprozessoranweisung

```
#define wertvon(variable) printf("#variable" = %d\n", variable)
```

der Aufruf von `wertvon(steuer);` zunächst in

```
printf("steuer" = %d\n", steuer);
```

und dann nach der Zeichenketten-Konkatenation in

```
printf("steuer = %d\n", steuer);
```

umgewandelt⁷.

Zusammensetzen neuer Namen mit dem Operator ##

Der Operator **##** ermöglicht es, neue Namen aus anderen Namen »zusammenzukleben«:

Beispiel

```
#define y(a,b) x##a##b
.....
int x12;
.....
printf("%d\n", y(1,2));
```

Die **printf**-Anweisung wird vom Präprozessor umgewandelt in

```
printf("%d\n", x12);
```

Beispiel

```
#define x_var_test(zahl) printf("x"##zahl" = %d\n", x##zahl)
```

Ein späterer Aufruf `x_var_test(7)` wird vom Präprozessor zunächst in

```
printf("x"7" = %d\n", x7);
```

umgewandelt, und nach Konkatenation der Zeichenketten ergibt sich

```
printf("x7 = %d\n", x7);
```

7. Noch allgemeingültiger ist

```
#define wertvon(var,format) printf("#var" = "format"\n", var).
```

Dann kann man sogar Werte von Variablen mit unterschiedlichen Datentypen ausgeben, z.B. mit `wertvon(ganz,"%d");` oder `wertvon(name, "%s");`

Beispiel

```
#define a(n)  nummer##n
#define x    3
```

Ein Aufruf `a(x)` wird dann durch `nummerx` und nicht durch `nummer3` oder `nummern` ersetzt.

Rekursive Makrodefinitionen

Definitionen wie

```
#define char  unsigned char
```

bringen ANSI-C-Compiler nicht mehr in Verlegenheit. Manche frühere C-Compiler (besser: C-Präprozessoren) haben sich bei Angaben wie

```
char  zeich;
/    \
unsigned char
/    \
unsigned char
/    \
unsigned char
/    \
.....
..... "tot geschachtelt".
```

Um solche Schachtelkaskaden zu vermeiden, stellte ANSI C folgende Regel auf:

Ein Makroname, der selbst wieder in seiner eigenen Definition angegeben wird, wird nicht wieder ersetzt, sondern unverändert übernommen.

Somit sind in ANSI C z.B. Makroangaben wie

```
#define sqrt(x)  printf("Die Wurzel von %lf ist %lf\n", x, sqrt(x))
```

möglich, da ein späterer Aufruf wie z.B. `sqrt(7.5)` vom Präprozessor durch

```
printf("Die Wurzel von %lf ist %lf\n", 7.5, sqrt(7.5));
```

ersetzt wird.

2.2.2 #include – Einkopieren ganzer Dateien

Üblicherweise haben die bei **#include** angegebenen Dateien die Endung `.h` und werden *Headerdateien* genannt. Man unterscheidet zwei Arten von Headerdateien:

Standard-Headerdateien

ANSI C legt genau fest, welche Headerdateien existieren müssen:

```
assert.h,  ctype.h,  errno.h,   float.h,  limits.h,
locale.h,  math.h,   setjmp.h,  signal.h, stdarg.h,
stddef.h,  stdio.h,  stdlib.h,  string.h, time.h
```

ANSI C legt darüber hinaus weitgehend den Inhalt dieser Standard-Headerdateien fest, indem es angibt, welche Datentypen, Konstanten, Makros und Funktionen in den einzelnen Dateien zu deklarieren oder zu definieren sind.

Die Deklarationen geben ein genaues Bild, welche Rückgabe-Datentypen von den einzelnen Bibliotheksfunktionen bereitgestellt werden; zudem geben sie Anzahl und Typ der geforderten Funktionsargumente (siehe Prototypen) an.

Standard-Headerdateien werden üblicherweise in spitzen Klammern⁸ beim **#include** angegeben, z.B.:

```
#include <math.h>
```

Benutzereigene Headerdateien

Solche Headerdateien enthalten üblicherweise nützliche Konstanten- und Makrodefinitionen, aber auch eigene Datentypfestlegungen. Z.B. kann eine Konstruktion wie

```
typedef struct {
    float real_teil;
    float imag_teil;
} complex;
```

in einer Headerdatei `complex.h` stehen. Jeder Programmteil, der diese Datei mit **#include** einkopiert, kann dann von diesem Datentyp Gebrauch machen.

Neben ihrer Funktion als Sammelplatz für nützliche Konstanten-, Makro- und Datentypdefinitionen werden die Headerdateien in der Praxis auch für die Schnittstellen-Vereinbarungen zwischen mehreren Programmteilen (Modulen) verwendet (siehe Prototypbeschreibung).

Benutzereigene Headerdateien werden üblicherweise in Anführungszeichen⁹ beim **#include** angegeben, z.B.:

```
#include "complex.h"
```

Neben der Angabe von Headerdateien in `< >` und `" "` können diese auch in Form von Makronamen angegeben werden, wie z.B.

```
#ifndef UNIX
    #define INC_DATEI "unix_kdo.h"
#else
    #define INC_DATEI "dos_kdo.h"
#endif
#include INC_DATEI
```

8. Spitze Klammern veranlassen den Präprozessor, in fest vorgegebenen Pfaden nach der entsprechenden Headerdatei zu suchen (in Unix z.B. im Standard-Directory für Headerdateien `/usr/include`)

9. Anführungszeichen veranlassen den Präprozessor, im aktuellen Directory nach der entsprechenden Headerdatei zu suchen. Wird diese dort nicht gefunden, so wird in denselben Pfaden gesucht, wie wenn spitze Klammern `< . >` hier angegeben worden wären.

In allen Fällen ersetzt der Präprozessor die entsprechende **#include**-Zeile durch den vollständigen Inhalt der entsprechenden Headerdatei.

2.2.3 Bedingte Kompilierung

Mit den Präprozessor-Direktiven dieser Klasse kann man die Übersetzung einzelner Programmteile von zur Präprozessorzeit auswertbaren Bedingungen abhängig machen.

Die bedingte Kompilierung macht es somit möglich, nur eine Quelldatei zu unterhalten, die von unterschiedlichen Compilern und sogar auf unterschiedlichen Maschinen übersetzt werden kann.

Beispiel

```
#if defined BIT32
    #define ANZAHL 32
#elif defined BIT16
    #define ANZAHL 16
#else
    #define ANZAHL 8
#endif
```

Darüber hinaus wird die bedingte Kompilierung dazu verwendet, um aus einer Quelldatei zu unterschiedlichen Zeitpunkten unterschiedliche ablauffähige Programme zu erzeugen, wie z.B.

```
#define wertvon(var) printf("#var" = %s\n", var)
.....
#ifdef TEST
    wertvon(zeich_kette);
#endif
```

Tabelle 2.3 gibt einen Überblick über die Schlüsselwörter für die bedingte Kompilierung.

Schlüsselwort	Bedeutung
<code>#if <i>ausdruck</i></code>	Abhängig davon, ob <i>ausdruck</i> erfüllt ist (Auswertung ergibt einen von 0 verschiedenen Wert), wird der darauffolgende Programmteil ausgeführt.
<code>#ifdef <i>name</i></code>	Wenn <i>name</i> definiert ist, dann wird der darauffolgende Programmteil ausgeführt. Dieser Ausdruck entspricht <code>#if defined <i>name</i></code> oder <code>#if defined(<i>name</i>)</code>
<code>#ifndef <i>name</i></code>	Wenn <i>name</i> nicht definiert ist, dann wird der darauffolgende Programmteil ausgeführt. Dieser Ausdruck entspricht <code>#if !defined <i>name</i></code> oder <code>#if !defined(<i>name</i>)</code> .
<code>#elif <i>ausdruck</i></code>	Abhängig davon, ob <i>ausdruck</i> erfüllt ist (Auswertung ergibt einen von 0 verschiedenen Wert), wird der darauffolgende Programmteil ausgeführt.

Tabelle 2.3: Schlüsselwörter für bedingte Kompilierung

Schlüsselwort	Bedeutung
<code>#else</code>	leitet else-Programmteil zu den 4 vorherigen Konstruktionen (<code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , <code>#elif</code>) ein.
<code>#endif</code>	zeigt das Ende einer bedingten Kompilierungs-Konstruktion an.

Tabelle 2.3: Schlüsselwörter für bedingte Kompilierung

2.2.4 Weitere Präprozessordirektiven

#line zahl

Die hierbei als *zahl* angegebene Zeilennummer wird als neue Zeilennummer für die Quelldatei angenommen. Solche Anweisungen sind z.B. dann wichtig, wenn Headerdateien durch den Präprozessor Bestandteil der Quelldatei werden. Die Hauptverwendung für diese Direktive liegt im Bereich des Compilerbaus oder bei Programmgeneratoren. Es ist auch die folgende Angabe möglich.

#line zahl dateiname

Diese Angabe bewirkt, daß als neue Zeilennummer *zahl* und als neuer Dateiname *dateiname* genommen wird.

#pragma spezielle-compiler-anweisung

Pragmas sind compilerspezifisch. So hat z.B. der Intel-C-Compiler 4.0 das Pragma

`#pragma large`

um das *LARGE*-Modell auf den Intel-Prozessoren 80xxx. auszuwählen. Kommt in einem Programm eine **#pragma**-Direktive vor, die der Compiler nicht kennt, so wird diese einfach ignoriert.

#undef name

erlaubt die »Rücknahme« eines zuvor definierten Symbols (Umkehrung zu **#define**).

#error zeichenkette

Es wird die angegebene *zeichenkette* am Bildschirm ausgegeben, wie z.B.:

`#error "Sie haben TEST und FREIGABE gleichzeitig definiert (Widerspruch !!!)"`

2.2.5 Von ANSI C vordefinierte Makros

Die in Tabelle 2.4 angegebenen Makros muß jeder ANSI-C-Compiler (Präprozessor) verstehen und auflösen können:

Makro	Bedeutung
<code>__LINE__</code>	Zeilennummer in der momentanen Quelldatei (ganzzahlige Konstante).
<code>__FILE__</code>	Name der momentanen Quelldatei (Zeichenkettenkonstante).
<code>__DATE__</code>	Übersetzungsdatum der momentanen Quelldatei (Zeichenkettenkonstante der Form »mmm tt jjjj«; z.B. »Jun 14 1989« oder »Jun 4 1989«).
<code>__TIME__</code>	Übersetzungszeit der momentanen Quelldatei (Zeichenkettenkonstante der Form »hh:mm:ss«; z.B.: »14:32:53«).
<code>__STDC__</code>	Erkennungsmerkmal für einen ANSI C Compiler: Ist diese ganzahlige Konstante mit Wert 1 gesetzt, so handelt es sich um einen ANSI-C-Compiler.

Tabelle 2.4: Von ANSI C vordefinierte Makros

Das folgende Programm 2.1 (`praeproz.c`) ist ein Demonstrationsbeispiel zu den vordefinierten ANSI-C-Makros.

```
#include <stdio.h>

int
main(void)
{
    printf("Zeile %d in Datei %s (um %s Uhr am %s)\n",
        __LINE__, __FILE__, __TIME__, __DATE__);
    # line 100 "test.c"
    printf("Zeile %d in Datei %s\n", __LINE__, __FILE__);
}
```

Programm 2.1 (**praeproz.c**): Demonstration zu den vordefinierten ANSI-C-Makros

Nachdem man dieses Programm 2.1 (`praeproz.c`) kompiliert und gelinkt hat

```
cc -o praeproz praeproz.c
```

liefert es beim Aufruf z.B. die folgende Ausgabe:

```
$ praeproz
Zeile 8 in Datei praeproz.c (um 11:33:11 Uhr am May 23 1995)
Zeile 100 in Datei test.c
$
```

2.3 Die Sprache ANSI C

In diesem Kapitel werden die wichtigsten Aspekte und Neuheiten von ANSI C gegenüber dem nicht standardisierten »Alt-C« vorgestellt.

2.3.1 Grunddatentypen

Hier wurde ein neues Schlüsselwort **signed** (Gegenstück zu **unsigned**) eingeführt, um explizit festlegen zu können, daß ein Wert mit Vorzeichen dargestellt werden soll. Nachfolgend werden die Grunddatentypen und die von ANSI C dafür vorgegebenen Eigenschaften kurz vorgestellt.

char

Objekte von diesem Datentyp können genau ein Zeichen aufnehmen. Es ist dabei der jeweiligen Implementierung überlassen, ob **char** vorzeichenbehaftet ist oder nicht.

Vorzeichenbehaftete Ganzzahltypen

- (a) **signed char**
- (b) **short, signed short, short int, signed short int**
- (c) **int, signed, signed int**, *keine Typ-Angabe*
- (d) **long, signed long, long int, signed long int**

Bezüglich der Wertebereiche muß folgende Forderung erfüllt sein:

- (a) $\leq$ (b) $\leq$ (c) $\leq$ (d)

Vorzeichenlose Ganzzahltypen

unsigned char
unsigned short, unsigned short int
unsigned, unsigned int
unsigned long, unsigned long int

Gleitpunkttypen

- (a) **float**
- (b) **double**
- (c) **long double**

Bezüglich der Wertebereiche muß folgende Forderung erfüllt sein:

- (a) $\leq$ (b) $\leq$ (c)

long float ist in ANSI C nicht mehr erlaubt.

Die genauen Wertebereiche, die von den einzelnen Datentypen abgedeckt werden, sind von Compiler und Maschine abhängig. ANSI C legt lediglich fest, daß diese Grenzen in den zwei Headerdateien `<limits.h>` und `<float.h>` definiert sein müssen.

enum-Angabe

Der Aufzählungsdattentyp **enum** ist zwar keine Neuerfindung vom ANSI-Komitee, dennoch brachte ANSI C es mit sich, daß **enum** nun ein fester Bestandteil der Sprache C ist, was in der Vor-ANSI-Zeit nicht immer der Fall war. ANSI C gibt zudem eine umfassende Beschreibung zum Aufzählungsschlüsselwort **enum** wieder, das verwendet wird, um C-Programme lesbarer zu machen:

- **enum** erlaubt es, Werten Namen zu geben.

So wird z.B. mit der Deklaration

```
enum hunde_art {schaeferhund, dackel, pudel};
```

ein neuer Datentyp `enum hunde_art` festgelegt, der genau drei gültige Werte umfaßt: `schaeferhund`, `dackel` und `pudel`. Mit

```
enum hunde_art ausgeh_hund;
```

wird eine Variable `ausgeh_hund` definiert, die genau diese drei Werte annehmen kann¹⁰. Man hätte das gleiche erreicht, wenn man folgendes angegeben hätte:

```
#define schaeferhund 0
#define dackel      1
#define pudel       2
```

und `ausgeh_hund` als **int**-Variable deklariert hätte.

- **enum**-»Wertenamen« dürfen nur einmal angegeben werden. So ist z.B. die folgende Angabe nicht erlaubt:

```
int variable;
enum hunde_art { schaeferhund, dackel, pudel };
enum haustiere { kanarien_vogel, papagei, schaeferhund };
```

denn bei einer späteren Zuweisung wie z.B.

```
variable=schaeferhund; /* ist erlaubt */
```

kann der Compiler nicht entscheiden, ob er den `schaeferhund`-Wert aus `hunde_art` oder `haustiere` zuweisen soll.

- **enum**-»Wertenamen« dürfen nicht als Variablennamen verwendet werden. So ist z.B. die folgende Angabe verboten:

```
enum hunde_art {schaeferhund, dackel, pudel};
int dackel=5;
```

10. oft auch mehr, da die meisten Compiler für `ausgeh_hund` 2 oder gar 4 Bytes reservieren.

Denn was wäre z.B. als Argument beim Funktionsaufruf `hundesteuer(dackel)` zu übergeben: `dackel`-Wert 1 aus `hunde_art` oder der Wert 5 der Variablen `dackel`.

- **enum**-Variable oder **enum**-Werte können überall dort verwendet werden, wo ganzzahlige Werte erlaubt sind, wie z.B.

```
enum hunde_art {schaeferhund, dackel, pudel};
int durchschnitts_hoehe[3] = {80, 20, 20};
:
printf("Ein Schaeferhund ist durchschnittl. %d cm hoch\n",
      durchschnitts_hoehe[schaeferhund]);
```

- **enum**-»Werte-Namen« können auch Werte zugewiesen werden, wie z.B.

```
enum stellen_wert { null=1, eins=2, zwei=4, drei=8,
                  vier=16, fuenf=32, sechs=64, sieben=128,
                  byte_max=128 };
```

Aus diesem Beispiel ist zu ersehen, daß jeder **enum**-Konstante ein eigener Wert zugewiesen werden kann, wobei unterschiedlichen Wertennamen auch gleiche Werte zugewiesen werden dürfen.

2.3.2 Datentyp void

ANSI C führt endgültig den Datentyp **void** (deutsch: nichts, wertlos) ein, der sich auf drei Gebieten verwenden läßt:

Rückgabedatentyp für Funktionen

Dieses neue Schlüsselwort erlaubt nun auch in C die Unterscheidung von Prozeduren¹¹ und Funktionen. Z.B. bedeutet folgende Deklaration, daß die Funktion **exit** keinen Wert zurückgibt.

```
void exit(int nummer);
```

Im ursprünglichen C mußte eine solche »Prozedur« mit

```
int exit(nummer)
    int nummer;
```

angegeben werden, woraus nicht klar erkennbar war, ob diese Funktion nun einen **int**-Wert liefert oder als Prozedur zu betrachten ist.

Zeiger auf void (Generische Zeiger)

Mit folgender Deklaration wird nur ein Zeiger festgelegt. Es wird noch nicht angegeben, auf welchen Datentyp dieser Zeiger einmal zeigen wird.

```
void *allg_zeiger;
```

11. procedure in PASCAL und Funktionen ohne Rückgabewert in C

Mit der nächsten Deklaration wird festgelegt, daß die Funktion **malloc** einen `void`-Zeiger zurückgibt.

```
void *malloc(size_t laenge);
```

malloc stellt einen zusammenhängenden Speicherbereich von `laenge`-Bytes zur Verfügung. Wie dieser Speicherbereich zu nutzen ist¹², ist Sache des Aufrufers, der *casting* verwendet, um diesem strukturlosen Speicherplatz seine Struktur zu geben. Aus der Sicht von **malloc** ist nur die Anfangsadresse wichtig, und die ist datentypfrei (**void ***).

Funktionen ohne Parameter

Wenn eine Funktion keine formalen Parameter besitzt, dann kann dies in ANSI C mit Angabe von `void` in den Funktionsklammern angegeben werden, wie z.B.:

```
int funk_name(void);
```

Ein anderes Beispiel ist die Deklaration der Bibliotheksfunktion **abort**:

```
void abort(void);
```

Die Funktion **abort** bewirkt einen Programmabbruch.

2.3.3 Die neuen Schlüsselwörter **const** und **volatile**

Die beiden neuen Schlüsselwörter **const** und **volatile** werden bei Variablendeklarationen und -definitionen verwendet:

const

Dieses Schlüsselwort teilt dem Compiler mit, daß das zugehörige Objekt nicht modifiziert werden darf, d.h., nach dieser Deklaration darf einem solchen Objekt weder ein Wert zugewiesen noch darf es inkrementiert oder dekrementiert werden. Noch eine Besonderheit, die es im Zusammenhang mit Zeigern und **const** zu beachten gibt, ist die Stelle, an der **const** angegeben ist:

```
const int *zgr_auf_konstante;  
int *const konstanter_zgr;
```

Der Inhalt des Speicherplatzes, auf den `zgr_auf_konstante` zeigt, darf beim Zugriff über `zgr_auf_konstante` nicht verändert werden. `zgr_auf_konstante` selbst dagegen darf verändert werden. Im Gegensatz dazu darf sehr wohl der Inhalt, auf den `konstanter_zgr` zeigt, verändert werden, aber `konstanter_zgr` selbst darf nicht modifiziert werden.

12. mit `int` oder `char`-Werten oder vielleicht mit einer vom Benutzer vorgegebenen Struktur?

volatile

Dieses Schlüsselwort kann als Gegenstück zu **const** verstanden werden: Es sollte für Variablen verwendet werden, die nicht nur durch das Programm selbst, sondern auch jederzeit von »außen« (z.B. durch *Interrupts*) verändert werden können¹³. Bei Angabe dieses Schlüsselworts muß der Compiler sicherstellen, daß jedes vom Programmierer vorgegebene Lesen und Beschreiben eines **volatile**-Objekts genau wie vorgegeben stattfindet. Ein Compiler darf also vorgegebene Lese- oder Schreiboperationen auf **volatile**-Objekte nicht »wegoptimieren«. Programm 2.2 (*sumunger.c*) verdeutlicht dies.

```
#include <stdio.h>

int
main(void)      /* Summe aller ungeraden Zahlen berechnen */
{
    int sum=0, i, n;

    printf("Gib N ein: ");
    scanf("%d", &n);
    for (i=1; i<=n; i=i+2)
        sum += i;

    /*.....Weiterer Code.....*/

    exit(0);
}
```

Programm 2.2 (*sumunger.c*): Summe von ungeraden Zahlen

Dieses Beispiel kann einen Optimierer in einem Compiler dazu bringen, nicht für jeden Schleifendurchlauf *sum* und *i* auf den wirklichen Wert zu setzen, sondern die entsprechenden Werte zu diesen beiden Variablen in den Registern zu halten und erst mit dem Abschluß der Schleife die ermittelten Werte aus den Registern in den Speicher und damit in die Variablen *sum* und *i* zu schreiben. In diesem Beispiel würde diese Vorgehensweise keinen Schaden anrichten. Bei hardwarenaher Programmierung (wie Gerätetreiber oder Zeiger auf ein E/A-Port) kann allerdings eine solche Optimierung unerwartete Folgen haben:

```
short *bildschirm_port = TTYADDR;
:
for (i=0 ; i<n ; i++)
    *bildschirm_port = vektor[i];
```

Da hier nicht garantiert ist, daß wirklich ein Code – wie vorgegeben – generiert wird, muß das Schlüsselwort **volatile** angegeben werden:

13. Volatile bedeutet ins Deutsche übersetzt: flatterhaft, unbeständig. Es weist den Compiler darauf hin, sich bei seiner Codegenerierung nicht darauf zu verlassen, daß der Inhalt des entsprechenden Objekts konstant bleibt, sondern sich jederzeit ändern kann.

```
volatile short *bildschirm_port = TTYADDR;
:
for (i=0 ; i<n ; i++)
    *bildschirm_port = vektor[i];
```

Die Kombination beider Schlüsselwörter ist auch möglich. So bedeutet z.B. die folgende Angabe

```
extern const volatile int  real_time_clock;
```

daß der Inhalt von `real_time_clock` zwar von der Hardware verändert werden darf, aber es kann dieser Variablen weder ein Wert zugewiesen, noch kann sie inkrementiert oder dekrementiert werden.

2.3.4 Primitive Systemdatentypen

ANSI C hat sogenannte primitive Systemdatentypen eingeführt, deren Name immer mit `_t` endet. Es handelt sich hierbei nicht um echte Datentypen wie **char** oder **double**. Diese Datentypen sind gewöhnlich mit **typedef** in unterschiedlichen Headerdateien, in Unix aber üblicherweise auch in `<sys/types.h>` definiert. Der Zweck dieser Systemdatentypen ist es, daß der Benutzer nicht mehr spezielle Daten mit **int**, **short** oder **long** definiert, sondern es der jeweiligen Implementierung überläßt, die geeigneten Typen für das spezielle System zu wählen. Nehmen wir z.B. die Funktion

```
void *malloc(size_t laenge);
```

Hier ist es implementierungsabhängig, ob beim Aufruf von **malloc** nur **unsigned**- oder auch **unsigned long**-Werte angegeben werden können.

2.3.5 Funktionsprototypen – Die große Neuheit von ANSI C

In »Alt-C« teilte eine Funktionsdeklaration dem Compiler lediglich den Datentyp des Rückgabewerts mit:

```
float  hoch();
char  *strcpy();
int    abort();
```

Wenn eine Funktion nicht vor ihrem Aufruf deklariert wurde, nahm der Compiler den Rückgabe-Datentyp **int** an, was dazu führte, daß Funktionen, die **int**-Werte zurücklieferten erst gar nicht mehr deklariert werden mußten. C bot auch keine Möglichkeit, den Typ und die Anzahl der Funktionsargumente anzugeben, was in anderen Programmiersprachen wie PASCAL schon immer möglich war. ANSI C führte nun Funktionsprototypen ein. Dies ist wahrscheinlich die bedeutendste Neuheit von ANSI C.

Funktionsprototypen ermöglichen es, bei der Deklaration einer Funktion nicht nur den Rückgabedatentyp, sondern auch die Typen der einzelnen formalen Parameter anzugeben, wie z.B.:

```
float hoch(float, int);
char *strcpy(char *, const char *);
void abort(void);
```

Es ist sogar möglich, neben dem Typ eines formalen Arguments noch einen Namen anzugeben:

```
float hoch(float zahl, int potenz);
char *strcpy(char *ziel, const char *quelle);
void abort(void); /* hier kein Name wahlbar */
```

Eine Kombination beider Methoden ist auch möglich:

```
float hoch(float zahl, int );
char *strcpy(char *, const char *quelle);
void abort(void); /* hier kein Name wahlbar */
```

2.3.6 Ellipsen-Prototypen für Funktionen mit variabler Parameterzahl

In »Alt-C« wurden alle übergebenen Parameter eines Funktionsaufrufs »von rechts nach links« auf den Stack abgelegt. Der Vorteil dieser Methode ist, daß eine variabel lange Liste von aktuellen Parametern beim Aufruf von Funktionen wie **printf** möglich war. Der Nachteil dieser Vorgehensweise war, daß manchmal von C-Compilern nicht so effizienter Code beim Funktionsaufruf erzeugt werden konnte, wie z.B. von PASCAL-Compilern, wo die Anzahl und der Typ der Argumente zum Aufrufzeitpunkt bekannt ist.

Ein weiterer Nachteil dieser Methode war, daß sie nicht den standardisierten Aufruffolgen einiger Betriebssysteme entsprach. Nichtsdestoweniger mußte bei allen Funktionsaufrufen die ineffizientere Aufrufsequenz gewählt werden, um gelegentlichen **printf**-Aufrufen gerecht zu werden.

Das Linken von Modulen aus anderen Sprachen wurde durch diese speziellen C-Aufruffolgen ebenfalls nicht erleichtert. Das ANSI-C-Komitee war über diese Nachteile nicht besonders glücklich und stellte folgende Regel auf:

Funktionen, die eine variable Anzahl von Argumenten erwarten, müssen mit sogenannten Ellipsen-Prototypen deklariert werden, wie z.B.:

```
int printf(const char *format, ...);
```

Die drei Punkte (Ellipse) bei einer Deklaration deuten an, daß beim Aufruf von **printf** neben einem fest vorgeschriebenen Parameter `format` beliebig weitere aktuelle Parameter angegeben werden können.

Mit der Einführung von Ellipsen kann der Compiler bei jedem Funktionsaufruf ohne vorheriger Ellipsen-Prototyp-Deklaration annehmen, daß diese Funktion eine feste Anzahl von Parametern hat. In solchen Fällen kann immer der effizientere Aufrufmechanismus (Argumente »von links nach rechts« ablegen) gewählt werden.

Der weniger effizientere Mechanismus (Argumente »von rechts nach links« auf den Stack legen) muß nur noch dann gewählt werden, wenn zu der entsprechenden Funktion ein »Ellipsen-Prototyp« vorliegt.

2.3.7 Abarbeiten variabel langer Argumentlisten

Um eine variable Anzahl von Argumenten innerhalb einer Funktion abarbeiten zu können, sind die folgenden Schritte notwendig:

1. Zugriff auf die fest vorgegebenen Parameter über deren Namen ist wie bisher möglich.
2. Deklaration einer Zeigervariablen des (in der Standard-Headerdatei `<stdarg.h>` definierten) Typs

```
va_list arg_list_zgr;
```

3. Aufruf des Makros **`va_start`** mit zwei Argumenten: dem Namen des zuvor deklarierten Zeigers (Typ `va_list`) und dem Namen des letzten fixen Parameters:

```
va_start(arg_list_zgr, letzt_param);
```

Dieser Aufruf ermittelt anhand des letzten fixen Arguments, wo das erste variable Argument (auf dem Stack) gespeichert ist, und setzt `arg_list_zgr` auf den Anfang dieser variablen Argumentenliste.

4. Wiederholter Aufruf des Makros **`va_arg`**, um die variable Argumentenliste »Stück für Stück« abzuarbeiten.

```
va_arg(arg_list_zgr, datentyp);
```

Dieses Makro schaltet `arg_list_zgr` immer ein Argument weiter in dieser Liste. Als erstes Argument ist bei **`va_arg`** der `arg_list_zgr` anzugeben. Das zweite Argument muß den Typ des zu erwartenden Arguments festlegen, um **`va_arg`** die Größe des entsprechenden variablen Arguments mitzuteilen. Es ist zu beachten, daß bei **`char`**-Argumenten der Typ **`int`** und bei **`float`**-Argumenten der Typ **`double`** anzugeben ist. Das Ende einer variabel langen Argumentenliste muß über getroffene Vereinbarungen erkannt werden, wie z.B. erstes Argument gibt die Anzahl der aktuellen Argumente an, oder letztes Argument ist `-1`¹⁴ usw. (siehe auch Beispiel).

5. Vor Rückkehr aus dieser Funktion muß noch das Makro **`va_end`** aufgerufen werden:

```
va_end(arg_list_zgr);
```

Dieser Aufruf setzt `arg_list_zgr` auf `NULL` und versetzt den Stack wieder in einen »sauberen« Zustand. Ohne diesen Aufruf kann der weitere Programmablauf ein seltsames Verhalten zeigen.

14. Das ist nur möglich, wenn alle Argumente numerische Werte von einem Typ (z.B. `int`) sind.

Abarbeiten von variablen Argumentlisten (zentrale Fehlerroutine)

Bei größeren Softwareprodukten ist es üblich, ein Modul (z.B. `fehlausg.c`) zu entwerfen, das für die Ausgabe von Fehlermeldungen zuständig ist. Jedes andere Modul, das Fehlermeldungen ausgeben möchte, kann dann die Fehlerroutine aus Modul `fehlausg.c` aufrufen. Da Fehlermeldungen meist variable Komponenten (wie z.B. Dateinamen) enthalten, bietet es sich bei der Verwirklichung der zentralen Fehlerroutine an, mit variabel langen Argumentenlisten zu arbeiten. Die Länge der variabel langen Argumentenliste kann über ein **`printf`**-ähnliches Format gesteuert werden, wie Programm 2.3 (`fehlausg.c`) zeigt. Es gibt zwei verschiedene Methoden, wie in diesem Fall die variabel lange Argumentliste abgearbeitet werden könnte:

1. Unter Zuhilfenahme der Funktion **`vsprintf`** (in `feh1_meld1`)
2. Durch wiederholten Aufruf von **`va_arg`** (in `feh1_meld2`)

```
#include <stdio.h>
#include <stdarg.h>

#define MAX_ZEICHEN 4096

/*----- feh1_meld1 -----*/
void feh1_meld1(const char *fmt, ...)
{
    va_list az;
    char puffer[MAX_ZEICHEN];

    va_start(az, fmt);
    vsprintf(puffer, fmt, az);
    fprintf(stderr, "%s\n", puffer);
    va_end(az);
    return;
}

/*----- feh1_meld2 -----*/
void feh1_meld2(const char *fmt, ...)
{
    va_list az;
    char puffer[MAX_ZEICHEN];

    va_start(az, fmt);
    while (*fmt) {
        if (*fmt != '%') {
            putc(*fmt, stderr);
        } else {
            switch(++fmt) {
                case 'c' : fprintf(stderr, "%c", va_arg(az, int));
                           break;
                case 'd' : fprintf(stderr, "%d", va_arg(az, int));
                           break;
            }
        }
    }
}
```

```

        case 'f' : fprintf(stderr, "%f", va_arg(az, double));
                    break;
        case 's' : fprintf(stderr, "%s", va_arg(az, char *));
                    break;
        case 'l' : if (++fmt=='d')
                        fprintf(stderr, "%ld", va_arg(az, long));
                    else
                        fprintf(stderr, "%lf", va_arg(az, double));
                    break;
    }
    }
    fmt++;
}
fprintf(stderr, "\n");
va_end(az);
}

#ifdef TEST
/*----- main -----*/
int
main(void)
{
    double wert = 3.0/7;
    char *name = "Hans";

    fehl_meld1("%d * %d = %d", 2, 3, 2*3);
    fehl_meld2("%d * %d = %d", 2, 3, 2*3);
    fehl_meld1("%s ist %lf", "Drei geteilt durch sieben", wert);
    fehl_meld2("%s ist %lf", "Drei geteilt durch sieben", wert);
    fehl_meld1("%s ist %d alt", name, 34);
    fehl_meld2("%s ist %d alt", name, 34);

    exit(0);
}
#endif

```

Programm 2.3 (fehlausg.c): Verwirklichung von FehlerROUTINEN mit variabel langen Argumentlisten

In Programm 2.3 wird zugleich über eine `#ifdef TEST..... #endif` – Klammer eine Möglichkeit zum Testen der beiden zentralen Fehlerbehandlungsroutine `fehl_meld1` und `fehl_meld2` gegeben. Dazu muß bei der Kompilierung nur der Name `TEST` definiert werden, wie z.B.:

```
cc -DTEST -o fehlausg fehlausg.c
```

Ruft man dann `fehlausg` auf, so gibt es folgendes aus:

```

$ fehlausg
2 * 3 = 6
2 * 3 = 6
Drei geteilt durch sieben ist 0.428571

```

```
Drei geteilt durch sieben ist 0.428571
Hans ist 34 alt
Hans ist 34 alt
$
```

Im Anhang befindet sich das Listing zum Programm `fehler.c`, das die Funktion `fehler_meld` definiert. Diese Funktion `fehler_meld` wird in den Beispielen der späteren Kapitel benutzt.

2.4 Die ANSI-C-Bibliothek

ANSI C hat den Inhalt der C-Bibliothek fest vorgeschrieben. Die Prototypdeklarationen für die einzelnen Bibliotheksroutinen befinden sich in den sogenannten Standard-Headerdateien. Ebenso sind in diesen Headerdateien noch Definitionen von Konstanten, Makros und Datentypen enthalten.

Tabelle 2.5 gibt eine Übersicht über die in ANSI C vorgeschriebenen Headerdateien. Die in dieser Tabelle mit * gekennzeichneten Headerdateien sind an anderer Stelle in diesem Buch ausführlich beschrieben. In der Tabelle 2.5 wird dabei noch ein Hinweis auf das entsprechende Kapitel gegeben. Alle anderen Headerdateien werden kurz in diesem Kapitel vorgestellt.

Headerdatei	definiert bzw. deklariert
<code>assert.h</code>	das Makro <i>assert</i> und nimmt Bezug auf das Symbol <code>NDEBUG</code> . Diese Headerdatei wird während der Testphase eines Programms benötigt;
<code>ctype.h</code>	Routinen zum Klassifizieren von Zeichen (z.B. stellt <i>islower</i> fest, ob es sich beim angegebenen Zeichen um einen Kleinbuchstaben handelt);
<code>errno.h</code>	die beiden Konstanten <code>EDOM</code> und <code>ERANGE</code> , die verwendet werden, um bestimmte Fehlersituationen anzuzeigen; außerdem wird hier die globale <i>int</i> -Variable <code>errno</code> definiert, die von bestimmten Bibliotheksfunktionen gesetzt wird, wenn bei deren Ausführung Fehler auftraten;
<code>float.h</code>	Konstanten für den Rundungsmodus und für maximale und minimale Werte von Gleitpunktzahlen;
<code>limits.h</code>	Konstanten, die die Limits für ganzzahlige Datentypen festlegen;
<code>locale.h</code>	Konstanten, Datentypen und Funktionen, die notwendig sind, um ein C-Programm auf einen speziellen Kultur- oder Sprachkreis anzupassen;
<code>math.h</code>	mathematische Funktionen und die Konstante <code>HUGE_VAL</code> , die einen sehr großen Wert (für nicht darstellbare Ergebnisse) repräsentiert;
<code>setjmp.h</code> *	Datentypen und Funktionen, die sogenannte nicht-lokale Sprünge über Funktionsgrenzen hinweg ermöglichen; diese nicht-lokalen Sprünge mit den beiden Funktionen <i>setjmp</i> und <i>longjmp</i> werden in Kapitel 8 ausführlich beschrieben;

Tabelle 2.5: Die ANSI-C-Headerdateien

Headerdatei	definiert bzw. deklariert
signal.h *	Datentypen und Funktionen, die benötigt werden, um während einer Programmausführung auftretende Signale abzufangen oder aber selbst Signale schicken zu können; die in dieser Headerdatei definierten Datentypen, Konstanten, Makros und Funktionen werden in Kapitel 13 bei der Vorstellung des Unix-Signalkonzepts detailliert beschrieben.
stdarg.h *	den Datentyp <code>va_list</code> und die Makros <code>va_start</code> , <code>va_arg</code> und <code>va_end</code> , um variabel lange Argumentlisten in einer Funktion abarbeiten zu können; die dabei zu verwendenden Verfahren und der Inhalt dieser Headerdatei <code><stdarg.h></code> wurde bereits in Kapitel 2.3 beschrieben;
stddef.h	die Datentypen <code>ptrdiff_t</code> (für das Ergebnis einer Subtraktion zweier Zeiger), <code>size_t</code> (für das Ergebnis des <code>sizeof</code> -Operators) und <code>wchar_t</code> (deckt den gesamten Bereich für alle möglichen Repräsentationen von Zeichen ab ¹⁵); daneben sind hier noch die beiden Makros <code>NULL</code> (Nullzeiger-Konstante) und <code>offsetof</code> (liefert Byte-Abstand einer Strukturkomponente vom Strukturanfang);
stdio.h *	Datentypen, Makros und Funktionen, die für die Standard-Ein- / Ausgabe von C-Programmen benötigt werden. Die hier definierten Konstanten und deklarierten Standard-Ein- / Ausgabefunktionen werden ausführlich in Kapitel 3 beschrieben;
stdlib.h	Datentypen, Makros und Funktionen, um allgemein nützliche Aufgaben durchzuführen, wie z.B. die Umwandlung von Zeichenketten in ganze Zahlen, Erzeugung von Zufallszahlen, Reservierung von Speicherplatz usw.;
string.h	einen Datentyp, Makros und Funktionen, die zur Bearbeitung von Strings benötigt werden.
time.h *	Datentypen, Konstanten und Funktionen, um Zeitabfragen und -konvertierungen durchzuführen; der Inhalt dieser Headerdatei wird ausführlich in Kapitel 7 beschrieben.

Tabelle 2.5: Die ANSI-C-Headerdateien

Im folgenden werden nur die Headerdateien, die in keinem späteren Kapitel genauer beschrieben werden, kurz vorgestellt, da deren Konstrukte und Funktionen in den späteren Programmbeispielen ohne jegliche weitere Erklärung verwendet werden.

2.4.1 <assert.h> – Testmöglichkeit mit der assert-Funktion

NDEBUG

Ist dieses Makro definiert, dann werden alle Aufrufe der **assert**-Funktion vom Compiler ignoriert.

15. `wchar_t` steht für wide character und wurde eingeführt, um auch asiatische Zeichensätze, welche teilweise über mehr als 10000 Zeichen verfügen, in C verwenden zu können

```
void assert(int ausdruck);
```

Wenn *ausdruck* == 0 ist, dann wird das Programm mit Fehlermeldung beendet. Dieses Makro ist sehr hilfreich bei der Entwicklung eines Programms, um logische Fehler aufzudecken.

Beispiel

Im Rahmen eines größeren Projekts besteht eine Teilaufgabe darin, eine Funktion zur Verfügung zu stellen, die eine positive Zahl in Worten ausgibt. Die dieser Routine übergebene Zahl muß positiv sein. Da das Austesten einer solchen Routine zum Zeitpunkt der Integration zu spät ist, verwendet man oft folgendes Verfahren:

Man bettet zusätzlich zum eigentlichen Programm noch eine *main*-Funktion in eine `#ifndef FREIGABE...#endif`-Klammer ein. Nachdem der Modultest erfolgreich durchgeführt wurde, muß nur noch `FREIGABE` definiert werden, um die Kompilierung der *main*-Funktion zu unterdrücken.

Programm 2.4 (`assert.c`) verdeutlicht dieses Verfahren, wobei hier zum Testen die Funktion **assert** verwendet wird.

```
#include <stdio.h>
#include <assert.h>

static char *ziffer_wort[] =
{ "null", "eins", "zwei", "drei", "vier",
  "fuenf", "sechs", "sieben", "acht", "neun" };

void ausgabe(long int zahl)
{
    int rest = zahl % 10;

    assert(zahl>=0);
    if (zahl/10 > 0) {
        ausgabe(zahl/10);
    }
    printf(" %s", ziffer_wort[rest]);
}

#ifdef FREIGABE
int main(void)
{
    long int zahl;
    printf("Gib Zahl ein: ");
    scanf("%ld", &zahl);
    ausgabe(zahl);
    printf("\n");

    exit(0);
}

#endif
```

Programm 2.4 (**assert.c**): Demonstrationsbeispiel zur Funktion **assert**

Falls in diesem Programm 2.4 (assert.c) die Funktion `ausgabe` mit einer negativen Zahl aufgerufen wird, beendet sich das Programm mit folgender Fehlermeldung:

```
assert.c:12: failed assertion `zahl>=0'
```

2.4.2 <ctype.h> – Klassifizieren oder Umwandeln von Zeichen

In der Headerdatei `<ctype.h>` sind Funktionen deklariert, die zur Klassifizierung von Zeichen oder zur Umwandlung zwischen Klein- und Großschreibung verwendet werden können. Alle haben ein `int`-Argument. Beim Aufruf sollte hierfür entweder ein **unsigned-char**-Wert oder EOF als aktuelles Argument angegeben werden, ansonsten ist das Verhalten undefiniert.

Funktion	liefert TRUE, wenn..., und sonst FALSE.
<code>int isalnum(int zeich)</code>	<code>zeich</code> ein alphanumerisches Zeichen (A...Z,a...z,0...9) ist
<code>int isalpha(int zeich)</code>	<code>zeich</code> ein Buchstabe aus dem Alphabet (A...Z,a...z) ist
<code>int iscntrl(int zeich)</code>	<code>zeich</code> ein Steuerzeichen (Hexa-Code: 0x00 ... 0x1f und 0x7f) ist
<code>int isdigit(int zeich)</code>	<code>zeich</code> eine Ziffer (0...9) ist
<code>int isgraph(int zeich)</code>	<code>zeich</code> ein druckbares Zeichen (Leerzeichen ausgenommen) ist
<code>int islower(int zeich)</code>	<code>zeich</code> ein Kleinbuchstabe (a...z) ist
<code>int isprint(int zeich)</code>	<code>zeich</code> ein druckbares Zeichen (Hexa-Code: 0x20..0x7E) ist
<code>int ispunct(int zeich)</code>	<code>zeich</code> ein druckbares Zeichen, aber kein Leerzeichen oder alphanumerisches Zeichen ist
<code>int isspace(int zeich)</code>	<code>zeich</code> ein Zwischenraum-Zeichen (Leerzeichen, \f, \n, \r, \t, \v) ist
<code>int isupper(int zeich)</code>	<code>zeich</code> ein Großbuchstabe (A...Z) ist
<code>int isxdigit(int zeich)</code>	<code>zeich</code> eine hexadezimale Ziffer (0...9,a...f,A...F) ist

Zusätzlich müssen laut ANSI C noch die beiden folgenden Funktionen in `<ctype.h>` definiert sein:

<code>int tolower(int zeich)</code>	Ist <code>zeich</code> ein Großbuchstabe, dann liefert <code>tolower</code> den entsprechenden Kleinbuchstaben, ansonsten wird <code>zeich</code> unverändert zurückgegeben.
<code>int toupper(int zeich)</code>	Ist <code>zeich</code> ein Kleinbuchstabe, dann liefert <code>toupper</code> den entsprechenden Großbuchstaben, ansonsten wird <code>zeich</code> unverändert zurückgegeben.

Neben den hier angegebenen Funktionen darf jede C-Realisierung noch eigene Funktionen anbieten, solange deren Namen mit `isk` oder `tok` (*k* steht für Kleinbuchstabe) beginnen. Oft wird z.B. noch die von »Alt-C« her bekannte Routine angeboten

`int isascii(int zeich)`
liefert TRUE, wenn es sich bei *zeich* um ein ASCII-Zeichen handelt, sonst FALSE.

2.4.3 <errno.h> – Anzeigen von Fehlersituationen durch Bibliotheksfunktionen

EDOM
ganzzahlige Konstante, die einen Domainfehler anzeigt. Diese Konstante wird immer dann von einer Bibliotheksfunktion verwendet, wenn diese anzeigen will, daß ihr ein ungültiges Argument übergeben wurde (z.B. `sqrt(-2.3)`)

ERANGE
ganzzahlige Konstante, die einen Bereichsfehler anzeigt. Diese Konstante wird immer dann von einer Bibliotheksfunktion verwendet, wenn diese anzeigen will, daß das wirkliche Ergebnis von ihr nicht dargestellt werden kann, z.B. weil es zu groß ist.

Ebenso wird ein Name (meist globale Variable) vom Typ **int** definiert:

errno
Viele Bibliotheksfunktionen setzen diese globale Variable auf einen von 0 verschiedenen Wert, wenn bei ihrer Ausführung ein Fehler auftritt. ANSI C garantiert nur, daß diese Variable beim Programmstart auf 0 gesetzt wird; allerdings wird diese Variable niemals von einer Bibliotheksfunktion zurückgesetzt. Folglich ist es gängige Praxis, daß man `errno` vor einem Bibliotheksaufruf explizit auf 0 setzt, wenn überprüft werden soll, ob ein Fehler während der Ausführung dieser Bibliotheksfunktion auftrat.

2.4.4 <float.h> – Limits und Eigenschaften für Gleitpunkt-Datentypen

Die in `<float.h>` definierten Konstanten legen maximale oder minimale Werte für Gleitpunktzahlen fest. In der folgenden Tabelle 2.6 ist die von ANSI C vorgeschriebene Mindestforderung in Klammern angegeben:

Konstante	Beschreibung
FLT_RADIX	Basis für die Exponentendarstellung; meist 2 (>=2)
FLT_MANT_DIG	Anzahl der Mantissenstellen in float
DBL_MANT_DIG	Anzahl der Mantissenstellen in double
LDBL_MANT_DIG	Anzahl der Mantissenstellen in long double
FLT_DIG	Anzahl der signifikanten dez. Ziffern in float (>=6)

Tabelle 2.6: Limits für Gleitpunktzahlen (in `<float.h>`)

Konstante	Beschreibung
DBL_DIG	Anzahl der signifikanten dez. Ziffern in double (≥ 10)
LDBL_DIG	Anzahl der signifikanten dez. Ziffern in long double (≥ 10)
FLT_MIN_EXP	kleinster negativer FLT_RADIX-Exponent für float-Werte
DBL_MIN_EXP	kleinster negativer FLT_RADIX-Exponent für double-Werte
LDBL_MIN_EXP	kleinster negativer FLT_RADIX-Exponent für long double
FLT_MIN_10_EXP	kleinster negativer Zehnerexponent für float-Werte (≤ -37)
DBL_MIN_10_EXP	kleinster negativer Zehnerexponent für double-Werte (≤ -37)
LDBL_MIN_10_EXP	kleinster negativer Zehnerexponent für long double (≤ -37)
FLT_MAX_EXP	größter FLT_RADIX-Exponent für float-Werte
DBL_MAX_EXP	größter FLT_RADIX-Exponent für double-Werte
LDBL_MAX_EXP	größter FLT_RADIX-Exponent für long double-Werte
FLT_MAX_10_EXP	größter Zehnerexponent für float-Werte ($\geq +37$)
DBL_MAX_10_EXP	größter Zehnerexponent für double-Werte ($\geq +37$)
LDBL_MAX_10_EXP	größter Zehnerexponent für long double-Werte ($\geq +37$)
FLT_MAX	größter darstellbarer endlicher float-Wert ($\geq 1E+37$)
DBL_MAX	größter darstellbarer endlicher double-Wert ($\geq 1E+37$)
LDBL_MAX	größter darstellbarer endlicher long double-Wert ($\geq 1E+37$)
FLT_EPSILON	kleinster positiver float-Wert x, für den noch gilt: $1.0+x!=x$ ($\leq 1E-5$)
DBL_EPSILON	kleinster positiver double-Wert x, für den noch gilt: $1.0+x!=x$ ($\leq 1E-9$)
LDBL_EPSILON	kleinster positiver long double-Wert x, für den noch gilt: $1.0+x!=x$ ($\leq 1E-9$)
FLT_MIN	kleinster normalisierter positiver float-Wert ($\leq 1E-37$)
DBL_MIN	kleinster normalisierter positiver double-Wert ($\leq 1E-37$)
LDBL_MIN	kleinster normalisierter positiver long double-Wert ($\leq 1E-37$)

Tabelle 2.6: Limits für Gleitpunktzahlen (in `<float.h>`)

In `<float.h>` ist zusätzlich eine Konstante definiert, die den Rundungsmodus für Gleitpunktwerte festlegt:

FLT_ROUNDS	1 nicht festgelegt
	0 zu 0 hin
	1 zum nächsten darstellbaren Wert hin
	2 auf <i>+unendlich</i> zu
	3 auf <i>-unendlich</i> zu

Beispiel

Für eine Umsetzung, die sich nach dem IEEE-Standard für binäre Gleitpunkt-Arithmetik richtet, sieht ein Ausschnitt aus `<float.h>` z.B. wie folgt aus:

```
#define FLT_RADIX      2
#define FLT_MANT_DIG   24
#define FLT_EPSILON    1.19209290E-07F
#define FLT_DIG        6
#define FLT_MIN_EXP    -125
#define FLT_MIN        1.17549435E-38F
#define FLT_MIN_10_EXP -37
#define FLT_MAX_EXP    +128
#define FLT_MAX        3.40282347E+38F
#define FLT_MAX_10_EXP +38
#define DBL_MANT_DIG   53
#define DBL_EPSILON    2.2204460492503131E-16
#define DBL_DIG        15
#define DBL_MIN_EXP    -1021
#define DBL_MIN        2.2250738585072014E-308
#define DBL_MIN_10_EXP -307
#define DBL_MAX_EXP    +1024
#define DBL_MAX        1.7976931348623157E+308
#define DBL_MAX_10_EXP +308
```

2.4.5 <limits.h> – Limits für ganzzahlige Datentypen

Diese Headerdatei definiert Grenzwerte¹⁶ für die verschiedenen Ganzzahl-Datentypen. Der dabei in der zweiten Spalte der Tabelle 2.7 angegebene Absolutbetrag dieses Mindestwerts (mit gleichem Vorzeichen) darf von dem ANSI-C-Compiler nicht unterschritten werden.

Konstantenname	geforderter Mindestwert	Beschreibung
CHAR_BIT	8	maximale Bitanzahl für ein Byte
SCHAR_MIN	-127	Minimalwert für signed char
SCHAR_MAX	+127	Maximalwert für signed char
UCHAR_MAX	255	Maximalwert für unsigned char
CHAR_MIN	SCHAR_MIN oder 0	Minimalwert für char
CHAR_MAX	SCHAR_MAX oder UCHAR_MAX	Maximalwert für char
MB_LEN_MAX	1	max. Bytes für Vielbytezeichen
SHRT_MIN	-32767	Minimalwert für short int

Tabelle 2.7: Limits für Ganzzahl-Datentypen (in `<limits.h>`)

16. Jede Definition muß einen konstanten Ausdruck ergeben, welcher geeignet ist, um in einer `#if`-Präprozessorkonstruktion angegeben werden zu können.

Konstantenname	geforderter Mindestwert	Beschreibung
SHRT_MAX	+32767	Maximalwert für short int
USHRT_MAX	65535	Maximalwert für unsigned short
INT_MIN	-32767	Minimalwert für int
INT_MAX	+32767	Maximalwert für int
UINT_MAX	65535	Maximalwert für unsigned int
LONG_MIN	-2147483647	Minimalwert für long int
LONG_MAX	+2147483647	Maximalwert für long int
ULONG_MAX	4294967295	Maximalwert für unsigned long int

Tabelle 2.7: Limits für Ganzzahl-Datentypen (in <limits.h>)

2.4.6 <locale.h> – Internationales C

Diese von ANSI C neu eingeführte Headerdatei versucht, aus C eine internationale Sprache zu machen. Unabhängig von Kulturkreis und Sprache werden C-Schlüsselwörter auch in Zukunft englisch anzugeben sein. Wem diese Aussage nicht behagt, steht es natürlich frei, sich z.B. eine eigene Headerdatei »deutsch.h« zu erstellen:

```
#define solange while
#define wenn if
#define sonst else
```

Eine solche Vorgehensweise erlaubt zwar »deutschgeschriebene« C-Programme, die nur in Verbindung mit dieser Headerdatei als streng ANSI-C-konform gewertet werden können, aber sie würde beispielsweise noch nicht das im Deutschen übliche Komma in gebrochenen Zahlen

```
float zinsen = 7,32
```

unterstützen. Um nun C-Anwendungen vollständig auf einen speziellen Kulturkreis umzustellen, wurde <locale.h> von ANSI C eingeführt. Man stelle sich ein C-Programm vor, das für Textverarbeitung geschrieben wurde, welchem plötzlich ein deutscher Text mit Umlauten vorgelegt wird: Der Aufruf des Makros **isalpha** würde sich bei Umlauten nicht mehr richtig verhalten.

ANSI C schreibt zur Lösung dieses Problems folgendes vor: Jede C-Realisierung muß zumindest die »englische C-Version« beherrschen (z.B. **isalpha** für 26 Buchstaben). Es ist aber erlaubt, daß zusätzlich andere Sprachen und Kulturkreise (von ANSI C **Locale** genannt) unterstützt werden, auf die während der Laufzeit eines Programms umgeschaltet werden kann.

Die Frage ist nun, welche Bereiche von solchen lokalen Eigenheiten betroffen sind:

- Alphabet:
Der chinesische Zeichensatz zeigt sicher kleinere Unterschiede zum dänischen Alphabet auf.

- ▶ Reihenfolge im Alphabet:
In welcher Reihenfolge würde ein Amerikaner die beiden Worte »mußte« und »Müll« sortieren (selbst im deutschen Kulturkreis kann es hier Unterschiede geben).
- ▶ Formatieren von Zahlen und Geldbeträgen:
Der deutschen Schreibweise des Geldbetrags 1.352,70 steht 1,352.70 im Amerikanischen gegenüber.
- ▶ Datum und Zeit:
Die Standard-Funktion **asctime** gibt eine Zeichenkette zurück, welche Abkürzungen für Wochentags- und Monatsnamen enthält. Das Format dieser Rückgabe entspricht in vielen Ländern nicht der dort üblichen Angabe für Datum und Zeit.

Beispiel

- ▶ übliche Datumsformate:

1987-07-14	ISO
14.7.87	Mitteleuropa und Großbritannien
7/14/87	USA
14JUL87	Flugzeiten
Dienstag, 14. Juli 1987	volles deutsches Format
Tuesday, July 14, 1987	volles USA-Format

- ▶ übliche Zeitformate

2:30 PM	USA und Großbritannien
1430	USA-Militär-Format
14h.30	Italienisches Format
14.30	Deutsches Format

Funktion **setlocale**

Umschalten auf eine neue Locale erfolgt durch den Aufruf der in `<locale.h>` definierten Funktion **setlocale**.

`char *setlocale(int categorie, const char *locale)`

Ein Aufruf der Funktion **setlocale** legt entsprechend den Vorgaben aus *categorie* eine neue *locale* für das momentan ablaufende Programm fest; allerdings muß nicht der komplette Satz von lokalen Eigenheiten gegen eine neue Locale ausgetauscht werden, sondern es ist auch möglich, nur Teile hiervon auszutauschen. Dazu werden hier neben dem Makro `NULL` (Nullzeiger-Konstante) noch sechs weitere Makros definiert, welche für das Argument *categorie* beim Aufruf von **setlocale** angegeben werden dürfen¹⁷:

<code>LC_ALL</code>	bisherige wird komplett gegen neue <i>locale</i> ausgetauscht.
<code>LC_COLLATE</code>	hat nur Auswirkungen auf das Verhalten der beiden in <code><string.h></code> definierten

17. Neben diesen Makros darf jeder C-Compiler noch eigene Makros definieren, solange diese mit `LC_G` (*G* steht für Großbuchstabe) beginnen.

	Funktionen <i>strcoll</i> und <i>strxfrm</i> .
LC_CTYPE	hat Auswirkungen auf alle Funktionen in <code><ctype.h></code> (außer <i>isdigit</i> und <i>isxdigit</i>) und auf Funktionen, welche sich mit Vielbytezeichen befassen.
LC_MONETARY	hat Auswirkungen auf das Formatieren von Geldbeträgen (siehe Funktion <i>localeconv</i>).
LC_NUMERIC	legt das Zeichen für den Dezimalpunkt fest.
LC_TIME	beeinflußt das Verhalten der Funktion <i>strtime</i> (siehe <code><time.h></code>).

Falls für das Argument `locale` beim Aufruf dieser Funktion »C« angegeben wird, wie z.B.

```
setlocale(LC_ALL, "C");
```

dann wird die »englische Version« von C gewählt¹⁸, welche immer angeboten werden muß (»kleinster gemeinsamer Nenner aller C-Compiler«).

Falls beispielsweise eine C-Realisierung auch die deutsche Sprache unterstützt, könnte z.B. ein Aufruf wie

```
setlocale(LC_ALL, "deutsch");
```

abgesetzt werden, um ihn deutsch sprechen und verstehen zu lassen.

Ein Aufruf

```
setlocale (LC_ALL, "");
```

bewirkt, daß ein Programm vom implementierungsdefinierten Verhalten einer speziellen Umsetzung Gebrauch machen will: Wenn beispielsweise ein C-Compiler in Brasilien und für den brasilianischen Markt hergestellt wurde, dann kann dieser Aufruf bewirken, daß auf die portugiesische Sprache umgeschaltet wird. Dieser Aufruf veranlaßt also die »Rückkehr eines C-Compilers in seine Heimat«.

Falls die entsprechende Realisierung die für `locale` angegebene Zeichenkette nicht kennt, wird ein NULL-Zeiger von dieser Funktion zurückgegeben und die Locale des Programms bleibt unverändert. Ansonsten wird ein Zeiger auf eine Zeichenkette zurückgegeben, welche die neue Locale für `categorie` darstellt.

Die Angabe eines NULL-Zeigers für `locale` bewirkt, daß ***setlocale*** einen Zeiger auf einen String, der mit `categorie` für die momentane Programm-Locale assoziiert ist, zurückgibt. In diesem Fall wird die Locale des Programms nicht geändert. Der Zeiger auf eine von ***setlocale*** zurückgegebene Zeichenkette kann dann bei nachfolgenden Aufrufen als Argument übergeben werden, um die alte Programm-Locale wieder herzustellen:

```
alt_zustand = setlocale(LC_MONETARY, NULL);
setlocale(LC_MONETARY, "BRASILIEN");
ueberweisung("Rio", datum, ...);
setlocale(LC_MONETARY, alt_zustand);
```

18. Bei jedem Start eines C-Programms wird implizit der Aufruf `setlocale(LC_ALL, "C")` ausgeführt.

Funktion localeconv

Neben der *setlocale*-Funktion wird in dieser Headerdatei noch eine weitere Funktion deklariert:

```
struct lconv *localeconv(void)
```

Die Funktion **localeconv** liefert über die Struktur `lconv` (ebenfalls hier definiert) Werte, die für das Formatieren von numerischen Größenangaben entsprechend der momentanen *Locale* geeignet sind.

Die einzelnen Komponenten der Struktur `lconv` sind in der Tabelle 2.8 angegeben.

Komponente	Bedeutung
char *decimal_point	Dezimalpunktzeichen für das Formatieren von Nicht-Geldbeträgen
char *thousands_sep	Zeichen zum Trennen von Zifferngruppen links vom Dezimalpunkt in formatierten Nicht-Geldbeträgen
char *grouping	String, dessen Elemente die Größe jeder Zifferngruppe in formatierten Nicht-Geldbeträgen anzeigen
char *int_curr_symbol	internationales Währungssymbol, das für momentane Locale gültig ist; die ersten drei Zeichen enthalten das alphabetische internationale Währungssymbol entsprechend ISO 4217, das vierte Zeichen (unmittelbar vor \0) ist das Trennzeichen zwischen Währungssymbol und Geldbetrag
char *currency_symbol	nationales Währungssymbol, das für momentane <i>Locale</i> verwendet wird
char *mon_decimal_point	Dezimalpunktzeichen für das Formatieren von Geldbeträgen
char *mon_thousands_sep	Trennzeichen für die Zifferngruppen vor dem Dezimalpunkt in formatierten Geldbeträgen
char *mon_grouping	String, dessen Elemente die Größe jeder Zifferngruppe in formatierten Geldbeträgen anzeigen
char *positive_sign	String, der verwendet wird, um nicht-negative Geldbeträge anzuzeigen
char *negative_sign	String, der verwendet wird, um negative Geldbeträge anzuzeigen
char int_frac_digits	Zahl der auszugebenden »Nachkommastellen« in einem international formatierten Geldbetrag
char frac_digits	Zahl der auszugebenden »Nachkommastellen« in einem formatierten Geldbetrag
char p_cs_precedes	Wert 1 zeigt an, daß das Währungssymbol (<i>currency_symbol</i>) vor einem nicht-negativen formatierten Geldbetrag steht, Wert 0 zeigt an, daß Währungssymbol hinten steht

Tabelle 2.8: Die Komponenten der Struktur `lconv`

Komponente	Bedeutung
char p_sep_by_space	Wert 1 zeigt an, daß das Währungssymbol durch ein Leerzeichen vom nicht-negativen formatierten Geldbetrag getrennt ist; Wert 0 deutet darauf hin, daß keine Trennung vorliegt
char n_cs_precedes	Wert 1 zeigt an, daß das Währungssymbol vor einem negativen formatierten Geldbetrag steht; Wert 0 zeigt an, daß Währungssymbol hinten steht
char n_sep_by_space	Wert 1 zeigt an, daß das Währungssymbol durch ein Leerzeichen vom negativen formatierten Geldbetrag getrennt ist; Wert 0 deutet darauf hin, daß keine Trennung vorliegt
char p_sign_posn	Wert, der die Position des positiven Vorzeichens für einen nicht-negativen formatierten Geldbetrag anzeigt
char n_sign_posn	Wert, der die Position des negativen Vorzeichens für einen negativen formatierten Geldbetrag anzeigt

Tabelle 2.8: Die Komponenten der Struktur lconv

Als Beispiele führt das ANSI-C-Papier die folgenden vier Länder auf:

Land	Positives Format	Negatives Format	Internationales Format
Italien	L.1.234	-L.1.234	ITL.1.234
Niederlande	F 1.234,56	F -1.234,56	NLG 1.234,56
Norwegen	kr1.234,56	kr1.234,56-	NOK 1.234,56
Schweiz	SFrS.1,234.56	SFrS.1,234.56C	CHF 1,234.56

Für diese vier Länder würde die Funktion **localeconv** die einzelnen Komponenten der Struktur lconv wie folgt besetzen und zurückgeben:

	Italien	Niederlande	Norwegen	Schweiz
nt_curr_symbol	»ITL.«	»NLG«	»NOK«	»CHF«
currency_symbol	»L.«	»F«	»kr«	»SFrS.«
mon_decimal_point	»«	»,«	»,«	»,«
mon_thousands_sep	»,«	»,«	»,«	»,«
mon_grouping	»\3«	»\3«	»\3«	»\3«
positive_sign	»«	»«	»«	»«
negative_sign	»-«	»-«	»-«	»C«
int_frac_digits	0	2	2	2

	Italien	Niederlande	Norwegen	Schweiz
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_spcae	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

Diese Beschreibung von `<locale.h>` soll dem Allgemeinverständnis dienen. Falls der Leser mit einer C-Realisierung arbeitet, die andere Sprachen oder Kulturkreise als die »englische Version« unterstützt und damit auch die Headerdatei `<locale.h>` anbietet, wird eine solche Realisierung mit Sicherheit von einer ausführlichen Beschreibung begleitet.

2.4.7 **<math.h> – Mathematische Funktionen**

Die Headerdatei `<math.h>` deklariert die in Tabelle 2.9 angegebenen mathematischen Funktionen.

Funktion	liefert als Ergebnis
double <i>acos</i> (double x)	Arcuscosinus von x
double <i>asin</i> (double x)	Arcussinus von x
double <i>atan</i> (double x)	Arcustangens von x
double <i>atan2</i> (double y,double x)	Arcustangens von y/x
double <i>ceil</i> (double x)	kleinste ganze Zahl nicht kleiner als x; wird zum Aufrunden verwendet (gelieferte ganze Zahl ist double !!)
double <i>cos</i> (double x)	Cosinus von x
double <i>cosh</i> (double x)	Cosinus hyperbolicus von x
double <i>exp</i> (double x)	e ^x (e steht 2.718281..)
double <i>fabs</i> (double x)	Absolutwert von x
double <i>floor</i> (double x)	größte ganze Zahl nicht größer als x; wird zum Abrunden verwendet (gelieferte ganze Zahl ist double !!)
double <i>fmod</i> (double x, double y)	Gleitpunktrest von x/y (x-i*y, wobei i solche Ganzzahl ist, daß Ergebnis gleiches Vorzeich. wie x und kleineren Betrag als y hat

Tabelle 2.9: Die mathematischen Funktionen aus `<math.h>`

Funktion	liefert als Ergebnis
double <i>frexp</i> (double <i>wert</i> , int * <i>exp</i>)	wandelt <i>wert</i> in normalisierte double-Form $[0.5,1) \cdot 2^{*exp}$ um, wobei Rückgabewert aus Intervall $[0.5,1)$ ist
double <i>ldexp</i> (double <i>x</i> , int <i>exp</i>)	$x \cdot 2^{exp}$
double <i>log</i> (double <i>x</i>)	natürlichen Logarithmus von <i>x</i>
double <i>log10</i> (double <i>x</i>)	Zehnerlogarithmus von <i>x</i>
double <i>modf</i> (double <i>wert</i> , double * <i>iptr</i>)	Nachkommateil von <i>wert</i> . Vorkommateil in * <i>iptr</i> gespeichert
double <i>pow</i> (double <i>x</i> , double <i>y</i>)	x^y
double <i>sin</i> (double <i>x</i>)	Sinus von <i>x</i>
double <i>sinh</i> (double <i>x</i>)	Sinus hyperbolicus von <i>x</i>
double <i>sqrt</i> (double <i>x</i>)	Quadratwurzel von <i>x</i>
double <i>tan</i> (double <i>x</i>)	Tangens von <i>x</i>

Tabelle 2.9: Die mathematischen Funktionen aus <math.h>

Zusätzlich wird in <math.h> eine Konstante definiert, die von den Funktionen zurückgegeben wird, falls der richtige Wert nicht darstellbar ist:

HUGE_VAL sehr großer **double**-Wert¹⁹

Daneben werden hier noch die beiden in <errno.h> definierten Konstanten verwendet:

EDOM zeigt einen Domainfehler an (meist ungültiges Argument)

ERANGE zeigt einen Bereichsfehler an (nicht darstellbarer Wert)

Diese beiden werden hier nur verwendet, definiert sind sie in <errno.h>.

Wenn ein Domainfehler in einer <math.h>-Funktion auftritt, dann ist der Rückgabewert implementierungsdefiniert, und EDOM wird in die globale Variable errno geschrieben.

Wenn ein Bereichsfehler in einer <math.h>-Funktion auftritt, dann wird unterschieden zwischen:

- Überlauf (*Overflow*)
Die entsprechende Funktion liefert den Wert HUGE_VAL mit gleichem Vorzeichen (außer bei **tan**) wie der richtige Wert, und errno wird der Wert ERANGE zugewiesen.
- Unterlauf (*Underflow*)
Die entsprechende Funktion gibt 0 zurück. Ob errno der Wert ERANGE zugewiesen wird oder nicht, ist implementierungsdefiniert.

19. Auf manchen Maschinen kann dieser Wert eine spezielle Kodierung für Unendlichkeit darstellen, wenn die entsprechende Implementierung dies unterstützt.

Das nachfolgende Programm 2.5 (mfunk1.c) ist ein erstes Demonstrationsbeispiel zu den mathematischen Funktionen.

```
#include <stdio.h>
#include <math.h>

int
main(void)
{
    double zahl;
    const double pi = 4*atan(1);

    printf("Gib eine Gleitpunktzahl ein: ");
    scanf("%lf", &zahl);

    printf("\nPI = %.10lf\n", pi);

    printf("Quadratwurzel zu %.4lf ist: %.4lf\n", zahl, sqrt(zahl));
    printf("%.4lf hoch 0.5 ist: %.4lf\n", zahl, pow(zahl,0.5));
    printf("%.4lf hoch -0.5 ist: %.4lf\n", zahl, pow(zahl,-0.5));
    printf("%.4lf hoch 3 ist: %.4lf\n", zahl, pow(zahl,3));
    printf("e hoch %.4lf ist: %.4lf\n", zahl, exp(zahl));
    printf("Natuerl. Logarithmus zu %.4lf ist: %.4lf\n", zahl, log(zahl));
    printf("Zehner-Logarithmus zu %.4lf ist: %.4lf\n", zahl, log10(zahl));

    printf("Cosinus zu %.4lf ist: %.4lf\n", zahl, cos(zahl));
    printf("Cosinus zu PI ist: %.4lf\n", cos(pi));
    printf("Sinus zu %.4lf ist: %.4lf\n", zahl, sin(zahl));
    printf("Sinus zu PI ist: %.4lf\n", sin(pi));
    printf("Tangens zu %.4lf ist: %.4lf\n", zahl, tan(zahl));
    printf("Tangens zu PI ist: %.4lf\n", tan(pi));

    exit(0);
}
```

Programm 2.5 (mfunk1.c): Demonstrationsbeispiel zu mathematischen Funktionen

Nachdem man das Programm 2.5 (mfunk1.c) kompiliert und gelinkt hat

```
cc -o mfunk1 mfunk1.c -lm
```

ergibt sich z.B. der folgende Ablauf:

```
$ mfunk1
```

```
Gib eine Gleitpunktzahl ein: 2.3
```

```
PI = 3.1415926536
```

```
Quadratwurzel zu 2.3000 ist: 1.5166
```

```
2.3000 hoch 0.5 ist: 1.5166
```

```
2.3000 hoch -0.5 ist: 0.6594
```

```
2.3000 hoch 3 ist: 12.1670
```

```
e hoch 2.3000 ist: 9.9742
```

```
Natuerl. Logarithmus zu 2.3000 ist: 0.8329
```

```
Zehner-Logarithmus zu 2.3000 ist: 0.3617
Cosinus zu 2.3000 ist: -0.6663
Cosinus zu PI ist: -1.0000
Sinus zu 2.3000 ist: 0.7457
Sinus zu PI ist: 0.0000
Tangens zu 2.3000 ist: -1.1192
Tangens zu PI ist: -0.0000
$
```

Das nachfolgende Programm 2.6 (mfunk2.c) ist ein weiteres Demonstrationsbeispiel zu den mathematischen Funktionen.

```
#include <stdio.h>
#include <math.h>

int
main(void)
{
    double a, b, c, d,
           vorkomma, nachkomma,
           mantisse;
    int exponent;

    printf("Gib 4 Gleitpunktzahlen durch Komma getrennt ein: ");
    scanf("%lf,%lf,%lf,%lf", &a, &b, &c, &d);

    printf("\nceil(%.4lf) = %.4lf\n", a, ceil(a));
    printf("ceil(%.4lf) = %.4lf\n", b, ceil(b));
    printf("ceil(%.4lf) = %.4lf\n", c, ceil(c));
    printf("ceil(%.4lf) = %.4lf\n", d, ceil(d));

    printf("\nfloor(%.4lf) = %.4lf\n", a, floor(a));
    printf("floor(%.4lf) = %.4lf\n", b, floor(b));
    printf("floor(%.4lf) = %.4lf\n", c, floor(c));
    printf("floor(%.4lf) = %.4lf\n", d, floor(d));

    printf("\nfabs(%.4lf) = %.4lf\n", a, fabs(a));
    printf("fabs(%.4lf) = %.4lf\n", b, fabs(b));
    printf("fabs(%.4lf) = %.4lf\n", c, fabs(c));
    printf("fabs(%.4lf) = %.4lf\n", d, fabs(d));

    printf("\nfmod(%.4lf,%.4lf) = %.4lf\n", b, a, fmod(b,a));
    printf("fmod(%.4lf,%.4lf) = %.4lf\n", d, c, fmod(d,c));

    printf("\n\nWeiter mit Return.....");
    getchar(); getchar();

    printf("\nmodf:\n");
    nachkomma=modf(a, &vorkomma);
    printf("%.4lf = %.0lf + %.4lf\n", a, vorkomma, nachkomma);
    nachkomma=modf(b, &vorkomma);
    printf("%.4lf = %.0lf + %.4lf\n", b, vorkomma, nachkomma);
    nachkomma=modf(c, &vorkomma);
```

```

printf("%.4lf = %.0lf + %.4lf\n", c, vorkomma, nachkomma);
nachkomma=modf(d, &vorkomma);
printf("%.4lf = %.0lf + %.4lf\n", d, vorkomma, nachkomma);

printf("\nfrexp / ldexp:\n");
mantisse=frexp(a, &exponent);
printf("%.4lf = %.4lf * 2 hoch %d (frexp); ", a, mantisse, exponent);
printf("%.4lf * 2 hoch %d = %.4lf (ldexp)\n",
        mantisse, exponent, ldexp(mantisse, exponent));
mantisse=frexp(b, &exponent);
printf("%.4lf = %.4lf * 2 hoch %d (frexp); ", b, mantisse, exponent);
printf("%.4lf * 2 hoch %d = %.4lf (ldexp)\n",
        mantisse, exponent, ldexp(mantisse, exponent));
mantisse=frexp(c, &exponent);
printf("%.4lf = %.4lf * 2 hoch %d (frexp); ", c, mantisse, exponent);
printf("%.4lf * 2 hoch %d = %.4lf (ldexp)\n",
        mantisse, exponent, ldexp(mantisse, exponent));
mantisse=frexp(d, &exponent);
printf("%.4lf = %.4lf * 2 hoch %d (frexp); ", d, mantisse, exponent);
printf("%.4lf * 2 hoch %d = %.4lf (ldexp)\n",
        mantisse, exponent, ldexp(mantisse, exponent));

exit(0);
}

```

Programm 2.6 (**mfunk2.c**): Weiteres Demonstrationsbeispiel zu mathematischen Funktionen

Nachdem man das Programm 2.6 (mfunk2.c) kompiliert und gelinkt hat

```
cc -o mfunk2 mfun2.c -lm
```

ergibt sich z.B. der folgende Ablauf:

\$ mfunk2

Gib 4 Gleitpunktzahlen durch Komma getrennt ein: 17.625, 1526.17, -0.1, 5.2

```

ceil(17.6250) = 18.0000
ceil(1526.1700) = 1527.0000
ceil(-0.1000) = -0.0000
ceil(5.2000) = 6.0000

```

```

floor(17.6250) = 17.0000
floor(1526.1700) = 1526.0000
floor(-0.1000) = -1.0000
floor(5.2000) = 5.0000

```

```

fabs(17.6250) = 17.6250
fabs(1526.1700) = 1526.1700
fabs(-0.1000) = 0.1000
fabs(5.2000) = 5.2000

```

```

fmod(1526.1700,17.6250) = 10.4200
fmod(5.2000,-0.1000) = 0.1000

```


Weiter mit Return.....

modf:

17.6250 = 17 + 0.6250

1526.1700 = 1526 + 0.1700

-0.1000 = -0 + -0.1000

5.2000 = 5 + 0.2000

frexp / ldexp:

17.6250 = 0.5508 * 2 hoch 5 (frexp); 0.5508 * 2 hoch 5 = 17.6250 (ldexp)

1526.1700 = 0.7452 * 2 hoch 11 (frexp); 0.7452 * 2 hoch 11 = 1526.1700 (ldexp)

-0.1000 = -0.8000 * 2 hoch -3 (frexp); -0.8000 * 2 hoch -3 = -0.1000 (ldexp)

5.2000 = 0.6500 * 2 hoch 3 (frexp); 0.6500 * 2 hoch 3 = 5.2000 (ldexp)

\$

2.4.8 <stddef.h> – Standarddefinitionen

Die hier definierten Datentypen und Makros sollten von Programmen, die sich portabel nennen, an den entsprechenden Stellen verwendet werden:

Datentyp `ptrdiff_t`

vorzeichenbehafteter Ganzzahltyp für das Subtraktionsergebnis zweier Zeiger

Datentyp `size_t`

vorzeichenloser Ganzzahltyp für das Ergebnis des **sizeof**-Operators. Meist als Typ für Funktionsargumente verwendet, die Größenangaben repräsentieren, wie z.B.

```
void *malloc(size_t groesse);
```

Datentyp `wchar_t`

ganzzahliger Datentyp, der den ganzen Wertebereich aller vorgegebenen Zeichen (wie z.B. auch ganz spezieller Graphikzeichen) abdecken kann²⁰

Makro `NULL`

Nullzeiger-Konstante (oft als 0, 0L oder (void*)0 definiert)

offsetof(*struktur_typ*, *struktur_komponente*)

liefert das Offset von *struktur_komponente* in *struktur_typ* (in Byte, wobei `size_t` der Rückgabotyp ist). Falls es sich bei der angegebenen *struktur_komponente* um ein Bitfeld handelt, dann ist das Verhalten undefiniert. Für C-Tüftler: `offsetof(s_typ, s_komp)` könnte z.B. mit

```
(size_t)&(((s_typ *)0)->s_komp)
```

definiert sein.

20. Dieser Datentyp wurde eingeführt, um auch asiatische Zeichensätze, welche oft mehr als 10000 Zeichen umfassen, darstellen zu können.

2.4.9 <stdlib.h> – Allgemein nützliche Funktionen

Diese Headerdatei ist der Sammelplatz für alle Funktionen, die keiner der anderen Kategorien (Headerdateien) zugeordnet werden können.

Es sind hier unter anderem auch die beiden in <stddef.h> vorhandenen Datentypen `size_t` und `wchar_t` und die NULL-Konstante definiert. Daneben sind noch die folgenden beiden Datentypen

<code>div_t</code>	Strukturtyp für den Rückgabewert der Funktion <i>div</i>
<code>ldiv_t</code>	Strukturtyp für den Rückgabewert der Funktion <i>ldiv</i>

und die folgenden vier Konstanten definiert.

`EXIT_SUCCESS`

Exit-Status für erfolgreiche Beendigung. Diese Konstante wird meist als Argument für die Funktion *exit* verwendet.

`EXIT_FAILURE`

Exit-Status für nicht erfolgreiche Beendigung. Diese Konstante wird meist als Argument für die Funktion *exit* verwendet.

`RAND_MAX`

maximaler Rückgabewert für Funktion ***rand***

`MB_CUR_MAX`

maximale Byteanzahl für Vielbyte-Zeichen (niemals > `MB_LEN_MAX`)

Nachfolgend werden die in <stdlib.h> deklarierten Funktionen kurz vorgestellt. Dabei werden sie nicht alphabetisch aufgezählt, sondern entsprechend ihrer Zusammengehörigkeit gruppiert.

Allokieren und Freigeben von Speicherplatz

`void *malloc(size_t groesse);`

allokiert (reserviert) einen Speicherbereich von `groesse` Byte.

`void *calloc(size_t anzahl, size_t groesse);`

allokiert einen Speicherbereich, der groß genug ist, um `anzahl` Objekte von `groesse` Byte aufzunehmen. Alle Byte in diesem Speicherbereich werden mit dem Wert 0 initialisiert.

`void *realloc(void *zeiger, size_t groesse);`

verändert die Größe des Speicherbereichs, auf den `zeiger` zeigt, nach `groesse`. Der Inhalt dieses neuen Objekts bleibt unverändert bis zur kleineren der alten oder neuen Größe. `realloc(NULL, groesse)` ist identisch zu `malloc(groesse)`.

`void free(void *zeiger);`

bewirkt die Freigabe des Speicherbereichs, auf den `zeiger` zeigt.

Die Funktionen **malloc**, **calloc**, **realloc** und **free** sind in Kapitel 9.4 ausführlich beschrieben.

Environment-Variablen

char ***getenv**(const char **name*);

durchsucht die Environment-Tabelle des entsprechenden Betriebssystems nach einer Environment-Variable mit Namen *name* und liefert den Inhalt dieser Environment-Variablen als Rückgabewert. Diese Funktion wird in Kapitel 9.3 detailliert beschrieben.

Programmbeendigung

int **atexit**(void (**func*) (void));

Diese Funktion trägt die Funktion, auf die *func* zeigt, in die Liste von Funktionen ein, die vor einer normalen Beendigung des Programms noch aufzurufen sind. In Kapitel 9.2 wird diese Funktion genauer beschrieben.

void **exit**(int *status*);

bewirkt eine »normale Programmbeendigung«. In Kapitel 9.2 wird diese Funktion genauer beschrieben.

void **abort**(void);

bewirkt einen abnormalen Programmabbruch. In Kapitel 13 wird diese Funktion genauer beschrieben.

int **system**(const char **string*);

Diese Funktion übergibt das Kommando *string* an das entsprechende Betriebssystem, damit dieses vom zugehörigen Kommandoprozessor²¹ interpretiert und ausgeführt wird. Diese Funktion, die in Kapitel 10.6 genauer beschrieben wird, erlaubt es, von C-Programmen aus Betriebssystem-Kommandos ausführen zu lassen, wie z.B.:

```
system("dir/p");    unter MSDOS
system("ls -al");   unter Unix
```

Zufallszahlen

int **rand**(void);

liefert als Funktionswert eine Pseudo-Zufallszahl aus dem Bereich 0 bis RAND_MAX (muß >= 32767 sein).

void **srand**(unsigned int *startwert*);

Diese Funktion verwendet das Argument *startwert*, um einen Startpunkt für eine neue Folge von Pseudo-Zufallszahlen zu setzen. Jeder nachfolgende Aufruf der Funktion **rand** liefert dann die nächste Zahl aus dieser Folge. Würde **srand** mit gleichem

21. `command.com` unter MSDOS oder die Shell (Bourne-, C-, Korn-Shell, ...) unter Unix

startwert wieder aufgerufen, dann würde mit den darauffolgenden **rand**-Aufrufen die gleiche Folge von Pseudo-Zufallszahlen nochmals generiert.

Wird **rand** aufgerufen, bevor **srand** aufgerufen wurde, so wird die gleiche Folge von Pseudo-Zufallszahlen erzeugt, wie wenn zuvor `srand(1)` aufgerufen worden wäre.

Das folgende Programm 2.7 (`rand.c`) ist ein Demonstrationsbeispiel zu den Funktionen **rand** und **srand**.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

long int wuerfel[6] = { 0L, 0L, 0L, 0L, 0L, 0L };

int
main(void)
{
    float          soll = 100.0 / 6.0,
                  ein_prozent,
                  prozent;

    long int  i, anzahl;

    /* Zufallszahlengenerator auf einen zufaelligen Startwert setzen */
    srand(time(NULL));
    /* noch besser unter Linux/Unix: srand(time(NULL) + getpid()); */

    printf("Wieoft ist Wuerfel zu werfen: ");
    scanf("%ld", &anzahl);
    ein_prozent = anzahl / 100.0;
    for (i=1 ; i<=anzahl ; i++)
        wuerfel[ rand() % 6 ]++;

    printf("%6.6s | %12.12s | %10.10s | %16.16s |\n",
           "Zahl", "Gewuerfelt", "Prozent", "Soll-Abweichung");
    printf("-----\n");
    for (i=0 ; i<6 ; i++) {
        prozent = wuerfel[i]/ein_prozent;
        printf("%6ld | %12ld | ", i+1, wuerfel[i]);
        printf("%10.2f | ", prozent);
        printf("%16.2f |\n", prozent-soll);
    }

    exit(0);
}
```

Programm 2.7 (`rand.c`): Simulation eines Würfels

Nachdem man dieses Programm 2.7 (`rand.c`) kompiliert und gelinkt hat

```
cc -o rand rand.c
```

können sich z.B. die folgenden Abläufe ergeben:

\$ rand

Wieoft ist Wuerfel zu werfen: 100

Zahl	Gewuerfelt	Prozent	Soll-Abweichung

1	22	22.00	5.33
2	15	15.00	-1.67
3	13	13.00	-3.67
4	13	13.00	-3.67
5	20	20.00	3.33
6	17	17.00	0.33

\$ rand

Wieoft ist Wuerfel zu werfen: 1000000

Zahl	Gewuerfelt	Prozent	Soll-Abweichung

1	165963	16.60	-0.07
2	166476	16.65	-0.02
3	167276	16.73	0.06
4	166603	16.66	-0.01
5	166868	16.69	0.02
6	166814	16.68	0.01

\$

Absolutwerte

```
long int labs(long int j);
```

```
int abs(int j);
```

Diese beiden Funktionen liefern den Absolutwert zum ganzzahligen Argument *j*. Falls das Ergebnis nicht dargestellt werden kann, liegt undefiniertes Verhalten vor. So kann z.B. auf einer Maschine, die mit Zweierkomplement arbeitet, der Absolutwert der größten negativen Zahl nicht dargestellt werden. Diese Funktionen wurden nicht in `<math.h>` untergebracht, da sie dort die einzigen Funktionen gewesen wären, die keine **double**-Arithmetik durchführen.

Konvertierung von Strings in numerische Werte

```
double atof(const char *string);
```

wandelt eine Zahl, die als *string* gespeichert ist, in einen **double**-Wert um, den sie als Funktionswert liefert. Außer dem Verhalten im Fehlerfall ist diese Funktion äquivalent zu `strtod(string, (char **)NULL)`.

```
int atoi(const char *string);
```

wandelt eine Zahl, die als *string* gespeichert ist, in einen **int**-Wert um, den sie als Funktionswert zurückliefert. Außer dem Verhalten im Fehlerfall ist diese Funktion äquivalent zu `(int)strtol(string, (char **)NULL, 10)`.

```
long int atol(const char *string);
```

wandelt eine Zahl, die als *string* gespeichert ist, in einen **long int**-Wert um, den sie als Funktionswert zurückliefert. Außer dem Verhalten im Fehlerfall ist diese Funktion äquivalent zu `strtol(string, (char **)NULL, 10)`.

```
double strtod(const char *string, char **end_zeig);
```

Die Funktion **strtod** (*string to double*) wandelt eine Zahl, die als `string` gespeichert ist, in einen **double**-Wert um und liefert diesen als Funktionswert.

Falls für `end_zeig` kein `NULL`-Zeiger übergeben wurde, wird nach einer erfolgreichen Umwandlung die Adresse eines nicht konvertierbaren Rests im Zeiger abgelegt, auf den `end_zeig` zeigt.

Bei einer erfolgreichen Umwandlung liefert **strtod** die durch Umwandlung erhaltene Gleitpunktzahl, andernfalls den Wert 0.

```
long strtol(const char *string, char **end_zeig, int basis);
```

```
unsigned long strtoul(const char *string, char **end_zeig, int basis);
```

Die Funktionen **strtol** (*string to long*) und **strtoul** (*string to unsigned long*) wandeln eine Zahl, die als `string` gespeichert ist, in einen **long**- bzw. **unsigned long**-Wert um und liefern diesen als Funktionswert. `basis` legt dabei die Basis des Zahlensystems fest, in das diese Zahl umzuwandeln ist.

Falls für `end_zeig` kein Nullzeiger übergeben wurde, wird nach einer erfolgreichen Umwandlung die Adresse eines nicht konvertierbaren Rests im Zeiger abgelegt, auf den `end_zeig` zeigt.

Bei einer erfolgreichen Umwandlung liefern **strtol** bzw. **strtoul** die durch Umwandlung erhaltene ganze Zahl, andernfalls den Wert 0.

Das folgende Programm 2.8 (`strtod.c`) demonstriert an der Funktion **strtod** die Verwendung der drei Funktionen **strtod**, **strtol** und **strtoul**.

```
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    double   zahl;
    char     string[100];
    char     *rest, zeichk[100];

    printf("Gib einen String ein: ");
    scanf("%s", string);
    rest = zeichk;
    zahl = strtod(string, &rest);
    if (string == rest)
        printf("%s ist keine erlaubte Gleitpunktzahl\n", string);

    printf("%lg (Gleitpunktzahl) / %s (Rest)\n", zahl, rest);

    exit(0);
}
```

*Programm 2.8 (**strtod.c**): Demonstrationsbeispiel zur Funktion **strtod***

Nachdem man dieses Programm 2.8 (`strtod.c`) kompiliert und gelinkt hat

```
cc -o strtod strtod.c
```

können sich z.B. die folgenden Abläufe ergeben:

```
$ strtod
Gib einen String ein: 1e6million
1000000.000000 (Gleitpunktzahl) / million (Rest)
$ strtod
Gib einen String ein: 3.1415pi
3.141500 (Gleitpunktzahl) / pi (Rest)
$ strtod
Gib einen String ein: -1232.78Kontoauszug
-1232.780000 (Gleitpunktzahl) / Kontoauszug (Rest)
$ strtod
Gib einen String ein: 1.2*3.4
1.200000 (Gleitpunktzahl) / *3.4 (Rest)
$ strtod
Gib einen String ein: zwei3vier
zwei3vier ist keine erlaubte Gleitpunktzahl
0.000000 (Gleitpunktzahl) / zwei3vier (Rest)
$
```

Quotient und Rest einer Division

```
div_t div(int zaehler, int nenner);
```

```
ldiv_t ldiv(long int zaehler, long int nenner);
```

Diese beiden Funktionen berechnen den Quotienten und Rest der Division `zaehler/nenner`. Wenn die Division ungenau ist, dann ergibt sich als Quotient der Betrag der Ganzzahl, welche kleiner als der Betrag des mathematischen Quotienten ist. Der Rückgabetyt `div_t` ist eine Struktur, welche die folgenden beiden Komponenten enthält:

```
int quot; /* Quotient */
int rem;  /* Rest      */
```

und der Rückgabetyt `ldiv_t` ist eine Struktur, welche die folgenden beiden Komponenten enthält:

```
long int quot; /* Quotient */
long int rem;  /* Rest      */
```

Wenn das Ergebnis nicht dargestellt werden kann²², dann liegt undefiniertes Verhalten vor, ansonsten muß folgendes gelten:

22. Z.B. »Division durch 0« ergibt undefiniertes Verhalten und bewirkt nicht das Setzen von `errno` auf `EDOM`. Eine Abfrage auf `nenner != 0` vor dem Aufruf einer diesen beiden Funktionen ist deshalb ratsam.

`quot * nenner + rest = zaehler`

Das folgende Programm 2.9 (`div.c`) zeigt, welche Vorzeichen jeweils aus den möglichen Vorzeichen-Kombinationen von `zaehler` und `nenner` bei der Funktion ***div*** resultieren. Dasselbe gilt natürlich auch für die Funktion ***ldiv***.

```
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    div_t pp = div(20,7),
          np = div(-20,7),
          pn = div(20,-7),
          nn = div(-20,-7);

    printf(" 20 div  7 = %2d Rest %2d\n", pp.quot, pp.rem);
    printf("-20 div  7 = %2d Rest %2d\n", np.quot, np.rem);
    printf(" 20 div -7 = %2d Rest %2d\n", pn.quot, pn.rem);
    printf("-20 div -7 = %2d Rest %2d\n", nn.quot, nn.rem);

    exit(0);
}
```

Programm 2.9 (div.c): Demonstrationsbeispiel zur Funktion div

Nachdem man dieses Programm 2.9 (`div.c`) kompiliert und gelinkt hat

```
cc -o div div.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ div
 20 div  7 =  2 Rest  6
-20 div  7 = -2 Rest -6
 20 div -7 = -2 Rest  6
-20 div -7 =  2 Rest -6
$
```

Binäre Suche und Quicksort

```
void *bsearch(const void *such_zeig, const void *start_addr,
              size_t anzahl, size_t groesse,
              int (*vergleichs_routine)(const void *, const void *))
```

Die Funktion ***bsearch*** dient der binären Suche. Sie durchsucht ein Array mit `anzahl` Elementen (`start_addr[0]`, ... , `start_addr[anzahl-1]`) nach einem Element, das dem Objekt entspricht, auf das `such_zeig` zeigt. Die Größe jedes einzelnen Elements wird mit Parameter `groesse` festgelegt. Die Inhalte des entsprechenden Arrays müssen in aufsteigender Reihenfolge sortiert sein, entsprechend dem Sortierkriterium, das von der Vergleichsfunktion `vergleichs_routine` verwendet wird. Diese vom Aufrufer erstellte Vergleichs-

funktion wird mit zwei Argumenten, die auf die zu vergleichenden Objekte (1. Argument: `such_zeig`, 2. Argument: `Arrayelement`) zeigen, aufgerufen. Die entsprechende Vergleichsfunktion muß zurückgeben:

- eine negative Zahl, wenn `*such_zeig < *argument2`
- 0, wenn `*such_zeig == *argument2`
- eine positive Zahl, wenn `*such_zeig > *argument2`

Falls das gesuchte Arrayelement gefunden wird, wird ein Zeiger auf das gefundene Element, andernfalls wird ein NULL-Zeiger zurückgegeben. Wenn mehrere Arrayelemente gleich sind, so ist nicht festgelegt, welches von diesen ausgewählt wird.

Das folgende Programm 2.10 (`bsearch.c`) demonstriert die Anwendung der Funktion ***bsearch***, indem es zunächst eine Monatszahl einliest, dann mit Hilfe von ***bsearch*** den zu dieser Monatszahl gehörigen Namen in einem zuvor initialisierten Array sucht, bevor es diesen Namen ausgibt.

```
#include <stdio.h>
#include <stdlib.h>

#define ANZAHL(array) (size_t) (sizeof(array) / sizeof(array[0]))

typedef struct {
    int mon_zahl;
    char mon_name[10];
} mon_element;

mon_element monate[12] = {
    { 1, "Januar" }, { 2, "Februar" }, { 3, "Maerz" },
    { 4, "April" }, { 5, "Mai" }, { 6, "Juni" },
    { 7, "Juli" }, { 8, "August" }, { 9, "September" },
    { 10, "Oktober" }, { 11, "November" }, { 12, "Dezember" }
};

/*----- vergleichs_fkt -----*/
int vergleichs_fkt(int *gesucht_zgr, mon_element *monat_zgr)
{
    return(*gesucht_zgr - monat_zgr->mon_zahl);
}

/*----- suche -----
    Diese Funktion ruft bsearch auf, um im Array 'monate'
    das Element mit Monatszahl 'monats_zahl' zu finden */
char *suche(int monats_zahl)
{
    mon_element *such_monat =
        bsearch(&monats_zahl, monate,
                ANZAHL(monate), (size_t) sizeof(monate[0]),
                &vergleichs_fkt);
    return(such_monat->mon_name);
}
```

```

/*----- main -----*/
int
main(void)
{
    int monat_zahl;

    while (1) {
        printf("Gib eine Monatszahl (Unerlaubte bewirkt Abbruch) ein: ");
        scanf("%d", &monat_zahl);
        if (monat_zahl < 1 || monat_zahl > 12) {
            break;
        }
        printf("      ----- %s -----\\n", suche(monat_zahl));
    }

    exit(0);
}

```

Programm 2.10 (bsearch.c): Demonstrationsbeispiel zur Funktion bsearch

Nachdem man dieses Programm 2.10 (bsearch.c) kompiliert und gelinkt hat

```
cc -o bsearch bsearch.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ bsearch
Gib eine Monatszahl (Unerlaubte bewirkt Abbruch) ein: 3
      ----- Maerz -----
Gib eine Monatszahl (Unerlaubte bewirkt Abbruch) ein: 7
      ----- Juli -----
Gib eine Monatszahl (Unerlaubte bewirkt Abbruch) ein: 12
      ----- Dezember -----
Gib eine Monatszahl (Unerlaubte bewirkt Abbruch) ein: 6
      ----- Juni -----
Gib eine Monatszahl (Unerlaubte bewirkt Abbruch) ein: 0
$

```

```

void qsort(void *array, size_t anzahl, size_t groesse,
           int (*vergl_funktion)(const void *, const void *));

```

Die Funktion **qsort** dient dem Quicksort von Hoare. Sie sortiert ein Array mit `anzahl` Elementen (in aufsteigender Form). Das Array beginnt bei `array`, und jedes Arrayelement (`array[0]...array[anzahl-1]`) hat eine Größe von `groesse` Bytes.

Das Sortierkriterium wird durch die Funktion `*vergl_funktion` festgelegt. Diese Vergleichsfunktion wird mit zwei Argumenten, die auf die zu vergleichenden Objekte zeigen, aufgerufen. Die entsprechende Vergleichsfunktion verhält sich wie **strcmp**, wo der Rückgabewert

- ▶ eine negative Zahl ist, wenn `*argument1 < *argument2`,
- ▶ 0, wenn `*argument1 == *argument2`,
- ▶ eine positive Zahl, wenn `*argument1 > *argument2`.

Das folgende Programm 2.11 (qsort.c), das den Inhalt einer Textdatei liest und alle Zeilen dieser Datei sortiert wieder ausgibt, demonstriert die Anwendung der Funktion **qsort**. Der Name der zu sortierenden Textdatei ist auf der Kommandozeile anzugeben.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define ZEIL_LAENG 200
#define MAX_ZEILEN 1000

/*----- string_vergl -----*/
int string_vergl(char **z1, char **z2)
{
    return( strcmp(*z1, *z2) );
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    FILE *dz;
    int  anzahl, i=0;
    char puffer[200], *zeile[MAX_ZEILEN];

    if (argc != 2) {
        fprintf(stderr, "Richtiger Aufruf: %s <dateiname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((dz=fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "Datei %s konnte nicht eroeffnet werden\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    /* Uebertragen des ganzen Dateiinhalts in das Zeichenketten-Array */
    /* zeile, um dann spaeter qsort auf dieses Array anzuwenden */
    while (fgets(puffer, ZEIL_LAENG, dz) != NULL) {
        char *zeiger = puffer;
        if ((zeile[i]=malloc(strlen(zeiger)+1)) == NULL) {
            fprintf(stderr, "Speicherplatzmangel in der %d. Zeile "
                          "aufgetreten\n", i+1);
            exit(EXIT_FAILURE);
        }
        strcpy(zeile[i], zeiger);
        if (++i >= MAX_ZEILEN) {
            fprintf(stderr, "Es ist nur moeglich, Dateien mit maximal "
                          "%d Zeilen zu sortieren\n", MAX_ZEILEN);
            exit(EXIT_FAILURE);
        }
    }
}
```

```

    anzahl = i;
    qsort(zeile, anzahl, sizeof(zeile[0]), &string_vergl);
    for (i=0 ; i<anzahl ; i++)
        printf("%s", zeile[i]);

    exit(0);
}

```

Programm 2.11 (*qsort.c*): Sortieren einer Datei

Vielbytezeichen

int **mblen**(const char *vb_zeig, size_t n);

Wenn vb_zeig kein NULL-Zeiger ist, so liefert diese Funktion die Anzahl von Bytes, aus denen sich das Vielbytezeichen, auf das vb_zeig zeigt, zusammensetzt. Diese Funktion ist äquivalent zu

```
mbtowc( (wchar_t *)0, vb_zeig, n );
```

int **mbtowc**(wchar_t *pwc, const char *vb_zeig, size_t n);

konvertiert ein Vielbytezeichen nach wchar_t.

int **wctomb**(char *vb_zeig, wchar_t wchar);

konvertiert ein wchar_t-Zeichen in ein Vielbytezeichen.

size_t **mbstowcs**(wchar_t *pwcs, const char *vb_zeig, size_t n);

konvertiert eine Folge von Vielbytezeichen aus dem Speicherplatz vb_zeig in den Datentyp wchar_t und speichert die entsprechenden Codes (nicht mehr als n) an die Adresse pwcs. Jedes einzelne Vielbytezeichen wird hierbei so konvertiert, als ob die Funktion **mbtowc** aufgerufen würde.

size_t **wcstombs**(char *vb_zeig, const wchar_t *pwcs, size_t n);

konvertiert eine Folge von Codes aus dem Speicherplatz pwcs in eine Folge von entsprechenden Vielbytezeichen (nicht mehr als n) und schreibt diese an die Adresse vb_zeig. Jeder einzelne Code wird konvertiert, als ob die Funktion **wctomb** aufgerufen würde.

Neben den hier angegebenen Funktionen darf jede C-Realisierung noch eigene hinzufügen, allerdings legt ANSI C fest, daß die Namen dieser zusätzlichen Funktionen dann mit *strk* (*k* steht für Kleinbuchstabe) beginnen.

2.4.10 <string.h> – Umgang mit Zeichenketten

Diese Headerdatei definiert ein weiteres Mal den bereits in <stddef.h> definierten Datentyp size_t und die ebenfalls dort definierte NULL-Zeigerkonstante.

Die hier deklarierten Funktionen sind geeignet, um Zeichenketten und Byte-Arrays zu analysieren, zu manipulieren oder zu kopieren. Das allgemeine Ziel von ANSI C ist es, äquivalente Möglichkeiten für drei unterschiedliche Typen von Byteketten zur Verfügung zu stellen:

- ▶ \0 abgeschlossene Zeichenketten. Die Namen der hierfür zuständigen Funktionen beginnen mit **str..**
- ▶ \0 abgeschlossene Zeichenketten mit maximaler Länge. Die Namen der hierfür zuständigen Funktionen beginnen mit **strn..**
- ▶ Byteketten einer bestimmten Länge²³. Die Namen der hierfür zuständigen Funktionen beginnen mit **mem..**

Folgende Funktionen sind nun in `<string.h>` deklariert:

`void *memchr(const void *adress, int such_zeich, size_t n);`

sucht das erste Vorkommen von `such_zeich` in den ersten `n` Zeichen des Speicherbereichs, auf den `adress` zeigt.

Diese Funktion gibt entweder die Adresse des gefundenen Zeichens zurück oder einen NULL-Zeiger, falls das Zeichen `such_zeich` nicht gefunden werden konnte.

`int memcmp(const void *adress1, const void *adress2, size_t n);`

vergleicht die ersten `n` Zeichen des Speicherbereichs, auf den `adress1` zeigt, mit den ersten `n` Zeichen des Speicherbereichs, auf den `adress2` zeigt.

Diese Funktion liefert als Funktionswert eine

- ▶ negative Zahl, wenn Bytekette von `adress1` < Bytekette von `adress2`,
- ▶ 0, wenn Bytekette von `adress1` == Bytekette von `adress2`,
- ▶ positive Zahl, wenn Bytekette von `adress1` > Bytekette von `adress2`.

Der Funktionswert entsteht als Differenz aus den beiden ersten nicht übereinstimmenden Zeichen in den Speicherbereichen `adress1` und `adress2`.

`void *memcpy(void *ziel, const void *quelle, size_t n);`

kopiert `n` Zeichen vom Speicherplatz, auf den `quelle` zeigt, in den Speicherbereich, auf den `ziel` zeigt. Falls die beiden `n`-byte langen Speicherbereiche sich überlappen, dann ist das Verhalten undefiniert (siehe auch **memmove**). **memcpy** liefert die Adresse `ziel` als Funktionswert.

`void *memmove(void *ziel, const void *quelle, size_t n);`

kopiert `n` Zeichen vom Speicherplatz, auf den `quelle` zeigt, in den Speicherbereich, auf den `ziel` zeigt. Im Gegensatz zu **memcpy** garantiert diese Funktion bei Überlappung der beiden Speicherbereiche einen korrekten Kopiervorgang. Wenn also *Sicherheit vor Schnelligkeit* geht, dann ist diese Funktion zu verwenden. Wenn man einen schnelleren, dafür aber unsicheren Kopiervorgang bevorzugt oder aber sicher ist, daß sich die beiden Speicherbereiche nicht überlappen, dann ist **memcpy** die richtige Funktion. **memmove** liefert die Adresse `ziel` als Funktionswert.

23. Inhalt der Bytes wird nicht interpretiert; somit wird nicht wie bei Zeichenketten \0 als Ende-Kennzeichnung ausgelegt.

Das folgende Programm 2.12 (`memmove.c`) ist ein Demonstrationsbeispiel zum Verhalten der Funktion ***memmove*** bei überlappenden Speicherbereichen.

```
#include <string.h>
#include <stdio.h>

char string[20]="pferdaepfel";
char *string1, *string2;

int
main(void)
{
    string1 = string;
    string2 = string1+2;
    printf("%s %s\n", string1, string2);
    memmove(string2, string1, 12);
    printf("%s %s\n", string1, string2);
}
```

Programm 2.12 (*memmove.c*): Demonstrationsbeispiel zur Funktion *memmove*

Nachdem man dieses Programm 2.12 (`memmove.c`) kompiliert und gelinkt hat

```
cc -o memmove memmove.c
```

ergibt sich der folgende Ablauf:

```
$ memmove
pferdaepfel erdaepfel
pfpferdaepfel pferdaepfel
$
```

```
void *memset(void *adress, int zeich, size_t n);
```

schreibt den Wert von `zeich` in jedes der ersten `n` Zeichen des Speicherbereichs mit Adresse `adress`. ***memset*** liefert die Adresse `adress` als Funktionswert.

Aufrufbeispiele sind

```
memset(striche, '-', 100);
memset(zeich_array, ' ', 2000);
memset(int_array, 0, 100*sizeof(int));
```

```
char *strcat(char *kett1, const char *kett2);
```

kopiert die Zeichenkette `kett2` (einschließlich abschließendes `\0`) an das Ende der Zeichenkette `kett1`, wobei das erste Zeichen von `kett2` das abschließende `\0` von `kett1` überschreibt. Falls die beiden Zeichenketten `kett1` und `kett2` sich überlappen, dann ist das Verhalten undefiniert.

strcat liefert als Funktionswert den Zeiger `kett1` auf den Anfang der gesamten Zeichenkette.

```
char *strchr(const char *kett, int such_zeich);
```

sucht das erste Vorkommen von `such_zeich` in der Zeichenkette `kett`. Das abschließende `\0` wird als Teil der Zeichenkette angesehen.

strchr gibt entweder die Adresse des gefundenen Zeichens zurück, oder einen NULL-Zeiger, falls das Zeichen `such_zeich` nicht in der Zeichenkette `kett` vorkommt.

int **strcmp**(const char *kett1, const char *kett2);
vergleicht die beiden Zeichenketten `kett1` und `kett2` byteweise und liefert einen

- positiven Wert, wenn `kett1 > kett2`,
- negativen Wert, wenn `kett1 < kett2`,
- 0, wenn `kett1` und `kett2` völlig gleich sind.

Der Funktionswert ergibt sich aus der Differenz der beiden ersten nicht übereinstimmenden Zeichen in `kett1` und `kett2`.

int **strcoll**(const char *kett1, const char *kett2);
verhält sich genau wie **strcmp**, außer daß lokalspezifische Vergleichsregeln (durch die categorie `LC_COLLATE` in der **setlocale** Funktion festgelegt) angewendet werden.

char **strcpy**(char *ziel, const char *quelle);
kopiert die Zeichenkette `quelle` (einschließlich `\0`) in den Speicherbereich, auf den `ziel` zeigt. Falls dieser Kopiervorgang auf Objekte angewendet wird, die sich gegenseitig überlappen, dann ist das Verhalten undefiniert.
strcpy liefert den Zeiger `ziel` als Funktionswert.

int **strcspn**(const char *kett1, const char *kett2);
berechnet die Länge der Teilzeichenkette in `kett1` (von Anfang an), die keine Zeichen aus `kett2` enthält. Die Länge dieser Teilzeichenkette wird als Funktionswert zurückgegeben.

char ***strerror**(int fehler_nr);
liefert die Adresse der zu einer `fehler_nr` gehörigen Fehlermeldung (dargestellt als Zeichenkette).

size_t **strlen**(const char *zeichk);
liefert die Länge der Zeichenkette `zeichk` (ohne abschließendes `\0`).

char ***strncat**(char *kett1, const char *kett2, size_t n);
kopiert von der Zeichenkette `kett2` nicht mehr als `n` Zeichen an das Ende der Zeichenkette `kett1`²⁴. Ein abschließendes `\0` wird immer an das Ende der so zusammengehängten Zeichenkette geschrieben. Somit ergibt sich als Zeichenzahl für die neu entstandene Zeichenkette:

```
if (strlen(kett2) > n)
    strlen(kett1)+n+1    /* + 1 für abschließendes \0 */
else
    strlen(kett1)+strlen(kett2)+1    /* + 1 für abschließendes \0 */
```

24. Erstes Zeichen von `kett2` überschreibt das abschließende `\0`.

Als Funktionswert liefert *strncat* den Zeiger *kett1* auf den Anfang der gesamten zusammengehängten Zeichenkette. Falls sich die beiden Zeichenketten *kett1* und *kett2* überlappen, dann liegt undefiniertes Verhalten vor.

int **strncmp**(const char **kett1*, const char **kett2*, size_t *n*);
vergleicht bis zu *n* Zeichen der beiden Zeichenketten *kett1* und *kett2* byteweise und liefert als Funktionswert:

- positiven Wert, wenn *kett1* > *kett2*,
- negativen Wert, wenn *kett1* < *kett2*,
- 0, wenn *kett1* und *kett2* völlig gleich sind.

Es ist hier zu beachten, daß nur bis zu *n* Zeichen in den beiden Zeichenketten verglichen werden. Der Funktionswert ergibt sich aus der Differenz der beiden ersten nicht übereinstimmenden Zeichen in *kett1* und *kett2*.

char ***strncpy**(char **kett1*, const char **kett2*, size_t *n*);
kopiert nicht mehr als *n* Zeichen aus *kett2* in die Zeichenkette *kett1*. Falls dieser Kopiervorgang auf sich gegenseitig überlappende Zeichenketten angewendet wird, dann ist das Verhalten undefiniert. Wenn die Länge von *kett2* kleiner als *n* Zeichen ist, dann wird in der Zeichenkette *kett1* für die fehlenden Zeichen \0 angehängt. **strncpy** liefert den Zeiger *kett1* als Rückgabewert. **Vorsicht:** wenn die Zeichenkette *kett2* länger als *n* Zeichen ist, wird kein \0 angehängt.

char ***strpbrk**(const char **kett1*, const char **kett2*);
sucht in *kett1* das erste Vorkommen eines Zeichens aus *kett2* und liefert dann entweder die Adresse des gefundenen Zeichens oder einen NULL-Zeiger, falls kein Zeichen aus *kett2* in *kett1* vorkommt.

Das folgende Programm 2.13 (*strpbrk.c*), das die Vokale in einer Datei zählt, ist ein Demonstrationsbeispiel zur Funktion **strpbrk**.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *vokale = "aeiou";

int
main(void)
{
    unsigned long int   vokal_zahl=0;
    FILE                *dz;
    char                dateiname[20], zeile[1000], *zeiger;

    printf("Welche Datei ? ");
    scanf("%s", dateiname);
```



```

if ((dz=fopen(dateiname,"r")) == NULL) {
    printf("Datei %s kann nicht geoeffnet werden\n", dateiname);
    exit(EXIT_FAILURE);
}
while (fgets(zeile, 1000, dz) != NULL) {
    zeiger = zeile;
    while ((zeiger = strpbrk(zeiger,vokale)) != NULL) {
        vokal_zahl++;
        zeiger++;
    }
}
printf("Datei %s enthaelt %ld Vokale\n", dateiname, vokal_zahl);
exit(0);
}

```

Programm 2.13 (strpbrk.c): Zählen der Vokale in einer Datei

Nachdem man dieses Programm 2.13 (strpbrk.c) kompiliert und gelinkt hat

```
cc -o strpbrk strpbrk.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ strpbrk
Welche Datei ? strpbrk.c
Datei strpbrk.c enthaelt 139 Vokale
$

```

char ***strrchr**(const char *zeichk, int zeich);

sucht in *zeichk* das letzte Vorkommen von *zeich*. Das abschließende `\0` wird hierbei als Bestandteil der Zeichenkette *zeichk* betrachtet.

Diese Funktion liefert entweder die Adresse des gefundenen Zeichens oder einen NULL-Zeiger, falls *zeich* nicht in *zeichk* gefunden werden kann.

Das folgende Programm 2.14 (strrchr.c), das den Dateinamen aus einem absoluten Pfadnamen ermittelt, ist ein Demonstrationsbeispiel zur Funktion **strrchr**. Der absolute Pfadname muß dabei auf der Kommandozeile angegeben werden.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRENnzeichen '/'

int
main(int argc, char *argv[])
{
    char *dateiname;

    if (argc != 2) {
        printf("Richtiger Aufruf: %s <absolut_pfadname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
}

```

```

    if ((dateiname=strrchr(argv[1], TRENnzeichen)) == NULL)
        dateiname = argv[0];
    else
        dateiname++; /* um voranstehenden / zu entfernen */

    printf("  ----- %s -----\\n", dateiname);

    exit(0);
}

```

Programm 2.14 (strrchr.c): Dateinamen zu einem absoluten Pfadnamen ermitteln

Nachdem man dieses Programm 2.14 (strrchr.c) kompiliert und gelinkt hat

```
cc -o strrchr strrchr.c
```

können sich z.B. die folgenden Abläufe ergeben:

```

$ strrchr /usr/include/ctype.h
  ----- ctype.h -----
$ strrchr /usr
  ----- usr -----
$ strrchr hans/meier
  ----- meier -----
$

```

```
size_t strspn(const char *kett1, const char *kett2);
```

berechnet die Länge der Teilzeichenkette in kett1 (von Anfang an), die nur aus Zeichen von kett2 besteht. Die Länge dieser Teilzeichenkette wird als Funktionswert zurückgegeben.

```
char *strstr(const char *kett1, const char *kett2);
```

sucht in kett1 das erste Vorkommen der Zeichenkette kett2 (ohne abschließendes \0). **strstr** liefert entweder einen Zeiger auf die gefundene Zeichenkette oder einen NULL-Zeiger, falls kett2 nicht eine Teilzeichenkette von kett1 ist. Wenn kett2 eine Zeichenkette der Länge 0 ist, so liefert diese Funktion kett1 zurück.

```
char *strtok(char *kett1, const char *kett2);
```

Eine Folge von Aufrufen der **strtok**-Funktion bricht die Zeichenkette kett1 in eine Folge von Teilzeichenketten²⁵, wobei die »Bruchstellen« durch kett2 festgelegt werden.

Der erste Aufruf von **strtok**, der kett1 als erstes Argument hat, bewirkt, daß in kett1 das erste Zeichen gesucht wird, das nicht als Trennzeichen in kett2 vorkommt. Falls kein solches Zeichen gefunden wird, dann gibt **strtok** einen NULL-Zeiger zurück. Wenn ein solches Nicht-Trennzeichen gefunden werden kann, dann ist dies der Anfang der ersten Teilzeichenkette.

25. ANSI C nennt diese Teilzeichenketten Token.

Von nun an sucht **strtok** nach einem Trennzeichen:

Falls keines gefunden werden kann, dann erstreckt sich die Teilzeichenkette bis zum Ende von `kett1` und nachfolgende Aufrufe von **strtok** werden fehlschlagen. Wenn ein solches Trennzeichen gefunden wird, dann wird es mit `\0` überschrieben und somit das Ende der Teilzeichenkette festgelegt. Die Funktion **strtok** merkt sich den Zeiger auf das nächste Zeichen, von wo aus bei einem Aufruf `strtok(NULL, ...)`; die nächste Suche nach einer Teilzeichenkette beginnt.

Diese Funktion gibt einen Zeiger auf das erste Vorkommen einer Teilzeichenkette zurück, oder einen `NULL`-Zeiger, falls keine gefunden werden kann.

Die Trennzeichen, die mit `kett2` angegeben werden, können bei jedem Aufruf verschieden sein. Das ANSI-C-Papier gibt hierzu folgendes Beispiel:

```
#include <string.h>
static char str[] = "?a???b,.,#c";
char *t;
t = strtok(str, "?");      /* t zeigt auf Teilzeichenkette "a" */
t = strtok(NULL, ",");     /* t zeigt auf Teilzeichenkette "??b" */
t = strtok(NULL, "#,");    /* t zeigt auf Teilzeichenkette "c" */
t = strtok(NULL, "?");     /* t ist ein NULL-Zeiger */
```

Das folgende Programm 2.15 (`strtok.c`) demonstriert die Anwendung der Funktion **strtok**.

```
#include <stdio.h>
#include <string.h>

char trennzeich[]=",:;";

int
main(void)
{
    char zeile[100], *einzel_name;
    int i=0;

    printf("Gib die Liste der Namen (mit , oder ; oder : getrennt ein\n");
    gets(zeile);
    einzel_name = strtok(zeile, trennzeich);
    while (einzel_name != NULL) {
        printf("Name %d : %s\n", ++i, einzel_name);
        einzel_name = strtok(NULL, trennzeich);
    }

    exit(0);
}
```

Programm 2.15 (strtok.c): Demonstrationsbeispiel zur Funktion strtok

Nachdem man dieses Programm 2.15 (`strtok.c`) kompiliert und gelinkt hat

```
cc -o strtok strtok.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ strtok
Gib die Liste der Namen (mit , oder ; oder : getrennt ein
Meier   Franz;;;;;Wasser-Fritz:Feuer Emil;Danne Doris-Annette::::
Name 1 : Meier   Franz
Name 2 : Wasser-Fritz
Name 3 : Feuer Emil
Name 4 : Danne Doris-Annette
$
```

size_t **strxfrm**(char *nach, const char *von, size_t max_groesse);

wandelt die lokalspezifische Zeichenkette von in eine »C-normale« Form (englisch-amerikanisch) um und speichert die umgewandelte Zeichenkette an der Adresse nach. Die Umwandlung garantiert, daß die Funktion **strcmp** auf zwei so umgewandelte Zeichenketten angewandt, das gleiche Ergebnis liefert, wie bei der Anwendung der Funktion **strcmp** auf die zwei Original-Zeichenketten.

Es werden niemals mehr als max_groesse Zeichen (\0 mitgerechnet) nach nach geschrieben. Wenn die beiden Zeichenketten sich überlappen, dann ist das Verhalten undefiniert. Falls für max_groesse der Wert 0 angegeben wird, so darf nach ein NULL-Zeiger sein.

Diese Funktion liefert als Funktionswert die Länge der umgewandelten Zeichenkette (ohne \0). Falls sie einen Wert >= max_groesse liefert, so ist der Speicherinhalt von nach unbestimmt.

Neben den hier vorgestellten Funktionen darf jede C-Realisierung noch eigene Funktionen in der Headerdatei <string.h> hinzufügen, wenn deren Namen mit

strk (*k* steht für Kleinbuchstabe) oder

memk (*k* steht für Kleinbuchstabe) oder

wcsk (*k* steht für Kleinbuchstabe)

beginnen.

2.5 Übung

2.5.1 Wertebereich der ganzzahligen Datentypen

Erstellen Sie ein Programm `wertber.c`, das unter Verwendung der Konstanten aus <limits.h> die Wertebereiche der einzelnen ganzzahligen Datentypen ausgibt, die Ihr C-Compiler für diese festlegt.

Nachdem man dieses Programm `wertber.c` kompiliert und gelinkt hat

```
cc -o wertber wertber.c
```

ergibt sich z.B. der folgende Ablauf:

\$ wertber

Hier verwendete Bitzahlen und daraus resultierende Wertebereiche

=====		
char	8	-128 .. 127
signed char	8	-128 .. 127
unsigned char	8	0 .. 255

short	16	-32768 .. 32767
unsigned short	16	0 .. 65535

int	32	-2147483648 .. 2147483647
unsigned int	32	0 .. 4294967295

long	32	-2147483648 .. 2147483647
unsigned long	32	0 .. 4294967295

\$

2.5.2 Duale Ausgabe von Gleitpunktzahlen

Jede Gleitpunktzahl kann in der Form $2.3756 \cdot 10^3$ angegeben werden. Bei dieser Darstellungsform setzt sich die Zahl aus zwei Bestandteilen zusammen:

- Mantisse (2.3756) und
- Exponent (3), welcher ganzzahlig ist.

Diese Form wird auch in C verwendet, außer daß der dort angegebene Exponent sich meist auf die in Computern übliche Basis 2 (nicht 10) bezieht. Die für die Darstellung einer Gleitpunktzahl verwendete Bytezahl legt fest, ob man mit

- einfacher Genauigkeit (Datentyp **float**) oder mit
- doppelter Genauigkeit (Datentyp **double**)

arbeitet. Die folgende Abbildung 2.1 zeigt das IEEE-Format für **float** und **double**, wobei 4 Bytes für **float** und 8 Bytes für **double** angenommen wird.

Das IEEE-Format geht von sogenannten normalisierten Gleitpunktzahlen aus. »Normalisierung« bedeutet, daß der Exponent so verändert wird, daß der gedachte Dezimalpunkt immer rechts von der ersten Nicht-Null-Ziffer (im Binärsystem ist dies eine 1) liegt.

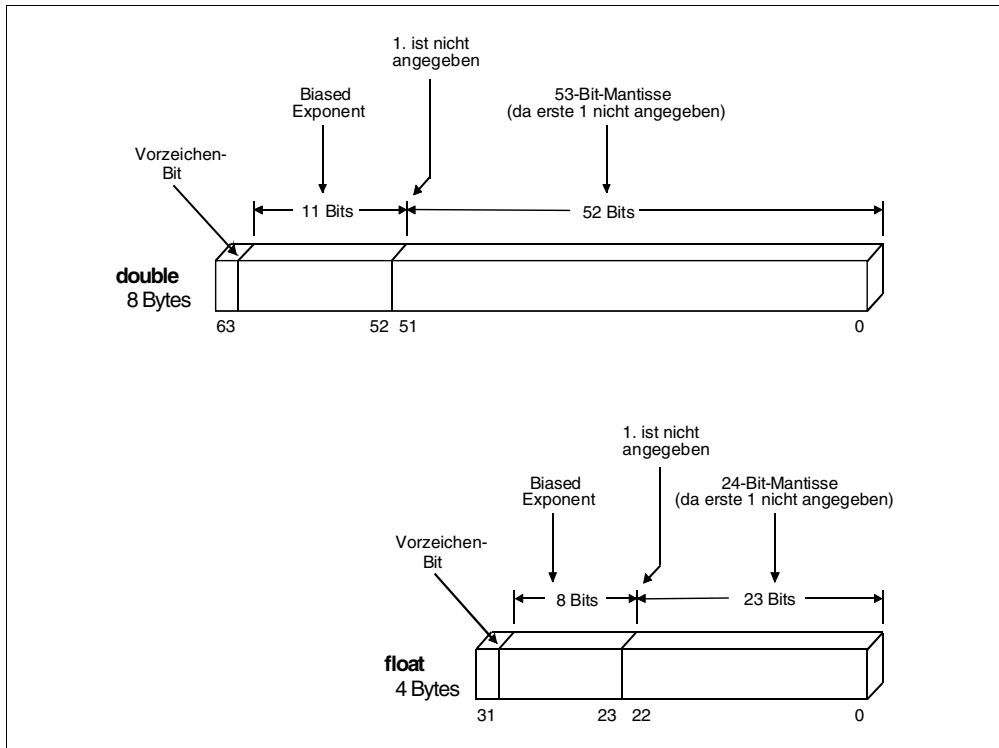


Abbildung 2.1: IEEE-Format von normalisierten Gleitpunktzahlen

Beispiel**Die Dezimalzahl**

$$17.625 = 1 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

entspricht der binären Zahl:

$$\begin{aligned} 16 + \frac{1}{2} + \frac{1}{8} &= \\ 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} &= \\ 10001.101 \cdot 2^0 \end{aligned}$$

Die entsprechende normalisierte Form erhält man, indem man den Dezimalpunkt hinter die erste signifikante Ziffer »schiebt« und den Exponenten entsprechend anpaßt:

$$1.0001101 \cdot 2^4$$

Gleitpunktzahlen sind immer in normalisierter Form dargestellt, und somit ist sichergestellt, daß das höchstwertige »Einser-Bit« immer links vom gedachten Dezimalpunkt²⁶ in

26. Außer für den Wert 0 natürlich.

Mantisse (52 Bits)

Wertebereich der Exponenten:

dual: $2^{-1021} \dots 2^{1024}$ dezimal: $10^{-307} \dots 10^{308}$

Wertebereich:

dezimal: $2.23\text{E-}308 \dots 1.80\text{E+}308$

Anzahl der signifikanten Dezimalstellen: 15

Epsilon: $2.22044604925031\text{e-}16$

\$

2.5.4 Ausgabe einer Cos-, Sin- und Tan-Tabelle

Erstellen Sie ein Programm `cosinta.c`, das eine Cosinus-, Sinus- und Tangentstabelle zu einem bestimmten Winkel-Bereich ausgibt.

Nachdem man dieses Programm `cosinta.c` kompiliert und gelinkt hat

```
cc -o cosinta cosinta.c -lm
```

können sich z.B. die folgenden Abläufe ergeben:

```
$ cosinta
```

```
Ausgabe einer Cos-, Sin- und Tan-Tabelle
```

```
=====
```

```
Startwert (in Grad): 0
```

```
Endwert (in Grad): 90
```

```
Schrittweite (in Grad): 10
```

Grad	Cosinus	Sinus	Tangens

0	1.00000	0.00000	0.00000
10	0.98481	0.17365	0.17633
20	0.93969	0.34202	0.36397
30	0.86603	0.50000	0.57735
40	0.76604	0.64279	0.83910
50	0.64279	0.76604	1.19175
60	0.50000	0.86603	1.73205
70	0.34202	0.93969	2.74748
80	0.17365	0.98481	5.67128
90	0.00000	1.00000	Unendlich

```
$ cosinta
```

```
Ausgabe einer Cos-, Sin- und Tan-Tabelle
```

```
=====
```

```
Startwert (in Grad): 30
```

```
Endwert (in Grad): 180
```

```
Schrittweite (in Grad): 25
```

Grad	Cosinus	Sinus	Tangens

30	0.86603	0.50000	0.57735

```

55 |          0.57358 |          0.81915 |          1.42815 |
80 |          0.17365 |          0.98481 |          5.67128 |
105 |         -0.25882 |          0.96593 |         -3.73205 |
130 |         -0.64279 |          0.76604 |         -1.19175 |
155 |         -0.90631 |          0.42262 |         -0.46631 |
180 |         -1.00000 |          0.00000 |         -0.00000 |
$

```

2.5.5 Runden auf eine beliebige Nachkommastellenzahl

Erstellen Sie ein C-Programm `runden.c`, das zunächst eine Gleitpunktzahl einliest, bevor es dann noch nach den Nachkommastellen fragt, auf die diese Zahl auf- bzw. abzurunden ist.

Das Programm soll nun die eingegebene Zahl auf die angegebenen Nachkommastellen auf- und abgerundet ausgeben. Zusätzlich soll dieses Programm die Zahl auf die angegebenen Nachkommastellen begrenzt ausgeben lassen, wobei es die Rundung den intern vorgegebenen Regeln überläßt. Am Ende soll dieses Programm für die eingegebene Zahl noch die deutsche Schreibweise (mit Komma) ausgeben.

Nachdem man dieses Programm `runden.c` kompiliert und gelinkt hat

```
cc -o runden runden.c -lm
```

können sich z.B. die folgenden Abläufe ergeben:

```

$ runden
Bitte Gleitpunktzahl eingeben: 12.345678
Auf wieviel Kommastellen runden: 4

Abgerundet: 12.3456
Aufgerundet: 12.3457
Nach Rundungsregeln: 12.3457
In deutscher Schreibweise: 12,3457
$ runden
Bitte Gleitpunktzahl eingeben: -347.56789
Auf wieviel Kommastellen runden: 1

Abgerundet: -347.6
Aufgerundet: -347.5
Nach Rundungsregeln: -347.6
In deutscher Schreibweise: -347,6
$

```

3

Standard-E/A-Funktionen

Haec alliis, ut, dum dicis, audias ipse.

Seneca

(Sage dies anderen, damit du, während du sprichst, es selber hörst.)

In diesem Kapitel werden E/A-Funktionen beschrieben, die sich in der Standard-E/A-Bibliothek befinden und in der Headerdatei `<stdio.h>` definiert sind. Da die meisten der hier vorgestellten E/A-Funktionen von ANSI C vorgeschrieben sind, sind sie auch auf anderen Betriebssystemen als Unix verfügbar.

Die Standard-E/A-Funktionen arbeiten im Gegensatz zu den im nächsten Kapitel behandelten elementaren E/A-Funktionen mit eigenen optimal eingestellten Puffern, so daß sich der Aufrufer darum nicht selbst kümmern muß. Auch bieten die Standard-E/A-Funktionen dem Benutzer mehr Komfort an, wie z.B. Formatierung der Ausgabe bei ***printf*** oder zeilenweises Einlesen bei ***fgets***.

3.1 Der Datentyp FILE

Wenn eine Datei geöffnet wird, gibt die Standard-E/A-Funktion ***fopen*** einen Zeiger vom Datentyp `FILE` zurück. `FILE` ist normalerweise eine Struktur, die alle Informationen enthält, die die Standard-E/A-Routinen für die Aktivitäten mit der geöffneten Datei benötigen, wie z.B.:

Anfangsadresse des Puffers

aktueller Pufferzeiger

Puffergröße

Filedeskriptor

Position des Schreib-/Lesezeigers in einer Datei

Fehler-Flag (zeigt an, ob ein Schreib-/Lesefehler auftrat)

EOF-Flag (zeigt an, ob beim Dateizugriff das Dateende erreicht wurde)

Im Normalfall sollte der Programmierer nichts mit den Interna der `FILE`-Struktur zu tun haben, sondern lediglich den von ***fopen*** gelieferten `FILE`-Zeiger als Argument bei den entsprechenden E/A-Funktionen angeben.

3.2 stdin, stdout und stderr

Für jeden Prozeß werden automatisch immer drei Filedescriptoren bereitgestellt:

STDIN_FILENO (standard input)
STDOUT_FILENO (standard output)
STDERR_FILENO (standard error)

Diesen drei Filedescriptoren entsprechen folgende FILE-Zeigerkonstanten, die in `<stdio.h>` definiert sind:

stdin (Standardeingabe)
stdout (Standardausgabe)
stderr (Standardfehlerausgabe)

3.3 Öffnen und Schließen von Dateien

Öffnet man eine Datei mit den Standard-E/A-Funktionen, so ordnet man dieser Datei einen sogenannten *Stream* zu, auf den man unter Verwendung des FILE-Zeigers schreiben oder aus dem man lesen kann.

3.3.1 fopen – Öffnen einer Datei

Um eine Datei zu öffnen, steht die ANSI-C-Funktion *fopen* zur Verfügung.

```
#include <stdio.h>

FILE *fopen(const char *pfadname, const char *modus);
```

gibt zurück: FILE-Zeiger (bei Erfolg); NULL bei Fehler

pfadname

Name der zu öffnenden Datei

modus

Mit dem Argument `modus` wird die Zugriffsart für die Datei `pfadname` festgelegt (siehe Tabelle 3.1).

modus-Arument	Bedeutung
»r« oder »rb«	(read) zum Lesen öffnen
»w« oder »wb«	(write) zum Schreiben öffnen (neu anlegen oder Inhalt einer existierenden Datei löschen)

Tabelle 3.1: Mögliche Angaben für *modus* bei *fopen* und *freopen*

<i>modus-Argument</i>	<i>Bedeutung</i>
»a« oder »ab«	(append) zum Schreiben am Dateiende öffnen; nicht existierende Datei wird angelegt
»r+«, »r+b« oder »rb+«	zum Lesen und Schreiben öffnen
»w+«, »w+b« oder »wb+«	zum Lesen und Schreiben öffnen; Inhalt einer existierenden Datei wird gelöscht
»a+«, »a+b« oder »ab+«	zum Lesen und Schreiben ab Dateiende öffnen

Tabelle 3.1: Mögliche Angaben für *modus* bei *fopen* und *freopen*

Der Buchstabe *b* bei der *modus*-Angabe wird benötigt, um zwischen Text- und Binärdateien zu unterscheiden. Da der Unixkern solche Dateiarnten nicht unterscheidet, hat dieses Zeichen *b* bei *modus* keinerlei Bedeutung in Unix. In anderen Betriebssystemen (wie z.B. MS-DOS) kann es jedoch wichtig sein, wenn z.B die systembedingte Interpretation von Neuezeilezeichen bei Binärdateien auszuschalten ist.

Die Tabelle 3.2 faßt zusammen, welche Einschränkungen bei den einzelnen Öffnungsmodi gelten.

Einschränkung bzw. Auswirkung	<i>r</i>	<i>w</i>	<i>a</i>	<i>r+</i>	<i>w+</i>	<i>a+</i>
Datei muß zuvor existieren	x			x		
alter Dateiinhalt geht verloren		x			x	
Aus Datei kann gelesen werden	x			x	x	x
In Datei kann geschrieben werden		x	x	x	x	x
Nur am Dateiende kann geschrieben werden			x			x

Tabelle 3.2: Einschränkungen und Auswirkungen bei den verschiedenen Öffnungsmodi

Fehler

Das Öffnen einer Datei im Lesemodus schlägt fehl, wenn die entsprechende Datei nicht existiert oder nicht gelesen werden kann.

Wenn eine Datei gleichzeitig zum Lesen und Schreiben geöffnet wird (+ in *modus*), dann ist folgendes zu beachten:

- Unmittelbares Lesen nach Schreibaktivitäten ist nicht möglich. Dazu muß zuerst ein Aufruf einer der Funktionen ***fflush***, ***fseek***, ***fsetpos*** oder ***rewind*** dazwischengeschaltet werden.
- Unmittelbares Schreiben nach Leseaktivitäten ist nicht ohne einen dazwischenliegenden Aufruf einer der Dateipositionierungsfunktionen ***fseek***, ***fsetpos*** oder ***rewind*** möglich, außer wenn zuvor das Dateiende gelesen wurde.

Hinweis

Die Fehler- und EOF-Flags werden beim Öffnen einer Datei zurückgesetzt.

Wenn eine Datei zum Schreiben am Dateiende (»a«, »a+«, ...) geöffnet wird, so findet jedes nachfolgende Schreiben am momentanen Ende der Datei statt. Falls mehrere Prozesse zur gleichen Zeit dieselbe Datei mit »append« öffnen, so werden die Daten jedes Prozesses korrekt in die Datei geschrieben.

Wenn eine neue Datei angelegt wird (Angabe von *w* oder *a* bei *modus*), können die Zugriffsrechte nicht wie bei den in Kapitel 4 vorgestellten Funktionen **open** und **creat** festgelegt werden. POSIX.1 legt fest, daß die Datei immer mit folgenden Rechten angelegt wird (siehe auch Kapitel 4.2):

S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH

was dem Unix-Zugriffsrechtemuster »rw-rw-rw-« entspricht.

Die Voreinstellung für jede geöffnete Datei (Stream) ist, daß diese voll gepuffert ist, außer für den Fall, daß es sich um ein Terminal handelt (zeilengepuffert). Soll nach dem Öffnen einer Datei die Pufferung geändert werden, so muß nach dem Öffnen, jedenfalls bevor erste Operationen stattfinden, mit den Funktionen **setbuf** oder **setvbuf** (siehe Kapitel 3.5) die gewünschte Pufferung eingestellt werden.

3.3.2 freopen – Öffnen einer Datei mit bereits existierendem Stream

Um eine Datei mit einem bereits existierenden FILE-Zeiger (Stream) zu verknüpfen, steht die ANSI-C-Funktion **freopen** zur Verfügung.

```
#include <stdio.h>
```

```
FILE *freopen(const char *pfadname, const char *modus, FILE *fz);
```

gibt zurück: FILE-Zeiger (bei Erfolg); NULL bei Fehler

freopen versucht zuerst, die entsprechende Datei, die mit *fz* verbunden ist, zu schließen. Mögliche Fehler beim Schließversuch werden ignoriert. Danach ordnet diese Funktion den FILE-Zeiger *fz* der Datei *pfadname* zu.

pfadname

Name der zu öffnenden Datei

modus

Mit dem Argument *modus* wird die Zugriffsart für die Datei *pfadname* festgelegt. Es entspricht dem *modus*-Argument von **fopen**. (siehe Tabelle 3.1).

Fehler

Für **freopen** gelten die gleichen Fehlerbedingungen wie für **fopen**; siehe vorherige Beschreibung von **fopen**.

Hinweis

Die hauptsächliche Anwendung von **freopen** ist, eine Datei mit den Standard-Dateizeigern `stdin`, `stdout` und `stderr` zu verbinden. Weitere Hinweise finden Sie bei der vorangegangenen Beschreibung von **fopen**, die auch für **freopen** zutreffen.

Beispiel

Standardausgabe zeitweise in eine Datei umlenken

Das nachfolgende C-Programm 3.1 (`catlog.c`) liest von der Standardeingabe Zeichen und gibt diese wieder auf das Terminal aus. Sobald es allerdings das Zeichen `>` liest, schreibt es die gelesenen Zeichen nicht mehr auf das Terminal, sondern in die Datei `prot.txt`. Erst wenn es das Zeichen `<` liest, gibt es die gelesenen Zeichen wieder auf das Terminal aus. Um `stdout` wieder zurück auf das Terminal zu lenken, muß der Dateiname `/dev/tty` verwendet werden.

```
#include "eighdr.h"

int
main(void)
{
    int    zeich, umgelenkt=0;

    while ( (zeich=getc(stdin)) != EOF) {

        if (zeich == '>') { /*----- stdout in Datei prot.txt umlenken ---*/
            if (freopen("prot.txt", "a", stdout) != stdout)
                fehler_meld(FATAL_SYS, "Fehler bei freopen mit stdout");
            umgelenkt = 1;

        } else if (umgelenkt && zeich == '<') { /*- stdout zurueck auf Terminal*/
            if (freopen("/dev/tty", "w", stdout) != stdout)
                fehler_meld(FATAL_SYS, "Fehler bei freopen mit stdout");
            umgelenkt = 0;

        } else if (putc(zeich, stdout) == EOF)
            fehler_meld(FATAL_SYS, "Fehler bei putc");
    }

    if (ferror(stdin))
        fehler_meld(FATAL_SYS, "Fehler bei getc");

    exit(0);
}
```

Programm 3.1 (catlog.c): Standardausgabe zeitweise in eine Datei umlenken

Nachdem man Programm 3.1 (catlog.c) kompiliert und gelinkt hat

```
cc -o catlog catlog.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ catlog
Ich gebe Geheimwort ein: >hansimglueck<
Ich gebe Geheimwort ein:      [>hansimglueck< wird nicht angezeigt]
Und noch ein Test>           [von > bis zum nächsten < wird nicht angezeigt]
Und noch ein Test-----<
Ende
Ende
Ctrl-D
$ cat prot.txt
hansimglueck
-----
$
```

3.3.3 fclose – Schließen einer Datei

Um eine geöffnete Datei wieder zu schließen, steht die ANSI-C-Funktion **fclose** zur Verfügung.

```
#include <stdio.h>
```

```
int fclose(FILE *fz);
```

gibt zurück: 0 (bei Erfolg); EOF bei Fehler

Bevor **fclose** die Verbindung zwischen einer Datei und dem FILE-Zeiger *fz* auflöst, überträgt diese Funktion alle Inhalte von noch nicht geleerten Ausgabepuffern in die entsprechende Datei (siehe auch Funktion **fflush** in Kapitel 3.5). Inhalte von Eingabepuffern gehen verloren.

Hinweis

Wenn ein Prozeß normal endet (entweder mit **exit** oder **return** in der *main*-Funktion), werden die Inhalte aller Standard-E/A-Puffer automatisch in die entsprechenden Dateien übertragen, bevor alle offenen Dateien (*Streams*) geschlossen werden.

3.4 Lesen und Schreiben in Dateien

Nachdem eine Datei zum Lesen und/oder Schreiben geöffnet wurde, kann man in ihr lesen und/oder schreiben. Es gibt dabei verschiedene Arten, in einer Datei zu lesen bzw. zu schreiben, wie z.B. zeichenweise, zeilenweise, formatiert oder blockweise.

3.4.1 feof und ferror – Prüfen des EOF- und Fehler-Flags

Die meisten der hier beschriebenen Eingabefunktionen liefern sowohl beim Erreichen des Dateiendes als auch bei Auftreten eines Lesefehlers EOF zurück. Um nun nachträglich feststellen zu können, welcher der beiden Fälle vorlag, stehen die beiden Funktionen **feof** und **ferror** zur Verfügung

```
#include <stdio.h>

int feof(FILE *fz);
    gibt zurück: Wert verschieden von 0, wenn EOF-Flag für Datei fz gesetzt ist; 0 sonst

int ferror(FILE *fz);
    gibt zurück: Wert verschieden von 0, wenn Fehler-Flag für Datei fz gesetzt ist; 0 sonst
```

In der FILE-Struktur befinden sich meist zwei Flags:

- ein Fehler-Flag und
- ein EOF-Flag

Tritt beim Lesen aus oder Schreiben in eine Datei (Stream) ein Fehler auf, so wird das Fehler-Flag gesetzt. Wird beim Lesen aus einer Datei (Stream) das Dateiende erreicht, so wird das EOF-Flag gesetzt. Um zu überprüfen, ob diese Flags gesetzt sind, stehen diese beiden Funktionen **feof** und **ferror** zur Verfügung.

3.4.2 clearerr – Löschen des Fehler- und EOF-Flags

Um das Fehler- und EOF-Flag zu löschen, steht die Funktion **clearerr** zur Verfügung.

```
#include <stdio.h>

void clearerr(FILE *fz);
```

3.4.3 getchar – Lesen eines Zeichen von stdin putchar – Schreiben eines Zeichen auf stdout

Um ein Zeichen von der Standardeingabe (stdin) zu lesen, steht die Funktion **getchar** und zum Schreiben eines Zeichens auf die Standardausgabe (stdout) steht die Funktion **putchar** zur Verfügung.

```
#include <stdio.h>

int getchar(void);
                                gibt zurück: nächstes Zeichen aus stdin (bei Erfolg); EOF bei Dateiende oder Fehler

int putchar(int zeich);
                                gibt zurück: zeich (bei Erfolg); EOF bei Fehler
```

Nach ANSI C ist der Aufruf

`getchar()` äquivalent mit dem Aufruf `getc(stdin)` und der Aufruf `putchar(zeich)` ist äquivalent mit dem Aufruf `putc(zeich, stdout)`.

Hinweis

getchar liefert das nächste Zeichen aus der Standardeingabe als **unsigned char**, das im Datentyp **int** abgelegt ist. Es wird **int** als Rückgabebetyp gewählt, um auch negative Rückgabewerte zu ermöglichen, wie z.B. die Konstante `EOF` (in `<stdio.h>` definiert), die immer eine negative Zahl sein muß (meist -1). Es ist deshalb zu beachten, daß die Variablen, in welche die mit **getchar** gelesenen Zeichen unterzubringen sind, mit **int** und nicht mit **unsigned char** deklariert werden. So führt z.B. das folgende Programm 3.2 (`endlos1.c`) zu einer Endlosschleife:

```
#include "eighdr.h"

int
main(void)
{
    unsigned char  zeich; /*--- Hier liegt Fehler; richtig waere: int zeich; ---*/

    while ( (zeich=getchar()) != EOF)
        putchar(zeich);

    exit(0);
}
```

Programm 3.2 (**endlos1.c**): Endlosschleife wegen falscher Deklaration bei `getchar`

getchar und **putchar** müssen laut ANSI C nicht als Funktionen, sondern können auch als Makros implementiert sein.

Rückgabewert `EOF` bei `getchar` (Lesefehler oder Dateiende erreicht?)

getchar gibt sowohl beim Erreichen des Dateiendes als auch bei Auftreten eines Lesefehlers `EOF` zurück. Um nun nachträglich feststellen zu können, welcher der beiden Fälle eingetreten ist, müssen die zuvor beschriebenen Funktionen **ferror** und **feof** verwendet werden.

3.4.4 **getc und fgetc – Lesen eines Zeichens aus einer Datei** **putc und fputc – Schreiben eines Zeichens in eine Datei**

Um ein Zeichen aus einer Datei zu lesen, stehen die beiden Funktionen **getc** und **fgetc**, zum Schreiben eines Zeichens in eine Datei stehen die Funktionen **putc** und **fputc** zur Verfügung.

```
#include <stdio.h>

int getc(FILE *fz);
int fgetc(FILE *fz);
    beide geben zurück: nächstes Zeichen aus Datei fz (bei Erfolg); EOF bei Dateiende oder Fehler

int putc(int zeich, FILE *fz);
int fputc(int zeich, FILE *fz);
    beide geben zurück: zeich (bei Erfolg); EOF bei Fehler
```

Die beiden Funktionen **getc** und **fgetc** lesen aus der Datei (Stream), der der FILE-Zeiger *fz* zugeteilt ist, das nächste Zeichen und liefern dieses Zeichen als Rückgabewert.

Die beiden Funktionen **putc** und **fputc** schreiben das Zeichen *zeich* (das zuvor nach **unsigned char** umgewandelt wird) in die Datei, der der FILE-Zeiger *fz* zugeteilt ist.

Unterschied zwischen (fgetc, fputc) und (getc, putc)

Der einzige Unterschied zwischen **fgetc** und **getc** bzw. zwischen **fputc** und **putc** ist, daß nach ANSI C **fgetc** und **fputc** in jedem Fall als Funktionen realisiert sein müssen, während **getc** und **putc** auch als Makros implementiert sein dürfen.

Hinweis

Nach ANSI C ist der Aufruf

`getchar()` äquivalent mit dem Aufruf `getc(stdin)` und der Aufruf
`putchar(zeich)` ist äquivalent mit dem Aufruf `putc(zeich, stdout)`.

getc und **fgetc** liefern das nächste Zeichen aus dem Stream *fz* als **unsigned char**, das jedoch im Datentyp **int** abgelegt ist. Es wird **int** als Rückgabetyt gewählt, um auch negative Rückgabewerte zu ermöglichen, wie z.B. die Konstante `EOF` (in `<stdio.h>` definiert), die immer eine negative Zahl sein muß (meist -1). Es ist deshalb zu beachten, daß die Variablen, in welche die mit **getc** oder **fgetc** gelesenen Zeichen unterzubringen sind, mit **int** und nicht mit **unsigned char** deklariert werden, sonst kann dies zu einer Endlosschleife führen; siehe auch Programm 3.2 (`endlos1.c`).

Da **getc** und **putc** nicht als Funktionen, sondern auch als Makros implementiert sein dürfen, sollte der Programmierer hier kein Argument mit Nebeneffekten angeben, da dieses Argument eventuell mehrmals ausgewertet wird. Es sollten deshalb Ausdrücke wie der folgende vermieden werden:

```
putc(zeich, f=fopen("dateiname"));
```

Rückgabewert EOF bei getc bzw. fgetc (Lesefehler oder Dateiende erreicht?)

getc und **fgetc** geben sowohl beim Erreichen des Dateiendes als auch bei Auftreten eines Lesefehlers EOF zurück. Um nun nachträglich feststellen zu können, welcher der beiden Fälle vorliegt, müssen die zuvor beschriebenen Funktionen **feof** und **ferror** aufgerufen werden.

Beispiel

Größe von Dateien ermitteln und ausgeben

Das folgende Programm 3.3 (bytzahl1.c) zählt alle Zeichen der auf der Kommandozeile angegebenen Dateien. Es gibt dabei zu jeder einzelnen Datei deren Bytezahl sowie am Ende auch die gesamte Bytezahl aller Dateien aus.

```
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    FILE          *fz;
    int           i;
    unsigned long int  b, total=0;

    if (argc < 2)
        fehler_meld(FATAL, "Es muss mind. ein Dateiname angegeben sein");

    for (i=1 ; i<argc ; i++) {
        if ( (fz=fopen(argv[i], "rb")) == NULL) /*-- Oeffnen der i.ten Datei --*/
            fehler_meld(FATAL_SYS, "Kann %s nicht eroeffnen", argv[i]);

        b=0; /*--- Lesen und Zaehlen aller Bytes der i.ten Datei -----*/
        while (fgetc(fz) != EOF)
            b++;
        total += b;
        if (ferror(fz))
            fehler_meld(FATAL_SYS, "Fehler beim Lesen aus %s", argv[i]);

        fclose(fz); /*--- Schliessen der i.ten Datei -----*/

        printf("%30s : %lu\n", argv[i], b);
    }
}
```

```
printf("-----\n");
printf("%30s : %lu\n", "Gesamt", total); /*-- Ausgabe gesamter Bytezahl --*/

exit(0);
}
```

Programm 3.3 (**bytzahl1.c**): Größe von Dateien ermitteln und ausgeben

3.4.5 ungetc – Zurückschieben eines gelesenen Zeichens in Eingabepuffer

Um ein aus einer Datei gelesenes Zeichen wieder ungelesen zu machen, d.h. wieder in den Eingabepuffer zurückzuschieben, steht die Funktion **ungetc** zur Verfügung.

```
#include <stdio.h>
```

```
int ungetc(int zeich, FILE *fz);
```

gibt zurück: *zeich* (bei Erfolg); EOF bei Fehler

ungetc »schiebt« das Zeichen *zeich* (nachdem es zuvor nach **unsigned char** umgewandelt wurde) zurück in die Datei, die mit *fz* verbunden ist. Somit ist *zeich* das erste Zeichen, das beim nächsten Lesen aus der Datei (Stream) *fz* gelesen wird.

Hinweis

Das Zeichen, das man mit **ungetc** in den Eingabepuffer zurückschreibt, muß nicht unbedingt das zuletzt gelesene Zeichen sein.

Ein erfolgreicher Aufruf von **ungetc** löscht das EOF-Flag. Deswegen ist es auch nach dem Erreichen des Dateiendes möglich, ein Zeichen mit **ungetc** zurückzuschreiben. Es ist jedoch nicht möglich, die Konstante EOF zurückzuschreiben.

Wenn auch viele Implementierungen es zulassen, daß nacheinander mehr als ein Zeichen in den Eingabepuffer zurückgeschoben wird, so garantiert ANSI C nur das Zurückschreiben eines einzigen Zeichens.

Wird vor dem nächsten »Lesevorgang« eine der Funktionen **fseek**, **fsetpos** oder **rewind** erfolgreich aufgerufen, dann ist das mit **ungetc** zurückgeschriebene Zeichen nicht mehr im Eingabepuffer verfügbar.

Beispiel

Herausfiltern von hexadezimalen Zahlen aus einem Text

Programm 3.4 (**hexextra.c**) filtert aus einem Text alle hexadezimalen Zahlen heraus:

```
#include <ctype.h>
#include "eighdr.h"
```

```
int
```

```

main(int argc, char *argv[])
{
    FILE          *fz;
    unsigned long int  hexzahl;
    int            zeich;

    if (argc != 2)
        fehler_meld(FATAL, "Es muss ein Dateiname angegeben sein");

    if ( (fz=fopen(argv[1], "r")) == NULL)
        fehler_meld(FATAL_SYS, "Kann %s nicht eroeffnen", argv[1]);

    while ( (zeich=fgetc(fz)) != EOF) {
        if (isxdigit(zeich)) {
            ungetc(zeich, fz);
            fscanf(fz, "%lx", &hexzahl);
            printf("%lx=%lu\n", hexzahl, hexzahl);
        }
    }

    if (ferror(fz))
        fehler_meld(FATAL_SYS, "Fehler beim Lesen aus %s", argv[1]);

    fclose(fz);

    exit(0);
}

```

Programm 3.4 (hexextra.c): Hexa-Zahlen aus einem Text herausfiltern

Immer wenn dieses Programm eine hexadezimale Ziffer (Makro **isxdigit** liefert Wert verschieden von 0) liest, schiebt es diese mit **ungetc** zurück in den Eingabepuffer und läßt dann die ganze Hexa-Zahl mit **fscanf** lesen, was wesentlich einfacher ist, als wenn es diese Zahl selbst zeichenweise einlesen und dann »zusammenbauen« würde. Ein solcher *Lookahead* ist eine typische Anwendung für **ungetc**. Nachdem man dieses Programm 3.4 (hexextra.c) kompiliert und gelinkt hat

```
cc -o hexextra hexextra.c fehler.c
```

könnte sich z.B. folgender Ablauf ergeben

```
$ cat xx.txt
```

Hier sind Hexzahlen versteckt

```
2Affen, 3babef, caba
```

```
$ hexextra xx.txt
```

```
e=14
```

```
d=13
```

```
e=14
```

```
a=10
```

```
e=14
```

```
e=14
```

```
ec=236
2affe=176126
3babef=3910639
caba=51898
$
```

3.4.6 **gets und fgets – Lesen einer ganzen Zeile von stdin oder aus Datei** **puts und fputs – Schreiben einer ganzen Zeile auf stdin oder in Datei**

Zum Lesen einer ganzen Zeile von der Standardeingabe (`stdin`) steht die ANSI-C-Funktion **gets** und zum Lesen einer ganzen Zeile aus einer Datei (Stream) steht die Funktion **fgets** zur Verfügung. Mit Funktion **puts** kann eine ganze Zeile auf die Standardausgabe (`stdout`) und mit der Funktion **fputs** in eine Datei geschrieben werden.

```
#include <stdio.h>

char *gets(char *puffer);
char *fgets(char *puffer, int n, FILE *fz);
           beide geben zurück: Adresse puffer (bei Erfolg); NULL bei Dateiende oder Fehler

int puts(const char *puffer);
int fputs(const char *puffer, FILE *fz);
           beide geben zurück: nichtnegativen Wert (bei Erfolg); EOF bei Fehler
```

gets und fgets

Beiden Funktionen **gets** und **fgets** wird mittels `puffer` die Speicheradresse mitgeteilt, an der die gelesene Zeile im Hauptspeicher (mit abschließenden `\0`) abzulegen ist.

Bei **fgets** muß zusätzlich noch die Größe des bereitgestellten `puffer` und der `FILE`-Zeiger `fz` der Datei angegeben werden, aus der zu lesen ist. **fgets** liest dann aus dem Stream `fz` entweder `n-1` Zeichen oder bis zum nächsten Neue-Zeile-Zeichen (`\n`) – je nachdem, was zuerst eintritt – und speichert die gelesenen Zeichen an der Adresse `puffer` ab, wobei hinter dem letzten Zeichen immer das String-Ende-Zeichen `\0` abgelegt wird.

puts und fputs

Beiden Funktionen **puts** und **fputs** wird mittels `puffer` die Speicheradresse mitgeteilt, an der sich die zu schreibende Zeile im Hauptspeicher befindet. Das abschließende `\0` der Zeichenkette `puffer` wird nicht geschrieben.

Bei **fputs** muß zusätzlich noch der `FILE`-Zeiger `fz` der Datei angegeben werden, in die zu schreiben ist. Es ist zu beachten, daß **puts** immer automatisch am Ende der ausgegebenen Zeichenkette noch ein `\n` ausgibt, was **fputs** nicht tut.

Unterschiede zwischen *gets* und *fgets*

fgets unterscheidet sich von der Funktion *gets* darin, daß es nicht nur von der Standardeingabe lesen kann und auch automatisch das `\n`-Zeichen am Ende der gelesenen Zeichenkette anhängt, wenn die Länge der gelesenen Zeichenkette kleiner gleich *n* ist.

Da bei *gets* der Aufrufer anders als bei *fgets* keine Möglichkeit hat, die Größe des Puffers zu wählen, kann es zum Überlaufen des von *gets* gewählten Puffers kommen, wenn eine gelesene Zeile mehr Zeichen als die intern gewählte Pufferlänge hat. Wenn möglich, sollte also immer *fgets* anstelle von *gets* benutzt werden.

Hinweis

fgets liefert den Zeiger `puffer` oder `NULL`, wenn das Dateiende erreicht wurde (Inhalt von `puffer` bleibt unverändert) oder beim Lesevorgang ein Fehler auftrat (Inhalt von `puffer` ist unbestimmt).

3.4.7 *scanf* und *fscanf* – Formatiertes Lesen von `stdin` oder aus Datei

Um formatiert von der Standardeingabe oder aus einer Datei zu lesen, stehen die beiden Funktionen *scanf* und *fscanf* zur Verfügung

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *fz, const char *format, ...);

beide geben zurück: Anzahl der gelesenen Eingabeeinheiten (bei Erfolg);
EOF bei Dateiende oder Fehler vor einer Umwandlung
```

Die Funktion *scanf* ist äquivalent mit

```
fscanf(stdin, format, ...);
```

Nachfolgend wird ein kurzer Überblick über die möglichen `format`-Angaben gegeben.

format

`format` gibt an, wie die einzelnen Argumente einzulesen sind und legt somit das Eingabeformat fest. In der `format`-Zeichenkette können angegeben sein:

- ein oder mehrere Zwischenraumzeichen (*Leerzeichen*, `\f`, `\n`, `\r`, `\t` oder `\v`); ein Zwischenraumzeichen in der `format`-Angabe bedeutet, daß alle in der Eingabezeile folgenden Leerzeichen, Tabulatoren, Seiten- und Zeilenvorschübe bis zum ersten Nicht-Zwischenraumzeichen zu überlesen sind.

- einfache Zeichen (weder % noch Zwischenraumzeichen)
Ein einfaches Zeichen in der `format`-Angabe bewirkt, daß die nächsten Zeichen in der Eingabezeile gelesen werden. Wenn jedoch ein Zeichen aus der Eingabe nicht dem angegebenen Zeichen entspricht, dann schlägt dieser Leseversuch fehl, und sowohl dieses wie auch nachfolgende Zeichen bleiben ungelesen.
- Umwandlungsvorgaben (beginnen immer mit %)

Umwandlungsvorgaben

Umwandlungsvorgaben beginnen immer mit % und beziehen sich auf die folgenden Argumente: 1. Umwandlungsvorgabe auf das 1. Argument, 2. Umwandlungsvorgabe auf das 2. Argument usw. Umwandlungsvorgaben legen immer fest, wie entsprechendes Argument einzulesen ist.

Eine Umwandlungsvorgabe setzt sich wie folgt zusammen: % S W L U

S = [*]	Argumenten wird kein Wert zugewiesen; es wird "übersprungen"
W = [Weite]	max. Anzahl der zu lesenden Zeichen
L = [Längenangabe]	legt Größe des entsprechenden Eingabelements fest (h für short ; l oder L für long)
U = Umwandlungszeichen	

Hier ist zu erkennen, daß allein das Umwandlungszeichen immer angegeben sein muß. Die anderen Angaben (*, *Weite* und *Längenangabe*) sind optional. Die Tabelle 3.3 zeigt alle bei **scanf** und **fscanf** möglichen Umwandlungszeichen.

Umwandlungszeichen	Eingabedaten	Argumenttyp (Adresse von ...)
d	ganze Zahl (Suffix u,U,I,L nicht erlaubt)	Ganzzahlvariable
i	ganze Zahl (Suffix u,U,I,L nicht erlaubt)	Ganzzahlvariable
o	ganze Oktalzahl	unsigned -Ganzzahlvariable
u	ganze Zahl	unsigned -Ganzzahlvariable
x, X	ganze Hexadezimalzahl	unsigned -Ganzzahlvariable
e,f,g,E,G	Gleitpunktzahl	Gleitpunktvariable
s	Zeichenkette (ohne Zwischenraumzeichen)	char -Variable
c	Zeichenkette (anders als bei %s werden hier Zwischenraumzeichen gelesen)	char -Variable
p	Zeigerwert	Zeigervariable
n	kein Lesevorgang (Anzahl der bisher gelesenen Zeichen wird in zugehörige Argument geschrieben)	Ganzzahlvariable

Tabelle 3.3: Die bei scanf und fscanf möglichen Umwandlungszeichen

Umwandlungszeichen	Eingabedaten	Argumenttyp (Adresse von ...)
[liste]	Zeichenkette (Einlesen bis Zeichen, das nicht in <i>liste</i> vorkommt) ¹	char -Variable
[^liste]	Zeichenkette (Einlesen bis Zeichen, das in <i>liste</i> vorkommt) ²	char -Variable
%	(das Zeichen) % (liest Zeichen % aus der Eingabe)	kein Argument

Tabelle 3.3: Die bei *scanf* und *fscanf* möglichen Umwandlungszeichen

Reihenfolge der Abarbeitung von Eingaben durch *scanf* oder *fscanf*

Für jede Umwandlungsvorgabe werden folgende Aktivitäten (in angegebener Reihenfolge) auf der Eingabezeile durchgeführt:

1. Zwischenraumzeichen in der Eingabezeile werden einfach übersprungen, außer die *format*-Angabe verwendet an dieser Stelle eines der Umwandlungszeichen *]*, *c* oder *n*.
2. Es wird eine Eingabeeinheit von der Eingabe gelesen³. Eine Eingabeeinheit ist die längste passende Folge von Eingabezeichen (bis zu einer eventuellen *weite*). Das erste Zeichen nach dieser Eingabeeinheit bleibt ungelesen.
3. Die Eingabeeinheit wird entsprechend den vorgegebenen Umwandlungszeichen in einen geeigneten Typ konvertiert. Wenn sich die Eingabeeinheit als nicht passend für dieses Umwandlungszeichen erweist, so liegt eine »falsche Eingabe« vor und ***scanf*** bzw. ***fscanf*** wird verlassen. Nachfolgende Zwischenraumzeichen bleiben ungelesen, außer sie werden durch eine Umwandlungsvorgabe angefordert.

Beispiel

Demonstrationsprogramme zu *fscanf*

Das folgende Programm 3.5 (*fscanf1.c*) demonstriert das Einlesen von Zeichenketten, die in Apostrophen oder Anführungszeichen angegeben sind. Um die Sonderbedeutung eines Anführungszeichens als String-Begrenzer im *format*-String auszuschalten, muß dem entsprechenden Anführungszeichen ein Backslash (**) vorangestellt werden.

```
#include "eighdr.h"

/* Lesen einer Zeichenkette, welche durch Apostroph oder
 * Anführungszeichen begrenzt ist
 */
```

-
1. Wenn *]* in *liste* angegeben werden soll, so ist es dort als 1. Zeichen anzugeben: *[]...*
 2. Wenn *[^* in *liste* angegeben werden soll, so ist es dort als 2. Zeichen anzugeben: *[^]...*
 3. Außer für das Umwandlungszeichen *n*.

```

int
main(void)
{
    char  zeichkettel[100], zeichkette2[100], begrenz;

    fscanf(stdin, "%[^'\"]%c %s", zeichkettel, &begrenz, zeichkette2);
    printf("%s (1. eingegeb. Zeichkette)\n", zeichkettel);
    printf("%s (2. eingegeb. Zeichkette)\n", zeichkette2);
}

```

Programm 3.5 (fscanf1.c): Einlesen von Zeichenketten in Apostrophe oder Anführungszeichen

Nachdem man dieses Programm 3.5 (fscanf1.c) kompiliert und gelinkt hat

```
cc -o fscanf1 fscanf1.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ fscanf1
"Mit Gaensefuesschen" Ohne Gaensefuesschen
Mit Gaensefuesschen (1. eingegeb. Zeichkette)
Ohne (2. eingegeb. Zeichkette)
$ fscanf1
"Zeichenkettel" "Zeichenkette2"
Zeichenkettel (1. eingegeb. Zeichkette)
"Zeichenkette2" (2. eingegeb. Zeichkette)
$

```

Das folgende Programm 3.6 (fscanf2.c) demonstriert die Wirkungsweise einiger format-Angaben.

```

#include "eighdr.h"

int
main(void)
{
    int    gelesen, i;
    float  gleit;
    char   zeichkette[100];

    gelesen = fscanf(stdin, "%d%f%s", &i, &gleit, zeichkette);
    printf("%d (gelesen) -- %d (i) -- %f (gleit) -- %s (zeichkette)\n",
           gelesen, i, gleit, zeichkette);

    gelesen = fscanf(stdin, "%2d%f%d %[0123456789]",
                     &i, &gleit, zeichkette);
    printf("%d (gelesen) -- %d (i) -- %f (gleit) -- %s (zeichkette)\n",
           gelesen, i, gleit, zeichkette);
}

```

Programm 3.6 (fscanf2.c): Wirkungsweise einzelner Formatangaben

Nachdem man dieses Programm 3.6 (fscanf2.c) kompiliert und gelinkt hat

```
cc -o fscanf2 fscanf2.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ fscanf2
1254 1652.2e-5 zeichen
3 (gelesen) -- 1254 (i) -- 0.016522 (gleit) -- zeichen (zeichkette)
264523 8865 623z8983
2 (gelesen) -- 26 (i) -- 4523.000000 (gleit) -- 623 (zeichkette)
$
```

Das folgende Programm 3.7 (fscanf3.c) demonstriert die Wirkungsweise weiterer format-Angaben.

```
#include <stdlib.h>
#include "eighdr.h"

int
main(void)
{
    int    gelesen;
    float  menge;
    char   einheit[21], artikel[21];
    FILE   *dz = fopen("fscanf.txt", "r");

    if (dz==NULL)
        fehler_meld(FATAL_SYS, "%s kann nicht eroeffnet werden", "fscanf.txt");

    while (!feof(dz) && !ferror(dz)) {
        gelesen = fscanf(dz, "%f%20s voller %20s", &menge, einheit, artikel);
        fscanf(dz, "%*[\n]");
        printf("%d (gelesen) -- %f (menge) -- %s (einheit) -- %s (artikel)\n",
               gelesen, menge, einheit, artikel);
    }
}
```

Programm 3.7 (fscanf3.c): Wirkungsweise einzelner Formatangaben

Nachdem man dieses Programm 3.7 (fscanf3.c) kompiliert und gelinkt hat

```
cc -o fscanf3 fscanf3.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ cat fscanf.txt
2 Faesser voller Oel
25.5Grad Celsius
Haus voller Maeuse
11.0Sack    voller
Kartoffel
100elefanten voller Gold
$ fscanf3
```

```
3 (gelesen) -- 2.000000 (menge) -- Faesser (einheit) -- Oel (artikel)
2 (gelesen) -- 25.500000 (menge) -- Grad (einheit) -- Oel (artikel)
0 (gelesen) -- 25.500000 (menge) -- Grad (einheit) -- Oel (artikel)
3 (gelesen) -- 11.000000 (menge) -- Sack (einheit) -- Kartoffel (artikel)
3 (gelesen) -- 100.000000 (menge) -- elefanten (einheit) -- Gold (artikel)
-1 (gelesen) -- 100.000000 (menge) -- elefanten (einheit) -- Gold (artikel)
$
```

3.4.8 printf und fprintf – Formatiertes Schreiben auf stdout oder in eine Datei

Um formatiert auf die Standardausgabe oder in eine Datei zu schreiben, stehen die beiden Funktionen **printf** und **fprintf** zur Verfügung.

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *fz, const char *format, ...);

    beide geben zurück: Anzahl der geschriebenen Zeichen (bei Erfolg); negativer Wert bei Ausgabefehler
```

Die Funktion **printf** ist äquivalent mit

```
fprintf(stdout, format, ...);
```

Nachfolgend wird ein kurzer Überblick über die möglichen `format`-Angaben gegeben.

format

`format` gibt an, wie die einzelnen Argumente auszugeben sind und legt somit das Ausgabeformat fest. In der `format`-Zeichenkette können sowohl normale ASCII-Zeichen, die unverändert ausgegeben werden, als auch die in Tabelle 3.4 aufgeführten Steuerzeichen enthalten sein.

Steuerzeichen	Bedeutung
<code>\a</code>	Klingelton (auch mit <code>\007</code> zu verwirklichen)
<code>\b</code>	Backspace (ein Zeichen zurück positionieren)
<code>\f</code>	Seitenvorschub
<code>\n</code>	Neue Zeile
<code>\r</code>	Wagenrücklauf (an Anfang der momentanen Zeile positionieren)
<code>\t</code>	Tabulator
<code>\v</code>	Vertikales Tabulatorzeichen
<code>\ooo</code>	Zeichen, das der Oktalzahl <code>ooo</code> entspricht

Tabelle 3.4: Sonderzeichen in der `format`-Angabe

Steuerzeichen	Bedeutung
<code>\xhh</code>	Zeichen, das der Hexadezimalzahl <i>hh</i> entspricht
<code>\'</code>	Hochkomma
<code>\"</code>	Anführungszeichen
<code>\\</code>	Backslash

Tabelle 3.4: Sonderzeichen in der *format*-Angabe

Neben den normalen ASCII-Zeichen und den obigen Steuerzeichen können in `format` noch *Umwandlungsvorgaben* angegeben sein.

Umwandlungsvorgaben

Umwandlungsvorgaben beginnen immer mit `%` und beziehen sich auf die nachfolgenden Argumente: 1. Umwandlungsvorgabe auf das 1. Argument, 2. Umwandlungsvorgabe auf das 2. Argument usw. Umwandlungsvorgaben legen immer fest, wie das entsprechende Argument auszugeben ist. Eine Umwandlungsvorgabe setzt sich wie folgt zusammen:
%FWGLU

F = [Formatierungszeichen]
W = [Weite] Mindestzahl der auszugebenden Zeichen
G = [Genauigkeit] . oder .^{*} oder .*ganzzahl*
L = [Längenangabe] h (short), l oder L (long)
U = Umwandlungszeichen

Hieran ist zu erkennen, daß nur das Umwandlungszeichen immer angegeben sein muß. Die anderen Angaben (*Formatierungszeichen*, *Weite*, *Genauigkeit* und *Längenangabe*) sind optional.

Umwandlungszeichen

Die Tabelle 3.5 zeigt alle bei ***printf*** und ***fprintf*** möglichen Umwandlungszeichen.

Zeichen	Wert des Arguments wird ausgegeben....
d, i	als eine vorzeichenbehaftete ganze Dezimalzahl (i ist neu in ANSI C)
o	als eine vorzeichenlose ganze Oktalzahl
u	als eine vorzeichenlose ganze Dezimalzahl
x, X	als eine vorzeichenlose ganze Hexazahl (a,b,c,d,e,f) bei x , und (A,B,C,D,E,F) bei X
f	in der Form [-]ddd.dddddd
e,E	in der Form [-]d.ddde±dd bzw. [-]d.dddE±dd; Exponent enthält mindestens 2 Ziffern

Tabelle 3.5: Die bei *printf* und *fprintf* möglichen Umwandlungszeichen

Zeichen	Wert des Arguments wird ausgegeben....
g,G	im e - bzw. E -Format, wenn Exponent <-4 oder >= Genauigkeit ist, sonst im f -Format
c	als Zeichen (unsigned char)
s	als Zeichenkette
p	als Zeigerwert (Sequenz von druckbaren Zeichen)
n	keine Ausgabe; entsprechendes Argument sollte Zeiger auf Ganzzahl sein. An diese Adresse wird Anzahl der bisher ausgegebenen Zeichen geschrieben.
%	Es wird %- Zeichen ausgegeben und kein Argument ausgewertet; nur als %% angeben

Tabelle 3.5: Die bei printf und fprintf möglichen Umwandlungszeichen

Formatierungszeichen

Die Tabelle 3.6 zeigt alle bei printf und fprintf mögliche Formatierungszeichen.

Formatierungszeichen	Bedeutung
-	linksbündige Justierung
+	Ausgabe des Vorzeichens '+' oder '-'
Leerzeichen	Falls 1. Zeichen des Arguments kein Vorzeichen ist, wird Leerzeichen ausgegeben
0	Bei einer numerischen Ausgabe wird mit Nullen bis zur angegeb. Weite aufgefüllt
#	Auswirkung von # hängt vom Umwandlungszeichen ab: bei o bzw. x, X Wert mit vorangestelltem 0 bzw. 0x ausgeben bei e,E,f Wert mit Dezimalpunkt, sogar wenn keine Nachkommastellen existieren bei g,G Wert mit Dezimalpunkt (überflüssige Nachkommanulln mit-ausgeben)

Tabelle 3.6: Die bei printf und fprintf möglichen Formatierungszeichen

Weite

gibt die Mindestanzahl der auszugebenden Stellen an. Wenn der umgewandelte Wert weniger Zeichen als Weite hat, so wird er links (rechts bei Linksjustierung) mit Leerzeichen oder Nullen (wenn Formatierungszeichen 0 angegeben ist) aufgefüllt. Erlaubte Angaben für Weite sind in der Tabelle 3.7 zusammengefaßt.

Weite-Angabe	Bedeutung
Zahl <i>n</i>	Mindestens <i>n</i> Stellen werden ausgegeben. Falls der Wert des entsprechenden Arguments weniger Stellen als <i>n</i> besitzt, dann werden dennoch <i>n</i> Stellen ausgegeben.
*	Wert des nächsten Arguments in Argumentenliste (muß ganzzahlig sein) legt <i>Weite</i> fest. Falls Wert dieses Argument negativ, wird linksbündige Justierung vorgenommen.

Tabelle 3.7: Die bei *printf* und *fprintf* möglichen *Weite*-Angaben

Niemals bewirkt eine nicht vorhandene oder zu kleine *Weite*-Angabe, daß Zeichen nicht ausgegeben werden. Falls das Ergebnis einer Umwandlung mehr Zeichen enthält als *Weite* vorgibt, dann werden trotzdem alle Zeichen ausgegeben.

Genauigkeit

Die Genauigkeit wird mit *.ganzzahl* angegeben. Die Auswirkung hängt vom angegebenen Umwandlungszeichen ab (siehe Tabelle 3.8).

Umwandlungszeichen	Genauigkeit legt folgendes fest
d,i,o,u,x,X	Mindestzahl von auszugebenden Ziffern
e,E,f	Zahl der auszugebenden Nachkommastellen
g,G	maximale Zahl von auszugebenden Ziffern
s	maximale Zahl von auszugebenden Zeichen
.*	das nächste Argument (muß ganzzahlig sein) in Argumentenliste legt Genauigkeit fest; ist Wert dieses Arguments negativ, wird diese Genauigkeitsangabe ignoriert
sonstige	undefiniertes Verhalten

Tabelle 3.8: Die bei *printf* und *fprintf* möglichen Genauigkeitsangaben

Längenangabe

Tabelle 3.9 zeigt die möglichen Längenangaben und ihre Auswirkung für die einzelnen Umwandlungszeichen.

Längenangabe	Auswirkung
h	für Umwandlungszeichen d,i,o,u,x,X wird entspr. Argument als short -Wert behandelt beim Umwandlungszeichen n wird Argument als »Zeiger auf short int « behandelt

Tabelle 3.9: Die bei *printf* und *fprintf* möglichen Längenangaben

Längenangabe	Auswirkung
l	für Umwandlungszeichen d,i,o,u,x,X wird entspr. Argument als long -Wert behandelt beim Umwandlungszeichen n wird Argument als »Zeiger auf long int « behandelt
L	für Umwandlungszeichen e,E,f,g,G wird entspr. Argument als long double -Wert behandelt

Tabelle 3.9: Die bei printf und fprintf möglichen Längenangaben

Falls h, l oder L mit einem anderen Umwandlungszeichen, als in Tabelle 3.9 angegeben, kombiniert wird, so liegt undefiniertes Verhalten vor.

Beispiel

Demonstrationsprogramme zu fprintf

Programm 3.8 (fprintf1.c) demonstriert die Wirkungsweise verschiedener Umwandlungszeichen bei printf bzw. fprintf.

```
#include <stdio.h>

int
main(void)
{
    int    ganz1 = 125,
           ganz2 = -19893;
    float  gleit1 = 1.23456789,
           gleit2 = 2.3e-5;

    printf("Demonstration zu den %s\n", "Umwandlungszeichen");
    printf("=====\n\n");

    printf("(1) dezimal:      ganz1=%d, ganz2=%i\n", ganz1, ganz2);
    printf("(2) oktal:       ganz1=%o, ganz2=%o\n", ganz1, ganz2);
    printf("(3) hexadezimal:   ganz1=%x, ganz2=%X\n", ganz1, ganz2);
    printf("(4) als unsigned-Wert: ganz1=%u, ganz2=%u\n", ganz1, ganz2);
    printf("(5) als char-Zeichen: ganz1=%c, ganz2=%c\n\n", ganz1, ganz2);

    printf("(6) f:    gleit1=%f, gleit2=%f\n", gleit1, gleit2);
    printf("(7) e,E:  gleit1=%e, gleit2=%E\n", gleit1, gleit2);
    printf("(8) g,G:  gleit1=%g, gleit2=%G\n\n", gleit1, gleit2);

    printf("(9) Adresse von ganz1=%p, Adresse von gleit2=%p\n\n",&ganz1,&gleit2);

    printf("(10) Das Prozentzeichen %%\n\n", &ganz2);
    printf("(11) ganz2 = %d\n", ganz2);
}
```

Programm 3.8 (fprintf1.c): Verschiedene Umwandlungszeichen bei printf bzw. fprintf

Dieses Programm 3.8 (fprintf1.c) liefert z.B. die folgende Ausgabe:

Demonstration zu den Umwandlungszeichen

```
=====

(1) dezimal:                ganz1=125, ganz2=-19893
(2) oktal:                  ganz1=175, ganz2=131113 [evtl.: ganz2=3777731113]
(3) hexadezimal:           ganz1=7d, ganz2=B24B          [evtl.: ganz2=FFFFB24B]
(4) als unsigned-Wert:     ganz1=125, ganz2=45643 [evtl.: ganz2=4294947403]
(5) als char-Zeichen:      ganz1=}, ganz2=K

(6) f:   gleit1=1.234568, gleit2=0.000023
(7) e,E: gleit1=1.23457e+00, gleit2=2.30000E-05
(8) g,G: gleit1=1.23457, gleit2=2.3E-05

(9) Adresse von ganz1=0xbffffda4, Adresse von gleit2=0xbffffd98

(10) Das Prozentzeichen %
(11) ganz2 = 25
```

Das folgende Programm 3.9 (fprintf2.c) ist ein weiteres Demonstrationsbeispiel für die Wirkungsweise verschiedener Formatierungszeichen und Weite-Angaben bei **printf** bzw. **fprintf**.

```
#include <stdio.h>

int
main(void)
{
    int    ganz1 = 125,
           ganz2 = -19893,
           ganz3 = 20;
    float  gleit1 = 1.23456789,
           gleit2 = 2.3e-5;

    printf("Demonstration zu den %s\n", "Formatierungszeichen und Weite");
    printf("=====\n\n");

    printf("(1) |%20d|    |%-+20d|\n", ganz1, ganz2);
    printf("(2) |%020o|    |%-020o|\n", ganz1, ganz2);
    printf("(3) |%#20x|    |%#20X|\n", ganz1, ganz2);
    printf("(4) |%+20i|    |%20u|\n", ganz1, ganz2);
    printf("(5) |%#-*x|    |%+*u|\n\n", ganz3, ganz1, 20, ganz2);

    printf("(6) |%-20f|    |%20f|\n", gleit1, gleit2);
    printf("(7) |%+-20f|    |%020f|\n", gleit1, gleit2);
    printf("(8) |%+#20g|    |%-#20g|\n", gleit1, gleit2);
    printf("(9) |%+#20f|    |%-#20f|\n", gleit1, gleit2);
    printf("(10) |%+#*e|    |%-#*E|\n", ganz3, gleit1, 20, gleit2);
}
```

Programm 3.9 (**fprintf2.c**): Verschiedene Formatierungs- und Weite-Angaben bei **printf** bzw. **fprintf**

Das Programm 3.9 (fprintf2.c) liefert z.B. die folgende Ausgabe:

Demonstration zu den Formatierungszeichen und Weite

=====

(1)		125		-19893				
(2)		0000000000000000175		131113		[evtl.:	37777731113	
(3)		0x7d				0XB24B	[evtl.:	0xFFFFB24B
(4)		+125				45643	[evtl.:	4294947403
(5)		0x7d				+45643	[evtl.:	4294947403
(6)		1.234568				0.000023		
(7)		+1.234568				000000000000.000023		
(8)		+1.23457				2.30000e-05		
(9)		+1.234568				0.000023		
(10)		+1.23457e+00				2.30000E-05		

Das folgende Programm 3.10 (fprintf3.c) demonstriert die Wirkungsweise unterschiedlicher Formatangaben für Strings bei **printf** bzw. **fprintf**:

```
#include <stdio.h>

int
main(void)
{
    printf("|%s|\n", "Kettenglied");
    printf("|%20s|\n", "Kettenglied");
    printf("|%-20s|\n", "Kettenglied");
    printf("|%-10s|\n", "Kettenglied");
    printf("|%20.8s|\n", "Kettenglied");
    printf("|%-20.7s|\n", "Kettenglied");
    printf("|%020s|\n", "Kettenglied");
    printf("|%.6s|\n", "Kettenglied");
    printf("|%-020s|\n", "Kettenglied");
}
```

Programm 3.10 (**fprintf3.c**): Unterschiedliche Formatangaben für Strings bei **printf** bzw. **fprintf**

Das Programm 3.10 (fprintf3.c) liefert z.B. die folgende Ausgabe:

```
|Kettenglied|
|      Kettenglied|
|Kettenglied      |
|Kettenglied|
|      Kettengl|
|Ketteng      |
|      Kettenglied|
|Ketten|
|Kettenglied      |
```

3.4.9 sscanf – Formatiertes Lesen aus einem String

Um formatiert aus einem String zu lesen, steht die Funktion **sscanf** zur Verfügung.

```
#include <stdio.h>
```

```
int sscanf(const char *puffer, const char *format, ...);
```

gibt zurück: Anzahl der gelesenen Eingabeeinheiten (bei Erfolg);
EOF bei Dateiende oder Fehler vor einer Umwandlung

Diese Funktion **sscanf** ist äquivalent mit Funktion **fscanf**, außer daß anstelle eines FILE-Zeigerarguments das Argument *puffer* anzugeben ist, das eine Speicheradresse festlegt, von der die Eingabezeichen zu lesen sind.

Das Erreichen des Zeichenkettenendes ist äquivalent mit dem Lesen des EOF-Zeichens bei der Funktion **fscanf**.

Hinweis

Die möglichen *format*-Angaben sind ausführlich bei **fscanf** auf den vorangegangenen Seiten beschrieben.

sscanf wird häufig verwendet, um Zahlen, die in Stringform vorliegen, in numerische Werte umzuwandeln.

3.4.10 sprintf – Formatiertes Schreiben in einen String

Um formatiert in einen String zu schreiben, steht die Funktion **sprintf** zur Verfügung.

```
#include <stdio.h>
```

```
int sprintf (char *puffer, const char *format, ...);
```

gibt zurück: Anzahl der nach *puffer* geschriebenen Zeichen

Diese Funktion **sprintf** ist äquivalent mit der Funktion **fprintf**, außer daß anstelle eines FILE-Zeigerarguments das Argument *puffer* anzugeben ist, das eine Speicheradresse festlegt, an die die Ausgabe zu schreiben ist. Ein `\0` wird automatisch an das Ende der geschriebenen Zeichenkette angehängt.

Die Funktion **sprintf** gibt die Zahl der nach *puffer* geschriebenen Zeichen (abschließendes `\0` nicht mitgezählt) als Funktionswert zurück.

Hinweis

Die möglichen *format*-Angaben sind ausführlich bei **fprintf** auf den vorangegangenen Seiten beschrieben.

Häufige Anwendung findet diese Funktion, wenn ganze Zahlen oder Gleitpunktzahlen in Strings umzuwandeln sind, wie z.B.:

```
char    text[100];
float   summe;
.....
sprintf(text, "Der Wert betraegt %.2f DM", summe);
```

3.4.11 **vprintf** und **vfprintf** – Formatiertes Schreiben auf stdout oder in eine Datei (Argumentzeiger)

Um formatiert auf die Standardausgabe oder in eine Datei zu schreiben, stehen mit **vprintf** und **vfprintf** zwei weitere Funktionen zur Verfügung.

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *format, va_list arg);
int vfprintf(FILE *fz, const char *format, va_list arg);
```

beide geben zurück: Anzahl der geschriebenen Zeichen (bei Erfolg); negativer Wert bei Ausgabefehler

Die Funktion **vprintf** ist äquivalent zu

```
fprintf(stdout, format, arg);
```

Diese beiden Funktionen **vprintf** und **vfprintf** sind äquivalent mit den Funktionen **printf** und **fprintf**, wobei allerdings die variable lange Argumentliste durch einen Parameter *arg* (vom Typ *va_list*) ersetzt wird.

arg sollte zuvor durch Aufruf des Makros **va_start** (und eventuell nachfolgenden Aufrufen von **va_arg**) initialisiert worden sein. **vprintf** und **vfprintf** rufen nicht das Makro **va_end** auf.

Hinweis

Bei Verwendung dieser Funktionen sollte

```
#include <stdarg.h>
```

angegeben sein. Es ist darauf hinzuweisen, daß die Routinen aus *<stdarg.h>* sich von den Routinen aus *<varargs.h>* unterscheiden. *<varargs.h>* wird bei SVR3 und früheren Versionen angeboten.

vprintf und **vfprintf** lassen sich vorzüglich in einer allgemeinen Fehlermeldungsroutine verwenden (siehe auch Programm 2.3 in Kapitel 2.3).

3.4.12 vsprintf – Formatiertes Schreiben in einen String (Argumentzeiger)

Um formatiert in einen String zu schreiben, steht mit **vsprintf** eine weitere Funktion zur Verfügung.

```
#include <stdarg.h>
```

```
#include <stdio.h>
```

```
int vsprintf(char *puffer, const char *format, va_list arg);
```

gibt zurück: Anzahl der nach *puffer* geschriebenen Zeichen

Diese Funktion **vsprintf** ist äquivalent mit der Funktion **sprintf** (siehe vorher), wobei allerdings die variable lange Argumentliste durch einen Parameter *arg* (vom Typ *va_list*) ersetzt wird.

arg sollte zuvor durch Aufruf des Makros **va_start** (und eventuell nachfolgenden Aufrufen von **va_arg**) initialisiert worden sein. **vsprintf** ruft nicht das Makro **va_end** auf.

Hinweis

Bei Verwendung dieser Funktion sollte

```
#include <stdarg.h>
```

angegeben sein. Es ist darauf hinzuweisen, daß die Routinen aus *<stdarg.h>* sich von den Routinen aus *<varargs.h>* unterscheiden. *<varargs.h>* wird bei SVR3 und früheren Versionen angeboten.

Die möglichen *format*-Angaben sind ausführlich bei **fprintf** auf den vorangegangenen Seiten beschrieben.

3.4.13 fread und fwrite – Binäres Lesen und Schreiben ganzer Blöcke

Wenn man ganze Blöcke von binären Daten lesen muß, so ist weder das zeilenweise Einlesen brauchbar, da für **fgets** die Zeichen *\0* und *\n* eine besondere Bedeutung haben, noch ist es sehr effizient, die Daten Zeichen für Zeichen mit **getc** oder **fgetc** einzulesen. Um ganze Blöcke von binären Daten zu lesen oder zu schreiben, stehen die Funktionen **fread** und **fwrite** zur Verfügung

```
#include <stdio.h>

size_t fread(void *puffer, size_t blockgroesse, size_t blockzahl, FILE *fz);

size_t fwrite(const void *puffer, size_t blockgroesse,
              size_t blockzahl, FILE *fz);
```

beide geben zurück: Anzahl der gelesenen bzw. geschriebenen Blöcke

fread liest bis zu `blockzahl` Objekte, jedes mit `blockgroesse` Byte, von der Datei (Stream), die mit `fz` verbunden ist, in den Speicherbereich, der mit `puffer` adressiert ist.

fwrite schreibt bis zu `blockzahl` Objekte, jedes mit `blockgroesse` Byte, von der Adresse `puffer` in die Datei (Stream), die mit `fz` verbunden ist.

fread und **fwrite** liefern als Funktionswert die *wirklich gelesene bzw. geschriebene Anzahl von Objekten*, die kleiner als `blockzahl` sein kann, wenn ein Lese- oder Schreibfehler aufgetreten ist oder im Falle von **fread** das Dateiende erreicht wurde. Der Aufrufer kann den Grund für weniger gelesene Blöcke mit **feof** bzw. **ferror** in Erfahrung bringen.

Typische Anwendung

Typische Anwendungen für diese Funktionen **fread** und **fwrite** sind:

- Einlesen und Schreiben eines ganzen Arrays, wie z.B.

```
double werte[100];

/*---- Arrayelemente werte[90], werte[91], ..., werte[99]
      mit den nächsten 10 double-Werten von Stream fz füllen */
if (fread(&werte[90], sizeof(double), 10, fz) != 10)
    fehler_meld(FATAL_SYS, "Fehler bei fread");
```

- Einlesen oder Schreiben einer ganzen Struktur, wie z.B.

```
struct {
    char vorname[20];
    char nachname[40];
    int alter;
} person;

/*---- Inhalt der Strukturvariable person auf Datei schreiben */
if (fwrite(&person, sizeof(person), 1, fz) != 1)
    fehler_meld(FATAL_SYS, "Fehler bei fwrite");
```

Hinweis

Bei `size_t` handelt es sich um einen `<stdio.h>` definierten vorzeichenlosen Ganzzahl-Datentyp, der für das Ergebnis des **sizeof**-Operators eingeführt wurde. Meist wird `size_t` als Typ für Funktionsargumente verwendet, die Größenangaben repräsentieren, wie z.B.:

```
void *malloc(size_t groesse);
```

Wenn für `blockzahl` oder `blockgroesse` der Wert 0 angegeben wurde, so liefert ***fread*** 0, der Speicherbereich ab Adresse `puffer` bleibt unverändert.

Beispiel

Hexadezimale Ausgabe einer Datei

Das folgende Programm 3.11 (`hexd.c`) gibt den Inhalt einer Datei Byte für Byte in Hexa-Mustern aus, wobei es rechts dazu die entsprechenden ASCII-Zeichen angibt, soweit diese darstellbar sind, andernfalls wird nur ein Punkt für dieses Zeichen angegeben.

```
#include "eighdr.h"
#include <ctype.h>

static void hex_druck(FILE *fz, char *s);

int
main( int argc, char *argv[] )
{
    FILE      *fz;
    int        i;

    if (argc < 2)
        fehler_meld(FATAL, "usage: %s dateil .....", argv[0]);

    for (i=1; i<argc; i++) {
        if ((fz=fopen(argv[i],"rb")) == NULL)
            fehler_meld(FATAL_SYS, "Kann %s nicht eroeffnen\n", argv[i]);
        else {
            hex_druck(fz,argv[i]);
            fclose(fz);
        }
    }
}

static void hex_druck( FILE *fz, char *s )
{
    unsigned char  puffer[16];
    int            gelesen, i;
    long           gesamt=0;

    printf("----%s----\n", s);

    while ( (gelesen=fread(puffer, 1, 16, fz)) > 0) {
        printf(" %06x ", gesamt);

        /*----- Ausgabe des Hexa-Musters */
        for (i=0 ; i<16 ; i++) {
            if (i < gelesen) {
                printf(" %02x", puffer[i]);
                if (iscntrl(puffer[i])) /* Falls puffer[i] ein Steuerzeichen */
                    puffer[i] = '.'; /* -> dann wird es mit . dargestellt */
            }
        }
    }
}
```



```

    } else {
        fputs(" ", stdout);
        puffer[i] = ' ';
    }
    if (i==7) /*--- Trennzeichen nach 8 Hexa-Bytemustern ausgeben */
        putchar(' ');
}

/*----- Ausgabe des zum Hexa-Muster gehoerigen Texts */
printf(" |%16.16s|\n", puffer);
gesamt += gelesen;
}
}

```

Programm 3.11 (**hexd.c**): Hexa-Dump einer Datei

Nachdem man dieses Programm 3.11 (**hexd.c**) kompiliert und gelinkt hat

```
cc -o hexd hexd.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ hexd /usr/bin/write
----/usr/bin/write----
000000 07 01 64 00 40 0d 00 00 f8 00 00 00 08 00 00 00 |..d.@.....|
000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000020 e8 f7 0b 00 00 b8 2d 00 00 00 bb 00 00 00 00 cd |.....~.....|
000030 80 a3 5c 0b 09 60 8b 44 24 08 a3 34 0b 09 60 0f |..\..\`D$.4..\`|
000040 b7 05 d0 0d 00 00 50 e8 b8 0c 00 00 83 c4 04 e8 |.....P.....|
000050 00 01 00 00 50 e8 96 03 00 60 5b b8 01 00 00 00 |....P....`[....|
000060 cd 80 eb f7 90 90 90 90 90 90 90 90 90 90 90 90 |.....|
000070 00 00 00 00 77 72 69 74 65 3a 20 63 61 6e 27 74 |....write: can't|
000080 20 66 69 6e 64 20 79 6f 75 72 20 74 74 79 0a 00 | find your tty..|
000090 77 72 69 74 65 3a 20 63 61 6e 27 74 20 66 69 6e |write: can't fin|
0000a0 64 20 79 6f 75 72 20 74 74 79 27 73 20 6e 61 6d |d your tty's nam|
0000b0 65 0a 00 77 72 69 74 65 3a 20 79 6f 75 20 68 61 |e..write: you ha|
0000c0 76 65 20 77 72 69 74 65 20 70 65 72 6d 69 73 73 |ve write permiss|
0000d0 69 6f 6e 20 74 75 72 6e 65 64 20 6f 66 66 2e 0a |ion turned off..|
0000e0 00 2f 64 65 76 2f 00 77 72 69 74 65 3a 20 25 73 |./dev/.write: %s|
0000f0 20 69 73 20 6e 6f 74 20 6c 6f 67 67 65 64 20 69 | is not logged i|
000100 6e 20 6f 6e 20 25 73 2e 0a 00 77 72 69 74 65 3a |n on %s...write:|
000110 20 25 73 20 68 61 73 20 6d 65 73 73 61 67 65 73 | %s has messages|
000120 20 64 69 73 61 62 6c 65 64 20 6f 6e 20 25 73 0a | disabled on %s.|
000130 00 75 73 61 67 65 3a 20 77 72 69 74 65 20 75 73 |.usage: write us|
000140 65 72 20 5b 74 74 79 5d 0a 00 00 00 00 00 00 00 |er [tty].....|
000150 00 00 00 00 55 89 e5 81 ec 0c 04 00 00 57 56 53 |....U.....WVS|
000160 e8 cb 09 00 00 68 3c 05 09 60 e8 09 03 00 60 50 |....h<..\`....`P|
::: ::::::::::::::: :::::::::::::::
::: ::::::::::::::: :::::::::::::::
::: ::::::::::::::: :::::::::::::::
000de0 39 30 00 00 cc 0d 00 00 00 00 00 00 00 00 00 00 |90.....|
000df0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000e00 24 0d 00 00 2e 0d 00 00 00 00 00 60 94 01 04 00 |$......`....|

```

```

000e10  00 f0 08 60 02 00 00 00  3c 0d 00 00 e0 0d 00 00 |...`....<.....|
000e20  00 00 00 00 01 00 00 00  f8 3f 00 60 00 00 00 00 |.....?.`.....|
000e30  03 00 00 00 40 0d 00 00  2c 0e 00 00 f0 0d 00 00 |....@.....|
000e40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|
000e50  00 00 00 00 00 00 00 00  04 00 00 00          |.....|
$

```

3.4.14 Unterschiedliches Zeitverhalten von Standard-E/A-Funktionen

Sind große Datenmengen in eine Datei zu schreiben, so ist es wichtig zu wissen, wie effizient die einzelnen E/A-Routinen arbeiten. Dazu werden nachfolgend drei Programme vorgestellt, die alle zwar das gleiche leisten (Kopieren von `stdin` nach `stdout`), aber unter Verwendung verschiedener E/A-Routinen unterschiedlich verwirklicht wurden:

Programm 3.12 (`copy2.c`) mit `getc` und `putc`

Programm 3.13 (`copy3.c`) mit `gets` und `puts`

Programm 3.14 (`copy4.c`) mit `fread` und `fwrite`

```

#include "eighdr.h"

int
main(void)
{
    int    zeich;

    while ( (zeich=getc(stdin)) != EOF)
        if (putc(zeich, stdout) == EOF)
            fehler_meld(FATAL_SYS, "Fehler bei putc");

    if (ferror(stdin))
        fehler_meld(FATAL_SYS, "Fehler bei getc");

    exit(0);
}

```

Programm 3.12 (`copy2.c`): Standardeingabe auf Standardausgabe kopieren (mit **`getc`** und **`putc`**)

```

#include "eighdr.h"

int
main(void)
{
    char    puffer[MAX_ZEICHEN];

    while (fgets(puffer, MAX_ZEICHEN, stdin) != NULL)
        if (fputs(puffer, stdout) == EOF)
            fehler_meld(FATAL_SYS, "Fehler bei fputs");
}

```

```

    if (ferror(stdin))
        fehler_meld(FATAL_SYS, "Fehler bei fgets");

    exit(0);
}

```

Programm 3.13 (copy3.c): Standardeingabe auf Standardausgabe kopieren (mit **fgets** und **fputs**)

```

#include "eighdr.h"

int
main(void)
{
    int    n;
    char   puffer[MAX_ZEICHEN];

    while ( (n = fread(puffer, 1, MAX_ZEICHEN, stdin)) > 0)
        if (fwrite(puffer, 1, n, stdout) == 0)
            fehler_meld(FATAL_SYS, "Fehler bei fwrite");

    if (ferror(stdin))
        fehler_meld(FATAL_SYS, "Fehler bei fread");

    exit(0);
}

```

Programm 3.14 (copy4.c): Standardeingabe auf Standardausgabe kopieren (mit **fread** und **fwrite**)

Wenn wir mit diesen drei Programmen nun die gleiche Datei (ca. 5 Megabyte groß mit etwa 150000 Zeilen) kopieren, können wir das unterschiedliche Zeitverhalten der einzelnen E/A-Funktionen messen. Die Ergebnisse sind in Tabelle 3.10 zusammengefaßt:

Funktion	User-CPU (in Sek.)	System-CPU (in Sek.)
getc, putc (copy2.c)	8,5	7,8
fgets, fputs (copy3.c)	5,4	7,9
fread, fwrite (copy4.c)	0,8	7,8

Tabelle 3.10: Benötigte Zeiten für das Kopieren von etwa 150000 Zeilen mit ca. 5 Megabyte

Die Systemzeit (System CPU) ist bei allen drei Programmen nahezu gleich, was sich auch leicht erklären läßt, da die gleiche Anzahl von Kernfunktionen aufgerufen wird.

In der Benutzerzeit (User CPU) ergeben sich dagegen erhebliche Unterschiede:

- Die Umsetzung mit **getc** und **putc** (Programm copy2.c) ist die langsamste, was sich damit erklären läßt, daß dort die für das Kopieren zuständige Schleife ca. 5,25 Millionen Mal durchlaufen werden muß.
- Die Umsetzung mit **fgets** und **fputs** (Programm copy3.c) ist schon etwas schneller, weil dort die Kopierschleife nur für jede Zeile, also ca. 150.000 Mal durchlaufen wird.

- Am schnellsten ist die Umsetzung mit **fread** und **fwrite** (Programm `copy4.c`), weil dort die Kopierschleife nur ca. 1250 Mal (5 Megabyte geteilt durch die Puffergröße, die hier 4096 ist) durchlaufen wird.

Diese hier gegebenen Zeiten sind natürlich abhängig vom System, auf dem diese Programme ablaufen. Die Ergebnisse hängen sehr stark von der jeweiligen Unix-Implementierung und den Hardwarevoraussetzungen ab. Nichtsdestoweniger sollten sie den Programmierer dahingehend sensibilisieren, daß die Verwendung der verschiedenen Routinen darüber entscheidet, wie schnell bzw. langsam ein Programm sein wird.

Auf Zeitmessungen dieser Art werden wir in Kapitel 4.5 bei der Vorstellung der elementaren E/A-Funktionen, die, abhängig von der gewählten Puffergröße, meist noch besseres Zeitverhalten zeigen, zurückkommen.

3.5 Pufferung

Die Standard-E/A-Funktionen arbeiten mit einem internen Puffer, um mit möglichst wenigen physikalischen Lese- und Schreiboperationen, die meist zeitintensiv sind, auszukommen. Zum Lesen und Schreiben verwenden sie dabei intern die in Kapitel 4.3 beschriebenen elementaren Funktionen **read** und **write**.

Der Anwender kann dabei für die Standard-E/A-Funktionen unterschiedliche Pufferarten einstellen. In `<stdio.h>` sind dazu drei verschiedene Konstanten definiert.

3.5.1 _IOFBF – Vollpufferung

Bei dieser Pufferungsart findet das eigentliche Lesen bzw. Schreiben in einer Datei (Stream) immer erst dann statt, wenn der entsprechende Puffer gefüllt ist. Lesen und Schreiben in Dateien, die sich auf der Festplatte oder einer Diskette befinden, wird normalerweise mit dieser Form der Pufferung durchgeführt. Dabei wird der Puffer normalerweise bei der ersten E/A-Operation von der betreffenden Standard-E/A-Routine durch einen **malloc**-Aufruf angelegt. Die Funktion **malloc** wird in Kapitel 9.4 beschrieben.

3.5.2 _IOLBF – Zeilenpufferung

Bei dieser Pufferungsart findet das eigentliche Lesen bzw. Schreiben in einer Datei (Stream) immer erst dann statt, wenn ein `\n` gelesen oder geschrieben wird. Bei dieser Pufferungsart bewirkt z.B. das Schreiben einzelner Zeichen mit **fputc**, daß diese Zeichen zunächst im Puffer abgelegt und erst beim Zeichen `\n` wirklich in die entsprechende Datei (Stream) physikalisch geschrieben werden. Zeilenpufferung wird immer dann verwendet, wenn Ein- und Ausgabe auf ein Terminal (wie `stdin` und `stdout`) stattfindet.

Hinweis

Wenn bei der Zeilenpufferung der Puffer gefüllt wird, bevor ein `\n` auftritt, so findet trotzdem die entsprechende E/A-Operation statt, um ein Überlaufen zu verhindern.

3.5.3 _IONBF – Keine Pufferung

Bei dieser Pufferungsart erfolgen die E/A-Operationen direkt ohne Dazwischenschalten eines Puffers. Schreibt man z.B. 10 Zeichen mit der Funktion **fputs**, so werden diese 10 Zeichen sofort in die entsprechende Datei (Stream) geschrieben.

Das Schreiben auf `stderr` ist z.B. normalerweise ungepuffert, um Fehler- oder Diagnosemeldungen so schnell wie möglich auszugeben, unabhängig davon, ob sie Neue-Zeile-Zeichen enthalten oder nicht.

3.5.4 Voreingestellte Pufferungsarten

ANSI C legt bezüglich der Pufferung folgende Regeln fest:

- ▶ Für Standardeingabe (`stdin`) und Standardausgabe (`stdout`) darf nur dann Vollpufferung stattfinden, wenn sie nicht auf ein interaktives Gerät (wie Terminal) eingestellt sind.
- ▶ Für Standardfehlerausgabe (`stderr`) darf niemals Vollpufferung stattfinden.

In SVR4 wurden diese Regeln wie folgt umgesetzt:

- ▶ `stderr` ist immer ungepuffert.
- ▶ Alle anderen Streams (Dateien) sind grundsätzlich zeilengepuffert, wenn sie auf ein Terminal eingestellt sind, ansonsten sind sie vollgepuffert.

Um andere Pufferungsarten für Streams (Dateien) einzustellen, stehen die beiden folgenden Funktionen zur Verfügung.

3.5.5 setbuf und setvbuf – Einstellen der Pufferungsart

Um die Pufferungsart für Dateien (Streams) festzulegen, die mit **fopen**, **freopen** oder **fdopen** geöffnet wurden, stehen die beiden Funktionen **setbuf** und **setvbuf** zur Verfügung.

```
#include <stdio.h>

void setbuf(FILE *fz, char *puffer);

int setvbuf(FILE *fz, char *puffer, int modus, size_t puffgroesse);

gibt zurück: 0 (bei Erfolg); Wert verschieden von 0 bei Fehler
```

Diese beiden Funktionen müssen aufgerufen werden, *nachdem* die Datei `fz` geöffnet wurde und *bevor* eine Lese- oder Schreiboperation für diese Datei stattgefunden hat.

setbuf

Mit **setbuf** kann die Pufferung ein- oder ausgeschaltet werden.

Um die Pufferung einzuschalten, muß die Adresse eines Puffers (Argument `puffer`) angegeben werden, der groß genug ist, um `BUFSIZ` Byte aufzunehmen. Normalerweise wird dann Vollpufferung eingeschaltet, wenn auch einige Systeme für Terminals Zeilenpufferung verwenden. `BUFSIZ` ist eine Konstante, die in `<stdio.h>` definiert ist (ANSI C garantiert eine Mindestgröße von 256 Byte).

Um die Pufferung auszuschalten, ist für `puffer` die Zeigerkonstante `NULL` anzugeben.

Mit der Ausnahme, daß **`setbuf`** keinen Wert zurückgibt, ist diese Funktion äquivalent mit dem Aufruf

```
(void)setvbuf(fz, puffer, _IOFBF, BUFSIZ);
```

oder falls `puffer` ein Nullzeiger ist:

```
(void)setvbuf(fz, NULL, _IONBF, BUFSIZ);
```

Eigentlich ist somit **`setbuf`** durch **`setvbuf`** abgedeckt, aber aus Kompatibilitätsgründen zu »Alt-C« wurde diese Funktion in ANSI C erhalten.

`setvbuf`

Mit **`setvbuf`** kann explizit die gewünschte Pufferungsart eingestellt werden. Dazu ist für das Argument `modus` eine der folgenden Konstanten anzugeben:

<code>_IOFBF</code>	Voll-Pufferung
<code>_IOLBF</code>	Zeilen-Pufferung
<code>_IONBF</code>	Keine Pufferung

Bei `_IONBF` werden die Argumente `puffer` und `puffgroesse` ignoriert.

Bei `_IOFBF` und `_IOLBF` wird über `puffer` die Pufferadresse und über `puffgroesse` die Größe dieses Puffers der Funktion **`setvbuf`** mitgeteilt. Falls für `puffer` die Zeigerkonstante `NULL` angegeben wird, so verwenden die Standard-E/A-Funktionen einen eigenen Puffer mit einer geeigneten Größe, der in der Komponente `st_blksize` der Struktur `stat` angegeben ist (siehe Kapitel 5.1). Sollte dieser Wert nicht verfügbar sein, weil der Stream z.B. einem Gerät oder einer Pipe zugeordnet ist, so wird als Puffergröße `BUFSIZ` gewählt.

Falls diese Funktion einen Rückgabewert verschieden von 0 liefert, dann wurde entweder ein unerlaubter Wert für das Argument `modus` angegeben oder die geforderte Pufferung konnte aus welchen Gründen auch immer nicht eingestellt werden.

Hinweis

Ein typischer Fehler ist die lokale Deklaration eines Arrays in einer Funktion, um dieses Array als Puffer zu verwenden. Wird dann die entsprechende Datei (Stream) in dieser Funktion nicht geschlossen, sondern in anderen Funktionen mit dieser geöffneten Datei (Stream) weitergearbeitet, so verwenden die dortigen E/A-Operationen eine nicht mehr gültige Adresse zur Pufferung, was zwangsläufig zum Überschreiben von fremdem Speicherplatz führt.

Zusammenfassung der Pufferungsarten für `setbuf` und `setvbuf`

Die Tabelle 3.11 zeigt die möglichen Pufferungsarten der beiden Funktionen **`setbuf`** und **`setvbuf`** im Überblick.

<i>Funktion</i>	<i>modus</i>	<i>puffer</i>	<i>Puffer und Puffergröße</i>	<i>Pufferungsart</i>
setbuf		Nicht NULL	Benutzerpuffer der Länge BUFSIZ	Voll- od. Zeilenpufferung
		NULL	kein Puffer	Keine Pufferung
setvbuf	_IOFBF	Nicht NULL	Benutzerpuffer der angegeb. Länge	Vollpufferung
		NULL	Systempuffer mit geeigneter Länge	
setvbuf	_IOLBF	Nicht NULL	Benutzerpuffer der angegeb. Länge	Zeilenpufferung
		NULL	Systempuffer mit geeigneter Länge	
setvbuf	_IONBF	ignoriert	kein Puffer	Keine Pufferung

Tabelle 3.11: Einstellung der Pufferungsart mit `setbuf` oder `setvbuf`

3.5.6 `fflush` – Inhalte von Puffern in eine Datei übertragen

Um die Inhalte von noch nicht geleerten Puffern in eine Datei (Stream) übertragen zu lassen, steht die Funktion **`fflush`** zur Verfügung.

```
#include <stdio.h>

int fflush(FILE *fz);
```

gibt zurück: 0 (bei Erfolg); EOF bei Fehler

Die Funktion **`fflush`** überträgt alle Inhalte von noch nicht geleerten Puffern in die Datei (Stream), der der FILE-Zeiger `fz` zugeordnet ist.

Wird für `fz` ein NULL-Zeiger angegeben, so werden bei ANSI C-Compilern alle Ausgabepuffer (wo die letzte Aktion kein Lesen war) übertragen.

Hinweis

Wenn **`fflush`** auf eine Datei angewendet wird, von der zuletzt gelesen wurde, so liegt undefiniertes Verhalten vor.

Um z.B. alle noch im Standardeingabepuffer befindlichen Zeichen zu entfernen, muß nur

```
fflush(stdin)
```

aufgerufen werden. Diesen Aufruf wendet man z.B. immer dann an, wenn nach dem Lesen von numerischen Werten nun Zeichen einzulesen sind, um das noch im Puffer befindliche `\n` (vom Drücken der Returntaste) zu entfernen.

3.6 Positionieren in Dateien

Um den »Schreib-/Lesezeiger« in einer Datei (Stream) neu zu positionieren oder seine momentane Position zu erfragen, stehen zwei Möglichkeiten zur Verfügung.

fseek und ***ftell***

Diese beiden älteren Funktionen setzen voraus, daß die Position des Schreib-/Lesezeigers durch den Datentyp **long** dargestellt wird.

fsetpos und ***fgetpos***

Diese beiden Funktionen wurden neu von ANSI C eingeführt und verwenden für die Position des Schreib-/Lesezeigers nicht mehr den Datentyp **long**, sondern einen in `<stdio.h>` definierten Datentyp `fpos_t`. Die Verwendung dieser Funktionen macht also ein Programm portabel für andere Systeme.

3.6.1 **fseek** und **ftell** – Positionieren in einer Datei (1. Möglichkeit)

Um den Schreib-/Lesezeiger in einer Datei zu positionieren oder seine momentane Position zu erfragen, stehen die beiden schon in »Alt-C« vorhandenen Funktionen ***fseek*** und ***ftell*** zur Verfügung.

```
#include <stdio.h>
```

```
int fseek(FILE *fz, long offset, int wie);
```

gibt zurück: 0 (bei Erfolg); Wert verschieden von 0 bei Fehler

```
long ftell(FILE *fz);
```

gibt zurück: momentane Position des Schreib-/Lesezeigers (bei Erfolg); -1L bei Fehler

fseek

fseek ermöglicht das Verschieben des Schreib-/Lesezeigers innerhalb der Datei (Stream), der der FILE-Zeiger `fz` momentan zugeordnet ist.

ANSI C unterscheidet, ob diese Funktion auf eine Binärdatei oder eine Textdatei angewendet wird:

Binärdatei

Tabelle 3.12 zeigt die möglichen Angaben für das `wie`-Argument und ihre Bedeutung.

wie-Angabe	Wirkung
SEEK_SET	Schreib-/Lesezeiger vom Dateianfang an um offset Byte versetzen
SEEK_CUR	Schreib-/Lesezeiger von momentanen Position an um offset Byte versetzen
SEEK_END	Schreib-/Lesezeiger vom Dateende an um offset Byte versetzen

Tabelle 3.12: Mögliche Angaben für das *wie*-Argument bei **fseek**

Textdatei

Hier sollte *offset* entweder 0 sein, oder für *offset* sollte ein Wert verwendet werden, der durch einen vorherigen Aufruf von **ftell** (für gleichen Stream *fz*) erhalten wurde, und *wie* sollte immer SEEK_SET (vom Dateianfang an) sein.

Diese Einschränkung für Textdateien gilt jedoch nicht unter Unix, da Unix nicht wie andere Systeme eine gesonderte Darstellung für Textdateien kennt.

fseek setzt die EOF-Marke zurück und macht Auswirkungen, bedingt durch einen **ungetc**-Aufruf (auf gleichen Stream *fz*), rückgängig.

ftell

ftell ermittelt die aktuelle Position des Schreib-/Lesezeigers in der Datei (Stream), der der FILE-Zeiger *fz* zugeordnet ist. Diese Position wird als **long**-Funktionswert geliefert und gibt den Abstand zum Dateianfang in Byte an.

Bei Binärdateien entspricht diese so ermittelte Zahl der Bytezahl ab Dateianfang. Bei Textdateien ist diese Aussage in anderen als Unix-Systemen eventuell nicht gültig.

Beispiel

Hexadump für einen Dateibereich

Das folgende Programm 3.15 (*datbytes.c*) liest zunächst einen Dateinamen ein, bevor es einen Hexadump für die betreffende Datei durchführt. Die Bytenummer, ab der dieser Hexadump durchzuführen ist, ist ebenso einzugeben wie die Bytenummer, bis zu der der Hexadump erfolgen soll. Das Programm wird beendet, wenn der Benutzer bei der Bytenummer, ab der der Hexadump erfolgen soll, den Wert -1 eingibt.

```
#include <limits.h>
#include "eighdr.h"

int
main(void)
{
    FILE      *dz;
    char      dateiname[NAME_MAX]; /*--- evtl.: dateiname[_POSIX_NAME_MAX]; ---*/
    long      von, bis;
    int       zeich;
```

```

fprintf(stderr, "Dateiname? ");
gets(dateiname);

if ( (dz=fopen(dateiname, "r")) == NULL)
    fehler_meld(FATAL_SYS, "kann %s nicht eroeffnen", dateiname);

do {
    fprintf(stderr, "Hexausgabe ab Bytenr (Ende=-1) ? ");
    scanf("%ld", &von);

    if (von >= 0) {
        fseek(dz, von, SEEK_SET);
        fprintf(stderr, "          bis Bytenr ? ");
        scanf("%ld", &bis);

        printf("Hexadump der Datei %s (von Bytenr %ld bis %ld)\n",
            dateiname, von, bis);
        while (von <= bis) {
            if ( (zeich=getc(dz)) != EOF)
                printf("%02x", zeich);
            else if (ferror(dz)) {
                fehler_meld(WARNUNG_SYS,
                    "Fehler beim Lesen aus Datei %s (Bytenr: %ld", dateiname, von);
            } else if (feof(dz)) {
                printf("--EOF--\n");
                break;
            }
            von++;
        }
        printf("\n\n");
        fflush(NULL);
    }
} while (von >= 0);

exit(0);
}

```

Programm 3.15 (*datbytes.c*): Hexadump für einen Ausschnitt einer Datei

3.6.2 fsetpos und fgetpos – Positionieren in einer Datei (2. Möglichkeit)

Um den Schreib-/Lesezeiger in einer Datei zu positionieren oder seine momentane Position zu erfragen, stehen mit **fsetpos** und **fgetpos** zwei weitere Funktionen zur Verfügung.

```
#include <stdio.h>
```

```
int fsetpos(FILE *fz, const fpos_t *pos);
```

```
int fgetpos(FILE *fz, fpos_t *pos);
```

beide geben zurück: 0 (bei Erfolg); Wert verschieden von 0 bei Fehler

fsetpos

fsetpos setzt den Schreib-/Lesezeiger der Datei (Stream), der der FILE-Zeiger *fz* zugeordnet ist, auf die Position, die mit dem Wert, auf den *pos* zeigt, festgelegt wird.

Der Wert, der hier über *pos* übergeben wird, sollte zuvor mit einem Aufruf an die Funktion ***fgetpos*** (für gleiche Datei) ermittelt worden sein.

fsetpos setzt die EOF-Marke zurück und macht Auswirkungen, bedingt durch einen ***ungetc***-Aufruf (auf gleichen Stream *fz*), rückgängig.

fgetpos

fgetpos schreibt die momentane Position des Schreib-/Lesezeigers der Datei (Stream), der der FILE-Zeiger *fz* zugeordnet ist, in den Speicherplatz, auf den *pos* zeigt. Dieser Wert sollte nur als Argument für die Funktion ***fsetpos*** verwendet werden, um den Schreib-/Lesezeiger auf die ursprüngliche Position zurückzusetzen.

3.6.3 **rewind** – Positionieren an den Dateianfang

Um den Schreib-/Lesezeiger auf den Anfang einer Datei zu setzen, bietet ANSI C die Funktion ***rewind*** an:

```
#include <stdio.h>

void rewind(FILE *fz);
```

rewind setzt den Schreib-/Lesezeiger der Datei (Stream), der der FILE-Zeiger *fz* zugeordnet ist, auf den Anfang der Datei. Somit ist

```
rewind(dateizeiger);
```

äquivalent mit

```
(void)fseek(dateizeiger, 0L, SEEK_SET);
```

außer, daß bei ***rewind*** neben der EOF-Marke auch die Fehlermarke mit zurückgesetzt wird.

3.7 Temporäre Dateien

Temporäre Dateien sind Dateien, die nur kurzfristig bei einer Programmausführung benötigt werden und am Ende eines Programms unwichtig sind. Auf Unix werden temporäre Dateien üblicherweise im Directory `/tmp` bzw. `/usr/tmp` angelegt.

Ein Beispiel für die Verwendung einer temporären Datei ist: Es sind Namen einzulesen, die sortiert auf eine bestimmte Datei ausgegeben werden sollen. Hier kann eine temporäre Datei für die Zwischenspeicherung angelegt werden, in die zunächst alle Namen in

der Eingabereihenfolge geschrieben werden. Der Inhalt dieser Datei wird dann sortiert und in eine »wichtige« Datei geschrieben. Danach ist die temporäre Datei »unwichtig« und kann entfernt werden. Namen von temporären Dateien sollten eindeutig sein, was bedeutet, daß an sie keine Namen vergeben werden sollten, die bereits existieren.

3.7.1 **tmpnam** – Einen eindeutigen Namen für eine temporäre Datei erzeugen

Um einen eindeutigen Namen für eine temporäre Dateien zu erhalten, steht die ANSI-C-Funktion **tmpnam** zur Verfügung.

```
#include <stdio.h>
```

```
char *tmpnam(char *zgr);
```

gibt zurück: Adresse eines eindeutigen temporären Dateinamens

Diese Funktion **tmpnam** erzeugt einen Dateinamen, der eindeutig ist, d.h. nicht einem Namen einer existierenden Datei entspricht. Jeder neue Aufruf dieser Funktion erzeugt einen neuen eindeutigen Namen. Diese Garantie eines neuen eindeutigen Dateinamens wird jedoch nur für TMP_MAX Aufrufe von **tmpnam** gegeben. Falls diese Funktion mehr als TMP_MAX-mal aufgerufen wird, ist das Verhalten je nach Implementierung verschieden.

TMP_MAX ist in <stdio.h> definiert. Während ANSI C als Wert für diese Konstante nur 25 vorschreibt, verlangt XPG3 als Wert für diese Konstante mindestens 10000.

Falls beim Aufruf von **tmpnam** für zgr ein NULL-Zeiger angegeben wird, wird der von dieser Funktion gefundene Dateiname in einem internen **static**-Speicherbereich untergebracht und dessen Adresse wird als Funktionswert zurückgegeben. Nachfolgende Aufrufe von **tmpnam** können dann den gleichen Speicherbereich wiederverwenden, weshalb in diesem Fall Umspeichern angebracht ist.

Falls für zgr kein NULL-Zeiger angegeben wird, dann sollte der angegebene Zeiger zgr einen Speicherplatz adressieren, der zumindest L_tmpnam Zeichen aufnehmen kann (L_tmpnam ist in <stdio.h> definiert). Die Funktion **tmpnam** schreibt dann ihr Resultat in diesen Speicherbereich und gibt die übergebene zgr-Adresse wieder als Funktionswert zurück.

Im Unterschied zur nachfolgenden Funktion **tmpfile** werden mit **tmpnam** keine Dateien kreiert, sondern lediglich Namen für Dateien gefunden, die explizit zu öffnen und auch wieder explizit zu löschen sind.

3.7.2 **tmpfile** – Eine temporäre Datei erzeugen und automatisch wieder löschen

Um sich eine »namenlose« temporäre Datei kreieren zu lassen, die am Programmende wieder automatisch gelöscht wird, steht die ANSI-C-Funktion **tmpfile** zur Verfügung.

```
#include <stdio.h>
```

```
FILE *tmpfile(void);
```

gibt zurück: FILE-Zeiger (bei Erfolg); NULL bei Fehler

Diese Funktion kreiert eine temporäre Binärdatei, die automatisch gelöscht wird, wenn sie geschlossen oder das Programm beendet wird. Diese temporäre Datei wird mit Modus »wb+« geöffnet. Wenn das Programm abnormal beendet wird, dann ist es nach ANSI C implementierungsdefiniert, ob die so erzeugten temporären Dateien gelöscht werden.

In Unix wird bei **tmpfile** meist die folgende Methode verwendet: Zuerst wird mit **tmpnam** ein eindeutiger Pfadname gefunden, dann wird die entsprechende Datei kreiert und sofort wieder mit **unlink** gelöscht. In Kapitel 5.5 bei der Vorstellung der Funktion **unlink** werden wir sehen, daß das Entfernen einer Datei mit **unlink** nicht zum Löschen deren Inhalts führt, sondern daß diese Datei erst beim Schließen wirklich gelöscht wird.

3.7.3 **tmpnam** – Das Erzeugen von temporären Dateinamen (mit Directory- und Präfixvorgabe)

Um einen eindeutigen Namen für eine temporäre Datei zu erhalten, bei dem man das Directory und das Namenspräfix selbst wählen kann, steht die Funktion **tmpnam** zur Verfügung.

```
#include <stdio.h>
```

```
char *tmpnam(const char *directory, const char *prefix);
```

gibt zurück: Adresse eines eindeutigen temporären Dateinamens

Die Funktion **tmpnam** bietet vier verschiedene Möglichkeiten für die Wahl eines Directory-Namens. Welche der folgenden vier Möglichkeiten zuerst zutrifft, tritt dann auch in Aktion:

1. Wenn die Environment-Variable `TMPDIR` definiert ist, dann wird deren Inhalt als Directory für den temporären Dateinamen verwendet, wenn dieses Directory existiert und für den betreffenden Benutzer Schreibrechte gewährt. Diese Möglichkeit wird im übrigen nicht von XPG3 unterstützt.

2. Wird für das Argument `directory` der Name eines existierenden und beschreibbaren Directorys angegeben, so wird dieses Directory für den temporären Dateinamen verwendet.
3. Der in der Konstante `P_tmpdir` (in `<stdio.h>` definiert) angegebene String wird als Directory für den temporären Dateinamen verwendet.
4. Sollte keine der drei zuvor angegebenen Bedingungen zutreffen, so wird ein lokales Directory für den temporären Dateinamen benutzt (meist `/tmp` oder `/usr/tmp`).

Wenn das Argument `präfix` kein NULL-Zeiger ist, so wird der hier angegebene String (bis zu 5 Zeichen) als Präfix dem temporären Dateinamen vorangestellt (siehe Beispiele).

Hinweis

`tempnam` ist zwar Bestandteil von XPG3, aber nicht von POSIX.1 oder ANSI C.

`tempnam` ruft zur Bereitstellung des für den Dateinamen benötigten Speicherplatzes die in Kapitel 9.4 beschriebene Funktion **`malloc`** auf. Diesen Speicherplatz kann der Benutzer später, wenn er die temporäre Datei nicht mehr benötigt, wieder explizit mit **`free`** freigeben.

Beispiel

Demonstrationsprogramm zu `tmpname` und `tmpfile`

```
#include "eighdr.h"

int
main(void)
{
    int    i;
    char    tmpdatei[L_tmpnam], zeile[MAX_ZEICHEN];
    FILE    *fz;

    printf(".....TMP_MAX=%ld\n", TMP_MAX);
    printf(".....L_tmpnam=%d\n", L_tmpnam);

    printf(".....Funktion tmpnam\n");
    for (i=1 ; i<=10 ; i++) {
        if (i%2==0)
            printf("%20d. %s\n", i, tmpnam(NULL));
        else {
            tmpnam(tmpdatei);
            printf("%20d. %s\n", i, tmpdatei);
        }
    }

    printf(".....Funktion tmpfile\n");
    if ( (fz=tmpfile()) == NULL)
        fehler_meld(FATAL_SYS, "Fehler bei tmpfile");
    fputs("Text in temporaere Datei schreiben und wieder lesen", fz);
    rewind(fz);
```

```

    if (fgets(zeile, sizeof(zeile), fz) == NULL)
        fehler_meld(FATAL_SYS, "Fehler bei fgets");
    printf("%s\n", zeile);

    exit(0);
}

```

Programm 3.16 (*tmpnam.c*): Demonstrationsbeispiel zu den Funktionen *tmpnam* und *tmpfile*

Nachdem man dieses Programm 3.16 (*tmpnam.c*) kompiliert und gelinkt hat

```
cc -o tmpnam tmpnam.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ tmpnam
.....TMP_MAX=238328
.....L_tmpnam=20
.....Funktion tmpnam
                1. /tmp/00147aaa
                2. /tmp/00147baa
                3. /tmp/00147caa
                4. /tmp/00147daa
                5. /tmp/00147eaa
                6. /tmp/00147faa
                7. /tmp/00147gaa
                8. /tmp/00147haa
                9. /tmp/00147iaa
               10. /tmp/00147jaa
.....Funktion tmpfile
Text in temporaere Datei schreiben und wieder lesen
$

```

Beispiel

Demonstrationsprogramm zu *tmpnam*

```

#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int    i;
    char   *tmpdir=NULL, *praefix=NULL;

    for (i=1 ; i<argc ; i+=2) {
        if (!strcmp(argv[i], "-t") && i+1 < argc)
            tmpdir = argv[i+1];
        else if (!strcmp(argv[i], "-p") && i+1 < argc)
            praefix = argv[i+1];
        else
            fehler_meld(FATAL, "usage:  %s [-t tmpdir] [-p praefix]", argv[0]);
    }
}

```

```

    printf("%s\n", tempnam(tmpdir, praefix));

    exit(0);
}

```

Programm 3.17 (**tempnam.c**): Demonstrationsbeispiel zur Funktion **tempnam**

Nachdem man Programm 3.17 (tempnam.c) kompiliert und gelinkt hat

```
cc -o tempnam tempnam.c fehler.c
```

ergeben sich z.B. folgende Abläufe:

```

$ tempnam -t $HOME -p xxx           [Home-Dir. und Präfix "xxx" für temporäre Datei]
/home/hh/xxx00692aaa
$ tempnam -p davor                  [Dir. aus P_tmpdir und Präfix "davor" für temporäre Datei]
/usr/tmp/davor00697aaa
$ TMPDIR=/home/hh tempnam -t /tmp    [Dir. aus TMPDIR (nicht aus Arg. von tmpdir) für temp. Datei]
/home/hh/00723aaa
$ tempnam -t /usr -p vvvv           [Voreingest. Dir. (/usr nicht beschreibbar) für temp. Datei]
/tmp/vvvv00730aaa
$

```

In den vorherigen Beispielen ist erkennbar, daß die Prozeß-ID in den temporären Dateinamen verwendet wird, um sicherzustellen, daß immer eindeutige temporäre Dateinamen vorliegen.

3.8 Löschen und Umbenennen von Dateien

In `<stdio.h>` müssen nach ANSI C auch die beiden Funktionen **remove** und **rename** definiert sein, die zum Löschen und Umbenennen von Dateien dienen.

3.8.1 remove – Löschen einer Datei

Zum Löschen einer Datei bietet ANSI C neben der in Kapitel 5.5 beschriebenen Funktion **unlink** auch die Funktion **remove** an.

```
#include <stdio.h>
```

```
int remove(const char *pfadname);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Der Aufruf dieser Funktion **remove** bewirkt, daß die Datei pfadname gelöscht wird. Falls zum Zeitpunkt des Aufrufs die entsprechende Datei geöffnet ist, ist das Verhalten von der jeweiligen Implementierung vorgegeben.

Hinweis

Für Dateien ist **remove** identisch zur Funktion **unlink** (siehe Kapitel 5.5). Für Directories dagegen ist **remove** identisch zur Funktion **rmdir** (siehe Kapitel 5.9).

3.8.2 rename – Umbenennen einer Datei

Zum Umbenennen einer Datei bietet ANSI C die Funktion **rename** an.

```
#include <stdio.h>
```

```
int rename(const char *altname, const char *neuname);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

ANSI-C-Definition für rename

Die Funktion **rename** ändert den Namen der Datei `altname` nach `neuname`. Falls die Datei `neuname` bereits existiert, ist das Verhalten implementierungsdefiniert. Der Rückgabewert 0 zeigt an, daß die Funktion erfolgreich ablief, ein von 0 verschiedener Rückgabewert deutet darauf hin, daß die Funktion fehlschlug. In diesem Fall wurde die Datei `altname` nicht nach `neuname` umgetauft.

ANSI C definiert diese Funktion nur für Dateien und läßt offen, ob sie auch auf Directories angewendet werden kann.

rename unter Unix

Da **rename** immer die beiden Dateien `neuname` und `altname` entfernt, müssen folgende Bedingungen für ein erfolgreiches Umbenennen mit **rename** vorliegen:

- ▶ Wenn `neuname` schon existiert, benötigt man für diese Datei die gleichen Rechte wie für das Löschen der Datei.
- ▶ Es müssen sowohl für das Directory, das `altname` enthält, als auch für das Directory, das `neuname` enthält, Schreibrechte vorliegen.

Wenn `altname` und `neuname` den gleichen Dateinamen enthalten, dann führt **rename** keinerlei Umbenennung durch und liefert den Rückgabewert 0 (erfolgreich).

POSIX.1 läßt das Umbenennen von Directories mit **rename** explizit zu. Deshalb sind unter Unix die folgenden beiden Möglichkeiten zu unterscheiden:

1. Wenn `altname` eine Datei (kein Directory) ist, dann muß dies, falls `neuname` bereits existiert, unbedingt eine Datei und darf kein Directory sein. Trifft dies zu, so wird die Datei `neuname` gelöscht und die Datei `altname` wird in `neuname` umbenannt, wenn entsprechende Rechte in den Directories vorliegen.

2. Wenn `altname` ein Directory ist, dann muß, falls `neuname` bereits existiert, dies unbedingt ein leeres Directory sein, das nur die Dateien `.` und `..` enthält. Trifft dies zu, so wird das Directory `neuname` gelöscht und das Directory `altname` wird in `neuname` umbenannt. Ein Umbenennen eines Directorys kann aber auch nur dann erfolgreich durchgeführt werden, wenn `neuname` nicht ein Subdirectory von `altname` ist. So kann man z.B. `/home/hh/work` nicht in `/home/hh/work/src` umbenennen, da der alte Name (`/home/hh/work`) nicht gelöscht werden kann.

3.9 Ausgabe von Systemfehlermeldungen

Wenn bei der Ausführung einer Systemfunktion ein Fehler auftritt, so liefern viele der Systemfunktionen -1 als Rückgabewert und setzen zusätzlich noch die global definierte Variable `errno` auf einen von 0 verschiedenen Wert. Diese Variable `errno` ist in `<errno.h>` mit

```
extern int errno;
```

definiert. Zusätzlich zu dieser Definition der Variablen `errno` definiert `<errno.h>` noch Konstanten für jeden Wert, der `errno` von den Systemfunktionen zugewiesen werden kann. Jede dieser Konstanten beginnt mit dem Buchstaben `E` (für *Error*). In den Unix-Manpages sind unter `intro(2)` alle in `<errno.h>` definierten Konstanten zusammengefaßt.

Bezüglich der Verwendung der Variablen `errno` ist folgendes zu beachten.

- ▶ ANSI C garantiert nur für den Programmstart, daß diese Variable `errno` auf 0 gesetzt wird. Die Systemfunktionen setzen diese Variable niemals zurück auf 0 und es gibt in `<errno.h>` keine Fehlerkonstante mit dem Wert 0.
- ▶ Deshalb ist es gängige Praxis, daß man `errno` vor dem Aufruf einer Systemfunktion explizit auf 0 setzt und nach dem Aufruf dieser Funktion den Wert von `errno` abprüft, um sicher zu sein, daß während der Ausführung dieser Funktion kein Fehler aufgetreten ist.

Um die Fehlermeldung zu erhalten, die zu einem in `errno` stehenden Fehlercode gehört, schreibt ANSI C die beiden Funktionen **`perror`** und **`strerror`** vor.

3.9.1 `perror` – Ausgabe der zu `errno` gehörenden Fehlermeldung

Die Funktion **`perror`** gibt auf `stderr` die zum momentan in `errno` stehenden Fehlercode gehörende Fehlermeldung aus.

```
#include <stdio.h>

void perror(const char *meldung);
```

perror gibt folgendes auf der Standardfehlerausgabe aus:

1. Wenn *meldung* kein NULL-Zeiger ist und nicht auf \0 zeigt, wird zuerst der String *meldung* gefolgt von »: « ausgegeben.
2. Dann wird die zum *errno*-Wert gehörige Fehlermeldung gefolgt von \n ausgegeben.

Die *errno*-Fehlermeldung entspricht genau dem Rückgabewert der nachfolgend beschriebenen Funktion **strerror**, falls diese mit dem gleichen *errno*-Wert als Argument aufgerufen wird. Somit liefern die beiden folgenden Anweisungen das gleiche Ergebnis:

```
perror("testausgabe")  
  
fprintf(stderr, "testausgabe: %s\n", strerror(errno));
```

3.9.2 strerror – Erfragen der zu einer Fehlernummer gehörenden Fehlermeldung

Die Funktion **strerror** (in <string.h> definiert) liefert die zu einer Fehlernummer (üblicherweise der *errno*-Wert) gehörende Fehlermeldung als Rückgabewert.

```
#include <string.h>
```

```
char *strerror(int fehler_nr);
```

gibt zurück: Zeiger auf die entsprechende Fehlermeldung

strerror ermittelt die zu *fehler_nr* gehörende Fehlermeldung, schreibt dann diese Fehlermeldung in einen eigenen Speicherbereich und liefert die Adresse dieses Fehlerstrings als Rückgabewert. Es ist zu beachten, daß der Speicherbereich, in dem sich die entsprechende Fehlermeldung befindet, bei nachfolgenden **strerror**-Aufrufe wiederverwendet und somit überschrieben wird. Wenn die Fehlermeldung aufzuheben ist, muß sie also zuvor umgespeichert werden.

Beispiel

Demonstrationsprogramm zu perror und strerror

In Kapitel 1.5 wurde bereits ein Demonstrationsprogramm zu den beiden Funktionen **strerror** und **perror** angegeben.

Das folgende Programm 3.18 (*feh1hand.c*) ist ein weiteres Demonstrationsbeispiel zu diesen beiden Funktionen **perror** und **strerror**, es zeigt aber auch eine typische Verwendung der Funktion **perror**:

```
#include <errno.h>  
#include "eighdr.h"  
  
int  
main(int argc, char *argv[])
```

```

{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}

```

Programm 3.18 (**fehlhand.c**): Demonstrationsbeispiel zu **perror** und **strerror**

Nachdem man Programm 3.18 (**fehlhand.c**) kompiliert und gelinkt hat

```
cc -o fehlhand fehlhand.c
```

ergibt sich z.B. folgender Ablauf:

```

$ fehlhand
EACCES: Permission denied
fehlhand: No such file or directory
$

```

In dem obigen Programm wird der Name des Programms (`argv[0]`) als Argument bei **perror** angegeben. Dies ist übliche Unix-Praxis, denn auf diese Art wird immer der Name des entsprechenden Programms gemeldet, in dem der Fehler auftrat, selbst wenn das Programm innerhalb einer Pipeline aufgerufen wird, wie z.B.

```
prog1 | prog2 | prog3
```

3.10 Übung

3.10.1 Buchstabenstatistik für Dateien

Erstellen Sie ein Programm `buchstat.c`, das die Häufigkeit des Vorkommens jedes einzelnen Buchstabens (aus dem englischen Alphabet) in den auf der Kommandozeile angegebenen Dateien ermittelt und ausgibt. Groß- und Kleinbuchstaben sollten dabei nicht unterschieden werden.

3.10.2 Ausgeben von bestimmten Zeilen einer Datei

Erstellen Sie ein Programm `zeilausg.c`, das aus einer Datei nur bestimmte Zeilen ausgibt. Welche Zeilen auszugeben sind, soll dabei auf der Kommandozeile angegeben werden, wie z.B.:

zeilausg 2-10 text

Die Zeilen 2 bis 10 von der Datei `text` ausgeben.

zeilausg 3,4-9,12,14- gebuehren

Die Zeilen 3, 4 bis 9, 12 und ab Zeile 14 alle Zeilen der Datei `gebuehren` ausgeben.

zeilausg -20,50- kunden

Von der Datei `kunden` die ersten 20 Zeilen und ab Zeile 50 alle Zeilen bis zum Dateiende ausgeben.

zeilausg maerchen

Die Datei `maerchen` vollständig ausgeben.

3.10.3 Einfache Realisierung des Kommandos `wc`

Erstellen Sie ein Programm `wz.c`, das wie das Kommando **wc** alle Zeichen, Wörter und Zeilen von den auf der Kommandozeile angegebenen Dateien zählt. Ist keine Datei angegeben, so soll es von der Standardeingabe (`stdin`) lesen.

Wie beim Kommando **wc** soll auch die Angabe der Optionen

```
l für Zeilen zählen
w für Wörter zählen
c für Zeichen zählen
```

möglich sein. Um die Implementierung hier zu vereinfachen, soll dieses Programm nur wirkliche Dateien verarbeiten können und nicht wie **wc** bei Angabe von Strich (-) als Dateiname von `stdin` lesen können.

3.10.4 Schachtelungsanalyse für C-Programme

Bei der Erstellung eines C-Programms kann es vorkommen, daß eine öffnende oder schließende Klammer vergessen oder ein Kommentar nicht abgeschlossen wird. Dies kann zu schwer auffindbaren Syntaxfehlern führen, da der C-Compiler eine völlig andere Klammerungsstruktur annimmt und damit den Überblick verliert.

Erstellen Sie ein Programm `cpruef.c`, das C-Programme analysiert, indem es am Anfang jeder Zeile die einzelnen Schachtelungstiefen angibt, die nach dieser Zeile vorliegen. Die Zeichen `{`, `}`, `(` oder `)` bewirken hierbei nur dann eine neue Schachtelung, wenn sie nicht in einem Kommentar angegeben sind.

Beispiele für den Ablauf dieses Programms sind:

```
$ cpruef tempnam.c
1: {0} (0) /*0*/ |#include "eighdr.h"
2: {0} (0) /*0*/ |
3: {0} (0) /*0*/ |int
4: {0} (0) /*0*/ |main(int argc, char *argv[])
5: {1} (0) /*0*/ |{
6: {1} (0) /*0*/ |    int    i;
7: {1} (0) /*0*/ |    char   *tmpdir=NULL, *praefix=NULL;
8: {1} (0) /*0*/ |
9: {2} (0) /*0*/ |    for (i=1 ; i<argc ; i+=2) {
10: {2} (0) /*0*/ |        if (!strcmp(argv[i], "-t") && i+1 < argc)
11: {2} (0) /*0*/ |            tmpdir = argv[i+1];
12: {2} (0) /*0*/ |        else if (!strcmp(argv[i], "-p") && i+1 < argc)
13: {2} (0) /*0*/ |            praefix = argv[i+1];
```

```

14: {2} (0) /*0*/ |     else
15: {2} (0) /*0*/ |     fehler_meld(FATAL, "usage: %s [-t tmpdir] [-p praefix]", argv[0]);
16: {1} (0) /*0*/ | }
17: {1} (0) /*0*/ |
18: {1} (0) /*0*/ |     printf("%s\n", tempnam(tmpdir, praefix));
19: {1} (0) /*0*/ |
20: {1} (0) /*0*/ |     exit(0);
21: {0} (0) /*0*/ | }

```

\$ cpruef datbytes.c

```

1: {0} (0) /*0*/ | #include <limits.h>
2: {0} (0) /*0*/ | #include "eighdr.h"
3: {0} (0) /*0*/ |
4: {0} (0) /*0*/ | int
5: {0} (0) /*0*/ | main(void)
6: {1} (0) /*0*/ | {
7: {1} (0) /*0*/ |     FILE *dz;
8: {1} (0) /*0*/ |     char dateiname[NAME_MAX];
9: {1} (0) /*0*/ |     long von, bis;
10: {1} (0) /*0*/ |     int zeich;
11: {1} (0) /*0*/ |
12: {1} (0) /*0*/ |     fprintf(stderr, "Dateiname? ");
13: {1} (0) /*0*/ |     gets(dateiname);
14: {1} (0) /*0*/ |
15: {1} (0) /*0*/ |     if ( (dz=fopen(dateiname, "r")) == NULL)
16: {1} (0) /*0*/ |         fehler_meld(FATAL_SYS, "kann %s nicht eroeffnen", dateiname);
17: {1} (0) /*0*/ |
18: {2} (0) /*0*/ |     do {
19: {2} (0) /*0*/ |         fprintf(stderr, "Hexausgabe ab Bytenr (Ende=0) ? ");
20: {2} (0) /*0*/ |         scanf("%ld", &von);
21: {2} (0) /*0*/ |
22: {3} (0) /*0*/ |         if (von != 0) {
23: {3} (0) /*0*/ |             fseek(dz, von, SEEK_SET);
24: {3} (0) /*0*/ |             fprintf(stderr, "          bis Bytenr ? ");
25: {3} (0) /*0*/ |             scanf("%ld", &bis);
26: {3} (0) /*0*/ |
27: {3} (0) /*0*/ |             printf("Hexdump der Datei %s (von Bytenr %ld bis %ld)\n",
28: {3} (0) /*0*/ |                 dateiname, von, bis);
29: {4} (0) /*0*/ |             while (von <= bis) {
30: {4} (0) /*0*/ |                 if ( (zeich=getc(dz)) != EOF)
31: {4} (0) /*0*/ |                     printf("%02x", zeich);
32: {5} (0) /*0*/ |                 else if (ferror(dz)) {
33: {5} (1) /*0*/ |                     fehler_meld(WARNUNG_SYS,
34: {5} (1) /*0*/ |                         "Fehler beim Lesen aus Datei %s (Bytenr: %ld", dateiname, von);
35: {5} (1) /*0*/ |                 } else if (feof(dz)) {
36: {5} (1) /*0*/ |                     printf("---EOF---\n");
37: {5} (1) /*0*/ |                     break;
38: {4} (1) /*0*/ |                 }
39: {4} (1) /*0*/ |                 von++;
40: {3} (1) /*0*/ |             }
41: {3} (1) /*0*/ |             printf("\n\n");

```

```
42: {3} (1) /*0*/ |          fflush(NULL);
43: {2} (1) /*0*/ |          }
44: {1} (1) /*0*/ |      } while (von != 0);
45: {1} (1) /*0*/ |
46: {1} (1) /*0*/ |      exit(0);
47: {0} (1) /*0*/ |}
```

```
-----
- Klammerung ( ) nicht ausgeglichen
$
```

Das letzte Beispiel zeigt, daß dieses Programm einige Schwächen hat, da es die Klammerung in einem String als »echte« Klammerung wertet. Diese konkreten Schwächen zu beseitigen, ist nicht allzu schwierig (Strings und **char**-Konstanten müßten eigens behandelt werden). Um den Umfang des Programms im Rahmen zu halten, wurde die Schachtelung jedoch auf Kommentare, Blöcke und runde Klammern beschränkt. Es steht dem Leser natürlich frei, dieses Programm entsprechend zu erweitern.

4 Elementare E/A-Funktionen

*Wer sie nicht konnte,
Die Elemente,
Ihre Kraft
Und Eigenschaft,
Wäre kein Meister
Über die Geister.*

Goethe

In Kapitel 4 werden wir zunächst die wichtigsten elementaren E/A-Operationen kennenlernen, die für das Arbeiten mit Dateien wichtig sind, wie z.B. das Öffnen, Beschreiben, Lesen und Schließen von Dateien. Diese einfachen elementaren E/A-Operationen bieten weder Pufferung noch andere Dienstleistungen, wie dies bei den im vorherigen Kapitel vorgestellten Standard-E/A-Funktionen der Fall ist. Anhand eines Beispiels wird gezeigt, wie wichtig die Größe des selbst gewählten Puffers beim Lesen oder Schreiben für das Zeitverhalten eines Programms ist. Die hier vorgestellten ungepufferten E/A-Routinen sind nicht Bestandteil von ANSI C, wohl aber von POSIX.1 und XPG4.

Zudem wird in diesem Kapitel auf die Datenstrukturen eingegangen, die der Kern für offene Dateien verwendet, bevor die gemeinsame Nutzung gleicher Dateien durch mehrere Prozesse (*file sharing*) erläutert wird. Die Schwierigkeiten, die bei *file sharing* auftreten können, führen uns dabei zu dem Konzept der atomaren Operationen (*atomic operation*). Atomare Operationen sind immer dann notwendig, wenn verschiedene Prozesse gleichzeitig dasselbe Betriebsmittel (wie Dateien oder Speicher) benutzen und sich so eine Ressource teilen (*resource sharing*).

4.1 Filedeskriptoren

Wird eine existierende Datei geöffnet oder eine neue Datei angelegt, so liefert die entsprechende Öffnungsroutine als Rückgabewert eine nichtnegative Zahl, den sogenannten *Filedeskriptor*. Um nun auf eine neu geöffnete Datei zuzugreifen, wie z.B. in sie zu schreiben oder aus ihr zu lesen, muß nicht der Dateiname, sondern dieser Filedeskriptor angegeben werden.

Bei Start eines Prozesses werden automatisch immer drei Filedeskriptoren eingerichtet, nämlich für die Standardeingabe, Standardausgabe und Standardfehlerausgabe. Diese drei Standard-Filedeskriptoren können sofort (ohne Öffnungsroutine) verwendet werden. Es ist Unix-Konvention, daß dabei die folgenden Nummern verwendet werden:

```
0 Standardeingabe      (standard input)
1 Standardausgabe      (standard output)
2 Standardfehlerausgabe (standard error)
```

Es zeugt aber von einem guten Programmierstil, nicht diese festen Nummern, sondern die in POSIX.1 festgelegten symbolischen Konstanten zu verwenden.

```
STDIN_FILENO
STDOUT_FILENO
STDERR_FILENO
```

Diese symbolischen Konstanten sind in der Headerdatei `<unistd.h>` definiert.

Die maximale Filedeskriptor-Nummer ist über die symbolische Konstante `OPEN_MAX` (in `<limits.h>`) festgelegt. `OPEN_MAX` legt somit fest, wie viele Dateien ein Prozeß maximal zu einem Zeitpunkt geöffnet haben darf. In älteren Unix-Versionen waren dies 20 (0-19). Auf den meisten heutigen Unix-Systemen ist diese Zahl auf mindestens 63 hochgesetzt. In SVR4 oder 4.4BSD-Unix ist diese Zahl nahezu unendlich, und nur durch Größen wie maximal darstellbare ganze Zahl oder maximal anlegbare Dateienzahl begrenzt.

4.2 Öffnen und Schließen von Dateien

Öffnet man eine Datei mit den elementaren E/A-Funktionen ***open*** oder ***creat***, so ordnet man dieser Datei einen *Filedeskriptor* zu, über den man nun in der Datei lesen oder schreiben kann.

4.2.1 open – Öffnen einer Datei

Um eine existierende Datei zu öffnen oder eine neue Datei anzulegen, steht die Funktion ***open*** zur Verfügung.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pfadname, int oflag, ... /*, mode_t modus */ );
```

gibt zurück: Filedeskriptor (bei Erfolg); -1 bei Fehler

pfadname

Name der zu öffnenden Datei

oflag

Für `oflag` kann eine der folgenden in `<fcntl.h>` definierten symbolischen Konstanten angegeben werden:

O_RDONLY

Datei nur zum Lesen öffnen (meist `O_RDONLY = 0`).

O_WRONLY

Datei nur zum Schreiben öffnen (meist `O_WRONLY = 1`).

O_RDWR

Datei zum Lesen und Schreiben öffnen (meist `O_RDWR = 2`).

Von diesen drei Konstanten muß eine und nur eine für `oflag` angegeben werden.

Neben diesen drei Konstanten existieren weitere für `oflag` erlaubte Konstanten, deren Angabe optional ist und die mit `|` (bitweises OR) verknüpft werden müssen.

O_APPEND

Datei zum »Schreiben am Ende« (Anhängen) öffnen.

O_CREAT

Datei neu anlegen, wenn sie nicht existiert. In diesem Fall muß auch das dritte Argument (`modus`) angegeben werden. `modus` legt die Zugriffsrechte (siehe Tabelle 4.1) für die neu anzulegende Datei fest. Falls eine Datei bereits existiert, hat diese Konstante keine Auswirkung.

O_EXCL

Falls `O_EXCL` zusammen mit `O_CREAT` angegeben ist, kann die Datei nicht geöffnet werden, wenn sie bereits existiert, und **`open`** liefert -1 (für Fehler).

O_TRUNC

Eine zum Schreiben geöffnete Datei wird vollständig geleert. Nachfolgende Schreiboperationen bewirken ein neues Beschreiben dieser Datei von Anfang an. Zugriffsrechte und Eigentümer der Datei bleiben hierbei erhalten.

O_NOCTTY

Falls `pfadname` der Name eines Terminals ist, so sollte dies nicht der Kontrollterminal des Prozesses werden.

O_NONBLOCK

Falls `pfadname` der Name einer FIFO oder einer Gerätedatei ist, wird diese beim Öffnen und bei nachfolgenden E/A-Operationen nicht blockiert (siehe Kapitel 12.1).

O_NDELAY

Veraltet, ähnlich zu `O_NONBLOCK`. Ist `O_NDELAY` gesetzt, liefert ein **`read`** von einer Pipe, FIFO oder Gerätedatei sofort den Rückgabewert 0, wenn dort keine Daten vorhanden sind, ansonsten würde es auf Daten warten. Da **`read`** auch beim Lesen des Dateiendes (EOF) den Rückgabewert 0 liefert, liegt hier eine Zweideutigkeit für den **`read`**-Aufrufer vor. Deswegen sollte man diese Konstante nicht mehr verwenden, sondern eben die Konstante `O_NONBLOCK`.

O_SYNC

Nach jedem Schreiben mit **write** darauf warten, bis der Schreibvorgang vollständig abgeschlossen ist. O_SYNC wird in SVR4 angeboten, auch wenn diese Konstante von POSIX.1 nicht vorgeschrieben ist.

modus

Dieses dritte Argument ist optional (durch Ellipsen-Prototyping mit drei Punkten ... in der Funktionsdeklaration angegeben) und wird auch nur bei der Angabe von O_CREAT für oflag ausgewertet.

Für modus sind eine oder mehrere mit | (bitweises OR) verknüpfte Konstanten aus Tabelle 4.1 anzugeben.

Konstante	Bedeutung
S_ISUID	set-user-ID Bit
S_ISGID	set-group-ID Bit
S_ISVTX	sticky Bit (saved-text Bit)
S_IRUSR	read (user; Leserecht für Eigentümer)
S_IWUSR	write (user; Schreibrecht für Eigentümer)
S_IXUSR	execute (user; Ausführrecht für Eigentümer)
S_IRWXU	read, write, execute (user; Lese-, Schreib- und Ausführrecht für Eigentümer)
S_IRGRP	read (group; Leserecht für Gruppe)
S_IWGRP	write (group; Schreibrecht für Gruppe)
S_IXGRP	execute (group; Ausführrecht für Gruppe)
S_IRWXG	read, write, execute (group; Lese-, Schreib- und Ausführrecht für Gruppe)
S_IROTH	read (others; Leserecht für alle anderen Benutzer)
S_IWOTH	write (others; Schreibrecht für alle anderen Benutzer)
S_IXOTH	execute (others; Ausführrecht für alle anderen Benutzer)
S_IRWXO	read, write, execute (others; Lese-, Schreib- und Ausführrecht für alle anderen Benutzer)

Tabelle 4.1: Mögliche Konstanten (aus <sys/stat.h>) für modus-Argument bei **open** und **creat**

In Kapitel 5.3 sind die einzelnen Zugriffsrechte ausführlich beschrieben.

Rückgabewert

Der von **open** zurückgegebene Filedeskriptor ist die kleinste momentan noch nicht vergabene Nummer. Dies machen sich einige Anwendungen zunutze, um anstelle der voreingestellten Standardeingabe (0), Standardausgabe (1) oder Standardfehlerausgabe (2) eine Datei zu verwenden. Dazu schließen sie zunächst (mit **close**) eine von diesen drei File-

deskriptoren und öffnen dann mit **open** eine neue Datei, welcher der gerade frei gewordene Filedeskriptor zugeteilt wird. Eine bessere Methode, dies zu tun, ist die Verwendung der Funktion **dup2** (siehe Kapitel 4.8).

Angabe zu langer Dateinamen bei open

Wenn die Konstante `_POSIX_NO_TRUNC` (POSIX.1) gesetzt ist, dann liefert **open** als Rückgabewert die Fehlerkonstante `ENAMETOOLONG`, wenn entweder der ganze Pfadname länger als `PATH_MAX` ist oder wenn eine Komponente des Pfadnamens länger als `NAME_MAX` ist. Ist `_POSIX_NO_TRUNC` nicht gesetzt, so werden zu lange Dateinamen einfach entsprechend gekürzt.

Bei zu langen Dateinamen liefert SVR4 im traditionellen System-V-Dateisystem (S5) keinen Fehler, in einem UFS-Dateisystem dagegen liefert SVR4 einen Fehler.

Hinweis

Der Datentyp `mode_t` ist in `<sys/types.h>` definiert und für Zugriffsrechte vorgesehen.

Bei jedem Öffnen einer Datei mit **open** sollte man den Rückgabewert überprüfen, um festzustellen, ob die Datei erfolgreich geöffnet werden konnte. Ein typischer Programmausschnitt für das Öffnen einer Datei ist z.B.:

```
int fd;
if ( (fd=open("adresse.txt", O_RDWR)) == -1)
    fehler_meld(FATAL_SYS, "kann adresse.txt nicht zum Lesen+Schreiben eroeffnen");
```

`O_TRUNC`

ist vorsichtig zu verwenden, denn dies ist die einzige Möglichkeit, den Inhalt einer bereits existierenden Datei mit **open** zu zerstören.

Die bei `O_CREAT` geforderten Zugriffsrechte werden nicht in jedem Fall gewährt, da eventuell die Dateikreierungsmaske die Vergabe von gewissen Rechten untersagt (siehe Funktion **umask** in Kapitel 5.3).

Beispiel

```
open("add", O_WRONLY|O_CREAT, S_IRWXU|S_IRGRP|S_IXGRP|S_IXOTH)
```

Neue Datei `add` mit den Zugriffsrechten `rwxr-x-x` anlegen und diese zum Schreiben öffnen.

```
open("kunden.txt", O_APPEND)
```

Datei `kunden.txt` zum Schreiben am Dateiende öffnen.

```
open("tempdat", O_WRONLY | O_TRUNC)
```

Datei `tempdat` zum Schreiben öffnen. Falls die Datei `tempdat` bereits existiert, wird ihr Inhalt gelöscht.

Der nachfolgende Programmausschnitt zeigt folgende Anwendung: Solange die Datei druckaktiv existiert, kann sie nicht geöffnet werden, und es wird nach 10 Sekunden eine erneute Eröffnung dieser Datei versucht. Wenn 10 Eröffnungsversuche fehlgeschlagen haben, wird das Programm abgebrochen.

```
.....
i=10;
while ( (fd=open("druckaktiv", O_RDWR | O_CREAT | O_EXCL, 660)) == -1 && i-- )
    sleep(10);
if (i==0)
    fehler_meld(FATAL, "Datei druckaktiv konnte in 10 Versuchen nicht geöffnet werden");
.....
```

4.2.2 creat – Anlegen einer neuen Datei

Um eine neue Datei anzulegen, steht neben **open** noch die Funktion **creat** zur Verfügung

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *pfadname, mode_t modus);
```

gibt zurück: Filedeskriptor (bei Erfolg); -1 bei Fehler

pfadname

ist Name der neu anzulegenden Datei.

modus

Für **modus** sind eine oder mehrere mit | (bitweises OR) verknüpften Konstanten aus Tabelle 4.1 anzugeben.

Hinweis

Der Aufruf

```
creat(pfad, modus)
```

ist identisch zu

```
open(pfad, O_RDWR | O_CREAT | O_TRUNC, modus)
```

In früheren Unix-Versionen war die Angabe von **O_CREAT** im zweiten Argument von **open** nicht möglich. Somit konnte dort mit **open** keine neue Datei angelegt werden, weswegen auch die Funktion **creat** notwendig war. Mit der Einführung der beiden Konstanten **O_CREAT** und **O_TRUNC** für das zweite Argument bei **open** ist aber die **creat**-Funktion eigentlich überflüssig geworden.

Ein Nachteil von **creat** ist, daß die neu angelegte Datei nur beschrieben werden kann. Um den Inhalt einer mit **creat** angelegten und nachfolgend beschriebenen Datei wieder zu lesen, muß diese Datei zunächst mit **close** geschlossen werden, bevor sie explizit mit **open** zum Lesen geöffnet wird. Eine bessere Vorgehensweise für eine solche Anwendung ist z.B. der Aufruf

```
open(pfad, O_RDWR | O_CREAT | O_TRUNC, modus)
```

Eine bereits existierende Datei `pfadname` verliert durch einen **creat**-Aufruf ihren alten Inhalt und kann von Beginn an neu beschrieben werden. Diese »neue« Datei behält aber die gleichen Zugriffsrechte wie die »alte« Datei; d.h., daß in diesem Fall der angegebene `modus` keine Wirkung hat.

Beispiel

Anlegen neuer Dateien mit entsprechenden Zugriffsrechten

Das nachfolgende Programm 4.1 (`neu.c`) liest einen Dateinamen mit zugehörigen Zugriffsmuster (als Oktalzahl) ein und kreiert dann – wenn möglich – eine Datei dieses Namens mit den angegebenen Zugriffsrechten. Dieses Programm `neu.c` kann durch die Eingabe von *Strg-D* (EOF) abgebrochen werden.

```
#include <stdio.h>
#include <limits.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(void)
{
    char    dateiname[_POSIX_PATH_MAX];
    int     fd;
    mode_t  rechte;

    umask(0); /* Voreingest. Dateikreierungsmaske fuer diesen Prozess loeschen*/

    while (scanf("%s %o", dateiname, &rechte) != EOF) {
        if ( (fd = creat(dateiname, rechte)) == -1)
            fehler_meld(WARNUNG_SYS, ".....kann %s nicht anlegen", dateiname);
        else {
            fprintf(stderr, "%s mit '%03o' angelegt\n", dateiname, rechte);
            close(fd);
        }
    }
    exit(0);
}
```

Programm 4.1 (**neu.c**): Anlegen neuer Dateien

Nachdem man das Programm 4.1 (neu.c) kompiliert und gelinkt hat

```
cc -o neu neu.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ neu
datei1 777
datei1 mit '777'angelegt
datei2 753
datei2 mit '753'angelegt
/usr/include/xyz.h 777
.....kann /usr/include/xyz.h nicht anlegen: Permission denied
Ctrl-D
$ ls -l datei1 datei2
-rwxrwxrwx  1 hh      bin          0 Jun  7 13:27 datei1
-rwxr-x-wx  1 hh      bin          0 Jun  7 13:27 datei2
$ neu
datei1 750
datei1 mit '750'angelegt  [Meldung falsch, da Datei ihre alten Rechte behielt]
Ctrl-D
$ ls -l datei1
-rwxrwxrwx  1 hh      bin          0 Jun  7 13:27 datei1
$
```

4.2.3 close – Schließen einer Datei

Um eine geöffnete Datei wieder zu schließen, steht die Funktion **close** zur Verfügung.

```
#include <unistd.h>
```

```
int close(int fd);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

close schließt die Datei mit dem Filedeskriptor *fd*.

Hinweis

Wenn ein Prozeß endet, werden alle von diesem Prozeß geöffneten Dateien automatisch geschlossen. Viele Anwendungen machen sich dies zunutze und schließen nicht explizit die Dateien, die sie mit **open** oder **creat** geöffnet haben.

Ein Prozeß kann maximal immer nur `OPEN_MAX` Dateien gleichzeitig offen haben. Falls diese Grenze erreicht ist, müssen Dateien mit **close** geschlossen werden, damit Filedeskriptoren wieder frei werden und das Öffnen neuer Dateien möglich wird.

4.3 Lesen und Schreiben in Dateien

Nachdem eine Datei zum Lesen und/oder Schreiben geöffnet wurde, kann man in ihr lesen und/oder schreiben.

4.3.1 `read` – Lesen von einer Datei

Um aus einer geöffneten Datei zu lesen, steht die Funktion **`read`** zur Verfügung.

```
#include <unistd.h>

ssize_t read(int fd, void *puffer, size_t bytezahl);
```

gibt zurück: Anzahl der gelesenen Bytes (bei Erfolg);
0 ("Lesezeiger" stand schon auf Dateiende) oder -1 (bei Fehler)

fd

Filedeskriptor der Datei, aus der zu lesen ist.

puffer

Speicheradresse, an der die aus der Datei *fd* gelesenen Daten zu schreiben sind.

bytezahl

Anzahl der Bytes, die aus Datei *fd* zu lesen sind.

Rückgabewert

Der Rückgabewert ist gleich der *bytezahl*, wenn das Lesen vollständig erfolgreich verlief.

Ist der Rückgabewert nicht gleich *bytezahl*, so kann dies unterschiedliche Ursachen haben:

- ▶ Das Dateiende (EOF) wurde erreicht, bevor die geforderte *bytezahl* von Bytes gelesen werden konnte. In diesem Fall hat **`read`** noch die restlichen vorhandenen Bytes gelesen und deren Anzahl als Rückgabewert geliefert. Erst der nächste **`read`**-Aufruf liefert dann 0, woran sich erkennen läßt, daß der »Lesezeiger« bereits am Dateiende stand.
- ▶ Wird von einer Terminalgerätedatei gelesen, so wird nur bis zum nächsten Zeilenende gelesen. In Kapitel 20 wird aufgezeigt, wie man dies ändern kann.
- ▶ Wenn von einem Netzwerk gelesen wird, dann kann die im Netz stattfindende Pufferung dazu führen, daß weniger als die geforderte *bytezahl* von Bytes gelesen wird.

In all diesen Fällen liefert **`read`** als Rückgabewert die wirklich gelesene Anzahl von Bytes.

Hinweis

- ▶ Während der primitive Systemdatentyp `size_t` nur nichtnegative Werte (drittes Argument bei **read**) aufnehmen kann, steht der mit POSIX.1 eingeführte Datentyp `ssize_t` (Datentyp des Rückgabewerts) für vorzeichenbehaftete Werte.
- ▶ Die häufigsten Werte für `bytezahl` sind 1 (Lesen eines Bytes) oder die vorgegebene Blockgröße (wie z.B. 512, 1024 usw.), wobei die Angabe der Blockgröße, wie in Kapitel 4.5 gezeigt wird, die wesentlich effizientere Vorgehensweise ist.
- ▶ Das Lesen beginnt **read** immer an der Position, auf die gerade der Schreib-/Lesezeiger der Datei zeigt. Nach dem Lesen wird der Schreib-/Lesezeiger um die Anzahl der gelesenen Bytes in der Datei weiterpositioniert.

Beispiel*Vergleichen von zwei Dateien*

Das Programm 4.2 (`vergl.c`) vergleicht die Inhalte von zwei auf der Kommandozeile angegebenen Dateien. Dazu liest es immer ein Byte (sicherlich nicht sehr effizient) aus jeder der beiden Dateien und vergleicht diese beiden Bytes.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int      fd1, fd2,
            gelesen1, gelesen2;
    char     puffer1[2], puffer2[2];
    long int i=1;

    /*---- Ueberpruefen der Argumentzahl-----*/
    if (argc != 3)
        fehler_meld(FATAL, "usage: %s datei1 datei2", argv[0]);

    /*---- Die beiden auf Kommandozeile angegeb. Dateien eroeffnen-----*/
    if ( (fd1 = open(argv[1], O_RDONLY)) == -1)
        fehler_meld(FATAL_SYS, "kann %s nicht zum Lesen eroeffnen", argv[1]);
    if ( (fd2 = open(argv[2], O_RDONLY)) == -1)
        fehler_meld(FATAL_SYS, "kann %s nicht zum Lesen eroeffnen", argv[2]);

    /*---- Bytes in den beiden Dateien nacheinander ueberpruefen -----*/
    while (1) {
        if ( (gelesen1 = read(fd1, puffer1, 1)) == -1)
            fehler_meld(FATAL_SYS, "Fehler beim Lesen aus %s (Bytenr %d)",
                        argv[1], i);
        if ( (gelesen2 = read(fd2, puffer2, 1)) == -1)
            fehler_meld(FATAL_SYS, "Fehler beim Lesen aus %s (Bytenr %d)",
                        argv[2], i);
```

```

if (gelesen1==0 && gelesen2==0) { /*-- Dateiende in beiden erreicht---*/
    fprintf(stderr, "%s und %s sind identisch\n", argv[1], argv[2]);
    exit(0);
} else if (gelesen1==0) {
    fprintf(stderr, "%s ist kleiner als %s (bis dorthin identisch)\n",
            argv[1], argv[2]);
    exit(1);
} else if (gelesen2==0) {
    fprintf(stderr, "%s ist groesser als %s (bis dorthin identisch)\n",
            argv[1], argv[2]);
    exit(1);
} else {
    if (puffer1[0] != puffer2[0]) {
        fprintf(stderr, "%ld. Bytenr: (%s:0x%02x) <> (%s:0x%02x)\n",
                i, argv[1], puffer1[0], argv[2], puffer2[0]);
        exit(1);
    } else
        i++;
    }
}
}

```

Programm 4.2 (**vergl.c**): Inhalt zweier Dateien vergleichen

4.3.2 write – Schreiben in eine Datei

Um in eine geöffnete Datei zu schreiben, steht die Funktion **write** zur Verfügung.

```
#include <unistd.h>
```

```
ssize_t write(int fd, void *puffer, size_t bytezahl);
```

gibt zurück: Anzahl der geschriebenen Bytes (bei Erfolg); -1 bei Fehler

fd

Filedeskriptor der Datei, in die zu schreiben ist

puffer

Speicheradresse der Daten, die in die Datei *fd* zu schreiben sind

bytezahl

Anzahl der Byte, die (von Speicheradresse *puffer*) in die Datei zu schreiben sind

Rückgabewert

Der Rückgabewert ist normalerweise gleich der `bytezahl`. Ist dies nicht der Fall, ist beim Schreiben ein Fehler aufgetreten, z.B. Speicherplatzmangel auf einem Datenträger (wie Festplatte oder Diskette).

Hinweis

Nach jedem erfolgreichen Schreiben mit **`write`** wird der Schreib-/Lesezeiger um die Anzahl der geschriebenen Bytes weiter positioniert.

Wurde `O_APPEND` beim Öffnen der Datei mit **`open`** angegeben, so wird bei jedem **`write`** ans Ende der Datei geschrieben.

Ein Rückgabewert verschieden von der geforderten `bytezahl` zeigt immer an, daß nicht alle geforderten Bytes geschrieben werden konnten, was auf einen Fehler schließen läßt. Ein typischer Programmausschnitt für das Schreiben in eine Datei ist z.B. der folgende:

```
if (write(fd, puffer, bytezahl) != bytezahl)
    fehler_meld(FATAL_SYS, "Fehler beim Schreiben mit write");
```

`write` schreibt seine Daten üblicherweise nicht sofort auf das entsprechende physikalische Medium (wie Festplatte), sondern in einen Cache (schneller Speicher) und kehrt dann vom Systemaufruf zurück. Zu einem geeigneten späteren Zeitpunkt werden dann die Daten aus dem Cache wirklich auf das physikalische Medium geschrieben. Wenn ein Prozeß auf die Daten zugreifen möchte, bevor sie physikalisch wirklich geschrieben wurden, so erhält er eben die Daten aus dem Cache. Dieses Zwischenspeichern der Daten in einem Cache-Puffer erhöht die Geschwindigkeit beim Schreiben mit **`write`** ganz erheblich, hat aber auch den Nachteil, daß bei einem Systemzusammenbruch die noch nicht physikalisch geschriebenen Daten aus dem Cache verloren sind. Wenn diese Unsicherheit ausgeschaltet werden soll, wie z.B. in Anwendungsfällen, in denen zuverlässige und sichere Daten gefordert sind, dann muß beim Öffnen der Datei mit **`open`** die Konstante `O_SYNC` angegeben werden. Dies bewirkt, daß jedes **`write`** (für diese Datei) erst alle Daten vollständig auf das physikalische Medium schreibt, bevor es zum Aufrufer zurückkehrt. Diese Sicherheit ist jedoch nicht umsonst, sondern wirkt sich erheblich auf die Schnelligkeit aus.

Beispiel

Einfache Umsetzung des Kommandos `cat`

Das folgende Programm 4.3 (`mcat.c`) ist eine einfache Umsetzung des Kommandos **`cat`**. Es gibt alle auf der Kommandozeile angegebenen Dateien nacheinander auf der Standardausgabe (`STDOUT_FILENO`) aus. Ist beim Aufruf überhaupt keine Datei angegeben, so liest es von der Standardeingabe (`STDIN_FILENO`) und gibt jede Zeile auf der Standardausgabe aus, wie **`cat`** dies auch tut.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```

#include "eighdr.h"

#define PUFF_GROESSE 512

static void ausgab(int fd);

int
main(int argc, char *argv[])
{
    int    i, fd;

    if (argc == 1) { /* wenn keine Datei auf Kommandozeile angegeben. */
        ausgab(STDIN_FILENO); /* dann von stdin lesen */
    } else {
        for (i=1 ; i<argc ; i++) {
            if ( (fd = open(argv[i], O_RDONLY)) == -1)
                fehler_meld(FATAL, "kann %s nicht zum Lesen oeffnen", argv[i]);
            ausgab(fd);
            close(fd);
        }
    }
    exit(0);
}

static void ausgab(int fd)
{
    int    n;
    char    puffer[PUFF_GROESSE];

    while ( (n = read(fd, puffer, PUFF_GROESSE)) > 0)
        if (write(STDOUT_FILENO, puffer, n) != n)
            fehler_meld(FATAL_SYS, "Fehler bei write");

    if (n == -1)
        fehler_meld(FATAL_SYS, "Fehler bei read");
}

```

Programm 4.3 (**mcac.c**): Einfache Realisierung des Kommandos *cat*

4.4 Positionieren in Dateien

Jede geöffnete Datei hat einen Schreib-/Lesezeiger, der auf die Position (*Offset*) zeigt, ab der nachfolgende Schreib-/Leseoperationen in der Datei stattfinden sollen. Nach dem Schreiben oder Lesen wird dieser Schreib-/Lesezeiger immer automatisch um die Anzahl der geschriebenen oder gelesenen Bytes weitergesetzt.

Normalerweise hat der Schreib-/Lesezeiger nach dem Öffnen einer Datei den Wert 0, was bedeutet, daß er auf den Dateianfang zeigt. Dies trifft nur dann nicht zu, wenn eine Datei mit `O_APPEND` geöffnet wird.

4.4.1 lseek – Positionieren des Schreib-/Lesezeigers in einer Datei

Um den Schreib-/Lesezeiger ohne Schreib-/Lesezugriff in einer Datei zu versetzen, steht die Funktion **lseek** zur Verfügung.

```
#include <sys/types.h>

#include <unistd.h>

off_t lseek(int fd, off_t offset, int wie);
```

gibt zurück: neue Position des Schreib-/Lesezeigers (bei Erfolg); -1 bei Fehler

fd

Filedeskriptor der Datei, in der Schreib-/Lesezeiger neu zu positionieren ist.

offset

legt die Byteanzahl fest, um die der Schreib-/Lesezeiger zu verschieben ist. Von welcher Position aus diese Verschiebung stattfindet, wird mit dem Argument *wie* festgelegt.

wie

Tabelle 4.2 zeigt die möglichen Angaben für das *wie*-Argument und ihre Bedeutung.

<i>wie</i> -Angabe	Wirkung
SEEK_SET	(meist 0) Schreib-/Lesezeiger vom Dateianfang an um <i>offset</i> Bytes versetzen; <i>offset</i> darf nur nichtnegativ sein.
SEEK_CUR	(meist 1) Schreib-/Lesezeiger von momentanen Position an um <i>offset</i> Bytes versetzen; <i>offset</i> darf positiv oder negativ sein.
SEEK_END	(meist 2) Schreib-/Lesezeiger vom Dateiende an um <i>offset</i> Bytes versetzen; <i>offset</i> darf positiv oder negativ sein.

Tabelle 4.2: Mögliche Angaben für das wie-Argument

Hinweis

Um die momentane Position des Schreib-/Lesezeigers in einer Datei zu ermitteln, muß man den Schreib-/Lesezeiger von der momentanen Position um 0 Byte weiterpositionieren, also nur stehen lassen, und man erhält über den Rückgabewert die aktuelle Position:

```
off_t aktuelle_position;
....
aktuelle_position = lseek(fd, 0, SEEK_CUR);
```

Der Anfangsbuchstabe *l* des Namens ***lseek*** steht für den Rückgabetyt **long int**. Vor der Einführung des primitiven Systemdatentyps `off_t` war der Rückgabetyt dieser Funktion und der Typ des Arguments `offset` nämlich **long int**.

Für reguläre Dateien ist die von ***lseek*** gelieferte Position des Schreib-/Lesezeigers immer nicht negativ. Da es aber auch Gerätedateien geben kann, bei denen der von ***lseek*** gelieferte Rückgabewert negativ ist, sollte man immer den Rückgabewert explizit auf -1 und nicht nur auf kleiner als 0 abfragen.

Wird ***lseek*** auf den Filedeskriptor einer Pipe oder einer FIFO angewendet, so liefert ***lseek*** als Rückgabewert -1 und setzt die globale Variable `errno` auf `EPIPE`. So kann mittels ***lseek*** eine Pipe oder FIFO durch einen Prozeß identifiziert werden.

Für das Argument `offset` kann ein Wert angegeben werden, der größer als die momentane Dateigröße ist. In diesem Fall schreibt ein nachfolgendes ***write*** an diese Position, und in der Datei entsteht ein nicht explizit beschriebenes Loch. Alle Bytes in diesem Loch haben den Wert 0.

Beispiel

```
lseek(fd, 0L, SEEK_SET)
```

Schreib-/Lesezeiger auf Dateianfang setzen.

```
lseek(fd, 25L, SEEK_CUR)
```

Schreib-/Lesezeiger von momentaner Position aus um 25 Bytes vorrücken.

```
lseek(fd, -1L, SEEK_END)
```

Schreib-/Lesezeiger auf das letzte relevante Byte (nicht auf EOF) setzen.

Mit ***lseek*** ist es möglich, eine Datei wie ein großes Array zu behandeln, allerdings mit einem langsameren Zugriff. Die nachfolgende Funktion ***get*** liest eine beliebige Zahl von Bytes ab einer bestimmten Position in einer Datei.

```
ssize_t  get(int fd, void *puffer, size_t bytezahl, off_t position)
{
    ssize_t gelesen;

    if (lseek(fd, position, SEEK_SET) == -1)
        fehler_meld(FATAL_SYS, "Fehler bei lseek");
    if ( (gelesen=read(fd, puffer, bytezahl)) == -1)
        fehler_meld(FATAL_SYS, "Fehler bei read");
    puffer[gelesen] = '\0';
    return(gelesen);
}
```

Beispiel

Test, ob Positionierung des Schreib-/Lesezeigers in `stdin` möglich ist

```
#include "eighdr.h"
```

```
int
```

```

main(int argc, char *argv[])
{
    fprintf(stderr, "Positionierung in stdin ");

    if (lseek(STDIN_FILENO, 0L, SEEK_CUR) == -1)
        fprintf(stderr, "nicht moeglich\n");
    else
        fprintf(stderr, "moeglich\n");

    exit(0);
}

```

Programm 4.4 (posi.c): Prüfung, ob eine Positionierung in der Standardeingabe möglich ist

Nachdem man das Programm 4.4 (posi.c) kompiliert und gelinkt hat

```
cc -o posi posi.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ posi
Positionierung in stdin nicht moeglich
$ posi </etc/passwd
Positionierung in stdin moeglich
$ cat /etc/passwd | posi
Positionierung in stdin nicht moeglich
$

```

Beispiel

Erzeugen einer Datei mit Löcher

Das folgende Programm 4.5 (lochgen.c) erzeugt Löcher in einer Datei, indem es immer den Schreib-/Lesezeiger 15 Bytes über das Dateende hinweg positioniert und dann mit **write** einen Kleinbuchstaben an diese neue Position schreibt, so daß in der Datei immer ein Loch von 15 Bytes entsteht. Die Bytes dieses Loches haben immer den ASCII-Wert 0.

```

#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(void)
{
    int      fd,
            zeich;

    if ( (fd = creat("datmitloch", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
        fehler_meld(WARNUNG_SYS, ".....kann datmitloch nicht anlegen");

    for (zeich='a' ; zeich<='m' ; zeich++) {
        if (lseek(fd, 15L, SEEK_CUR) == -1) /* Schreib/ Lesezgr 15 Bytes weiter */
            fehler_meld(WARNUNG_SYS, "Fehler bei lseek");
    }
}

```



```

        if (write(fd, &zeich, 1) != 1)
            fehler_meld(WARNUNG_SYS, "Fehler bei write");
    }

    exit(0);
}

```

Programm 4.5 (**lochgen.c**): Erzeugen einer Datei mit Löchern

Nachdem wir dieses Programm 4.5 (lochgen.c) kompiliert und gelinkt haben

```
cc -o lochgen lochgen.c fehler.c
```

lassen wir es ablaufen

```
$ lochgen
$
```

Wir erhalten die Datei **datmitloch**, deren Inhalt wir uns mit dem Programm **od** anschauen werden.

```

$ od -c datmitloch
0000000 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 a
0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 b
0000040 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 c
0000060 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 d
0000100 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 e
0000120 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 f
0000140 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 g
0000160 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 h
0000200 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 i
0000220 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 j
0000240 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 k
0000260 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 l
0000300 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 m
0000320
$

```

Hier ist zu erkennen, daß die Bytes in den Löchern der Datei automatisch mit ASCII-Wert 0 besetzt wurden.

4.5 Effizienz von E/A-Operationen

Das nachfolgende Programm 4.6 (incpout.c) zeigt deutlich, wie wichtig die Größe des gewählten E/A-Puffers bei **read**- und **write**-Funktionen für das Zeitverhalten eines Programmes ist. Dieses Programm kopiert dabei immer mit unterschiedlichen Puffergrößen die Standardeingabe auf die Standardausgabe, mißt jeweils mit der Funktion **times** (siehe Kapitel 10.8) die benötigten Zeiten und gibt sie in Form einer Tabelle aus.

```

#include <sys/times.h>
#include <sys/stat.h>

```



```
        zeit_ausgabe(puffer_groesse, uhr_ende-uhr_start,
                    &start_zeit, &ende_zeit, i);
        j++;
    }
    fprintf(stderr, "+-----+-----+-----+
                    "+-----+-----+\n");

    exit(0);
}

static void
zeit_ausgabe(long int puff_groesse,
             clock_t realzeit, struct tms *start_zeit, struct tms *ende_zeit,
             long int schleiflaeufer)
{
    static long    ticks=0;

    if (ticks == 0)
        if ( (ticks = sysconf(_SC_CLK_TCK)) < 0)
            fehler_meld(FATAL_SYS, "Fehler bei sysconf");

    fprintf(stderr, "| %10ld | %10.2lf | %10.2lf | %10.2lf | %10ld |\n",
            puff_groesse,
            (ende_zeit->tms_utime - start_zeit->tms_utime) / (double)ticks,
            (ende_zeit->tms_stime - start_zeit->tms_stime) / (double)ticks,
            realzeit / (double)ticks, schleiflaeufer);

    return;
}
```

Programm 4.6 (incpout.c): stdin auf stdout mit unterschiedlichen Puffern kopieren (mit Zeitmessung)

Nachdem man das Programm 4.6 (incpout.c) kompiliert und gelinkt hat

```
cc -o incpout incpout.c fehler.c
```

starten wir es, indem wir es die 2 MegaByte große Datei xx ständig nach /dev/null kopieren lassen:

```
$ ls -l xx
-rw-r--r--  1 hh      bin      2097152 Jun  8 14:27 xx
$ incpout <xx >/dev/null
```

Puffer- groesse	UserCPU (Sek)	SystemCPU (Sek)	Gebrauchte Uhrzeit	Schleifen- laeufer
1	16.42	346.13	368.66	2097152
2	8.22	170.92	181.14	1048576
4	4.25	85.16	89.54	524288
8	2.00	42.78	46.10	262144
16	0.95	21.45	22.44	131072
32	0.42	10.87	11.29	65536
64	0.22	5.51	5.87	32768
128	0.12	2.84	2.96	16384
256	0.09	1.47	1.56	8192
512	0.03	0.84	0.87	4096

	1024		0.02		0.50		0.52		2048	
	2048		0.01		0.47		0.48		1024	
	4096		0.00		0.44		0.44		512	
	8192		0.00		0.41		0.41		256	
	16384		0.00		0.41		0.41		128	
	32768		0.00		0.40		0.40		64	
	65536		0.01		0.40		0.41		32	
	131072		0.00		0.40		0.40		16	
	262144		0.00		0.40		0.40		8	
	524288		0.00		0.42		0.42		4	
	1048576		0.00		0.44		0.62		2	
+-----+-----+-----+-----+-----+										

\$

Für das hier verwendete Dateisystem zeigt also die Puffergröße 8192 das beste Zeitverhalten. Bei größeren Werten erzielt man keine nennenswerten Zeitgewinne mehr.

4.6 Kerntabellen für offene Dateien

Der Kern verwendet drei Tabellen (Datenstrukturen), um geöffnete Dateien zu verwalten.

4.6.1 Prozeßtabelleneintrag

Zu jedem Prozeß existiert ein Eintrag in der Prozeßtabelle. In einem solchen Prozeßtabelleneintrag befindet sich unter anderem eine Tabelle für alle offenen Filedeskriptoren. Zu jedem Filedeskriptor ist dabei folgende Information vorhanden:

Filedeskriptor-Flags (*fd flags*)
Zeiger auf einen Eintrag in der Dateitabelle (*file table*)

4.6.2 Dateitabelle (file table)

Der Kern unterhält eine Dateitabelle, in der zu jeder offenen Datei ein eigener Eintrag existiert. Ein solcher Eintrag enthält folgende Information:

file status flags für die Datei (read, write, append, nonblocking, ...)
aktuelle Position des Schreib-/Lesezeigers
Zeiger auf einen Eintrag in der sogenannten v-node-Tabelle

4.6.3 v-node-Tabelle (v-node table)

Die v-node-Tabelle enthält Einträge (*v-nodes*) zu jeder offenen Datei. Ein *v-node* für eine Datei enthält dabei neben typischen v-node-Informationen wie Dateityp auch meist noch die i-node-Informationen (Eigentümer, Größe, Zugriffsrechte usw.), die beim Öffnen der Datei aus der i-node-Tabelle (siehe Kapitel 5.5) in den v-node kopiert werden, so daß diese Daten immer sofort verfügbar sind. Zudem enthält ein v-node immer noch die aktuelle Dateigröße.

Die v-node-Tabelle wurde erst in den achtziger Jahren in Unix aufgenommen, um unterschiedliche Filesystem-Typen auf einem System unterstützen zu können. Der Name v-node wurde von dem sogenannten *Virtual File System* (VFS) abgeleitet. Das VFS ist die übergeordnete Schnittstelle im Kern zwischen den einzelnen Filesystemen und dem Rest des Kerns (siehe Kapitel 5.5).

Wir gehen hier nicht näher auf Implementierungsdetails dieser Tabellen ein, da diese für das Verständnis der grundlegenden Arbeitsweise nicht von Wichtigkeit sind. Abbildung 4.1 faßt die Zusammenhänge zwischen diesen drei Tabellen für einen Prozeß anschaulich zusammen. Dieser Prozeß hat zu diesem Zeitpunkt neben der Standardeingabe, Standardausgabe und Standardfehlerausgabe zwei weitere Dateien mit den Filedeskriptoren fd3 und fd4 offen.

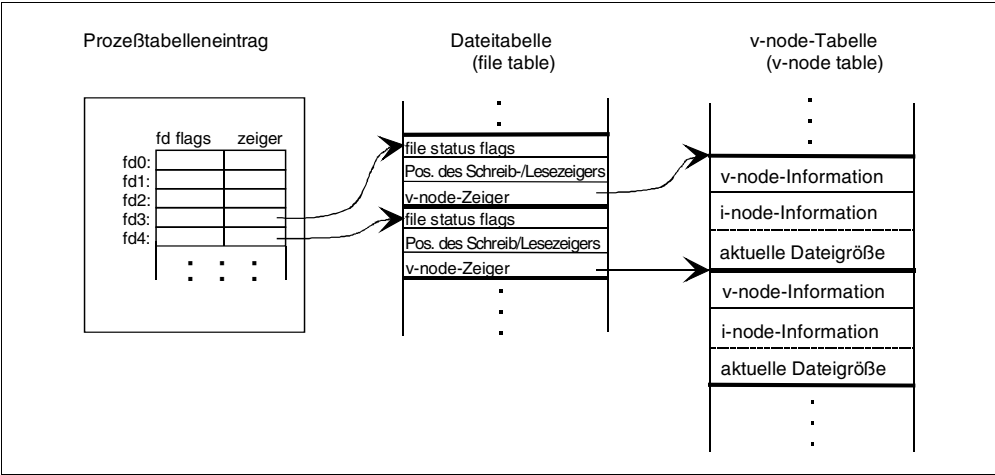


Abbildung 4.1: Kerntabellen für offene Dateien

4.7 File Sharing und atomare Operationen

4.7.1 File Sharing

Wenn zwei Prozesse die gleiche Datei öffnen, dann nennt man das *File Sharing*. Während in diesem Fall jeder Prozeß seinen eigenen Eintrag in der Dateitabelle erhält, existiert aber weiterhin nur ein v-node für die entsprechende Datei. Abbildung 4.2 veranschaulicht dies.

Ein Grund dafür, warum jeder Prozeß seinen eigenen Dateitabelleneintrag beim Öffnen einer Datei erhält, ist, daß jeder Prozeß seinen eigenen Schreib-/Lesezeiger hat, der auch jeweils an unterschiedlicher Position in der gleichen Datei stehen kann.

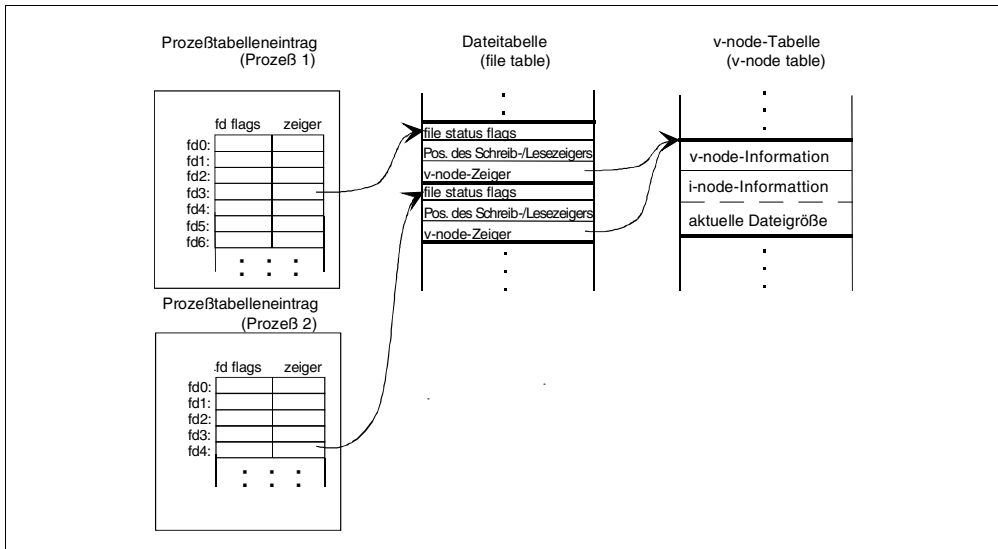


Abbildung 4.2: Zwei Prozesse haben zu einem Zeitpunkt die gleiche Datei geöffnet

Legt man die Konstellation der Tabelle aus Abbildung 4.2 zugrunde, können wir die Auswirkungen von bestimmten Dateioperationen wie folgt beschreiben:

- Nach jedem **write** wird die Position des Schreib-/Lesezeigers (im zugehörigen Dateitabelleneintrag des betreffenden Prozesses) um die Anzahl der geschriebenen Bytes erhöht. Falls dieses Schreiben dazu führt, daß die Datei vergrößert wird, so wird automatisch die neue Dateigröße im i-node eingetragen.
- Wird eine Datei mit `O_APPEND` geöffnet, so wird das entsprechende Bit bei den *file status flags* (im Dateitabelleneintrag) gesetzt. Jedesmal, wenn ein **write** auf eine Datei stattfindet, bei der dieses `O_APPEND`-Bit gesetzt ist, wird zuerst die Position des Schreib-/Lesezeigers im Dateitabelleneintrag auf die aktuelle Dateigröße (aus dem entsprechenden i-node) gesetzt. Dies führt dazu, daß jedes **write** auf diese Datei ein Schreiben ans Dateiende bewirkt.
- Bei einem **lseek**-Aufruf wird niemals eine E/A-Operation durchgeführt, sondern nur die Position des Schreib-/Lesezeigers (im Dateitabelleneintrag) modifiziert. Beim Positionieren ans Dateiende (mit **lseek**) wird die Position des Schreib-/Lesezeigers in der Dateitabelle auf die aktuelle Dateigröße (aus i-node) gesetzt.

Solange Prozesse aus gemeinsam geöffneten Dateien nur lesen, gibt es mit dem hier vorgestellten Konzept keinerlei Schwierigkeiten. Die treten erst dann auf, wenn mehrere Prozesse auf eine gemeinsam geöffnete Datei schreiben. Um die dabei möglicherweise auftretenden Probleme zu lösen, braucht man sogenannte atomare Operationen.

4.7.2 Atomare Operationen

Nehmen wir an, daß zwei Prozesse an das Ende der gleichen Datei schreiben, wie z.B. einer gemeinsamen Protokolldatei, in der jeder Prozeß seine durchgeführten Aktionen mitprotokolliert.

In älteren Unix-Versionen war `O_APPEND` für **open** nicht verfügbar. Um an das Ende einer Datei zu schreiben, mußten dort zwei Funktionen aufgerufen werden:

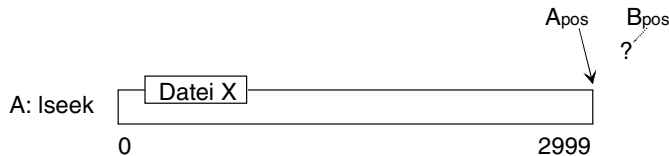
```
lseek(fd, 0L, SEEK_END)    /* Zuerst an Dateiende positionieren */
write(fd, puffer, bytezahl); /* und dann schreiben */
```

Während eine solche Vorgehensweise für einen einzelnen Prozeß sehr gut funktionierte, können jedoch Probleme entstehen, wenn mehrere Prozesse diese Methode verwenden, um an das Ende der gleichen Datei zu schreiben.

Nehmen wir z.B. an, daß zwei Prozesse A und B diese Vorgehensweise benutzen, um an das Ende der gleichen Datei X zu schreiben. Jeder Prozeß benutzt dabei – wie in Abbildung 4.2 gezeigt – den gleichen v-node-Eintrag. Da ein Prozeß aber immer nur eine gewisse Zeit die CPU zugeteilt bekommt, kann es passieren, daß er nach der Ausführung von **lseek** aus der CPU entfernt wird, und ein anderer Prozeß die CPU zugeteilt bekommt. Nachfolgend soll dies schrittweise veranschaulicht werden, wobei angenommen wird, daß die Datei X zu Anfang 3000 Bytes groß ist. Die Position des Schreib-/Lesezeigers (im entsprechenden Dateitabellen-Eintrag) wird mit A_{pos} und B_{pos} bezeichnet.

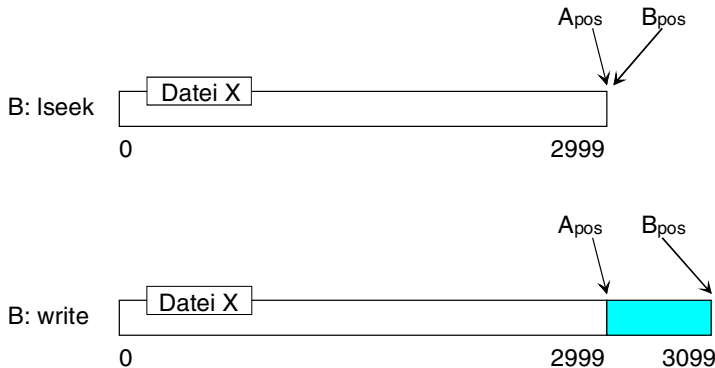
1. Schritt:

Prozeß A ist aktiv und kann gerade noch **lseek** (zum Positionieren ans Dateiende) ausführen, bevor ihm die CPU entzogen wird, so daß er nicht mehr zum Schreiben kommt.



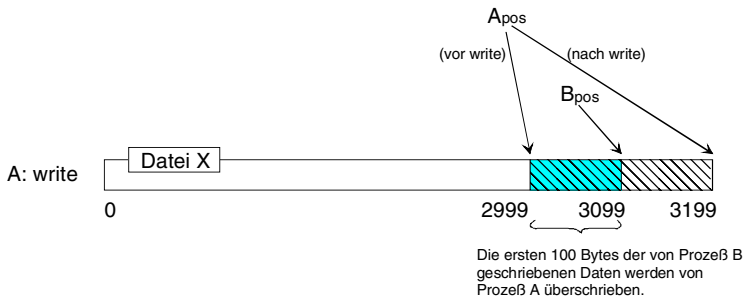
2. Schritt:

Nun ist Prozeß B aktiv und schreibt ans Dateiende z.B. 100 Bytes.



3. Schritt:

Nun wird wieder Prozeß A aktiv, dessen Schreib-/Lesezeiger immer noch – durch den 1. Schritt bedingt – auf das 3000.Byte zeigt. Das nun stattfindende **write** (mit z.B. 200 Bytes) von Prozeß A überschreibt also die zuvor geschriebenen Daten von Prozeß B ab dem 3000. Byte.



Das Problem besteht hier darin, daß die logische Operation »ans Dateiende positionieren und anschließendes Schreiben« zwei getrennte Funktionsaufrufe erfordert. Die Lösung zu diesem Problem ist, daß das Positionieren ans Dateiende und anschließendes Schreiben als eine atomare Operation ausgeführt wird. Neuere Unix-Versionen erreichen dies durch das Flag `O_APPEND` bei **open**. Wie weiter oben in Kapitel 4.2 beschrieben, bewirkt dies, daß vor jedem **write** der Kern den Schreib-/Lesezeiger auf das aktuelle Dateiende positioniert, so daß man nicht zwei Funktionen (auf Dateiende positionieren mit **lseek** und Schreiben mit **write**) benötigt.

Eine Operation, die nämlich zwei oder mehr Funktionsaufrufe erfordert, kann niemals eine atomare Operation sein.

Allgemein kann festgehalten werden, daß eine Operation, die sich aus mehreren Einzelaktionen zusammensetzt, dann atomar ist, wenn entweder alle einzelnen Aktionen in

einem Schritt erfolgreich ausgeführt werden oder überhaupt keine der Einzelaktionen. Es ist also gesichert, daß niemals nur ein Teil der Einzelaktionen in einem Schritt ausgeführt wird, sondern entweder alle oder gar keine.

4.8 Duplizieren von Filedeskriptoren

Es gibt Anwendungsfälle, in denen man existierende Filedeskriptoren duplizieren muß.

4.8.1 **dup** und **dup2** – Duplizieren von Filedeskriptoren

Um einen existierenden Filedeskriptor zu duplizieren, stehen die beiden Funktionen **dup** und **dup2** zur Verfügung.

```
#include <unistd.h>

int dup(int fd);
int dup2(int fd, int fd2);
```

beide geben zurück: Neuer Filedeskriptor (bei Erfolg); -1 bei Fehler

fd

der zu duplizierende Filedeskriptor

fd2 (bei dup2)

Wert des neuen duplizierten Filedeskriptors

Falls fd2 bereits geöffnet ist, wird die zugehörige Datei erst geschlossen. Falls fd2 gleich fd ist, dann gibt **dup2** fd2 ohne Schließen der entsprechenden Datei zurück.

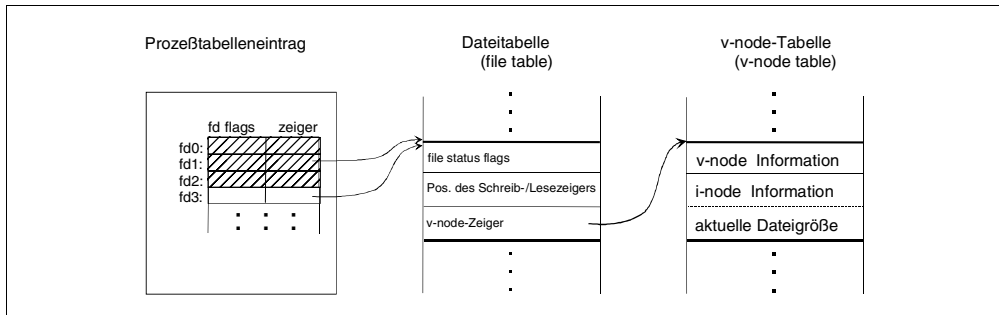
Rückgabewert

Der von **dup** zurückgegebene Filedeskriptor ist immer die kleinste noch freie nichtnegative Zahl, die noch nicht für andere Filedeskriptoren vergeben wurde. Der von den beiden Funktionen **dup** und **dup2** zurückgegebene neue Filedeskriptor zeigt auf den gleichen Dateitabellen-Eintrag wie der als Argument angegebene Filedeskriptor fd.

Ruft man z.B.

```
neufd = dup(1)
```

auf, so wird der Filedeskriptor 1 (fast immer die Standardausgabe) dupliziert. Nehmen wir z.B. an, daß neben den für die Standardeingabe, Standardausgabe und Standardfehlerausgabe reservierten Filedeskriptoren 0, 1 und 2 keine weiteren Dateien in diesem Prozeß offen sind, so wird dem neuen duplizierten Filedeskriptor `neufd` die Zahl 3 zugeordnet. Abbildung 4.3 verdeutlicht dies.

Abbildung 4.3: Kerntabellen nach `dup(1)`

Da nach diesem **dup**-Aufruf die beiden Filedeskriptoren 1 und 3 auf den gleichen Dateitabelleneintrag zeigen, benutzen sie auch beide die gleichen *file status flags* (read, write, append usw.) und die gleichen Positionen des Dateizeigers. Dagegen besitzt jeder dieser beiden Filedeskriptoren aber seine eigenen *fd flags* (im Prozeßeintrag).

Hinweis

Um einen Filedeskriptor zu duplizieren, kann auch die im nächsten Kapitel beschriebene Funktion **fcntl** verwendet werden. Der Aufruf `dup(fd)` ist identisch mit

```
fcntl(fd, F_DUPFD, 0);
```

und der Aufruf `dup2(fd, fd2)` ist nahezu identisch mit

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

Während es sich bei **dup2** um eine atomare Operation handelt, sind bei der letzteren Vorgehensweise zwei Funktionsaufrufe involviert.

Für den neu erzeugten Filedeskriptor löscht **dup** immer das *close-on-exec flag* in den *fd flags* des Prozeßtabelleneintrags. *close-on-exec* wird im nächsten Kapitel genauer beschrieben.

Beispiel

Duplizieren des `stdout`-Filedeskriptors mit `dup` und `dup2`

Das nachfolgende Programm 4.7 (`dupdup2.c`) ist ein Demonstrationsbeispiel zu den beiden Funktionen **dup** und **dup2**. Zunächst dupliziert es mit **dup** den Filedeskriptor für die Standardausgabe (`STDOUT_FILENO`) und schreibt dann über diesen duplizierten Filedeskriptor alle Kleinbuchstaben auf die Standardausgabe. Danach dupliziert es mit **dup2** den vorher duplizierten Filedeskriptor (für die Standardausgabe), legt diesmal aber die zu vergebende Nummer auf 10 fest und schreibt dann über diesen duplizierten Filedeskriptor (10) alle Großbuchstaben auf die Standardausgabe.

```
#include <sys/types.h>
#include "eighdr.h"
```

```
int
main(void)
{
    int    zeich, stdaus1, stdaus2=10;

    if ( (stdaus1=dup(STDOUT_FILENO)) == -1)
        fehler_meld(FATAL_SYS, "kann Filedeskriptor 1 nicht duplizieren");
    fprintf(stderr, "... Ausgabe ueber Filedeskriptor %d ....\n", stdaus1);
    for (zeich='a' ; zeich<='z' ; zeich++)
        write(stdaus1, &zeich, 1);
    printf("\n");

    if ( (stdaus2=dup2(stdaus1, stdaus2)) == -1)
        fehler_meld(FATAL_SYS, "kann Filedeskriptor %d nicht duplizieren",
                    stdaus1);
    fprintf(stderr, "... Ausgabe ueber Filedeskriptor %d ....\n", stdaus2);
    for (zeich='A' ; zeich<='Z' ; zeich++)
        write(stdaus2, &zeich, 1);
    printf("\n");

    exit(0);
}
```

*Programm 4.7 (**dupdup2.c**): Duplizieren des *stdout*-Filedeskriptors mit **dup** und **dup2***

Nachdem man dieses Programm 4.7 (dupdup2.c) kompiliert und gelinkt hat

```
cc -o dupdup2 dupdup2.c fehler.c
```

liefert es beim Aufruf folgende Ausgabe:

```
$ dupdup2
.... Ausgabe ueber Filedeskriptor 3 ....
abcdefghijklmnopqrstuvwxyz
.... Ausgabe ueber Filedeskriptor 10 ....
ABCDEFGHIJKLMNOPQRSTUVWXYZ
$
```

4.9 Ändern oder Abfragen der Eigenschaften einer offenen Datei

In gewissen Anwendungsfällen kann es notwendig sein, daß man nachträglich erfahren möchte, welche Einstellungen für eine schon offene Datei gelten, und eventuell möchte man diese Einstellungen auch ändern, ohne die Datei zu schließen.

4.9.1 **fcntl** – Ändern und Abfragen der Einstellungen einer offenen Datei

Um die Eigenschaften einer geöffneten Datei zu ändern oder abzufragen, steht die Funktion **fcntl** zur Verfügung

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int kdo, ... /* int arg */);
```

gibt zurück: abhängig von *kdo* (bei Erfolg); -1 bei Fehler

Die Funktion **fcntl** hat fünf Anwendungsfälle:

- ▶ Duplizieren eines schon existierenden Filedeskriptors (*kdo*=F_DUPFD)
- ▶ Setzen oder Abfragen der *fdflags* aus Prozeßtabelleneintrag (*kdo*=F_SETFD oder *kdo*=F_GETFD)
- ▶ Setzen oder Abfragen der *file status flags* aus Dateitabelleneintrag (*kdo*=F_SETFL oder *kdo*=F_GETFL)
- ▶ Setzen oder Abfragen der Eigentumsrechte bei asynchroner Ein-/Ausgabe (*kdo*=F_SETOWN oder *kdo*=F_GETOWN)
- ▶ Setzen oder Abfragen von sogenannten *record locks* (*kdo*=F_GETLK, *kdo*=F_SETLK oder *kdo*=F_SETLKW); dieser Anwendungsfall wird in Kapitel 12.2 beschrieben. Im übrigen ist hierbei das 3. Argument nicht vom Typ **int**, sondern ein Zeiger auf eine Struktur.

fd

Dieses Argument gibt den Filedeskriptor der Datei an, von der entsprechende Einstellungen zu erfragen oder zu setzen sind.

kdo

Hierfür kann eine ganze Reihe von symbolischen Konstanten angegeben werden. Nachfolgend sind die meisten dieser möglichen Konstanten beschrieben. Die restlichen sind in Kapitel 12.2 (beim sogenannten *record locking*) beschrieben.

F_DUPFD

Filedeskriptor *fd* duplizieren. In diesem Fall gibt **fcntl** den neuen Filedeskriptor zurück, der immer die kleinste noch nicht für offene Dateien benutzte (nichtnegative) Zahl ist. Für diese Zahl gilt zusätzlich, daß sie größer oder gleich dem 3. Argument *arg* ist, wenn dies angegeben ist. Der neue Filedeskriptor benutzt dabei zwar den gleichen Dateitabelleneintrag wie *fd*, besitzt aber seine eigenen *fdflags* (siehe auch Abbil-

dung 4.3), in denen das *close-on-exec*-Bit (FD_CLOEXEC) gelöscht ist. Ist dieses Bit für einen Filedeskriptor nicht gesetzt, so bleibt dieser Filedeskriptor bei einem *exec*-Aufruf (siehe Kapitel 10.5) bestehen.

F_GETFD

Als Rückgabewert liefert *fcntl* die *fdflags* von *fd*. Zur Zeit existiert allerdings nur ein *fdflag*, nämlich FD_CLOEXEC. Das bedeutet, daß in den aktuellen Unix-Versionen *fcntl* hier nur 0 oder 1 liefert.

F_SETFD

Die *fdflags* von *fd* mit *arg* setzen. Zur Zeit kann als 3. Argument (*arg*) nur FD_CLOEXEC oder !FD_CLOEXEC angegeben werden (siehe auch Hinweise).

F_GETFL

Als Rückgabewert liefert *fcntl* die *file status flags* von *fd*.

Folgende file status flags existieren:	
O_RDONLY	nur zum Lesen geöffnet
O_WRONLY	nur zum Schreiben geöffnet
O_RDWR	zum Lesen und Schreiben geöffnet
O_APPEND	zum Schreiben am Dateiende geöffnet
O_NONBLOCK	kein Blockieren bei FIFOS oder Gerätedateien
O_SYNC	nach jedem Schreiben auf Beendigung des physikalischen Schreibvorgangs warten
O_ASYNC	asynchrone E/A (nur in BSD)

(siehe auch Hinweise)

F_SETFL

Die *file status flags* von *fd* mit *arg* setzen. Die einzigen Flags, die geändert werden können, sind O_APPEND, O_NONBLOCK, O_SYNC und O_ASYNC (siehe auch Hinweise).

F_GETOWN

Als Rückgabewert liefert *fcntl* die PID (process ID) oder GID (group ID) des Prozesses, der gerade die Signale SIGIO und SIGURG empfängt (wird genauer in Kapitel 15.2 erläutert).

F_SETOWN

Das 3. Argument *arg* legt die PID oder GID des Prozesses fest, der die Signale SIGIO oder SIGURG empfängt. Ein positiver Wert für *arg* legt die PID und ein negativer Wert für *arg* die GID fest; im zweiten Fall ist die GID der Absolutwert vom angegebenen *arg*-Wert.

arg

Dieses dritte Argument wird nur ausgewertet, wenn ein Filedeskriptor zu duplizieren (F_DUPFD) ist oder die Einstellungen einer offenen Datei neu zu setzen sind (F_SETFD, F_SETFL, F_SETOWN).

Rückgabewert

Bei einem Fehler liefert **fcntl** immer den Wert -1. Bei Erfolg ist der Rückgabewert von **fcntl** vom Argument *kdo* abhängig. Tabelle 4.3 zeigt die möglichen Rückgabewerte in Abhängigkeit von der *kdo*-Angabe.

<i>kdo</i> -Angabe	Rückgabewert
F_DUPFD	neuer Filedeskriptor
F_GETFD	<i>fdflags</i> des Filedeskriptors <i>fd</i>
F_GETFL	<i>file status flags</i> des Filedeskriptors <i>fd</i>
F_GETOWN	PID (positiver Wert) oder GID (negativer Wert); wirkliche GID ist im zweiten Fall der Absolutwert
sonst	verschieden von -1

Tabelle 4.3: Rückgabewerte von **fcntl** bei den unterschiedlichen *kdo*-Angaben

Hinweis

- ▶ Bei F_GETFL muß Rückgabewert durch O_ACCMODE gefiltert werden.
Bei F_GETFL liefert **fcntl** die *file status flags* vom entsprechenden Filedeskriptor. Leider kann keiner der drei Öffnungsmodi O_RDONLY (meist 0), O_WRONLY (meist 1) oder O_RDWR (meist 2) direkt aus dem Rückgabewert herausgelesen werden. Angaben wie die folgende sind deshalb nicht möglich:

```
wert = fcntl(3, F_GETFL, 0);  
if (wert == O_RDONLY)
```

Um den Öffnungsmodus einer Datei zu überprüfen, muß man immer zuerst den Rückgabewert von **fcntl** über **&** (Bitweises AND) mit der Konstante O_ACCMODE verknüpfen, wie z.B.:

```
wert = fcntl(3, F_GETFL, 0);  
open_modus = wert & O_ACCMODE;  
if (open_modus == O_RDONLY)
```

- ▶ Mit O_SETFD und O_SETFL ist nur absolutes Setzen von Flags möglich.
Will man die *fdflags* bzw. die *file status flags* modifizieren, indem man einen bestimmten Status hinzufügen oder wegnehmen möchte, muß man zuerst die momentan gesetzten Flags mit **fcntl** unter Verwendung von O_GETFD bzw. O_GETFL erfragen. Das hierbei erhaltene Bitmuster kann man nun modifizieren, bevor man dieses für die entsprechende Datei mit **fcntl** unter Verwendung von O_SETFD bzw. O_SETFL neu setzt.

Um z.B. für eine Datei das Flag `O_APPEND` bei den *file status flags* hinzuzufügen, kann man nicht nur

```
fcntl(fd, F_SETFL, O_APPEND) /* löscht die zuvor gesetzten Flags */
```

aufrufen. Dies würde dazu führen, daß die momentan gesetzten Flags in *file status flags* zerstört würden.

Für Programme, bei denen eine Modifizierung der *fdflags* und *file status flags* notwendig ist, empfiehlt es sich, Funktionen zu definieren, die ähnlich denen im Programm 4.8 (`modfdfl.c`) sind.

```
#include <fcntl.h>
#include "eighdr.h"

/*----- Hinzufuegen von fdflags -----*/
void add_fdflags(int fd, int neuflags)
{
    int fdflags;
    if ( (fdflags=fcntl(fd, F_GETFD, 0)) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_GETFD");
    fdflags |= neuflags; /*----- Hinzufuegen der neuen Flags */
    if (fcntl(fd, F_SETFD, fdflags) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_SETFD");
}

/*----- Loeschen von fdflags -----*/
void loesch_fdflags(int fd, int wegflags)
{
    int fdflags;
    if ( (fdflags=fcntl(fd, F_GETFD, 0)) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_GETFD");
    fdflags &= ~wegflags; /*----- Entfernen der Flags 'wegflags' */
    if (fcntl(fd, F_SETFD, fdflags) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_SETFD");
}

/*----- Hinzufuegen von file status flags -----*/
void add_fstatus_flags(int fd, int neuflags)
{
    int fsflags;
    if ( (fsflags=fcntl(fd, F_GETFL, 0)) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_GETFL");
    fsflags |= neuflags; /*----- Hinzufuegen der neuen Flags */
    if (fcntl(fd, F_SETFL, fsflags) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_SETFL");
}

/*----- Loeschen von file status flags -----*/
void loesch_fstatus_flags(int fd, int wegflags)
{
    int fsflags;
    if ( (fsflags=fcntl(fd, F_GETFL, 0)) < 0 )
```

```

    fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_GETFL");
    fsflags &= ~wegflags; /*----- Entfernen der Flags 'wegflags' */
    if (fcntl(fd, F_SETFL, fsflags) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_SETFL");
}

```

Programm 4.8 (modfdl.c): Funktionen zum Modifizieren von fdflags und file status flags

Um z.B. O_SYNC für eine offene Datei zu setzen, könnte man nun

```
add_fdflags(fd, O_SYNC);
```

aufrufen. Ist O_SYNC für eine Datei gesetzt, so wird bei jedem Schreiben (mit **write**) gewartet, bis die Schreibaktion vollständig physikalisch abgeschlossen ist. Dies kann bei wichtigen Daten erforderlich sein, wo man erst dann mit dem Programm fortfahren möchte, wenn die entsprechenden Daten wirklich auf die Festplatte oder Diskette geschrieben sind. Dieses O_SYNC-Flag wirkt sich jedoch sehr negativ auf das Zeitverhalten eines Programms aus. Normalerweise werden Daten bei **write** immer erst in einem Puffer-Cache geschrieben, der erst nach der Rückkehr aus **write** weggeschrieben wird.

Beispiel

Ausgeben der file status flags für einen Filedeskriptor

Das folgende Programm 4.9 (fcntl.c) erwartet beim Aufruf eine Filedeskriptor-Nummer als erstes Argument auf der Kommandozeile und gibt dann die für diesen Filedeskriptor gesetzten *file status flags* aus.

```

#include <sys/types.h>
#include <fcntl.h>
#include <ctype.h>
#include <stdlib.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int i, open_modus, wert;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s fd", argv[0]);
    for (i=0 ; i<strlen(argv[1]) ; i++)
        if ( !isdigit(argv[1][i]) )
            fehler_meld(FATAL, "%s ist keine Dezimalzahl", argv[1]);

    if ( (wert=fcntl(atoi(argv[1]), F_GETFL, 0)) == -1)
        fehler_meld(FATAL_SYS, "Fehler bei fcntl");

    open_modus = wert & O_ACCMODE;

    if (open_modus == O_RDONLY)    printf("read only");
    else if (open_modus == O_WRONLY) printf("write only");
}

```



```

else if (open_modus == O_RDWR)    printf("read write");
else    fehler_meld(FATAL, "unbekannter open-modus fuer %s", argv[0]);

if ( wert & O_APPEND )    printf(", append");
if ( wert & O_NONBLOCK )    printf(", nonblocking");
#ifdef O_SYNC
if ( wert & O_SYNC )    printf(", O_SYNC gesetzt");
#endif
printf("\n");

exit(0);
}

```

Programm 4.9 (fcntl.c): Ausgeben der file status flags für einen Filedeskriptor

Nachdem man das Programm 4.9 (fcntl.c) kompiliert und gelinkt hat

```
cc -o fcntl fcntl.c fehler.c
```

kann man es aufrufen:

```

$ fcntl 0 </dev/null
read only
$ fcntl 1 >>/tmp/ttt    [in Bourne- und Korn-Shell]
$ cat /tmp/ttt
write only, append
$ fcntl 2 2>/tmp/ttt    [in Bourne- und Korn-Shell]
write only
$ fcntl 7 7>>/dev/null    [in Bourne- und Korn-Shell]
write only, append
$ fcntl 6 6<>/tmp/ttt    [nur in ksh; /tmp/ttt zum Lesen und Schreiben eroeffnen]
read write
$

```

4.10 Filedeskriptoren und der Datentyp FILE

In Kapitel 3.1 wurde der Datentyp FILE beschrieben, der von den Standard-E/A-Funktionen verwendet wird. Um zu einem FILE-Zeiger einer offenen Datei den zugehörigen Filedeskriptor bzw. umgekehrt zu einem Filedeskriptor einer offenen Datei einen entsprechenden FILE-Zeiger zu erhalten, bietet Unix zwei Funktionen an.

4.10.1 fileno – Erfragen des zu einem FILE-Zeiger gehörigen Filedeskriptors

Um den zu einem FILE-Zeiger einer offenen Datei gehörigen Filedeskriptor zu erhalten, steht die Funktion **fileno** zur Verfügung.

#include <stdio.h>

int **fileno**(FILE *fz);

gibt zurück: den zum FILE-Zeiger fz gehörigen Filedeskriptor

Die Funktion **fileno** wird z.B. immer dann benötigt, wenn eine Datei mit den Standard-E/ A-Funktionen **fopen** oder **freopen** geöffnet wurde und somit ein FILE-Zeiger für diese Datei vorhanden ist, man nun auf diese Datei aber eine Funktion (wie z.B. **dup** oder **fcntl**) anwenden möchte, die einen Filedeskriptor verlangt.

4.10.2 **fdopen – Erzeugen eines FILE-Zeigers zu einem Filedeskriptor**

Um zu einem existierenden Filedeskriptor einen FILE-Zeiger zu generieren, steht die Funktion **fdopen** zur Verfügung.

#include <stdio.h>

FILE ***fdopen**(int fd, const char *modus);

gibt zurück: FILE-Zeiger (bei Erfolg); NULL bei Fehler

Die Funktion **fdopen** erzeugt zu dem Filedeskriptor fd (durch eine der Funktionen **open**, **dup**, **dup2**, **fcntl** oder **pipe** erhalten) einen entsprechenden FILE-Zeiger.

modus

Mit dem *modus*-Argument wird die Zugriffsart für die Datei mit dem Filedeskriptor fd festgelegt (siehe Tabelle 4.4).

<i>modus-Argument</i>	<i>Bedeutung</i>
»r« oder »rb«	(read) Lesen
»w« oder »wb«	(write) Schreiben (Inhalt der Datei wird nicht wie bei fopen gelöscht)
»a« oder »ab«	(append) Schreiben am Dateiende
»r+«, »r+b« oder »rb+«	Lesen und Schreiben
»w+«, »w+b« oder »wb+«	Lesen und Schreiben (Inhalt der Datei wird nicht wie bei fopen gelöscht)
»a+«, »a+b« oder »ab+«	Lesen und Schreiben am Dateiende

Tabelle 4.4: Mögliche Angaben für *modus*-Argument bei **fdopen**

Der Buchstabe `b` bei der `modus`-Angabe wird benötigt, um zwischen Text- und Binärdateien zu unterscheiden. Da der Unixkern solche Dateiarnten nicht unterscheidet, hat unter Unix dieses Zeichen `b` keinerlei Bedeutung.

Hinweis

fdopen wird oft auf Filedeskriptoren angewendet, die von Funktionen zurückgegeben werden, die Pipes oder Kommunikationskanäle in Netzwerken einrichten. Diese speziellen Dateiarnten können nämlich nicht mit der Standard-E/A-Funktion ***fopen***, sondern nur mit speziellen Funktionen, die immer Filedeskriptoren liefern, geöffnet werden. Um nachträglich einen Stream (FILE-Zeiger) für eine solche spezielle Dateiarnt einzurichten, muß ***fdopen*** benutzt werden.

fdopen ist Bestandteil von POSIX.1, aber nicht von ANSI C.

Beispiel

Demonstrationsprogramm zu den Funktionen `fileno` und `fdopen`

```
#include <sys/types.h>
#include <fcntl.h>
#include "eighdr.h"

static void file_status( int fd );

int
main(void)
{
    FILE    *fz, *fz2;
    int     fd, fd2;

    /*----- Filedeskriptor zu stdin, stdout und stderr ermitteln -----*/
    printf("stdin  (%d)\n", fileno(stdin));
    printf("stdout (%d)\n", fileno(stdout));
    printf("stderr (%d)\n", fileno(stderr));

    /*--- abc.txt mit fopen oeffnen; Filedeskriptor zu FILE-Zeiger ermitteln ---*/
    if ( (fz=fopen("abc.txt", "r")) == NULL )
        fehler_meld(FATAL_SYS, "kann abc.txt nicht eroeffnen");
    fd = fileno(fz);
    printf("abc.txt (%d): ", fd);
    file_status(fd);

    /*--- Filedeskriptor von abc.txt duplizieren; FILE-Zeiger dazu mit fdopen
        ermitteln; danach Filedeskriptor zu diesen FILE-Zeiger ermitteln ---*/
    if ( (fd2=dup2(fd,10)) == -1)
        fehler_meld(FATAL_SYS, "kann Filedeskriptor %d nicht duplizieren", fd);
    if ( (fz2=fdopen(fd2, "w")) == NULL)
        fehler_meld(FATAL_SYS, "Fehler bei fdopen");
    fd2 = fileno(fz2);
    printf("abc.txt (%d): ", fd2);
    file_status(fd2);
}
```

```

    exit(0);
}

static void file_status( int fd )
{
    int    open_modus, wert;

    if ( (wert=fcntl(fd, F_GETFL, 0)) == -1)
        fehler_meld(FATAL_SYS, "Fehler bei fcntl");

    open_modus = wert & O_ACCMODE;

    if      (open_modus == O_RDONLY)    printf("read only");
    else if (open_modus == O_WRONLY)    printf("write only");
    else if (open_modus == O_RDWR)      printf("read write");
    else    fehler_meld(FATAL, "unbekannter open-modus fuer %d", fd);

    if ( wert & O_APPEND )    printf(", append");
    if ( wert & O_NONBLOCK )   printf(", nonblocking");
#ifdef O_SYNC
    if ( wert & O_SYNC )       printf(", O_SYNC gesetzt");
#endif
    printf("\n");
}

```

*Programm 4.10 (fdfz.c): Demonstrationsbeispiel zu den beiden Funktionen **fileno** und **fdopen***

Nachdem man dieses Programm 4.10 (fdfz.c) kompiliert und gelinkt hat

```
cc -o fdfz fdfz.c fehler.c
```

kann man es aufrufen:

```

$ touch abc.txt    [Datei abc.txt anlegen, wenn sie noch nicht existiert]
$ fdfz
stdin (0)
stdout (1)
stderr (2)
abc.txt (3): read only
abc.txt (10): read only
$

```

Beispiel

*Testen der Auswirkungen aller möglichen modus-Angaben bei **fdopen***

Das folgende Programm 4.11 (fdopen.c) testet alle Kombinationen bezüglich der möglichen Öffnungsmodi bei **fopen** und einem darauffolgenden **fdopen** auf die gleiche Datei (mit dupliziertem Filedeskriptor).

```

#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include "eighdr.h"

```

```
char *modus[6] = { "r", "w", "a", "r+", "w+", "a+" };
char string[MAX_ZEICHEN];

void file_status( int fd );

int
main(void)
{
    FILE      *fz, *fz2;
    int        fd, fd2;
    int        i, j;

    printf("| fopen |   file status flags || fdopen |   file status flags |\n"
           "+-----+-----+-----+-----+-----+\n");

    /*----- Alle Kombinationen von fopen/fdopen-Modi durchprobieren ----*/
    for (i=0 ; i<=5 ; i++) {
        for (j=0 ; j<=5 ; j++) {
            /*--- temp mit modus[i] eroeffnen -----*/
            if ( (fz=fopen("temp", modus[i])) == NULL )
                fehler_meld(FATAL_SYS, "kann temp nicht mit %s eroeffnen",
                           modus[i]);
            fd = fileno(fz); /*---- Filedeskriptor zu fz ermitteln */
            printf("| %5s |", modus[i]);
            strcpy(string, " ");
            file_status(fd);
            printf("%19s ||", string);

            /*--- fd duplizieren -----*/
            if ( (fd2=dup(fd)) == -1)
                fehler_meld(FATAL_SYS, "kann Filedeskr. %d nicht duplizieren", fd);
            /*--- Duplizierten Filedeskr. neu mit fdopen (modus[j] oeffnen --*/
            if ( (fz2=fdopen(fd2, modus[j])) == NULL)
                fehler_meld(FATAL_SYS, "Fehler bei fdopen");
            fd2 = fileno(fz2);
            printf(" %6s |", modus[j]);
            strcpy(string, " ");
            file_status(fd2);
            printf("%19s |\n", string);

            fclose(fz);
            fclose(fz2);
        }
    }

    exit(0);
}

/*----- file status flags ermitteln und als String nach string schreiben----*/
void file_status( int fd )
{
    int  open_modus, wert;
```

```
if ( (wert=fcntl(fd, F_GETFL, 0)) == -1)
    fehler_meld(FATAL_SYS, "Fehler bei fcntl");

open_modus = wert & O_ACCMODE;

if      (open_modus == O_RDONLY)   strcat(string, "read only");
else if (open_modus == O_WRONLY)   strcat(string, "write only");
else if (open_modus == O_RDWR)    strcat(string, "read write");
else    fehler_meld(FATAL, "unbekannter open-modus fuer %d", fd);

if ( wert & O_APPEND )   strcat(string, ", append");
if ( wert & O_NONBLOCK ) strcat(string, ", nonblocking");
#ifdef O_SYNC
if ( wert & O_SYNC )     strcat(string, ", O_SYNC gesetzt");
#endif
}
```

Programm 4.11 (**fdopen.c**): Ausgabe aller Auswirkungen der *modus*-Angabe bei **fdopen**

Nachdem man dieses Programm 4.11 (fdopen.c) kompiliert und gelinkt hat

```
cc -o fdopen fdopen.c fehler.c
```

kann man es aufrufen:

```
$ touch temp      [Datei temp anlegen, wenn sie noch nicht existiert]
$ fdopen
```

fopen	file status flags	fdopen	file status flags
r	read only	r	read only
r	read only	w	read only
r	read only	a	read only, append
r	read only	r+	read only
r	read only	w+	read only
r	read only	a+	read only, append
w	write only	r	write only
w	write only	w	write only
w	write only	a	write only, append
w	write only	r+	write only
w	write only	w+	write only
w	write only	a+	write only, append
a	write only, append	r	write only
a	write only, append	w	write only
a	write only, append	a	write only, append
a	write only, append	r+	write only
a	write only, append	w+	write only
a	write only, append	a+	write only, append
r+	read write	r	read write
r+	read write	w	read write
r+	read write	a	read write, append
r+	read write	r+	read write
r+	read write	w+	read write
r+	read write	a+	read write, append
w+	read write	r	read write

```
| w+ | read write || w | read write |
| w+ | read write || a | read write, append |
| w+ | read write || r+ | read write |
| w+ | read write || w+ | read write |
| w+ | read write || a+ | read write, append |
| a+ | read write, append || r | read write |
| a+ | read write, append || w | read write |
| a+ | read write, append || a | read write, append |
| a+ | read write, append || r+ | read write |
| a+ | read write, append || w+ | read write |
| a+ | read write, append || a+ | read write, append |
$
```

4.11 Das Directory /dev/fd

SVR4 und neuere BSD-Unix-Versionen bieten das Directory /dev/fd an. Die Dateien in diesem Directory haben Nummern (0, 1, 2, ...) als Namen.

Öffnet man eine Datei in diesem Directory mit

```
fd = open("/dev/fd/n", modus); /* Filedeskr. n muß geöffnet sein */
```

so ist das gleich bedeutend mit

```
fd = dup(n);
```

Nach beiden Aufrufformen besitzt jeder der beiden Filedeskriptoren `fd` und `n` zwar seinen eigenen Prozeßtabelleneintrag, jedoch benutzen beide den gleichen Dateitabelleneintrag (siehe Abbildung 4.3).

Die meisten Unix-Systeme ignorieren das Argument `modus` beim Öffnen einer Datei aus /dev/fd, so daß z.B. trotz eines erfolgreichen Aufrufs wie

```
fd = open("/dev/fd/1", O_RDWR); /* Lesen aus stdout!!; nicht mögl. */
```

ein Lesen aus `fd` (Kopie des `stdout`-Filedeskriptors) nicht möglich ist. Andere Systeme dagegen fordern, daß das angegebene `modus`-Argument eine Untermenge der `modus`-Angabe ist, die beim ursprünglichen Öffnen der Datei festgelegt wurde.

Die Dateien in /dev/fd sind hauptsächlich für die Shell gedacht, um bei Kommandos über die Angabe von Pfadnamen auf die Standardeingabe, Standardausgabe und Standardfehlerausgabe zuzugreifen. Bisher mußte z.B. bei **sort**, wenn dieses Kommando nach dem Lesen aus Dateien von der Standardeingabe lesen sollte, immer der Bindestrich (-) angegeben werden, wie z.B.:

```
kdo datei2 | cat datei1 - datei3 | sort
```

In diesem Beispiel liest **cat** zuerst die `datei1`, dann liest es von der Standardeingabe (hier aus der Pipe) und zuletzt dann die `datei3`. Mit der Einführung von /dev/fd ist der Bindestrich als Argument für Kommandos überflüssig geworden, und man kann die obige Kommandozeile wie folgt angeben:

```
kdo datei2 | cat datei1 /dev/fd/0 datei3 | sort
```

Hinweis

Das Directory `/dev/fd` ist nicht Bestandteil von POSIX.1.

Einige Systeme bieten die Directories `/dev/stdin`, `/dev/stdout` und `/dev/stderr` an. Diese Directories sind identisch mit den Directories `/dev/fd/0`, `/dev/fd/1` und `/dev/fd/2`.

Der Pfadname `/dev/fd/n` darf auch bei der Funktion **creat** oder bei Verwendung von `O_CREAT` bei der Funktion **open** angegeben werden. In beiden Fällen wird keine neue Datei `/dev/fd/n` angelegt, sondern nur der Filedeskriptor `n` dupliziert.

4.12 Übung

4.12.1 Anhängen einer Datei an eine andere

Erstellen Sie ein Programm `anhaeng.c`, das zwei Dateinamen auf der Kommandozeile erwartet und dann unter Verwendung der elementaren E/A-Funktionen den Inhalt der zuerst angegebenen Datei an die zweite Datei anhängt.

4.12.2 Rückwärtiges Ausgeben einer Datei

Erstellen Sie ein Programm `reverse.c`, das unter Verwendung der elementaren E/A-Funktionen eine Datei, deren Name auf der Kommandozeile anzugeben ist, Zeile für Zeile rückwärts ausgibt.

4.12.3 Duplizieren und mehrmaliges Öffnen derselben Datei

Hier nehmen wir an, daß ein Prozeß die folgenden Aufrufe durchführt:

```
fd1 = open("datei1", oflag);
fd2 = dup(fd1);
fd3 = open("datei1", oflag);
fd4 = dup(fd3);
```

Zeichnen Sie (ähnlich zur Abbildung 4.3) die aus diesen Aufrufen resultierende Konstellation.

Wie würde sich ein **fcntl** mit `F_SETFD` und wie ein **fcntl** mit `F_SETFL` auf die einzelnen Filedeskriptoren auswirken?

4.12.4 Nachvollziehen einer Notation aus der Bourne- und Korn-Shell

Im Band »Linux-Unix-Shells« wurde die folgende Konstruktion der Bourne- und Korn-Shell beschrieben.


```
kdo n1>&n2
```

Diese Angabe bedeutet, daß der Filedeskriptor *n1* in die Datei umgelenkt wird, auf die der Filedeskriptor *n2* zeigt.

Dort wurde auch auf den Unterschied zwischen den beiden folgenden Angaben eingegangen:

```
kdo >aus 2>&l
```

```
kdo 2>&l >aus
```

Erklären Sie den Unterschied zwischen diesen beiden Angaben. Hierbei ist es wichtig zu wissen, daß die Shell eine Kommandozeile von links nach rechts auswertet.

5 Dateien, Directories und ihre Attribute

*Wir lernen die Menschen nicht kennen,
wenn sie zu uns kommen; wir müssen
zu ihnen gehen, um zu erfahren, wie
es mit ihnen steht.*

Goethe

In diesem Kapitel werden Attribute vorgestellt, die zu jeder Datei und jedem Directory im sogenannten i-node gespeichert sind. Für jedes einzelne Attribut bietet die Struktur `stat`, die als erstes vorgestellt wird, eine eigene Komponente an.

Die einzelnen Attribute dieser Struktur werden hier ebenso detailliert besprochen wie die Funktionen, mit denen man diese Attribute erfragen oder modifizieren kann.

Neben den Attributen von Dateien und Directories wird auf die Struktur des Unix-Dateisystems und auf symbolische Links eingegangen. Zudem stellt dieses Kapitel Funktionen vor, mit denen man Directories anlegen, deren Inhalt lesen oder in andere Directories wechseln kann.

5.1 Dateiattribute

5.1.1 Struktur `stat`

Die Struktur `stat` enthält für jedes einzelne Dateiattribut eine eigene Komponente. Die Komponenten dieser Struktur sind nicht alle fest vorgeschrieben und können sich in den einzelnen Unix-Derivaten unterscheiden. Eine Definition der Struktur `stat` kann z.B. wie folgt aussehen:

```
struct stat {  
    mode_t    st_mode;    /* Dateart und Zugriffsrechte      */  
    ino_t     st_ino;     /* i-node Nummer                */  
    dev_t     st_dev;     /* Gerätenummer (Dateisystem)    */  
    dev_t     st_rdev;    /* Gerätenummer für Gerätedateien*/  
                                /* (nur für special files)      */  
    nlink_t   st_nlink;   /* Anzahl der Links              */  
    uid_t     st_uid;     /* User-ID des Eigentümers       */  
    gid_t     st_gid;     /* Group-ID des Eigentümers      */  
    off_t     st_size;    /* Größe in Byte für normale Dateien */  
                                /* (nur für regular files)      */  
    time_t    st_atime;   /* Zeit d. letzt. Zugriffs (access time)*/  
};
```

```
time_t  st_mtime;    /* Zeit d. letzt. Änderung in der Datei */
                /* (modification time) */
time_t  st_ctime;    /* Zeit der letzten Änderung des i-node */
long    st_blksize;  /* voreingestellte Blockgröße */
long    st_blocks;   /* Anzahl der benötigten 512-Byte-Blöcke */
};
```

Bis auf die drei Komponenten `st_rdev`, `st_blksize` und `st_blocks` sind alle aufgezählten Komponenten von POSIX.1 vorgeschrieben.

Bis auf die letzten beiden sind alle Komponenten dieser Struktur als primitive Systemdatentypen definiert.

In den folgenden Kapiteln werden alle Komponenten dieser Struktur im einzelnen genauer besprochen.

5.1.2 stat, fstat und lstat – Erfragen von Dateiattributen

Um die Attribute von Dateien zu erfragen, stehen die Funktionen **stat**, **fstat** und **lstat** zur Verfügung.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *pfadname, struct stat *puffer);

int fstat(int fd, struct stat *puffer);

int lstat(const char *pfadname, struct stat *puffer);
```

alle drei geben zurück: 0 (bei Erfolg); -1 bei Fehler

Allen drei Funktionen ist die Adresse einer Variablen vom Datentyp `struct stat` zu übergeben. Die Funktionen schreiben dann die entsprechenden Informationen (Attribute) der betreffenden Datei in die einzelnen Komponenten dieser Strukturvariablen.

stat

schreibt die Attribute der Datei mit dem Pfadnamen `pfadname` in die Strukturvariable `*puffer`.

fstat

schreibt die Attribute der schon geöffneten Datei mit dem Filedeskriptor `fd` in die Strukturvariable `*puffer`.

lstat

schreibt wie **stat** die Attribute der Datei mit dem Namen `pfadname` in die Strukturvariable `*puffer`. Im Unterschied zu **stat** schreibt ***lstat*** für den Fall, daß es sich bei `pfadname` um einen symbolischen Link handelt, die Attribute des symbolischen Links selbst und nicht der Datei, auf die dieser symbolische Link verweist, nach `*puffer`.

5.2 Dateiarnten

SVR4 kennt verschiedene Arten von Dateien:

1. *Regular File (Reguläre Datei, Einfache Datei, Gewöhnliche Datei)*

Eine solche Datei ist eine Sammlung von Zeichen, die unter den entsprechenden Dateinamen gespeichert sind. Dateien dieser Art können sowohl Text als auch maschinenlesbaren Binärcode (Programme, Projektdateien) oder von speziellen Programmen vorgegebene Dateiformate (wie z.B. **ar**, **cpio**, **tar**) enthalten. Unix kennt keinerlei spezielles Dateiformat, sondern überläßt die Interpretation der Dateiinhalte den jeweiligen Programmen (wie z.B. dem Archivierungsprogramm **ar** oder dem Linker **ld**).

2. *Directory (Dateiverzeichnis, Dateikatalog)*

Eine Directory-Datei enthält die Namen von anderen Dateien mit zugehöriger i-node-Nummer. Im i-node sind weitere Information zur jeweiligen Datei angegeben. Jeder Prozeß, der Leserechte für eine Directory-Datei besitzt, kann deren Inhalt lesen. Ein direktes Schreiben in eine Directory-Datei ist aber grundsätzlich nur dem Kern erlaubt.

3. *Special file (Gerätedatei)*

Gerätedateien repräsentieren die logische Beschreibung von physikalischen Geräten wie z.B. Bildschirmen, Druckern oder Disks. Das Besondere am Unix-System ist, daß es von solchen Gerätedateien in der gleichen Weise liest oder auf sie schreibt, wie es dies bei gewöhnlichen Dateien tut. Jedoch wird hierbei nicht der normale Dateizugriff aktiviert, sondern der entsprechende Gerätetreiber (*device driver*). Es werden zwei Klassen von Geräten unterschieden:

► **character special file** (*zeichenorientierte Geräte*)

Datentransfer erfolgt zeichenweise, wie z.B. Terminal.

► **block special file** (*blockorientierte Geräte*)

Datentransfer erfolgt nicht bytewise, sondern in Blöcken, wie z.B. bei Festplatten.

4. *FIFO (first in first out, Named Pipes)*

FIFOS – auch *Named Pipes* genannt – dienen zur Kommunikation und Synchronisation verschiedener Prozesse. Prinzipiell können sie wie einfache Dateien benutzt werden, mit dem wesentlichen Unterschied, daß Daten nur einmal gelesen werden können. Zudem können die Daten aus ihnen nur in derselben Reihenfolge gelesen werden, wie sie geschrieben wurden. FIFOS werden in Kapitel 17.3 beschrieben.

- 5. *Sockets*
Sockets dienen zur Kommunikation von Prozessen in einem Netzwerk, können aber auch zur Kommunikation von Prozessen auf einem lokalen Rechner benutzt werden. Sockets werden in Kapitel 19.2 zur Interprozeßkommunikation benutzt.
- 6. *Symbolic Links (Symbolische Links)*
Symbolische Links sind Dateien, die lediglich auf andere Dateien zeigen. In Kapitel 5.6 werden die symbolischen Links beschrieben.

Die Komponente `st_mode` der Struktur `stat` informiert über die entsprechende Dateiarart. Dazu muß der Aufrufer die in `<sys/stat.h>` definierten und in Tabelle 5.1 angegebenen Makros mit dem in `st_mode` gespeicherten Wert aufrufen.

Makro	liefert TRUE, wenn es sich bei Datei um ... handelt
S_ISREG()	reguläre Datei
S_ISDIR()	Directory
S_ISCHR()	zeichenorientierte Gerätedatei
S_ISBLK()	blockorientierte Gerätedatei
S_ISFIFO()	Pipe oder FIFO
S_ISLNK()	symbolischen Link (nicht in POSIX.1 oder SVR4)
S_ISSOCK()	Socket (nicht in POSIX.1 oder SVR4)

Tabelle 5.1: Makros in `<sys/stat.h>` zur Bestimmung der Dateiarart über `st_mode`

Beispiel

Ausgeben der Dateiarart von Dateien

```
#include <sys/types.h>
#include <sys/stat.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int i;
    struct stat attribut;

    for (i=1 ; i<argc ; i++) {
        printf("%40s: ", argv[i]);
        if (lstat(argv[i], &attribut) == -1)
            fehler_meld(WARNUNG_SYS, "....lstat-Fehler");
        else if (S_ISREG(attribut.st_mode)) printf("Regulaere Datei\n");
        else if (S_ISDIR(attribut.st_mode)) printf("Directory\n");
        else if (S_ISCHR(attribut.st_mode)) printf("Zeichenorient.Geraetedatei\n");
        else if (S_ISBLK(attribut.st_mode)) printf("Blockorient.Geraetedatei\n");
        else if (S_ISFIFO(attribut.st_mode)) printf("FIFO\n");
```

```

#ifdef S_ISLNK
    else if (S_ISLNK(attribut.st_mode)) printf("Symbolischer Link\n");
#endif
#ifdef S_ISSOCK
    else if (S_ISSOCK(attribut.st_mode)) printf("Socket\n");
#endif
    else printf("Unbekannte Dateart\n");
}
exit(0);
}

```

Programm 5.1 (**dateiart.c**): Ausgeben der Dateart von Dateien

Nachdem man Programm 5.1 (`dateiart.c`) kompiliert und gelinkt hat

```
cc -o dateiart dateiart.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ dateiart /etc/passwd /home /dev/tty /dev/fd0 /var/spool/cron/FIFO /dev/printer /dev/cdrom
      /etc/passwd: Regulaere Datei
      /home: Directory
      /dev/tty: Zeichenorient. Geraetedatei
      /dev/fd0: Blockorient. Geraetedatei
/var/spool/cron/FIFO: ....lstat-Fehler: Permission denied
      /dev/printer: Socket
      /dev/cdrom: Symbolischer Link
$

```

Hinweis

Ältere Unix-Versionen stellten die Makros `S_IS...` aus Tabelle 5.1 nicht zur Verfügung. In solchen Versionen muß man die Komponente `st_mode` und die Konstante `S_IFMT` mit bitweisem AND (&) verknüpfen und das Ergebnis dieser Operation mit den entsprechenden Konstanten vergleichen. Die Namen dieser Konstanten sind dort dann in `<sys/stat.h>` definiert und entsprechen den Makronamen aus Tabelle 5.1, nur daß sie als Präfix nicht `S_IS`, sondern `S_IF` haben. Um z.B. in solchen Systemen zu überprüfen, ob eine reguläre Datei vorliegt, müßte man den folgenden Ausdruck angeben:

```
if ( ((variable.st_mode) & S_IFMT) == S_IFREG)
```

5.3 Zugriffsrechte einer Datei

Die Komponente `st_mode` der Struktur `stat` enthält neben der Dateart auch die Zugriffsrechte einer Datei. Unix kennt für eine Datei neben den einfachen Zugriffsrechten (*read*, *write*, *execute*) für die drei Benutzerklassen (*owner*, *group*, *others*) noch das *Set-User-ID*-Bit, das *Set-Group-ID*-Bit und das *Sticky*-Bit.

5.3.1 Einfache Zugriffsrechte für die drei Benutzerklassen

Jeder Datei (reguläre Datei, Directory ...) ist ein aus 9 Bit bestehendes Zugriffsrechte-muster zugeordnet. Jeweils 3 Bits geben dabei die Zugriffsrechte (*read*, *write*, *execute*) der entsprechenden Benutzerklasse (*owner*, *group*, *others*) an. In Tabelle 5.2 sind die einzelnen Zugriffsrechte mit den entsprechenden Konstanten, mit denen sie abgeprüft werden können, zusammengefaßt.

Konstante	Bedeutung
S_IRUSR	<i>user-read</i> (Leserecht für Dateieigentümer)
S_IWUSR	<i>user-write</i> (Schreibrecht für Dateieigentümer)
S_IXUSR	<i>user-execute</i> (Ausführrecht für Dateieigentümer)
S_IRGRP	<i>group-read</i> (Leserecht für Gruppe des Dateieigentümers)
S_IWGRP	<i>group-write</i> (Schreibrecht für Gruppe des Dateieigentümers)
S_IXGRP	<i>group-execute</i> (Ausführrecht für Gruppe des Dateieigentümers)
S_IROTH	<i>other-read</i> (Leserecht für alle anderen Benutzer)
S_IWOTH	<i>other-write</i> (Schreibrecht für alle anderen Benutzer)
S_IXOTH	<i>other execute</i> (Ausführrecht für alle anderen Benutzer)

Tabelle 5.2: Einfache Zugriffsrechte für die 3 Benutzerklassen (aus <sys/stat.h>)

Diese Zugriffsrechte können von Dateieigentümern mit dem Kommando **chmod** verändert werden.

Bezüglich der Zugriffsrechte sind folgende Punkte zu beachten:

- ▶ Das Leserecht für eine Datei legt fest, daß man diese Datei mit der Funktion **open** zum Lesen (`O_RDONLY` oder `O_RDWR`) eröffnen kann.
- ▶ Das Schreibrecht für eine Datei legt fest, daß man diese Datei mit der Funktion **open** zum Schreiben (`O_WRONLY` oder `O_RDWR`) oder zum vollständigen Überschreiben (`O_TRUNC`) eröffnen kann.
- ▶ Um eine neue Datei anzulegen oder eine bereits existierende Datei zu löschen, benötigt man im entsprechenden Directory Schreib- und Ausführrechte. Wichtig ist, daß man keine Lese-, Schreib- oder Ausführrechte für eine zu löschende Datei selbst benötigt.
- ▶ Um eine Datei unter Angabe ihres Pfadnamens zu öffnen, muß man in jedem im Pfadnamen angegebenen Directory Ausführrechte besitzen. Um z.B. die Datei `/home/hans/doku12` zu öffnen, benötigt man Ausführrechte für die Directories `/`, `/home` und `/home/hans`. Zusätzlich braucht man natürlich, abhängig von gewünschten Öffnungsmodi, die entsprechenden Rechte (*read-only*, *read-write*, usw.) für die Datei `doku12` selbst.

- ▶ Um eine Datei im Working-Directory zu öffnen, muß man das Ausführrecht für das Working-Directory besitzen. Befindet man sich z.B. gerade im Directory `/home/hans`, dann muß man Ausführrechte für dieses Directory besitzen, wenn man die Datei `doku12` öffnen möchte, denn diese Namensangabe ist lediglich die Kurzform für die relative Pfadangabe `./doku12`.
- ▶ Leseerlaubnis für ein Directory berechtigt zum Lesen des Directory-Inhalts, was bedeutet, daß man die in diesem Directory enthaltenen Dateinamen erfragen darf. So kann man z.B. das Kommando **ls** nur für ein Directory erfolgreich aufrufen, für das man auch Leserecht hat.
- ▶ Ausführrecht für ein Directory erlaubt das Wechseln zu oder auch durch dieses Directory, wenn es Teil eines Pfadnamens ist.
- ▶ Um eine Datei mit den in Kapitel 10.5 beschriebenen **exec**-Funktionen ausführen zu lassen, muß man Ausführrechte für diese Datei haben.

5.3.2 Set-User-ID und Set-Group-ID

Jede Datei hat einen Eigentümer und einen Gruppeneigentümer. Der Eigentümer ist durch die Komponente `st_uid` und der Gruppeneigentümer durch die Komponente `st_gid` in der Struktur `stat` festgelegt.

Jedem Prozeß (ablaufendes Programm) wird nun neben der *realen User-ID* und der *realen Group-ID* des Aufrufers noch eine sogenannte *effektive User-ID* und *effektive Group-ID* zugeordnet. Normalerweise ist die effektive User-ID gleich der realen User-ID und die effektive Group-ID ist gewöhnlich auch gleich der realen Group-ID.

Da sich die realen und effektiven IDs aber auch unterscheiden können, existieren neben den zuvor vorgestellten einfachen Zugriffsrechten (für die 3 Benutzerklassen) für eine Datei noch das *Set-User-ID*-Bit und das *Set-Group-ID*-Bit (in `st_mode` der Struktur `stat`), was, wenn eines oder auch beide gesetzt sind, dazu führt, daß sich die entsprechende reale und effektive User-ID / Group-ID eines Prozesses unterscheidet.

Ist z.B. das Set-User-ID-Bit für eine Datei gesetzt, so wird bei der Ausführung dieser Datei dem entsprechenden Prozeß als effektive User-ID die User-ID des Dateieigentümers (aus `st_uid`) und nicht seine eigene User-ID zugewiesen. Somit unterscheidet sich in diesem Fall die reale User-ID (ID des Aufrufers) von der effektiven User-ID (ID des Dateieigentümers).

Wenn z.B. der Eigentümer eines Programms der Superuser ist, und für dieses Programm ist das Set-User-ID-Bit gesetzt, dann hat jeder Aufrufer dieses Programms für die Dauer der Ausführung die Superuser-Privilegien. Ein typisches Beispiel für ein solches Programm, bei dem das Set-User-ID-Bit gesetzt ist, ist das Kommando **passwd**, mit dem jeder Benutzer sein Paßwort ändern kann. Das set-User-ID Bit ist in diesem Fall notwendig, damit jeder Benutzer mittels des Kommandos **passwd** sein neues Paßwort in die dem Superuser gehörigen und schreibgeschützten Dateien `/etc/passwd` oder `/etc/shadow` eintragen kann.

Genauso kann auch das Set-Group-ID Bit gesetzt werden, was bewirkt, daß die effektive Group-ID für die Dauer der Ausführung des entsprechenden Programms gleich der Group-ID des Dateieigentümers (aus `st_gid`) ist.

Um zu erfahren, ob das Set-User-ID-Bit oder Set-Group-ID-Bit für eine Datei gesetzt ist, muß man die Komponente `st_mode` mit den Konstanten `S_ISUID` oder `S_ISGID` mit **&** (bitweises AND) verknüpfen, wie z.B.:

```
if (variable.st_mode & S_ISUID)
    printf("Set-User-ID-Bit gesetzt\n");
else
    printf("Set-User-ID-Bit nicht gesetzt\n");
```

Während die User-ID (`st_uid`) und die Group-ID (`st_gid`) immer der entsprechenden Datei zugeordnet sind, sind die effektive User-ID und die effektive Group-ID (eventuell mit zusätzlichen Group-IDs¹) immer dem Prozeß zugeordnet.

Abbildung 5.1 zeigt die Reihenfolge der Zugriffsprüfungen, die der Kern jedesmal durchführt, wenn ein Prozeß auf eine Datei zugreifen (Lesen, Schreiben, Ausführen) möchte.

Hinweis

In BSD-Unix ist eine Sicherung eingebaut, die den Mißbrauch der Set-User-ID- oder Set-Group-ID-Bits verhindern soll. Sobald ein Prozeß, der keine Superuser-Rechte hat, in eine Datei schreibt, werden für diese Datei in jedem Fall das Set-User-ID-Bit und das Set-Group-ID-Bit gelöscht. Dies macht auch Sinn. Nehmen wir z.B. an, daß ein Benutzer eine Datei mit den folgenden Zugriffsrechten besitzt:

```
rws rwx rwx (s bedeutet Set-User-ID Bit gesetzt)
```

Ein böswilliger Benutzer könnte nun ein Shell-Programm wie z.B. `/bin/sh` in diese Datei kopieren. Nun müßte er nur noch diese Datei (nun ein Shell-Programm) aufrufen und würde für die Dauer der Shell-Ausführung als effektive User-ID die UID dieses Benutzers zugeteilt bekommen. Ihm stünden somit alle Dateien dieses Benutzers ungehindert zur Verfügung, und er könnte diese beliebig verändern, lesen oder sogar löschen.

5.3.3 Saved Set-User-ID und Saved Set-Group-ID

Das *Saved Set-User-ID*-Bit und *Saved Set-Group-ID*-Bit erhält beim Start eines Programms eine Kopie der effektiven User-ID und der effektiven Group-ID.

Diese beiden Bits werden weiter unten bei der Vorstellung der Funktion ***setuid*** genauer beschrieben.

1. Zusätzliche Group-IDs (supplementary Group-IDs) sind in Kapitel 6.2 beschrieben

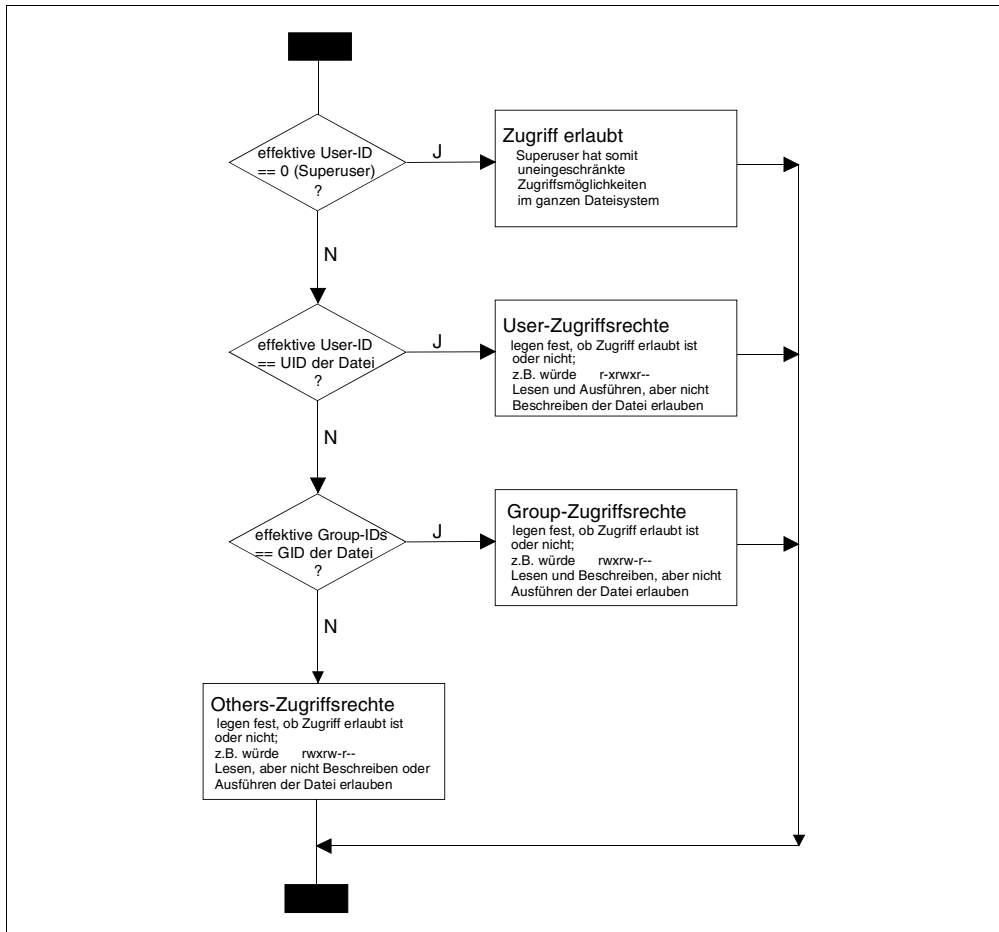


Abbildung 5.1: Zugriffsprüfungen bei Start eines Programms durch den Kern

Hinweis

Während SVR4 diese beiden Bits zwingend vorschreibt, sind sie in POSIX.1 optional. Um festzustellen, ob die jeweilige Implementierung diese Bits kennt, gibt es zwei verschiedene Möglichkeiten

- ▶ Abprüfen der Konstante `_POSIX_SAVED_IDS` zur Kompilierungszeit.
- ▶ Aufruf von `sysconf(_SC_SAVED_IDS)` zur Laufzeit.

5.3.4 Eigentümer von neuen Dateien

Als Eigentümer für eine mit **open** oder **creat** (siehe Kapitel 4.2) neu angelegte Datei wird immer die effektive User-ID des Prozesses eingetragen. Bezüglich der für eine neue Datei einzutragenden Group-ID läßt POSIX.1 die folgenden beiden Alternativen zu:

1. Als Group-ID für die neue Datei wird die effektive GID des Prozesses eingetragen.
2. Als Group-ID für die neue Datei wird die Group-ID des Directorys eingetragen, in dem die Datei angelegt wurde. Hiermit wird eine konsistente Gruppenzugehörigkeit für einen ganzen Directory-Baum (wie z.B. `/var/spool`) sichergestellt.

Hinweis

SVR4 verwendet die erste Alternative, wenn für das entsprechende Directory, in dem die neue Datei angelegt wird, nicht das Set-Group-ID-Bit gesetzt ist, andernfalls benutzt es die zweite Alternative.

BSD-Unix verwendet immer die zweite Alternative.

Bei anderen Systemen ist es beim Montieren des entsprechenden Dateisystems mit dem Kommando **mount** die Angabe einer speziellen Option möglich, um zwischen diesen beiden Alternativen zu wählen.

5.3.5 Sticky-Bit (Saved-Text-Bit)

Wenn das sogenannte *Sticky-Bit* für eine ausführbare Programmdatei gesetzt ist, dann wird nach dem ersten Aufruf dieses Programms das Textsegment (enthält den ausführbaren Programmcode) in den Swap-Bereich kopiert. Dies bewirkt, daß bei einem erneuten Aufruf dieses Programm wesentlich schneller in den Hauptspeicher geladen und somit natürlich auch schneller gestartet werden kann. Das Sticky-Bit wurde vor allen Dingen in früheren Unix-Versionen für häufig verwendete Programme wie Editoren oder C-Compiler gesetzt. Da der Swap-Bereich jedoch nur eine begrenzte Größe hat, konnte das Sticky-Bit natürlich nur für wenige ausgewählte Programme gesetzt werden.

In späteren Unix-Versionen sprach man nicht mehr vom Sticky-Bit, sondern vom *Saved-Text-Bit*, da nur das Textsegment im Swap-Bereich gehalten wird.

Bei heutigen Systemen, die mit schnelleren und virtuellen Dateisystemen arbeiten, besteht keine Notwendigkeit mehr für diese alte Funktion des *Saved-Text-Bits*. Deswegen hat man die Bedeutung des *Saved-Text-Bits* auf Directories erweitert. Ist in heutigen Unix-Systemen das *Saved-Text-Bit* für ein Directory gesetzt, so kann ein Benutzer eine Datei in diesem Directory nur dann löschen oder umbenennen, wenn er Schreibrechte für dieses Directory besitzt, und entweder Eigentümer der Datei, Eigentümer des Directorys oder aber Superuser ist.

Um zu überprüfen, ob das *Saved-Text-Bit* für eine Datei gesetzt ist, muß die Komponente `st_mode` mit der Konstanten `S_ISVTX` mit **&** (bitweises AND) verknüpft werden, wie z.B.:

```
if (variable.st_mode & S_ISVTX)
    printf("Saved-Text-Bit gesetzt\n");
else
    printf("Saved-Text-Bit nicht gesetzt\n");
```

Hinweis

Das Sticky-Bit kann in älteren Unix-Systemen nur vom Superuser gesetzt werden. So wird verhindert, daß der Swap-Bereich überläuft, da der Superuser nur wenige ausgewählte Programme für den Swap-Bereich vorsieht.

Ein typisches Beispiel für ein Directory mit gesetztem *Saved-Text-Bit* ist `/tmp`, denn in diesem Directory kann üblicherweise jeder Benutzer neue Dateien anlegen, wobei oft `rwxr-wxrw` als Zugriffsrechtemuster für diese Dateien gewählt wird. Trotz dieser freizügigen Zugriffsrechte sollte es jedoch keinem fremden Benutzer möglich sein, diese temporären Dateien zu löschen oder umzubenennen.

Das *Saved-Text-Bit* ist nicht in POSIX.1 definiert, wird aber von SVR4 und 4.BSD angeboten.

5.3.6 **chmod** und **fchmod** – Ändern der Zugriffsrechte für eine Datei

Um Zugriffsrechte einer bereits existierenden Datei zu ändern, stehen sie beiden Funktionen **chmod** und **fchmod** zur Verfügung.

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *pfd, mode_t modus);
int fchmod(int fd, mode_t modus);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Während mit **fchmod** nur die Zugriffsrechte einer bereits geöffneten Datei (mit Filedeskriptor `fd`) geändert werden können, ist dies bei **chmod** für eine nicht geöffnete Datei möglich.

modus

Für `modus` sind eine oder mehrere mit **|** (bitweises OR) verknüpfte Konstanten aus Tabelle 5.3 anzugeben. Die angegebenen Konstanten sind in `<sys/stat.h>` definiert.

Konstante	Bedeutung
S_ISUID	Set-User-ID-Bit
S_ISGID	Set-Group-ID Bit
S_ISVTX	Saved-Text Bit (Sticky Bit)
S_IRUSR	<i>read (user; Leserecht für Eigentümer)</i>
S_IWUSR	<i>write (user; Schreibrecht für Eigentümer)</i>
S_IXUSR	<i>execute (user; Ausführrecht für Eigentümer)</i>
S_IRWXU	<i>read, write, execute (user; Lese-, Schreib- und Ausführrecht für Eigentümer)</i>
S_IRGRP	<i>read (group; Leserecht für Gruppe)</i>
S_IWGRP	<i>write (group; Schreibrecht für Gruppe)</i>
S_IXGRP	<i>execute (group; Ausführrecht für Gruppe)</i>
S_IRWXG	<i>read, write, execute (group; Lese-, Schreib- und Ausführrecht für Gruppe)</i>
S_IROTH	<i>read (others; Leserecht für alle anderen Benutzer)</i>
S_IWOTH	<i>write (others; Schreibrecht für alle anderen Benutzer)</i>
S_IXOTH	<i>execute (others; Ausführrecht für alle anderen Benutzer)</i>
S_IRWXO	<i>read, write, execute (others; Lese-, Schreib- und Ausführrecht für alle anderen Benutzer)</i>

Tabelle 5.3: Mögliche Konstanten für *modus*-Argument bei **chmod** und **fchmod**.

Hinweis

Um die Zugriffsrechte für eine Datei zu ändern, muß die effektive User-ID des Prozesses gleich der User-ID des Dateieigentümers sein oder der Prozeß muß Superuser-Rechte haben.

fchmod ist nicht Bestandteil von POSIX.1, wird aber sowohl von SVR4 als auch 4.4BSD angeboten.

Die Konstante S_ISVTX ist nicht Bestandteil von POSIX.1.

Die beiden Funktionen **chmod** und **fchmod** löschen in den folgenden beiden Situationen automatisch das entsprechende Zugriffsrecht, selbst wenn es vom Aufrufer gefordert ist:

- ▶ Sticky-Bit (S_ISVTX) für eine reguläre Datei wird ausgeschaltet, wenn der Aufrufer nicht der Superuser ist.
- ▶ Set-Group-ID-Bit für eine neu angelegte Datei wird ausgeschaltet, wenn der Aufrufer nicht der Superuser ist und einer anderen Gruppe als die Datei angehört. Diese Situation liegt eventuell dann vor, wenn das System automatisch die neue Datei der gleichen Gruppe wie das Parent-Directory zuordnet (siehe auch zweite Alternative im vorherigen Unterpunkt »Neuer Eigentümer einer Datei«). So wird verhindert, daß ein Benutzer das Set-Group-ID Bit für eine Datei setzt, die einer Gruppe gehört, in der der Benutzer selbst nicht Mitglied ist.

Beispiel*Demonstrationsprogramm zur Funktion **chmod***

Das folgende Programm 5.2 (**chdemo.c**) vergibt an die Datei **ch1** das Zugriffsrechtemuster »**rwxr-x--x**« und löscht bei der Datei **ch2** das Ausführrecht für die Gruppe, setzt dafür aber das Set-User-ID-Bit und Set-Group-ID-Bit.

```
#include <sys/types.h>
#include <sys/stat.h>
#include "eighdr.h"

int
main(void)
{
    struct stat    dateiattr;

    /*--- Zugriffsrechtemuster "rwxr-x--x" fuer Datei ch1 setzen -----*/
    if (chmod("ch1", S_IRWXU | S_IRGRP|S_IXGRP | S_IXOTH) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei chmod (Datei 'ch1')");

    /*--- Bei Datei ch2 group-execute loeschen und set-user/group-ID setzen---*/
    if (stat("ch2", &dateiattr) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei stat (Datei 'ch2')");
    if (chmod("ch2", (dateiattr.st_mode & ~S_IXGRP) | S_ISUID | S_ISGID) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei chmod (Datei 'ch2')");

    exit(0);
}
```

*Programm 5.2 (**chdemo.c**): Demonstrationsbeispiel zur Funktion **chmod***

Nachdem man Programm 5.2 (**chdemo.c**) kompiliert und gelinkt hat

```
cc -o chdemo chdemo.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ touch ch1      [Anlegen der leeren Dateien ch1 und ch2]
$ touch ch2
$ ls -l ch[12]
-rw-r--r--  1 hh      bin          0 Sep 21 15:23 ch1
-rw-r--r--  1 hh      bin          0 Sep 21 15:23 ch2
$ chdemo
$ ls -l ch[12]
-rwxr-x--x  1 hh      bin          0 Sep 21 15:23 ch1
-rwSr-Sr--  1 hh      bin          0 Sep 21 15:23 ch2
$ chmod 750 ch[12]
$ ls -l ch[12]
-rwxr-x---  1 hh      bin          0 Sep 21 15:23 ch1
-rwxr-x---  1 hh      bin          0 Sep 21 15:23 ch2
$ chdemo
$ ls -l ch[12]
```

```
-rwxr-x--x  1 hh      bin          0 Sep 21 15:23 ch1
-rwsr-S---  1 hh      bin          0 Sep 21 15:23 ch2
$
```

Bei der Ausgabe von `ls -l` bedeutet in den Zugriffsrechten:

- ein großgeschriebenes **S**, daß hierfür das *Set-User-ID*-Bit bzw. *Set-Group-ID*-Bit, aber nicht zusätzlich das *Execute*-Recht gesetzt ist.
- ein kleingeschriebenes **s** bedeutet, daß hierfür das *Set-User-ID*-Bit bzw. *Set-Group-ID*-Bit und zusätzlich noch das *Execute*-Recht gesetzt ist.

Dieses Programm demonstriert neben dem absoluten Setzen von Zugriffsrechten (bei `ch1`) noch das relative Setzen von Zugriffsrechten (bei `ch2`). Um nur ein bestimmtes Zugriffsrecht `z` zu löschen, muß das von `stat` zurückgelieferte Muster wie folgt verknüpft werden:

```
dateiattr.st_mode & ~z
```

Soll zu einem bestehenden Zugriffsrechtemuster ein weiteres Zugriffsrecht `z` hinzugefügt werden, muß man folgende Konstruktion angeben

```
dateiattr.st_mode | z
```

Wie aus den Ablaufbeispielen ersichtlich wird, hat ***chmod*** keinen Einfluß auf die bei `ls -l` angezeigte Zeit der Datei. Die hier angezeigte Zeit bezieht sich nur auf die letzte Änderung des Dateiinhalts und der wird von ***chmod*** nicht verändert (siehe auch die Beschreibung von *i-nodes* in Kapitel 5.5).

5.3.7 access – Zugriffserlaubnis für reale User-/Group-ID auf eine Datei

In Abbildung 5.1 wurden die Prüfungen gezeigt, die der Kern jedesmal durchführt, wenn ein Prozeß auf eine Datei zugreifen (Lesen, Schreiben, Ausführen) möchte. Alle diese Überprüfungen werden – wie aus Abbildung 5.1 ersichtlich – mit der effektiven User-ID und der effektiven Group-ID durchgeführt. Möchte ein Prozeß aber die Zugriffsmöglichkeiten der realen User-ID und der realen Group-ID wissen, so muß er die Funktion ***access*** aufrufen.

```
#include <unistd.h>
```

```
int access(const char *pfad, mode_t modus);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Besteht für die reale User-ID bzw. reale Group-ID (in Abbildung 5.1 jedes »effektive« durch »reale« ersetzen) keine Zugriffserlaubnis für die Datei mit dem Namen `pfad`, so liefert ***access*** -1.

Für `modus` sind bei ***access*** eine oder mehrere mit `|` (bitweises OR) verknüpfte Konstanten aus Tabelle 5.4 anzugeben.

Konstante	Bedeutung
R_OK	Prüfung, ob Leserecht vorhanden
W_OK	Prüfung, ob Schreibrecht vorhanden
X_OK	Prüfung, ob Ausführrecht vorhanden
F_OK	Prüfung, ob Datei existiert

Tabelle 5.4: Mögliche Konstanten für *modus*-Argument bei ***access***

Die in Tabelle 5.4 angegebenen Konstanten sind in `<unistd.h>` definiert.

Beispiel

Demonstrationsprogramm zur Funktion *access*

```
#include <unistd.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int i;

    if (argc < 2)
        fehler_meld(FATAL, "usage: %s datei(en)", argv[0]);

    for (i=1 ; i<argc ; i++) {
        printf("%20s ", argv[i]);

        if (access(argv[i], F_OK) < 0)
            fehler_meld(WARNUNG, "existiert nicht");
        else {
            if (access(argv[i], R_OK) < 0) /*-- Testen der realen IDs */
                printf("-");
            else
                printf("r");

            if (access(argv[i], W_OK) < 0)
                printf("-");
            else
                printf("w");

            if (access(argv[i], X_OK) < 0)
                printf("-");
            else
                printf("x");

            if (open(argv[i], O_WRONLY) < 0) /*-- Testen der effektiven ID */
```

```

        printf("  -(effektiv)\n");
    else
        printf("  w(effektiv)\n");
    }
}
exit(0);
}

```

Programm 5.3 (**accesdem.c**): Demonstrationsbeispiel zur Funktion **access**

Nachdem man dieses Programm 5.3 (**accesdem.c**) kompiliert und gelinkt hat

```
cc -o accesdem accesdem.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ accesdem chmod* /etc/passwd
      chmodemo rwx  w(effektiv)
      chmodemo.c rw- w(effektiv)
      /etc/passwd r-- -(effektiv)
$ su          [Zum Superuser wechseln]
Password:    [hier Superuser-Passwort eingeben]
$ chown root accesdem  [Datei-Eigentümer von accesdem auf root setzen]
$ chmod u+s accesdem   [Set-User-ID Bit fuer accesdem setzen]
$ ls -l accesdem
-rwsr-xr-x  1 root    bin          16905 Sep 21 17:05 accesdem
$ exit        [Superuser-Session wieder verlassen (zurueck zum normalen Benutzer)]
$ accesdem chmod* /etc/passwd
      chmodemo rwx  w(effektiv)
      chmodemo.c rw- w(effektiv)
      /etc/passwd r-- w(effektiv)
$

```

An diesem Ablauf ist erkennbar, daß beim erstenmal für Datei `/etc/passwd` keinerlei Schreibzugriff (weder für reale noch effektive User-ID) besteht. Nachdem `root` sich zum Eigentümer des Programms `accesdem` gemacht und das Set-User-ID-Bit für diese Programmdatei gesetzt hat, wird die Datei `/etc/passwd` (entsprechend der Abbildung 5.1) für die effektive User-ID von `accesdem` nun beschreibbar, während das Schreiben für die reale User-ID weiterhin untersagt bleibt.

5.3.8 umask – Setzen und Abfragen der Dateikreierungsmasken

Um die Dateikreierungsmaske für einen Prozeß neu zu setzen oder aber deren momentanen Wert zu erfragen, steht die Funktion **umask** zur Verfügung.

```

#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t maske);

```

gibt zurück: vorherige Dateikreierungsmaske

Die Dateikreierungsmaske für einen Prozeß legt fest, welche Rechte beim Anlegen einer neuen Datei oder eines neuen Directorys nicht zu vergeben sind, selbst wenn sie bei den entsprechenden Routinen wie **open** oder **creat** im `modus`-Argument (siehe Kapitel 4.2) gefordert werden:

Für `maske` sind eine oder mehrere mit `|` (bitweises OR) verknüpften Konstanten aus Tabelle 5.5 anzugeben. Die angegebenen Konstanten sind in `<sys/stat.h>` definiert.

Konstante	Bedeutung
S_IRUSR	<i>read (user; Leserecht für Eigentümer)</i>
S_IWUSR	<i>write (user; Schreibrecht für Eigentümer)</i>
S_IXUSR	<i>execute (user; Ausführrecht für Eigentümer)</i>
S_IRWXU	<i>read, write, execute (user; Lese-, Schreib- und Ausführrecht für Eigentümer)</i>
S_IRGRP	<i>read (group; Leserecht für Gruppe)</i>
S_IWGRP	<i>write (group; Schreibrecht für Gruppe)</i>
S_IXGRP	<i>execute (group; Ausführrecht für Gruppe)</i>
S_IRWXG	<i>read, write, execute (group; Lese-, Schreib- und Ausführrecht für Gruppe)</i>
S_IROTH	<i>read (others; Leserecht für alle anderen Benutzer)</i>
S_IWOTH	<i>write (others; Schreibrecht für alle anderen Benutzer)</i>
S_IXOTH	<i>execute (others; Ausführrecht für alle anderen Benutzer)</i>
S_IRWXO	<i>read, write, execute (others; Lese-, Schreib- und Ausführrecht für alle anderen Benutzer)</i>

Tabelle 5.5: Mögliche Konstanten für `maske`-Argument bei **umask**

Beispiel

Demonstrationsprogramm zur Funktion umask

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(void)
{
    /*--- Alle Zugriffsrechte in Dateikreierungsmaske erlauben -----*/
    umask(0);

    /*--- Neue Datei 'uml' mit Zugriffsrechten "rw-r--r--" anlegen ---*/
    if (creat("uml", S_IRUSR|S_IWUSR | S_IRGRP | S_IROTH) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei creat (Datei 'uml')");

    /*--- Dateikreierungsmaske auf 137 setzen -----*/
```

```

umask(S_IXUSR | S_IWGRP|S_IXGRP | S_IROTH|S_IWOTH|S_IXOTH);

/*--- Neue Datei 'um2' mit Zugriffsrechten "rwxrwxrwx" anlegen ---*/
if (creat("um2", S_IRWXU | S_IRWXG | S_IRWXO) < 0)
    fehler_meld(FATAL_SYS, "Fehler bei creat (Datei 'um2')");

exit(0);
}

```

Programm 5.4 (**umaskdem.c**): Demonstrationsbeispiel zur Funktion **umask**

Das Programm 5.4 (**umaskdem.c**) setzt zuerst die Dateikreierungsmaske auf 0, was alle Zugriffsrechte für neue Dateien ermöglicht. Der nachfolgende **creat**-Aufruf erzeugt die Datei **um1** mit den Zugriffsrechten **rw-r--r--**, die wegen der Dateikreierungsmaske von 0 auch gewährt werden sollten. Mit einem zweiten **umask**-Aufruf wird die Dateikreierungsmaske **--x-wxrwX** (137) festgelegt, was bedeutet, daß für neue Dateien – unabhängig von den geforderten Rechten – dem Eigentümer kein Ausführrecht, der Gruppe keine Schreib- und Ausführrechte, und den anderen Benutzern überhaupt keine Rechte gewährt werden. Der nachfolgende **creat**-Aufruf legt dann die Datei **um2** an, für die er alle Rechte (**rwxrwxrwx**) fordert. Aufgrund der zu diesem Zeitpunkt gültigen Dateikreierungsmaske (**--x-wxrwX**) kann der Datei **um2** aber nur das Zugriffsrechtemuster **rw-r-----** zugeteilt werden.

Nachdem man dieses Programm 5.4 (**umaskdem.c**) kompiliert und gelinkt hat

```
cc -o umaskdem umaskdem.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ umask
22
$ umaskdem
$ ls -l um1 um2
-rw-r--r--  1 hh      bin          0 Sep 22 09:11 um1
-rw-r-----  1 hh      bin          0 Sep 22 09:11 um2
$ umask
22
$

```

Hinweis

Zum Anmeldezeitpunkt wird jedem Benutzer eine Dateikreierungsmaske, wie z.B. 022, zugeteilt. Möchte ein Benutzer seine eigene Dateikreierungsmaske festlegen, so kann er dies mit dem Builtin-Kommando **umask** der Shell erreichen. In diesem Fall ist es empfehlenswert, den entsprechenden **umask**-Aufruf in der entsprechenden Startup-Datei (wie **.profile** oder **.cshrc**) anzugeben, die beim Start der jeweiligen Shell, mit der man arbeitet, automatisch ausgeführt wird.

Um in einem eigenem Programm sicherzustellen, daß die geforderten Rechte beim Anlegen von neuen Dateien auch wirklich gewährt werden, ist es empfehlenswert, am Anfang des entsprechenden Programms folgenden Aufruf anzugeben:

umask(0)

Ein Prozeß erbt immer die Dateikreierungsmaske seines Elternprozesses und kann dann mit **umask** immer nur diese kopierte lokale Dateikreierungsmaske, niemals die seines Elternprozesses verändern.

Während die Dateikreierungsmaske Einfluß auf die bei **creat**, **open** oder **mknod** angegebenen Zugriffsrechte hat, so hat sie jedoch keinerlei Einfluß auf die bei **chmod** angegebenen Zugriffsrechte.

5.4 Eigentümer und Gruppe einer Datei

Jede Datei hat einen Eigentümer und einen Gruppeneigentümer. Der Eigentümer ist durch die Komponente `st_uid` und der Gruppeneigentümer durch die Komponente `st_gid` in der Struktur `stat` festgelegt. Diese geltenden Besitzverhältnisse einer Datei können mit einer der folgenden Funktionen geändert werden.

5.4.1 **chown**, **fchown** und **lchown** – Ändern der User-ID und Group-ID einer Datei

Um die User-ID und Group-ID einer Datei zu ändern, stehen die drei Funktionen **chown**, **fchown** und **lchown** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *pfad, uid_t eigentümer, gid_t gruppe);

int fchown(int fd, uid_t eigentümer, gid_t gruppe);

int lchown(const char *pfad, uid_t eigentümer, gid_t gruppe);
```

alle drei geben zurück: 0 (bei Erfolg); -1 bei Fehler

Während **fchown** nur auf eine geöffnete Datei (mit Filedeskriptor `fd`) angewendet werden kann, ist bei **chown** und **lchown** das Ändern der Besitzverhältnisse von nicht geöffneten Dateien möglich. **chown** und **lchown** unterscheiden sich in ihrem Verhalten nur bei symbolischen Links:

chown

Wird in SVR4 bei **chown** ein symbolischer Link angegeben, so wird der Eigentümer der Datei geändert, auf die der symbolische Link zeigt. In anderen Systemen (wie z.B. BSD-Unix) dagegen wird bei **chown** der Eigentümer des symbolischen Links selbst geändert. Um in diesen Systemen die Eigentümer der Datei zu ändern, auf die der symbolische Link zeigt, muß dort der Pfadname dieser entsprechenden Datei angegeben werden.

lchown

Diese Funktion ist nur unter SVR4 verfügbar. Wird bei **lchown** ein symbolischer Link angegeben, so wird der Eigentümer des symbolischen Links selbst geändert, und nicht der Datei, auf die der symbolische Link zeigt.

Konstante `_POSIX_CHOWN_RESTRICTED`

Wenn die POSIX.1-Konstante `_POSIX_CHOWN_RESTRICTED` in `<unistd.h>` definiert ist, so kann nur der Superuser den Eigentümer einer Datei ändern. Während in SVR4 diese Konstante bei der Konfiguration des Systems definiert wird (oder auch nicht), ist sie bei BSD-Unix immer definiert. Ob diese Konstante für ein spezielles System oder sogar für ein spezielles Filesystem gesetzt ist, kann mit dem Aufruf der Funktion ***pathconf*** oder ***fpathconf*** (siehe Kapitel 1.10) festgestellt werden.

Wenn `_POSIX_CHOWN_RESTRICTED` für eine Datei gesetzt ist, so gilt folgendes:

1. Nur ein Superuser-Prozeß kann die User-ID dieser Datei ändern.
2. Ein Nicht-Superuser-Prozeß kann die Group-ID einer Datei ändern, wenn er Eigentümer der Datei ist (effektive User-ID ist gleich der User-ID der Datei) und wenn zugleich das Argument `eigentümer` gleich der User-ID der Datei und das Argument `gruppe` gleich der effektiven Group-ID des Prozesses oder gleich einer der zusätzlichen Group-IDs (*supplementary Group-IDs*) des Prozesses ist.

Wenn also `_POSIX_CHOWN_RESTRICTED` definiert ist, kann ein »normaler« Benutzer nicht die User-ID von Dateien ändern, die ihm nicht gehören. Er kann aber die Group-ID von eigenen Dateien ändern, allerdings nur auf eine Gruppe, in der er selbst auch Mitglied ist.

Hinweis

Für die Argumente `eigentümer` oder `gruppe` darf -1 angegeben werden, wenn das entsprechende Besitzverhältnis nicht geändert werden soll. Dies ist jedoch nicht Bestandteil von POSIX.1.

Ist das Set-User-ID-Bit oder Set-Group-ID-Bit für eine Datei gesetzt, so wird es bei erfolgreichem Ablauf von diesen Funktionen gelöscht, wenn der aufrufende Prozeß nicht der Superuser ist.

5.5 Partitionen, Filesysteme und i-nodes

Für das Verständnis eines Filesystems und seines Aufbaus ist der i-node von fundamentaler Wichtigkeit. Zunächst werden hier die wichtigsten Filesysteme vorgestellt und die Zuordnung eines Filesystems zu einer Partition behandelt, bevor dann auf den i-node näher eingegangen wird.

5.5.1 Filesysteme

Inzwischen existieren eine Vielzahl von Filesystemen unter Unix. Das traditionelle Filesystem wurde in SVR4 durch das *Virtual File System* (VFS) ersetzt. Das VFS ist dabei die übergeordnete Schnittstelle im Systemkern zwischen den einzelnen Dateisystemen und dem Rest des Systemkerns (siehe auch Abbildung 5.2).

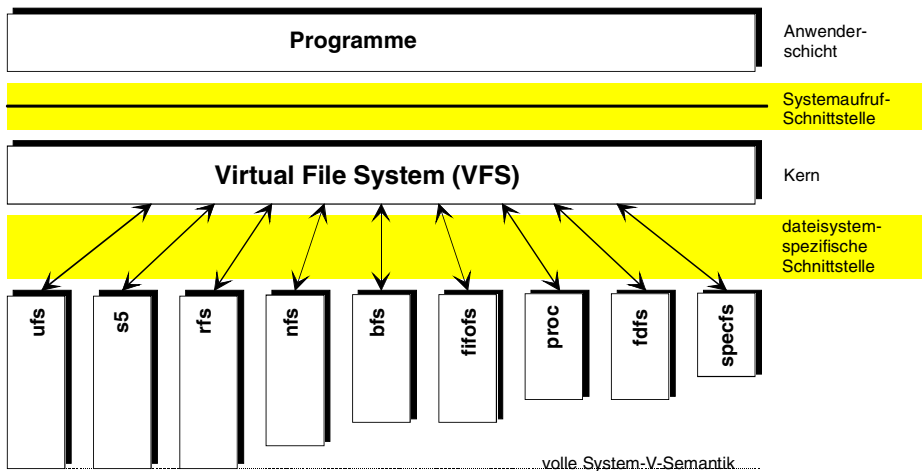


Abbildung 5.2: Das Virtual File System (VFS) von SVR4

Das VFS verwaltet die folgenden Dateisysteme:

s5

ist das traditionelle Dateisystem von SVR3, bei dem die Namen von Dateien nur 14 Zeichen lang sein dürfen. Intern ist das Dateisystem in Blöcken strukturiert. Die Blockgröße ist dabei einstellbar: 512 Byte, 1 oder 2 KByte. Das **s5**-Dateisystem ist aus Kompatibilitätsgründen noch in SVR4 enthalten, da manche Anwendungen (z.B. Datenbanken) diese interne Struktur voraussetzen. Bei anderen Programmen, die nicht diese Struktur voraussetzen, wird meist schon das neuere **ufs**-Dateisystem verwendet.

ufs

ist eine Implementierung des *Fast Filesystems* aus BSD-Unix. Bei diesem Dateisystem dürfen die Namen bis zu 255 Zeichen lang sein. Intern ist das Dateisystem in Blöcken strukturiert. Die Blockgröße ist dabei einstellbar auf 4 oder 8 KByte. Damit bei kleineren Dateien nicht zuviel Platz verschwendet wird, verwendet das **ufs**-Dateisystem fragmentierte Blöcke, so daß sich auf einem Block mehrere kleine Blöcke befinden können.

rfs

ist eine Implementierung des *Remote File Sharing* (RFS) von AT&T. RFS eignet sich hervorragend für homogene Netze, in denen ausschließlich System-V-Rechner miteinander vernetzt sind, da es hierbei einen netzweiten Zugriff auf die gemeinsamen Ressourcen der Systeme ermöglicht.

nfs

ist eine Implementierung des *Network File Systems* (NFS) von SunOS. Mit NFS können heterogene Netze aufgebaut werden, da NFS nicht nur für Unix-Systeme angeboten wird.

proc

ist ein ganzes neues Dateisystem in SVR4, über das auf Datenstrukturen von Prozessen zugegriffen werden kann. Ein aktiver Prozeß wird in diesem Dateisystem als Datei abgebildet und ein anderes Programm kann mit gewöhnlichen Systemaufrufen auf Daten dieses Prozesses zugreifen. Dieses Dateisystem wird hauptsächlich von Programmen benutzt, die den Prozeßverlauf verfolgen und darstellen.

bfs

enthält alle für den Systemstart notwendigen Dateien, den Kern und den *Bootloader*, der beim Systemstart den Kern in den Hauptspeicher lädt. In SVR3 setzte der Bootloader eine bestimmte Struktur des Root-Dateisystems voraus, da der Kern **unix** dort im Root-Directory untergebracht war. Durch die Einführung des **bfs**-Dateisystems, das nach dem Boot an das Directory **/stand** montiert wird, und die Verlagerung des Kerns in dieses Directory kann z.B. das Root-Dateisystem in einem Dateisystem beliebigen Typs (**s5** oder **ufs**) oder der Kern in einem EEPROM untergebracht sein.

fdfs

erlaubt Zugriffe auf Dateikanäle eines Prozesses.

fifofs

bietet eine Schnittstelle zu Named Pipes.

specfs

ist eine Schnittstelle zu den Gerädateien.

Während das **s5**-, das **ufs**- und das **rfs**-Dateisystem »echte« Dateisysteme sind, stehen auf den anderen Dateisystemen nicht unbedingt alle zur Dateibearbeitung notwendigen Operationen zur Verfügung.



Kaum ein anderes Betriebssystem unterstützt so viele Filesysteme wie Linux. Welche Filesysteme die aktuelle Linux-Version unterstützt, kann in der Datei `/usr/src/linux/fs/filesystems.c` nachgeschlagen werden.

An dieser Stelle ist darauf hinzuweisen, daß bei Nicht-Unix-Filesystemen oft nicht der volle Unix-Funktionsumfang angeboten wird: Zum Beispiel dürfen auf einem MS-DOS-Filesystem nur Dateinamen der Länge 8 plus 3 Zeichen für die Endung verwendet werden, auch wird dort nicht zwischen Groß- und Kleinschreibung unterschieden und es können keine Links erstellt werden usw.

Die wichtigsten von Linux unterstützten Filesysteme sind:

ext2 (*extended filesystem, Version2*)

dies ist heute das Standard-Filesystem unter Linux. Es unterstützt Dateinamen bis zu 255 Zeichen, Dateien bis zu 2 Gbyte und kann Datenträger bis zu 4 Tbyte (Terabyte = 1024 Gbyte) verwalten. Es gilt als das sicherste aller unter Linux verfügbaren Filesystemtypen.

ext

war der Vorgänger von ext2. Dieses Filesystem ist nur noch auf alten Linux-Distributionen (etwa bis 1993) zu finden und wird heute kaum mehr eingesetzt.

xiafs

wurde parallel zu ext und ext2 als ein weiteres neues Filesystem für Linux entwickelt, hat sich aber nicht durchgesetzt und wird heute kaum mehr eingesetzt.

minix

wurde ganz zu Anfang von Linux verwendet, wurde aber aufgrund einer Vielzahl von Mängeln sehr bald von ext abgelöst. minix wird aber weiter von Linux unterstützt, da viele frei verfügbaren Unix-Programme auch weiterhin auf Datenträger im minix-Format angeboten werden.

sysv

ermöglicht den Zugriff auf SCO-, XENIX- und Coherent-Partitionen.

ufs

ermöglicht den Lesezugriff auf Partitionen von SunOS, FreeBSD, NetBSD und Next-Step.

msdos

ermöglicht den Zugriff auf MS-DOS-Disketten und -Festplatten. Dabei ist nicht nur Lesen, sondern auch Schreiben möglich.

umsdos

ermöglicht wie das Filesystem msdos den Zugriff auf MS-DOS-Disketten und -Festplatten. Dabei ist auch wieder nicht nur Lesen, sondern auch Schreiben möglich. Im Unterschied zum msdos-Filesystem können hier auch lange Dateinamen mit Unix-Zugriffsrechten und Links verwendet werden. Dieses Filesystem wurde entwickelt, um Linux auch in einer MS-DOS-Partition zu installieren.

vfat

ermöglicht den Zugriff auf Filesysteme von Windows95. Dies funktioniert allerdings nur, wenn nicht Windows95-OEM bzw. Windows95b verwendet wird, denn diese Versionen verwenden ein neues, inkompatibles Filesystem namens vfat32. Windows-NT-FAT-Partitionen können ebenfalls als vfat-Partitionen angesprochen werden.

ntfs
ermöglicht nun auch den Zugriff auf das Windows-NT-Filesystem.

hpfs
ermöglicht den Lesezugriff auf Partitionen von OS/2.

iso9660
hat sich als Norm für die Dateiverwaltung auf CD-ROMs durchgesetzt.

nfs (*Network File System*)
ist unter Unix das übliche Netzwerk-Filesystem.

ncp (*Network Core Protocol*)
ist das Netzwerk-Filesystem von Novell.

smb (*Server Message Buffer*)
ist das Netzwerk-Filesystem von Microsoft.

proc
ist nicht wirklich ein Filesystem. Es wird vielmehr unter Linux zur Abbildung von Verwaltungsinformationen des Kernels bzw. der Prozeßverwaltung benutzt (dazu später mehr).



5.5.2 Partitionen und Filesysteme

Eine Festplatte (Disk) ist immer in eine oder mehrere Partitionen aufgeteilt, wobei jede Partition ihr eigenes Filesystem enthalten kann, wie dies in Abbildung 5.3 gezeigt ist.

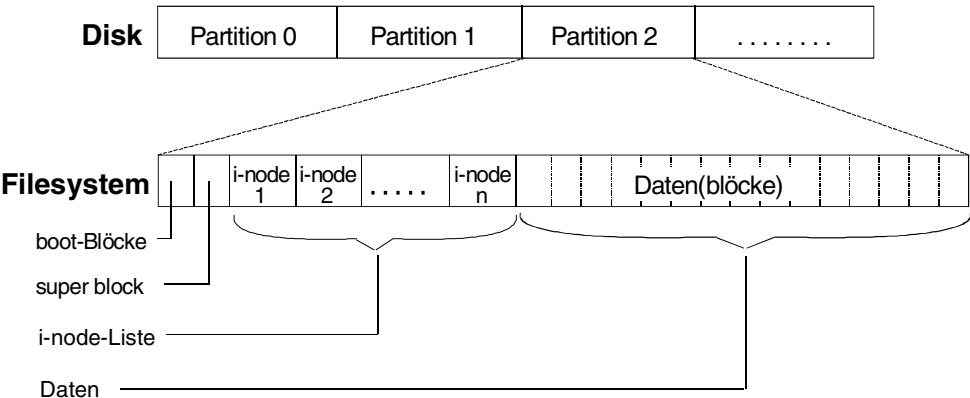


Abbildung 5.3: Disk, Partitionen und Filesysteme

Der Superblock enthält alle wichtigen Informationen, die für die Verwaltung des Filesystems notwendig sind. An späterer Stelle in diesem Kapitel wird der Aufbau des Superblocks an einem konkreten Filesystem (ext2) genauer beschrieben.

Der Boot-Block enthält ein kleines Programm zum Starten (Booten) des Betriebssystems. Da jedes Filesystem grundsätzlich den gleichen Aufbau haben soll, existiert der Boot-Block auch auf Filesystemen, die nicht für das Booten des Systems vorgesehen sind. In diesem Fall ist der Boot-Block zwar vorhanden, wird aber nicht genutzt.



Nachfolgend wird kurz der Boot-Prozeß unter Linux beschrieben:

Auf einem PC übernimmt das BIOS das Booten. Nach der Beendigung des POST (Power-On Self Test) versucht das BIOS, den ersten Sektor auf dem ersten Diskettenlaufwerk zu lesen. Ist dies nicht möglich, z.B. weil sich keine Diskette im Laufwerk befindet, versucht das BIOS als nächstes, den Boot-Sektor von der ersten Festplatte zu lesen².

Nach diesem Lesen des Boot-Sektors wird meist aus Platzgründen im Boot-Sektor ein zweiter Lader nachgeladen, der für das eigentliche Laden des Betriebssystems-kerns zuständig ist. Der Aufbau eines Boot-Sektors, der immer 512 Byte lang ist, wird in Abbildung 5.4 gezeigt.

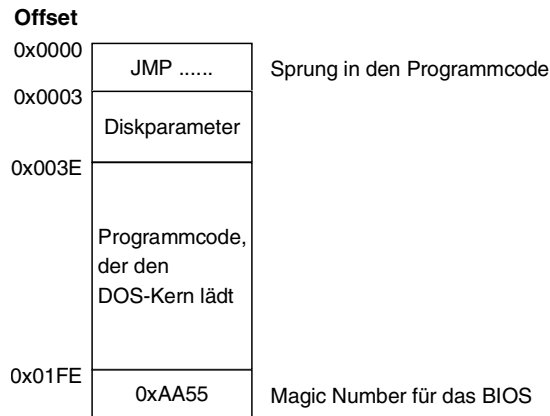


Abbildung 5.4: Boot-Sektor für MS-DOS

Dieser Boot-Sektor von Abbildung 5.4 ist für das Booten von einer Diskette geeignet, da eine Diskette nur eine Partition und damit auch nur einen Boot-Sektor enthält, der immer der erste Sektor ist.

2. Bei den neueren BIOS-Versionen kann diese Reihenfolge auch anders eingestellt werden.

Dagegen ist das Booten von einer Festplatte, die meist in mehrere Partitionen unterteilt ist und damit auch mehrere Boot-Sektoren (je Partition einen) enthält, etwas komplizierter. Bei Festplatten wird deshalb anstelle eines Boot-Sektors ein sogenannter MBR (Master Boot Record) verwendet, der ebenfalls an erster Stelle (auf der Partition) steht und vom BIOS gelesen wird.

Der MBR muß deshalb auch denselben Aufbau wie ein einfacher Boot-Sektor besitzen: am Anfang muß sich der Code und am Ende (Offset 0x01FE) muß sich die Magic Number 0xAA55 befinden. Nach dem Code ist – wie Abbildung 5.5 zeigt – die Partitionstabelle untergebracht.

Offset		Länge
0x0000	Code, der den Boot-Sektor der aktiven Partition lädt und startet	0x01BE
0x01BE	Partition 1	0x0010
0x01CE	Partition 2	0x0010
0x01DE	Partition 3	0x0010
0x01EE	Partition 4	0x0010
0x01FE	0xAA55	0x0002

Abbildung 5.5: Aufbau eines Master Boot Records (MBR)

Wie Abbildung 5.5 zeigt, ist der MBR nur für vier Partitionen auf einer Festplatte ausgelegt. Dies liegt daran, daß Festplatten nur in vier Partitionen, den sogenannten Primären Partitionen, unterteilt werden können. Sollte dies nicht ausreichen, kann eine sogenannte erweiterte Partition angelegt werden, die zumindest ein logisches Laufwerk enthält. Der erste Sektor einer erweiterten Partition enthält dann wieder einen MBR, wobei jedoch hier nun die erste Partition in der Partitionstabelle das erste logische Laufwerk der Partition enthält.

Falls mehrere logische Laufwerke existieren, so ist der zweite Eintrag in der Partitionstabelle ein Zeiger, der hinter das erste logische Laufwerk zeigt, wo sich wiederum eine Partitionstabelle mit dem Eintrag für das nächste logische Laufwerk befindet. Es wird also mit einer einfach vorwärts verketteten Liste für weitere logische Laufwerke gearbeitet, was bedeutet, daß eine erweiterte Partition theoretisch beliebig viele logische Laufwerke enthalten könnte.

Der erste Sektor einer jeden primären oder erweiterten Partition enthält einen Boot-Sektor mit dem bereits beschriebenen Aufbau. Welche von diesen Partitionen für das Booten verwendet wird, also die aktive Partition ist, wird über das Bootflag festgelegt. Die Auf-

gaben des Codes im MBR sind folglich: Ermitteln der aktiven Partition, Laden des Boot-Sektors der aktiven Partition mit Hilfe des BIOS und Sprung an den Anfang des Boot-Sektors.

Neben dem Standard-MS-DOS-MBR gibt es inzwischen viele Bootmanager, die alle entweder dem MBR durch eigenen Code ersetzen oder den Boot-Sektor einer aktiven Partition belegen. Der unter Linux übliche Bootmanager ist LILO (Linux Loader). Der LILO-Boot-Sektor enthält Platz für eine Partitionstabelle, weswegen LILO sowohl in einer Partition als auch in den MBR installiert werden kann. LILO besitzt die volle Funktionalität des Standard-MS-DOS-Boot-Sektors. Zusätzlich kann er auch logische Laufwerke oder Partitionen auf der zweiten, dritten ... Festplatte booten. LILO kann auch in Kombination mit einem anderen Bootmanager benutzt werden, so daß viele Installationsvarianten möglich sind, auf die hier nicht eingegangen wird, die aber in den Installationsmanuals von Linux ausführlich beschrieben sind.



5.5.3 Der i-node

Die zur Verwaltung nötigen Informationen werden unter Unix streng von den eigentlichen Dateien getrennt. Für jede Datei sind diese Verwaltungsinformationen in einem eigenen i-node (index node oder indirect node) untergebracht. Abbildung 5.6 zeigt den typischen Aufbau eines i-nodes unter Unix.

Die einzelnen i-nodes haben eine feste Länge im jeweiligen Filesystem und enthalten alle wesentlichen Informationen zu einer Datei, wie z.B. Zugriffsrechte, Eigentümer, Dateigröße, Dateart, Adressen der Datenblöcke dieser Datei usw. Ein Großteil der Information in der Struktur `stat` wird aus dem entsprechenden i-node gelesen.



Als Beispiel für die Adressen einer Datei soll hier der Adreßteil eines i-nodes im `ext2`-Filesystem von Linux dienen:

Die im i-node eines `ext2`-Filesystems gespeicherte Information entspricht weitgehend dem, was auch in anderen Filesystemen dort gespeichert wird, wie z.B. Kennung des Besitzers und der Gruppe, Zugriffsrechte, Dateigröße, Anzahl der Links, Zeitpunkt der Erstellung, der letzten Änderung, des letzten Lesezugriffs und des Löschens der Datei. Zur Adressierung der Daten stehen folgende Verweise zur Verfügung:

- ▶ Verweise auf die ersten 12 Datenblöcke der Datei
- ▶ Verweis auf 1. Indirektionsblock (einfach indirekt)
- ▶ Verweis auf 2. Indirektionsblock (zweifach indirekt)
- ▶ Verweis auf 3. Indirektionsblock (dreifach indirekt)

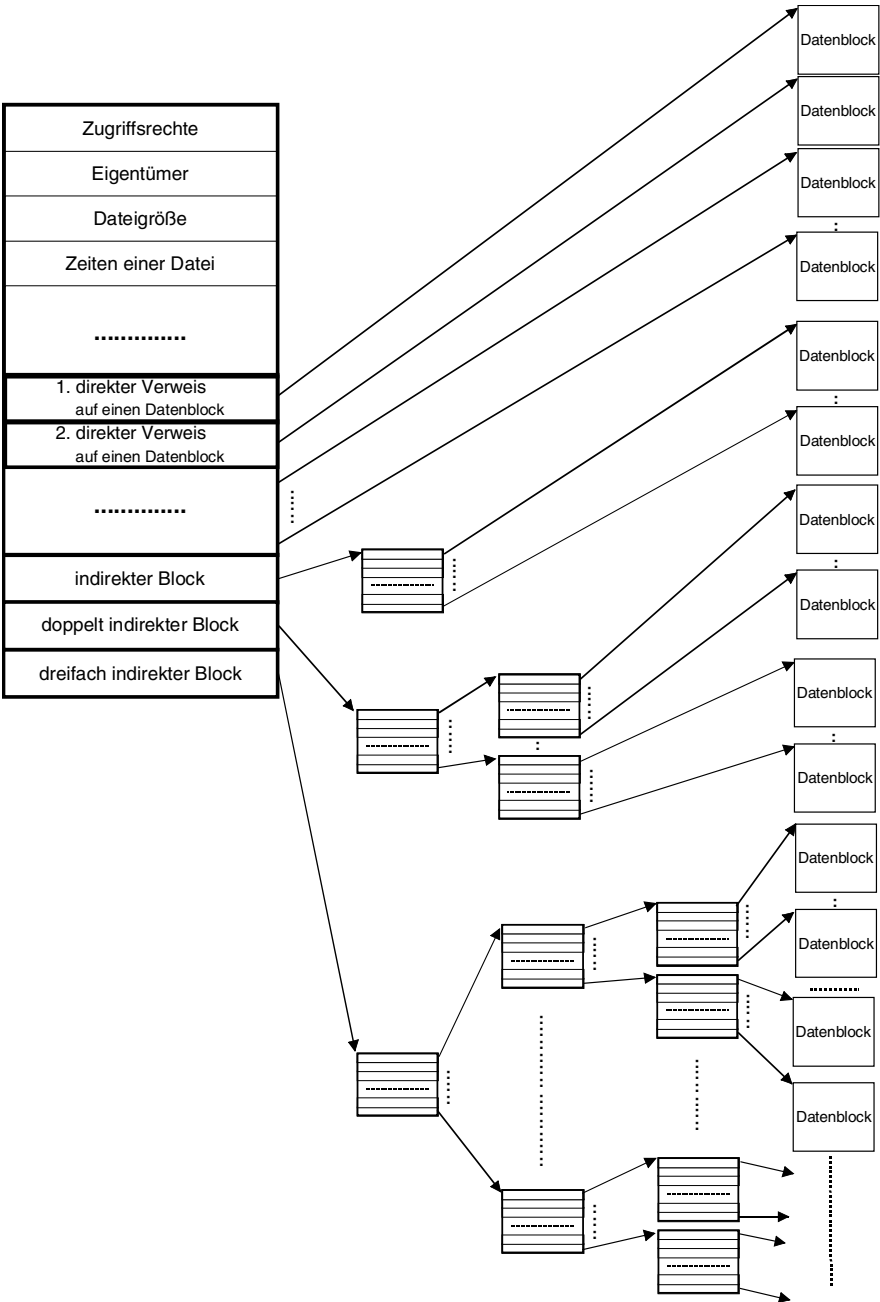


Abbildung 5.6: Typischer Aufbau eines i-nodes in einem Unix-Filesystem

Mit dieser Verweisstruktur können Dateien mit bis zu 16 Millionen Datenblöcken (=16 Gbyte) verwaltet werden, was sich aus folgender Rechnung ermitteln läßt:

$$12 + 256 + 256 \cdot 256 + 256 \cdot 256 \cdot 256 = 16843020 \text{ Datenblöcke mit 1KByte.}$$

Beim Formatieren eines ext2-Filesystems mit dem Kommando **mke2fs** kann die i-node-Dichte angegeben werden. Normalerweise wird beim Formatieren für je 4 Kbyte ein i-node vorgesehen, was z.B. bei einer Partition von 400 Mbyte 100000 i-nodes entspricht. Das bedeutet, daß in der Partition maximal 100000 Dateien gespeichert werden können, selbst wenn die Dateien sehr klein sind. Wenn also bekannt ist, daß auf einer Partition sehr viele kleine Dateien oder auch symbolische Links angelegt werden sollen, kann man beim Formatieren mit **mke2fs** auch eine größere i-node-Dichte wählen, wie z.B. ein inode für je 2 Kbyte.



Es ist offensichtlich, daß ein Zugriff auf kleine Dateien sehr schnell erfolgen kann, da dabei über die direkten Verweise im i-node ohne Zwischenschritt direkt auf die Datenblöcke dieser Dateien zugegriffen werden kann. Im ext2-Filesystem gilt dies für Dateien, die nicht größer als 12 Kbytes sind, da dort im i-node 12 direkte Verweise auf die ersten Datenblöcke vorhanden sind (siehe auch oben). Übersteigt eine Datei diese Größe, erfolgt der Zugriff über weitere Indirektionsstufen (bis zu dreifach, wie dies in Abbildung 5.6 gezeigt ist), was natürlich nicht so schnelle Zugriffe auf die entsprechenden Datenblöcke erlaubt wie bei den ersten 12 direkten Verweisen.

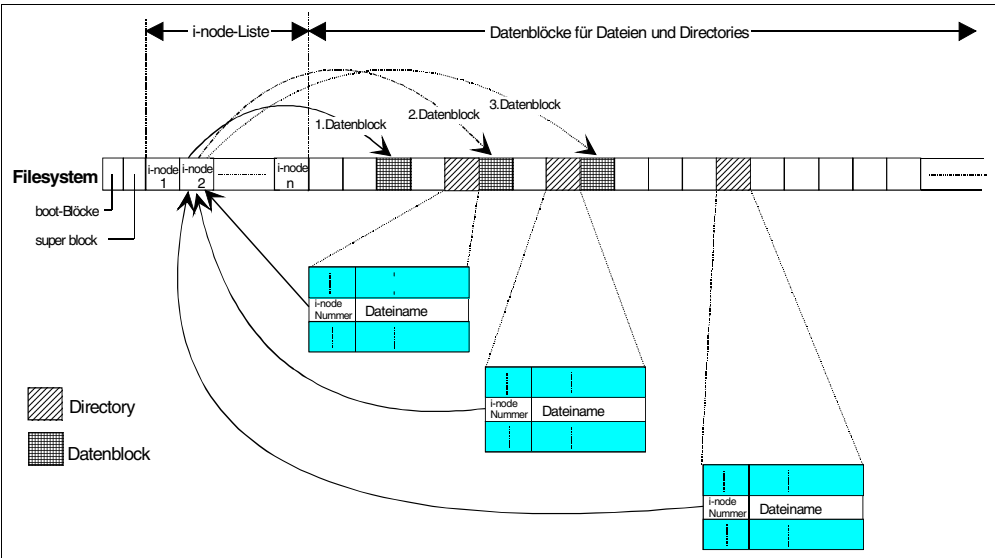


Abbildung 5.7: Detailliertere Darstellung eines typischen Unix-Filesystems

Jede Datei wird durch genau einen i-node repräsentiert. Innerhalb des Filesystems besitzt jeder i-node deshalb eine eindeutige Nummer. Somit läßt sich auch die Datei selbst über diese i-node-Nummer ansprechen. Diese Tatsache machen sich Directories zunutze, die für den hierarchischen Aufbau eines Filesystems verantwortlich sind. Sie liegen ebenfalls als Dateien vor, wobei sie jedoch nur für jede Datei, die sich in diesem Directory befindet, folgende Information enthalten: Dateiname und dazugehörige i-node-Nummer.

Abbildung 5.7 zeigt eine detailliertere Sicht des Filesystems.

Hinweis

In BSD-Unix umfaßt ein i-node 128 Bytes. In SVR4 hängt die Größe eines i-nodes vom filesystem-Typ ab: In **s5** 64 Bytes und in **ufs** (*Unified File System*) 128 Bytes.

5.5.4 Hard-Links

Unter Unix werden auch Directories als Dateien realisiert. Für jede Datei in einem Directory existieren in der Directory-Datei zwei Einträge:

i-node-Nummer | Dateiname

Wenn eine neue Datei in einem Directory angelegt wird, so wird zunächst ein i-node für diese Datei in der i-node-Liste erzeugt, und dann die i-node-Nummer und der Name der neuen Datei in der entsprechenden Directory-Datei eingetragen.

Ein neuer i-node wird jedoch nur dann erzeugt, wenn es sich bei der neuen Datei nicht um einen Link handelt. Denn im Falle eines Links, der mit dem Kommando **ln** angelegt werden kann, existiert bereits ein i-node für die »Originaldatei«, und es wird nur deren i-node-Nummer und der neue Dateiname in das Directory eingetragen. So zeigt z.B. die Abbildung 5.4 eine Situation, in der die Daten einer Datei (mit i-node 2) physikalisch nur einmal vorhanden sind. Diese Datei kann aber über drei verschiedene Namen, die sich in verschiedenen Directories befinden, angesprochen werden.

Diese Art von Links werden mit *Hard-Links* bezeichnet. Daneben gibt es noch die symbolischen Links, die in Kapitel 5.6 vorgestellt und mit *Soft-Links* bezeichnet werden.

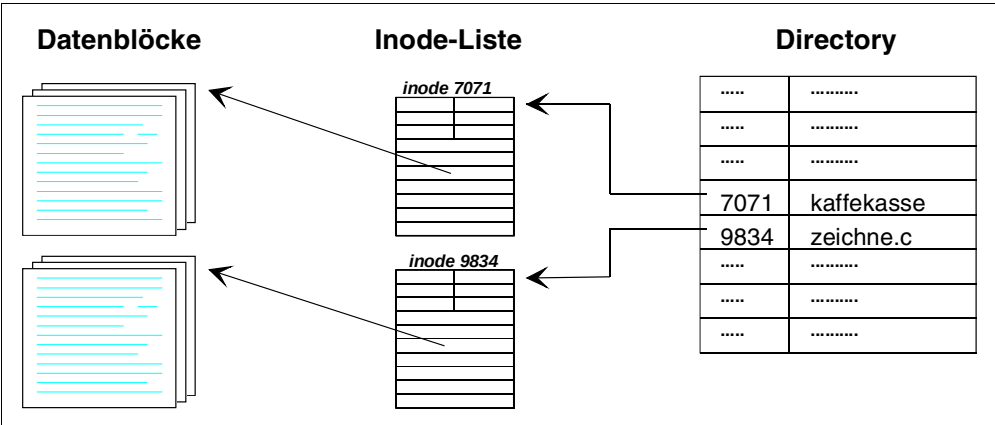


Abbildung 5.8: Zwei »echte« Dateien kaffee-kasse und zeichne.c (Ausgangssituation)

Wenn man z.B. die in Abbildung 5.5 gezeigte Konstellation hat und man erzeugt mit

```
ln kaffee-kasse cafe
```

einen Hard-Link `cafe` (auf `kaffee-kasse`), dann wird keine neue Datei angelegt, sondern es wird im Directory lediglich ein neuer Eintrag `cafe` eingetragen, der die gleiche i-node-Nummer erhält wie `kaffee-kasse` (7071). Abbildung 5.6 zeigt diese neue Konstellation.

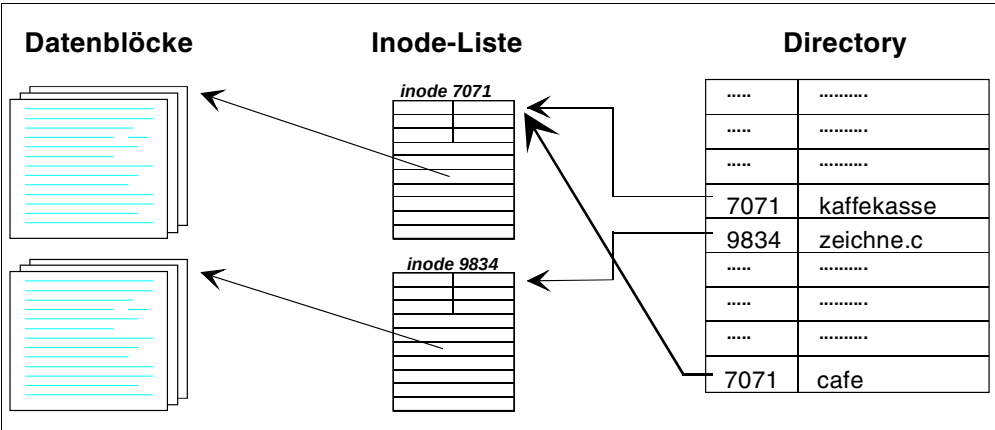


Abbildung 5.9: Auswirkung von »ln kaffee-kasse cafe« auf die Ausgangssituation in Abb. 5.5

Ein Zugriff auf `cafe` liefert somit immer das gleiche wie ein Zugriff auf die Datei `kaffee-kasse`. So gibt z.B. sowohl

```
cat kaffee-kasse
```

als auch

```
cat cafe
```

das gleiche am Bildschirm aus.

Jeder i-node hat einen sogenannten Link-Zähler, der angibt, wie viele Links (Dateinamen) momentan auf diesen i-node zeigen. Bei einem neuen Hinzufügen eines Links wird dieser Zähler inkrementiert und bei einem Löschen eines Links wird er dekrementiert. Erst wenn dieser Link-Zähler 0 wird, können die Datenblöcke zu diesem i-node und der i-node selbst freigegeben werden. Das Löschen einer Datei führt also nicht zur Freigabe der entsprechenden Datenblöcke, wenn noch weitere Links auf diese Datei existieren.

Neben dem Anlegen von Links auf reguläre Dateien ist es auch möglich, Links auf Directories anzulegen. Dies macht sich Unix z.B. immer beim Anlegen eines neuen Directories zunutze, wenn es dabei automatisch die beiden Einträge `.` (für Working-Directory) und `..` (für Parent-Directory) erzeugt. Der nachfolgende Ablauf verdeutlicht dies:

```
$ ls -ali
total 2
24134 drwxr-xr-x  2 hh      bin      1024 Sep 23 12:34 ./
12325 drwxr-xr-x 13 hh      users    1024 Sep 23 12:35 ../
24135 -rw-r--r--  1 hh      bin        0 Sep 23 12:34 datei1
24136 -rw-r--r--  1 hh      bin        0 Sep 23 12:34 datei2
24137 -rw-r--r--  1 hh      bin        0 Sep 23 12:34 datei3
24138 -rw-r--r--  1 hh      bin        0 Sep 23 12:34 datei4
$ mkdir subdir
$ cd subdir
$ ls -ali
total 2
24139 drwxr-xr-x  2 hh      bin      1024 Sep 23 12:37 ./
24134 drwxr-xr-x  3 hh      bin      1024 Sep 23 12:37 ../
$
```

Es ist hier erkennbar, daß beim Anlegen des neuen Directories `subdir` automatisch zwei neue Einträge generiert werden (`.` für Working-Directory und `..` für Parent-Directory). In beiden Fällen wird ein Hard-Link auf die schon existierenden Directories erzeugt. So sieht man z.B., daß `..` in `subdir` die gleiche i-node-Nummer hat wie `.` im Parent-Directory, nämlich 24134.

Bei der letzten `ls`-Ausgabe wird für das Parent-Directory `..` angezeigt, daß hierfür 3 Links existieren. Dies läßt sich auch nachvollziehen, denn es existiert zum einen der wirkliche Namenseintrag im Parent-Parent-Directory (`../..`), dann existiert im Parent-Directory der Link `.` (für Working-Directory), und im momentanen Subdirectory wurde mit `..` (für Parent-Directory) ein weiterer Link für dieses Directory erzeugt.

Hinweis

Die Struktur `stat` stellt den Inhalt des Link-Zählers über die Komponente `st_nlink` zur Verfügung.

Die POSIX.1-Konstante `LINK_MAX` legt die maximal mögliche Anzahl von Links fest, die für eine Datei existieren können.

Da die i-node-Nummer in einem Directory sich immer auf einen i-node im aktuellen Filesystem bezieht, kann ein Directory niemals einen Eintrag enthalten, der ein Link auf eine Datei in einem anderen Filesystem ist. Dies ist auch der Grund, warum das Kommando **ln** kein Anlegen von Hard-Links über Filesystem-Grenzen hinweg erlaubt.

Wenn eine Datei mit **mv** verlagert wird, so wird sie nicht wirklich physikalisch umkopiert, sondern es wird lediglich der neue Dateiname im entsprechenden Directory mit der gleichen i-node-Nummer eingetragen, bevor der alte Dateiname in der betreffenden Directory-Datei gelöscht oder durch Setzen der i-node-Nummer auf 0 als »gelöscht« markiert wird. Der Link-Zähler des i-nodes bleibt hierbei unverändert.

5.5.5 link – Erzeugen eines Links auf eine existierende Datei

Um auf eine existierende Datei einen Link zu erzeugen, steht die Funktion **link** zur Verfügung.

```
#include <unistd.h>
```

```
int link(const char *name, const char *linkname);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Funktion **link** erzeugt einen Hard-Link (zusätzlichen Dateinamen) *linkname*, der auf die existierende Datei *name* zeigt. Falls die Datei *linkname* bereits existiert, kann **link** diese nicht anlegen und liefert -1 (für Fehler) als Rückgabewert.

Hinweis

Während POSIX.1 Links über Filesystem-Grenzen hinweg zuläßt, ist dies in SVR4 und BSD-Unix nicht erlaubt.

Nur der Superuser kann Links auf Directories erzeugen. So soll vermieden werden, daß sich in Filesystemen endlose Rekursionen von Directories ergeben, die immer wieder auf sich selbst zeigen. Wären nämlich solche rekursiven Links auf Directories erlaubt, so könnte dies zu Endlosschleifen führen, wie dies im nachfolgenden hypothetischen Ablauf verdeutlicht wird:

```
$ mkdir dir1
$ touch dir1/datei
$ cd dir1
$ ln ../dir1 dir1/dir2
$ cd ..
$ ls -R dir1
./      ../      datei  dir2/

dir1/dir2:
./      ../      datei  dir2/

dir1/dir2/dir2:
```

```

./    ../    datei  dir2/

dir1/dir2/dir2/dir2:
./    ../    datei  dir2/

dir1/dir2/dir2/dir2/dir2:
./    ../    datei  dir2/

dir1/dir2/dir2/dir2/dir2/dir2:
./    ../    datei  dir2/

.....      [Endlos-Ausgabe, die niemals stoppt]
.....
.....
Ctrl-C      [Abbruch mit Ctrl-C]
$

```

Das Anlegen des Links (Datei `linkname`) und das Inkrementieren des Link-Zählers im `i-node` müssen eine atomare Operation sein.

5.5.6 **unlink** – Entfernen eines Dateinamens aus einem Directory

Um einen Dateinamen aus einem Directory zu entfernen, steht die Funktion ***unlink*** zur Verfügung.

```
#include <unistd.h>
```

```
int unlink(const char *name);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Funktion ***unlink*** entfernt den Dateinamen `name` aus der entsprechenden Directory-Datei und erniedrigt den Link-Zähler um 1. Falls der Link-Zähler dadurch 0 wird, so werden auch der zugehörige `i-node` und die physikalischen Daten zu dieser Datei freigegeben. Wird der Link-Zähler aber nicht 0, so bleibt der betreffende `i-node` weiterhin verfügbar, da in diesem Fall noch andere Dateinamen existieren, über die auf diese Datei zugegriffen werden kann.

Tritt bei der Ausführung von ***unlink*** ein Fehler auf, so bleibt der Dateiname `name` im entsprechenden Directory erhalten und die Funktion ***unlink*** hat keinerlei Auswirkung.

Hinweis

Um einen Dateinamen aus einem Directory mit ***unlink*** zu entfernen, muß man Schreib- und Ausführrechte für dieses Directory besitzen.

Um eine Datei in einem Directory, bei dem das Sticky-Bit gesetzt ist, löschen zu können, muß man Schreibrechte für dieses Directory besitzen und entweder Eigentümer der Datei oder Eigentümer des Directorys sein oder aber Superuser-Rechte besitzen.

Wenn eine Datei geschlossen wird, so prüft der Kern immer zuerst, ob noch weitere Prozesse diese Datei geöffnet haben. Wenn dies nicht der Fall ist, so prüft der Kern, ob der Link-Zähler im i-node gleich 0 ist. Nur wenn diese beiden Bedingungen erfüllt sind, wird die Datei auch physikalisch gelöscht.

Die beim **unlink**-Aufruf angegebene Datei wird nicht sofort entfernt, sondern erst wenn sich der Prozeß beendet, in dem **unlink** aufgerufen wurde. Diese Tatsache machen sich viele Programme zunutze, wenn sie temporäre Dateien benötigen, wie der nachfolgende Programmausschnitt zeigt:

```
if ( (fd=open("tempdatei", O_RDWR)) < 0)
    fehler_meld(FATAL_SYS, "kann tempdatei nicht oeffnen");

if (unlink("tempdatei") < 0) /* tempdatei loeschen (nicht wirklich) */
    fehler_meld(FATAL_SYS, "kann tempdatei nicht loeschen");

.....
/* Hier kann nun trotz des unlink-Aufrufs mittels des
   Filedesktors fd in die Datei "tempdatei"
   geschrieben oder aus ihr gelesen werden */
.....

exit(0); /* Jetzt erst wird "tempdatei" geschlossen und
          damit auch wirklich gelöscht */
```

Bei dieser Vorgehensweise ist sichergestellt, daß die entsprechende temporäre Datei bei Beendigung des Programms wirklich gelöscht wird, selbst wenn das Programm sich vorzeitig (z.B. durch einen Fehler oder ein Abbruchsignal) beendet, denn der Kern entfernt bei Ende dieses Prozesses, wenn er alle noch geöffneten Dateien schließt, in jedem Fall die als »gelöscht markierte« temporäre Datei.

Wenn bei **unlink** für *name* ein symbolischer Link angegeben ist, so wird der symbolische Link selbst und nicht die Datei, auf die dieser symbolische Link zeigt, gelöscht.

Nur der Superuser kann mit **unlink** ein Directory entfernen. Zum Entfernen eines Directories sollte jedoch die in Kapitel 5.9 beschriebene Funktion **rmdir** benutzt werden.

Mit der in Kapitel 3.8 beschriebenen Funktion **remove** steht eine weitere Funktion zum Löschen von Dateien zur Verfügung.

5.6 Symbolische Links

In SVR4 wurden sogenannte *symbolische Links* (Option **-s** beim Kommando **ln**) eingeführt, mit denen sich ebenfalls zusätzliche Namen an Dateien vergeben lassen. Anders als bei den in Kapitel 5.5 beschriebenen Links (*Hard-Links*) wird bei den symbolischen Links (*Soft-Links*) eine spezielle Datei erzeugt, die den Namen der Zieldatei enthält. Im Gegensatz zu den normalen Links erlauben symbolische Links auch Verweise auf Directories (bei Hard-Links nur Superuser erlaubt) und Verweise über Filesystem-Grenzen hinweg.

Zum Anlegen von symbolischen Links (Soft-Links) steht die Option **-s** zur Verfügung.

```
(1) ln -s datei1 datei2
(2) ln -s datei(en) directory
(3) ln -s dir1 dir2
```

Die einzelnen Aufrufe bewirken im einzelnen:

1. *datei2* wird als zusätzlicher Name für *datei1* angelegt, wobei jedoch die folgenden Ausnahmen gelten:
 - Wenn *datei2* bereits existiert, gibt **ln** immer einen Fehler aus.
 - Wenn beide Dateien nicht existieren, wird eine *datei2* angelegt, deren Inhalt der Name *datei1* ist. Bei Zugriffen auf *datei2* erscheint dann solange eine Fehlermeldung, bis *datei1* angelegt ist.
2. verhält sich weitgehend wie (1) mit dem Unterschied, daß im *directory* die Basisnamen der *datei(en)* als symbolische Links eingetragen werden.
3. verhält sich ebenfalls weitgehend wie (1), nur daß hier ein symbolischer Link *dir2* auf ein Directory *dir1* angelegt wird.

Löscht man die Zieldatei, auf die ein Soft-Link verweist, führt ein Zugriff auf die Datei über den Soft-Link zu einer Fehlermeldung. Richtet man später wieder eine Datei mit entsprechenden Namen ein, funktioniert alles wie zuvor.

Symbolische Links werden bei der Ausgabe mit **ls -l** durch die Angabe von **l** als erstes Zeichen gekennzeichnet. Zusätzlich wird

-> *name*

ausgegeben. *name* ist dabei die Datei, auf die dieser symbolische Link verweist, wie z.B.:

```
$ ls -ld /usr/spool /usr/tmp
lrwxrwxrwx 1 root root 12 May 5 10:28 /usr/spool -> ../var/spool/
lrwxrwxrwx 1 root root 10 May 5 10:28 /usr/tmp -> ../var/tmp/
$
```

Wird die Option **-F** beim **ls**-Kommando angegeben, werden symbolischen Links durch einen angehängten **@** gekennzeichnet, wie z.B.:

```
$ ls -F /usr
Info@      X11/      X386@     adm@      bin/
dict/      doc/      etc/      games/    include/
info/      lib/      local/    man/      openwin/
preserve@  sbin/     share/    spool@    src/
tmp@
$
```

Für die einzelnen Systemfunktionen ist es nun wichtig zu wissen

- ▶ ob sie den symbolischen Link folgen, also sich auf die Datei beziehen, auf die der Link zeigt, oder
- ▶ ob sie sich auf den symbolischen Link selbst beziehen.

Die Tabelle 5.6 zeigt das entsprechende Verhalten für die einzelnen Funktionen.

<i>Funktion</i>	<i>Symbolischer Link selbst</i>	<i>Folgt symbolischemLink</i>
access		x
chdir		x
chmod		x
chown	x	x (implementierungsabhängig; siehe Kapitel 5.4)
creat		x
exec		x
lchown	x	
link		x
lstat	x	
mkdir		x
mkfifo		x
mknod		x
open		x
opendir		x
pathconf		x
readlink	x	
remove	x	
rmdir	---- nicht definiert für symbolische Links (liefert Fehler)	
rename	x	
stat		x
truncate		x
unlink	x	

Tabelle 5.6: Verhalten der einzelnen Funktionen bei symbolischen Links

In der Tabelle 5.6 sind keine Funktionen aufgeführt, die ein Filedeskriptor-Argument erwarten, wie z.B. **fchdir**, **fchmod**, **fchown**, ..., da in diesem Fall die Auswertung des symbolischen Links bereits durch die entsprechende Öffnungsroutine (wie z.B. **open**) durchgeführt wird.

Hinweis

Eine Hauptanwendung von symbolischen Links sind Verweise über Filesystem-Grenzen hinweg oder Verweise auf Directories, die mit Hard-Links nicht möglich sind.

Ebenso werden symbolische Links oft in SVR4 verwendet, um eine zu SVR3 kompatible Directory-Struktur zu erhalten. So existieren z.B. Links für die Directories */bin* auf */usr/bin* und */lib* auf */usr/lib*.

Symbolische Links wurden mit 4.2BSD eingeführt und wurden in SVR4 neu eingeführt. Sie sind nun auch Bestandteil von POSIX.1.

5.6.1 Vorsicht mit endlosen rekursiven Links

Während Hard-Links auf Directories nur dem Superuser gestattet sind, sind symbolische Links auf Directories jedem einzelnen Benutzer erlaubt. Der Benutzer muß dabei jedoch darauf achten, daß sich keine endlosen Rekursionen von Directories ergeben, wie z.B.

```
$ mkdir dir1
$ touch dir1/datei      [Anlegen der leeren Datei dir1/datei]
$ ln ../dir1 dir1/dir2  [Symbol. Link von dir1/dir2 auf's eigene Parent-Directory]
$ ls -LR dir1           [Option -L ---> symbol. Link folgen]
./    ../    datei  dir2/

dir1/dir2:
./    ../    datei  dir2/

dir1/dir2/dir2:
./    ../    datei  dir2/

dir1/dir2/dir2/dir2:
./    ../    datei  dir2/

dir1/dir2/dir2/dir2/dir2:
./    ../    datei  dir2/

dir1/dir2/dir2/dir2/dir2/dir2:
./    ../    datei  dir2/

.....      [Endlos-Ausgabe, die niemals stoppt]
.....
.....
Ctrl-C      [Abbruch mit Ctrl-C]
$
```


Durch diese Kommandofolge haben wir in `dir1` ein Directory `dir2` angelegt, das auf sein eigenes Parent-Directory `dir1` zeigt. Abbildung 5.7 verdeutlicht die daraus resultierende Konstellation.

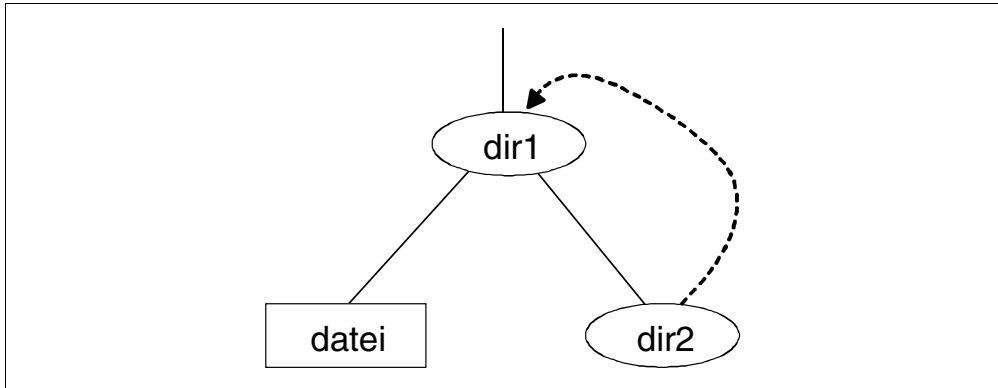


Abbildung 5.10: Symbolischer Link von Subdirectory auf sein eigenes Parent-Directory

Während die meisten Systemfunktionen eine Endlos-Rekursion bei symbolischen Links erkennen, und in diesem Fall die globale Variable `errno` auf `ELOOP` setzen, gilt dies nicht für die in Kapitel 5.9 vorgestellte Funktion **ftw** (*file transfer walk*) zum rekursiven Durchlauf von Directory-Bäumen. Mit SVR4 wurde deshalb die Funktion **nftw** (*new file transfer walk*) neu eingeführt, die dem Aufrufer über eine Option wählen lässt, ob symbolischen Links zu folgen ist oder nicht.

Hinweis

Das Löschen eines symbolischen Links ist leicht mit der Funktion **unlink** möglich, da **unlink** nicht die Datei, auf die der symbolische Link zeigt, sondern den symbolischen Link selbst löscht.

5.6.2 symlink – Anlegen eines symbolischen Link

Um einen symbolischen Link anzulegen, steht die Funktion **symlink** zur Verfügung.

```
#include <unistd.h>
```

```
int symlink(const char *ziel, const char *symlink);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

symlink erzeugt einen symbolischen Link (neue Datei) mit dem Namen `symlink` und dieser symbolische Link zeigt auf die Datei mit dem Pfadnamen `ziel`. Dabei müssen sich `ziel` und `symlink` nicht im gleichen Filesystem befinden.

5.6.3 readlink – Erfragen des Namens, auf den ein symbolischer Link zeigt

Um den Namen der Datei zu erfragen, auf die ein symbolischer Link zeigt, steht die Funktion **readlink** zur Verfügung.

```
#include <unistd.h>
```

```
int readlink(const char *symbollink, char *puffer, int puffgroesse);
```

gibt zurück: Anzahl der gelesenen Bytes des Pfadnamens,
auf die der symbol. Link zeigt (bei Erfolg); -1 bei Fehler

Da die Funktion **open** immer die Datei eröffnet, auf die ein symbolischer Link zeigt, wird mit **readlink** eine Funktion angeboten, die sich auf den symbolischen Link selbst bezieht.

readlink vereinigt in sich die drei Funktionen:

- ▶ **open** (Öffnen des symbolischen Links)
- ▶ **read** (Lesen des symbolischen Link-Inhalts = Dateiname, auf den symbolischer Link zeigt)
- ▶ **close** (Schließen des symbolischen Links)

Wenn die Funktion **readlink** erfolgreich ausgeführt wurde, liefert sie die Anzahl der gelesenen Bytes, die sie nach `puffer` geschrieben hat, als Rückgabewert. Der nach `puffer` geschriebene Name der »Zieldatei« wird dabei nicht mit `\0` abgeschlossen.

Beispiel

Demonstrationsprogramm zu den Funktionen symlink und readlink

Das folgende Programm 5.5 (`symlink.c`) liest aus den auf der Kommandozeile angegebenen Dateien die anzulegenden symbolischen Links. In dieser Datei müssen die einzelnen Zeilen folgenden Inhalt haben:

```
symlink_name  ziel_pfad
```

Das Programm legt dann für jede gültige Zeile einen symbolischen Link `symlink_name` an, der auf `ziel_pfad` zeigt.

```
#include <unistd.h>
#include "eighdr.h"
```

```
int
main(int argc, char *argv[])
{
    int    i, n;
    FILE   *dz;
    char   von[MAX_ZEICHEN], nach[MAX_ZEICHEN], puffer[MAX_ZEICHEN];
```

```

if (argc < 2)
    fehler_meld(FATAL, "usage: %s datei(en)", argv[0]);

for (i=1 ; i<argc ; i++) {
    if ( (dz=fopen(argv[i], "r")) == NULL)
        fehler_meld(WARNUNG_SYS, "kann %s nicht oeffnen", argv[i]);
    else {
        while (fscanf(dz, "%s %s", von, nach) != EOF) {
            fgets(puffer, MAX_ZEICHEN, dz); /* Rest der Zeile ignorieren */
            if (symlink(nach, von) == -1)
                fehler_meld(WARNUNG_SYS, "kann %s -> %s nicht anlegen",
                            von, nach);
            else if ( (n=readlink(von, puffer, MAX_ZEICHEN)) == -1)
                fehler_meld(WARNUNG_SYS, "Fehler bei Link %s", von);
            else
                printf("%20s -> %.*s angelegt\n", von, n, puffer);
        }

        fclose(dz);
    }
}

exit(0);
}

```

Programm 5.5 (**symlink.c**): Demonstrationsbeispiel zu den Funktionen **symlink** und **readlink**

Nachdem man das Programm 5.5 (symlink.c) kompiliert und gelinkt hat

```
cc -o symlink symlink.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ cat links.txt
hochfritz ../fritz
tmpdir /tmp
$ symlink links.txt
    hochfritz -> ../fritz angelegt
    tmpdir -> /tmp angelegt
$ ls -l hochfritz tmpdir
lrwxrwxrwx  1 hh      bin           8 Sep 26 14:19 hochfritz -> ../fritz
lrwxrwxrwx  1 hh      bin           4 Sep 26 14:19 tmpdir -> /tmp/
$

```

5.7 Größe einer Datei

Die Komponente `st_size` der Struktur `stat` enthält die Größe einer Datei in Byte. Der in `st_size` enthaltene Wert ist jedoch nur für reguläre Dateien, Directories und symbolische Links aussagekräftig. In SVR4 hat dieser Wert auch noch bei Pipes eine Bedeutung.

Blöcke

In einem Filesystem wird der verfügbare Speicherplatz nicht in einzelnen Bytes, sondern immer nur in Blöcken von Bytes vergeben. Die Blockgröße ist in den einzelnen Filesystemen unterschiedlich. Typische Blockgrößen sind 512 oder 1024 Bytes. Mit dem Kommando **du** kann man die von Dateien belegten Blöcke erfragen.

SVR4 und 4.4BSD bieten in der Struktur `stat` die beiden Komponenten `st_blksize` und `st_blocks` an. `st_blksize` enthält die voreingestellte Blockgröße für E/A-Operationen bei dieser Datei, und `st_blocks` enthält die Anzahl der von der entsprechenden Datei belegten 512-Byte-Blöcke.

Reguläre Dateien

Hier enthält `st_size` die Anzahl von Bytes, die in die entsprechende Datei geschrieben wurden, was nicht dem physikalischen Speicherplatz entsprechen muß, der durch diese Datei wirklich belegt wird, da dieser immer ein Vielfaches der Blockgröße ist.

```
$ ls -l cptime.c symlink.c
-rw-r--r--  1 hh      bin           1403 Jul 12 17:47 cptime.c
-rw-r--r--  1 hh      bin           953 Sep 26 14:17 symlink.c
$ du cptime.c symlink.c
2  cptime.c
1  symlink.c
$
```

Eine reguläre Datei kann auch die Dateigröße 0 haben.

```
$ touch leerdatei
$ ls -l leerdatei
-rw-r--r--  1 hh      bin           0 Sep 26 18:43 leerdatei
$ du leerdatei
0 leerdatei
$
```

Directory

Für Directories enthält `st_size` gewöhnlich einen Wert, der abhängig vom Filesystem ein Vielfaches von 16 oder 512 ist (siehe auch Kapitel 5.9).

Symbolische Links

Für symbolische Links enthält `st_size` die Länge des Dateinamens, auf den dieser symbolische Link zeigt.

```
$ ln -s abc slink
$ ls -l slink
lrwxrwxrwx  1 hh      bin           3 Sep 26 18:47 slink -> abc
$
```

In obigen Beispiel hat `slink` 3 Bytes zum Inhalt, nämlich den Namen `abc` (ohne abschließendes `\0`).

Pipes

In SVR4 enthält `st_size` bei Pipes die Anzahl von Bytes, die für das Lesen aus der Pipe verfügbar sind.

5.7.1 truncate und ftruncate – Abschneiden von Dateien

Um Dateien (am Ende) abzuschneiden, stehen die beiden Funktionen ***truncate*** und ***ftruncate*** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

int truncate(const char *pfad, off_t laenge);
int ftruncate(int fd, off_t laenge);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Beide Funktionen »beschneiden« eine Datei auf `laenge` Bytes. Hierbei muß man zwei Fälle unterscheiden:

1. Datei hat mehr als `laenge` Bytes.
In diesem Fall sind die Daten nach `laenge` Bytes nicht mehr Bestandteil der Datei.
2. Datei hat weniger als `laenge` Bytes.
In diesem Fall ist das Verhalten systemabhängig. SVR4 verlängert die Datei auf `laenge` Bytes und erzeugt so ein Loch (siehe unten). Ein Zugriff auf Daten in diesem Loch liefert dabei immer den Wert 0. Bei BSD-Unix hat in diesem Fall der entsprechende *truncate*- bzw. *ftruncate*-Aufruf keine Auswirkung.

Hinweis

Die beiden Funktionen ***truncate*** und ***ftruncate*** sind nicht Bestandteil von POSIX.1 und XPG3.

Das Leeren einer Datei mit dem Flag `O_TRUNC` bei ***open*** ist ein Spezialfall für das Abschneiden einer Datei. Man kann das gleiche auch mit

```
truncate(dateiname, 0);
```

erreichen.

SVR4 bietet bei der Funktion ***fcntl*** das zusätzliche Flag `F_FREESP` an, um einen beliebigen Teil (nicht nur das Ende) aus einer Datei herauszuschneiden.

5.7.2 Löcher in Dateien

Das folgende Programm 5.6 (`lochgen2.c`) erzeugt Löcher in einer Datei, indem es den Schreib-/Lesezeiger eine Million Bytes über das Dateiende hinweg positioniert und dann mit **write** einen Kleinbuchstaben schreibt, so daß in der Datei immer Löcher von einer Million Bytes entstehen. Die Bytes dieser Löcher haben den ASCII-Wert 0.

```
#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(void)
{
    int      fd,
            zeich;

    if ( (fd = creat("datmitloch", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
        fehler_meld(WARNUNG_SYS, ".....kann datmitloch nicht anlegen");

    for (zeich='a' ; zeich<='m' ; zeich++) {
        /*----- Schreib-/Lesezeiger 1 Mio. Bytes weiter setzen-----*/
        if (lseek(fd, 1000000L, SEEK_CUR) == -1)
            fehler_meld(WARNUNG_SYS, "Fehler bei lseek");
        /*----- 1 Zeichen schreiben -----*/
        if (write(fd, &zeich, 1) != 1)
            fehler_meld(WARNUNG_SYS, "Fehler bei write");
    }

    exit(0);
}
```

Programm 5.6 (**lochgen2.c**): Erzeugen einer Datei mit Löchern

Nachdem wir das Programm 5.6 (`lochgen2.c`) kompiliert und gelinkt haben

```
cc -o lochgen2 lochgen2.c fehler.c
```

lassen wir es ablaufen:

```
$ lochgen2
$
```

Wir erhalten dann die sehr große Datei `datmitloch`.

```
$ ls -l datmitloch
-rw-r--r--  1 hh          group   13000013 Jul 11 12:02 datmitloch
$ du -s datmitloch
27 datmitloch
$
```

Wie die Ausgabe von **ls -l** erkennen läßt, ist die Datei `datmitloch` über 13 Millionen Bytes groß, während die Ausgabe von **du -s** für die gleiche Datei nur 27 1024-Byte-Blöcke (27648 Bytes) anzeigt. Hieraus läßt sich schließen, daß die Datei Löcher enthält.

Würden wir uns die Anzahl der Bytes mit **wc -c** zählen lassen, würden wir das gleiche Ergebnis wie bei **ls -l** erhalten, da dieses Kommando mit der Funktion **read** bis ans Dateiende liest.

```
$ wc -c datmitloch
13000013 datmitloch
$
```

Würden wir z.B. mittels **cat** und Ausgabeumlenkung die Datei **datmitloch** duplizieren, so würden in der Kopie die Löcher wirklich mit Nullbytes aufgefüllt, da die auch von **cat** verwendete **read**-Funktion für alle nicht wirklich geschriebenen Bytes den Wert 0 (als Inhalt) liefert.

```
$ cat datmitloch >d2
$ ls -l d*
-rw-r--r--  1 hh      group   13000013 Jul 11 12:13 d2
-rw-r--r--  1 hh      group   13000013 Jul 11 12:02 datmitloch
$ du -s d*
12747      d2
27         datmitloch
$
```

Die Kopie **d2** belegt also wirklich 13052928 Bytes (12747 x 1024). Der Unterschied zwischen dieser Zahl und der Ausgabe von **ls -l** bzw. **wc -c** (13000013) liegt daran, daß bei **du** die wirklich benötigten Bytes gezählt werden, wozu z.B. auch Adreßblöcke gehören, die keine echten Daten, sondern nur Adressen von anderen Blöcken enthalten.

5.8 Zeiten einer Datei

Für jede Datei sind in der Struktur **stat** drei Zeiten vorgesehen, die in Tabelle 5.7 aufgeführt sind:

Komponente	Bedeutung des Inhalts	ls-Option
st_atime	Zeit des letzten Zugriffs (<i>access time</i>)	-u
st_mtime	Zeit der letzten Änderung des Dateiinhalts (<i>modification time</i>)	(default)
st_ctime	Zeit der letzten i-node-Änderung	-c

Tabelle 5.7: Die drei Zeiten, die für jede Datei unterhalten werden.

Das Kommando **ls** gibt bei **-l** immer nur eine der drei Zeiten aus. Genauso sortiert es bei der Option **-t** immer nur nach einer Zeit. Voreingestellt ist in beiden Fällen immer die *modification time* (Zeit der letzten Änderung des Dateiinhalts). Soll bei **-l** oder **-t** eine andere Zeit verwendet werden, so muß entweder **-u** (letzte Zugriffszeit) oder **-c** (letzte i-node-Änderung) angegeben werden.

Die Tabelle 5.8 zeigt, welche Zeiten durch einige der wichtigsten Dateizugriffsfunktionen verändert werden.

Funktion	Datei selbst			Parent-Directory		
	a	m	c	a	m	c
chmod, chown, fchmod, fchown, lchown			x			
mkdir, mkfifo	x	x	x		x	x
open, creat (neue Datei mit O_CREAT)	x	x	x		x	x
open, creat (existierende Datei mit O_TRUNC)		x	x			
pipe	x	x	x			
read	x					
remove (reguläre Datei), unlink, rename, link			x		x	x
remove (Directory), rmdir					x	x
truncate, ftruncate		x	x			
utime	x	x	x			
write		x	x			
a = st_atime						
m = st_mtime						
c = st_ctime						

Tabelle 5.8: Auswirkung einiger wichtiger Funktionen auf die 3 Zeiten einer Datei

In Tabelle 5.8 sind nicht nur die Auswirkungen auf die Zeiten der Datei selbst, sondern auch auf die Zeiten des Parent-Directories aufgeführt, in dem sich die entsprechende Datei befindet. Der Grund dafür liegt in der Tatsache, daß Directories unter Unix auch Dateien sind, die einen speziellen Inhalt haben: Dateinamen mit zugehöriger i-node-Nummer (siehe Kapitel 5.5). Das Hinzufügen oder Löschen von Dateien in diesem Directory hat also immer Auswirkung auf die entsprechenden Zeiten der Directory-Datei.

5.8.1 utime und utimes – Ändern der Zugriffs- und Modifikationszeit

Um die Zugriffszeit (*access time*) und die Zeit der letzten Änderung (*modification time*) explizit zu verändern, steht die Funktion **utime** zur Verfügung.

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *pfad, const struct utimbuf *zeitgr);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Struktur `utimbuf` ist wie folgt definiert:

```
struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};
```

Es gibt keine Möglichkeit, die Zeit der letzten i-node-Änderung (`st_ctime`) direkt zu setzen, denn diese Zeit wird immer dann automatisch gesetzt, wenn die Funktion ***utime*** aufgerufen wird.

Für die beiden Komponenten `actime` und `modtime` ist immer die entsprechende Kalenderzeit (seit 00:00:00 Uhr des 1. Januars 1970 vergangene Sekunden; siehe Kapitel 7.2) anzugeben.

Es sind bei der Funktion ***utime*** zwei Fälle zu unterscheiden:

1. Ist für `zeitzgr` ein NULL-Zeiger angegeben, so werden die beiden Zeiten (*access time* und *modification time*) für die betreffende Datei auf die momentane Zeit gesetzt. Um dies ausführen zu können, muß entweder die effektive User-ID des aufrufenden Prozesses gleich der Eigentümer-ID der entsprechenden Datei sein, oder der aufrufende Prozeß muß Schreibrechte für die entsprechende Datei besitzen.
2. Ist für `zeitzgr` kein NULL-Zeiger angegeben, so werden die beiden Zeiten (*access time* und *modification time*) für die betreffende Datei auf die in `struct utimbuf` angegebenen Zeiten gesetzt. Um dies ausführen zu können, muß entweder die effektive User-ID des aufrufenden Prozesses gleich der Eigentümer-ID der entsprechenden Datei sein oder der aufrufende Prozeß muß mit Superuser-Privilegien ablaufen (Schreibrechte für die entsprechende Datei reichen in diesem Fall nicht aus).

Von BSD-Unix stammt eine weitere Funktion ***utimes*** zum Ändern des Zeitstempels einer Datei, die auch unter Linux verfügbar ist.

```
#include <sys/time.h>
```

```
int utimes(const char *pfad, const struct timeval *zeitzgr);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Funktion ***utimes*** entspricht weitgehend der Funktion ***utime***. Sie unterscheidet sich nur dadurch, daß die neue Zugriffszeit und die neue Zeit der letzten Änderung in der Struktur `struct timeval` übergeben werden:

```
struct timeval {
    long tv_sec; /* access time */
    long tv_usec; /* modification time */
};
```

`tv_sec` enthält dabei die neue Zugriffszeit und `tv_usec` die neue Zeit der letzten Änderung. Ansonsten gilt für ***utimes*** das gleiche wie für ***utime***.

Beispiel

Kopieren einer Datei ohne Verändern der Zeitmarken

Wenn eine Datei mit dem Unix-Kommando ***cp*** kopiert wird, so werden bei der kopierten Datei alle drei Zeiten auf die aktuelle Zeit gesetzt. Wird eine Datei mit dem folgenden Programm 5.7 (`cptime.c`) kopiert, so wird für die kopierte Datei die *access time* und *modification time* der ursprünglichen Datei übernommen.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <utime.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    char            puffer[MAX_ZEICHEN];
    struct stat      statpuff;
    struct utimbuf    zeitpuff;
    FILE            *fz1, *fz2;
    int              n;

    if (argc != 3)
        fehler_meld(FATAL, "usage: %s quelldatei zieldatei", argv[0]);

    /*----- Zeiten von Datei1 ermitteln -----*/
    if (stat(argv[1], &statpuff) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei stat (%s)", argv[1]);

    zeitpuff.actime = statpuff.st_atime;
    zeitpuff.modtime = statpuff.st_mtime;

    /*----- Datei1 nach Datei2 kopieren -----*/
    if ( (fz1 = fopen(argv[1], "r")) == NULL)
        fehler_meld(FATAL_SYS, "kann %s nicht oeffnen", argv[1]);
    if ( (fz2 = fopen(argv[2], "w")) == NULL)
        fehler_meld(FATAL_SYS, "kann %s nicht oeffnen", argv[2]);

    while ( (n=fread(puffer, 1, MAX_ZEICHEN, fz1)) > 0)
        if (fwrite(puffer, 1, n, fz2) != n)
            fehler_meld(FATAL_SYS, "Fehler bei fwrite");
    if (ferror(fz1))
        fehler_meld(FATAL_SYS, "Fehler bei fread");
    fclose(fz1);
    fclose(fz2);

    /*----- Zeiten von Datei1 auch fuer Datei2 eintragen -----*/
    if (utime(argv[2], &zeitpuff) < 0)
```

```

    fehler_meld(WARNUNG_SYS, "Fehler bei utime (%s)", argv[2]);

    exit(0);
}

```

Programm 5.7 (**cptime.c**): Kopieren einer Datei mit Übernahme der Zeitmarken der Originaldatei

Nachdem wir dieses Programm 5.7 (**cptime.c**) kompiliert und gelinkt haben

```
cc -o cptime cptime.c fehler.c
```

wollen wir es testen.

```

$ ls -l lochgen2.c           [Ausgabe der modification time]
-rw-r--r--  1 hh      group      680 Jul 12 15:11 lochgen2.c
$ ls -lu lochgen2.c         [Ausgabe der access time]
-rw-r--r--  1 hh      group      680 Jul 12 17:44 lochgen2.c
$ cp lochgen2.c lochneu.c   [Kopieren von lochgen2.c mit Unix-cp]
$ ls -l loch*.c             [lochneu.c erhielt akt. Zeit als modification time]
-rw-r--r--  1 hh      group      680 Jul 12 15:11 lochgen2.c
-rw-r--r--  1 hh      group      680 Jul 12 17:50 lochneu.c
$ ls -lu loch*.c            [lochgen2.c und lochneu.c erhielten akt. Zeit als access time]
-rw-r--r--  1 hh      group      680 Jul 12 17:50 lochgen2.c
-rw-r--r--  1 hh      group      680 Jul 12 17:50 lochneu.c
$ rm lochneu.c              [Löschen von lochneu.c]
$ cptime lochgen2.c lochneu.c [Kopieren von lochgen2.c mit cptime]
$ ls -l loch*.c             [lochneu.c erhielt modification time von lochgen2.c]
-rw-r--r--  1 hh      group      680 Jul 12 15:11 lochgen2.c
-rw-r--r--  1 hh      group      680 Jul 12 15:11 lochneu.c
$ ls -lu loch*.c            [lochneu.c erhielt ursprgl. access time von lochgen2.c]
-rw-r--r--  1 hh      group      680 Jul 12 17:51 lochgen2.c
-rw-r--r--  1 hh      group      680 Jul 12 17:50 lochneu.c
$ ls -lc loch*.c            [Durch utime wurde i-node-Änderung für lochneu.c bewirkt]
-rw-r--r--  1 hh      group      680 Jul 12 15:11 lochgen2.c
-rw-r--r--  1 hh      group      680 Jul 12 17:51 lochneu.c
$ rm lochneu.c
$

```

Hinweis

Die Funktion **utime** wird üblicherweise vom Kommando **touch** und den beiden Archivierungskommandos **tar** und **cpio** verwendet.

5.9 Directories

In diesem Kapitel werden Funktionen vorgestellt, die Aktionen auf Directories ermöglichen, wie z.B. Anlegen von neuen Directories, Löschen von Directories, Lesen der Dateinamen in Directories, Wechseln in andere Directories usw. Zunächst wird die Bedeutung der einzelnen Zugriffsrechtebits für Directories behandelt.

5.9.1 Zugriffsrechte für Directories

Die Tabelle 5.9 stellt die Bedeutung der einzelnen Zugriffsrechtebits bei Dateien und Directories einander gegenüber.

Konstante	Bedeutung	bei regulären Dateien bei Directories
S_IRUSR	<i>user-read</i>	Leserecht für Dateieigentümer Eigentümer darf Directory-Einträge lesen (z.B. mit ls)
S_IWUSR	<i>user-write</i>	Schreibrecht für Dateieigentümer Eigentümer darf Dateien im Directory anlegen oder löschen
S_IXUSR	<i>user-execute</i>	Ausführrecht für Dateieigentümer Eigentümer darf im Directory nach Einträge suchen (cd ist mögl.)
S_IRGRP	<i>group-read</i>	Leserecht für Gruppe des Dateieigentümers Gruppenmitglieder dürfen Directory-Einträge lesen (z.B. mit ls)
S_IWGRP	<i>group-write</i>	Schreibrecht für Gruppe des Dateieigentümers Gruppenmitglieder dürfen Dateien im Directory anlegen / löschen
S_IXGRP	<i>group-execute</i>	Ausführrecht für Gruppe des Dateieigentümers Gruppenmitglieder dürfen im Directory Einträge suchen (cd ist mögl.)
S_IROTH	<i>other-read</i>	Leserecht für alle anderen Benutzer Alle anderen dürfen Directory-Einträge lesen (z.B. mit ls)
S_IWOTH	<i>other-write</i>	Schreibrecht für alle anderen Benutzer Alle anderen dürfen Dateien im Directory anlegen oder löschen
S_IXOTH	<i>other-execute</i>	Ausführrecht für alle anderen Benutzer Alle anderen dürfen im Directory Einträge suchen (cd ist mögl.)
S_ISUID	<i>Set-User-ID</i>	effektive User-ID bei Ausführung auf User-ID des Dateieigentümers setzen keine Bedeutung
S_ISGID	<i>Set-Group-ID</i>	wenn group-execute gesetzt, dann wird effektive Group-ID bei Ausführung für Group-ID der Datei gesetzt; sonst wird record locking eingeschaltet. Group-ID von neuen Dateien im Directory wird immer auf Group-ID des Directories gesetzt
S_ISVTX	<i>sticky bit</i>	Textsegment des Programms verbleibt nach Ausführung im swap-Bereich eingeschränkte Rechte zum Neuanlegen und Löschen von Dateien des Directories

Tabelle 5.9: Bedeutung der Zugriffsrechtebits bei Dateien und Directories (aus `<sys/stat.h>`)

Daneben sind noch die Konstanten `S_IRWXU`, `S_IRWXG` und `S_IRWXO` definiert:

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR  
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP  
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

5.9.2 mkdir – Anlegen eines neuen Directorys

Um ein neues Directory anzulegen, steht die Funktion **mkdir** zur Verfügung.

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int mkdir(const char *pfad, mode_t modus);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Funktion **mkdir** legt ein neues leeres Directory mit dem Namen `pfad` an, wobei in diesem Directory automatisch die beiden Dateien (Links) `.` (für Working-Directory) und `..` (für Parent-Directory) angelegt werden.

Die Zugriffsrechte für das Directory werden über `modus` festgelegt. Es ist zu beachten, daß dieses Zugriffsrechtmuster noch durch die Dateikreierungsmaske modifiziert wird (siehe Kapitel 5.3).

Die User-ID und Group-ID des neuen Directorys wird dabei durch die in Kapitel 5.3 beschriebenen Regeln festgelegt.

Hinweis

Ist in SVR4 für das Parent-Directory das *Set-Group-ID*-Bit gesetzt, so wird auch für das neu angelegte Directory automatisch das *Set-Group-ID*-Bit gesetzt, so daß bei Dateien, die in diesem neuen Directory angelegt werden, auch automatisch das *Set-Group-ID*-Bit gesetzt wird.

In BSD-Unix erben immer alle in einem Directory neu angelegten Dateien und Directorys die *Group-ID* des Parent-Directorys.

Man sollte darauf achten, daß bei einem **mkdir**-Aufruf im `modus`-Argument immer die entsprechenden execute-Bits gesetzt sind, um einen Zugriff auf die Dateien des neuen Directorys zu ermöglichen.

5.9.3 rmdir – Löschen eines leeren Directorys

Um ein leeres Directory zu löschen, steht die Funktion **rmdir** zur Verfügung.

```
#include <unistd.h>
```

```
int rmdir(const char *pfad);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Das zu löschende Directory *pfad* muß leer sein, was bedeutet, daß es nur die beiden Einträge *.* und *..* enthalten darf. Nur wenn der Link-Zähler (im i-node) des betreffenden Directorys 0 wird und kein anderer Prozeß dieses Directory gerade geöffnet hat, wird auch der physikalische Speicherplatz freigegeben, der von der Directory-Datei belegt wird.

Hinweis

Wenn andere Prozesse noch ein Directory geöffnet haben, und der Link-Zähler 0 wird, so bewirkt der **rmdir**-Aufruf das Löschen des Directory-Links und der beiden in diesem Directory enthaltenen Links *.* (Working-Directory) und *..* (Parent-Directory). Dadurch ist es nicht mehr möglich, neue Dateien in diesem Directory anzulegen, obwohl der durch dieses Directory belegte physikalische Speicherplatz erst dann freigegeben wird, wenn der letzte Prozeß dieses Directory schließt.

5.9.4 chdir und fchdir – Wechseln in ein neues Directory

Mit den beiden Funktionen **chdir** und **fchdir** kann ein Prozeß in ein neues Directory wechseln.

```
#include <unistd.h>
```

```
int chdir(const char *pfad);
```

```
int fchdir(int fd);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Jeder Prozeß hat zu einem Zeitpunkt ein aktuelles Working-Directory. Dieses kann er durch den Aufruf von **chdir** (unter Angabe eines relativen oder absoluten Pfadnamens) oder von **fchdir** (unter Angabe eines Filedesktors) wechseln.

Hinweis

fchdir wird zwar von SVR4 und 4.4BSD angeboten, ist aber nicht Bestandteil von POSIX.1.

Mit **chdir** und **fchdir** kann immer nur das Working-Directory des Prozesses gewechselt werden, der eine dieser beiden Routinen aufruft. Endet der entsprechende Prozeß, so wird immer wieder automatisch in das Working-Directory des Elternprozesses gewechselt. Dies ist im übrigen auch der Grund, warum es sich beim Kommando **cd** nicht um ein eigenständiges Programm handeln darf, sondern es ein Builtin-Kommando der Shell sein muß.

Beispiel*Demonstrationsprogramm zur Funktion **chdir***

Das folgende Programm 5.8 (`mchdir.c`) wechselt in das Directory, das auf der Kommandozeile angegeben wird.

```
#include "eighdr.h"

int
main(int argc, char*argv[])
{
    if (argc != 2)
        fehler_meld(FATAL, "usage: %s directory", argv[0]);

    if (chdir(argv[1]) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei chdir(%s)", argv[1]);

    printf("--- Neues working directory: %s ---\n", argv[1]);

    exit(0);
}
```

*Programm 5.8 (**mchdir.c**): Beispiel zur Funktion **chdir***

Nachdem wir das Programm 5.8 (`mchdir.c`) kompiliert und gelinkt haben

```
cc -o mchdir mchdir.c fehler.c
```

wollen wir es testen:

```
$ pwd
/home/hh
$ mchdir /usr      [Wechseln in das directory /usr; nur für Dauer der Programmausführung]
--- Neues working directory: /usr ---
$ pwd              [Nach Rückkehr aus Programm (Prozeß) befindet man sich wieder im ursprgl. work. dir.]
/home/hh
$
```

5.9.5 getcwd – Erfragen des Working-Directory-Pfadnamens

Um den momentanen Pfadnamen des Working-Directorys zu ermitteln, steht die Funktion **getcwd** zur Verfügung.

```
#include <unistd.h>
```

```
char *getcwd(char *puffer, size_t puffgroesse);
```

gibt zurück: puffer (bei Erfolg); NULL bei Fehler

getcwd schreibt an die Speicheradresse `puffer` den Pfadnamen des Working-Directorys (einschließlich des abschließenden `\0`). Die Größe des Puffers wird **getcwd** über das Argument `puffgroesse` mitgeteilt.

Hinweis

Manche Unix-Systeme erlauben die Angabe von `NULL` für das erste Argument `puffer`. In diesem Fall allokiert **getcwd** selbst mittels **malloc**(`puffgroesse`) den benötigten Speicherplatz für den Pfadnamen. Dies ist jedoch nicht Bestandteil von POSIX.1 oder XPG3, weshalb davon auch abzuraten ist.

Beispiel

*Demonstrationsprogramm zur Funktion **getcwd***

Das folgende Programm 5.9 (`getcwd.c`) wechselt in das als erstes Argument angegebene Directory und gibt dort dann mittels eines **getcwd**-Aufrufs das neue Working-Directory aus.

```
#include "eighdr.h"

#define MAX_PFAD 500
int
main(int argc, char*argv[])
{
    char    pfadname[MAX_PFAD];

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s directory", argv[0]);

    if (chdir(argv[1]) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei chdir(%s)", argv[1]);

    if (getcwd(pfadname, MAX_PFAD) == NULL)
        fehler_meld(FATAL_SYS, "Fehler bei getcwd");

    printf("--- Neues working directory: %s ---\n", pfadname);

    exit(0);
}
```

*Programm 5.9 (**getcwd.c**): Beispiel zur Funktion **getcwd***

Nachdem wir das Programm 5.9 (`getcwd.c`) kompiliert und gelinkt haben

```
cc -o getcwd getcwd.c fehler.c
```

wollen wir es testen:

```
$ pwd
/home/hh
$ getcwd /usr
--- Neues working directory: /usr ---
```



```
$ pwd
/home/hh
$
```

Wechselt man in ein Directory, das ein symbolischer Link auf ein anderes Directory ist, so wird immer in das Directory gewechselt, auf das der symbolische Link zeigt.

```
$ ls -l /usr/spool
lrwxrwxrwx 1 root      bin      .... /usr/spool -> ../var/spool
$ getcwd /usr/spool
--- Neues working directory: /var/spool ---
$
```

5.9.6 struct dirent – Aufbau eines Eintrags in einer Directory-Datei

Das Format der Einträge in einer Directory-Datei hängt vom jeweiligen Unix-System ab. In früheren Unix-Versionen wurde für jede Datei eines Directories 16 Bytes in der Directory-Datei hinterlegt, wobei die ersten beiden Bytes die i-node-Nummer und die restlichen 14 Bytes den Namen der Datei enthielten.

Neuere Unix-Systeme lassen nun aber variabel lange Dateinamen (nicht mehr auf 14 Bytes begrenzt) zu. Um nun Programme schreiben zu können, die systemunabhängig sind, schreibt POSIX.1 die Struktur `dirent` vor, die in `<dirent.h>` definiert sein muß. In SVR4 und BSD-Unix sind in dieser Struktur mindestens die beiden folgenden Komponenten enthalten:

```
struct dirent {
    ino_t    d_ino;           /* i-node-Nr (nicht in POSIX.1) */
    char     d_name[NAME_MAX + 1]; /* Dateiname (mit abschl. \0) */
};
```

Unter BSD-Unix ist die Konstante `NAME_MAX` meist mit dem Wert 255 definiert. Da in BSD-Unix aber jeder Dateiname in einer Directory-Datei sowieso mit `\0` abgeschlossen ist, ist der Wert von `NAME_MAX` nicht von Interesse.

In SVR4 ist `NAME_MAX` nicht standardgemäß definiert, da diese Konstante vom Filesystem abhängig ist, in dem sich das betreffende Directory befindet. Deswegen erhält man den Wert von `NAME_MAX` dort üblicherweise mit der Funktion ***fpathconf***.

5.9.7 opendir, readdir, rewinddir und closedir – Lesen von Directories

Der Inhalt einer Directory-Datei darf von jedermann gelesen werden, der die entsprechenden Zugriffsrechte auf diese Directory-Datei hat. Das explizite Beschreiben einer Directory-Datei (z.B. mittels ***write***) ist jedoch nur dem Kern gestattet, um zu verhindern, daß das ganze Filesystem korumpiert wird.

Um neue Dateien in einem Directory (z.B. mittels **fopen** oder **mkdir**) anzulegen oder (mittels **remove**, **unlink** oder **rmdir**) zu löschen, muß man für das betreffende Directory Schreib- und Execute-Rechte besitzen, was – wie bereits oben erwähnt – nicht bedeutet, daß man direkt (z.B. mittels **write**) in die Directory-Datei schreiben kann.

Um eine einheitliche Schnittstelle für das Lesen der doch sehr systemabhängigen Directory-Formate zu erhalten, schreibt POSIX.1 die folgenden vier Funktionen **opendir**, **readdir**, **rewinddir** und **closedir** vor.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *pfd);
                                     gibt zurück: DIR-Zeiger (bei Erfolg); NULL bei Fehler

struct dirent *readdir(DIR *zgr);
                                     gibt zurück: struct dirent-Zeiger (bei Erfolg); NULL bei Fehler

void rewinddir(DIR *zgr);

int closedir(DIR *zgr);
                                     gibt zurück: 0 (bei Erfolg); -1 bei Fehler
```

Die Struktur **DIR** ist eine interne Struktur, die von diesen vier Funktionen benutzt wird, um Informationen über das zu lesende Directory zu erhalten und untereinander auszutauschen.

Der von der Funktion **opendir** zurückgegebene Zeiger auf die Struktur **DIR** wird von den anderen drei Funktionen benutzt, um den Inhalt eines Directorys schrittweise zu lesen (**readdir**), den »Lesezeiger« im Directory wieder auf den Anfang der Namensliste zu stellen (**rewinddir**) oder aber die Directory-Datei zu schließen (**closedir**) und damit den Lesevorgang in diesem Directory zu beenden.

Hinweis

Nach einem **opendir** wird mit dem ersten **readdir** der erste Eintrag aus der Directory-Datei gelesen. Jedes weitere **readdir** liest dann immer den nächsten Eintrag.

Die Reihenfolge, in der die Einträge in einem Directory von **readdir** gelesen werden, ist implementierungsabhängig und muß nicht alphabetisch sein.

System V bietet eine eigene Systemfunktion **ftw** (*file transfer walk*) an, die einen Directory-Baum rekursiv durchläuft und für jede Datei des Directory-Baums eine Funktion aufruft, die der Benutzer selbst definieren muß. Die Funktion **ftw** hat jedoch die Eigenheit, daß sie für jede gefundene Datei die Funktion **stat** aufruft, was dazu führt, daß sie symbolischen Links folgt (siehe auch Beispiel unten). Da dies nicht in allen Anwendungsfällen erwünscht ist, wird seit SVR4 eine weitere Funktion **nftw** (*new file transfer walk*) angeboten, die eine eigene Option besitzt, mit der der Aufrufer festlegen kann, ob symbolischen Links zu folgen ist oder nicht.

Beispiel*Ausgeben einer Directory-Hierarchie in Baumform (mit eigenen Funktionen)*

```

#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include <string.h>
#include "eighdr.h"

/*---- Konstantendefinitionen -----*/
#define FTW_F 1 /* Datei ist kein Directory */
#define FTW_D 2 /* Datei ist ein Directory */
#define FTW_DNR 3 /* Nichtlesbares Directory */
#define FTW_NS 4 /* Datei, auf die stat erfolglos ist */
#define MAX_PFAD 1000

/*---- Typdefinitionen -----*/
typedef int MEIN_AUSWERT(const char *, const struct stat *, int);

/*---- Variablendefinitionen -----*/
static char pfadname[MAX_PFAD];
static int tiefe = 0;
static long int dateizahl = 0;

/*---- Forward-Funktionsdeklarationen -----*/
static MEIN_AUSWERT mein_auswert;
static int mein_ftw(char *, MEIN_AUSWERT *);
static int pfad_behandel(MEIN_AUSWERT *);
/*---- main -----*/
int
main(int argc, char *argv[])
{
    if (argc != 2)
        fehler_meld(FATAL, "usage: %s directory", argv[0]);

    exit( mein_ftw(argv[1], mein_auswert) );
}

/*---- mein_ftw -----*/
static int
mein_ftw(char *pfad, MEIN_AUSWERT *funktion)
{
    int n;

    if (chdir(pfad) < 0) /* In angegebenen Pfad wechseln */
        fehler_meld(FATAL_SYS, "kann nicht zu %s wechseln", pfad);

    if (getcwd(pfadname, MAX_PFAD) == NULL) /* Absoluten Pfadnamen ermitteln */
        fehler_meld(FATAL_SYS, "fehler bei getcwd fuer %s", pfad);

    n = pfad_behandel(funktion);

```

```

    printf("\n=== %ld Datei(en) ===\n", dateizahl);
    return(n);
}

/*---- pfad_behandel -----*/
static int
pfad_behandel(MEIN_AUSWERT *funktion)
{
    struct stat    statpuff;
    struct dirent  *direntz;
    DIR            *dirz;
    int            n;
    char           *zgr;

    if (lstat(pfadname, &statpuff) < 0)
        return(funktion(pfadname, &statpuff, FTW_NS)); /* Fehler bei stat */

    if (S_ISDIR(statpuff.st_mode) == 0)
        return(funktion(pfadname, &statpuff, FTW_F)); /* kein Directory */

    /* Es liegt ein Directory vor, fuer das zuerst funktion()
     * aufgerufen wird, bevor jeder einzelne Dateiname dieses Directorys
     * bearbeitet wird.                                     */

    if ( (dirz = opendir(pfadname)) == NULL) { /* Directory nicht lesbar */
        closedir(dirz);
        return(funktion(pfadname, &statpuff, FTW_DNR));
    }

    if ( (n = funktion(pfadname, &statpuff, FTW_D)) != 0) /*Ausg.:Directorypfad*/
        return(n);

    zgr = pfadname + strlen(pfadname); /* Slash an Pfadnamen anhaengen */
    *zgr++ = '/';
    *zgr = '\0';

    while ( (direntz = readdir(dirz)) != NULL) {
        /* . und .. ignorieren */
        if (strcmp(direntz->d_name, ".") && strcmp(direntz->d_name, "..")) {
            strcpy(zgr, direntz->d_name); /* Dateinamen nach Slash anhaengen */
            tiefe++;
            if (pfad_behandel(funktion) != 0) { /* Rekursion */
                tiefe--;
                break;
            }
            tiefe--;
        }
    }

    *(zgr-1) = '\0'; /* Nach Slash alles wieder loeschen */

    if (closedir(dirz) < 0)
        fehler_meld(WARNUNG, "closedir fuer %s schlug fehl", pfadname);
}

```

```

    return(n);
}

/*---- mein_auswert -----*/
static int
mein_auswert(const char *pfad, const struct stat *statzgr, int dateityp)
{
    static bool erstmal=TRUE;
    int i;

    dateizahl++;
    if (!erstmal) {
        for (i=1 ; i<=tiefe ; i++)
            printf("%4c|", ' ');
        printf("----%s", strrchr(pfad, '/')+1);
    } else {
        printf("%s", pfad);
        erstmal = FALSE;
    }

    switch (dateityp) {
        case FTW_F:
            switch (statzgr->st_mode & S_IFMT) {
                case S_IFREG:          break;
                case S_IFCHR:   printf(" c"); break;
                case S_IFBLK:   printf(" b"); break;
                case S_IFIFO:   printf(" f"); break;
                case S_IFLNK:   printf("@"); break;
                case S_IFSOCK:  printf(" s"); break;
                default:        printf(" ?"); break;
            }
            printf("\n");
            break;

        case FTW_D:
            printf("/\n");
            break;

        case FTW_DNR:
            printf("/-\n");
            break;

        case FTW_NS:
            fehler_meld(WARNUNG_SYS, "Fehler bei stat auf Datei %s", pfad);
            break;

        default:
            fehler_meld(FATAL_SYS, "Unbekannter Dateityp (%d) bei Datei %s", dateityp, pfad);
            break;
    }

    return(0);
}

```

Programm 5.10 (tree.c): Ausgabe einer Directory-Hierarchie in Baumform (mit eigenen Funktionen)

Nachdem wir das Programm 5.10 (`tree.c`) kompiliert und gelinkt haben

```
cc -o tree tree.c fehler.c
```

wollen wir es testen:

```
$ tree /usr/include
/usr/include/
|----X11@
|----assert.h
|----arpa/
|    |----ftp.h
|    |----inet.h
|    |----nameser.h
|    |----telnet.h
|    |----tftp.h
|----gnu/
|    |----types.h
|----nan.h
.....
|----bsd/
|    |----bsd.h
|    |----curses.h
|    |----errno.h
|    |----sgtty.h
|    |----signal.h
|    |----stdlib.h
|    |----sys/
|    |    |----ttychars.h
|    |----tzfile.h
|    |----unistd.h
|    |----utmp.h
.....
|----asm@
|----vga.h
|----vgagl.h
|----vgamouse.h
|----vgakeyboard.h
|----olgx@
|----pixrect@
|----xview@
|----sspkg@
|----uit@
==== 292 Datei(en) ====
$
```

Wie an der Ausgabe zu erkennen ist, werden nicht einfache Dateien bei der Ausgabe durch Anhängen eines Sonderzeichens gekennzeichnet, wie z.B. `@` für symbolische Links.

Beispiel*Ausgeben einer Directoryhierarchie in Baumform (mit Funktion ftw)*

```

#include <sys/types.h>
#include <sys/stat.h>
#include <ftw.h>
#include <dirent.h>
#include <limits.h>
#include <string.h>
#include "eighdr.h"

/*---- Typdefinitionen -----*/
typedef int MEIN_AUSWERT(const char *, struct stat *, int);

/*---- Variablendefinitionen -----*/
static long int dateizahl = 0;

/*---- Forward-Funktionsdeklarationen -----*/
static MEIN_AUSWERT mein_auswert;

/*---- main -----*/
int
main(int argc, char *argv[])
{
    if (argc != 2)
        fehler_meld(FATAL, "usage: %s directory", argv[0]);

    if ( ftw(argv[1], mein_auswert, 10) == 0 ) {
        printf("\n=== %ld Datei(en) ===\n", dateizahl);
        exit(0);
    } else {
        fehler_meld(FATAL_SYS, "Fehler bei ftw");
    }
}

/*---- dir_tiefe -----*/
static int
dir_tiefe(const char *pfad)
{
    int z=0;
    char *zgr = (char *)pfad;

    while (zgr=strchr(zgr, '/')) {
        zgr++;
        z++;
    }
    return(z);
}

/*---- mein_auswert -----*/
static int
mein_auswert(const char *pfad, struct stat *statzgr, int dateityp)

```

```

{
    static bool erstmal=TRUE;
    static int  ausgangs_tiefe;
    int i;

    dateizahl++;
    if (!erstmal) {
        for (i=1 ; i<=dir_tiefe(pfad)-ausgangs_tiefe ; i++)
            printf("%4c|", ' ');
        printf("----%s", strrchr(pfad, '/')+1);
    } else {
        ausgangs_tiefe = dir_tiefe(pfad);
        printf("%s", pfad);
        erstmal = FALSE;
    }

    switch (dateityp) {
        case FTW_F:
            switch (statzgr->st_mode & S_IFMT) {
                case S_IFREG:      break;
                case S_IFCHR:      printf(" c"); break;
                case S_IFBLK:      printf(" b"); break;
                case S_IFIFO:      printf(" f"); break;
                case S_IFLNK:      printf("@"); break;
                case S_IFSOCK:     printf(" s"); break;
                default:           printf(" ?"); break;
            }
            printf("\n");
            break;

        case FTW_D:
            printf("/\n");
            break;

        case FTW_DNR:
            printf("/-\n");
            break;

        case FTW_NS:
            fehler_meld(WARNUNG_SYS, "Fehler bei stat auf Datei %s", pfad);
            break;

        default:
            fehler_meld(FATAL_SYS, "Unbekannter Dateityp (%d) bei Datei %s", dateityp, pfad);
            break;
    }

    return(0);
}

```

Programm 5.11 (tree2.c): Ausgabe einer Directory-Hierarchie in Baumform (mit Funktion **ftw**)

Nachdem wir dieses Programm 5.11 (tree2.c) kompiliert und gelinkt haben

```
cc -o tree2 tree2.c fehler.c
```

wollen wir es testen:

```
$ tree2 /usr/include
/usr/include/
|----X11/
|   |----xpm.h
|   :
|   :
|   |----StringDefs.h
|   |----Vendor.h
|   |----VendorP.h
|   |----Xmu/
|   |   |----Xmu.h
|   |   |----Atoms.h
|   :
|   :
|   |----WidgetNode.h
|   |----WinUtil.h
|   |----Xct.h
.....
.....
.....
.....

==== 844 Datei(en) ====
$
```

Für das gleiche Directory erhalten wir hier also einen wesentlich umfangreicheren Baum, was darin liegt, daß **ftw** symbolischen Links folgt.

5.10 Gerätedateien

Jedem Dateisystem sind unter Unix zwei Zahlenwerte zugeordnet: eine *Major Device Number* und eine *Minor Device Number*. Für diese beiden Nummern existiert ein eigener primitiver Systemdatentyp `dev_t`.

Um aus diesem Datentyp `dev_t` die beiden Nummern zu extrahieren, stehen üblicherweise die beiden Makros **major** und **minor** zur Verfügung, so daß man sich nicht um die interne Darstellung dieser beiden Zahlen kümmern muß.

In der Struktur `stat` sind die zwei Komponenten `st_dev` und `st_rdev` enthalten:

```
st_dev
```

enthält für jeden Dateinamen die Gerätenummer des Filesystems, in dem sich diese Datei und ihr zugehöriger i-node befindet.

st_rdev

hat nur für zeichen- und blockorientierte Gerätedateien einen definierten Wert, nämlich die Gerätenummer des zugeordneten Geräts. Die *major number* legt dabei den Gerätetyp fest, während die *minor number*, die dem entsprechenden Gerätetreiber übergeben wird, zur Unterscheidung von verschiedenen Geräten des gleichen Typs dient.

Beispiel

Ausgeben der Nummern von Gerätedateien

Das Programm 5.12 (`devnr.c`) gibt für jeden auf der Kommandozeile angegebenen Dateinamen dessen Gerätenummer aus. Handelt es sich dabei um eine zeichen- oder blockorientierte Datei, so gibt es zusätzlich noch die Gerätenummer des zugeordneten Geräts aus.

```
#include <sys/sysmacros.h> /* fuer Makros minor/minor; in BSD:<sys/types.h> */
#include <sys/stat.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    struct stat    statpuff;
    int            i;

    for (i=1 ; i<argc ; i++) {
        printf("%20s: ", argv[i]);

        if (lstat(argv[i], &statpuff) < 0)
            fehler_meld(WARNUNG_SYS, "Fehler bei lstat (%s)", argv[i]);
        else {
            printf("dev = %2d/%2d",
                major(statpuff.st_dev), minor(statpuff.st_dev));
            if (S_ISCHR(statpuff.st_mode) || S_ISBLK(statpuff.st_mode) ) {
                printf("; rdev = %2d/%2d (%s",
                    major(statpuff.st_rdev), minor(statpuff.st_rdev),
                    (S_ISCHR(statpuff.st_mode)) ? "zeichen" : "block");
                printf("orient.)");
            }
        }
        printf("\n");
    }

    exit(0);
}
```

Programm 5.12 (`devnr.c`): Ausgabe der Gerätenummern (`st_dev` und `st_rdev`) von Dateien

Nachdem wir das Programm 5.12 (`devnr.c`) kompiliert und gelinkt haben

```
cc -o devnr devnr.c fehler.c
```

wollen wir es testen:

```
$ devnr / /home/hh /c/windows /a /dev/tty1 /dev/fd0
      /: dev = 8/ 3
      /home/hh: dev = 8/ 3
      /c/windows: dev = 8/ 1
      /a: dev = 2/ 0
      /dev/tty1: dev = 8/ 3; rdev = 4/ 1 (zeichenorient.)
      /dev/fd0: dev = 8/ 3; rdev = 2/ 0 (blockorient.)

$ mount      [Ausgabe, welche Directories an welche Gerätedatei montiert sind]
/dev/sda3 on / ...
/dev/sda1 on /c type msdos
none on /proc type proc (rw)
/dev/fd0 on /a type msdos
$
```

An der obigen Ausgabe kann man erkennen, daß sich die Dateien `/`, `/home/hh`, `/dev/tty1` und `/dev/fd0` im gleichen Filesystem auf einer Plattenpartition befinden. Dagegen befinden sich die beiden Directories `/c/windows` und `/a` auf einer anderen Partition. Während die Gerätedatei `/dev/fd0` (Diskettenlaufwerk) blockorientiert ist, ist die Gerätedatei `/dev/tty1` (für ein Terminal) zeichenorientiert.

Hinweis

SVR4 verwendet 32 Bit für den Datentyp `dev_t`: 14 für die *Major Number* und 18 für die *Minor Number*.

BSD-Unix verwendet 16 Bit für den Datentyp `dev_t`: 8 für die *Major Number* und 8 für die *Minor Number*.

In welcher Headerdatei die beiden Makros ***major*** und ***minor*** definiert sind, ist systemabhängig.

5.11 Der Puffercache

Die meisten Unix-Systeme unterhalten im Kern einen Puffercache, über den die E/A-Aktionen (wie Schreiben) durchgeführt werden, bevor sie wirklich physikalisch (auf Festplatte, Diskette usw.) stattfinden. Wenn man z.B. mittels ***write*** Daten in eine Datei schreibt, so findet das physikalische Schreiben nicht sofort statt, sondern die betreffenden Daten werden vom Kern zunächst in einen seiner Puffer kopiert. Das wirkliche Schreiben (vom Puffer auf das physikalische Gerät) findet erst später statt, z.B. wenn der Kern den Puffer für andere zu schreibende Daten benötigt. Dieser Vorgang wird mit *delayed write* bezeichnet.

Um in jedem Fall ein konsistentes Filesystem zu gewährleisten, auch wenn keine weiteren Daten zu schreiben sind, stehen die beiden Funktionen ***sync*** und ***fsync*** zur Verfügung.

5.11.1 sync und fsync – Schreiben des Puffercaches

Um das wirkliche Schreiben des Puffercache-Inhalts auf das entsprechende physikalische Speichermedium zu veranlassen, stehen die beiden Funktionen **sync** und **fsync** zur Verfügung.

```
#include <unistd.h>
```

```
void sync(void);
```

```
int fsync(int fd);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

sync

Die Funktion **sync** veranlaßt das physikalische Schreiben aller noch im Puffercache stehenden Daten, indem sie sie in eine entsprechende Warteschlange einreiht, und dann sofort zum Aufrufer zurückkehrt, ohne auf die Beendigung des physikalischen Schreibvorgangs zu warten. **sync** wird üblicherweise alle 30 Sekunden von einem System-Dämonprozeß (meist **update** genannt) aufgerufen, um die Konsistenz des Filesystems zu gewährleisten. Das Unix-Kommando **sync** bedient sich im übrigen auch dieser Funktion.

fsync

Die Funktion **fsync** bezieht sich nur auf eine Datei, deren Filedeskriptor beim Aufruf anzugeben ist. Sie veranlaßt das physikalische Schreiben aller noch im Puffercache stehenden Daten dieser Datei, und wartet – im Gegensatz zu **sync** – auf die Beendigung des physikalischen Schreibvorgangs, bevor sie zum Aufrufer zurückkehrt.

Hinweis

Wird beim Öffnen einer Datei (siehe Kapitel 4.2) oder auch später (siehe Funktion **fcntl** in Kapitel 4.9) das Flag **O_SYNC** gesetzt, so wird bei jedem Schreiben auf die Beendigung des physikalischen Schreibvorgangs gewartet, während bei der Funktion **fsync** nur immer zum Zeitpunkt des Aufrufs der entsprechende Puffer physikalisch geschrieben wird.

Während **fsync** Bestandteil von XPG3 und XPG4 ist, ist weder **sync** noch **fsync** Bestandteil von POSIX.1. Beide Funktionen werden aber sowohl von SVR4 als auch von BSD-Unix angeboten.

5.12 Realisierung von Filesystemen unter Linux



Wie unter Unix, so werden auch unter Linux die internen Strukturen der einzelnen Filesysteme vom Virtual File System (VFS) verwaltet (siehe auch Abb. 5.2).

Das VFS ruft die für die jeweiligen Filesysteme speziell konzipierten Funktionen auf, um diese internen Strukturen zu füllen.

Um die von einem konkreten Filesystem zur Verfügung gestellten Funktionen dem VFS bekannt zu machen, muß die Funktion **register_filesystem** aufgerufen werden, wie dies nachfolgend als Beispiel für das ext2-Filesystem gezeigt ist:

```
static struct file_system_type ext2_fs_type = {
    ext2_read_super, "ext2", 1, NULL
};

int init_ext2_fs(void)
{
    return register_filesystem(&ext2_fs_type);
}
```

Das VFS erhält somit als erstes Argument die sogenannte Mount-Schnittstelle (**ext2_read_super**), den Namen des Filesystems (**ext2**) und ein Flag, das anzeigt, ob ein Gerät zum Mounten unbedingt notwendig ist (in diesem Fall: 1=ja). Durch einen solchen **register_filesystem**-Aufruf werden die weiteren filesystemspezifischen Funktionen dem VFS bekannt gemacht.

Die an **register_filesystem** übergebene Variable (Adresse) hat als Datentyp die Struktur **file_system_type**, die wie folgt in `<linux/fs.h>` deklariert ist:

```
struct file_system_type {
    struct super_block *(*read_super)(struct super_block *, void *, int);
    const char *name;
    int requires_dev;
    struct file_system_type * next;
};
```

Die Funktion **register_filesystem** fügt die übergebene Strukturvariable (Adresse) an das Ende einer einfach verketteten Liste ein. Auf den Anfang dieser Liste zeigt immer ein Zeiger mit dem Namen **file_systems**.

In früheren Linux-Kernen (vor Version 1.1.8) wurden die Strukturen noch in einem statischen Array gehalten, da damals noch alle Filesysteme zum Zeitpunkt der Kern-Kompilierung eingebunden wurden. Mit der Einführung von Modulen mußte man auf eine verkettete Liste umstellen, um nun auch zur Laufzeit nachträglich Filesysteme einbinden zu können.

Nach der erfolgreichen Registrierung eines spezifischen Filesystems beim VFS, können Filesysteme dieses Typs verwaltet werden.

5.12.1 Mounten von Filesystemen

Um überhaupt auf die einzelnen Dateien eines Filesystems zugreifen zu können, muß dieses Filesystem zuerst einmal gemountet (montiert) werden. Dies erfolgt entweder mit der Funktion **mount_root** oder dem Systemaufruf **mount**.

Mounten des Root-Filesystems mit mount_root

Die Funktion `mount_root`, die für das Mounten des ersten Filesystems (dem Root-Filesystem) zuständig ist, wird vom Systemaufruf **setup** nach der Registrierung aller im Kern fest eingebundenen Filesystemen aufgerufen. Die Funktion **setup**, die in der Datei `fs/filesystems.c` definiert ist, ist z.B. wie folgt implementiert:

```
asmlinkage int sys_setup(void)
{
    static int callable = 1;

    if (!callable)
        return -1;
    callable = 0;
    device_setup();
    binfmt_setup();
#ifdef CONFIG_EXT_FS
    init_ext_fs();
#endif
#ifdef CONFIG_EXT2_FS
    init_ext2_fs();
#endif
#ifdef CONFIG_XIA_FS
    init_xiafs_fs();
#endif
#ifdef CONFIG_MINIX_FS
    init_minix_fs();
#endif
    .....
    .....
    mount_root();
    return 0;
}
```

Um zu verhindern, daß **setup** mehr als einmal aufgerufen wird, wird die lokale statische Variable `callable` verwendet.

setup initialisiert zunächst die Gerätetreiber für die vorhandenen Festplatten (mit **device_setup**) und registriert dann die bei der Konfiguration des Kerns angegebenen Binärformate (mit **binfmt_setup**) und Filesysteme (mit den entsprechenden **init_...**-Routinen). Danach wird mit **mount_root** das Root-Filesystem eingerichtet.

Der Systemaufruf **setup** wird im übrigen gleich nach dem Erzeugen des Init-Prozesses in der Kernfunktion **init** (befindet sich in `init/main.c`) genau einmal aufgerufen.

Dieser Systemaufruf ist erforderlich, da der Zugriff auf Kernstrukturen im Benutzer-Modus, in dem sich der Init-Prozeß befindet, nicht erlaubt ist.

Mounten weiterer Filesysteme mit dem Systemaufruf mount

Ist das Root-Filesystem einmal montiert, werden weitere Filesysteme mit dem Systemaufruf **mount**, der sich in der Datei `fs/super.c` befindet und in der Headerdatei `<linux/fs.h>` deklariert ist, montiert:

```
asm linkage int sys_mount(char * dev_name, char * dir_name, char * type,
                        unsigned long new_flags, void * data);
asm linkage int sys_umount(char * dev_name);
```

mount richtet das Filesystem, das sich auf dem blockorientierten Gerät `dev_name` befindet, im Directory `dirname` ein.

In `type` steht der Typ des zu montierenden Filesystems (wie z.B. `ext2` oder `msdos`).

In `new_flags` können die in Tabelle gezeigten Makros angegeben werden.

Makro ^a	Wert	Bedeutung
MS_RDONLY	1	Filesystem ist nur lesbar.
MS_NOSUID	2	Set-User-ID Bit und Set-Group-ID Bit werden ignoriert.
MS_NODEV	4	Zugriff auf Gerätedateien ist nicht erlaubt.
MS_NOEXEC	8	Ausführen von Dateien ist nicht erlaubt.
MS_SYNCHRONOUS	16	Schreibzugriffe werden sofort (ohne Zwischenspeicherung im Puffercache) auf der Festplatte durchgeführt.
MS_REMOUNT	32	Flags bei schon gemounteten Filesystem werden entsprechend geändert.
MS_MANDLOCK	64	<i>Mandatory Locks</i> (starke Sperren) sind auf Filesystem erlaubt.
S_WRITE	128	Löschen eines i-nodes bewirkt die Freigabe der Quota-Struktur.
S_APPEND	256	Dateien können nur mit dem Flag <code>O_APPEND</code> geöffnet werden.
S_IMMUTABLE	512	Dateien und ihre i-nodes dürfen nicht geändert werden.
S_NOATIME	1024	Kein Update für Zugriffszeiten (<i>access time</i>) findet statt.
S_BAD_INODE	2048	Markierung für nicht lesbare i-nodes.
MS_MGC_VAL		Zeigt die neuere Version des Systemaufrufs mount an. Ohne dieses Flag in den Bits 16-31 werden nur die ersten vier Optionen ausgewertet.

Die filesystemspezifischen Mount-Flags des Superblocks

a. in `<linux/fs.h>` definiert

`data` ist ein Zeiger auf eine beliebige, maximal `PAGE_SIZE-1` große Struktur, die filesystem-spezifische Informationen enthalten kann (diese Daten werden in der Union `u` des Superblocks abgelegt; siehe weiter unten).

Bei `MS_REMOUNT` muß kein Typ und kein Gerät angegeben werden. In diesem Fall aktualisiert **`mount`** nur die in `new_flags` und `data` stehenden Informationen (siehe auch unten).

`umount` demontiert ein Filesystem, indem es den Superblock zurückschreibt und das zugehörige Gerät wieder freigibt. Befindet sich auf `dev_name` das Root-Directory, werden die Quotas abgeschaltet, die Routine **`fsync_dev`** aufgerufen und das Gerät mit `MS_REMOUNT` wieder anmontiert. So können Inkonsistenzen in den Filesystemen verhindert werden. Beide Systemaufrufe (**`sys_mount`** und **`sys_umount`**) sind nur dem Superuser erlaubt.

5.12.2 Initialisierung des Superblocks

Zu jedem montierten Filesystem existiert eine Struktur `super_block`, die die erforderlichen Verwaltungsdaten für dieses Filesystem enthält.

Die Strukturen der montierten Filesysteme werden in einem statischen Array `super_blocks[]` der Größe `NR_SUPER` gehalten.

Die Struktur `super_block` (definiert in `<linux/fs.h>`) hat folgendes Aussehen:

```
struct super_block {
    kdev_t          s_dev;          /* Gerät des Filesystems */
    unsigned long    s_blocksize;    /* Blockgröße */
    unsigned char     s_blocksize_bits; /* Blockgröße als dualer
                                        Logarithmus für
                                        Shift-Operationen */
    unsigned char     s_lock;        /* Sperre für Superblock */
    unsigned char     s_rd_only;     /* ungenutzt (=0) */
    unsigned char     s_dirt;        /* Superblock geändert */
    struct file_system_type *s_type; /* Typ des Filesystems */
    struct super_operations *s_op;   /* Superblockoperationen */
    struct dquot_operations *dq_op; /* Quotaoperationen */
    unsigned long     s_flags;       /* Flags */
    unsigned long     s_magic;       /* Filesystemkennung */
    unsigned long     s_time;        /* Änderungszeit */
    struct inode       *s_covered;    /* Mount-Punkt */
    struct inode       *s_mounted;    /* Root-Inode */
    struct wait_queue  *s_wait;      /* s_lock-Warteschlange */
    union {            /* Filesystemspezifische Informationen */
        struct minix_sb_info minix_sb;
        struct ext_sb_info  ext_sb;
        struct ext2_sb_info ext2_sb;
        struct hpfs_sb_info hpfs_sb;
        struct msdos_sb_info msdos_sb;
        struct isofs_sb_info isofs_sb;
        struct nfs_sb_info  nfs_sb;
        struct xiafs_sb_info xiafs_sb;
        struct sysv_sb_info sysv_sb;
        struct affs_sb_info affs_sb;
    };
};
```



```

    struct ufs_sb_info  ufs_sb;
    void *generic_sbp;
} u;
};

```

Der Superblock enthält Informationen über das gesamte Filesystem, wie etwa die Blockgröße, Zugriffsrechte und Zeit der letzten Änderung. Des weiteren enthält die Union `u` am Ende der Struktur spezielle Informationen über das entsprechende Filesystem. Für nachträglich eingebundene Filesystem-Module existiert der Zeiger `generic_sbp`.

Für die Initialisierung eines Superblocks ist die Funktion ***read_super*** des VFS zuständig, die in `fs/super.c` wie folgt definiert ist.

```

struct super_block * read_super(kdev_t dev, const char *name, int flags,
                                void *data, int silent)
{
    struct super_block * s;
    struct file_system_type *type;

    if (!dev)
        return NULL;
    check_disk_change(dev);
    s = get_super(dev);
    if (s)
        return s; /* Rueckgabe eines schon existierenden Superblocks */
    if (!(type = get_fs_type(name))) {
        printk("VFS: on device %s: get_fs_type(%s) failed\n",
               kdevname(dev), name);
        return NULL;
    }
    for (s = 0+super_blocks ;; s++) {
        if (s >= NR_SUPER+super_blocks)
            return NULL;
        if (!(s->s_dev))
            break;
    }
    s->s_dev = dev;
    s->s_flags = flags;
    /* Aufruf der filesystemspezifischen Funktion read_super */
    if (!(type->read_super(s, data, silent))) {
        s->s_dev = 0;
        return NULL;
    }
    s->s_dev = dev;
    s->s_covered = NULL;
    s->s_rd_only = 0;
    s->s_dirt = 0;
    s->s_type = type;
    return s;
}

```

Die Funktion ***read_super*** überprüft, ob der Superblock schon existiert und liefert ihn als Rückgabewert.

Existiert der Superblock noch nicht, sucht die Funktion **read_super** einen freien Eintrag im Array `super_blocks` und ruft die von dem speziellen Filesystem bereitgestellte Funktion zur Generierung des Superblocks auf. Diese filesystemspezifische Funktion wurde dem VFS bei der Registrierung mit **register_filesystem** bekanntgemacht. Die Deklaration der filesystemspezifischen Systemfunktion **read_super** hat z.B. für das `ext2`-Filesystem folgendes Aussehen:

```
struct super_block * ext2_read_super (struct super_block * sb, void * data,
                                     int silent)
```

Sie erhält beim Aufruf die Adresse der entsprechenden Superblockstruktur (`sb`), in der die Komponenten `s_dev` und `s_flags` entsprechend gesetzt sind.

Weitere mount-Optionen für das Filesystem werden über den void-Zeiger `data` übergeben, und das Flag `silent` gibt an, ob bei einem nicht erfolgreichem Mounten Fehlermeldungen auszugeben sind (0) oder nicht (1). Die Kernfunktion **mount_root** setzt z.B. das Flag `silent`, da sie nacheinander alle vorhandenen filesystemspezifischen **read_super** zum Mounten aufruft und dabei ständige Fehlermeldungen beim Hochfahren des Systems sehr störend wären.

Über die Komponenten `s_lock` und `s_wait` wird der Zugriff auf den Superblock synchronisiert.

Dies geschieht mit den Funktionen **lock_super** und **unlock_super**, die in der Datei `<linux/locks.h>` wie folgt definiert sind:

```
extern inline void lock_super(struct super_block * sb)
{
    if (sb->s_lock)
        __wait_on_super(sb);
    sb->s_lock = 1;
}

extern inline void unlock_super(struct super_block * sb)
{
    sb->s_lock = 0;
    wake_up(&sb->s_wait);
}
```

Außerdem enthält der Superblock Verweise auf den Root-Inode des Filesystems (`s_mounted`) und auf den Mount-Point (`s_covered`).

5.12.3 Operationen auf den Superblock

Die Superblockstruktur stellt über die Komponente `s_op` Funktionen zum Zugriff auf das Filesystem zur Verfügung:

```
struct super_operations {
    void (*read_inode) (struct inode *);
    int (*notify_change) (struct inode *, struct iattr *);
    void (*write_inode) (struct inode *);
```

```

void (*put_inode) (struct inode *);
void (*put_super) (struct super_block *);
void (*write_super) (struct super_block *);
void (*statfs) (struct super_block *, struct statfs *);
int (*remount_fs) (struct super_block *, int *, char *);
};

```

Operationen auf den Superblock werden üblicherweise nur über diese Funktionen vorgenommen, so daß die eigentliche Struktur des Superblocks nach außen nicht sichtbar ist. Es gibt sogar Anwendungsfälle, wo die i-nodes und der Superblock gar nicht in der vorliegenden Form existieren, aber über diese Funktionen nachgebildet werden. Dies geschieht z.B. bei einem MS-DOS-Filesystem, bei dem die FAT (File Allocation Table) und die Daten im Superblock in die Linux-internen Strukturen des Superblocks und der i-nodes transformiert werden.

Wird eine der obigen Superblockoperationen für ein spezielles Filesystem nicht angeboten, so ist der entsprechende Funktionszeiger auf NULL gesetzt und es findet beim Aufruf einer solchen Funktion keinerlei Aktion statt.

Im folgenden werden die einzelnen Superblockoperationen (Funktionen) etwas genauer erläutert. Die zugehörigen filesystemspezifischen Funktionen befinden sich im entsprechenden Subdirectory in der Datei `super.c` bzw. `inode.c`, wie z.B. **ext2_write_inode** in `fs/ext2/inode.c`.

read_inode(&inode)

Diese Funktion ist für das Setzen der einzelnen Komponenten in der Strukturvariablen `inode` zuständig. Eine ihrer Hauptaufgaben ist – in Abhängigkeit von der jeweiligen Dateart – das Eintragen der entsprechenden i-node-Operationen in die Strukturvariable `inode`, wie z.B. für das ext2-Filesystem:

```

read_inode(inode)
{
    .....
    else if (S_ISREG(inode->i_mode))
        inode->i_op = &ext2_file_inode_operations;
    else if (S_ISDIR(inode->i_mode))
        inode->i_op = &ext2_dir_inode_operations;
    else if (S_ISLNK(inode->i_mode))
        inode->i_op = &ext2_symlink_inode_operations;
    else if (S_ISCHR(inode->i_mode))
        inode->i_op = &chrdev_inode_operations;
    else if (S_ISBLK(inode->i_mode))
        inode->i_op = &blkdev_inode_operations;
    else if (S_ISFIFO(inode->i_mode))
        init_fifo(inode);
    .....
}

```

Die Funktion **read_inode** wird von der Funktion **__iget** aufgerufen, nachdem diese zuvor die Komponenten **i_dev**, **i_ino**, **i_sb** und **i_flags** in der Strukturvariablen **inode**, deren Adresse übergeben wird, gesetzt hat.

notify_change(&inode, &iattr)

Diese Funktion bewirkt, daß i-node-Änderungen, die durch Systemaufrufe verursacht wurden, allen beteiligten Rechnern mitgeteilt werden und auch dort entsprechend durchgeführt werden. Dies ist bei NFS wichtig, da bei diesem Filesystem nicht nur ein lokaler, sondern auch ein externer i-node auf einem anderen Rechner existiert. Die vorzunehmenden Änderungen befinden sich dabei in der übergebenen Strukturvariablen **iattr**:

```
struct iattr {
    unsigned int    ia_valid; /* Flags, die geänderte
                               Komponenten anzeigen */
    umode_t         ia_mode; /* Neue Zugriffsrechte */
    uid_t           ia_uid; /* Neuer Eigentümer */
    gid_t           ia_gid; /* Neue Gruppenzugehörigkeit */
    off_t           ia_size; /* Neue Größe */
    time_t          ia_atime; /* Zeit des letzten Zugriffs */
    time_t          ia_mtime; /* Zeit der letzten Änderung */
    time_t          ia_ctime; /* Zeit der letzten i-node-Änderung */
};
```

In **ia_valid** zeigen die einzelnen Bits an, welche Komponenten in der Struktur **iattr** von Änderungen betroffen sind. Welche Bits sich dabei auf welche Komponente beziehen, ist in **<linux/fs.h>** definiert, wie z.B.:

```
/*
 * Attribute flags. These should be or-ed together to figure out what
 * has been changed!
 */
#define ATTR_MODE      1
#define ATTR_UID       2
#define ATTR_GID       4
#define ATTR_SIZE      8
#define ATTR_ATIME     16
#define ATTR_MTIME     32
#define ATTR_CTIME     64
#define ATTR_ATIME_SET 128
#define ATTR_MTIME_SET 256
#define ATTR_FORCE     512 /* Not a change, but a change it */
```

Tabelle 5.10 zeigt, welche Funktionen **notify_change** aufrufen und welche Flags von diesen Funktionen in der Komponente **ia_valid** der übergebenen Strukturvariablen **iattr** gesetzt werden.

Kernfunktion	ATTR_MODE	ATTR_UID	ATTR_GID	ATTR_SIZE	ATTR_ETIME	ATTR_MTIME	ATTR_CTIME	ATTR_ETIME_SET	ATTR_MTIME_SET
sys_chmod	x						x		
sys_fchmod	x						x		
sys_chown	x	x	x				x		
sys_fchown	x	x	x				x		
sys_truncate				x		x	x		
sys_ftruncate				x		x	x		
sys_write	x								
open_namei				x					
sys_utime					x	x	x	x	x

Tabelle 5.10: Die Flags von *ia_valid* für die Funktion *notify_change*

write_inode(&inode)

Diese Funktion sichert den übergebenen *inode*, was bedeutet, daß der im Cache befindliche *inode* nun in jedem Fall auf die Festplatte zurückgeschrieben wird. Die Konsistenz des Filesystems muß dabei nicht unbedingt gewährleistet sein, was bedeutet, daß die entsprechenden Datenblöcke, Freispeicherlisten usw. nicht zurückgeschrieben werden müssen, weshalb das Filesystem eventuell nicht mehr konsistent ist. Unterstützt das jeweilige Filesystem ein auf Inkonsistenz hinweisendes Flag (Validflag), so sollte dieses gesetzt werden.

put_inode(&inode)

Die Aufgabe dieser Funktion ist es, die entsprechende Datei physikalisch zu löschen und die von ihr belegten Blöcke freizugeben, wenn *i_nlink* den Wert 0 hat. Diese Funktion wird von *iput* aufgerufen, wenn ein *i-node* nicht mehr benötigt wird.

put_super(&super_block)

Diese Funktion ruft das VFS beim Unmounten eines Filesystems auf. Die Aufgabe dieser Funktion ist das Freigeben des Superblocks und der dazugehörigen Informationspuffer bzw. die Wiederherstellung der Konsistenz des Filesystems. Dazu sollte das Validflag wieder entsprechend und die Komponente *s_dev* der Superblockstruktur auf 0 gesetzt werden, damit der Superblock nach dem Unmounten wieder korrekt zur Verfügung steht.

write_super(&super_block)

Diese Funktion sichert den übergebenen `super_block`, was bedeutet, daß der im Cache befindliche `super_block` nun in jedem Fall auf die Festplatte zurückgeschrieben wird. Die Konsistenz des Filesystems muß dabei nicht unbedingt gewährleistet sein, was bedeutet, daß die entsprechenden Datenblöcke, Freispeicherlisten usw. nicht zurückgeschrieben werden müssen, weshalb das Filesystem eventuell nicht mehr konsistent ist. Unterstützt das jeweilige Filesystem ein auf Inkonsistenz hinweisendes Flag (Validflag), so sollte dieses gesetzt werden.

statfs(&super_block, &statfs)

Diese Funktion, die für das Füllen der Strukturvariablen `statfs` verantwortlich ist, wird von den beiden Systemfunktionen **`statfs`** und **`fstatfs`** aufgerufen, die in `fs/open.c` definiert und in `<sys/vfs.h>` wie folgt deklariert sind:

```
int sys_statfs(const char *path, struct statfs *buf);
int sys_fstatfs(unsigned int fd, struct statfs *buf);
```

Die Funktion **`sys_statfs`** gibt Informationen zum Filesystem zurück, auf dem sich die Datei `path` befindet. Bei **`sys_fstatfs`** wird anstelle eines Dateinamens der Filedeskriptor einer geöffneten Datei angegeben. Die Struktur `statfs` ist in `<linux/vfs.h>` wie folgt definiert:

```
struct statfs {
    long   f_type;      /* Typ des Filesystems          */
    long   f_bsize;     /* Optimale Blockgröße         */
    long   f_blocks;    /* Anzahl der Blöcke           */
    long   f_bfree;     /* Gesamtzahl der freien Blöcke */
    long   f_bavail;    /* Frei Blöcke für den Benutzer */
    long   f_files;     /* Anzahl der i-nodes          */
    long   f_ffree;     /* Anzahl der freien i-nodes   */
    fsid_t f_fsid;      /* ID (Kennung) des Filesystems */
    long   f_namelen;   /* maximale Länge für Dateinamen */
    long   f_spare[6];  /* nicht genutzt               */
};
```

Komponenten, die in einem speziellen Filesystem nicht definiert sind, werden auf -1 gesetzt.

remount_fs(&super_block, &flags, &data)

Diese Funktion wird bei Änderungen eines Filesystems aufgerufen, wobei nur die neuen Attribute im Superblock eingetragen werden und so die Konsistenz des Filesystems wiederhergestellt wird.

5.12.4 Der i-node

Beim Mounten eines Filesystems wird der Superblock erzeugt und in der i-node-Struktur des anmontierten Filesystems wird in der Komponente `i_mount` der Root-i-node eingetragen. Die Struktur `inode` ist dabei wie folgt in `<linux/fs.h>` definiert:

```
struct inode {
    kdev_t      i_dev;          /* Gerätenummer der Datei */
    unsigned long i_ino;        /* i-node-Nummer */
    umode_t     i_mode;         /* Dateiarart und Zugriffsrechte */
    nlink_t     i_nlink;        /* Anzahl der Links (Hard-Links) */
    uid_t       i_uid;          /* Eigentümer */
    gid_t       i_gid;          /* Gruppe */
    kdev_t      i_rdev;         /* Gerät bei Gerätedateien */
    off_t       i_size;         /* Größe */
    time_t      i_atime;        /* Zeit des letzten Zugriffs */
    time_t      i_mtime;        /* Zeit der letzten Änderung */
    time_t      i_ctime;        /* Zeit der letzten i-node-Änderung */
    unsigned long i_blksize;     /* Blockgröße */
    unsigned long i_blocks;      /* Blockanzahl */
    unsigned long i_version;     /* Dcache-Versionsnummer */
    unsigned long i_nrpages;     /* Anzahl der Pages */
    struct semaphore i_sem;      /* Zugriffsteuerung über Semaphore */
    struct inode_operations *i_op; /* i-node-Operationen */
    struct super_block *i_sb;     /* Superblock */
    struct wait_queue *i_wait;    /* Warteschlange-Information */
    struct file_lock *i_flock;    /* Dateisperren */
    struct vm_area_struct *i_mmap; /* Speicherbereiche */
    struct page *i_pages;         /* Page-Informationen */
    struct dquot *i_dquot[MAXQUOTAS]; /* Quota-Informationen */
    struct inode *i_next, *i_prev; /* Nachfolger/Vorgänger in i-node-Liste */
    struct inode *i_hash_next, *i_hash_prev; /* ..... in Hashtabelle */
    struct inode *i_bound_to, *i_bound_by;
    struct inode *i_mount;        /* Root-i-node des Filesystems */
    unsigned short i_count;        /* Referenzzähler */
    unsigned short i_flags;        /* Flags (aus Superblock) */
    unsigned char i_lock;          /* Sperre */
    unsigned char i_dirt;          /* zeigt an, daß i-node geändert wurde */
    unsigned char i_pipe;          /* zeigt an, daß i-node eine Pipe ist */
    unsigned char i_sock;          /* zeigt an, daß i-node Socket ist */
    unsigned char i_seek;          /* ungenutzt */
    unsigned char i_update;        /* zeigt an, ob i-node uptodate ist */
    unsigned short i_writecount;    /* Schreibzugriffe */
    union {                       /* filesystemspezifische Informationen */
        struct pipe_inode_info pipe_i;
        struct minix_inode_info minix_i;
        struct ext_inode_info ext_i;
        struct ext2_inode_info ext2_i;
        struct hpfs_inode_info hpfs_i;
        struct msdos_inode_info msdos_i;
        struct umsdos_inode_info umsdos_i;
        struct iso_inode_info isofs_i;
        struct nfs_inode_info nfs_i;
    };
};
```

```

    struct xiafs_inode_info xiafs_i;
    struct sysv_inode_info sysv_i;
    struct affs_inode_info affs_i;
    struct ufs_inode_info ufs_i;
    struct socket socket_i;
    void * generic_ip;
} u;
};

```

Freie i-nodes lassen sich daran erkennen, daß bei ihnen die Komponenten `i_count`, `i_dirt` und `i_lock` auf 0 gesetzt sind. Die Anzahl aller vorhandenen i-nodes wird in der statischen Variablen `nr_inodes` und die Anzahl der freien i-nodes in der statischen Variablen `nr_free_inode` gehalten.

Die Verwaltung der i-nodes erfolgt im Speicher auf zwei verschiedene Arten:

- ▶ Als doppelt verkettete Ringliste, auf deren Anfangsknoten die Zeigervariable `first_inode` zeigt. Das Durchlaufen der Liste ist dabei vorwärts mit der Komponente `i_next` und rückwärts mit der Komponente `i_prev` möglich. Da auch freie i-nodes in der Ringliste gehalten werden, ist ein Zugriff auf einzelne i-nodes über diese Ringliste sehr langsam.
- ▶ Als offene Hashtabelle (`hash_tabelle[NR_IHASH]`) für einen schnellen Zugriff auf einzelne i-nodes. Kollisionen sind dabei als doppelt verkettete Liste organisiert, die mittels den Komponenten `i_hash_next` und `i_hash_prev` vorwärts bzw. rückwärts durchlaufen werden kann. Der Index für den Zugriff auf die Hashtabelle wird über die i-node- bzw. Gerätenummer ermittelt.

Operationen auf i-nodes sind mit den Funktionen ***iget***, ***namei***, ***lnamei*** und ***iput*** möglich, die wie folgt in `<linux/fs.h>` definiert sind:

```

inline struct inode * iget(struct super_block * sb, int nr)
{
    return __iget(sb, nr, 1);
}

struct inode * __iget(struct super_block * sb, int nr, int crsmnt);

void iput(struct inode * inode);

```

iget(&super_block, nr)

Diese Funktion liefert den über `super_block` und über die i-node-Nummer `nr` spezifizierten i-node. Die Funktion ***iget*** wiederum ruft ihrerseits die Funktion ***__iget*** auf.

__iget(&super_block, nr, crsmnt)

Diese Funktion kann über den zusätzlichen Parameter `crsmnt` angewiesen werden, auch Mount-Points aufzulösen, was bedeutet, daß sie den entsprechenden Root-i-node des ammontierten Filesystems liefert, wenn der angeforderte i-node ein Mount-Point ist.

Wird ein angeforderter i-node in der Hashtabelle gefunden, wird dort der Referenzzähler `i_count` um 1 inkrementiert und dessen Adresse als Rückgabewert geliefert. Ist der entsprechende i-node noch nicht in der Hashtabelle enthalten, wird mit dem Aufruf der Funktion **`get_empty_inode`** ein noch freier i-node gesucht, dieser über die filesystemspezifische Superblockoperation **`read_inode`** entsprechend gefüllt und in die Hashtabelle eingetragen, bevor dessen Adresse als Rückgabewert geliefert wird.

`iput(&inode)`

Diese Funktion veranlaßt wieder die Freigabe eines mit **`iget`** erhaltenen i-nodes. Dazu verringert sie den Referenzzähler des entsprechenden i-nodes um 1. Sollte dadurch der Referenzzähler in `i_count` den Wert 0 annehmen, markiert sie diesen i-node wieder als freien i-node.

`namei` und `lnamei`

Diese beiden Funktionen sind wie folgt in `<linux/fs.h>` deklariert:

```
int namei(const char * pathname, struct inode ** res_inode);
int lnamei(const char * pathname, struct inode ** res_inode);
```

Die Funktion **`namei`** löst den ihr übergebenen Pfadnamen `pathname` auf und speichert die Adresse des zur Datei `pathname` gehörenden i-node in `res_node`. Die Funktion **`lnamei`** unterscheidet sich von **`namei`** dadurch, daß **`lnamei`** symbolische Links nicht auflöst und somit den i-node eines Links selbst liefert.

Beide Funktionen verwenden die zuvor beschriebenen Funktionen **`iget`** und **`iput`** zum Zugriff auf den i-node. Zudem rufen beide Funktionen die Funktion **`_namei`** auf, die in `fs/namei.c` definiert ist und folgende Deklaration besitzt:

```
static int _namei(const char * pathname, struct inode * base,
                 int follow_links, struct inode ** res_inode)
```

Diese Funktion hat zwei zusätzliche Parameter: den i-node des entsprechenden Basisdirectorys (`base`), von dem aus aufzulösen ist, und ein Flag `follow_links`, das anzeigt, ob mit Hilfe der Funktion **`follow_link`** symbolische Links aufzulösen sind oder nicht. **`_namei`** wiederum läßt die Hauptarbeit durch einen Aufruf der Funktion **`dir_namei`** leisten.

`dir_namei`, dessen Definition in `fs/namei.c` wie folgt beginnt, liefert den i-node des Directorys, in dem sich die Datei mit dem entsprechenden Namen befindet:

```
/*
 *      dir_namei()
 *
 *  dir_namei() returns the inode of the directory of the
 *  specified name, and the name within that directory.
 */
static int dir_namei(const char *pathname, int *namelen, const char **name,
                    struct inode * base, struct inode **res_inode)
```

Ein negativer Rückgabewert (Fehlercode) zeigt bei allen hier vorgestellten Funktionen einen Fehler an.

5.12.5 i-node-Operationen

Die i-node-Struktur stellt über die Komponente `i_op` filesystemspezifische Funktionen zum Zugriff auf i-nodes und damit auf Dateien des speziellen Filesystems zur Verfügung:

```
struct inode_operations {
    struct file_operations * default_file_ops;
    int (*create) (struct inode *,const char *,int,int,struct inode **);
    int (*lookup) (struct inode *,const char *,int,struct inode **);
    int (*link) (struct inode *,struct inode *,const char *,int);
    int (*unlink) (struct inode *,const char *,int);
    int (*symlink) (struct inode *,const char *,int,const char *);
    int (*mkdir) (struct inode *,const char *,int,int);
    int (*rmdir) (struct inode *,const char *,int);
    int (*mknod) (struct inode *,const char *,int,int,int);
    int (*rename) (struct inode *,const char *,int,struct inode *,
                    const char *,int, int);
    int (*readlink) (struct inode *,char *,int);
    int (*follow_link) (struct inode *,struct inode *,int,int,struct inode **);
    int (*bmap) (struct inode *,int);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*smap) (struct inode *,int);
};
```

Da der Referenzzähler der diesen Funktionen übergebenen i-nodes schon vor ihrem Aufruf um 1 inkrementiert wurde, um die Verwendung der entsprechenden i-nodes anzuzeigen, ist allen diesen Funktionen gemeinsam, daß sie vor ihrer Rückkehr immer die ihnen übergebenen i-nodes mit einem Aufruf der Funktion ***iput*** wieder freigeben.

Nachfolgend werden die einzelnen Funktionen etwas genauer vorgestellt. Alle diese Funktionen können nur erfolgreich ablaufen, wenn sie die entsprechenden Rechte für die betreffende Aktion haben.

create

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_create (struct inode * dir,const char * name, int len, int mode,
                 struct inode ** result)
```

Diese Funktion kreiert mit dem Aufruf einer Funktion (wie z.B. ***ext2_new_inode***) einen neuen i-node und füllt diesen filesystemspezifisch. Zusätzlich trägt ***create*** den Dateinamen `name` der Länge `len` in das durch den i-node `dir` angegebene Directory ein. Den neu erzeugten i-node liefert sie über den Parameter `result` zurück. ***create*** wird in der Funktion ***open_namei*** des VFS aufgerufen.

lookup

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_lookup (struct inode * dir, const char * name, int len,
                 struct inode ** result)
```

lookup liefert den i-node des Dateinamens `name` (mit der Länge `len`) in dem durch den i-node `dir` angegebenen Directory über den Parameter `result` zurück.

link

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_link (struct inode * oldinode, struct inode * dir,
               const char * name, int len)
```

link ist für das Anlegen von Hard-Links zuständig. Diese Funktion legt in dem durch den i-node `dir` festgelegten Directory einen Dateinamen `name` (mit der Länge `len`) an, der als i-node den angegebenen `oldinode` erhält.

unlink

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_unlink (struct inode * dir, const char * name, int len)
```

Diese Funktion löscht die angegebene Datei `name` (mit der Länge `len`) in dem durch den i-node `dir` spezifizierten Directory.

symlink

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_symlink (struct inode * dir, const char * name, int len,
                  const char * symname)
```

symlink ist für das Anlegen von Soft-Links zuständig. Diese Funktion legt in dem durch den i-node `dir` festgelegten Directory einen symbolischen Link `name` (mit der Länge `len`) an, der auf den Pfad `symname` zeigt.

mkdir

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_mkdir (struct inode * dir, const char * name, int len, int mode)
```

mkdir legt in dem durch den i-node `dir` festgelegten Directory ein Directory `name` (mit der Länge `len`) und den Zugriffsrechten `mode` an.

rmdir

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_rmdir (struct inode * dir, const char * name, int len)
```

rmdir löscht in dem durch den i-node `dir` festgelegten Directory das Subdirectory `name` (mit der Länge `len`). Das entsprechende Subdirectory muß leer sein und darf nicht von einem Prozeß benutzt werden.

mknod

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_mknod (struct inode * dir, const char * name, int len, int mode,  
                int rdev)
```

mknod legt einen neuen i-node mit dem Modus `mode` an. Dieser i-node erhält im Directory `dir` den Namen `name` (mit der Länge `len`). Falls es sich beim i-node um eine Gerätedatei handelt, enthält der Parameter `rdev` die Gerätenummer.

rename

Diese filesystemspezifische Funktion ist in der Datei `namei.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/namei.c`) wie folgt definiert:

```
int ext2_rename (struct inode * old_dir, const char * old_name, int old_len,  
                struct inode * new_dir, const char * new_name, int new_len,  
                int must_be_dir)
```

rename ändert den Namen einer Datei. Dazu muß in dem durch den i-node festgelegten Directory `old_dir` der Name `old_name` (mit der Länge `old_len`) gelöscht und in dem durch den i-node festgelegten Directory `new_dir` der Name `new_name` (mit der Länge `new_len`) eingetragen werden. Falls das Flag `must_be_dir` gesetzt ist, muß es sich bei `old_dir` um den i-node eines Directories handeln.

readlink

Diese filesystemspezifische Funktion ist in der Datei `symlink.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/symlink.c`) wie folgt definiert:

```
static int ext2_readlink (struct inode * inode, char * buffer, int buflen)
```

readlink liest den symbolischen Link aus, der sich in der mit i-node spezifizierten Datei befindet. Den Pfad, auf den der symbolische Link zeigt, kopiert diese Funktion an die übergebene Adresse `buffer`, wobei sie aber maximal `buflen` Zeichen dorthin schreibt. Diese Funktion wird direkt von der Systemfunktion **sys_readlink** aufgerufen.

follow_link

Diese filesystemspezifische Funktion ist in der Datei `symlink.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/symlink.c`) wie folgt definiert:

```
static int ext2_follow_link(struct inode * dir, struct inode * inode,  
                           int flag, int mode, struct inode ** res_inode)
```

follow_link liefert den Ziel-i-node, auf den ein symbolischer Link oder auch eventuell mehrfach verkettete symbolische Links zeigen. Diese Funktion liefert im Parameter `res_inode` den i-node, auf den der über `dir` (Directory) und `inode` (Datei) spezifizierte i-node zeigt. Unter Linux ist festgelegt, daß bei symbolischen Links, die wiederum auf symbolische Links zeigen, maximal 5 nacheinander verkettete symbolische Links aufgelöst werden. So können Endlosschleifen vermieden werden.

bmap

Diese filesystemspezifische Funktion ist in der Datei `inode.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/inode.c`) wie folgt definiert:

```
int ext2_bmap(struct inode * inode, int block)
```

bmap wird verwendet, um das Memory-Mapping von Dateien zu ermöglichen. Der Parameter `block` gibt die Nummer eines logischen Datenblocks einer Datei an. Diese Nummer muß von **bmap** in die logische Blocknummer des Blocks auf dem Gerät umgeformt werden.

truncate

Diese filesystemspezifische Funktion ist in der Datei `truncate.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/truncate.c`) wie folgt definiert:

```
void ext2_truncate(struct inode * inode)
```

truncate dient zum Kürzen von Dateien (Abschneiden am Dateiende), kann aber auch zum Verlängern eingesetzt werden. Der übergebene `inode` legt die zu verändernde Datei fest. Die Komponente `i_size` der entsprechenden `inode`-Struktur muß vor dem **truncate**-Aufruf bereits auf die neue Länge gesetzt werden.

Die Funktion **truncate**, die auch für die Freigabe von nicht mehr benötigten Blöcken zuständig ist, wird nicht nur von der Systemfunktion **sys_truncate**, sondern auch an vielen anderen Stellen verwendet, wie z.B. beim Öffnen einer Datei zum Schreiben oder zum physikalischen Löschen einer Datei, bevor der entsprechende i-node entfernt wird.

permission

Diese filesystemspezifische Funktion ist in der Datei `acl.c` im Directory des jeweiligen Filesystems (wie z.B. `fs/ext2/acl.c`) wie folgt definiert:

```
int ext2_permission(struct inode * inode, int mask)
```

permission überprüft für den übergebenen `inode`, ob die durch `mask` angegebenen Zugriffsrechte für den aktuellen Prozeß vorliegen. Die möglichen Werte für `mask` sind `MAY_READ`, `MAY_WRITE` und `MAY_EXEC`.

smmap

Diese filesystemspezifische Funktion ist in der Datei `cache.c` im Directory des `fat`-Filesystems (`fs/fat/cache.c`) wie folgt definiert:

```
int fat_smmap(struct inode * inode, int sector)
```

smmap ist für das Arbeiten mit Swap-Dateien auf einem UMSDOS-Filesystem zuständig. Wie **bmap** liefert die Funktion **smmap** die logische Sektornummer (nicht Block oder Cluster) auf dem Gerät des angegebenen Sektors der Datei.

5.12.6 Fileoperationen

Die Struktur `file` enthält Informationen über Zugriffsrechte, Position des Schreib-/Lesezeigers, Zugriffsart (Lesen, Schreiben ...), Anzahl der Zugriffe einer geöffneten Datei usw.:

```
struct file {
    mode_t      f_mode;           /* Zugriffsart */
    loff_t      f_pos;           /* Position des Schreib-/Lesezeigers */
    unsigned short f_flags;      /* Flags der open-Funktion */
    unsigned short f_count;      /* Referenzzähler */
    unsigned long f_reada, ...;  /* Read ahead-Flag und andere Flags */
    struct file  *f_next, *f_prev; /* Nachfolger/Vorgänger in Ringliste */
    struct fown_struct f_owner;  /* Eigentümer-Informationen */
    struct inode *f_inode;       /* zugehöriger i-node */
    struct file_operations *f_op; /* File-Operationen */
    unsigned long f_version;     /* Dcache-Versionsnummer */
    void         *private_data;  /* Daten für Terminal-Treiber */
};
```

Die Verwaltung von `file`-Strukturen erfolgt im Speicher in Form einer doppelt verkettete Ringliste, auf deren Anfangsknoten die Zeigervariable `first_file` zeigt. Das Durchlaufen dieser Ringliste ist dabei vorwärts mit der Komponente `f_next` und rückwärts mit der Komponente `f_prev` möglich.

Die `file`-Struktur stellt über die Komponente `f_op` Funktionen zum Arbeiten mit Dateien (Öffnen, Lesen, Schreiben usw.) zur Verfügung. Neben diesen Funktionen enthält die Struktur `inode_operations` (siehe oben) eine eigene Komponente `default_file_ops`, in der

Standardoperationen für Dateien bereits festgelegt sind. Die Struktur `file_operations` hat das folgende Aussehen:

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, const char *, int);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *,
                  unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

Während die früher vorgestellten i-node-Operationen nur mit der Repräsentation eines Sockets oder Geräts in dem entsprechenden Filesystem bzw. dessen Darstellung im Speicher arbeiten, beinhalten die hier angegebenen Funktionen die wirkliche Funktionalität von Geräten und Sockets.

Nachfolgend werden die einzelnen Funktionen kurz beschrieben:

`lseek(&inode, &file, offset, wie)`

ist für die Positionierung des Schreib/Lesezeigers zuständig.

`read(&inode, &file, buffer, count)`

kopiert `count` Bytes aus der Datei `file` in den `buffer` (im Benutzeradreßraum).

`write(&inode, &file, buffer, count)`

kopiert `count` Bytes aus dem `buffer` (im Benutzeradreßraum) in die Datei `file`.

`readdir(&inode, &file, dirent, count)`

liefert den nächsten Directory-Eintrag in der Struktur `dirent` zurück.

`select(&inode, &file, type, &select_table)`

prüft, ob Daten von einer Datei gelesen oder in eine Datei geschrieben werden können oder ob Ausnahmebedingungen vorliegen. Diese Funktion ist nur für Gerätetreiber und Sockets sinnvoll.

ioctl(&inode, &file, cmd, arg)

dient zur Einstellung von gerätespezifischen Parametern. Vor einem Aufruf der `ioctl` - Funktion prüft das VFS, ob im `cmd`-Argument eines der folgenden Flags gesetzt ist:

FIONCLEX	close-on-exec-Bit löschen
FIOCLEX	close-on-exec-Bit setzen
FIONBIO	Falls das Argument <code>arg</code> ein von 0 verschiedener Wert ist, wird das Flag <code>O_NONBLOCK</code> gesetzt, ansonsten wird dieses Flag gelöscht
FIOASYNC	Falls das Argument <code>arg</code> ein von 0 verschiedener Wert ist, wird das Flag <code>O_SYNC</code> gesetzt, ansonsten wird dieses Flag gelöscht

Enthält `cmd` keines dieser Flags, wird geprüft, ob der übergebene `file`-Zeiger auf eine reguläre Datei zeigt. Trifft dies zu, wird die Funktion **`file_ioctl`** aufgerufen. Für andere Dateiarnten prüft das VFS, ob eine entsprechende **`ioctl`**-Funktion verfügbar ist. Wenn ja, wird diese filesystemspezifische **`ioctl`**-Funktion aufgerufen, andernfalls wird der Fehler `EINVAL` zurückgegeben.

mmap(&inode, &file, &vm_area_struct)

bildet einen Teil einer Datei in den Benutzeradreßraum des aktuellen Prozesses ab. Die übergebene Struktur `vm_area_struct` legt die Eigenschaften für den entsprechenden Speicherraum fest. Diese Struktur ist in `<linux/mm.h>` definiert und enthält unter anderem die folgenden drei Komponenten:

<code>vm_start</code>	Startadresse des Speicherbereichs, in den Datei abzubilden ist
<code>vm_end</code>	Endadresse des Speicherbereichs, in den Datei abzubilden ist
<code>vm_offset</code>	Position in der Datei, ab der Abbildung erfolgt

release(&inode, &file)

wird für die Freigabe der `file`-Struktur benötigt und wird – wie die Funktion **`open`** – nur für Gerätetreiber benötigt, da das VFS von sich aus über alle notwendigen Operationen für Dateien (wie z.B. die Aktualisierung des i-nodes) verfügt.

fsync(&inode, &file)

wird für das Leeren aller Puffer und das Zurückschreiben dieser auf das entsprechende Gerät benötigt, weshalb diese Funktion auch nur für Filesysteme von Interesse ist. Bietet ein Filesystem diese Funktion nicht an, wird `EINVAL` zurückgegeben.

fasync(&inode, &file, flag)

wird vom VFS aufgerufen, wenn sich ein Prozeß mittels *fcntl* eine asynchrone Benachrichtigung durch das Signal SIGIO einrichtet bzw. eine solche Einrichtung wieder abschaltet. Der betreffende Prozeß soll dabei benachrichtigt werden, wenn Daten für ihn eintreffen und wenn *flag* gesetzt ist. Ist *flag* nicht gesetzt, so bedeutet dies, daß der Prozeß seine eingerichtete Benachrichtigung wieder abschalten möchte. Terminaltreiber und Sockets stellen diese Funktion zur Verfügung.

check_media_change(kdev_t)

wird nur für wechselbare Medien (wie z.B. Diskettenlaufwerke, JAZZ-Laufwerke usw.) benötigt. Diese Funktion muß prüfen, ob das über *kdev_t* festgelegte Medium seit der letzten darauf stattgefundenen Aktion gewechselt wurde (Rückgabe 1) oder nicht (Rückgabe 0). **check_media_change** wird von der VFS-Funktion **check_disk_change** aufgerufen. Im Falle eines Medienwechsels entfernt diese VFS-Funktion durch einen Aufruf von **put_super** einen eventuell zu diesem Gerät gehörigen Superblock, gibt alle diesem Gerät zugeteilten Puffer im Cachepuffer und alle i-nodes frei. Danach wird **revalidate** (siehe weiter unten) aufgerufen. **check_disk_change** wird nur beim Mounten eines Geräts aufgerufen. Steht diese Funktion nicht zur Verfügung, wird immer der Rückgabewert 0 (kein Wechsel) geliefert.

revalidate(kdev_t)

wird vom VFS nach einem Medienwechsel aufgerufen, um die Konsistenz des zugehörigen Blockgeräts wiederherzustellen.

open(&inode, &file)

wird nur für Gerätetreiber benötigt, da das VFS von sich aus über alle notwendigen Operationen für Dateien (wie z.B. die Allokierung der *file*-Struktur) verfügt.

Wird die Systemfunktion **open** für Dateien aufgerufen, so ist es die Aufgabe des VMS die entsprechenden Operationen für die Interaktion zwischen dem speziellen Filesystem und dem zugehörigen Gerät durchzuführen. Dazu existiert die Funktion **do_open** (in *fs/open.c*), die zunächst eine neue *file*-Struktur mittels der Funktion **get_empty_filep** anfordert. Diese zurückgelieferte Struktur wird dann in die Dateitabelle des aufrufenden Prozesses eingetragen, wobei die Komponenten *f_flags* und *f_mode* gesetzt werden.

Zum Erfragen des i-nodes der zu öffnenden Datei ruft **do_open** die Funktion **open_namei**, die ihrerseits zunächst die Funktion **dir_namei** aufruft, um den i-node des Directorys zu erhalten, in dem sich der Name und der i-node der zu öffnenden Datei befindet. Nach diesem Aufruf führt **open_namei** eine Vielzahl von Prüfungen durch, ob z.B. die geforderte Zugriffsart für diese Datei erlaubt ist oder ob es sich um einen symbolischen Link handelt, der zunächst aufzulösen ist. Sind diese Prüfungen alle positiv, trägt **open_namei** den i-node der nun geöffneten Datei in *res_inode* ein und gibt 0 an **do_open** zurück.

Für den Fall, daß für die zu öffnende Datei Schreibzugriff gefordert wurde, verlangt **do_open** nun mit **get_write_access** Schreibrechte für diese Datei. Zudem füllt **do_open** die file-Struktur mit entsprechenden Standardwerten, wie z.B.

```
struct file *f;

f->f_pos = 0;
f->f_reada = 0;
f->f_op = inode->i_op->default_file_ops;
.....
```

Danach erst wird die Operation **open** aufgerufen, wenn sie definiert ist. In dieser Funktion finden die dateiartspezifischen Operationen statt. So wird z.B. für eine zeichenorientierte Gerätedatei die Funktion **chrdev_open** (in `fs/devices.h`) aufgerufen:

```
/*
 * Called every time a character special file is opened
 */
int chrdev_open(struct inode * inode, struct file * filp)
{
    int ret = -ENODEV;

    filp->f_op = get_chrfops(MAJOR(inode->i_rdev), MINOR(inode->i_rdev));
    if (filp->f_op != NULL){
        ret = 0;
        if (filp->f_op->open != NULL)
            ret = filp->f_op->open(inode, filp);
    }
    return ret;
}
```

Die Funktion **chrdev_open** ruft ihrerseits wieder die Funktion **get_chrfops** auf, die ebenfalls in `fs/devices.h` definiert ist:

```
struct file_operations * get_chrfops(unsigned int major, unsigned int minor)
{
    return get_fops (major, minor, MAX_CHRDEV, "char-major-%d", chrdevs);
}
```

Wie aus dieser Definition zu ersehen ist, ruft die Funktion **get_chrfops** ihrerseits die Funktion **get_fops** (auch in `fs/devices.h` definiert) auf:

```
/*
 * Return the function table of a device.
 * Load the driver if needed.
 */
static struct file_operations * get_fops(
    unsigned int major,
    unsigned int minor,
    unsigned int maxdev,
    const char *mangle,          /* String to use to build the module name */
    struct device_struct tb[])
{
```

```

struct file_operations *ret = NULL;

if (major < maxdev){
    .....
    ret = tb[major].fops;
}
return ret;
}

```

Aus dieser Aufrufhierarchie wird ersichtlich, daß sich die Fileoperationen für die entsprechenden Gerätetreiber in dem Array `chrdevs[]` befinden. Die Eintragung dieser Operationen erfolgte mit der Funktion **register_chrdev** (auch in `fs/devices.h` definiert) bei der Initialisierung der entsprechenden Gerätetreiber.

Waren nun alle diese **open**-Operationen erfolgreich, ist das Öffnen der entsprechenden Datei gelungen und die Funktion **do_open** liefert dem aufrufenden Prozeß den Filedeskriptor zurück.

5.12.7 Der Directorycache

Im Directorycache werden Directory-Einträge untergebracht, um schneller den Inhalt von Directories zu erfragen. Directory-Inhalte müssen z.B. bei jedem Öffnen einer Datei gelesen werden. Für Einträge in diesen Directorycache ist in `fs/dcache.c` die folgende Struktur definiert:

```

/*
 * The dir_cache_entry must be in this order
 */
struct dir_cache_entry {
    struct hash_list h;           /* Verwaltung der Hashlisten */
    kdev_t dc_dev;               /* Gerätenummer */
    unsigned long dir;           /* i-node-Nummer des Directorys */
    unsigned long version;       /* Directory-Version */
    unsigned long ino;           /* i-node-Nummer der Datei */
    unsigned char name_len;      /* Länge des Dateinamens */
    char name[DCACHE_NAME_LEN]; /* Dateiname */
    struct dir_cache_entry ** lru_head; /* Listenkopf */
    struct dir_cache_entry * next_lru; /* Nachfolger in Liste */
                                /* prev_lru; /* Vorgänger in Liste */
};

```

In diesem Directorycache werden nur Dateinamen eingetragen, deren Namen nicht länger als `DCACHE_NAME_LEN` (in `fs/dcache.c` auf **15** festgelegt) sind. Da die meisten benutzten Datei- oder Directory-Namen diese Länge nicht überschreiten, stellt dies keine große Einschränkung dar.

Der Directorycache ist als zweistufiger Cache organisiert, wobei jede Stufe nach dem LRU-Algorithmus (Last Recently Used) arbeitet. Neue Einträge werden zunächst am Ende der ersten Stufe hinzugefügt. Wird erneut auf einen Eintrag aus der ersten Stufe (cache hit) zugegriffen, so wird er aus dieser Stufe entfernt und am Ende der zweiten Stufe eingefügt.

Jede Stufe ist als eine doppelt verkettete Ringliste realisiert, die immer `DCACHE_SIZE` (in `fs/dcache.c` definiert) Einträge enthält.

```
static struct dir_cache_entry level1_cache[DCACHE_SIZE];
static struct dir_cache_entry level2_cache[DCACHE_SIZE];
```

Die Zeiger `level1_head` und `level2_head` zeigen auf das jeweils älteste Element in der Liste, welches also als nächstes überschrieben wird.

```
/*
 * The LRU-lists are doubly-linked circular lists, and do not change in size
 * so these pointers always have something to point to (after _init)
 */
static struct dir_cache_entry * level1_head;
static struct dir_cache_entry * level2_head;
```

Da die Komponente `lru_head` der Struktur `dir_cache_entry` ebenfalls auf das älteste Element in der jeweiligen Liste zeigt, ist jedem Cache-Eintrag bekannt, in welcher Stufe er sich gerade befindet.

Zum schnellen Auffinden eines Cache-Eintrags steht eine offene Hashtabelle zur Verfügung.

```
/*
 * The hash-queues are also doubly-linked circular lists, but the head is
 * itself on the doubly-linked list, not just a pointer to the first entry.
 */
struct hash_list {
    struct dir_cache_entry * next;
    struct dir_cache_entry * prev;
};

static struct hash_list hash_table[DCACHE_HASH_QUEUES];
```

Der Hashschlüssel (Index) wird dabei aus der Gerätenummer, der i-node-Nummer und dem Namen des Directorys ermittelt.

```
#define DCACHE_HASH_QUEUES 32
#define hash_fn(dev,dir,namehash) \
    ((HASHDEV(dev) ^ (dir) ^ (namehash)) % DCACHE_HASH_QUEUES)
```

Zum Zugriff auf den Directorycache stehen die beiden folgenden in `fs/dcache.c` definierten Funktionen zur Verfügung:

```
void dcache_add(struct inode * dir, const char * name, int len,
               unsigned long ino);

int dcache_lookup(struct inode * dir, const char * name, int len,
                 unsigned long * ino);
```

`dcache_add` trägt den Directoryeintrag `name` mit der Länge `len`, der sich im Directory `dir` befindet, in den Cache ein. Die Nummer `ino` ist die i-node-Nummer des Directoryeintrags. Befindet sich der neu einzutragende Eintrag bereits im Cache, wird er als jünger

in seiner Liste angeordnet, bevor sich diese Funktion beendet. Handelt es sich dagegen um einen neuen Eintrag, so wird dieser in jedem Fall in der ersten Stufe eingetragen. Dazu wird der älteste Eintrag, auf den `level1_head` zeigt, zunächst aus der Hashtabelle entfernt und dann mit den Daten des neuen Directoryeintrags überschrieben. Durch das Weiterpositionieren des Zeigers `level1_head` um einen Eintrag in der Ringliste, ist der neue Eintrag damit automatisch der jüngste in der Liste. Zum Schluß wird der neue Eintrag noch mit **`add_hash`** in die Hashtabelle eingetragen.

```
void dcache_add(struct inode * dir, const char * name, int len,
               unsigned long ino)
{
    struct hash_list * hash;
    struct dir_cache_entry * de;

    if (len > DCACHE_NAME_LEN)
        return;
    hash = hash_table + hash_fn(dir->i_dev, dir->i_ino, namehash(name, len));
    if ((de = find_entry(dir, name, len, hash)) != NULL) {
        de->ino = ino;
        update_lru(de);
        return;
    }
    de = level1_head;
    level1_head = de->next_lru;
    remove_hash(de);
    de->dc_dev = dir->i_dev;
    de->dir = dir->i_ino;
    de->version = dir->i_version;
    de->ino = ino;
    de->name_len = len;
    memcpy(de->name, name, len);
    add_hash(de, hash);
}
```

Zum Lesen von Einträgen im Directorycache steht die Funktion **`dcache_lookup`** zur Verfügung. Kann der Eintrag `name` nicht gefunden werden, liefert diese Funktion 0 zurück. Ist der Eintrag schon in der Stufe 1 vorhanden, wird er mit der Funktion **`move_to_level2`** in die Stufe 2 übertragen bzw. dort entsprechend umpositioniert, falls er in dieser Stufe 2 bereits existiert. Im Argument `ino` wird die i-node-Nummer des gefundenen Directoryeintrags zurückgeliefert.

```
int dcache_lookup(struct inode * dir, const char * name, int len,
                 unsigned long * ino)
{
    struct hash_list * hash;
    struct dir_cache_entry * de;

    if (len > DCACHE_NAME_LEN)
        return 0;
    hash = hash_table + hash_fn(dir->i_dev, dir->i_ino, namehash(name, len));
    de = find_entry(dir, name, len, hash);
```

```

if (!de)
    return 0;
*ino = de->ino;
move_to_level2(de, hash);
return 1;
}

```

5.12.8 Das ext2-Filesystem von Linux

Das ursprüngliche Filesystem von Linux war MINIX, was jedoch große Beschränkungen hatte: Partitionen konnten maximal 64 MByte groß sein und die Länge von Dateinamen war auf 14 Zeichen beschränkt.

Das Nachfolgesystem von MINIX war das ext-Filesystem, das bereits Partitionen bis zu 2 GByte und Dateinamen bis zu 255 Zeichen erlaubte. Mängel in der Geschwindigkeit und der Fragmentierung bewegten die Linux-Entwickler dazu, das ext-Filesystem weiterzuentwickeln und zu verbessern. Aus dieser Initiative entstand das ext2-Filesystem, das heute als das Standard-Filesystem von Linux gilt.

Struktur des ext2-Filesystems

Im ext2-Filesystem ist eine Partition in mehrere Blockgruppen unterteilt. Wie Abbildung 5.11 zeigt, enthält jede Blockgruppe sowohl eine Kopie des Superblocks als auch der i-node- und Datenblöcke.

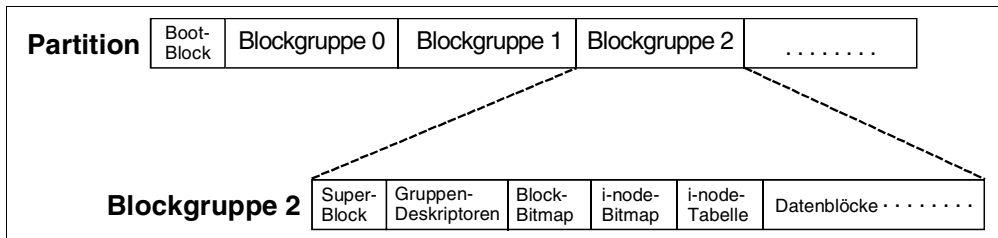


Abbildung 5.11: Die Struktur des ext2-Filesystems

Für diese Strukturierung einer Partition in mehreren Blockgruppen gibt es zwei Gründe:

► **Schnellerer Zugriff auf die Daten**

Da die Datenblöcke in der Nähe ihrer i-nodes und die i-nodes der Dateien in der Nähe ihrer Directory-i-nodes liegen, muß ein Schreib-/Lesekopf einer Festplatte viel weniger positioniert werden, was sich natürlich in einem schnelleren Zugriff bemerkbar macht.

► **Höhere Datensicherheit**

Da jede Blockgruppe den Superblock sowie Informationen über alle Blockgruppen enthält, ist eine Restaurierung der entsprechenden Partition auch bei einer Korruption des Superblocks in der ersten Blockgruppe möglich.

Superblock des ext2-Filesystems

Die Struktur des Superblocks ist in `<linux/ext2_fs.h>` wie folgt definiert:

```
struct ext2_super_block {
    __u32   s_inodes_count;      /* Inodes count */
    __u32   s_blocks_count;     /* Blocks count */
    __u32   s_r_blocks_count;   /* Reserved blocks count */
    __u32   s_free_blocks_count; /* Free blocks count */
    __u32   s_free_inodes_count; /* Free inodes count */
    __u32   s_first_data_block; /* First Data Block */
    __u32   s_log_block_size;   /* Block size (dual logarithmic) */
    __s32   s_log_frag_size;    /* Fragment size (dual logarithmic) */
    __u32   s_blocks_per_group; /* # Blocks per group */
    __u32   s_frags_per_group;  /* # Fragments per group */
    __u32   s_inodes_per_group; /* # Inodes per group */
    __u32   s_mtime;            /* Mount time */
    __u32   s_wtime;            /* Write time */
    __u16   s_mnt_count;        /* Mount count */
    __s16   s_max_mnt_count;    /* Maximal mount count */
    __u16   s_magic;            /* Magic signature */
    __u16   s_state;            /* File system state */
    __u16   s_errors;           /* Behaviour when detecting errors */
    __u16   s_minor_rev_level;  /* minor revision level */
    __u32   s_lastcheck;        /* time of last check */
    __u32   s_checkinterval;    /* max. time between checks */
    __u32   s_creator_os;       /* OS */
    __u32   s_rev_level;        /* Revision level */
    __u16   s_def_resuid;       /* Default uid for reserved blocks */
    __u16   s_def_resgid;       /* Default gid for reserved blocks */
    /*
     * These fields are for EXT2_DYNAMIC_REV superblocks only.
     *
     * Note: the difference between the compatible feature set and
     * the incompatible feature set is that if there is a bit set
     * in the incompatible feature set that the kernel doesn't
     * know about, it should refuse to mount the filesystem.
     *
     * e2fsck's requirements are more strict; if it doesn't know
     * about a feature in either the compatible or incompatible
     * feature set, it must abort and not try to meddle with
     * things it doesn't understand...
     */
    __u32   s_first_ino;        /* First non-reserved inode */
    __u16   s_inode_size;       /* size of inode structure */
    __u16   s_block_group_nr;   /* block group # of this superblock */
    __u32   s_feature_compat;   /* compatible feature set */
    __u32   s_feature_incompat; /* incompatible feature set */
    __u32   s_feature_ro_compat; /* readonly-compatible feature set */
    __u32   s_reserved[230];    /* Padding to the end of the block */
};
```

Bildlich lässt sich diese Struktur – wie in Abbildung 5.12 gezeigt – darstellen.

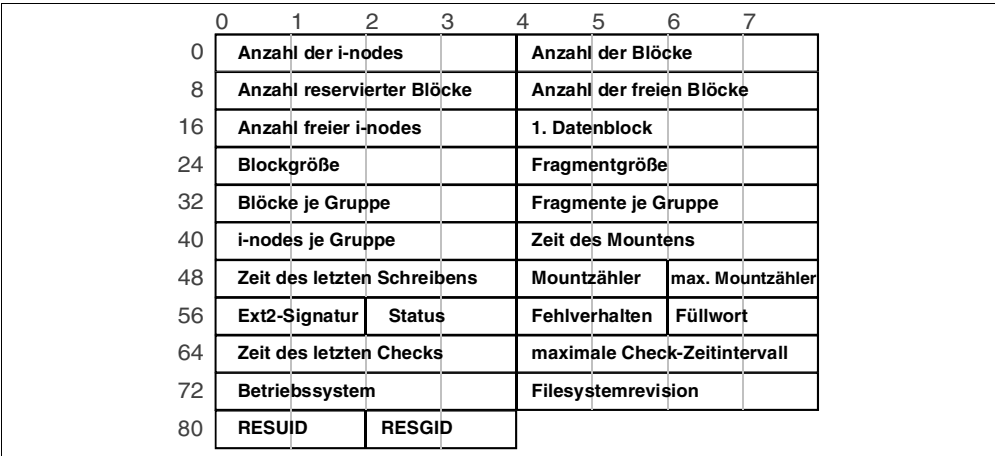


Abbildung 5.12: Struktur des ext2-Superblocks

Die verwendete Blockgröße ist nicht direkt, sondern als Zweierlogarithmus der Blockgröße angegeben. Die Blockgröße kann dann mit dem in `<linux/ext2_fs.h>` definierten Makro `EXT2_BLOCK_SIZE` ermittelt werden:

```
# define EXT2_BLOCK_SIZE(s) (EXT2_MIN_BLOCK_SIZE << (s)->s_log_block_size)
```

Der Superblock wird auf ein vielfaches von 1024 Byte aufgefüllt.

Nach dem Superblock folgen in einer Blockgruppe die Blockgruppendskriptoren.

Blockgruppendskriptoren

Diese umfassen 32 Byte und geben Informationen über die jeweilige Blockgruppe. Die Struktur eines Blockgruppendskriptors ist in `<linux/ext2_fs.h>` wie folgt definiert:

```
/*
 * Structure of a blocks group descriptor
 */
struct ext2_group_desc
{
    __u32    bg_block_bitmap;        /* Blocks bitmap block */
    __u32    bg_inode_bitmap;        /* Inodes bitmap block */
    __u32    bg_inode_table;         /* Inodes table block */
    __u16    bg_free_blocks_count;    /* Free blocks count */
    __u16    bg_free_inodes_count;    /* Free inodes count */
    __u16    bg_used_dirs_count;      /* Directories count */
    __u16    bg_pad;
    __u32    bg_reserved[3];
};
```

Bildlich lässt sich diese Struktur – wie in Abbildung 5.13 gezeigt – darstellen.

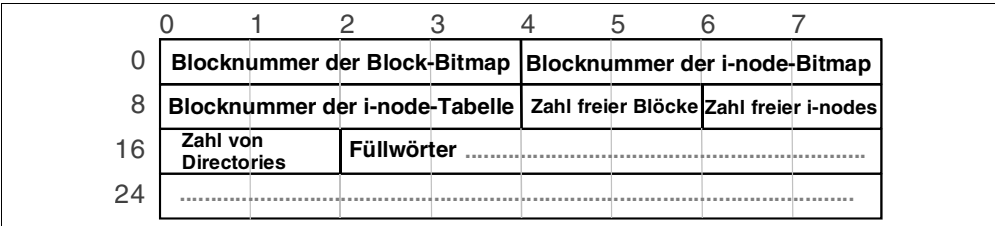


Abbildung 5.13: Struktur der Blockgruppenskriptoren im ext2-Filesystem

Die Blockgruppenskriptoren enthalten die folgenden Komponenten:

- **Blocknummer der Block-Bitmap**
Diese Blocknummer verweist auf die Block-Bitmap. Eine Block-Bitmap hat immer die Größe eines Blockes. Dies bedeutet, daß beispielsweise bei einer Blockgröße von 1024 Byte maximal 8192 Blöcke (1024*8 Bit) in einer Blockgruppe untergebracht werden können.
- **Blocknummer der i-node-Bitmap**
Diese Blocknummer verweist auf die i-node-Bitmap. Eine i-node-Bitmap hat immer die Größe eines Blockes.
- **Blocknummer der i-node-Tabelle**
Diese Blocknummer verweist auf die i-node-Tabelle.
- Zahl freier Blöcke und freier i-nodes
- **Zahl der Directories**
Diese Zahl wird beim Anlegen neuer Directories benötigt. Der dabei verwendete Algorithmus versucht, Directories möglichst gleichmäßig über die Blockgruppen zu verteilen, was bedeutet, daß ein neues Directory immer in der Blockgruppe mit der kleinsten Anzahl von Directories angelegt wird.

i-node-Tabelle

Die Struktur der i-node-Tabelle ist in <linux/ext2_fs.h> wie folgt definiert:

```
#define EXT2_NDIR_BLOCKS 12 /* 12 direkte Adressen von Blöcken */
#define EXT2_IND_BLOCK EXT2_NDIR_BLOCKS /* einfach indirekt */
#define EXT2_DIND_BLOCK (EXT2_IND_BLOCK + 1) /* zweifach indirekt */
#define EXT2_TIND_BLOCK (EXT2_DIND_BLOCK + 1) /* dreifach indirekt */
#define EXT2_N_BLOCKS (EXT2_TIND_BLOCK + 1) /* Anzahl der Adressen */
.....
.....
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
    __u16 i_mode; /* File mode */
    __u16 i_uid; /* Owner Uid */
    __u32 i_size; /* Size in bytes */
    .....
```

```

__u32 i_atime;      /* Access time      */
__u32 i_ctime;      /* Creation time    */
__u32 i_mtime;      /* Modification time */
__u32 i_dtime;      /* Deletion Time    */
__u16 i_gid;        /* Group Id         */
__u16 i_links_count; /* Links count      */
__u32 i_blocks;     /* Blocks count     */
__u32 i_flags;      /* File flags       */
union {
    struct {
        __u32 l_i_reserved1;
    } linux1;
    struct {
        __u32 h_i_translator;
    } hurd1;
    struct {
        __u32 m_i_reserved1;
    } masix1;
} osd1; /* OS dependent 1 */
__u32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
__u32 i_version; /* File version (for NFS) */
__u32 i_file_acl; /* File ACL */
__u32 i_dir_acl; /* Directory ACL */
__u32 i_faddr; /* Fragment address */
union {
    struct {
        __u8 l_i_frag; /* Fragment number */
        __u8 l_i_fsize; /* Fragment size */
        __u16 i_pad1;
        __u32 l_i_reserved2[2];
    } linux2;
    struct {
        __u8 h_i_frag; /* Fragment number */
        __u8 h_i_fsize; /* Fragment size */
        __u16 h_i_mode_high;
        __u16 h_i_uid_high;
        __u16 h_i_gid_high;
        __u32 h_i_author;
    } hurd2;
    struct {
        __u8 m_i_frag; /* Fragment number */
        __u8 m_i_fsize; /* Fragment size */
        __u16 m_pad1;
        __u32 m_i_reserved2[2];
    } masix2;
} osd2; /* OS dependent 2 */
};

```

Bildlich lässt sich diese Struktur – wie in Abbildung 5.14 gezeigt – darstellen.

	0	1	2	3	4	5	6	7
0	Dateiart/Rechte	Eigentümer (UID)		Dateigröße				
8	Zeit des letzten Zugriffs			Zeit der letzten i-node-Änderung				
16	Zeit der letzten Dateiänderung			Zeit des Löschens				
24	Gruppe (GID)		Linkzähler		Anzahl der Blöcke			
32	Dateiattribute/-flags			reserviert (systemabhängig)				
40	Adresse des 1. Datenblocks			Adresse des 2. Datenblocks				
48	Adresse des 3. Datenblocks			Adresse des 4. Datenblocks				
56	Adresse des 5. Datenblocks			Adresse des 6. Datenblocks				
64	Adresse des 7. Datenblocks			Adresse des 8. Datenblocks				
72	Adresse des 9. Datenblocks			Adresse des 10. Datenblocks				
80	Adresse des 11. Datenblocks			Adresse des 12. Datenblocks				
88	Adresse (einfach indirekt)			Adresse (zweifach indirekt)				
96	Adresse (dreifach indirekt)			Dateiversion				
104	Datei-ACL (für NFS)			Directory-ACL				
112	Fragment-Adresse							
120	reserviert (systemabhängig)							

Abbildung 5.14: Struktur eines i-node im ext2-Filesystem

Die i-node-Tabelle einer Blockgruppe belegt aufeinanderfolgende Blöcke, deren jeweilige Größe immer 128 Byte ist. Neben den schon erwähnten Informationen (wie z.B. Dateiart, Zugriffsrechte, User-ID des Eigentümers, Zeitmarken für die einzelnen Zugriffsarten usw.) enthält ein i-node im ext2-Filesystem noch weitere Informationen:

- **Zeitpunkt des Löschens der Datei**
wird für die Implementierung der Restaurierung gelöschter Dateien benötigt.
- **ACL-Einträge**
ACL steht für Access Control Lists und ist für detailliertere Zugriffsrechte vorgesehen. Da zur Zeit die ACLs noch nicht implementiert sind, werden nur die üblichen Unix-Zugriffsrechte unterstützt.
- **Betriebssystemabhängige Informationen**

Für Gerädateien und symbolische Links gelten die folgenden Besonderheiten:

- Bei Gerädateien zeigt die Adresse des 1. Datenblocks (`i_block[0]`) auf einen Block, der die Gerätenummer enthält.
- Bei symbolischen Links, die einen kurzen Namen (nicht länger als `EXT2_N_BLOCKS * sizeof(long)`) haben, wird dafür kein eigener Datenblock vergeudet, sondern der Name direkt in den Adreßeinträgen (Byteoffset 40-99) untergebracht. In diesem Fall enthält die Komponente `i_blocks` (Anzahl der Blöcke) den Wert 0. Sollte der Name länger sein, wird er im ersten Datenblock abgelegt.

Directories im ext2-Filesystem

Directories werden im ext2-Filesystem in Form einer einfach verketteten Liste organisiert. Jeder Directoryeintrag hat dabei die folgende (in `<linux/ext2_fs.h>` definierte) Struktur:

```
/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
    __u32    inode;        /* Inode number          */
    __u16    rec_len;      /* Directory entry length */
    __u16    name_len;     /* Name length           */
    char     name[EXT2_NAME_LEN]; /* File name          */
};
```

Die Komponente `inode` enthält die i-node-Nummer.

Die Komponente `rec_len`, die immer ein vielfaches von 4 (eventuell aufgerundet) ist, enthält die Länge des aktuellen Directoryeintrags. Hiermit läßt sich also der Beginn des nächsten Eintrags berechnen.

Die Komponente `name_len` enthält die Länge des Dateinamens.

Das Löschen eines Directoryeintrags erfolgt durch das Nullsetzen der i-node-Nummer und das Aushängen aus der verketteten Liste, was bedeutet, daß der vorherige Directoryeintrag sich nur verlängert. So ist keinerlei Verschiebung innerhalb eines Directories notwendig. Ein so freigegebener Speicherplatz kann später wieder für neue Directoryeinträge verwendet werden.

Das folgende Programm `dirlese.c` liest den Inhalt von Directories byteweise und gibt dann immer die i-node-Nummer mit dem zugehörigen Dateinamen aus.

```
#include <stdio.h>
#include <fcntl.h>
#include <dirent.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>

#define PUFFER_GROESSE 1<<16

int main(int argc, char *argv[])
{
    int          f, ac=argc,
                i, j, fd, laenge, rlen, neu_i;
    char         *av[PUFFER_GROESSE];
    unsigned char buffer[PUFFER_GROESSE];
    off_t        zgr = 0;
    unsigned long inode=0, offset=0;
    unsigned short rec_len=0;
```

```

for (f=1; f<argc; f++)
    av[f] = argv[f];
if (argc == 1) {
    av[1] = ".";
    ac++;
}
for (f=1; f<ac; f++) {
    printf("Directory '%s':\n", av[f]);
    if ( (fd = open(av[f], O_RDONLY)) < 0 ||
        (laenge = getdirentries(fd, buffer, PUFFER_GROESSE, &zgr)) < 0 ) {
        perror("....Fehler");
        continue;
    }
    i=0;
    while (i < laenge) {
        inode = offset = rec_len = rlen = 0;
        for (j=3; j>=0; j--)
            inode = (inode<<8)+ buffer[i+j];
        i += 4; rlen += 4;
        for (j=3; j>=0; j--)
            offset = (offset<<8)+ buffer[i+j];
        i += 4; rlen += 4;
        for (j=1; j>=0; j--)
            rec_len = (rec_len<<8)+ buffer[i+j];
        i += 2; rlen += 2;
        printf("%15ld ", inode);
        neu_i = i + rec_len-rlen;
        for (j=rlen; buffer[i] != 0; j++)
            printf("%c", buffer[i++]);
        printf("\n");
        i = neu_i;
    }
    close(fd);
}
}

```

Nachdem wir das Programm kompiliert und gelinkt haben

```
cc -o dirlese dirlese.c
```

könnte sich der folgende Ablauf ergeben:

\$ pwd	
...../subdir	<i>Man befindet sich gerade im Subdirectory <code>subdir</code></i>
\$ dirlese .. . /etc	<i>Gib die i-node-Nummern zum Parent Dir., Work. Dir. und zum Dir. /usr aus</i>
Directory '..':	
24134 .	
12325 ..	
24135 datei1	
24136 datei2	
24137 datei3	
24138 datei4	
Directory '':	
24139 .	

```

24134 ..
Directory '/etc':
20081 .
    2 ..
20082 fstab
20215 mtab
20211 passwd
20111 group
20087 DIR_COLORS
20098 motd
    ::::::::::
20125 hosts.allow
20126 hosts.deny
20127 hosts.equiv
20128 hosts.lpd
20129 inetd.conf
20130 networks
20131 protocols
20132 rpc
$

```

Blockallokierung im ext2-Filesystem

Um eine zu große Fragmentierung (Zersplitterung) der Datenblöcke von Dateien – bedingt durch das ständige Löschen und Neuanlegen von Dateien – im ext2-Filesystem zu verhindern, verwendet das ext2-Filesystem zwei spezielle Strategien beim Allokieren neuer Datenblöcke:

► Neue Datenblöcke werden immer in der Nähe des Zielblocks gesucht.

Falls dieser Zielblock frei ist, wird er allokiert. Ansonsten wird versucht, innerhalb eines Bereiches von 32 Blöcken (davor und danach) einen freien Block zu finden und zu allokieren. Ist auch dies nicht möglich, wird versucht, zumindest einen freien Block in derselben Blockgruppe wie der Zielblock zu finden und zu allokieren. Was ein Zielblock ist, wird nachfolgend geklärt.

► Preallokation

Wurde ein freier Block gefunden, werden bis zu acht folgende Blöcke, wenn diese frei sind, vorgemerkt, um sie mit weiteren Blöcken derselben Datei zu belegen. Wird die Datei geschlossen, werden die restlichen noch vorgemerkten und nicht benutzten Blöcke wieder freigegeben. So stellt man sicher, daß möglichst viele Datenblöcke einer Datei zusammen in einem Cluster liegen. Möchte man diese Preallokation von Blöcken abschalten, muß man nur die Definition der Konstante `EXT2_PREALLOCATE` aus der Datei `<linux/ext2_fs.h>` entfernen.

Wenn n die relative Nummer des zu allozierenden Blocks in der Datei ist und b die logische Blocknummer, dann legt die entsprechende Allokierungsroutine den Zielblock entsprechend dem folgenden Pseudocode fest:

```
zielblock = 0;
```

```
if (relative Nummer des zuletzt allokierten Blocks == n-1)
```

```

    zielblock = b+1;
else {
    for (i=n-1; i>=0; i--) { /* alle bisher vorhandenen Blöcke der Datei, */
                            /* angefangen beim Block mit Nummer n-1, danach */
                            /* durchsuchen, ob ihnen logische Blöcke */
                            /* zugewiesen sind (also kein Loch sind). */
        if (logische Blocknummer des i. ten Blocks der Datei != 0) {
            zielblock = logische Blocknummer des i-ten Block;
            break;
        }
    }
    if (zielblock == 0)
        zielblock = Blocknummer des ersten Blocks der Blockgruppe,
                       in der der i-node der Datei liegt;
}

```

Erweiterungen des ext2-Filesystems

Das ext2-Filesystem kennt gegenüber normalen Unix-Filesystemen zusätzliche Dateiattribute, die in `<linux/ext2_fs.h>` wie folgt definiert sind:

```

/*
 * Inode flags
 */

#define EXT2_SECRM_FL      0x00000001 /* Sicheres Löschen
    Besitzt eine Datei dieses Attribut, werden ihre Daten zunächst
    mit zufälligen Werten überschrieben, bevor sie mit der Funktion
    truncate freigegeben werden. So kann nach dem Löschen der Datei
    ihr Inhalt nicht wieder restauriert werden. */

#define EXT2_UNRM_FL      0x00000002 /* Undelete (nicht implementiert)
    Dieses Attribut ist für die Implementierung der Restauration
    von gelöschten Dateien vorgesehen. */

#define EXT2_COMPR_FL      0x00000004 /* Komprimierte Datei (nicht implem.)
    Dieses Attribut soll anzeigen, daß die Datei komprimiert ist. */

#define EXT2_SYNC_FL      0x00000008 /* Synchrones Schreiben
    Besitzt eine Datei dieses Attribut, wird jedes Schreiben
    synchron (physikalisch) - ohne eine Zwischenspeicherung
    im Puffercache - durchgeführt. */

#define EXT2_IMMUTABLE_FL 0x00000010 /* Nicht änderbare Datei
    Besitzt eine Datei dieses Attribut, kann sie weder gelöscht noch
    kann ihr Inhalt geändert werden. Ebenso ist kein Umkopieren und
    auch kein Anlegen eines Hardlinks auf diese Datei möglich.
    Handelt es sich bei der Datei um ein Directory, kann deren
    Inhalt nicht verändert werden, was heißt, daß keine neue
    Dateien dort angelegt und auch keine Dateien in diesem Directory
    gelöscht werden können. Der Inhalt der Dateien in diesem
    Directory kann jedoch beliebig geändert werden. */

```

```
#define EXT2_APPEND_FL      0x00000020 /* Für Datei ist nur Anhängen erlaubt
    Besitzt eine Datei dieses Attribut, kann sie nicht gelöscht,
    nicht umkopiert werden und es ist auch kein Anlegen eines Hardlinks
    auf diese Datei möglich. Anders als beim vorherigen Attribut ist
    dagegen ein Anhängen (Schreiben am Dateiende) erlaubt.
    Handelt es sich bei der Datei um ein Directory, können in diesem
    zwar keine Dateien gelöscht werden, aber - anders als
    beim vorherigen Attribut - können sehr wohl neue Dateien
    angelegt werden, welche das EXT2_APPEND_FL-Attribut erben.      */

#define EXT2_NODUMP_FL      0x00000040 /* keine Archivierung für diese Datei
    Dieses Attribut wird vom Kern nicht verwendet. Dieses Attribut
    kann für Dateien gesetzt werden, die für einen Backup nicht
    benötigt werden.                                              */

#define EXT2_NOATIME_FL      0x00000080 /* keine Aktualisierung der Zugriffszeit
    Besitzt eine Datei dieses Attribut, wird bei einem Zugriff
    auf sie die Zugriffszeit nicht aktualisiert.                  */
```

Diese Attribute können mit dem Kommando **chattr** geändert und mit dem Kommando **lsattr** aufgelistet werden. Mehr Informationen zu diesen Kommandos lassen sich mit dem **man**-Kommando erfragen.



5.13 Übung

5.13.1 Ermitteln der Größe von Dateien

Erstellen Sie ein Programm `groesse.c`, das sowohl die einzelnen Größen als auch die Gesamtgröße der auf der Kommandozeile angegebenen Dateien ausgibt. Bei der Ausgabe soll es sowohl die wirkliche Anzahl von Bytes als auch den durch diese Datei belegten Speicherplatz (Blöcke) ausgeben.

Ein Beispiel für den Ablauf dieses Programms ist:

```
$ groesse *.c
accesdem.c:      902 (      1024)
chmodemo.c:      658 (      1024)
  cptime.c:     1403 (     2048)
dateiart.c:      928 (      1024)
  devnr.c:       837 (      1024)
  fehler.c:     2783 (     3072)
  getcwd.c:      453 (      1024)
lochgen2.c:      680 (      1024)
  mchdir.c:      300 (      1024)
symlink.c:      953 (      1024)
  tree.c:     4668 (     5120)
  tree2.c:     2489 (     3072)
```



```

umaskdem.c:      733 (      1024)
zeitaend.c:     2877 (      3072)
-----
Gesamtgroesse:   20664 (      25600)   14 Dateien
$

```

5.13.2 Ausgeben der Attribute von Dateien

Erstellen Sie ein Programm `datattr.c`, das die Attribute (`stat`-Struktur) der auf der Kommandozeile angegebenen Dateien ausgibt.

Ein Beispiel für den Ablauf dieses Programms ist:

```

$ datattr . tree.c /
-----
Dateiart      : Directory
Zugriffsrechte : rwxr-xr-x
inode-Nummer  : 10450
Geraetenummern : dev =  8/ 3
Anzahl der Links : 2
UID           : 2021
GID           : 1
Dateigroesse  : 1024
Letzter Zugriff : Fri Jun 23 10:01:39 1995
Letzte Aenderung : Wed Jun 21 10:52:00 1995
Letzte inode-Aenderung: Wed Jun 21 10:52:00 1995
----- tree.c -----
Dateiart      : Regulaere Datei
Zugriffsrechte : rw-r--r--
inode-Nummer  : 10475
Geraetenummern : dev =  8/ 3
Anzahl der Links : 1
UID           : 2021
GID           : 1
Dateigroesse  : 4668
Letzter Zugriff : Thu Jun 22 16:25:22 1995
Letzte Aenderung : Wed Jun 21 09:48:29 1995
Letzte inode-Aenderung: Wed Jun 21 09:48:29 1995
----- / -----
Dateiart      : Directory
Zugriffsrechte : rwxr-xr-x
inode-Nummer  : 2
Geraetenummern : dev =  8/ 3
Anzahl der Links : 25
UID           : 0
GID           : 0
Dateigroesse  : 2048
Letzter Zugriff : Fri Jun 23 10:00:01 1995
Letzte Aenderung : Thu Jan 12 18:05:50 1995
Letzte inode-Aenderung: Thu Jan 12 18:05:50 1995
$

```

5.13.3 Makro S_ISLNK für SVR4

In Tabelle 5.1 ist angegeben, daß in SVR4 kein Makro `S_ISLNK` existiert, mit dem geprüft werden kann, ob ein symbolischer Link vorliegt. SVR4 unterstützt aber symbolische Links und definiert in `<sys/stat.h>` auch die Konstante `S_IFLNK`. Mit welcher Angabe könnte nun das in SVR4 fehlende Makro `S_ISLNK` nachgebildet werden?

5.13.4 Ändern der Zugriffsrechte existierender Dateien mit `creat` oder `open`

Ist es möglich, daß man mit **`open`** oder **`creat`** die Zugriffsrechte bereits existierender Dateien ändern kann? Um dies zu testen, legen Sie zunächst die beiden Dateien `um1` und `um2` an, bevor sie das Programm 5.4 (`umaskdem.c`) aufrufen, das diese beiden Dateien mit **`creat`** und eigenen Zugriffsrechten neu anlegt.

5.13.5 Relatives Ändern der Zugriffs- und Modifikationszeiten von Dateien

Erstellen Sie ein Programm `zeitaend.c`, das ein relatives Ändern der aktuellen Zugriffs- und Modifikationszeiten ermöglicht. Die relative Zeit soll dabei auf der Kommandozeile in Tagen (**t**), Stunden (**h**), Minuten (**m**) und Sekunden (**s**) angegeben werden können. Nachfolgend sind mögliche Abläufe dieses Programms `zeitaend.c` und die daraus resultierenden Auswirkungen gezeigt.

```
$ ls -l groesse.c
-rw-r--r--  1 hh      bin           811 Jun 23  1995 groesse.c
$ zeitaend +100t groesse.c           [Zeiten um 100 Tage weitersetzen]
...groesse.c (+8640000sek = 100tage,0sek)
$ ls -l groesse.c
-rw-r--r--  1 hh      bin           811 Oct  1  1995 groesse.c
$ zeitaend -100t-2h-20m-10s groesse.c [Zeiten um 100 Tage,2 Std,20 Min.,10 Sek. vor]
...groesse.c (-8648410sek = 100tage,2std,20min,10sek)
$ ls -l groesse.c
-rw-r--r--  1 hh      bin           811 Jun 23 11:09 groesse.c
$ zeitaend -1t-2h-20m-10s groesse.c   [Zeiten um 1 Tag,2 Std,20 Min.,10 Sek. vor]
...groesse.c (-94810sek = 1tage,2std,20min,10sek)
$ ls -l groesse.c
-rw-r--r--  1 hh      bin           811 Jun 22 08:49 groesse.c
$
```

5.13.6 `unlink` und Zeit der letzten i-node-Änderung

Verändert ein **`unlink`**-Aufruf die Zeit der letzten i-node-Änderung für eine Datei?

5.13.7 Maximale Tiefe eines Directory-Baums

Hier wird die Frage gestellt, ob Unix ein Limit bezüglich der Tiefe eines Directory-Baums kennt. Um dies herauszufinden, sollten Sie ein Programm `treetief.c` erstellen, das in einer Schleife ein Directory kreiert und dann in dieses neue Directory wechselt, dort wie-

der ein Directory anlegt und dorthin wechselt usw. Diese beiden Schritte (Anlegen und Wechseln des Directories) sollten in der Schleife z.B. 50 oder auch mehr Mal wiederholt werden. In der tiefsten Ebene soll dann noch eine Datei angelegt werden. In jedem Fall sollte die Länge des absoluten Pfadnamens der untersten Ebene dieses Directory-Baums größer als `PATH_MAX` sein. Kann man dann noch mit ***getcwd*** den Pfadnamen dieser Ebene erfragen?

5.13.8 Root-Directory eines Prozesses

Jeder Prozeß besitzt ein Root-Directory, das für absolute Pfadnamen verwendet wird. Dieses Root-Directory kann mit der Funktion ***chroot*** gewechselt werden (siehe auch Manpages). ***chroot*** kann jedoch nur von privilegierten Benutzern (wie Superuser) verwendet werden. Auch ist zu beachten, daß nach einem Wechseln des Root-Directories mit ***chroot*** ein Zurückwechseln in das ursprüngliche Root-Directory nicht mehr möglich ist.

Können Sie sich Anwendungsfälle vorstellen, bei denen diese Funktion gebraucht werden könnte?

5.13.9 Suchen eines Dateinamens im Directory-Baum

Erstellen Sie ein Programm `woist.c`, das nach einem Dateinamen im Directory-Baum sucht. Der zu suchende Dateiname ist als erstes Argument auf der Kommandozeile anzugeben. Sind keine weitere Argumente angegeben, so wird der ganze Directory-Baum (ab Root-Directory) durchsucht. Soll nur in bestimmten Directories gesucht werden, so sind diese als weitere Argumente anzugeben.

Mögliche Abläufe dieses Programms `woist.c` sind z.B.:

```
$ woist file           [Ab Root-Directory alle Directories nach Datei file durchsuchen]
/usr/bin/file
/usr/lib/tclX/7.3a/help/tcl/files/file
$ woist tree.c $HOME  [Im Home-Directory nach Datei tree.c suchen]
/home/hh/sysprog/src/kap5/tree.c
$
```


6 Informationen zum System und seinen Benutzern

Quidam fallere docuerunt, dum timent falli.

Seneca

(Manche haben anderen Betrügen beigebracht, weil sie fürchteten, betrogen zu werden)

In einem Unix-System gibt es viele Dateien, die von einzelnen Systemkommandos benötigt werden. Dabei sind `/etc/passwd` und `/etc/group` wohl die herausragenden Dateien. So wird z.B. die Paßwortdatei `/etc/passwd` immer dann gebraucht, wenn sich ein Benutzer am System anmeldet oder auch jedesmal, wenn ein Benutzer **ls -l** aufruft, damit dieser Aufruf den Loginnamen der Besitzer der einzelnen Dateien ermitteln und ausgeben kann. In diesem Kapitel werden Funktionen vorgestellt, die es ermöglichen, sich Informationen aus der Paßwortdatei, der Gruppendatei oder aus den Netzwerkdateien zu beschaffen. Daneben beschreibt es auch noch Funktionen, mit denen Informationen zum lokalen System und seinen Benutzern erfragt werden können.

6.1 Informationen aus der Paßwortdatei

6.1.1 Paßwortdatei `/etc/passwd`

Die Paßwortdatei `/etc/passwd`, die in POSIX.1 als Benutzerdatenbank (*user database*) bezeichnet wird, enthält die in Tabelle 6.1 aufgeführten Felder. Diese Felder sind als Komponenten in der `passwd`-Struktur (`struct passwd`) enthalten. Diese Struktur ist in der Headerdatei `<pwd.h>` definiert.

	<i>Komponente in struct passwd</i>	<i>POSIX.1</i>
Benutzername	<code>char *pw_name</code>	x
Verschlüsseltes Paßwort	<code>char *pw_passwd</code>	
Benutzernummer (UID)	<code>uid_t pw_uid</code>	x
Gruppennummer (GID)	<code>gid_t pw_gid</code>	x
Kommentarfeld	<code>char *pw_gecos</code>	
Logindirectory	<code>char *pw_dir</code>	x
Loginshell	<code>char *pw_shell</code>	x

Tabelle 6.1: Felder in der Datei `/etc/passwd`

Wie Tabelle 6.1 zeigt, definiert POSIX.1 nur fünf der sieben Felder. Die anderen beiden Felder werden zusätzlich von SVR4 angeboten.

Seit den Anfängen von Unix befinden sich die in Tabelle 6.1 angegebenen Benutzerinformationen in der ASCII-Datei `/etc/passwd`. Jede Zeile in dieser Datei beschreibt einen Benutzer und enthält die in Tabelle 6.1 beschriebenen Felder, die mit Doppelpunkt (:) voneinander getrennt sind. Ein Ausschnitt aus `/etc/passwd` kann z.B. folgendes Aussehen haben:

```
root:x:0:1:Superuser:/:/bin/sh
daemon:x:1:1:0000-Admin(0000):/:
nobody:*:60001:60001:/:
hh:x:178:14:Helmut HeroId:/home/hh:/bin/ksh
```

Hinweis

Das 2. Feld enthielt früher das verschlüsselte Paßwort, das mit einem Einweg-Verschlüsselungsalgorithmus verschlüsselt wurde. Heute steht das verschlüsselte Paßwort in der nur für privilegierte Benutzer lesbaren Datei `/etc/shadow`. Der momentan benutzte Verschlüsselungsalgorithmus generiert immer ein Paßwort, das 13 Zeichen lang ist und Kleinbuchstaben, Großbuchstaben, Ziffern, Punkt (.) oder Slash (/) enthält. Da der Eintrag für den Benutzer `nobody` einen Stern (*) enthält, gibt es für diesen Benutzer kein Paßwort. Dieser Loginname `nobody` kann von Netzwerk-Servern benutzt werden, um sich am lokalen System mit einer UID und GID anzumelden, die keinerlei Privilegien hat. Anmelden unter diesem Loginnamen ermöglicht also nur Zugriff auf Dateien, die für jedermann (`world`, `others`) lesbar oder beschreibbar sind, was bedeutet, daß auf dem lokalen System keine Dateien existieren, die einem Benutzer mit der UID 60001 und GID 60001 gehören.

Einige Felder in einer `/etc/passwd`-Zeile (Paßwort, Kommentar, Loginshell) können auch leer sein, was im einzelnen folgendes bedeutet:

- Leeres Paßwortfeld: es ist kein Paßwort vorhanden, was aus Sicherheitsgründen vermieden werden sollte.
- Leeres Kommentarfeld: es ist kein Kommentar (meist der richtige Benutzername) vorhanden.
- Leeres Loginshell-Feld: es ist keine Loginshell vorhanden. In diesem Fall wird als Loginshell die Bourne-Shell **/bin/sh** verwendet.

SVR4 bietet das **finger**-Kommando an, das zusätzliche Information zu einem Benutzer ausgibt. Dazu liest es das Kommentarfeld in der Paßwortdatei, das hierfür folgende durch Komma getrennte Informationen enthalten kann.

Benutzername,Büroadresse,Dienstl. Telefonnr,Private Telefonnr

Falls sich dabei im Benutzernamen noch ein & befindet, so wird dieses & von **finger** durch den Loginnamen (groß geschrieben) ersetzt.

6.1.2 **getpwuid und getpwnam – Erfragen eines /etc/passwd-Eintrags über UID bzw. Loginnamen**

Um mittels UID oder Loginnamen einen Eintrag aus der Paßwortdatei zu erfragen, stehen die beiden POSIX.1-Funktionen **getpwuid** und **getpwnam** zur Verfügung.

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *loginname);
```

beide geben zurück: struct passwd-Zeiger (bei Erfolg); NULL-Zeiger bei Fehler

Beide Funktionen geben einen Zeiger auf struct passwd zurück. Diese Struktur ist normalerweise in diesen Funktionen als lokale **static**-Variable definiert, so daß jeder neue Aufruf dazu führt, daß ihr alter Inhalt überschrieben wird.

getpwuid wird z.B. vom Kommando **ls -l** benutzt, um über die im i-node enthaltene UID den entsprechenden Loginnamen herauszufinden.

getpwnam wird z.B. vom Kommando **login** benutzt, um über den eingegebenen Loginnamen die zugehörigen Benutzerdaten zu erfragen.

6.1.3 **getpwent, setpwent und endpwent – Sukzessives Erfragen aller /etc/passwd-Einträge**

Um nacheinander alle Einträge aus einer Paßwortdatei zu erfragen, stehen die drei Funktionen **getpwent**, **setpwent** und **endpwent** zur Verfügung.

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent(void);
                gibt zurück: struct passwd-Zeiger (bei Erfolg); NULL-Zeiger bei Dateiende oder Fehler

void setpwent(void);
void endpwent(void);
```

getpwent

Die Funktion **getpwent** liefert den nächsten Eintrag aus der Paßwortdatei (als struct passwd-Zeiger). Diese Struktur ist normalerweise in dieser Funktion als lokale **static**-Variable definiert, so daß jeder neue Aufruf dazu führt, daß der Inhalt dieser Strukturvariablen überschrieben wird.

Beim ersten Aufruf von **getpwent** wird die Paßwortdatei geöffnet und der erste Eintrag zurückgeliefert. Jeder weitere Aufruf dieser Funktion liefert dann den nächsten Eintrag aus der geöffneten Paßwortdatei. Die Reihenfolge, mit der die Einträge aus `/etc/passwd` gelesen werden, kann beliebig sein, da manche Systeme sich die Paßwortdatei intern in Form einer Hashtabelle halten.

setpwent

Die Funktion **setpwent** öffnet die Datei `/etc/passwd`, wenn sie nicht schon geöffnet ist, und setzt den Lesezeiger auf den Anfang dieser Datei.

endpwent

Die Funktion **endpwent** schließt die entsprechenden Paßwortdateien. Wenn man mit **getpwent** arbeitet, so sollte man nach Abschluß des Arbeitens mit der Paßwortdatei immer die Funktion **endpwent** aufrufen, um die Paßwortdatei zu schließen. So stellt man sicher, daß bei einem erneuten Zugriff auf die Paßwortdatei mit **getpwent** diese wieder neu geöffnet und von Anfang gelesen wird.

Hinweis

Die drei Funktionen **getpwent**, **setpwent** und **endpwent** werden von SVR4 angeboten, sind aber nicht Bestandteil von POSIX.1

Beispiel

Suchen eines Strings in Loginnamen- und Kommentarfeldern von `/etc/passwd`

```
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    struct passwd *zgr;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s string", argv[0]);

    setpwent(); /*-- Zuruecksetzen der Paßwortdatei (auf Nr. Sicher gehen) */

    while ( (zgr=getpwent()) != NULL) {
        if (strstr(zgr->pw_name, argv[1]) || strstr(zgr->pw_gecos, argv[1])) {
            printf("%s:%s:%d:%d:%s:%s\n",
                zgr->pw_name, zgr->pw_passwd, zgr->pw_uid, zgr->pw_gid,
                zgr->pw_gecos, zgr->pw_dir, zgr->pw_shell);
        }
    }
}
```



```

    endpwent();

    exit(0);
}

```

Programm 6.1 (**pwsuch.c**): Durchsuchen von Loginnamen und Kommentaren in `/etc/passwd`

Beispiel

Implementierung der Funktion `getpwuid`

```

#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid)
{
    struct passwd *pw;

    while (pw = getpwent()) {
        if (pw->pw_uid == uid) {
            endpwent();
            return(pw);
        }
    }
    endpwent();
    return(NULL);
}

```

Programm 6.2 (**getpwuid.c**): Implementierung von `getpwuid` mit Hilfe von `getpwent`

6.1.4 /etc/shadow

Seit SVR4 wird das Paßwort nicht mehr in der für jedermann lesbaren Datei `/etc/passwd` hinterlegt, denn dies war eine nicht unerhebliche Sicherheitslücke in Unix-Systemen. Wenn auch Entschlüsseln der dort öffentlich zugänglichen Paßwörter so gut wie unmöglich war, so benutzten Hacker doch diese Paßwörter, um in Unix-Systeme einzubrechen.

Sie wendeten einen ganz einfachen, aber wirkungsvollen Trick an. Sie griffen auf das Kommando **crypt** zurück, von dem sie wußten, daß es den gleichen Verschlüsselungsalgorithmus benutzt, den auch das System zum Verschlüsseln der Paßwörter verwendet. Dieses Kommando **crypt** riefen sie mit einer Vielzahl von Wörtern auf, wie z.B. alle Wörter aus der unter Unix vorhandenen `spell`-Datei und ließen sich zu allen diesen Wörtern die zugehörigen Verschlüsselungen in eine Datei schreiben. Nun mußten sie diese Verschlüsselungen nur noch mit den verschlüsselten Paßwörtern aus `/etc/passwd` vergleichen. Fanden sie eine Übereinstimmung, so kannten sie das unverschlüsselte Paßwort, da sie ja wußten, aus welchem ursprünglichen Wort diese Verschlüsselung entstanden war. Wenn Benutzer – was sie leider oft nicht tun – Sonderzeichen in ihre Paßwörter mischen würden, wie z.B. `jim4son` oder `drei4.1`, so würde dies das Knacken der Paßwörter mit dieser Methode ganz erheblich erschweren.

In SVR4 schloß man diese Sicherheitslücke, indem man das Paßwort nicht mehr in der weiterhin für jedermann lesbaren Datei `/etc/passwd`, sondern in der nur noch für privilegierte Benutzer (wie Superuser) lesbaren Datei `/etc/shadow` hinterlegt. `/etc/shadow` enthält dabei neben dem Loginnamen und dem verschlüsselten Paßwort meist weitere Informationen, wie z.B. das Datum, an dem das Paßwort ungültig wird.

Hinweis

Die Funktionen für den Zugriff auf die Daten in `/etc/shadow` sind bei SVR4 in der Headerdatei `<shadow.h>` deklariert und in der Manualpage `getspent(3)` beschrieben.

In BSD-Unix wird bei den Funktionen ***getpwnam*** oder ***getpwuid*** das verschlüsselte Paßwort automatisch aus `/etc/shadow` geholt und in die Strukturkomponente `pw_passwd` geschrieben, wenn die effektive UID des Aufrufers 0 (Superuser) ist.

6.2 Informationen aus der Gruppendatei

6.2.1 Gruppendatei `/etc/group`

Die Gruppendatei `/etc/group`, die in POSIX.1 als Gruppendatenbank (*group database*) bezeichnet wird, enthält die in Tabelle 6.2 aufgeführten Felder. Diese Felder sind als Komponenten in der `group`-Struktur (`struct group`) enthalten. Diese Struktur ist in der Headerdatei `<grp.h>` definiert.

	<i>Komponente in struct group</i>	<i>POSIX.1</i>
Gruppenname	<code>char *gr_name</code>	x
Verschlüsseltes Paßwort	<code>char *gr_passwd</code>	
Gruppennummer (GID)	<code>gid_t gr_gid</code>	x
Array von zur Gruppe gehörigen Loginnamen	<code>char **gr_mem</code>	x

Tabelle 6.2: Felder in der Datei `/etc/group`

Wie Tabelle 6.2 zeigt, definiert POSIX.1 nur drei der vier Felder. Das andere Feld `gr_passwd` wird zusätzlich von SVR4 angeboten. Die Komponente `gr_mem` ist ein Array von Loginnamen, wobei der letzte Eintrag ein NULL-Zeiger ist.

6.2.2 ***getgrgid*** und ***getgrnam*** – Erfragen eines `/etc/group`-Eintrags über GID bzw. Loginnamen

Um mittels einer GID oder einem Gruppennamen einen Eintrag aus der Gruppendatei zu erfragen, stehen die beiden POSIX.1-Funktionen ***getgrgid*** und ***getgrnam*** zur Verfügung.

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *gruppname);
```

beide geben zurück: struct group-Zeiger (bei Erfolg); NULL-Zeiger bei Fehler

Beide Funktionen geben einen Zeiger auf struct group zurück. Diese Struktur ist normalerweise in diesen Funktionen als lokale **static**-Variable definiert, so daß jeder neue Aufruf dazu führt, daß ihr alter Inhalt überschrieben wird.

6.2.3 getgrent, setgrent und endgrent – Sukzessives Erfragen aller /etc/group-Einträge

Um nacheinander alle Einträge aus der Gruppendatei zu erfragen, stehen die drei Funktionen **getgrent**, **setgrent** und **endgrent** zur Verfügung.

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrent(void);
                        gibt zurück: struct group-Zeiger (bei Erfolg); NULL-Zeiger bei Dateiende oder Fehler

void setgrent(void);
void endgrent(void);
```

Diese drei Funktionen entsprechen weitgehend ihren Gegenstücken für die Paßwortdatei (siehe Kapitel 6.1 bei **getpwent**, **setpwent** und **endpwent**), nur beziehen sie sich eben nicht auf /etc/passwd, sondern auf /etc/group:

- ▶ **setgrent** öffnet die Gruppendatei, wenn sie nicht schon geöffnet ist, und setzt den Lesezeiger auf den Anfang dieser Datei.
- ▶ **getgrent** liefert den nächsten Eintrag aus der Gruppendatei, wobei diese Funktion eventuell diese Datei erst öffnet, sollte sie noch nicht offen sein.
- ▶ **endgrent** schließt die Gruppendatei.

Hinweis

Die drei Funktionen **getgrent**, **setgrent** und **endgrent** werden von SVR4 angeboten, sind aber nicht Bestandteil von POSIX.1

6.2.4 **getgroups, setgroups und initgroups – Erfragen und Setzen von Zusatz-GIDs**

Es ist möglich, daß ein Benutzer Mitglied mehrerer Gruppen ist. Man denke z.B. an einen Benutzer, der gleichzeitig in mehreren Projekten mitarbeitet und somit Mitglied in mehreren Projektgruppen sein muß.

In früheren Unix-Versionen wurde jeder Benutzer beim Anmelden nur der Gruppe zugeordnet, deren GID in seinem `/etc/passwd`-Eintrag angegeben war. Um die Gruppe zu wechseln, mußte der Benutzer das Kommando **newgrp** aufrufen. War der Gruppenwechsel erfolgreich, so war der Benutzer ab nun Mitglied der neuen (und nicht mehr der alten) Gruppe. Um zu seiner alten Gruppe zurück zu wechseln, mußte er lediglich **newgrp** ohne Argumente aufrufen.

Im Gegensatz dazu gibt es in SVR4 sogenannte Zusatz-GIDs (*supplementary group IDs*). Ein Benutzer kann somit zu einem Zeitpunkt nicht nur zu der in der Paßwortdatei angegebenen Gruppe (GID) gehören, sondern kann gleichzeitig auch Mitglied von weiteren Gruppen sein. Bei Dateizugriffen wird nicht nur die effektive GID mit der GID der Datei verglichen, sondern es werden zusätzlich alle Zusatz-GIDs des entsprechenden Benutzers mit der Datei-GID verglichen. Der Vorteil dieser Zusatz-GID ist, daß man nicht mehr mit **newgrp** seine Gruppenzugehörigkeit wechseln muß, wenn man auf Dateien einer anderen Gruppe zugreifen möchte, in der man ebenfalls Mitglied ist. Um Zusatz-GIDs zu erfragen oder weitere einzutragen, stehen die Funktionen **getgroups**, **setgroups** und **initgroups** zur Verfügung.

```
#include <sys/types.h>
#include <grp.h>

int getgroups(int anzahl, gid_t gruppenliste[]);
                                     gibt zurück: Anzahl von Zusatz-GIDs (bei Erfolg); -1 bei Fehler

int setgroups(int gruppzahl, const gid_t gruppenliste[]);
int initgroups(const char *loginname, gid_t passwdgid);
                                     beide geben zurück: 0 (bei Erfolg); -1 bei Fehler
```

getgroups

Diese Funktion schreibt in das Array *gruppenliste* bis zu *anzahl* Zusatz-GIDs und liefert als Rückgabewert die Anzahl der wirklich in diesem Array hinterlegten Zusatz-GIDs. Wie viele Zusatz-GIDs maximal an einem System erlaubt sind, enthält die in `<limits.h>` definierte Konstante `NGROUPS_MAX`. Ein üblicher Wert für `NGROUPS_MAX` ist 16. Falls das entsprechende System keine Zusatz-GIDs kennt, so hat diese Konstante den Wert 0. In diesem Fall liefert **getgroups** als Rückgabewert 0 und nicht -1 für Fehler.

Falls für *anzahl* der Wert 0 angegeben wird, so liefert **getgroups** nur die Anzahl der Zusatz-GIDs ohne den Inhalt von *gruppenliste* zu modifizieren. So kann man immer im voraus die benötigte Größe des Arrays *gruppenliste* ermitteln.

setgroups

Diese Funktion kann vom Superuser aufgerufen werden, um die Zusatz-GIDs für den aufrufenden Prozeß zu setzen. `gruppenliste` enthält dabei die Zusatz-GIDs und `gruppzahl` die Anzahl der im Array `gruppenliste` enthaltenen Zusatz-GIDs. Die einzige Verwendung für **setgroups** ist, daß diese Funktion von **initgroups** aufgerufen wird.

initgroups

Diese Funktion liest mittels der zuvor beschriebenen Funktion **getgrent**, **setgrent** und **endgrent** die ganze Gruppendatei und ermittelt so alle Gruppenmitgliedschaften des Benutzers `loginname`. Danach ruft sie **setgroups** auf, um die Zusatz-GIDs für den Benutzer `loginname` einzurichten. Das Argument `passwdgid` legt dabei die GID fest, die in `/etc/passwd` für den Benutzer `loginname` einzutragen ist. Diese GID wird auch als Zusatz-GID eingetragen. Da **initgroups** die Routine **setgroups** aufruft, kann nur der Superuser **initgroups** aufrufen.

initgroups wird nur von wenigen Programmen, wie z.B. dem Kommando **login** aufgerufen, wenn sich ein Benutzer anmeldet.

Hinweis

Von diesen drei Funktionen ist nur **getgroups** von POSIX.1 vorgeschrieben. SVR4 stellt jedoch alle drei Funktionen zur Verfügung.



Die Konstante `NGROUPS_MAX` ist unter Linux in der Headerdatei `<linux/limits.h>` definiert. Unter Linux 2.0 ist `NGROUPS_MAX` z.B. auf 32 gesetzt.



6.3 Informationen aus Netzwerkdateien

Neben der Paßwort- und Gruppendatei gibt es weitere Informationsdateien in Unix, wie z.B. Dateien der BSD-Netzwerk-Software

/etc/services	Dienste, die von den verschiedenen Netzwerk-Servern angeboten werden
/etc/networks	Informationen über die Netzwerke
/etc/protocols	Netzwerkprotokolle
/etc/hosts	Benutzer, die über Netz Zugriff auf den lokalen Rechner haben

Um Informationen aus diesen Netzwerkdateien zu erfragen, wird die gleiche Art von Routinen angeboten, wie wir sie bei der Paßwort- und Gruppendatei in den beiden vor-

herigen Kapiteln kennengelernt haben. Grundsätzlich werden dabei für jede Netzwerkdatei mindestens drei Funktionen angeboten:

- 1. Eine Funktion mit dem Präfix **get**, die immer den nächsten Eintrag aus der betreffenden Datei liefert und – falls erforderlich – zuvor diese Datei öffnet. Dieser Typ von Funktion liefert immer einen Zeiger auf eine **static**-Struktur, wobei ein gelieferter NULL-Zeiger anzeigt, daß das Dateiende erreicht wurde.
- 2. Eine Funktion mit dem Präfix **set**, die die entsprechende Datei öffnet, wenn sie noch nicht offen ist, und den Lesezeiger in jedem Fall auf den Dateianfang setzt.
- 3. Eine Funktion mit dem Präfix **end**, die die entsprechende Datei schließt.

Zusätzlich werden für diese Dateien noch Funktionen angeboten, die ein gezieltes Erfragen eines bestimmten Eintrags ermöglichen, wie dies auch schon bei der zuvor beschriebenen Paßwortdatei (**getpwuid**, **getpwnam**) oder Gruppendatei (**getgrgid**, **getgrnam**) der Fall war. Tabelle 6.3 faßt die Funktionen dieser Art für die betreffenden Dateien zusammen.

	Headerdatei	Struktur	Funktionen zum gezielten Erfragen eines Eintrags
/etc/services	<netdb.h>	servent	getservbyname , getservbyport
/etc/networks	<netdb.h>	netent	getnetbyname , getnetbyaddr
/etc/protocols	<netdb.h>	protoent	getprotobyname , getprotobynumber
/etc/hosts	<netdb.h>	hostent	gethostbyname , gethostbyaddr

Tabelle 6.3: Funktionen zum gezielten Erfragen von Einträgen in Netzwerkdateien

Kapitel 19.7, das die Netzwerkprogrammierung mit TCP/IP behandelt, stellt diese Funktionen detaillierter vor.

Hinweis

Unter SVR4 sind diese vier Dateien /etc/services, /etc/networks, /etc/protocols und /etc/hosts symbolische Links zu gleichnamigen Dateien im Directory /etc/inet oder eventuell auch anderen Directories, wie z.B. /usr/etc oder /conf/etc.

Es gibt in SVR4 weitere ähnliche Funktionen, die für die Systemadministration benötigt werden und von der jeweiligen Implementierung abhängig sind.

6.4 Informationen zum lokalen System

6.4.1 uname – Erfragen von Informationen zum lokalen System

Um Informationen zum lokalen System zu erfragen, steht die von POSIX.1 definierte Funktion **uname** zur Verfügung.

```
#include <sys/utsname.h>
```

```
int uname(struct utsname *name);
```

gibt zurück: nicht negativen Wert (bei Erfolg); -1 bei Fehler

`name` ist die Adresse einer Struktur (`struct utsname`), die von der Funktion **`uname`** gefüllt wird. Die Komponenten von `struct utsname` entsprechen der Ausgabe des Kommandos **`uname`**:

```
struct utsname {  
    char sysname[9]; /* Betriebssystemname */  
    char nodename[9]; /* Knotenname */  
    char release[9]; /* Release-Name */  
    char version[9]; /* Versionsname dieses Releases */  
    char machine[9]; /* Zugrundeliegende Hardware */  
};
```

POSIX.1 schreibt diese Komponenten als Minimalausstattung von `struct utsname` vor. So wird z.B. unter SVR4 oft noch eine weitere Komponente `domainname` angeboten.

POSIX.1 schreibt die Arraygröße von 9 nicht vor. In SVR4 sind oft 255 relevante Zeichen (und abschließendes `\0`) für die einzelnen Komponenten vorgesehen.

6.4.2 **gethostname** – Erfragen des Hostnamens in einem TCP/IP-Netzwerk

Um den Hostnamen des lokalen Systems in einem TCP/IP-Netzwerk zu erfragen, steht die Funktion **`gethostname`** zur Verfügung.

```
#include <unistd.h>
```

```
int gethostname(char *name, int namlaenge);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Funktion **`gethostname`** schreibt den Hostnamen des lokalen Systems an die Adresse `name`, wobei sie diesen String mit `\0` abschließt. Wie viele Zeichen diese Funktion maximal an die Adresse schreiben soll, wird ihr über das Argument `namlaenge` mitgeteilt.

Die maximal mögliche Länge des Hostnamens wird über die in `<sys/params.h>` definierte Konstante `MAXHOSTNAMELEN` (in SVR4 256) festgelegt.

Hinweis

Ist das lokale System in einem TCP/IP-Netzwerk eingebettet, so ist der Hostname der vollständige Domainname.

Mit dem Kommando **hostname** kann man entweder den momentanen Hostnamen erfragen oder einen neuen Hostnamen an das lokale System vergeben. Das letztere, wofür die Funktion **sethostname** benötigt wird, ist jedoch nur dem Superuser erlaubt.

gethostname und **sethostname** waren ebenso wie das Kommando **hostname** ursprünglich nur auf BSD-Unix verfügbar. In SVR4 werden sie aber mit dem *BSD Compatibility Package* angeboten.

6.5 Informationen zu Systemanmeldungen

Die meisten Unix-Systeme enthalten zwei Dateien, in denen sie alle Benutzermeldungen mitprotokollieren.

- Datei **utmp**
enthält Informationen zu allen momentan angemeldeten Benutzern. Das Kommando **who** liest diese Datei und gibt ihren Inhalt in einer lesbaren Form aus.
- Datei **wtmp**
enthält Informationen zu allen stattgefundenen An- und Abmeldungen am System. Das Kommando **last** durchsucht den Inhalt dieser Datei nach bestimmten Einträgen und gibt die gefundenen Informationen in einer lesbaren Form aus.

Beide Dateien enthalten je Eintrag die in der Struktur **utmp** festgelegten Komponenten. Diese Struktur ist in `<utmp.h>` definiert und enthält eine Vielzahl von Informationen, wie z.B.:

```
struct utmp {
    short   ut_type;           /* Typ des Logins          */
    pid_t   ut_pid;           /* PID des Login-Prozesses */
    char    ut_line[12];      /* Gerätename von tty - "/dev/" */
    char    ut_id[2];         /* abgek. ttyname, wie 01, sl etc. */
    time_t  ut_time;         /* Login-Zeit              */
    char    ut_user[UT_NAMESIZE]; /* Benutzername, ohne \0    */
    char    ut_host[16];      /* Hostname für entfernte Logins */
    long    ut_addr;         /* IP-Adresse von entfernten Host */
};
```

Beim Anmelden mit **login** wird diese Struktur gefüllt und in die Datei **utmp** geschrieben. Beim Abmelden wird dieser Eintrag durch den **init**-Prozeß aus der Datei **utmp** gelöscht und in die Datei **wtmp** eingetragen.

Auch ein **reboot** oder das Ändern der Systemzeit wird über spezielle Einträge in der Datei **wtmp** festgehalten.

Hinweis

Neben dieser Struktur existieren noch eine ganze Reihe von Funktionen, mit denen man Informationen aus den beiden Dateien **utmp** und **wtmp** erfragen bzw. in diese eintragen kann. Diese Funktionen sind in SVR4 in den Manpages **getut(3)** bzw. **getutx(3)** und unter BSD-Unix in der der Manpage **utmp(5)** beschrieben.

Unter SVR4 befinden sich die beiden Dateien `utmp` und `wtmp` im Directory `/var/adm` und in BSD-Unix im Directory `/var/log`.

6.6 Übung

6.6.1 Ausgeben von allen Loginnamen und Paßwörtern

Erstellen Sie ein Programm `pwoert.c`, das alle Loginnamen mit zugehörigen Paßwörtern ausgibt. Dieses Programm kann natürlich nur vom Superuser erfolgreich aufgerufen werden.

6.6.2 Ausgeben von Informationen zum lokalen System

Erstellen Sie ein Programm `lokalsys.c`, das Informationen zum lokalen System in folgender Form ausgibt.

```
Betriebssystem-Name: SunOS
Knoten-Name: server001
Release-Name: 5.1
Versions-Name: Generic
Hardware: i86pc
```

6.6.3 Ausgeben von Netzwerkinformationen

Erstellen Sie ein Programm `netinfo.c`, das alle Informationen aus den in Kapitel 6.3 vorgestellten Netzwerkdateien liest und ausgibt.

6.6.4 Ausgeben aller momentan angemeldeten Benutzer

Erstellen Sie ein Programm `wer.c`, das ähnlich zum Kommando **who** alle momentan angemeldeten Benutzer in folgender Form ausgibt.

```
root          console
emil          tty03
anja          tty06
fritz         tty11
.....
```

6.6.5 Ausgeben von Informationen zu bestimmten Benutzern

Erstellen Sie ein Programm `pwinfo.c`, das alle in `/etc/passwd` verfügbaren Informationen zu den Benutzern ausgibt, deren Loginname oder User-ID auf der Kommandozeile angegeben ist, wie z.B.:

```
$ pwinfo hh 7 11
----- 1. Argument: hh -----
Name: hh
Home directory: /home/hh, Login Shell: /bin/tcsh
```

```

    UID: 2021, GID: 1
    Passwort: Igk5vho4xpCXg, Kommentar: Helmut Herold
----- 2. Argument: 7 -----
    Name: halt
    Home directory: /sbin, Login Shell: /sbin/halt
    UID: 7, GID: 0
    Passwort: *, Kommentar: halt
----- 3. Argument: 11 -----
    Name: operator
    Home directory: /root, Login Shell: /bin/bash
    UID: 11, GID: 0
    Passwort: *, Kommentar: operator
$

```

6.6.6 Ausgeben von Informationen zu bestimmten Gruppen

Erstellen Sie ein Programm `grinfo.c`, das die zu bestimmten Gruppen verfügbare Information ausgibt. Die Gruppen sind dabei entweder über Loginname oder über Group-ID auf der Kommandozeile zu spezifizieren, wie z.B.:

```

$ grinfo bin adm 12 grafik
----- 1. Argument: bin -----
    Gruppenname: bin
    GID: 1
    Mitglieder:
        root
        bin
        daemon
----- 2. Argument: adm -----
    Gruppenname: adm
    GID: 4
    Mitglieder:
        root
        adm
        daemon
----- 3. Argument: 12 -----
    Gruppenname: mail
    GID: 12
    Mitglieder:
        mail
----- 4. Argument: grafik -----
    Gruppenname: grafik
    GID: 100
    Mitglieder:
        hans
        sven
        martin
        franky
        rh
        ug
        maik
        petra
        chris
$

```

6.6.7 Implementierung des Kommandos **id**

Erstellen Sie ein Programm `id.c`, das das Linux/Unix-Kommando **id** nachbildet. Ruft man **id** ohne Argumente auf, so gibt es Informationen zum Aufrufer (IDs, Loginname, Gruppennamen) aus.

```
$ id
uid=500(hh) gid=100(users) groups=100(users)
$ id xxx
id: xxx: No such user
$
```

Wird **id** mit einem Loginnamen aufgerufen, so gibt es die entsprechenden Informationen zu diesem Benutzer aus.

```
$ id root
uid=0(root) gid=0(root) groups=0(root),1(bin),65534(nogroup)
$
```


7 Datums- und Zeitfunktionen

Die Zeit weilt, eilt, teilt und heilt.

Spruchwort

Die Headerdatei `<time.h>` enthält von ANSI C vorgeschriebene Konstanten, Datentypen und Funktionen, die sich für das Setzen und Erfragen von Datums- und Zeitwerten eignen.

7.1 Datentypen und Konstanten

ANSI C schreibt vor, daß die folgenden Datentypen und Konstanten in `<time.h>` definiert sein müssen.

7.1.1 Datentypen

size_t

Bei `size_t` handelt es sich um einen (unter anderem auch in `<stdio.h>` und `<stddef.h>` definierten) vorzeichenlosen Ganzzahl-Datentyp, der für das Ergebnis des **sizeof**-Operators eingeführt wurde. Dieser Typ `size_t` wird meist als Typ für Funktionsargumente verwendet, welche Größenangaben repräsentieren, wie z.B.:

```
void *malloc(size_t groesse);
```

clock_t

ist ein arithmetischer Datentyp, der für CPU-Zeiten verwendet wird.

time_t

ist ein arithmetischer Datentyp, der für Datums- und Zeitangaben verwendet wird.

struct tm

Diese Struktur enthält alle zu einer Kalenderzeit (Datum und Zeit im Gregorianischen Kalender) relevanten Komponenten. In dieser Struktur sollten laut ANSI C zumindest die folgenden Komponenten enthalten sein (Reihenfolge ist dabei nicht festgelegt):

```
int tm_sec;    /* Sekunden nach der Minute:  [0,61]1 */
int tm_min;    /* Minuten nach der Stunde:   [0,59] */
int tm_hour;   /* Stunden seit Mitternacht: [0,23] */
int tm_mday;   /* Monatstag:                    [1,31] */
```

1. Erlaubt ein Uhrticken im Zweisekunden-Rhythmus (1, 3, 5, ..., 59, 61).

```

int tm_mon;    /* Monat seit Januar:      [0,11] */
int tm_year;   /* Jahr seit 1900                        */
int tm_wday;   /* Tag seit Sonntag:                    [0,6] */
int tm_yday;   /* Tag seit 1. Januar:                  [0,365] */
int tm_isdst;  /* (is daylight saving time) zeigt an, ob es sich um Sommer-
                        zeit handelt (positiv) oder nicht (0).
                        Negativer Wert bedeutet: Diese Information ist nicht
                        verfügbar */

```

Bei einigen Systemen, wie z.B. auch bei Linux, enthält die Struktur `struct tm` zusätzlich noch die beiden folgenden nicht standardisierten Komponenten:

```

long int tm_gmtoff;
    gibt die Sekunden östlich von UTC bzw. negative Sekunden westlich von UTC für
    Zeitzonen an, die östlich der Datumslinie liegen. Manchmal ist der Name dieser Kom-
    ponente auch __tm_gmtoff.

const char *tm_zone;
    enthält den Namen der aktuellen Zeitzone, wobei zu beachten ist, daß manche Zeitzo-
    nen auch mehrere Namen haben können. Manchmal ist der Name dieser Komponente
    auch __tm_zone.

```

7.1.2 Konstanten

CLOCKS_PER_SEC

Diese Konstante enthält Anzahl von `clock_t`-Einheiten pro Sekunde

NULL

Nullzeiger, der auch in anderen Headerdateien (wie z.B. `<stdio.h>`) definiert ist.

7.2 Datums- und Zeitfunktionen

Die Zeit, mit der der Unix-Kern arbeitet, sind die seit 00:00:00 Uhr des 1. Januars 1970 (UTC)² verstrichenen Sekunden. Diese Zeit (Kalenderzeit) wird immer im Datentyp `time_t` dargestellt und enthält sowohl das Datum als auch die Zeit.

Unix unterscheidet sich bei der Handhabung der Kalenderzeit in einigen Punkten von anderen Systemen:

- ▶ Es verwendet intern die UTC-Zeit anstelle der lokalen Zeit.
- ▶ Es stellt automatisch von Sommer- auf Winterzeit und umgekehrt um.
- ▶ Intern hält Unix die Zeit und das Datum getrennt.

2. Abkürzung für Universal Time Coordinated, die der GMT (Greenwich Mean Time) entspricht.

7.2.1 **time** und **gettimeofday** – Erfragen der momentanen Kalenderzeit

Um die momentane Kalenderzeit zu erfragen, steht die Funktion **time** zur Verfügung.

```
#include <time.h>

time_t time(time_t *time_tzgr);
```

gibt zurück: momentane Kalenderzeit (bei Erfolg); -1 bei Fehler

Wird für `time_tzgr` kein Nullzeiger angegeben, dann wird der entsprechende Rückgabewert (Kalenderzeit) auch noch im Speicherplatz hinterlegt, auf den `time_tzgr` zeigt.

Hinweis

Um die Kernzeit zu setzen, steht die Funktion **stime** zur Verfügung.

Um z.B. den Zufallszahlengenerator auf einen nicht vorhersagbaren Startwert zu setzen, wird meist folgende Vorgehensweise gewählt:

```
#include <stdlib.h>
#include <time.h>
....
    srand(time(NULL)); /*Noch besser unter Linux/Unix: srand (time(NULL) + getpid ()); */
....
/* Jeder Aufruf von rand() liefert dann einen zufälligen
   nicht vorhersagbaren Wert zwischen 0 und RAND_MAX
   (RAND_MAX ist definiert in <stdlib.h>) */
```

Nachdem man mit der Funktion **time** die seit Beginn des Jahres 1970 verstrichenen Sekunden ermittelt hat, kann man unter Verwendung einer der Funktionen aus Abbildung 7.1 diese »Sekunden-Zeit« in ein verständliches Datums- und Zeitformat konvertieren.

Die in Abbildung 7.1 mit schwächeren Linien gezeichneten Funktionen **localtime**, **mktime**, **ctime** und **strftime** werden alle durch die Environment-Variable TZ, die später in diesem Kapitel beschrieben wird, beeinflusst.

Das Messen der Kalenderzeit in Sekunden reicht für manche Anwendungen nicht aus, weshalb viele Systeme – wie z.B. BSD, SVR4 und Linux – eine zusätzliche Funktion **gettimeofday** anbieten, die zusätzlich zu den Sekunden noch die abgelaufenen Mikrosekunden und Informationen zur Zeitzone und Sommerzeit liefert.

```
#include <sys/time.h>
#include <unistd.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

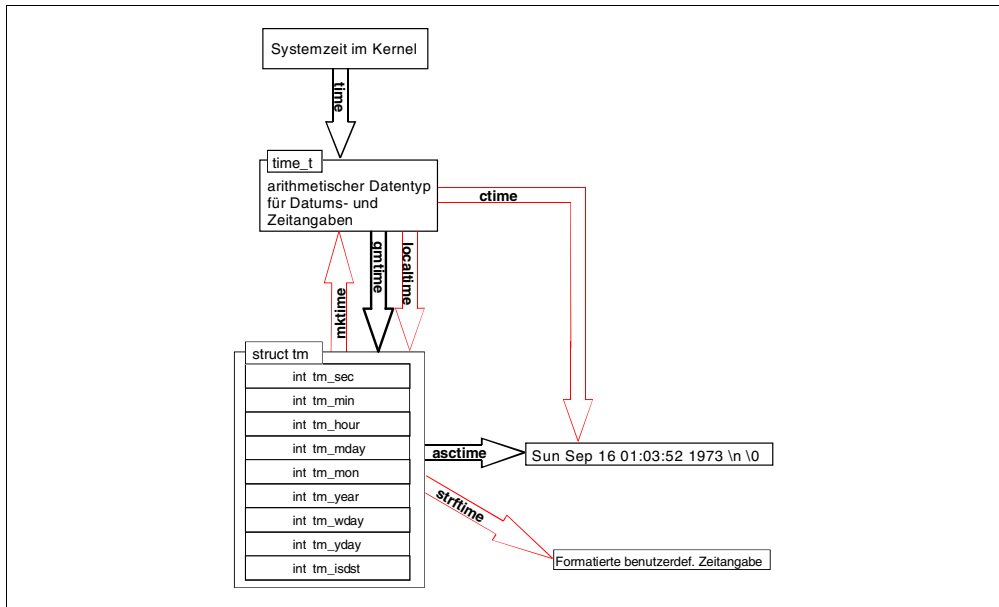


Abbildung 7.1: Zusammenfassung der wichtigsten Zeitformatumwandlungen

Die beiden Strukturen `struct timeval` und `struct timezone` sind in `<sys/time.h>` bzw. `<linux/time.h>` wie folgt definiert:

```
struct timeval {
    long tv_sec;    /* Sekunden */
    long tv_usec;   /* Mikrosekunden */
};

struct timezone {
    int tz_minuteswest; /* Minuten westlich von Greenwich */
    int tz_dsttime;     /* Art der Sommerzeitregelung */
};
```

Zusätzlich bietet `<sys/time.h>` drei Makros zum Arbeiten mit der `timeval`-Struktur an.

```
#define timerclear(tvp) ((tvp)->tv_sec = (tvp)->tv_usec = 0)
```

setzt beide Komponenten der `timeval`-Struktur auf 0.

```
#define timerisset(tvp) ((tvp)->tv_sec || (tvp)->tv_usec)
```

überprüft, ob eine der beiden Komponenten der `timeval`-Struktur ungleich 0 ist.

```
#define timercmp(tvp, uvp, cmp) \
    (((tvp)->tv_sec == (uvp)->tv_sec && \
      (tvp)->tv_usec cmp (uvp)->tv_usec) \
     || (tvp)->tv_sec cmp (uvp)->tv_sec)
```


vergleicht die beiden `timeval`-Strukturen, auf die die Parameter `tv` und `uv` zeigen mittels des Vergleichsoperators `cmp`, so daß dies dem Ausdruck `tv cmp uv` entspricht. Hierbei ist lediglich zu beachten, daß dieses Makro nur für Vergleichsoperatoren funktioniert, die aus einem Zeichen bestehen, also nicht für die Operatoren `<=` und `>=`. Um diese beiden Operatoren mit diesem Makro nachzubilden, müßte man `!timercmp(tv, uv, >)` bzw. `!timercmp(tv, uv, <)` angeben.

7.2.2 `gmtime` und `localtime` – Umwandeln von `time_t`-Zeit in `struct tm`-Zeit

Um die im Datentyp `time_t` (in Sekunden) gespeicherte Kalenderzeit in die Struktur `struct tm` umzuwandeln, stehen die beiden Funktionen **`gmtime`** und **`localtime`** zur Verfügung.

```
#include <time.h>

struct tm *gmtime(const time_t *time_tzgr);
struct tm *localtime(const time_t *time_tzgr);
```

beide geben zurück: Zeiger auf `struct tm`

Der Unterschied zwischen **`localtime`** und **`gmtime`** ist, daß **`localtime`** die Kalenderzeit, auf die `time_tzgr` zeigt, in die lokale Ortszeit (unter Berücksichtigung der lokalen Zeitzone und Sommer- bzw. Winterzeit) umwandelt, während **`gmtime`** die Kalenderzeit in die UTC-Zeit umwandelt.

7.2.3 `mktime` – Umwandeln von `struct tm`-Zeit in `time_t`-Zeit

Um die im Datentyp `struct tm` gespeicherte Zeit in eine `time_t`-Zeit umzuwandeln, steht die Funktion **`mktime`** zur Verfügung.

```
#include <time.h>

time_t mktime(const struct tm *tmzgr);
```

gibt zurück: Kalenderzeit im Datentyp `time_t` (bei Erfolg); -1 bei Fehler

Hinweis

Die originalen Werte der Komponenten `tm_wday` und `tm_yday` in `*tmzgr` werden ignoriert, und die Originalwerte der anderen Komponenten sind nicht auf die angegebenen Bereiche begrenzt. Bei erfolgreicher Ausführung dieser Funktion werden die Werte von `tm_wday` und `tm_yday` geeignet gesetzt, und die anderen Komponenten aus `*tmzgr` werden entsprechend angepaßt, um die angegebene Kalenderzeit darzustellen, aber diesesmal liegen die Werte in den angegebenen Bereichen. Der endgültige Wert von `tm_mday` wird

nicht gesetzt, bis `tm_mon` und `tm_year` festgelegt sind. So könnte z.B. `tm_mday` mit Wert 35 besetzt sein. **mktime** ist dann verpflichtet, die Komponenten wieder richtig zu setzen, d.h., in ihre definierten Bereiche (siehe `struct tm` in Kapitel 7.1) zu transformieren.

Beispiel

Wochentag zu einem Datum bestimmen

Das nachfolgende C-Programm 7.1 (`welchtag.c`) liest ein Datum ein und gibt dann aus, um welchen Wochentag es sich dabei handelt.

```
#include <time.h>
#include "eighdr.h"

static const char *const wochentag[] = {
    "Sonntag",    "Montag",  "Dienstag",  "Mittwoch",
    "Donnerstag", "Freitag",  "Samstag",   "unbekannt"
};

int
main(void)
{
    struct tm    tmzeit;
    long int     tag, monat, jahr;

    printf("Datum (tt.mm.jjjj) ? (jjjj muss >= 1900 sein) ");
    scanf("%d.%d.%d", &tag, &monat, &jahr);
    while (jahr < 1900) {
        printf("Das Jahr muss >= 1900 sein !\a\n\n");
        printf("Datum (tt.mm.jjjj) ? (jjjj muss >= 1900 sein) ");
        scanf("%d.%d.%d", &tag, &monat, &jahr);
    }
    tmzeit.tm_year = jahr-1900;
    tmzeit.tm_mon  = monat-1;
    tmzeit.tm_mday = tag;
    tmzeit.tm_hour = tmzeit.tm_min = 0;
    tmzeit.tm_sec  = 1;
    tmzeit.tm_isdst = -1;

    if (mktime(&tmzeit) == -1)
        fehler_meld(FATAL, "Fehler bei mktime");
    else
        printf("Dieses Datum war/ist ein %s\n", wochentag[tmzeit.tm_wday]);

    exit(0);
}
```

Programm 7.1 (`welchtag.c`): Wochentag zu einem Datum bestimmen

Nachdem man dieses Programm 7.1 (`welchtag.c`) kompiliert und gelinkt hat

```
cc -o welchtag welchtag.c fehler.c
```

ergeben sich z.B. beim Start folgende Abläufe:

```
$ welchtag
Datum (tt.mm.jjjj) ? (jjjj muss >= 1900 sein) 12.4.2015
Dieses Datum war/ist ein Sonntag
$ welchtag
Datum (tt.mm.jjjj) ? (jjjj muss >= 1900 sein) 24.12.1980
Dieses Datum war/ist ein Mittwoch
$
```

Es ist darauf hinzuweisen, daß auf den meisten Unix-Systemen **mktime** nur für einen begrenzten Zeitraum ausgelegt ist (siehe auch Übungen in Kapitel 7.3).

7.2.4 asctime und ctime – Umwandeln von struct tm- und time_t-Zeit in date-String

Um die im Datentyp `struct tm` bzw. die im Datentyp `time_t` gespeicherte Zeit in einen String umzuwandeln, der der Ausgabe des Kommandos **date** entspricht, stehen die beiden Funktionen **asctime** und **ctime** zur Verfügung.

```
#include <time.h>

char *asctime(const struct tm *tmzgr);
char *ctime(const time_t *time_tzgr);
```

beide geben zurück: Zeiger auf String, der date-Ausgabe entspricht

Beide Funktionen liefern einen Zeiger auf einen String, der die entsprechende Zeit in Form der **date**-Ausgabe enthält:

```
Sun Sep 16 01:03:52 1973\n\0
```

Während bei **asctime** ein `struct tm`-Zeiger als Argument anzugeben ist, muß bei **ctime** als Argument ein `time_t`-Zeiger angegeben werden.

Während **ctime** die lokale Zeit liefert, benutzt **asctime** die Zeitzone, die in `struct tm` angegeben ist, also UTC, wenn diese mit **gmtime** ermittelt wurde, und die lokale Zeit, wenn diese mit **localtime** ermittelt wurde.

Beispiel

Datum vor bzw. in x Tagen bestimmen

Das nachfolgende C-Programm 7.2 (`welchdat.c`) beantwortet die Frage: *Welches Datum ist/war heute in/vor x Tagen ?*

```
#include <time.h>
#include "eighdr.h"

int
main(void)
{
    struct tm  zeit_string;
    time_t     heute, neu_datum;
    long int    tage;

    printf("Wieviele Tage von heute ab ? ");
    scanf("%ld", &tage);

    time(&heute);
    printf("\nHeute ist %s", ctime(&heute));

    zeit_string = *localtime(&heute);
    zeit_string.tm_mday += tage;

    if ( (neu_datum=mktime(&zeit_string)) == -1 )
        fehler_meld(FATAL, "Fehler bei mktime");
    else
        printf("Datum/Zeit %s %d Tage %s %s\n",
               tage>0?"in":"vor", abs(tage), tage>0?"ist":"war",
               ctime(&neu_datum));
    exit(0);
}
```

Programm 7.2 (welchdat.c): Datum vor bzw. in *x* Tagen bestimmen

Nachdem man dieses Programm 7.2 (welchdat.c) kompiliert und gelinkt hat

```
cc -o welchdat welchdat.c fehler.c
```

ergeben sich z.B. beim Start folgende Abläufe:

```
$ welchdat
Wieviele Tage von heute ab ? 150

Heute ist Tue Sep 22 16:18:06 1992
Datum/Zeit in 150 Tage ist Fri Feb 19 16:18:06 1993
$ welchdat
Wieviele Tage von heute ab ? -5000

Heute ist Tue Sep 22 16:19:21 1992
Datum/Zeit vor 5000 Tage war Sun Jan 14 15:19:21 1979
$
```

7.2.5 strftime – Umwandeln einer struct tm-Zeit in formatierten benutzerdefinierten String

Um die im Datentyp `struct tm` gespeicherte Zeit in einen formatierten benutzerdefinierten String umzuwandeln, steht die Funktion **strftime** zur Verfügung.

```
#include <time.h>

size_t strftime(char *puffer, size_t max, const char *format,
                const struct tm *tmzgr);
```

gibt zurück: Anzahl der nach *puffer* geschriebenen Zeichen;
0, wenn mehr als *max* Zeichen nach *puffer* zu schreiben sind

Die Funktion **strftime** ist in etwa ein **sprintf** für Zeit- und Datumswerte.

Sie schreibt die Kalenderzeit aus der Struktur **tmzgr* entsprechend der *format*-Angabe an die Adresse *puffer*.

In der *format*-Zeichenkette können entweder einfache Zeichen (nicht %) oder Umwandlungsvorgaben angegeben werden. Die einfachen Zeichen werden unverändert nach *puffer* geschrieben. Eine Umwandlungsvorgabe ist ein %, gefolgt von einem Zeichen, das die »Ersetzung« festlegt. Die möglichen Umwandlungszeichen sind in der Tabelle 7.1 zusammengefaßt.

Angabe	wird ersetzt durch	Beispiel
%a	abgekürzter Wochentagsname	Mon
%A	ausgeschriebener Wochentagsname	Monday
%b	abgekürzter Monatsname	Apr
%B	ausgeschriebener Monatsname	April
%c	entspr. Datums- und Zeitdarstellung	Mon Apr 25
	MET 1994	21:32:59
%d	Monatstag (01-31)	25
%H	Stunde (00-23)	21
%I	Stunde (01-12)	09
%j	Tag des Jahres (001-365)	114
%m	Monat (01-12)	04
%M	Minute (00-59)	32
%p	AM oder PM (für amerik. AM/PM-Schreibweise	PM
%S	Sekunden (00-61)	59

Tabelle 7.1: Umwandlungszeichen für **strftime** mit Beispielen zu **Mon Apr 25 21:32:59 MET 1994**

Angabe	wird ersetzt durch	Beispiel
%U	Wochennummer (00-53; 1.Sonntag=1.Tag der 1.Woche)	17
%w	Wochentag (0-6; 0 = Sonntag)	1
%W	Wochennummer (00-53; 1.Montag=1.Tag der 1.Woche)	17
%x	geeignete Datum-Darstellung	04/25/94
%X	geeignete Zeit-Darstellung	21:32:59
%y	Jahreszahl (ohne Jahrhundertzahl: 00-99)	94
%Y	Jahreszahl (mit Jahrhundertzahl)	1994
%Z	Zeitzone	MET
%%	%	%

Tabelle 7.1: Umwandlungszeichen für **strftime** mit Beispielen zu **Mon Apr 25 21:32:59 MET 1994**

Die dritte Spalte in der Tabelle 7.1 ist eine Beispielausgabe unter SVR4 zu folgendem von **strftime** gelieferten Datums- und Zeitstring:

Mon Apr 25 21:32:59 MET 1994

Es werden niemals mehr als `max` Zeichen nach `puffer` geschrieben. Wenn die Gesamtzahl der nach `puffer` geschriebenen Zeichen nicht größer als `max` ist, dann liefert die Funktion **strftime** die Gesamtzahl der geschriebenen Zeichen, ansonsten gibt sie 0 zurück und der Inhalt von `puffer` ist unbestimmt.

Hinweis

SVR4 bietet neben der in Tabelle 7.1 aufgezählten Umwandlungszeichen weitere Umwandlungszeichen an, wie z.B. `%n` für `\n` oder `%T` für Zeit im Format `%H:%M:%S`. Um alle Umwandlungszeichen für SVR4 zu erfahren, sollte man folgendes aufrufen.

`man strftime`

Manche Systeme, wie z.B. Linux, bieten noch eine nicht standardisierte Funktion **strptime** an, die die Umkehrung zur Funktion **strftime** ist, also einen String in eine `struct tm`-Zeit umformt.

```
#include <time.h>

char *strptime(char *puffer, const char *format, const struct tm *tmzgr);
```

gibt zurück: Zeiger auf Zeichen in `puffer`, das hinter dem letzten konvertierten Zeichen steht

strptime liest – ähnlich zu **scanf** – den angegebenen `puffer` entsprechend den gegebenen `format`-Angaben und schreibt die dazugehörige `struct tm`-Information an die Adresse, auf die der `tmzgr` zeigt.

Die möglichen Umwandlungszeichen in `format` sind in Tabelle 7.2 zusammengefaßt.

Angabe	gelesen wird (in <i>puffer</i>)
%a	abgekürzter Wochentagsname (wie z.B. Mon)
%A	ausgeschriebener Wochentagsname (wie z.B. Monday)
%b	abgekürzter Monatsname (wie z.B. Apr)
%B	ausgeschriebener Monatsname (wie z.B. April)
%h	abgekürzter oder ausgeschriebener Monatsname (wie z.B. Apr oder April)
%c	Datum und Zeit entsprechend der Formatangabe »%x %X«
%C	Datum und Zeit entsprechend der länderspezifischen (locale) Darstellung (wie von strptime bei Formatangabe »%c«)
%d	Monatstag (01-31)
%e	Monatstag (01-31); wie %d
%D	Datum in der Form »%m/%d/%y«
%H	Stunde (00-24)
%k	Stunde (00-24); wie %H
%I	Stunde (00-12)
%l	Stunde (00-12); wie %I
%j	Tag des Jahres (001-366)
%m	Monatsnummer (01-12)
%M	Minute (00-59)
%p	AM oder PM (für amerikanische AM / PM-Schreibweise)
%r	Zeit in der Form »%I:%M:%S %p«
%R	Zeit in der Form »%H:%M«
%S	Sekunden (00-61)
%T	Zeit in der Form »%H:%M:%S«
%w	Wochentag (0-6; 0=Sonntag)
%x	entsprechende lokale Form der Datumsangabe
%X	entsprechende lokale Form der Zeitangabe
%y	Jahreszahl (ohne Jahrhundertzahl; 00-99)
%Y	Jahreszahl (mit Jahrhundertzahl); wenn möglich, sollte diese Form benutzt werden, um das Jahr-2000-Problem zu vermeiden.
%%	%-Zeichen

Tabelle 7.2: Umwandlungszeichen für **strptime**

Bei den Umwandlungszeichen in Tabelle 7.2, die sich auf Zahlen beziehen, müssen bei einstelligen Ziffern nicht unbedingt führende Nullen vorhanden sein, um diese auf die entsprechende Stellenzahl aufzufüllen.

7.2.6 TZ – Environment Variable für die Zeitzone

Wie zuvor erwähnt, werden die in Abbildung 7.1 mit schwächeren Linien gezeichneten Funktionen **localtime**, **mktime**, **ctime** und **strftime** durch die von POSIX.1 definierte Environment Variable TZ beeinflusst. Wenn diese Variable definiert ist, so wird deren Inhalt anstelle der voreingestellten Zeitzone von diesen vier Funktionen benutzt. Ist diese Environment-Variable leer (z.B. mit TZ=) oder nicht definiert, dann wird normalerweise die UTC-Zeit von diesen Funktionen benutzt. Nachfolgend wird der Einfluß von TZ auf das Kommando **date** gezeigt.

```
$ echo $TZ
MET
$ date
TUE Apr 26 12:26:54 MET DST 1994
$ TZ=
$ date
TUE Apr 26 10:27:24 GMT 1994
$ TZ=MET
$
```

Wenn dieses Beispiel auch einen typischen Inhalt von TZ zeigt, so erlaubt POSIX.1 jedoch noch detailliertere Angaben in TZ. Um mehr Information über den möglichen Inhalt der Environment-Variablen TZ zu erfahren, sollte man

```
man -a environ
```

aufrufen. Die entsprechende Beschreibung befindet sich in der Manualpage `environ(5)`.

Hinweis

Die ersten drei Zeichen von TZ definieren den Namen der Zeitzone. Die folgende Zahl gibt den Abstand zu UTC in Stunden an. Aus historischen Gründen gibt eine negative Zahl an, wie viele Stunden diese Zeit der UTC voraus ist. Die letzten drei Zeichen definieren den Namen der Zeitzone bei eingestellter Sommerzeit. So ist z.B. der Wert von TZ für unsere mitteleuropäische Zeitzone `MET-1MST` und der Wert für Colorado ist z.B. `MST7MDT`.

7.2.7 difftime – Ermitteln der Differenz zwischen zwei Uhrzeiten

Um die Differenz zwischen zwei Kalenderzeiten (vom Datentyp `time_t`) zu ermitteln, steht die Funktion **difftime** zur Verfügung.

```
#include <time.h>
```

```
double difftime(time_t zeit1, time_t zeit0);
```

gibt zurück: Differenz der beiden Zeiten *zeit1* und *zeit0* (in Sekunden)

Die Funktion **difftime** liefert die Differenz zwischen zwei Kalenderzeiten: `zeit1 - zeit0` als **double**-Wert (entspricht Sekunden) zurück.

Beispiel

Differenz zwischen zwei Daten (in Sekunden) ermitteln

```
#include <time.h>
#include "eighdr.h"
int
main(void)
{
    struct tm  zeit1={0}, zeit2={0};
    time_t     tzeit1, tzeit2;

    printf("Erstes Datum mit Zeit:\n");
    printf("    Datum (tt.mm.jjjj): ");
    scanf("%d.%d.%d", &zeit1.tm_mday, &zeit1.tm_mon, &zeit1.tm_year);
    printf("    Zeit (hh.mm.ss): ");
    scanf("%d.%d.%d", &zeit1.tm_hour, &zeit1.tm_min, &zeit1.tm_sec);
    zeit1.tm_year -= 1900;

    printf("Zweites Datum mit Zeit:\n");
    printf("    Datum (tt.mm.jjjj): ");
    scanf("%d.%d.%d", &zeit2.tm_mday, &zeit2.tm_mon, &zeit2.tm_year);
    printf("    Zeit (hh.mm.ss): ");
    scanf("%d.%d.%d", &zeit2.tm_hour, &zeit2.tm_min, &zeit2.tm_sec);
    zeit2.tm_year -= 1900;

    if ( (tzeit1=mktime(&zeit1)) == -1)
        fehler_meld(FATAL, "Fehler bei mktime (zeit1)");
    if ( (tzeit2=mktime(&zeit2)) == -1)
        fehler_meld(FATAL, "Fehler bei mktime (zeit2)");

    printf("\n ---> Differenz ist %.2lf Sekunden\n", difftime(tzeit2, tzeit1));

    exit(0);
}
```

Programm 7.3 (zeitdiff.c): Differenz zwischen zwei Kalenderzeiten (in Sekunden) ermitteln.

Nachdem man dieses Programm 7.3 (zeitdiff.c) kompiliert und gelinkt hat

```
cc -o zeitdiff zeitdiff.c fehler.c
```

ergibt sich z.B. beim Start folgender Ablauf:

```
$ zeitdiff
Erstes Datum mit Zeit:
    Datum (tt.mm.jjjj): 24.4.1980
```

```

Zeit (hh.mm.ss): 12.00.00
Zweites Datum mit Zeit:
Datum (tt.mm.jjjj): 1.5.1994
Zeit (hh.mm.ss): 17.15.23
---> Differenz ist 442473323.00 Sekunden
$

```

7.2.8 clock – Erfragen der seit Programmstart verbrauchten CPU-Zeit

Um die seit Programmstart vergangene CPU-Zeit zu ermitteln, steht die Funktion **clock** zur Verfügung.

```

#include <time.h>

clock_t clock(void);

gibt zurück: seit Programmstart vergangene CPU-Zeit (im Datentyp clock_t);
-1, wenn verbrauchte CPU-Zeit nicht verfügbar

```

Die Funktion **clock** liefert die von einem Programm seit seinem Start verbrauchte CPU-Zeit (in »Uhr-Ticks«) als `clock_t`-Wert. Falls die verbrauchte CPU-Zeit in Sekunden benötigt wird, dann muß der zurückgegebene Wert noch durch die Konstante `CLOCKS_PER_SEC` dividiert werden.

Beispiel

Zeitmessung in einem Programm

Das nachfolgende Programm 7.4 (`zeitmess.c`) demonstriert die Anwendung der Funktion **clock**, indem es alle Werte eines großen Arrays verachtachtet, wobei es zwei verschiedene Algorithmen anwendet.

```

#include <time.h>
#include "eighdr.h"

#define GROESSE 200000

int
main(void)
{
    long int    wert, array[GROESSE]={0}, *zgr, i;
    clock_t     start, mitte, ende;

    start = clock();

    for (i=0 ; i<GROESSE ; i++)
        array[i] = array[i]+array[i]+array[i]+array[i]+
            array[i]+array[i]+array[i]+array[i];
}

```

```

    mitte = clock();

    zgr = array;
    for (i=0 ; i<GROESSE ; i++)
        *zgr++ <= 3;

    ende = clock();

    printf("Durchlaufen mit Index          : %10.3f Sek\n",
           (float)(mitte-start)/CLOCKS_PER_SEC);
    printf("Durchlaufen mit Zeiger         : %10.3f Sek\n",
           (float)(ende-mitte)/CLOCKS_PER_SEC);
    printf("Gesamte vom Programm verbrauchte Zeit: %10.3f Sek\n",
           (float)(ende-start)/CLOCKS_PER_SEC);

    exit(0);
}

```

Programm 7.4 (zeitmess.c): Zeitmessung in einem Programm mit **clock**

Nachdem man dieses Programm 7.3 (zeitdiff.c) kompiliert und gelinkt hat

```
cc -o zeitmess zeitmess.c fehler.c
```

ergibt sich z.B. beim Start folgender Ablauf:

```

$ zeitmess
Durchlaufen mit Index          :      0.550 Sek
Durchlaufen mit Zeiger         :      0.150 Sek
Gesamte vom Programm verbrauchte Zeit:    0.700 Sek
$

```

Hieraus ist erkennbar, daß der zweite Algorithmus doch erheblich schneller ist.

7.2.9 Die Zeitgrenzen

Unter den meisten Unix-Systemen ist `time_t` eine vorzeichenbehaftete 32 Bit lange ganze Zahl, deren Nullwert für 00:00:00 Uhr des 1. Januars 1970 (UTC) steht. Mit diesen 32 Bit können alle Sekunden für die Zeitperiode vom 13. Dezember 1901 (größter negativer Wert) bis 19. Januar 2038 (größter positiver Wert) erfaßt werden.

Beispiel

Ausgeben von Zeitgrenzen und anderen Zeitinformatoren

```

#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

int
main(void)
{
    struct timeval  tv;

```

```

struct timezone  tz;
time_t          jetzt, anfang_zeit, ende_zeit, null_punkt = 0;
int             i, time_t_groesse = sizeof(time_t)*8;

anfang_zeit = 1L << (time_t_groesse-1);
ende_zeit   = ~anfang_zeit;

gettimeofday(&tv, &tz);
jetzt = tv.tv_sec;

printf("....time_t      : %d Bits\n", time_t_groesse);
printf("....Aktuelle Zeit: %ld Sek. (time)\n", time(NULL));
printf("....Aktuelle Zeit: %ld,%ld Sek. (gettimeofday)\n\n",
       tv.tv_sec, tv.tv_usec);
printf("....Jetzt ist es   : %s", ctime(&jetzt));
printf("....Anfang der Zeit: %s", ctime(&anfang_zeit));
printf("....Ende der Zeit  : %s", ctime(&ende_zeit));
printf("....Nullpunkt     : %s", ctime(&null_punkt));

exit(0);
}

```

Programm 7.5 (zeitgren.c): Ausgeben von Zeitgrenzen und anderen Zeitinformatoren

Nachdem man dieses Programm 7.5 (zeitgren.c) kompiliert und gelinkt hat

```
cc -o zeitgren zeitgren.c
```

kann man es starten und es liefert dann z.B. die folgende Ausgabe:

```

$ zeitgren
....time_t      : 32 Bits
....Aktuelle Zeit: 917188035 Sek. (time)
....Aktuelle Zeit: 917188035,477926 Sek. (gettimeofday)

....Jetzt ist es   : Sun Jan 24 15:27:15 1999
....Anfang der Zeit: Fri Dec 13 21:45:52 1901
....Ende der Zeit  : Tue Jan 19 04:14:07 2038
....Nullpunkt     : Thu Jan  1 01:00:00 1970

$

```

Auf 64-Bit-Systemen wird `time_t` als 64 Bit lange ganze Zahl dargestellt, wodurch eine astronomische Zeitperiode dargestellt werden kann, weshalb das obige Programm 7.5 (zeitgren.c) auf solchen 64-Bit-Systemen eventuell nicht zu einem Ende kommt. In diesem Fall muß es mit Strg-C beendet werden.

7.3 Übung

7.3.1 Letztes Jahr bei 32 Bit für time_t

In welchem Jahr wird der Datentyp `time_t` für die Kalenderzeit überlaufen, wenn für ihn 32-Bit-Integer mit Vorzeichen verwendet wird?

7.3.2 Maximale Prozeßlaufzeit bei 32 Bit für clock_t

Nach wieviel Tagen wird der Datentyp `clock_t` für die CPU-Zeit überlaufen, wenn für ihn 32-Bit-Integer mit Vorzeichen verwendet wird?

7.3.3 Simulieren einer digitalen Uhr

Erstellen Sie ein Programm `diguhr.c`, das eine digitale Uhr am Bildschirm simuliert. Die Uhr soll im Sekundentakt arbeiten.

7.3.4 Umsetzen des Kommandos cal

Erstellen Sie ein Programm `kal.c`, das ähnlich dem Unix-Kommando **cal** ist. Dieses Programm soll ein Jahr einlesen und dann den zugehörigen Kalender ausgeben.

Möglicher Ablauf des Programms `kal.c`:

```
$ kal
```

```
Kalender zu welchem Jahr ? 1996
```

```

                                1996

      Januar      Februar      März
So Mo Di Mi Do Fr Sa So Mo Di Mi Do Fr Sa So Mo Di Mi Do Fr Sa
  1  2  3  4  5  6      1  2  3      1  2
  7  8  9 10 11 12 13  4  5  6  7  8  9 10  3  4  5  6  7  8  9
14 15 16 17 18 19 20 11 12 13 14 15 16 17 10 11 12 13 14 15 16
21 22 23 24 25 26 27 18 19 20 21 22 23 24 17 18 19 20 21 22 23
28 29 30 31      25 26 27 28 29      24 25 26 27 28 29 30
                                   31

      April      Mai      Juni
So Mo Di Mi Do Fr Sa So Mo Di Mi Do Fr Sa So Mo Di Mi Do Fr Sa
  1  2  3  4  5  6      1  2  3  4      1
  7  8  9 10 11 12 13  5  6  7  8  9 10 11  2  3  4  5  6  7  8
14 15 16 17 18 19 20 12 13 14 15 16 17 18  9 10 11 12 13 14 15
21 22 23 24 25 26 27 19 20 21 22 23 24 25 16 17 18 19 20 21 22
28 29 30      26 27 28 29 30 31      23 24 25 26 27 28 29
                                   30

```

Juli							August							September						
So	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa
	1	2	3	4	5	6				1	2	3		1	2	3	4	5	6	7
7	8	9	10	11	12	13	4	5	6	7	8	9	10	8	9	10	11	12	13	14
14	15	16	17	18	19	20	11	12	13	14	15	16	17	15	16	17	18	19	20	21
21	22	23	24	25	26	27	18	19	20	21	22	23	24	22	23	24	25	26	27	28
28	29	30	31				25	26	27	28	29	30	31	29	30					
Oktober							November							Dezember						
So	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa	So	Mo	Di	Mi	Do	Fr	Sa
		1	2	3	4	5						1	2	1	2	3	4	5	6	7
6	7	8	9	10	11	12	3	4	5	6	7	8	9	8	9	10	11	12	13	14
13	14	15	16	17	18	19	10	11	12	13	14	15	16	15	16	17	18	19	20	21
20	21	22	23	24	25	26	17	18	19	20	21	22	23	22	23	24	25	26	27	28
27	28	29	30	31			24	25	26	27	28	29	30	29	30	31				

\$

7.3.5 Ausgabe der Zeit und des Datums in eigenem Format

Erstellen Sie ein Programm heute.c, das beim Aufruf die momentane Zeit und das heutige Datum in folgendem Format ausgibt.

10:58:59
03.May.1994 (Tue; 122.Tag des Jahres; 18.Kalenderwoche)

8

Nicht-lokale Sprünge

*Wenn auch die Welt im ganzen fortschreitet,
die Jugend muß doch immer wieder von vorne anfangen.*

Goethe

In C sind normalerweise nur lokale Sprünge (mit **goto**) möglich. Sprünge über Funktionsgrenzen hinweg sind nicht erlaubt. Mit ANSI C wurde eine eigene Headerdatei `<setjmp.h>` eingeführt, die zwei Funktionen anbietet, mit denen Sprünge über Funktionsgrenzen hinweg möglich sind.

8.1 Die Headerdatei `<setjmp.h>`

Normalerweise ist es nur möglich, von einer aufgerufenen Funktion in die aufrufende Funktion zurückzukehren. Abbildung 8.1 verdeutlicht dies am Stack-Layout, indem sie zeigt, daß aufgerufene Funktionen immer nur zum direkten Aufrufer, aber niemals zu einem indirekten Aufrufer in der Aufrufhierarchie zurückkehren können.

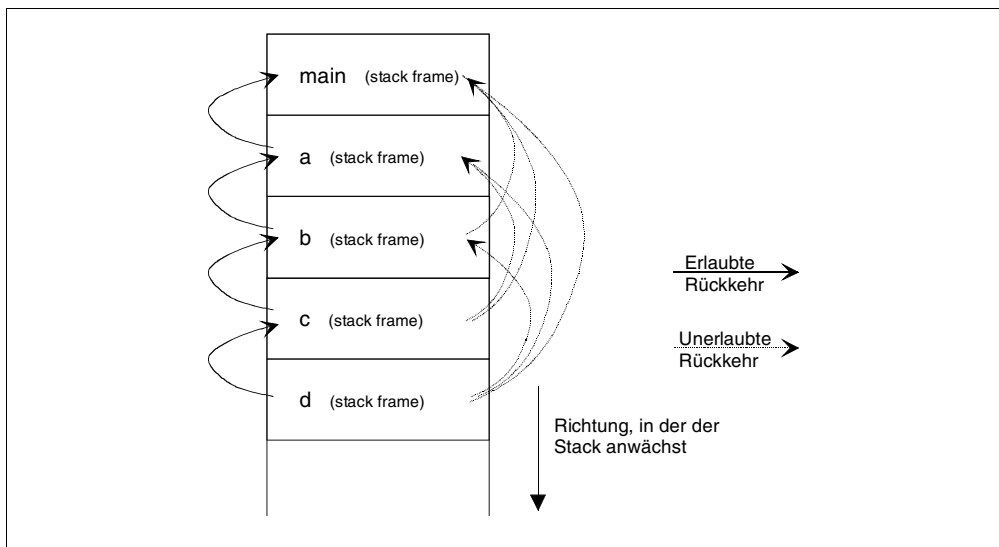


Abbildung 8.1: Erlaubte und unerlaubte Rücksprünge von Funktionen

Mit den beiden folgenden Funktionen **setjmp** und **longjmp** dagegen ist ein Rücksprung zu einem indirekten Aufrufer möglich.

8.1.1 `setjmp` und `longjmp` – Springen über Funktionsgrenzen hinweg

Mit den beiden Funktionen **`setjmp`** und **`longjmp`** ist es möglich, aus einer beliebig tief in der Aufrufhierarchie befindlichen Funktion an einen zuvor durchlaufenen und markierten Punkt (**`setjmp`**) – über mehrere Ebenen hinweg – zurückzukehren (**`longjmp`**):

```
#include <setjmp.h>

int setjmp(jmp_buf env);

gibt zurück: 0 (bei direktem Aufruf); Wert verschieden von 0 bei einer Rückkehr bedingt durch einen longjmp-Aufruf

void longjmp(jmp_buf env, int wert);
```

Um einen nicht-lokalen Sprung mit **`longjmp`** zu veranlassen, muß zuvor in einer aufrufenden Funktion mit **`setjmp`** ein Ansprungpunkt (Marke) gesetzt werden. Jeder Aufruf von **`longjmp`** in einer »tieferliegenden« Funktion bewirkt einen Rücksprung an diese mit **`setjmp`** markierte Stelle. In Abbildung 8.2 wird dies verdeutlicht, wobei angenommen wird, daß in *main* mit **`setjmp`** eine Rücksprungmarke gesetzt wurde.

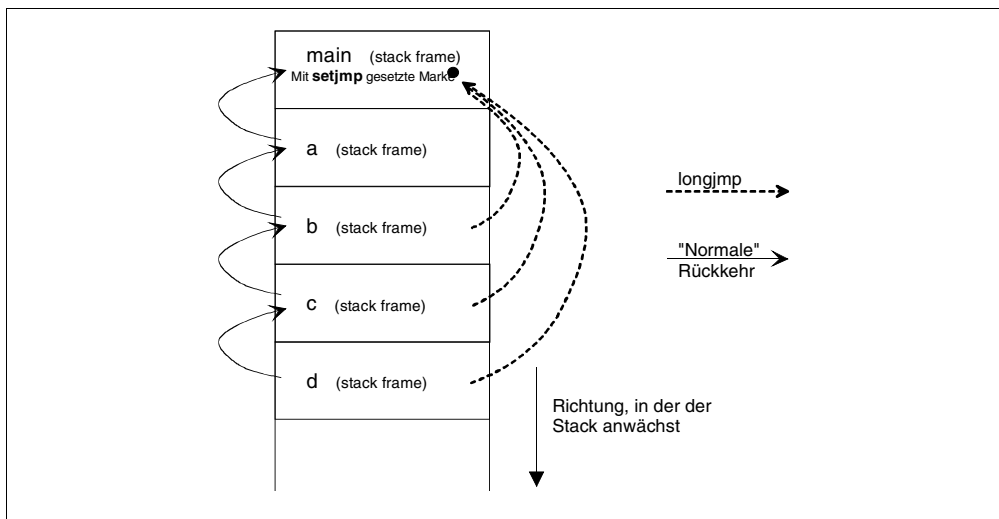


Abbildung 8.2: »Normale« und `longjmp`-Rücksprünge von Funktionen

Abbildung 8.2 zeigt einen Rücksprung zur ***main***-Funktion, aber es kann auch zu einer anderen Funktion zurückgesprungen werden, unter der Voraussetzung, daß dort mit **`setjmp`** eine Rücksprungmarke gesetzt wurde.

jmp_buf (Datentyp)

Beide Funktionen erwarten ein Argument `env` (vom Datentyp `jmp_buf`). `env` ist der Puffer, der den mit **setjmp** eingefrorenen Programzustand enthält und mit **longjmp** wieder hergestellt werden soll. Der Datentyp `jmp_buf`, der in <setjmp.h> definiert ist, ist dabei eine Art von Array, das alle Informationen¹ enthält, die notwendig sind, um den gleichen Stack-Zustand wieder herstellen zu können, der beim Aufruf von **setjmp** vorlag. Normalerweise ist `env` eine globale Variable, da meist in einer anderen Funktion auf diese Variable zugegriffen werden muß.

setjmp

Das »Funktionsmakro« **setjmp** »merkt« sich den momentanen Punkt im Programmablauf, indem es alle notwendigen Informationen im Argument `env` speichert, um an diesen Punkt zurückkehren zu können. Wird zu einem späteren Zeitpunkt die Funktion **longjmp** aufgerufen, um mit Hilfe der in `env` gemerkten Information an diese Programmstelle zurückzukehren, dann wird zum **return** des Makros **setjmp** verzweigt; d.h. von **setjmp** wird zweimal zurückgekehrt:

- ▶ das erstmal beim direkten Aufruf dieses Makros (zum Setzen der Ansprungs-marke), in diesem Fall liefert es den Wert 0 zurück;
- ▶ das zweitemal bei der Verzweigung von der Funktion **longjmp** zum Makro **setjmp**; in diesem Fall wird ein von 0 verschiedener Wert von **setjmp** zurückgegeben, um anzuzeigen, daß diese Rückkehr durch einen **longjmp**-Aufruf in einer »tieferliegenden« Funktion bewirkt wurde.

Ein portables Programm sollte **setjmp** nur in einer der folgenden Konstruktionen verwenden:

```
switch (setjmp(env))
if (setjmp(env) == 0)
if (setjmp(env) != 0)
```

longjmp

Die Funktion **longjmp** bewirkt, daß an die Programmstelle zurückgekehrt wird, die durch den letzten Aufruf von **setjmp** (im übergebenen Argument `env`) »gemerkt« wurde. Falls zuvor kein Aufruf des Makros **setjmp** stattfand, oder die Funktion, die **setjmp** aufrief, in der Zwischenzeit beendet wurde, dann liegt undefiniertes Verhalten vor.

Die Ganzzahl `wert` wird von der aufgerufenen Funktion **setjmp** als Funktionswert zurückgegeben. Die Funktion **longjmp** kann allerdings niemals bewirken, daß die Funktion **setjmp** den Wert 0 (reserviert für den direkten Aufruf von **setjmp**) zurückgibt; falls das aktuelle Argument zu `wert` gleich 0 ist, dann gibt **setjmp** den Wert 1 zurück.

1. Z.B. Registerinhalte, Stackpointer, Instruction Pointer usw.

Der Anwendungsbereich für **setjmp** und **longjmp** liegt z.B. beim Abfangen von nicht-fatalen Fehlern. Meist soll in solchen Situationen eine zentrale Fehlerbehandlungsroutine ausgeführt werden. Nach dieser Aktion (Rückkehr über mehrere Funktionen) soll sie (evtl. nach einigen Aufräumarbeiten) das Programm direkt nach der mit **setjmp** markierten Stelle als neuen »Aufsetzpunkt« wieder fortsetzen, wie es z.B. der folgende Programmausschnitt zeigt.

```
#include <setjmp.h>

jmp_buf prog_zustand;

int main( .... )
{
    .....
    if ( setjmp(prog_zustand) != 0 ) /* Rückgabewert 0 --> Schnappschuss installiert */
        non_fatal_fehler();
    eigentliches_programm();
    .....
}

void non_fatal_fehler( .... )
{
    .....
    /* Behandlung des nicht-fatalen Fehlers */
    .....
}

void eigentliches_programm( ... )
{
    .....
    if ( nonfatal_fehler_aufgetreten )
        longjmp(prog_zustand, 1);
    .....
}
.....
```

Wenn während der Ausführung der Funktion *eigentliches_programm* ein nicht-fataler Fehler auftritt, dann wird vor die Aufrufstelle von *eigentliches_programm* zurückgesprungen, dort eine Fehlermeldung ausgegeben und *eigentliches_programm* von neuem aufgerufen.

Beispiel

Umsetzung eines einfachen Taschenrechners (ohne Fehlerbehandlung)

Das nachfolgende Programm 8.1 (rechner1.c) stellt einen einfachen Taschenrechner dar, der folgende Operatoren kennt: + (Addition), - (Subtraktion), * (Multiplikation) und / (Division). Die mathematischen Ausdrücke dürfen dabei beliebig geklammert sein. Als Operanden sind dabei Gleitpunktzahlen erlaubt. Der berechnete Wert jedes in einer Zeile eingegebenen Ausdrucks wird unmittelbar wieder ausgegeben.

```

#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "eighdr.h"

#define ZAHL 256
#define PLUS 257 /* + */
#define MINUS 258 /* - */
#define MULT 259 /* * */
#define DIV 260 /* / */
#define AUF 261 /* ( */
#define ZU 262 /* ) */
#define ZEILENENDE 263

static double tokenwert;
static char *zeilen_zgr;

int lexan(void); /* Lexikalische Analyse */
double ausdruck(int *token); /* Abarbeitung eines Ausdrucks */
double term(int *token); /* " " Terms */
double factor(int *token); /* " " Factors */

int
main(void)
{
    int token;
    double ergeb;
    char zeile[MAX_ZEICHEN];

    while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
        zeilen_zgr = zeile;
        token = lexan();
        ergeb = ausdruck(&token);
        printf(".... = %.21f\n", ergeb);
    }
}

int lexan( void ) /* Lexikalische Analyse */
{
    char zeich;

    while (1) {
        zeich = *zeilen_zgr++;
        if (isdigit(zeich) || zeich=='.' ) {
            zeilen_zgr--; /* Zuviel gelesenes Zeichen zurueck in Zeile */
            tokenwert = strtod(zeilen_zgr, &zeilen_zgr);
            return(ZAHL);
        } else {
            switch (zeich) {
                case ' ':
                case '\t': break; /* Leer- und Tabzeichen ueberlesen*/
                case '\n': return(ZEILENENDE);
                case '+': return(PLUS);
            }
        }
    }
}

```

```

        case '-' : return(MINUS);
        case '*' : return(MULT);
        case '/' : return(DIV);
        case '(' : return(AUF);
        case ')' : return(ZU);
    }
}

double ausdruck( int *token )
{
    double  ergeb = term(token);

    while (1) {
        switch(*token) {
            case PLUS : *token=lexan(); ergeb += term(token); break;
            case MINUS: *token=lexan(); ergeb -= term(token); break;
            default   : return(ergeb);
        }
    }
}

double term( int *token )
{
    double erg = factor(token);

    while (1) {
        switch (*token) {
            case MULT: *token=lexan(); erg *= factor(token); break;
            case DIV : *token=lexan(); erg /= factor(token); break;
            default  : return(erg);
        }
    }
}

double factor( int *token )
{
    double erg;

    switch (*token) {
        case ZAHL : erg = tokenwert; *token=lexan(); return(erg);
        case MINUS: switch (*token=lexan()) {
            case ZAHL : erg = tokenwert; *token=lexan(); return(-erg);
        }
        case AUF  : *token=lexan(); erg=ausdruck(token);
                   *token=lexan();
                   return(erg);
    }
}

```

Programm 8.1 (rechner1.c): Umsetzung eines einfachen Taschenrechners (ohne Fehlerbehandlung)

Nachdem man dieses Programm 8.1 (rechner1.c) kompiliert und gelinkt hat

```
cc -o rechner1 rechner1.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ rechner1
2+3 *5
.... = 17.00
(2+3) * 5
.... = 25.00
10+4*(5
.... = 30.00          [Unerlaubter Ausdruck; trotzdem Ausgabe eines Ergebnisses]
4+-12*6+3
.... = -65.00
(6*(((2+3)*4)/2+100)+5)*3
.... = 1995.00
3+*4
.... = 3.00          [Unerlaubter Ausdruck; trotzdem Ausgabe eines Ergebnisses]
-7--2--3
.... = -2.00
Ctrl-D
$
```

Das Problem bei dieser Realisierung des Taschenrechners liegt hierin, daß Fehler einfach ignoriert werden. Bei Eingabe eines falschen Ausdrucks wird keine Fehlermeldung, sondern einfach ein Ergebnis ausgegeben.

Beispiel

Umsetzung des einfachen Taschenrechners (mit Fehlerbehandlung)

Tritt während der Abarbeitung eines Ausdrucks ein Fehler auf, so sollte eine Fehlermeldung ausgegeben und der restliche Teil des Ausdrucks (Rest der Zeile) ignoriert werden. In diesem Fall muß man also alle auf dem Stack befindlichen Routinen verlassen und mit der Eingabe eines neuen Ausdrucks (neue Zeile) fortfahren. Das Programm 8.2 (rechner2.c) setzt diese Art der Fehlerbehandlung um.

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <setjmp.h>
#include "eighdr.h"

#define ZAHL 256
#define PLUS 257 /* + */
#define MINUS 258 /* - */
#define MULT 259 /* * */
#define DIV 260 /* / */
#define AUF 261 /* ( */
#define ZU 262 /* ) */
#define ZEILENENDE 263
```

```

static double tokenwert;
static char *zeilen_zgr;
static jmp_buf jmppuffer;

int lexan(void); /* Lexikalische Analyse */
double ausdruck(int *token); /* Abarbeitung eines Ausdrucks */
double term(int *token); /* " " Terms */
double factor(int *token); /* " " Factors */

int
main(void)
{
    int token;
    double ergeb;
    char zeile[MAX_ZEICHEN];

    if (setjmp(jmppuffer) != 0)
        printf(".....Syntaxfehler im Ausdruck.....\n");

    while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
        zeilen_zgr = zeile;
        token = lexan();
        ergeb = ausdruck(&token);
        if (token == ZEILENENDE)
            printf(".... = %.2lf\n", ergeb);
        else
            longjmp(jmppuffer, 1);
    }
}

int lexan( void ) /* Lexikalische Analyse */
{
    char zeich;

    while (1) {
        zeich = *zeilen_zgr++;
        if (isdigit(zeich) || zeich=='.' ) {
            zeilen_zgr--; /* Zuviel gelesenes Zeichen zurueck in Zeile */
            tokenwert = strtod(zeilen_zgr, &zeilen_zgr);
            return(ZAHL);
        } else {
            switch (zeich) {
                case ' ':
                case '\t': break; /* Leer- und Tabzeichen ueberlesen*/
                case '\n': return(ZEILENENDE);
                case '+': return(PLUS);
                case '-': return(MINUS);
                case '*': return(MULT);
                case '/': return(DIV);
                case '(': return(AUF);
                case ')': return(ZU);
                default : longjmp(jmppuffer, 1);
            }
        }
    }
}

```

```

    }
}

double ausdruck( int *token )
{
    double  ergeb = term(token);

    while (1) {
        switch(*token) {
            case PLUS : *token=lexan(); ergeb += term(token); break;
            case MINUS: *token=lexan(); ergeb -= term(token); break;
            default   : return(ergeb);
        }
    }
}

double term( int *token )
{
    double erg = factor(token);

    while (1) {
        switch (*token) {
            case MULT: *token=lexan(); erg *= factor(token); break;
            case DIV : *token=lexan(); erg /= factor(token); break;
            default  : return(erg);
        }
    }
}

double factor( int *token )
{
    double erg;

    switch (*token) {
        case ZAHL : erg = tokenwert; *token=lexan(); return(erg);
        case MINUS: switch (*token=lexan()) {
                        case ZAHL : erg = tokenwert; *token=lexan(); return(-erg);
                        default  : longjmp(jmppuffer, 1);
                    }
        case AUF  : *token=lexan(); erg=ausdruck(token);
                    if (*token != ZU)
                        longjmp(jmppuffer, 1);
                    *token=lexan();
                    return(erg);
        default  : longjmp(jmppuffer, 1);
    }
}

```

Programm 8.2 (**rechner2.c**): Realisierung eines einfachen Taschenrechners (mit Fehlerbehandlung)

Nachdem man dieses Programm 8.2 (rechner2.c) kompiliert und gelinkt hat

```
cc -o rechner2 rechner2.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ rechner2
2+3    *5
.... = 17.00
(2+3) * 5
.... = 25.00
10+4*(5
.....Syntaxfehler im Ausdruck.....
4+-12*6+3
.... = -65.00
(6*(((2+3)*4)/2+100)+5)*3
.... = 1995.00
3+*4
.....Syntaxfehler im Ausdruck.....
-7--2--3
.... = -2.00
Ctrl-D
$
```

Würde man in diesem Programm 8.2 (rechner2.c) bei den einzelnen **longjmp**-Aufrufen noch unterschiedliche Werte (nicht immer 1) angeben, so könnte man sogar noch eine Fehlerklassifizierung beim **setjmp**-Aufruf vornehmen, wie z.B.

```
if ( (rwert=setjmp(jmppuffer)) != 0) {
    printf(".....Syntaxfehler ");
    switch (rwert) {
        case 1 : printf("(unvollständiger Ausdruck).....\n");
        case 2 : printf("(unerlaubtes Zeichen).....\n");
        case 3 : printf("(fehlender Operand zum Minuszeichen).....\n");
        case 4 : printf("(fehlende Klammer).....\n");
        .....
    }
}
```

8.1.2 Automatic-, register-, static- und volatile-Variable bei nicht-lokalen Sprüngen

Es stellt sich die Frage, welche Werte die einzelnen Variablen nach einem **longjmp**-Aufruf haben: Ist dies der alte Wert, der zum Zeitpunkt des **setjmp**-Aufrufs vorlag, oder ein neuer Wert, der ihnen zwischenzeitlich zugewiesen wurde.

ANSI C beantwortet diese Frage wie folgt:

- Der Inhalt von **static**-Variablen (global oder lokal) und **volatile**-Variablen (global oder lokal) entspricht immer deren Inhalt zum Zeitpunkt des **longjmp**-Aufrufs.
- Die **automatic**- und **register**-Variablen der Funktion, die **setjmp** aufrief, können in einem unbestimmten Zustand sein, wenn zwischen den Aufrufen von **setjmp** und **longjmp** ihre Inhalte verändert wurden.

Beispiel*Auswirkung von longjmp auf Variablen der unterschiedlichen Speicherklassen*

```

#include <stdlib.h>
#include <setjmp.h>
#include "eighdr.h"

int          global_var = 100;
static int   static_global_var = 100;
volatile int volatile_global_var = 100;

static jmp_buf schnapp;

void weit_sprung(void);

int main(void)
{
    int          lokal_var = 100;
    static int    static_lokal_var = 100;
    volatile int  volatile_lokal_var = 100;
    register int  register_lokal_var = 100;

    if (setjmp(schnapp) != 0) {
        printf("-----\n");
        printf("    Nach 2.Rueckkehr von setjmp\n"
               "-----\n"
               "    lokal_var = %d\nstatic_lokal_var = %d\n"
               "    volatile_lokal_var = %d\nregister_lokal_var = %d\n",
               lokal_var, static_lokal_var,
               volatile_lokal_var, register_lokal_var);
        printf("-----\n");
        printf("global_var = %d\nstatic_global_var = %d\n"
               "volatile_global_var = %d\n",
               global_var, static_global_var, volatile_global_var);
        exit(0);
    }

    printf("-----\n");
    printf("    Nach 1.Rueckkehr von setjmp\n"
           "-----\n"
           "    lokal_var = %d\nstatic_lokal_var = %d\n"
           "    volatile_lokal_var = %d\nregister_lokal_var = %d\n",
           lokal_var, static_lokal_var, volatile_lokal_var, register_lokal_var);
    printf("-----\n");
    printf("global_var = %d\nstatic_global_var = %d\n"
           "volatile_global_var = %d\n",
           global_var, static_global_var, volatile_global_var);

    /*----- Veraendern der lokalen und globalen Variablen ---*/
    lokal_var = -11111;
    static_lokal_var = -11111;
    volatile_lokal_var = -11111;

```

```

    register_lokal_var = -11111;
    global_var = -11111;
    static_global_var = -11111;
    volatile_global_var = -11111;

    weit_sprung();
    printf("Ende\n"); /* Dieser Code wird nie erreicht werden */
}

void weit_sprung(void)
{
    longjmp(schnapp, 1);

    printf("Ende: weit_sprung\n"); /* Dieser Code wird nie erreicht werden */
}

```

Programm 8.3 (*farjmp.c*): Auswirkung von **longjmp** auf Variablen der unterschiedlichen Speicherklassen

Nachdem man dieses Programm 8.3 (*farjmp.c*) kompiliert und gelinkt hat

```
cc -o farjmp farjmp.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ farjmp
-----
    Nach 1.Rueckkehr von setjmp
-----
lokal_var = 100
static_lokal_var = 100
volatile_lokal_var = 100
register_lokal_var = 100
-----
global_var = 100
static_global_var = 100
volatile_global_var = 100
-----
    Nach 2.Rueckkehr von setjmp
-----
lokal_var = -11111
static_lokal_var = -11111
volatile_lokal_var = -11111
register_lokal_var = 100
-----
global_var = -11111
static_global_var = -11111
volatile_global_var = -11111
$

```

Würde man das Programm 8.3 (farjmp.c) mit Optimierung kompilieren lassen

```
cc -O -o farjmp farjmp.c fehler.c
```

ergäbe sich z.B. folgender Ablauf:

```
$ farjmp
-----
    Nach 1.Rueckkehr von setjmp
-----
lokal_var = 100
static_lokal_var = 100
volatile_lokal_var = 100
register_lokal_var = 100
-----
global_var = 100
static_global_var = 100
volatile_global_var = 100
-----
    Nach 2.Rueckkehr von setjmp
-----
lokal_var = 100
static_lokal_var = -11111
volatile_lokal_var = -11111
register_lokal_var = 100
-----
global_var = -11111
static_global_var = -11111
volatile_global_var = -11111
$
```

Die obige Ausgabe läßt sich dadurch erklären, daß bei vielen Compilern

- alle Variablen, die sich nicht in einem Register (der CPU) befinden, den neuen Wert behalten, der beim **longjmp**-Aufruf vorliegt,
- während alle Variablen, die sich in einem Register befinden, den alten Wert erhalten, den sie beim **setjmp**-Aufruf hatten.

Im obigen Beispiel bewirkt das Kompilieren mit Optimierung (Option **-O**), daß der Compiler die Variable `lokal_var` in einem Register hält, was dazu führt, daß sie nach dem **longjmp**-Aufruf den alten Wert erhält, der beim **setjmp**-Aufruf vorlag.

Da dieses Verhalten nicht durch ANSI C abgedeckt ist, sollte man in portablen Programmen alle Variablen, die ihren neuen Wert auch nach einem **longjmp**-Aufruf behalten sollen, mit **volatile** deklarieren.

8.2 Übung

8.2.1 Mehrfaches Aufrufen von `setjmp`

Was würde das folgende Programm 8.4 (`zweijmp.c`) ausgeben ?

```
#include <setjmp.h>
#include "eighdr.h"

static jmp_buf progzust1;
static jmp_buf progzust2;

void a(void), b(void), c(void);

int
main(void)
{
    int z=2;

    if ( setjmp(progzust1) != 0)
        printf("main.....\n");

    a();
    b();
    if (--z)
        c();

    exit(0);
}

void a(void)
{
    if ( setjmp(progzust2) != 0)
        printf("a.....\n");
}

void b(void)
{
    printf(".....Rueckkehr von b ----> ");
    longjmp(progzust2, 1);
}

void c(void)
{
    printf(".....Rueckkehr von c ----> ");
    longjmp(progzust1, 1);
}
```

Programm 8.4 (`zweijmp.c`): Zweimaliges Aufrufen von **`setjmp`**

8.2.2 Rückkehr zu einer nicht mehr im Stack vorhandenen Funktion

Was würde das folgende Programm 8.5 (overjmp.c) ausgeben ?

```
#include <setjmp.h>
#include "eighdr.h"

static jmp_buf progzust;

void a(void), b(void), c(void), d(void);

int
main(void)
{
    a();
    exit(0);
}

void a(void)
{
    while (1) {
        b();
        d();
    }
}

void b(void)
{
    c();
}

void c(void)
{
    if ( setjmp(progzust) != 0)
        printf("c.....\n");
}

void d(void)
{
    printf(".....Rueckkehr von d ----> ");
    longjmp(progzust, 1);
}
```

Programm 8.5 (**overjmp.c**): **longjmp** zu einer nicht mehr aktiven Funktion

9

Der Unix-Prozeß

*Es soll sich regen, schaffend handeln,
erst sich gestalten, dann verwandeln;
nur scheinbar stehts Momente still.
Das Ewige regt sich fort in allen;
denn alles muß in Nichts zerfallen,
wenn es im Sein beharren will.*

Goethe

Der Begriff »Prozeß« läßt sich am einfachsten und verständlichsten durch folgende Definition beschreiben:

Prozeß = Programm während der Ausführung

Wird ein Programm aufgerufen, so wird der entsprechende Programmcode in den Hauptspeicher geladen und dann gestartet. Das dann ablaufende Programm wird als Prozeß bezeichnet. Wird dasselbe Programm (wie z.B. das Unix-Kommando **ls**) gleichzeitig mehrmals (z.B. von verschiedenen Benutzern) gestartet, so handelt es sich dabei um mehrere verschiedene Prozesse, obwohl alle das gleiche Programm ausführen.

In diesem Kapitel wird zunächst der Start und die Beendigung eines Unix-Prozesses beschrieben, bevor auf die Umgebung (*environment*) und Speicherbelegung eines Unix-Prozesses genauer eingegangen wird. Zum Abschluß werden die Ressourcenlimits vorgestellt, die jedem Unix-Prozeß auferlegt sind.

9.1 Start eines Unix-Prozesses

Die Ausführung eines C-Programms beginnt immer bei der Funktion **main**. Jedoch ist dieser **main**-Funktion immer eine eigene startup-Routine vorgelagert.

9.1.1 Startup-Routine – Startadresse eines Programms

Wird ein Programm vom Kern (mit einer der **exec**-Funktionen aus Kapitel 10.5) gestartet, so wird immer zuerst eine spezielle Startup-Routine (vor der eigentlichen **main**-Funktion) aufgerufen. Diese Startup-Routine, die immer vom Linker zum ausführbaren Programm gebunden wird, ist die eigentliche Startadresse des entsprechenden Programms.

Die Startup-Routine sorgt dafür, daß vor dem eigentlichen Aufruf von **main** der Prozeß mit Daten (Kommandozeilenargumente und Environment-Variablen) aus dem Kern versorgt wird.

9.1.2 main – Benutzerdefinierter Startpunkt eines Programms

Die Prototypdeklaration für *main* ist

```
int main(int argc, char *argv[]);
```

argc ist dabei die Anzahl der Argumente auf der Kommandozeile und *argv* ist ein Array von Zeigern auf die einzelnen Argumente.

Beispiel

Ausgabe aller Kommandozeilen-Argumente

```
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int i;

    for (i=0 ; i<argc ; i++) /* Ausgabe aller Kommandozeilenargumente */
        printf("argv[%d]: %s\n", i, argv[i]);

    exit(0);
}
```

Programm 9.1 (mainarg.c): Ausgabe aller Kommandozeilenargumente auf stdout

Nachdem man das Programm 9.1 (mainarg.c) kompiliert und gelinkt hat

```
cc -o mainarg mainarg.c fehler.c
```

ergeben sich beim Start z.B. folgende Abläufe:

```
$ mainarg eins ZWEI Three quatre
argv[0]: mainarg
argv[1]: eins
argv[2]: ZWEI
argv[3]: Three
argv[4]: quatre
$ ./mainarg "nur eins"
argv[0]: ./mainarg
argv[1]: nur eins
$
```

Hinweis

argv[0] ist immer das erste Argument, nämlich genau der beim Aufruf angegebene Programmname.

Sowohl ANSI C als auch POSIX.1 garantieren, daß *argv[argc]* ein NULL-Zeiger ist. Wir hätten also die Schleife aus dem Programm 9.1 (mainarg.c) auch wie folgt angeben können:

```
for (i=0 ; argv[i] != NULL ; i++)
```


9.2 Beendigung eines Unix-Prozesses

Ein Unix-Prozeß kann auf unterschiedlichste Weise beendet werden:

1. Normale Beendigung

- ▶ normales Beenden der Funktion **main** (mit oder ohne **return**)
- ▶ Aufruf der Funktionen **exit** oder **_exit**

2. Anormale Beendigung

- ▶ Aufruf der Funktion **abort**
- ▶ durch interne oder externe Signale

Wir werden uns hier nur mit der normalen Beendigung eines Prozesses beschäftigen. Die anormale Beendigung eines Prozesses mittels **abort** oder durch ein Signal wird ausführlich in Kapitel 13 besprochen.

9.2.1 Exit-Status eines Prozesses

Jeder Prozeß hat einen *Exit-Status*, den er bei seiner Beendigung an den aufrufenden Prozeß zurückgibt. Es zeugt von einem sauberen Programmierstil, wenn jedes Programm einen Exit-Status liefert. Beendet man ein Programm ohne die Rückgabe eines Exit-Status, so ist dieser undefiniert, was andere Prozesse (wie z.B. Shell-Skripts), die sich auf den Exit-Status verlassen, in Schwierigkeiten bringen kann.

Der Exit-Status für ein Programm ist in folgenden Fällen nicht definiert:

- ▶ Automatische Rückkehr aus der Funktion **main** durch Beendigung des Codes.
- ▶ Aufruf von

```
return; /* Keine Angabe eines Rückgabewerts */
in main.
```

- ▶ Aufruf von

```
exit;
```

oder

```
_exit;
```

im Programm.

So ist z.B. beim folgenden Programm 9.2 (*noexstat.c*) der Exit-Status undefiniert.

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    printf("-----\n");
```

```

printf(".....Ich habe keinen exit-status, das ist schlecht!!!\n");
printf("-----\n");
}

```

Programm 9.2 (noexstat.c): »Unsauberes« Programm ohne Exit-Status

Nachdem man das Programm 9.2 (noexstat.c) kompiliert und gelinkt hat

```
cc -o noexstat noexstat.c
```

gibt es folgendes aus:

```

$ noexstat
-----
.....Ich habe keinen exit-status, das ist schlecht!!!
-----
$

```

Es gibt also das Erwartete aus und scheint damit richtig zu sein. Rufen wir dieses Programm aber aus einem Shell-Skript heraus auf, und erfragen seinen Exit-Status, dann treten Schwierigkeiten auf.

```

$ cat teste
echo "Ausfuehrung von noexstat"
if noexstat
then
    echo "war erfolgreich"
else
    echo "ging schief"
fi
$ chmod u+x teste
$ teste
Ausfuehrung von noexstat
-----
.....Ich habe keinen exit-status, das ist schlecht!!!
-----
ging schief
$

```

Der fehlende und damit undefinierte Exit-Status führt also dazu, daß hier angenommen wird, daß das Programm noexstat nicht erfolgreich ablief.

Um dieses Programm zu vervollständigen, müßte vor der abschließenden geschweiften Klammer entweder

```
exit(0);
```

oder

```
_exit(0);
```

oder

```
return(0);
```

angegeben werden, was dazu führt, daß dieses Programm bei erfolgreichem Ablauf dem aufrufenden Prozess den Exit-Status 0 (erfolgreich) liefert.

Ein weiterer Kritikpunkt an dem obigen Programm, das nach dem früher gängigen und spätestens seit ANSI C veralteten C-Programmiersstil erstellt wurde, ist die Angabe:

```
main()
```

Hierfür sollte man folgendes angeben:

```
int main(void)
```

9.2.2 Normales Beenden der Funktion **main** mit **return**

Die in Kapitel 9.1 erwähnte Startup-Routine ist nicht nur für den Start eines Prozesses zuständig, sondern auch für seine Beendigung, wenn die Funktion **main** sich »ganz normal« wie jede andere Funktion beendet: durch Erreichen des Code-Endes, was nicht empfehlenswert ist (wegen fehlendem Exit-Status), oder durch einen expliziten Aufruf von **return**. Wenn die Startup-Routine in C geschrieben ist, kann sie den Aufruf von **main** wie folgt durchführen:

```
exit( main(argc, argv) );
```

Hinweis

Die Startup-Routine ist meist (aus Performancegründen) in Assembler geschrieben.

9.2.3 **exit** – Normales Beenden eines Programms mit **cleanup**

Um ein Programm normal zu beenden, wobei zuvor jedoch noch einige »Aufräumarbeiten« durchgeführt werden (wie z.B. alle noch nicht auf Dateien geschriebenen Pufferinhalte auch wirklich physikalisch schreiben), steht die Funktion **exit** zur Verfügung.

```
#include <stdlib.h>

void exit(int status);
```

Diese von ANSI C vorgeschriebene Funktion bewirkt eine normale Programmbeendigung, wobei sie jedoch zuvor noch alle gefüllten Puffer leert, alle geöffneten Dateien schließt und alle temporären Dateien, die mit der Funktion **tmpfile** angelegt wurden, löscht.

Hinweis

Nach dem **cleanup** ruft **exit** seinerseits die Routine **_exit** auf, um den Prozeß zu beenden und zum Kern zurückzukehren. In Kapitel 10.3 wird genauer auf diese Funktion eingegangen.

9.2.4 `_exit` – Normales Beenden eines Programms ohne cleanup

Um ein Programm normal zu beenden, wobei jedoch keinerlei »Aufräumarbeiten« wie bei **`exit`** durchgeführt werden, steht die Funktion **`_exit`** zur Verfügung.

```
#include <unistd.h>

void _exit(int status);
```

Diese von POSIX.1 vorgeschriebene Funktion bewirkt eine sofortige Programmbeendigung und Rückkehr zum Kern.

Hinweis

In Kapitel 10.3 wird genauer auf diese Funktion eingegangen.

9.2.5 `atexit` – Einrichten von Exithandlern

ANSI C hat eine neue Funktion **`atexit`** eingeführt, mit der bis zu 32 Funktionen registriert werden können, die automatisch bei Beendigung eines Prozesses aufgerufen werden:

```
#include <stdlib.h>

int atexit(void (*funktion)(void));
```

gibt zurück: 0 (bei Erfolg); Wert verschieden von 0 bei Fehler

`atexit` trägt die Funktion, auf die *funktion* (Funktionsname) zeigt, in die Liste von Funktionen ein, die bei normaler Programmbeendigung aufzurufen sind. Solche Funktionen bezeichnet man auch als *Exithandler*.

Die mit **`atexit`** registrierten Funktionen (Exithandler) werden bei der Programmbeendigung automatisch in umgekehrter Reihenfolge zur Registrierung aufgerufen. Bei diesem automatischen Aufruf werden keinerlei Argumente an diese Funktionen übergeben und es wird auch kein Rückgabewert erwartet. Jede Funktion wird dabei so oft aufgerufen, wie sie registriert wurde.

Beispiel

Demonstrationsprogramm zur Funktion `atexit`

```
#include <stdlib.h>
#include <time.h>
#include "eighdr.h"

static void goodbye(void), tschuess(void), kopfrech(void);

int
```

```

main(void)
{
    if (atexit(tschuess) != 0)
        fehler_meld(FATAL_SYS, "Installation von Exithandler 'tschuess'"
                    " misslang");

    if (atexit(goodbye) != 0)
        fehler_meld(FATAL_SYS, "Installation von Exithandler 'goodbye' misslang");

    if (atexit(kopfrech) != 0)
        fehler_meld(FATAL_SYS, "Installation von Exithandler 'kopfrech'"
                    " misslang");

    if (atexit(goodbye) != 0)
        fehler_meld(FATAL_SYS, "Installation von Exithandler 'goodbye' misslang");

    printf(".... Funktion main ist beendet ....\n\n");

    exit(0); /* return(0) waere auch moeglich,
              _exit(0) dagegen wuerde die Exithandler nicht aufrufen */
}

static void goodbye(void)
{
    printf("\nGood Bye");
}

static void tschuess(void)
{
    printf(" und ....T s c h u e s s\n");
}

static void kopfrech(void)
{
    int x, y, sum, ergebn;

    srand(time(NULL)); /* Initialisieren des Zufallszahlengenerators */
    x = rand()%100+1; /* 2 Zufallszahlen aus Intervall [1,100] ermitteln */
    y = rand()%100+1;
    sum = x+y;

    printf("\n\nZum Abschluss eine kleine Rechenaufgabe: %d + %d = ", x, y);
    scanf("%d", &ergebn);
    if (sum == ergebn)
        printf(" Richtig!!!!\n\n");
    else
        printf(" Leider Falsch!\n\n %d + %d = %d\n", x, y, sum);
}

```

Programm 9.3 (atexit.c): Beispielprogramm zur Funktion **atexit**

Nachdem man das Programm 9.3 (atexit.c) kompiliert und gelinkt hat

```
cc -o atexit atexit.c fehler.c
```

zeigt es folgenden Ablauf:

```
$ atexit
.... Funktion main ist beendet .....
```

Good Bye

```
Zum Abschluss eine kleine Rechenaufgabe: 69 + 55 = 125
      Leider Falsch!
      69 + 55 = 124
```

```
Good Bye und ....T s c h u e s s
$
```

Hinweis

atexit wurde erst von ANSI C eingeführt, so daß diese Funktion in früheren Unix-Systemen, die über keinen ANSI C-Compiler verfügen, nicht vorhanden ist. Bei neueren Systemen mit ANSI C-Compilern – wie SVR4 – ist diese Funktion verfügbar.

9.2.6 Start und Beendigung eines Benutzerprozesses

Die Abbildung 9.1 faßt zusammen, wie ein Benutzerprozeß vom Kern gestartet wird und wie er beendet werden kann.

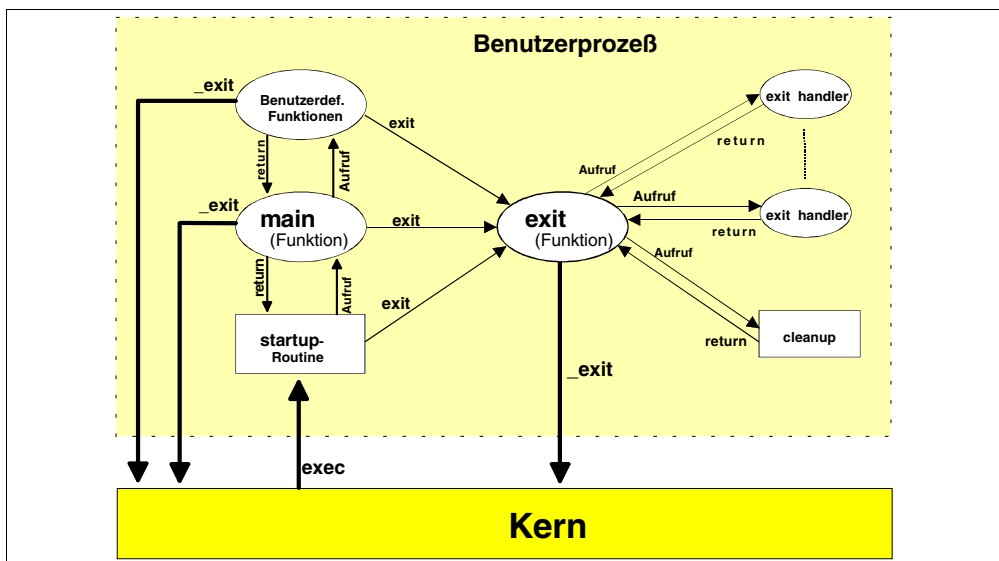


Abbildung 9.1: Überblick über Start und normale Beendigung eines Prozesses

In Abbildung 9.1 ist zu erkennen, daß ein Unix-Prozeß immer mit einem Aufruf der in Kapitel 10.5 beschriebenen **exec**-Funktionen gestartet wird, und er sich immer nur mit einem **_exit** (explizit oder implizit über **exit** oder **return** in **main**) beenden kann.

Neben dieser normalen Beendigung eines Prozesses besteht noch die Möglichkeit, daß ein Prozeß anormal beendet (durch **abort**-Aufruf oder ein Signal) wird. Dies ist in Abbildung 9.1 nicht berücksichtigt, wird aber in Kapitel 13 ausführlich beschrieben.

9.3 Environment eines Unix-Prozesses

Jeder Unix-Prozeß besitzt seine eigene Umgebung (*environment*). Diese Environment liegt in Form einer Liste vor, die ihm von der Startup-Routine übergeben wird.

9.3.1 Environment-Liste

Die Environment-Liste ist – wie die Argumenten-Liste (*argv*) – ein Array von Zeigern auf Strings. Die Strings sind – wie bei *argv* – mit `\0` abgeschlossen sind. Die Adresse dieser Environment-Liste ist immer in der globalen Variablen *environ* enthalten:

```
extern char **environ;
```

Abbildung 9.2 zeigt ein Beispiel einer Environment-Liste mit 6 Strings. Die Environment eines Unix-Prozesses besteht aus Strings der folgenden Form

```
name=wert
```

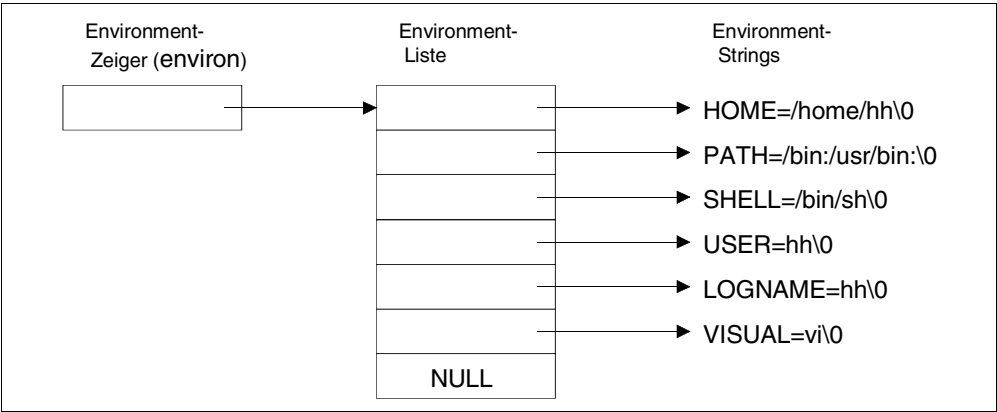


Abbildung 9.2: Environment-Liste mit 6 Strings

9.3.2 Zugriff auf die ganze Environment-Liste

Um die ganze Environment-Liste in einem Prozeß zu durchlaufen und dabei auf alle einzelnen Einträge zuzugreifen, gibt es zwei Möglichkeiten:

1. Zugriff über die globale Variable *environ*.

Das folgende Programm 9.4 (*envlist1.c*) zeigt diese Möglichkeit, indem es die ganze Environment-Liste mit Hilfe von *environ* durchläuft und alle Einträge aus dieser Liste auf der Standardausgabe ausgibt.

```
#include "eighdr.h"

extern char **environ;

int
main(int argc, char *argv[])
{
    int i;

    for (i=0 ; environ[i] != NULL ; i++)
        printf("%s\n", environ[i]);

    exit(0);
}
```

Programm 9.4 (envlist1.c): Ausgabe der ganzen Environment-Liste mit Hilfe von environ

Nachdem man das Programm 9.4 (*envlist1.c*) kompiliert und gelinkt hat

```
cc -o envlist1 envlist1.c fehler.c
```

liefert es z.B. die folgende Ausgabe:

```
$ envlist1
HOME=/home/hh
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/etc:/usr/etc:/usr/local/bin:/usr/bin/X11:/usr/openwin/
bin:/home/hh/bin:.
SHELL=/bin/sh
TERM=console
USER=hh
MAIL=/var/spool/mail/hh
LOGNAME=hh
PWD=/home/hh/work
HOST=hh
PRINTER=lp
EDITOR=vi
VISUAL=vi
PAGER=less
MANPATH=/usr/man:/usr/man/preformat:/usr/X11/man:/usr/openwin/man
OPENWINHOME=/usr/openwin
.....
.....
$
```


2. Zugriff über ein drittes Argument in der *main*-Funktion.

Das folgende Programm 9.5 (*envlist2.c*) zeigt diese zweite Möglichkeit, indem es die ganze Environment-Liste mit Hilfe eines dritten Arguments in **main** (*envp*) durchläuft und alle Einträge aus dieser Liste auf der Standardausgabe ausgibt. Es leistet das gleiche wie das Programm 9.4 (*envlist1.c*).

```
#include "eighdr.h"

int
main(int argc, char *argv[], char *envp[])
{
    int i;

    for (i=0 ; envp[i] != NULL ; i++)
        printf("%s\n", envp[i]);

    exit(0);
}
```

Programm 9.5 (**envlist2.c**): Ausgabe der ganzen Environment-Liste mit Hilfe eines dritten *main*-Arguments

Hinweis

Die zweite Möglichkeit ist heute veraltet, da ANSI C festlegt, daß die **main**-Funktion nur zwei Argumente hat. Deshalb ist die erste Möglichkeit (Zugriff über die globale Variable *environ*) der zweiten Möglichkeit (mit drittem **main**-Argument) vorzuziehen. POSIX.1 legt deshalb auch fest, daß immer von der ersten Möglichkeit Gebrauch gemacht werden sollte.

Um auf spezielle Environment-Variablen zuzugreifen, sollten immer die eigens dafür vorgesehenen Funktionen **getenv** und **putenv**, die nachfolgend beschrieben sind, verwendet und niemals die globale Variable *environ* herangezogen werden.

9.3.3 **getenv** – Erfragen des Werts einer einzelnen Environment-Variablen

Die einzelnen Einträge in der Environment-Liste sind – wie schon früher erwähnt – Strings der folgenden Form:

name=wert

Die in der Environment-Liste angegebenen *namen* der Variablen haben keinerlei Bedeutung für den Kern, sie werden von den entsprechenden Applikationen festgelegt. So gibt z.B. die Shell Variablenamen vor, die sie entweder selbst mit Werten belegt (wie *TERM*, *LOGNAME* usw.) oder aber den Benutzer mit Werten belegen läßt (wie *PATH*, *CDPATH*, *MAIL-PATH*, usw.).¹

1. Siehe Band »Linux-Unix-Shells«.

Um den *wert* zu einer bestimmten Variablen *name* zu erfragen, steht die ANSI-C-Funktion **getenv** zur Verfügung.

```
#include <stdlib.h>
```

```
char *getenv(const char *name);
```

gibt zurück: Zeiger auf den zu *name* gehörigen *wert* (wenn *name* vorhanden); sonst NULL-Zeiger

ANSI C macht bezüglich **getenv** noch folgende Einschränkungen:

- ▶ Ein streng portables Programm sollte nicht den Speicherplatz modifizieren, den **getenv** verwendet. Die Adresse dieses Speicherplatzes wird als Rückgabewert geliefert.
- ▶ Ebenso ist zu beachten, daß ein späterer Aufruf von **getenv** denselben Speicherplatz wieder verwenden kann, was zum Verlust des alten Inhalts führt. Deshalb ist es empfehlenswert, den von **getenv** zurückgegebenen String vor einem erneuten **getenv**-Aufruf in einen eigenen Speicherplatz zu kopieren, wenn dieser String später noch benötigt wird.

Hinweis

ANSI C schreibt keinerlei Namen von Environment-Variablen vor. Es hängt von der jeweiligen Implementierung ab, welche Environment-Variablen definiert sind.

9.3.4 putenv, setenv und unsetenv – Ändern, Hinzufügen oder Löschen von Environment-Variablen

Um in der Environment-Liste Einträge zu ändern, neue Einträge hinzuzufügen oder Einträge zu löschen, stehen die Funktionen **putenv**, **setenv** und **unsetenv** zur Verfügung.

```
#include <stdlib.h>
```

```
int putenv(const char *eintrag);
```

```
int setenv(const char *name, const char *wert, int ueberschreib);
```

beide geben zurück: 0 (bei Erfolg); Wert verschieden von 0 bei Fehler

```
void unsetenv(const char *name);
```

putenv

putenv nimmt den String *eintrag*, der die Form *name=wert* haben muß, und trägt ihn in die Environment-Liste ein. Falls *name* bereits existiert, wird dessen alte Definition zuvor aus der Environment-Liste entfernt.

setenv

setenv macht in der Environment-Liste einen Eintrag der Form *name=wert*. Falls *name* bereits existiert, wird dessen alte Definition nur dann aus der Environment-Liste entfernt, wenn *ueberschreib* einen Wert verschieden von 0 hat, andernfalls bleibt die Environment-Liste unverändert, was nicht als Fehler gewertet wird.

unsetenv

unsetenv löscht in der Environment-Liste den zum angegebenen *namen* gehörigen Eintrag. Es wird nicht als Fehler gewertet, wenn ein solcher Eintrag nicht existiert.

Hinweis

Während SVR4 nur die beiden Funktionen **getenv** und **putenv** kennt, bietet das neue BSD-Unix alle vier Funktionen **getenv**, **putenv**, **setenv** und **unsetenv** an.

Die folgenden beiden Aufrufe bewirken genau das gleiche: Sie ändern die aktuelle Environment-Variable **PATH** für das aktuell ablaufende Programm:

```
putenv("PATH=/bin:/usr/bin:.");  
setenv("PATH", "/bin:/usr/bin:.", 1);
```

In Zukunft wird wohl POSIX.1 eine weitere Funktion **clearenv** aufnehmen, die das Löschen der ganzen Environment-Liste ermöglicht.

9.4 Speicherbelegung eines Unix-Prozesses

Wird ein Programm aufgerufen, so wird zunächst der entsprechende Programmcode in den Hauptspeicher geladen.

9.4.1 Unix-Prozeß im Hauptspeicher

Ein Unix-Prozeß setzt sich üblicherweise aus den in Abbildung 9.3 gezeigten Teilen zusammen.

Bei Abbildung 9.3 handelt es sich um eine typische, aber nicht allgemeingültige Möglichkeit der Speicheranordnung für einen Prozeß. Die einzelnen Segmente aus Abbildung 9.3 haben dabei die folgende Bedeutung:

text segment

Das *text segment* enthält den ausführbaren Maschinencode und ist normalerweise *sharable*, was bedeutet, daß es von mehreren Prozessen gleichzeitig benutzt werden kann. Wenn beispielsweise der C-Compiler zur gleichen Zeit von mehreren Benutzern aufgerufen wird, so werden zwar mehrere Prozesse gestartet, im Speicher wird aber, um nicht unnötig kostbaren Speicherplatz zu vergeuden, der ausführbare Maschinencode des C-Compilers nur einmal abgelegt. Die einzelnen Prozesse teilen (*share*) sich also das gleiche Textsegment.

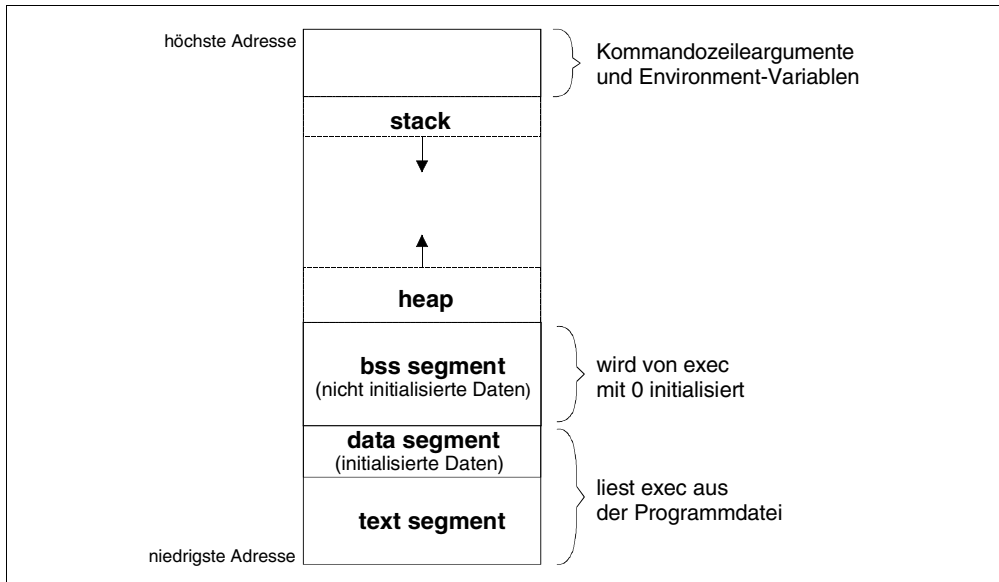


Abbildung 9.3: Typisches Aussehen eines Unix-Prozesses im Speicher

Um zu verhindern, daß ein Prozeß versehentlich (oder auch absichtlich) den Maschinencode verändert, ist das Textsegment meist auch nur lesbar (*read only*).

data segment

Das *data segment* enthält alle Daten, die bereits bei globalen Deklarationen (außerhalb einer Funktion) im C-Programm mit Daten vorbesetzt wurden, wie z.B.

```
int    summe = 0;
char  *meldung = ".....Bitte Diskette einlegen";
unsigned besucher[1000] = {0};
```

bss segment

Der Name *bss segment* stammt von einem früheren Assembler-Operator **bss** (*block started by symbol*). Daten dieses Segments werden vom Kern beim Prozeßstart mit 0 initialisiert. In diesem Segment befinden sich alle globalen Variablen (Deklaration befindet sich außerhalb einer Funktion), die nicht explizit mit Werten vorbesetzt sind, wie z.B.

```
int    i;
char  *zgrrl;
double umsatz[100];
```

stack

Im *Stack* werden alle *automatic* Variablen (lokalen Variablen) einer Funktion abgelegt, jedesmal wenn diese aufgerufen wird.

Jedesmal, wenn eine Funktion aufgerufen wird, werden die Rückkehradresse sowie weitere benötigten Daten des Aufrufers auf dem Stack abgelegt. Danach legt die aufgerufene Funktion ihre *automatic* Variablen auf dem Stack ab.

heap

Fordert ein Prozeß während seines Ablaufs neuen (dynamischen) Speicher an, so wird ihm dieser in seinem *Heap*-Bereich zugeteilt.

Hinweis

Die Inhalte von *text segment* und *data segment* sind in einer Programmdatei enthalten. Die Inhalte des *bss segment* sind dagegen nicht in der entsprechenden Programmdatei gespeichert. Der Kern setzt diesen Bereich beim Start des Programms auf 0.

Mit dem Kommando **size** läßt sich die Bytegröße der *text*-, *data*- und *bss*-Segmente eines Programms ausgeben, wie z.B.

```
$ size /bin/c*
text    data    bss      dec      hex           /bin/cat
4644    160      32      4836    12e4
3652    120      40      3812    ee4          /bin/chgrp
6104    120      40      6264    1878         /bin/chmod
4516    128      48      4692    1254         /bin/chown
8192    4096     410304   422592   672c0        /bin/compress
13992   240       56      14288   37d0         /bin/cp
36864   4096       0       40960   a000         /bin/cpio
196608  12288     57096   265992  40f08        /bin/csh
5036    120       64       5220    1464         /bin/cut
$
```

Die *dec*-Spalte zeigt die Gesamtgröße dezimal und die *hex*-Spalte hexadezimal an.

9.4.2 malloc, calloc, realloc – Dynamisches Anfordern von Speicherplatz

ANSI C stellt die drei Funktionen **malloc**, **calloc** und **realloc** zur dynamischen Speicheranforderung zur Verfügung.

```
#include <stdlib.h>
```

```
void *malloc(size_t groesse);
```

```
void *calloc(size_t anzahl, size_t groesse);
```

```
void *realloc(void *zgr, size_t neuegroesse);
```

alle drei geben zurück: Adresse des allokierten Speicherbereichs (bei Erfolg); NULL bei Fehler

Die von den drei Funktionen **malloc**, **calloc** und **realloc** zurückgegebene Adresse ist für die Speicherung jedes beliebigen Datenobjekts geeignet.

Da alle drei Funktionen einen generischen Zeiger (`void *`) als Rückgabewert liefern, muß man kein *casting* verwenden, wenn man diese zurückgegebene Adresse einer Zeigervariablen eines anderen Datentyps zuweist.

malloc

reserviert (allokiert) einen Speicherbereich mit *groesse* Bytes. Die Bytes dieses Speicherbereichs haben keine definierten Werte als Inhalt, da **malloc** – anders als **calloc** – sie nicht mit Wert 0 initialisiert.

calloc

reserviert (allokiert) einen Speicherbereich für *anzahl* Objekte mit *groesse* Bytes. Alle Bytes dieses Speicherbereichs werden dabei mit dem Wert 0 initialisiert.

realloc

verändert die Größe eines bereits zuvor allokierten Speicherbereichs (*zgr* ist seine Anfangsadresse) auf *neuegroesse* Bytes.

Bei einer Verkleinerung wird der hintere Teil des ursprünglichen Speicherplatzes freigegeben, der Inhalt des vorderen Teils bleibt unverändert erhalten.

Bei einer Vergrößerung, was der häufigste Anwendungsfall ist, behält in jedem Fall der »vordere alte« Teil seine ursprünglichen Werte, während der Inhalt des »angehängten neuen« Teils undefiniert ist, also nicht explizit (wie bei **calloc**) mit 0 vorbesetzt wird.

Bei einer Vergrößerung muß jedoch möglicherweise der ganze Inhalt des alten Speicherbereichs zuvor in einen größeren neuen Speicherbereich umkopiert werden. Wenn z.B. ursprünglich ein Speicherplatz für 1000 Elemente eines Arrays allokiert wurde, aber während des Programmlaufs mehr als 1000 Elemente zu speichern sind, so kann dieser Speicherplatz nachträglich mit **realloc** vergrößert werden. Wenn noch genügend Platz hinter dem alten Speicherbereich vorhanden ist, dann kann **realloc** den zusätzlich geforderten Speicherplatz dort hinzufügen, was das Umkopieren erspart. In diesem Fall liefert die Funktion **realloc** die gleiche Adresse zurück, die ihr als Argument für *zgr* übergeben wurde. Sollte aber hinter dem alten Speicherbereich nicht mehr genügend freier Speicherplatz vorhanden sein, so muß die Funktion **realloc** zunächst einen zusammenhängenden freien Speicherbereich mit *neuegroesse* Bytes finden und allokieren, die bereits gespeicherten 1000 Elemente dorthin kopieren, und dann den alten Speicherplatz freigeben, bevor sie die Adresse des neuen Speicherbereichs zurückgibt. Diese interne Arbeitsweise sollte man kennen, denn dann wird auch verständlich, warum keine Zeiger gehalten werden sollten, die Adressen aus einem solchen Speicherbereich enthalten, denn diese Adressen sind – für den Fall eines Umkopierens – nicht weiter verwendbar.

ANSI C schreibt zusätzlich vor, daß die beiden folgenden Aufrufe identisch sind

```
realloc(NULL, groesse)
malloc(groesse)
```

Jedoch sollte man diese Besonderheit von ANSI C nur bei ANSI-C-Compilern verwenden, bei älteren Compilern kann diese Aufrufform zu äußerst seltsamen Verhalten führen.

Auch sind die beiden folgenden Aufrufe identisch

```
realloc(adresse, 0)
free(adresse)
```

Hinweis

Es zeugt von einem sauberen Programmierstil, daß man den Rückgabewert von **malloc**, **calloc** und **realloc** immer überprüft, und sich nicht auf das Vorhandensein von genügendem Speicherplatz verläßt. Eine typische Allokierung sieht z.B. wie folgt aus:

```
if ( (adr = malloc(100000)) == NULL)
    fehler_meld(FATAL_SYS, "Speicherplatzmangel");
```

Ein häufiger Fehler ist, daß man in einer Funktion neuen Speicherplatz mit einer der drei obigen Funktionen allokiert und die zurückgegebene Adresse in einer lokalen Zeigervariablen dieser Funktion speichert. Da nach dem Verlassen der Funktion diese lokale Zeigervariable nicht mehr gültig ist, ist es nicht mehr möglich, auf den reservierten Speicherplatz zuzugreifen. Man kann ihn sogar nicht mehr freigeben, da seine Adresse nun unbekannt ist.

Die Funktionen **malloc**, **calloc** und **realloc** verwenden intern die Funktion **sbrk**. Diese Funktion kann den *Heap* eines Prozesses vergrößern oder verkleinern.

Die meisten Implementierungen dieser Funktionen allokieren etwas mehr Speicherplatz, als wirklich gefordert, und benutzen den zusätzlichen Speicherplatz für verwaltungstechnische Informationen (wie z.B. Größe des allokierten Speicherblocks, Zeiger auf den nächsten allokierten Speicherblock usw.). Dies bedeutet, daß das Schreiben über einem reservierten Speicherplatz hinaus dazu führen kann, daß die interne Information des nächsten Speicherblocks überschrieben wird. Dies hat meist fatale Folgen. Erschwerend kommt hinzu, daß Fehler dieser Art schwer aufzufinden sind, da sie meist erst später im Einsatz des Softwareprodukts (bei größeren Anwendungen) und auch dann nur sporadisch auftreten.

Da Programmfehler bei der dynamischen Speichieranforderung nur schwer auffindbar sind, bieten einige Systeme inzwischen in eigenen Bibliotheken verbesserte Versionen dieser Funktionen an, die eine zusätzliche Fehlerprüfung durchführen, wenn eine der Funktionen **malloc**, **calloc**, **realloc** oder **free** (siehe unten) aufgerufen wird.

Beispiel*Demonstrationsprogramm zu den Funktionen malloc und realloc*

Das folgende Programm 9.6 (primza.c) berechnet die Primzahlen zwischen 1 und n (n ist dabei einzugeben). Es verwendet dabei sicherlich nicht den elegantesten Algorithmus, sondern das Sieb des Eratosthenes. Dieser Algorithmus ist sehr speicheraufwendig, da er zunächst alle natürlichen Zahlen zwischen 1 und n speichert, bevor er alle Nicht-Primzahlen aus dem Array streicht.

Zunächst wird dabei Speicherplatz für 100 Werte (Primzahlen bis 100) reserviert. Wenn dieser Speicherplatz nicht ausreicht, wird mit **realloc** der vorreservierte Speicherplatz immer wieder vergrößert.

```
/*-----
 * Dieses Programm berechnet Primzahlen bis zu einem bestimmten Wert, der
 * einzugeben ist.
 * Zunaechst wird Speicherplatz fuer 100 Werte (Primzahlen bis 100)
 * reserviert.
 * Wenn dieser Speicherpl. nicht ausreicht, wird mit realloc "nachallokiert".
 * Bei jedem neuen Durchlauf ist zu pruefen, ob bisher reservierter
 * Speicherpl.
 * ausreicht (ueber max nachpruefbar), ansonsten wird "nachallokiert".
 * Es wird immer max+1 allokiert, um Indizierung bei 1 beginnen zu lassen.
 *-----*/
#include <stdlib.h>
#include "eighdr.h"

int
main(void)
{
    long int    max=100, i, j, ende, *array;

    /*--- Speicherplatz fuer 100 Werte (Voreinstellung) reservieren -----*/
    if ( (array=malloc((max+1)*sizeof(long int))) == NULL)
        fehler_meld(FATAL_SYS, "Speicherplatzmangel");

    while (1) {

        /*-- Einlesen, bis wohin Primzahlen zu berechnen sind (Ende = 0)--*/
        printf("Bis wohin sollen die Primzahlen berechnet werden (Ende=0) ? ");
        scanf("%ld", &ende);
        if (ende==0)
            break;

        /*-- Im Bedarfsfall (ende>max) Speicherpl. vergroessern (realloc)--*/
        if (ende>max) {
            max = ende;
            if ( (array=realloc(array,(max+1)*sizeof(long int))) == NULL)
                fehler_meld(FATAL_SYS, "Speicherplatzmangel");
        }

        /*-- Primzahlen nach Sieb des Eratosthenes berechnen und ausgeben--*/
```



```

    for (i=1 ; i<=ende ; i++)
        array[i] = i;
    for (i=2 ; i<=ende/2 ; i++)
        if (array[i])
            for (j=2*i ; j<=ende ; j += i)
                array[j] = 0;
    for (i=2 ; i<=ende ; i++)
        if (array[i])
            printf("%10ld", i);
    printf("\n");
}

exit(0);
}

```

Programm 9.6 (**primza.c**): Berechnung der Primzahlen nach dem Sieb des Eratosthenes

Nachdem man dieses Programm 9.6 (primza.c) kompiliert und gelinkt hat

```
cc -o primza primza.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ primza
Bis wohin sollen die Primzahlen berechnet werden (Ende=0) ? 70
    2      3      5      7      11      13      17      19
   23     29     31     37     41     43     47     53
   59     61     67
Bis wohin sollen die Primzahlen berechnet werden (Ende=0) ? 10000
    2      3      5      7      11      13      17      19
   23     29     31     37     41     43     47     53
   59     61     67     71     73     79     83     89
   97    101    103    107    109    113    127    131
  137    139    149    151    157    163    167    173
.....
.....
.....
.....
  9739    9743    9749    9767    9769    9781    9787    9791
  9803    9811    9817    9829    9833    9839    9851    9857
  9859    9871    9883    9887    9901    9907    9923    9929
  9931    9941    9949    9967    9973
Bis wohin sollen die Primzahlen berechnet werden (Ende=0) ? 10000000
Speicherplatzmangel: Out of memory
$

```

Dieses Programm 9.6 (primza.c) zeigt im übrigen auch eine Programmiertechnik, um in C »dynamische Arrays« nachzubilden. Die Vorgehensweise ist dabei die folgende:

1. Man deklariert einen Zeiger vom Typ der entsprechenden Array-Elemente, im obigen Beispiel:

```
long int *array;
```

2. Nachdem man die erforderliche Größe des Arrays kennt, allokiert man mit einer der Funktionen **malloc**, **calloc** oder **realloc** den benötigten Speicherplatz, und weist dessen Anfangsadresse dem zuvor deklarierten Zeiger zu, im obigen Beispiel mit

```
array = malloc((max+1)*sizeof(long int))   bzw.  
array = realloc(array,(max+1)*sizeof(long int))
```

3. Nun kann man den allokierten Speicherbereich (mittels des Zeigers) wie ein Array behandeln. Um z.B. im obigen Beispiel auf die *i*.te Zahl im allokierten Speicherbereich zuzugreifen, muß man nur

```
array[i]  
angeben.
```

9.4.3 free – Freigeben von dynamisch angefordertem Speicherplatz

ANSI C stellt zur Freigabe von dynamisch angefordertem Speicherplatz die Funktion **free** zur Verfügung.

```
#include <stdlib.h>  
  
void free(void *zgr);
```

Die Funktion **free** gibt den Speicherbereich, auf den *zgr* zeigt, wieder frei. Der frei gewordene Speicherbereich kann bei späteren Speicheranforderungen wieder vergeben werden.

Falls für *zgr* eine Adresse eines Speicherbereichs angegeben wird, der nicht zuvor mit **malloc**, **calloc** oder **realloc** allokiert wurde, oder wenn die für *zgr* angegebene Adresse auf einen Speicherbereich zeigt, der zuvor mit `free(zgr)` oder `realloc(zgr,0)` wieder freigegeben wurde, dann liegt laut ANSI C *undefiniertes Verhalten* vor. In der praktischen Anwendung kann dies katastrophale Folgen für den Prozeß haben, da die ganze Speicherverwaltung inkonsistent wird.

Hinweis

Nicht mehr benötigter Speicherplatz sollte immer freigegeben werden, um Speicherplatz-Engpässe zu vermeiden.

Der mit **free** freigegebene Speicherplatz wird nicht wirklich dem Kern als freier Speicherplatz zurückgegeben, sondern er wird intern im sogenannten *malloc pool* gehalten, um ihn bei späteren Speicheranforderungen des Prozesses wieder verwenden zu können.

Der Aufruf `free(NULL)` hat keinerlei Auswirkung.

9.4.4 **alloca** – Dynamisches Anfordern von Speicherplatz im Stack

Um Speicherplatz auf dem Stack anzufordern, steht die Funktion **alloca** zur Verfügung.

```
#include <stdlib.h>
```

```
void *alloca(size_t groesse);
```

gibt zurück: Adresse des allokierten Speicherbereichs (bei Erfolg); NULL bei Fehler

Die Funktion **alloca** ist weitgehend identisch mit der Funktion **malloc**, nur daß sie Speicherplatz nicht vom *Heap*, sondern vom *Stack* anfordert. Sie allokiert dabei Speicherplatz vom *stack frame* (Stackbereich) der momentanen Funktion. Der Vorteil von **alloca** ist, daß der so allokierte Speicherplatz nicht explizit freizugeben ist, sondern automatisch beim Verlassen der betreffenden Funktion freigegeben wird.

alloca vergrößert den *stack frame* (Stackbereich) der aktuellen Funktion.

Hinweis

Diese Funktion ist nicht überall verfügbar, denn bei manchen Systemen ist es nicht möglich, den *stack frame* zu vergrößern, nachdem eine Funktion aufgerufen wurde.

9.5 Ressourcenlimits eines Unix-Prozesses

Jedem Unix-Prozeß wird von den verfügbaren Ressourcen eines Systems oft nur eine begrenzte Teilmenge zugeteilt.

9.5.1 **getrlimit** und **setrlimit** – Erfragen und Setzen der Ressourcenlimits

Einige der für einen Prozeß geltenden Ressourcenlimits können mit den Funktionen **getrlimit** und **setrlimit** erfragt und verändert werden:

```
#include <sys/time.h>
```

```
#include <sys/resource.h>
```

```
int getrlimit(int ressource, struct rlimit *rlimit_zgr);
```

```
int setrlimit(int ressource, const struct rlimit *rlimit_zgr);
```

beide geben zurück: 0 (bei Erfolg); Wert verschieden von 0 bei Fehler

getrlimit erfragt und **setrlimit** setzt ein bestimmtes Limit. Bei beiden Funktionen wählt das erste Argument die entsprechende *ressource* (vordefinierte **int**-Konstante) aus, und das zweite Argument muß ein Zeiger auf die Struktur `rlimit` sein:

```
struct rlimit {
    rlim_t rlim_cur; /* Soft-Limit: aktuelles Limit */
    rlim_t rlim_max; /* Hard-Limit: maximaler Wert für rlim_cur */
};
```

Ist für die Komponenten `rlim_cur` oder `rlim_max` die vordefinierte Konstante `RLIM_INFINITY` angegeben, so bedeutet dies *unbegrenzt*, also keinerlei Limit.

Grundsätzlich gelten dabei die folgenden Regeln:

1. Ein Soft-Limit kann von jedem Prozeß verändert werden, wobei der neue Wert aber immer nur kleiner oder gleich dem Hard-Limit sein kann.
2. Jeder Prozeß kann sein Hard-Limit auf einen Wert heruntersetzen, der größer oder gleich dem Soft-Limit ist. Ein erneutes Hochsetzen des Hard-Limits ist jedoch für diesen Prozeß nicht mehr möglich, denn normale Benutzerprozesse können grundsätzlich das Hard-Limit immer nur erniedrigen, und niemals erhöhen.
3. Nur der Superuser kann das Hard-Limit erhöhen.

Für den Parameter *ressource* kann eine der folgenden vordefinierten Konstanten angegeben werden:

`RLIMIT_CORE`

(SVR4 und BSD) Maximale Größe einer *core*-Datei (in Byte). Ein Limit von 0 legt fest, daß keine *core*-Datei angelegt werden kann.

`RLIMIT_CPU`

(SVR4 und BSD) Limit für die CPU-Zeit (in Sekunden). Wenn das Soft-Limit überschritten wird, wird dem Prozeß das Signal `SIGXCPU` gesendet.

`RLIMIT_DATA`

(SVR4 und BSD) Maximale Größe des gesamten Datensegments (in Byte). Gesamtes Datensegment umfaßt dabei *data segment*, *bss segment* und *heap* (siehe auch Abbildung 9.3).

`RLIMIT_FSIZE`

(SVR4 und BSD) Maximale Größe einer Datei, die beschrieben werden kann (in Byte). Wenn das Soft-Limit überschritten wird, wird dem Prozeß das Signal `SIGXFSZ` gesendet.

`RLIMIT_MEMLOCK`

(BSD und Linux) Maximale Speichergröße, die mit **unlock** gesperrt werden kann. Der Aufruf **mlock** erlaubt es Prozessen, einen bestimmten Speicherbereich vom Auslagern auszuschließen.

RLIMIT_NOFILE

(nur in SVR4) Maximale Anzahl von gleichzeitig geöffneten Dateien. Eine Änderung dieses Limits wirkt sich auch auf den Rückgabewert der Funktion **sysconf** aus, wenn diese mit dem Argument `_SC_OPEN_MAX` aufgerufen wird.

RLIMIT_NPROC

(nur in BSD) Maximale Anzahl von Kindprozessen je realer UID. Eine Änderung dieses Limits wirkt sich auch auf den Rückgabewert der Funktion **sysconf** aus, wenn diese mit dem Argument `_SC_CHILD_MAX` aufgerufen wird.

RLIMIT_OFILE

(nur in BSD) Maximale Anzahl von gleichzeitig geöffneten Dateien. Eine Änderung dieses Limits wirkt sich auch auf den Rückgabewert der Funktion **sysconf** aus, wenn diese mit dem Argument `_SC_OPEN_MAX` aufgerufen wird.

RLIMIT_RSS

(nur in BSD) Maximale *resident set size* (RSS) (in Byte). Falls Speicherengpässe entstehen, entzieht der Kern den Prozessen, die ihre RSS überschreiten, diesen den zuviel angeforderten Speicher.

RLIMIT_STACK

(SVR4 und BSD) Maximale Größe des Stacks (in Byte); siehe auch Abbildung 9.3.

RLIMIT_VMEM

(nur in SVR4) Maximale Größe des *Memory Mapped*-Adreßraums (in Byte). Dies wirkt sich auf die Funktion **mmap** aus, die in Kapitel 15.3 beschrieben wird.

Die in einem Prozeß gesetzten Ressourcenlimits werden auch an seine Kindprozesse vererbt.

Hinweis

Die beiden Funktionen **setrlimit** und **getrlimit** werden in SVR4 und BSD-Unix angeboten, sind aber nicht Bestandteil von POSIX.1.

Die Ressourcenlimits eines Prozesses können auch mit dem in der Bourne- und Korn-Shell vorhandenen builtin-Kommando **ulimit** erfragt oder gesetzt werden. Das **ulimit** neuerer Korn-Shell-Versionen bietet sogar die Optionen **-S** und **-H** zur Unterscheidung von Soft- und Hard-Limits an. Diese beiden Optionen sind oft nicht dokumentiert.

Die Ressourcenlimits eines Prozesses können in der C-Shell mit dem builtin-Kommando **limit** erfragt oder gesetzt werden.

Die allgemein für Prozesse geltenden Ressourcenlimits werden normalerweise vom Prozeß 0 festgelegt, wenn das System initialisiert wird. Alle Folgeprozesse erben dann diese Limits. In SVR4 sind z.B. die voreingestellten Limits in der Datei `/etc/conf/cf.d/mtune` hinterlegt, während in BSD-Unix die Limitvorgaben über mehrere Dateien verstreut sind.

Beispiel*Ausgeben der aktuellen Ressourcenlimits*

```

#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "eighdr.h"

#define ausgabe(name) druck_limit(#name, name)

static void druck_limit(char *name, int resource);

int
main(void)
{
    printf("%15s %-14s%s\n", "", "Soft-Limit", "Hard-Limit");
    printf("-----\n");

    ausgabe(RLIMIT_CORE);
    ausgabe(RLIMIT_CPU);
    ausgabe(RLIMIT_DATA);
    ausgabe(RLIMIT_FSIZE);

    #ifdef RLIMIT_MEMLOCK
    ausgabe(RLIMIT_MEMLOCK);
    #endif
    #ifdef RLIMIT_NOFILE
    ausgabe(RLIMIT_NOFILE);
    #endif
    #ifdef RLIMIT_OFILE
    ausgabe(RLIMIT_OFILE);
    #endif
    #ifdef RLIMIT_NPROC
    ausgabe(RLIMIT_NPROC);
    #endif
    #ifdef RLIMIT_RSS
    ausgabe(RLIMIT_RSS);
    #endif
    #ifdef RLIMIT_STACK
    ausgabe(RLIMIT_STACK);
    #endif
    #ifdef RLIMIT_VMEM
    ausgabe(RLIMIT_VMEM);
    #endif
    printf("-----\n");
    exit(0);
}

static void
druck_limit(char *name, int resource)
{
    struct rlimit limit;

```

```

if (getrlimit(resource, &limit) < 0)
    fehler_meld(FATAL_SYS, "getrlimit-Fehler bei %s", name);
printf("%-15s  ", name);
if (limit.rlim_cur == RLIM_INFINITY)
    printf("(unbegrenzt)  ");
else
    printf("%12ld  ", limit.rlim_cur);
if (limit.rlim_max == RLIM_INFINITY)
    printf("(unbegrenzt)\n");
else
    printf("%12ld\n", limit.rlim_max);
}

```

Programm 9.7 (**limits.c**): Ausgabe der aktuellen Ressourcenlimits

Nachdem man dieses Programm 9.7 (`limits.c`) kompiliert und gelinkt hat

```
cc -o limits limits.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ limits

```

	Soft-Limit	Hard-Limit
RLIMIT_CORE	(unbegrenzt)	(unbegrenzt)
RLIMIT_CPU	(unbegrenzt)	(unbegrenzt)
RLIMIT_DATA	536870912	536870912
RLIMIT_FSIZE	(unbegrenzt)	(unbegrenzt)
RLIMIT_NOFILE	64	1024
RLIMIT_STACK	8683520	133464064
RLIMIT_VMEM	(unbegrenzt)	(unbegrenzt)

```

$

```

Diese Ausgabe wurde auf SOLARIS 2.0 erhalten.

9.6 Ressourcenbenutzung eines Unix-Prozesses

Der Systemkern führt Buch darüber, wie viele Ressourcen ein Prozeß benutzt. Mit der Funktion **getrusage** kann ein Prozeß seine eigene Benutzung von Ressourcen, die Benutzung von Ressourcen durch alle seine Kindprozesse oder die Summe aus beiden erfragen.

```

#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrusage(int wessen, struct rusage *usage);

```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Der erste Parameter `wessen` wählt eine der drei möglichen Ressourcenermittlungen aus:

RUSAGE_SELF	Benutzung der Ressourcen des Prozesses selbst
RUSAGE_CHILDREN	Benutzung der Ressourcen aller Kindprozesse
RUSAGE_BOTH	Benutzung der Ressourcen des Prozesses und aller seiner Kindprozesse

Der zweite Parameter `usage` ist die Adresse einer Variablen vom Datentyp `struct rusage`. In die Komponenten dieser Strukturvariablen schreibt *getrusage* die entsprechenden Informationen. Die Struktur `rusage` ist in `<sys/resource.h>` bzw. `<linux/resource.h>` wie folgt definiert:

```
struct rusage {
    struct timeval ru_utime; /* user time used;
                             CPU-Zeit, die der Prozeß
                             im Benutzermodus aktiv war */
    struct timeval ru_stime; /* system time used;
                             CPU-Zeit, die der Prozeß
                             im Systemmodus aktiv war */
    long ru_maxrss; /* maximum resident set size */
    long ru_ixrss; /* integral shared memory size */
    long ru_idrss; /* integral unshared data size */
    long ru_isrss; /* integral unshared stack size */
    long ru_minflt; /* page reclaims (minor faults);
                    Prozeß mußte in Systemmodus
                    wechseln, wobei jedoch kein
                    Festplattenzugriff notwendig
                    ist (z.B. wenn Stack zu
                    vergroessern ist) */
    long ru_majflt; /* page faults (major faults);
                    Prozeß mußte in Systemmodus
                    wechseln, wobei jedoch ein
                    Festplattenzugriff notwendig
                    ist (z.B. wenn eine Page noch
                    nicht im Hauptspeicher ist
                    oder auf die Swap-Partition
                    ausgelagert wurde) */
    long ru_nswap; /* swaps; Anzahl der Pages, die
                   aufgrund von Page Faults
                   eingelagert werden mußten */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* messages sent */
    long ru_msgrcv; /* messages received */
    long ru_nsignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary " " */
};
```


Die Struktur `rusage` stammt von BSD. Da unter Linux die komplette Implementierung von **getrusage** noch nicht abgeschlossen ist, werden dort noch nicht alle Komponenten dieser Struktur durch einen **getrusage**-Aufruf gefüllt. Die in jedem Fall schon verfügbaren Informationen sind in der obigen Struktur ausführlicher dokumentiert.

9.7 Die Speicherverwaltung unter Linux



Hier wird ein Einblick in die Speicherverwaltung und das Abbilden von Dateien in den Speicher (*Memory Mapping*) unter Linux gegeben. Dieses Kapitel ist nur für Leser von Interesse, die mehr über die interne Speicherverwaltung eines existierenden Systems wissen möchten. Andere Leser, die nicht an solche Interna eines Systemkerns, sondern nur an der reinen Systemprogrammierung interessiert sind, was wohl für die meisten Unix-Programmierer zutrifft, können dieses Kapitel ohne Bedenken überblättern.

9.7.1 Allgemeine Begriffe und Konzepte

Pages

Der physikalisch vorhandene Speicher wird in sogenannten *Pages* – im Deutschen oft auch als Speicherseiten oder früher auch als Kacheln bezeichnet – aufgeteilt. Die Größe einer Page ist durch das in der Datei `<asm/page.h>` definierte Makro `PAGE_SIZE` festgelegt. Bei Intel-Prozessoren ist diese Größe z.B. auf 4 KByte (4096 Byte) und beim Alpha-Prozessor auf 8 KByte (8192 Byte) festgelegt. Hieran ist zu erkennen, daß Linux nicht für einen speziellen Prozessor konzipiert wurde, sondern mit einem sogenannten *architekturunabhängigen Speichermodell* arbeitet.

Virtueller Adreßraum

Ein Prozeß arbeitet nicht direkt im physikalischen Speicher, sondern in einem sogenannten virtuellen Adreßraum, wobei sich eine virtuelle Adresse aus zwei Komponenten zusammensetzt: Einem *Segmentsektor*, der die Anfangsadresse des entsprechenden Segments enthält und einem *Offset*, das die Adresse des jeweiligen Objekts relativ zum Segmentanfang angibt.

Der virtuelle Adreßraum besteht aus zwei Segmenten, dem Kernsegment (*kernel segment* oder *system segment*) und dem Benutzersegment (*user segment*). Der Code und die Daten des Kerns werden im Kernsegment, während der Code und die Daten eines Prozesses im Benutzersegment untergebracht werden. Beim Abarbeiten des Codes ist der Segmentsektor bereits gesetzt, und die Zeiger, mit denen im Programm gearbeitet wird, enthalten nur die Offsets der jeweiligen Objekte.

Manchmal muß aber das Kernsegment auf Daten des Benutzersegments zugreifen, z.B. wenn im Benutzercode eine Systemfunktion (aus dem Kernsegment) mit Argumenten aufgerufen wird. In diesem Fall muß das Kernsegment auf Daten (die übergebenen Argumente) aus dem Benutzersegment zugreifen.

Während in der Version 2.0 des Linux-Kerns noch die Datei `<asm/segment.h>` die entsprechenden Funktionen für die Zugriffe auf Daten des Benutzersegments enthält, befinden sich diese Funktionen in der Version 2.1 in der Headerdatei `<asm/uaccess.h>`. Eine weitere wichtige Neuheit gegenüber Version 2.0 ist, daß beim Zugriff auf das Benutzersegment zur Verifizierung nicht mehr die Funktion ***verify_area*** verwendet wird, sondern diese Verifizierung nun weitgehend von der CPU durchgeführt wird.

Die neuen Funktionen für das Lesen und Schreiben von Daten im Benutzersegment sind:

```
int access_ok(int type, unsigned long addr,
              unsigned long size);
```

Diese Funktion liefert den Wert 1, wenn der aktuelle Prozeß auf den Speicher an der Adresse `addr` zugreifen darf, und ansonsten den Wert 0. Diese Funktion weist eine wesentlich bessere Performance auf als die Funktion ***verify_area***, deren Aufgabe sie nun weitgehend übernimmt. Vor einem Zugriff auf das Benutzersegment sollte mit dieser Funktion zunächst geprüft werden, ob der gewünschte Zugriff überhaupt erlaubt ist.

```
int get_user(lvalue, addr);
```

Das im Kern 2.1 verwendete Makro ***get_user*** unterscheidet sich von dem gleichnamigen Makro im Kern 2.0. Der Rückgabewert ist 0 im Erfolgsfall und ansonsten eine negative Fehlernummer (`-EFAULT`). ***get_user*** liest die Daten an der Adresse `addr` und schreibt sie nach `lvalue`. Wie im Kern 2.0 hängt die Größe der zu lesenden Daten vom Datentyp des Zeigers `addr` ab. Die Funktion ***get_user*** ruft intern ***access_ok***, so daß ein expliziter Aufruf von ***access_ok*** vor dem Aufruf von ***get_user*** nicht notwendig ist.

```
int __get_user(lvalue, addr);
```

Die Funktion ***__get_user*** leistet das gleiche wie die zuvor vorgestellte Funktion ***get_user***, mit der Ausnahme, daß sie nicht ***access_ok*** aufruft. Diese Funktion wird z.B. dann in Kernfunktionen verwendet, wenn diese auf Adressen im Benutzersegment zugreifen, die bereits zuvor von derselben Kernfunktion überprüft wurden.

```
int get_user_ret(lvalue, addr, retval);
```

Dieses Makro ***get_user_ret*** ruft seinerseits nur die Funktion ***get_user*** und liefert `retval`, wenn diese Funktion nicht erfolgreich war.

```
int put_user(ausdruck, addr);
int __put_user(ausdruck, addr);
int put_user_ret(ausdruck, addr, retval);
```

Diese drei Funktionen verhalten sich genau wie die drei zuvor vorgestellten **get_**-Funktionen, mit dem Unterschied, daß sie in das Benutzersegment schreiben und nicht aus ihm lesen. Sie schreiben den Wert, der aus der Auswertung von `ausdruck` resultiert, an die Adresse `addr`.

Zusätzlich sind im Kern 2.0 noch die folgenden Funktionen zum Kopieren von Datenbytes definiert :

```
void memcpy_fromfs(void *to, const void *from, unsigned long n);
void memcpy_tofs(void *to, const void *from, unsigned long n);
```

Die Namen dieser Funktionen gehen zurück auf die ersten Linux-Versionen, als die einzig unterstützte Hardware der i386-Intel-Prozessor war, bei dem das Benutzersegment über das FS-Register adressiert wurde.

Ab der Kernversion 2.1 werden diese beiden Funktionen durch die folgenden Funktionen ersetzt.

```
unsigned long copy_from_user(unsigned long to,
                             unsigned long from,
                             unsigned long n);
```

Diese Funktion kopiert Datenbytes aus dem Benutzersegment in das Kernsegment und ersetzt somit die alte Funktion **memcpy_fromfs**. Intern ruft diese Funktion **access_ok** auf. Der Rückgabewert von **copy_from_user** ist immer die Anzahl der Bytes, die nicht übertragen werden konnten, was bedeutet, daß ein Rückgabewert größer als 0 auf einen Fehler hinweist.

```
unsigned long __copy_from_user(unsigned long to,
                               unsigned long from,
                               unsigned long n);
```

Diese Funktion entspricht weitgehend der zuvor vorgestellten Funktion **copy_from_user**, nur daß sie anders als diese intern nicht **access_ok** aufruft.

```
unsigned long copy_from_user_ret(to, from, n, retval);
```

Dieses Makro **copy_from_user_ret** ruft seinerseits nur die Funktion **copy_from_user** und liefert `retval`, wenn diese Funktion nicht erfolgreich war.

```
unsigned long copy_to_user(unsigned long to,
                           unsigned long from,
                           unsigned long n);
```

```
unsigned long __copy_to_user(unsigned long to,
                             unsigned long from,
                             unsigned long n);
```

```
unsigned long copy_to_user_ret(unsigned long to,
                               unsigned long from,
                               unsigned long n);
```

Diese drei Funktionen verhalten sich genau wie die drei zuvor vorgestellten **copy_from_**-Funktionen, mit dem Unterschied, daß sie in das Benutzersegment schreiben und nicht aus ihm lesen.

Weitere Funktionen für den Zugriff auf das Benutzersegment in der Kernversion 2.1 sind: **clear_user**, **strncpy_from_user** und **strlen_user**. Interessierte Leser können diese in `<asm/uaccess.h>` nachschlagen.

Die Segmentselektoren für die Kern- und Benutzerdaten sind über die beiden Makros **KERNEL_DS** und **USER_DS** definiert. Die Definition dieser beiden Makros befindet sich in der Kernversion 2.0 in `<asm/segment.h>` und in der Kernversion 2.1 in `<asm/uaccess.h>`.

Im Kernsegment kann der aktuelle Segmentselektor des Datensegments mit der Funktion **get_ds** erfragt werden. Zum Lesen und Setzen des für das Benutzersegment im Kern verwendeten Selektorregisters stehen die beiden Funktionen **get_fs** und **set_fs** zur Verfügung. Sie dienen zum Aufruf von Systemfunktionen innerhalb des Kerns, da der Code von Systemfunktionen davon ausgeht, daß alle der Funktion übergebenen Argumente Adressen im Benutzersegment sind. Wird aber das Segmentselektorregister für das Benutzersegment (FS bei x86-Prozessoren) so umgesetzt, daß es das Kernsegment adressiert, wird bei Zugriffen über Funktionen, die eigentlich auf das Benutzersegment eingestellt sind (wie z.B. **copy_from_user**), nicht auf das Benutzer-, sondern auf das Kernsegment zugegriffen. Die drei Funktionen **get_ds**, **get_fs** und **set_fs** sind in `<asm/segment.h>` (in Kernversion 2.0) bzw. in `<asm/uaccess.h>` (in Kernversion 2.1) definiert.

Linearer Adreßraum

Bei Intel-Prozessoren wird die virtuelle Adresse durch das MMU (*Memory Management Unit*) in eine lineare Adresse umgesetzt. Bei diesen Prozessoren ist der lineare Adreßraum auf 4 GByte beschränkt, da für lineare Adressen 4 Byte verwendet werden. Da alle Segmente im linearen Adreßraum untergebracht werden müssen, muß dieser zwischen dem Benutzer- und dem Kernsegment aufgeteilt werden. Das in `<asm/processor.h>` definierte Makro **TASK_SIZE** legt die Größe des Benutzersegments auf 3 GByte fest, was bedeutet, daß 1 GByte für das Kernsegment vorgesehen ist.

Da der Alpha-Prozessor keine Segmentierung kennt, sondern mit linearen Adressen arbeitet, entspricht bei diesem Prozessortyp das Offset direkt der linearen Adresse. Hierbei ist lediglich sicherzustellen, daß sich die Adressen (Offsets) des Benutzersegments nicht mit denen des Kernsegments überschneiden, was bei einem verfügbaren linearen Adreßraum von 2^{64} Byte leicht möglich ist. Die für den Alpha-Prozessor angebotenen Funktionen zum Zugriff auf das Benutzersegment arbeiten intern direkt mit den Offsetadressen und die bereitgestellten Funktionen zum Lesen bzw. Setzen des Segmentselektorregisters lesen bzw. setzen lediglich ein Flag im Task-Statussegment. Dieses Flag legt fest, ob es sich bei den Argumenten von Systemaufrufen um Daten aus dem Benutzer- oder Kernsegment handelt.

Die Adresse des linearen Adreßraums wird unter Linux in vier Teile zerlegt (siehe dazu auch Abbildung 9.4).

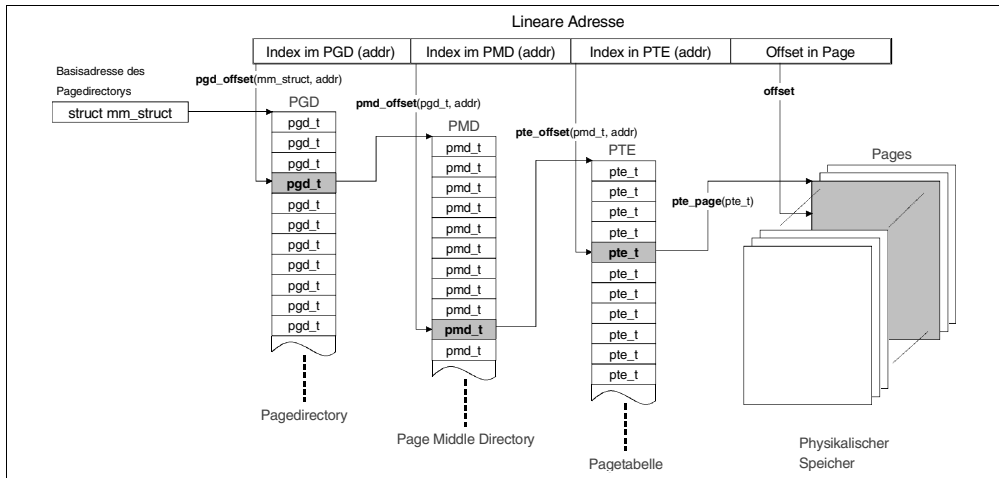


Abbildung 9.4: Die Abbildung von linearen Adressen auf physikalische Adressen

Jeder Prozeß hat ein Pagedirectory, das über eine `mm_struct`-Strukturvariable adressiert wird. Der erste Teil einer linearen Adresse ist ein Index für das Pagedirectory (PGD). Der so ausgewählte Eintrag im Pagedirectory zeigt auf ein sogenanntes Page Middle Directory (PMD). Der zweite Teil der linearen Adresse ist dann ein Index in diesem Page Middle Directory. Der indizierte Eintrag im Page Middle Directory zeigt auf eine Pagetabelle. Der dritte Teil der linearen Adresse ist dann ein Index in der ausgewählten Pagetabelle (PTE). Die in dem Eintrag stehende Adresse adressiert dann die entsprechende Page. Auf das entsprechende Objekt, das über die vierteilige lineare Adresse adressiert wird, kann dann durch Addition des Offsets, das sich im vierten Teil der linearen Adresse befindet, zugegriffen werden.

Da Intel-Prozessoren nur eine zweistufige Übersetzung einer linearen Adresse unterstützen, legt Linux die Größe des Page Middle Directory bei diesen Prozessoren auf 1 fest. Da aber Prozessoren wie der Alpha-Prozessor lineare 64 Bit-Adressen unterstützen, mußte man mit einem dreistufigem Speichermodell arbeiten, damit die Pagedirectories und Pagetabellen nicht zu groß werden. Um das architekturunabhängige Speichermodell auf dem Alpha-Prozessor zu realisieren, wurde festgelegt, für die einzelnen Pagedirectories (Pagedirectory und Page Middle Directory) und für die Pagetabelle jeweils eine Page (8 KByte beim Alpha-Prozessor) zu verwenden, was eine maximale Anzahl von 1024 Einträgen in den jeweiligen Tabellen erlaubt. Dies wiederum bedeutet, daß der virtuelle Adreßraum auf einem Alpha-Prozessor bis zu 8184 GByte (fast 8 Terabyte) groß sein kann:

<i>Tabelle</i>	<i>maximale Einträge</i>	
Pagedirectory	1023 x	8 GByte
Page Middle Directory	1024 x	
Pagetabelle	1024 x	
Page	8 KByte	
	8184 GByte (1023*8 GByte)	

Für das Pagedirectory sind nur 1023 und nicht 1024 Einträge möglich, da die Basisadresse des Pagedirectorys sich ebenfalls in dieser Tabelle befindet.

Von diesen fast 8 Terabyte werden 2 Terabyte für das Benutzersegment zur Verfügung gestellt.

Die in Abbildung 9.4 eingeführten Datentypen sind in <asm/page.h> definiert:

```
typedef unsigned long pte_t;
typedef unsigned long pmd_t;
typedef unsigned long pgd_t;
```

bzw. sie sind auch wie folgt definiert:

```
typedef struct { unsigned long pte; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
```

Nachfolgend werden die wichtigsten Datentypen, Makros und Funktionen (aus <asm/page.h> bzw. aus <asm/pgtable.h>) vorgestellt, mit denen auf die Pagedirectories und Pagetabellen zugegriffen werden kann bzw. mit denen diese modifiziert werden können.

Das Pagedirectory

Ein Eintrag im Pagedirectory hat wie oben erwähnt den Datentyp pgd_t. Der Zugriff auf den Wert eines Eintrags im Pagedirectory erfolgt mit dem Makro **pgd_val**, das in <asm/page.h> wie folgt definiert ist:

```
#define pgd_val(x) (x)
```

bzw.

```
#define pgd_val(x) ((x).pgd)
```

Die Anzahl der Einträge im Pagedirectory ist in <asm/pgtable.h> z.B. wie folgt definiert:

```
#define PTRS_PER_PGD 1024
```

Nachfolgend werden die wichtigsten Funktionen/Makros zum Pagedirectory, die in `<asm/pgtable.h>` definiert sind, kurz erläutert:

```
pgd_t * pgd_alloc(void)
```

allokiert eine Page für das Pagedirectory und füllt diese mit Nullen.

```
int pgd_bad(pgd_t pgd)
```

dient zum Testen, ob der Eintrag im Pagedirectory ungültig ist.

```
void pgd_clear(pgd_t * pgdp)
```

löscht den Eintrag im Pagedirectory.

```
void pgd_free(pgd_t * pgdp)
```

gibt die vom Pagedirectory belegte Page wieder frei.

```
int pgd_none(pgd_t pgd)
```

prüft, ob der entsprechende Eintrag im Pagedirectory noch nicht initialisiert ist.

```
pgd_t * pgd_offset(struct mm_struct * mm, unsigned long address)
```

gibt zu einer linearen Adresse den Zeiger auf den zugehörigen Eintrag im Pagedirectory zurück.

```
int pgd_present(pgd_t pgd)
```

prüft, ob der Eintrag im Pagedirectory auf ein Page Middle Directory zeigt.

```
SET_PAGE_DIR(tsk, pgdir)
```

setzt eine neue Basisadresse für das Pagedirectory einer Task.

Das Page Middle Directory

Ein Eintrag im Page Middle Directory hat wie oben erwähnt den Datentyp `pmd_t`. Der Zugriff auf den Wert eines Eintrags im Page Middle Directory erfolgt mit dem Makro **`pmd_val`**, das in `<asm/page.h>` wie folgt definiert ist:

```
#define pmd_val(x) (x)
```

bzw.

```
#define pmd_val(x) ((x).pmd)
```

Die Anzahl der Einträge im Page Middle Directory ist in `<asm/pgtable.h>` z.B. wie folgt definiert:

```
#define PTRS_PER_PMD 1 /* Für Intel-Prozessoren */
```

```
#define PTRS_PER_PMD 1024 /* Für Alpha-Prozessor */
```

Nachfolgend werden die wichtigsten Funktionen/Makros zum Page Middle Directory, die `<asm/pgtable.h>` definiert sind, kurz erläutert:

```
pmd_t * pmd_alloc(pgd_t * pgd, unsigned long address)
```

allokiert ein Page Middle Directory für die Speicherverwaltung im Benutzersegment.

`pmd_t * pmd_alloc_kernel(pgd_t * pgd, unsigned long address)`
 allokiert ein Page Middle Directory für die Speicherverwaltung im Kernsegment, wobei dort alle Einträge auf ungültig gesetzt werden.

`int pmd_bad(pmd_t pmd)`
 dient zum Testen, ob der Eintrag im Page Middle Directory ungültig ist.

`void pmd_clear(pmd_t * pmdp)`
 löscht den Eintrag im Page Middle Directory.

`void pmd_free(pmd_t * pmd)`
 gibt den für ein Page Middle Directory belegten Speicher im Benutzersegment wieder frei.

`void pmd_free_kernel(pmd_t * pmd)`
 gibt den für ein Page Middle Directory belegten Speicher im Kernsegment wieder frei.

`int pmd_none(pmd_t pmd)`
 prüft, ob der Eintrag im Page Middle Directory noch nicht gesetzt ist.

`pmd_t * pmd_offset(pgd_t * dir, unsigned long address)`
 gibt zu einer linearen Adresse (`address`) den Zeiger auf den zugehörigen Eintrag im Page Middle Directory zurück. Die Adresse des entsprechenden Page Middle Directory wird dabei über das Argument `dir` übergeben.

`unsigned long pmd_page(pmd_t pmd)`
 liefert die Basisadresse der Pagetabelle, auf die der entsprechende Eintrag im Page Middle Directory zeigt.

`int pmd_present(pmd_t pmd)`
 prüft, ob der Eintrag im Page Middle Directory auf eine Pagetabelle zeigt.

Die Pagetabelle

Ein Eintrag in der Pagetabelle hat wie oben erwähnt den Datentyp `pte_t`. Der Zugriff auf den Wert eines Eintrags in der Pagetabelle erfolgt mit dem Makro **`pte_val`**, das in `<asm/page.h>` wie folgt definiert ist:

```
#define pte_val(x) (x)
```

bzw.

```
#define pte_val(x) ((x).pte)
```

Die Anzahl der Einträge in der Pagetabelle ist in `<asm/pgtable.h>` z.B. wie folgt definiert:

```
#define PTRS_PER_PTE 1024
```

Ein Eintrag in der Pagetabelle enthält neben der Adresse einer Page im physikalischen Speicher noch einige Flags, die den Zustand und die gültigen Zugriffsrechte für diese Page beschreiben. Die wichtigsten dazugehörigen Konstanten sind in `<asm/page.h>`


```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT      12
#define PAGE_SIZE      (1UL << PAGE_SHIFT)
#define PAGE_MASK      (~(PAGE_SIZE-1))
```

bzw. in `<asm/pgtable.h>` definiert:

```
#define _PAGE_PRESENT    0x001    /* Page im virtuellen Speicher */
#define _PAGE_RW         0x002    /* Page darf beschrieben werden */
#define _PAGE_USER       0x004    /* Page darf gelesen werden */
#define _PAGE_ACCESSED   0x020    /* Auf Page wurde zugegriffen */
#define _PAGE_DIRTY      0x040    /* Page wurde modifiziert */
```

Zusätzlich sind die folgenden Attributkombinationen in `<asm/pgtable.h>` als Makros definiert:

```
#define _PAGE_CHG_MASK  (PAGE_MASK | _PAGE_ACCESSED | _PAGE_DIRTY)

#define PAGE_NONE      __pgprot(_PAGE_PRESENT | _PAGE_ACCESSED)
/* durch Pagetableneintrag wird keine physikalische
   Page referenziert */

#define PAGE_SHARED     __pgprot(_PAGE_PRESENT | _PAGE_RW | \
                                _PAGE_USER | _PAGE_ACCESSED)
/* auf dieser Page sind alle Zugriffe erlaubt */

#define PAGE_READONLY   __pgprot(_PAGE_PRESENT | _PAGE_USER | \
                                _PAGE_ACCESSED)
/* Auf diese Page ist nur lesender oder ausführender
   Zugriff erlaubt. Bei einem Schreiben auf diese Page
   wird eine Exception generiert, die es ermöglicht, den
   Fehler zu behandeln. Als Reaktion auf diese Exception
   wird dann die Page kopiert, der Pagetableneintrag
   auf die physikalische Adresse der neuen Page und
   seine Attribute auf PAGE_SHARED gesetzt,
   was genau dem COW-Verfahren entspricht */

#define PAGE_COPY       __pgprot(_PAGE_PRESENT | _PAGE_USER | \
                                _PAGE_ACCESSED)
/* aus historischen Gründen noch vorhanden;
   entspricht dem Makro PAGE_READONLY */

#define PAGE_KERNEL     __pgprot(_PAGE_PRESENT | _PAGE_RW | \
                                _PAGE_DIRTY | _PAGE_ACCESSED)
/* Der Zugriff auf diese Page ist nur dem Kernsegment
   erlaubt. */
```

Zusätzlich sind in `<asm/pgtable.h>` noch die folgenden Makros definiert, die die Definition beliebiger Kombinationen von Attributen ermöglichen:

```
/* The i386 can't do page protection for execute, and considers
   that the same are read.
   Also, write permissions imply read permissions.
   This is the closest we can get.. */
```

```

#define __P000 PAGE_NONE
#define __P001 PAGE_READONLY
#define __P010 PAGE_COPY
#define __P011 PAGE_COPY
#define __P100 PAGE_READONLY
#define __P101 PAGE_READONLY
#define __P110 PAGE_COPY
#define __P111 PAGE_COPY

#define __S000 PAGE_NONE
#define __S001 PAGE_READONLY
#define __S010 PAGE_SHARED
#define __S011 PAGE_SHARED
#define __S100 PAGE_READONLY
#define __S101 PAGE_READONLY
#define __S110 PAGE_SHARED
#define __S111 PAGE_SHARED

```

Die Makros ***pgprot_val***, ***__pgprot*** und die zugehörige Struktur sind in `<asm/page.h>` wie folgt definiert:

```

typedef struct { unsigned long pgprot; } pgprot_t;
#define pgprot_val(x) ((x).pgprot)
#define __pgprot(x) ((pgprot_t) { (x) } )

```

Zum Lesen und Modifizieren der Pagetableneinträge und ihrer Attribute sind die folgenden Funktionen bzw. Makros (für Intel-Prozessoren) in `<asm/pgtable.h>` definiert:

```

pte_t mk_pte(unsigned long page, pgprot_t pgprot) {
    pte_t pte;
    pte_val(pte) = page | pgprot_val(pgprot);
    return pte;
}

```

erzeugt einen Pagetableneintrag, der aus den übergebenen Attributen (***pgprot***) und der Speicheradresse der Page (***page***) ermittelt wird. Diesen so gebildeten Pagetableneintrag liefert ***mk_pte*** als Rückgabewert.

```

pte_t * pte_alloc(pmd_t * pmd, unsigned long address)
    ....
}

```

allokiert eine neue Pagetabelle.

```

pte_t * pte_alloc_kernel(pmd_t * pmd, unsigned long address) {
    ....
}

```

allokiert eine neue Pagetabelle für Speicher im Kernsegment.

```

#define pte_clear(xp) do { pte_val(*(xp)) = 0; } while (0)

```

löscht den entsprechenden Pagetableneintrag.

```
int pte_dirty(pte_t pte) { return pte_val(pte) & _PAGE_DIRTY; }
```

prüft, ob für den Pagetableneintrag das Attribut *Dirty* gesetzt ist.

```
int pte_exec(pte_t pte) { return pte_val(pte) & _PAGE_USER; }
```

prüft, ob für den Pagetableneintrag das Attribut *Ausführerlaubnis* gesetzt ist, ob also die Ausführung von Code in der entsprechenden Page erlaubt ist.

```
pte_t pte_exprotect(pte_t pte) {
    pte_val(pte) &= ~_PAGE_USER; return pte;
}
```

löscht das Attribut *Ausführerlaubnis* für die entsprechende Page.

```
void pte_free(pte_t * pte) {
    free_page((unsigned long) pte);
}
```

gibt die entsprechende Page frei.

```
void pte_free_kernel(pte_t * pte) {
    free_page((unsigned long) pte);
}
```

gibt die entsprechende Page frei, die vom Kernsegment verwaltet wird.

```
pte_t pte_mkclean(pte_t pte) {
    pte_val(pte) &= ~_PAGE_DIRTY; return pte;
}
```

löscht das Attribut *Dirty* für den entsprechenden Pagetableneintrag.

```
pte_t pte_mkdirty(pte_t pte) {
    pte_val(pte) |= _PAGE_DIRTY; return pte;
}
```

setzt das Attribut *Dirty* für den entsprechenden Pagetableneintrag.

```
pte_t pte_mkexec(pte_t pte) {
    pte_val(pte) |= _PAGE_USER; return pte;
}
```

setzt das Attribut *Ausführerlaubnis* für den entsprechenden Pagetableneintrag.

```
pte_t pte_mkold(pte_t pte) {
    pte_val(pte) &= ~_PAGE_ACCESSED; return pte;
}
```

setzt das Attribut *old* für den entsprechenden Pagetableneintrag, was bedeutet, daß für diese Page davon ausgegangen wird, daß bereits auf sie zugegriffen wurde.

```
pte_t pte_mkread(pte_t pte) {
    pte_val(pte) |= _PAGE_USER; return pte;
}
```

setzt das Attribut *Leserlaubnis* für den entsprechenden Pagetableneintrag.

```
pte_t pte_mkwrite(pte_t pte) {
    pte_val(pte) |= _PAGE_RW; return pte;
}
```

setzt das Attribut *Schreiberlaubnis* für den entsprechenden Pagetableneintrag.

```
pte_t pte_mkyoung(pte_t pte) {
    pte_val(pte) |= _PAGE_ACCESSED; return pte;
}
```

löscht das Attribut *old* für den entsprechenden Pagetableneintrag, was bedeutet, daß für diese Page davon ausgegangen wird, daß auf sie noch nicht zugegriffen wurde.

```
pte_t pte_modify(pte_t pte, pgprot_t newprot) {
    pte_val(pte) = (pte_val(pte) &
                    _PAGE_CHG_MASK) | pgprot_val(newprot);
    return pte;
}
```

setzt die Attribute für den entsprechenden Pagetableneintrag auf die im Argument *newprot* angegebenen Attribute.

```
#define pte_none(x) (!pte_val(x))
```

prüft, der Pagetableneintrag gesetzt ist.

```
pte_t * pte_offset(pmd_t * dir, unsigned long address) {
    return (pte_t *) pmd_page(*dir) +
        ((address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1));
}
```

liefert die Adresse des Pagetableneintrags, der sich aus der linearen Adresse (*address*) und dem Eintrag im Page Middle Directory (*dir*), der auf die entsprechende Pagetabelle zeigt, ergibt.

```
unsigned long pte_page(pte_t pte){
    return pte_val(pte) & PAGE_MASK;
}
```

liefert die Adresse der Page, die durch den übergebenen Pagetableneintrag referenziert wird.

```
#define pte_present(x) (pte_val(x) & _PAGE_PRESENT)
```

prüft, ob die durch den Pagetableneintrag referenzierte Page im Speicher vorhanden ist.

```
pte_t pte_rdprotect(pte_t pte) {
    pte_val(pte) &= ~_PAGE_USER; return pte;
}
```

löscht das Leserecht für die entsprechende Page.

```
int pte_read(pte_t pte) {
    return pte_val(pte) & _PAGE_USER;
}
```

prüft, ob für den Pagetableneintrag das Attribut *Leseerlaubnis* gesetzt ist.

```
int pte_write(pte_t pte) {
    return pte_val(pte) & _PAGE_RW;
}
```

prüft, ob für den Pagetableneintrag das Attribut *Schreiberlaubnis* gesetzt ist.

```
pte_t pte_wrprotect(pte_t pte) {
    pte_val(pte) &= ~_PAGE_RW; return pte;
}
```

löscht das Schreibrecht für die entsprechende Page.

```
int pte_young(pte_t pte) {
    return pte_val(pte) & _PAGE_ACCESSED;
}
```

prüft, ob das Attribut *old* für die entsprechende Page gesetzt ist, ob also auf diese Page noch nicht zugegriffen wurde.

```
#define set_pte(ptepr, pteval) ((*(ptepr)) = (pteval))
```

setzt den Pagetableneintrag.

9.7.2 Der virtuelle Adreßraum eines Prozesses

Das Paging ist die unterste Ebene der Speicherverwaltung. Um aber die Ressourcen des Rechners effizient nutzen zu können, benötigt der Systemkern einen Mechanismus auf einer höheren Ebene, der die Sicht eines Prozesses auf seinen Speicher bereitstellt. Dieser Mechanismus wird unter Linux durch virtuelle Speicherbereiche (VMA=Virtual Memory Areas) bereitgestellt.

Virtuelle Speicherbereiche (Virtual Memory Areas)

Ein virtueller Speicherbereich ist ein zusammenhängender Bereich von Adressen im virtuellen Speicher eines Prozesses. Über diese virtuellen Speicherbereiche werden Segmente nachgebildet. Ein virtueller Speicherbereich wird durch die in `<linux/mm.h>` definierte Struktur `vm_area_struct` definiert:

```
struct vm_area_struct {
    /* Parameter für VMA */
    struct mm_struct * vm_mm; /* Zeiger auf Pagedirectory */
    unsigned long vm_start; /* Anfangsadresse des VMA */
    unsigned long vm_end; /* Endadresse des VMA */
    pgprot_t vm_page_prot; /* Schutzattribute für die
                           Pages des VMA */
}
```

```

unsigned short vm_flags;    /* Typ des Speicherbereichs,
                             wie z.B. Zugriffsrechte
                             auf den Speicherbereich
                             und Angaben, welche
                             Schutzattribute gesetzt
                             werden dürfen          */

/* AVL-Baum für die einzelnen Speicherbereiche
   eines Prozesses; sortiert nach Adressen          */
short  vm_avl_height;      /* Höhe des AVL-Baums          */
struct vm_area_struct *vm_avl_left; /* linker Nachfolger */
struct vm_area_struct *vm_avl_right; /* rechter Nachfolger */

/* Einfach verkettete Liste für die einzelnen Speicher-
   bereiche eines Prozesses; sortiert nach Adressen */
struct vm_area_struct * vm_next;

/* Doppelt verkettete Ringliste für einzelne Speicher-
   bereiche eines Prozesses; wird für spezielle Zwecke
   benötigt: Einblenden einer Datei oder Benutzen des
   Shared-Memory-Konzepts (von System V).
   Wird keine dieser beiden Punkte für den aktuellen
   Prozeß verwendet, werden die beiden folgenden
   Komponenten nicht genutzt.          */
struct vm_area_struct * vm_next_share;
struct vm_area_struct * vm_prev_share;

/* Liste von Operationen (Funktionszeiger) für die
   einzelnen Speicherbereiche des Prozesses
   (siehe unten)          */
struct vm_operations_struct * vm_ops;

/* Informationen zu einer in den virtuellen Speicher-
   bereich eingeblendete Datei bzw. Gerät:
   vm_inode zeigt auf die entsprechende Datei/Gerät,
   deren/dessen Inhalt ab
   vm_offset in des virtuellen Speicherbereich
   eingeblendet ist.          */
unsigned long vm_offset;
struct inode * vm_inode;

/* Information für das System V Shared Memory Konzept */
unsigned long vm_pte;
};

```

Die Speichertabelle eines Prozesses besteht aus einem Bereich für den Programmcode (text), einem Bereich für Daten (data: nicht initialisierte Daten und BSS²) und einem Bereich für den Stack. Zudem enthält diese Speichertabelle einen Bereich für jede aktive Speicherabbildung. Um sich die Speicherbereiche eines Prozesses anzeigen zu lassen,

2. Der Name BSS stammt aus den Assemblerzeiten. Damals existierte ein Assembleroperator namens *Block Started by Symbol*.

muß man sich nur die Datei `maps` im Directory `/proc/pid` (`pid` steht für die entsprechende Prozeßnummer) ausgeben lassen. Möchte man sich die Speicherbereiche des aktuellen Prozesses ausgeben lassen, muß anstelle der PID das Directory `self` angegeben werden.

```
$ cat /proc/self/maps
08048000-0804a000 r-xp 00000000 08:01 72334 [text für cat]
0804a000-0804b000 rw-p 00001000 08:01 72334 [data für cat]
0804b000-0804d000 rwxp 00000000 00:00 0 [BSS auf Null-Seite abgebildet]
40000000-40006000 r-xp 00000000 08:01 64273 [text für /lib/ld-linux.so.1.9.6]
40006000-40007000 rw-p 00005000 08:01 64273 [data für /lib/ld-linux.so.1.9.6]
40007000-40008000 rw-p 00000000 00:00 0 [BSS auf Null-Seite abgebildet]
40008000-4000b000 r--p 00000000 08:02 46923 [data für C-Lokale]
4000b000-4000f000 r-xp 00000000 08:01 64296 [text für /lib/libc.so.5.4.44]
4000f000-40095000 rw-p 00083000 08:01 64296 [data für /lib/libc.so.5.4.44]
40095000-400c7000 rw-p 00000000 00:00 0 [BSS auf Null-Seite abgebildet]
bffffd00-c0000000 rwxp fffff000 00:00 0 [auf Null abgebildeter Stack]
$ ls -li `which cat`
72334 /bin/cat
$ ls -li /lib/* | grep 64273
64273 /lib/ld-linux.so.1.9.6
$ ls -li /lib/* | grep 64296
64296 /lib/libc.so.5.4.44
$ ls -li /usr/share/locale/de_DE/LC_CTYPE
46923 /usr/share/locale/de_DE/LC_CTYPE
$ file /usr/share/locale/de_DE/LC_CTYPE
/usr/share/locale/de_DE/LC_CTYPE: data
$
```

Das Format einer Zeile in der `maps`-Datei ist das folgende:

start-end zugriffsrechte offset major:minor inode-nr

Bei den Zugriffsrechten steht

r	für Leseerlaubnis,
w	für Schreiberlaubnis,
x	für Ausführerlaubnis,
p	für »private« (bzw. auch s für »shared«).

Über die in `vm_area_struct` enthaltene Komponente `vm_ops` wird eine Liste von Operationen (Funktionszeiger) auf die unterschiedlichen Speicherbereiche des VMA angeboten. Der Datentyp von `vm_ops` ist die Struktur `vm_operations_struct`, die in `<linux/mm.h>` wie folgt definiert ist:

```
/*
 * These are the virtual MM functions - opening of an area, closing and
 * unmapping it (needed to keep files on disk up-to-date etc), pointer
 * to the functions called when a no-page or a wp-page exception occurs.
 */
```

```

struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    void (*unmap)(struct vm_area_struct *area,
                  unsigned long, size_t);
    void (*protect)(struct vm_area_struct *area,
                    unsigned long, size_t,
                    unsigned int newprot);
    int (*sync)(struct vm_area_struct *area,
                unsigned long, size_t,
                unsigned int flags);
    void (*advise)(struct vm_area_struct *area,
                   unsigned long, size_t,
                   unsigned int advise);
    unsigned long (*nopage)(struct vm_area_struct * area,
                            unsigned long address,
                            int write_access);
    unsigned long (*wppage)(struct vm_area_struct * area,
                            unsigned long address,
                            unsigned long page);
    int (*swapout)(struct vm_area_struct *,
                   unsigned long, pte_t *);
    pte_t (*swpin)(struct vm_area_struct *,
                   unsigned long, unsigned long);
};

```

Nachfolgend werden die einzelnen Funktionen kurz erläutert:

open

wird aufgerufen, wenn ein neuer virtueller Speicherbereich in das Benutzersegment eingeblendet wird.

close

wird aufgerufen, wenn ein virtueller Speicherbereich aus dem Benutzersegment auszublenden ist. Dabei ist eventuell eine Aktualisierung der entsprechenden Daten auf dem entsprechenden Speichermedium (wie z.B. Festplatte) notwendig.

unmap

wird aufgerufen, wenn ein Teil eines virtuellen Speicherbereichs ausgeblendet wird. Sollte der Teil den gesamten virtuellen Speicherbereich umfassen, wird anschließend noch **close** aufgerufen.

protect

wird in der Kernversion 2.0 nicht verwendet, da die Verwaltung der Zugriffsrechte nicht vom Bereich selbst abhängt.

sync

wird vom Systemaufruf **sync** aufgerufen, um einen veränderten Speicherbereich auf das Speichermedium zurückzuschreiben. Ist dieser Aufruf erfolgreich, liefert er 0 und sonst einen negativen Wert.

advise

wird in der Kernversion 2.0 nicht verwendet.

nopage

wird aufgerufen, wenn ein Prozeß versucht, auf eine Page zuzugreifen, die noch nicht im Speicher ist. Diese Funktion liefert dann die physikalische Adresse der Page zurück. Sollte diese Funktion nicht definiert sein, allokiert der Systemkern selbst eine leere Page. Das dritte Argument (*write_access*) zeigt an, ob eine gemeinsame (*shared*) Benutzung der Page durch mehrere Prozesse möglich ist: Der Wert 0 zeigt eine *shared*-Benutzung an, während ein von 0 verschiedener Wert anzeigt, daß diese Page *privat* ist, also nur vom aktuellen Prozeß genutzt werden kann.

wppage

ist für die Bearbeitung von *Page Faults* (Seitenfehler) bei schreibgeschützten Pages zuständig, wird jedoch in der Kernversion 2.0 nicht verwendet. Der Systemkern (von Version 2.0) behandelt Versuche, auf eine schreibgeschützte Page zu schreiben, selbst. Page Faults auf schreibgeschützte Pages werden verwendet, um das COW-Verfahren zu implementieren.

swapout

wird aufgerufen, wenn eine Page auszulagern ist. Welche Page auszulagern ist, wird über die einzelnen Argumente festgelegt: Das erste Argument gibt den Speicherbereich an, das zweite das Offset und das dritte die entsprechende Pagetabelle. Es ist sichergestellt, daß beim Aufruf von **swapout** bereits das *dirty*-Attribut für die entsprechende Page gesetzt ist

swpin

wird aufgerufen, wenn eine Page wieder zurück in den Speicher zu laden ist.

Im Systemkern werden virtuelle Speicherbereiche für einen Prozeß mit der in `<linux/mm.h>` deklarierten und in `mm/mmap.c` definierten Funktion **do_mmap** eingerichtet.

```
extern unsigned long do_mmap(struct file * file, unsigned long addr,  
                             unsigned long len, unsigned long prot,  
                             unsigned long flags, unsigned long off);
```

Für *file* ist dabei die *file*-Struktur der in den Speicher abzubildenden Datei anzugeben. Die weiteren Argumente entsprechen dem **mmap**-Aufruf für das Einrichten von Memory Mapped I/O (siehe auch Kapitel 15.3). Wird für *file* ein NULL-Zeiger angegeben, wird eine leere Page in das Benutzersegment eingeblendet, was man auch mit *anonymous mapping* bezeichnet.

Diese Funktion wird auch von den beiden in `<sys/mman.h>` deklarierten und in Kapitel 15.3 beschriebenen Funktionen **mmap** und **munmap** aus der C-Bibliothek verwendet.

Speicherallokierung im Kernsegment

Beim Starten des Systemkerns wird vor der Kreierung des ersten Prozesses mit entsprechenden Routinen statisch Speicher im Kernsegment allokiert, wie dies z.B. der folgende Ausschnitt aus der Routine **start_kernel** aus `init/main.c` zeigt:

```
memory_start = paging_init(memory_start, memory_end);
.....
memory_start = console_init(memory_start, memory_end);
.....
memory_start = kmalloc_init(memory_start, memory_end);
.....
memory_start = inode_init(memory_start, memory_end);
memory_start = file_table_init(memory_start, memory_end);
memory_start = name_cache_init(memory_start, memory_end);
.....
```

Die entsprechende Initialisierungsroutine reserviert Speicher dadurch, daß sie für das übergebene Argument `memory_start` einen neuen entsprechend erhöhten Wert zurückgibt. Der so von der Initialisierungsroutine reservierte Speicher kann von ihr dann beliebig für eigene Zwecke benutzt werden.

Zur dynamischen Speicherallokierung bzw. -freigabe verwendet der Systemkern die in `mm/kmalloc.c` definierten Funktionen **kmalloc** und **kfree**:

```
void *kmalloc(size_t size, int priority);
void kfree(void *__ptr); /* für die Freigabe von Speicher,
                        der mit kmalloc allokiert wurde */
```

Das erste Argument von **kmalloc** gibt die Größe des zu allozierenden Speichers an. Das zweite Argument `priority` legt das Verhalten von **kmalloc** fest. Meist wird hierfür die in `<linux/mm.h>` definierte Konstante `GFP_KERNEL` angegeben. Diese Konstante legt fest, daß die Allokierung durch einen Systemaufruf (also im Kernsegment) durchgeführt wird. In diesem Fall kann **kmalloc** seine Rückkehr verzögern, wenn weniger als `min_free_pages` Pages freier Speicher zur Zeit vorhanden ist. Sollte freier Speicher knapp sein, suspendiert diese Funktion den aktuellen Prozeß, bis eine neue Page frei wird.

Eine weitere mögliche Angaben für `priority` ist `GFP_ATOMIC` (atomare Speicherallokierung ohne Rücksicht auf den Wert von `min_free_pages`). Diese Konstante wird beispielsweise von Interrupthandlern verwendet.

Es existieren zwar noch weitere Konstanten, auf deren Erläuterung wird hier aber verzichtet.

Die Angabe der zu allozierenden Größe (`size`) durch **kmalloc** bedarf jedoch einiger Erläuterungen. Das von der Kernversion 2.0 verwendete Allokierungsverfahren bringt es mit sich, daß nur bestimmte vordefinierte Bytearrays fester Größe allokiert werden können. Fordert man eine beliebige Menge von Speicher an, bekommt man wahrscheinlich etwas mehr, als man anforderte. Die angebotenen Größen von Datenblöcken sind allgemein etwas weniger als eine Zweierpotenz. Benötigt man also in der Kernversion 2.0 z.B.

900 Byte, sollte man auch genau 900 anfordern und nicht etwa 1024 Byte. In Kernversionen vor 2.1.38 werden in diesem Fall dann doppelt soviel Byte (2048) allokiert, was natürlich eine Speicherplatzvergeudung ist. Zudem muß man wissen, daß **kmalloc** in der Kernversion 2.0 maximal etwas weniger als 32 Pages (256 KByte auf einem Alpha-Prozessor und 128 KByte auf einem Intel-Prozessor) allokieren kann.

Während die Verwendung von **kmalloc** für die Allokierung von kleineren Speicherbereichen (kleiner als 4072 Byte) ratsam ist, sollten für die Allokierung größerer Speicherbereiche die beiden in `mm/vmalloc.c` definierten Funktionen **vmalloc** und **vfree** verwendet werden:

```
void * vmalloc(unsigned long size);  
void vfree(void * addr); /* für die Freigabe von Speicher,  
                           der mit vmalloc allokiert wurde */
```

Für `size` kann dabei eine durch 4096 teilbare Zahl angegeben werden, die dann entsprechend von **vmalloc** aufgerundet wird. Natürlich kann nicht mehr Speicher angefordert werden, als zur Zeit frei ist. Da der von **vmalloc** allokierte Speicher nicht ausgelagert wird, sollte mit dieser Funktion nicht allzu großzügig Speicher allokiert werden. Da **vmalloc**, ebenso wie **kmalloc**, die Funktion `__get_free_pages` aufruft, kann auch hier der aufrufende Prozeß blockiert werden, wenn zur Zeit nicht genug freier Speicher vorhanden ist.

Nach dem Aufrunden der angegebenen `size` sucht **vmalloc** eine Adresse, an der der zu allozierende Speicherbereich komplett in das Kernsegment eingeblendet werden kann.

Der Vorteil von **vmalloc** liegt darin, daß die Größe des wirklich allokierten Speicherbereichs nicht allzu weit von dem angeforderten Speicherbereich (`size`) abweicht, was bei **kmalloc** nicht der Fall ist. Fordert man bei **kmalloc** etwa 64 KByte an, so werden in Wirklichkeit 128 KByte allokiert, was eine erhebliche Speicherplatzvergeudung ist. Ein weiterer Vorteil von **vmalloc** ist, daß der von dieser Funktion zu allozierende Speicher nur durch die Größe des physikalisch vorhandenen Speichers beschränkt ist und nicht durch die Segmentierung wie bei **kmalloc**.

Da **vmalloc** keine physikalischen Adressen zurückgibt und die allokierten Speicherbereiche über nicht zusammenhängenden Pages im Speicher verstreut sein können, eignet sich **vmalloc** nicht für die Speicherallokierung von Speicher für das DMA (*Direct Memory Access*).

9.7.3 Paging

Linux arbeitet nach einem Konzept, das mit *Demand Paging* bezeichnet wird. Dabei wird mit Hilfe des MMU (*Memory Management Unit*) der gesamte Speicher in *Pages* (Speicherseiten) unterteilt. Es werden bei diesem Verfahren nun nicht – wie beim traditionellen und nicht sehr effektiven *Swapping*-Verfahren – ganze Prozesse aus dem Hauptspeicher (primärer Speicher) auf einen sekundären Speicher (wie z.B. eine Festplatte) ausgelagert und bei Bedarf wieder eingelagert, sondern eben immer nur einzelne Pages, unabhängig davon welchen Prozessen diese zugeteilt sind.

Es gelten dabei die folgenden allgemeinen Regeln:

- ▶ Pages des Kernsegments dürfen niemals ausgelagert werden, da diese Informationen enthalten, die für das Einlagern wieder benötigt werden und deshalb immer im primären Speicher vorhanden sein müssen.
- ▶ Pages, die ohne Schreiberlaubnis direkt mit **do_mmap** in den virtuellen Adreßraum eines Prozesses eingeblendet wurden, werden erst gar nicht ausgelagert, sondern einfach weggeworfen. Ihr Inhalt kann jederzeit wieder aus den eingeblendeten Dateien gelesen werden.
- ▶ Pages, deren Inhalt verändert wurde, müssen in jedem Fall in Auslagerungsbereiche übertragen werden.
- ▶ Als Auslagerungsbereich kann unter Linux entweder eine ganze Partition (*Swap-Partition*) oder aber eine Datei (*Swap-Datei*) mit fester Größe verwendet werden. Dazu ist zu sagen, daß diese Begriffe eigentlich falsch sind (siehe dazu auch vorher), und man korrekterweise von einer *Paging-Partition* bzw. *Paging-Datei* sprechen müßte. Um nicht allzu große Konfusion aufkommen zu lassen, werden hier aber die üblichen Begriffe (*Swap-Partition* und *Swap-Datei*) verwendet.

Sowohl für eine Swap-Partition als auch für eine Swap-Datei wird die gleiche Struktur verwendet: Die ersten 4086 Byte enthalten eine Bitmap, bei der gesetzte Bits anzeigen, daß die entsprechende Page für Auslagerungen zur Verfügung steht. An der Adresse 4086 befindet sich dann als Kennung der String »SWAP-SPACE« .

```
$ fdisk -l
```

```
Disk /dev/sda: 255 heads, 63 sectors, 292 cylinders
Units = cylinders of 16065 * 512 bytes
```

Device	Boot	Begin	Start	End	Blocks	Id	System
/dev/sda1		1	1	20	160618+	83	Linux native
/dev/sda2		21	21	171	1212907+	83	Linux native
/dev/sda3		172	172	280	875542+	5	Extended
<u>/dev/sda4</u>		281	281	292	96390	82	<u>Linux swap</u>
/dev/sda5		172	172	222	409626	83	Linux native
/dev/sda6		223	223	254	257008+	83	Linux native
/dev/sda7		255	255	280	208813+	83	Linux native

```
$ od -xc --address-radix=d /dev/sda4
0000000 fffe ffff ffff ffff ffff ffff ffff ffff
      p  y  y  y  y  y  y  y  y  y  y  y  y  y  y  y  y
0000016 ffff ffff ffff ffff ffff ffff ffff ffff
      y  y  y  y  y  y  y  y  y  y  y  y  y  y  y  y
*
0003008 ffff ffff 0001 0000 0000 0000 0000 0000
      y  y  y  y 001 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0003024 0000 0000 0000 0000 0000 0000 0000 0000
      \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0004080 0000 0000 0000 5753 5041 532d 4150 4543
      \0 \0 \0 \0 \0 \0  S  W  A  P  -  S  P  A  C  E
```

```

0004096 7462 6572 0065 4009 0000 0000 0011 0000
      b t r e e \0 \t @ \0 \0 \0 \0 021 \0 \0 \0
0004112 696e 0073 3240 4009 0000 0000 0011 0000
      n i s \0 @ 2 \t @ \0 \0 \0 \0 021 \0 \0 \0
0004128 6562 7473 786d 4000 ffff ffff 0019 0000
      b e s t m x \0 @ ŷ ŷ ŷ ŷ 031 \0 \0 \0
0004144 6f68 7473 3200 4009 0000 0000 0000 0000
      h o s t \0 2 \t @ \0 \0 \0 \0 \0 \0
0004160 0018 0000 0109 0000 6164 6d65 6e6f 7800
      030 \0 \0 \0 \t 001 \0 \0 d a e m o n \0 x
.....
$

```

Aus dieser vorgegebenen Struktur kann man ableiten, daß maximal 32687 ($4086 \cdot 8 - 1$) Pages in einer Swap-Partition bzw. Swap-Datei untergebracht werden können, was in etwa 130 MByte entspricht. Da man mehrere Swap-Partitionen bzw. Swap-Dateien gleichzeitig benutzen kann, ist dies keine allzu große Einschränkung. Wie viele solche Swap-Partitionen bzw. Swap-Dateien man parallel verwenden kann, ist in `<linux/swap.h>` festgelegt:

```
#define MAX_SWAPFILES 8
```

Diese Konstante kann man bei Bedarf bis auf 63 hochsetzen.

Das Anmelden einer Swap-Partition oder Swap-Datei erfolgt mit der folgenden in `mm/swapfile.c` definierten Funktion **swapon**:

```
asmlinkage int sys_swapon(const char * specialfile, int swap_flags)
```

Dabei gibt der Parameter `specialfile` den Namen der Swap-Partition bzw. der Swap-Datei an, und der Parameter `swap_flags` legt die Priorität des Auslagerungsbereichs fest. Dazu werden in `<linux/swap.h>` die folgenden Konstanten angeboten:

```

#define SWAP_FLAG_PREFER      0x8000 /* set if swap priority specified */
#define SWAP_FLAG_PRIO_MASK   0x7fff
#define SWAP_FLAG_PRIO_SHIFT  0

```

Ist `SWAP_FLAG_PREFER` gesetzt, dann geben die Bits in `SWAP_FLAG_PRIO_MASK` die positive Priorität des Auslagerungsbereichs an. Ist keine Priorität für einen Auslagerungsbereich vorgesehen, wird diesem automatisch eine negative Priorität zugeordnet, die bei jedem Aufruf von **swapon** weiter abnimmt. Die Prioritätsbehandlung zeigt der folgende Codeausschnitt der Funktion **swapon** (aus Datei `mm/swapfile.c`):

```

static int least_priority = 0;
.....
if (swap_flags & SWAP_FLAG_PREFER) {
    p->prio =(swap_flags & SWAP_FLAG_PRIO_MASK)>>SWAP_FLAG_PRIO_SHIFT;
} else {
    p->prio = --least_priority;
}

```

Die Funktion **swapon** richtet für den Auslagerungsbereich einen Eintrag in der ebenfalls in `mm/swapfile.c` definierten Tabelle `swap_info` ein:

```
int nr_swapfiles = 0;

struct swap_info_struct swap_info[MAX_SWAPFILES];
```

Der Datentyp eines Eintrags in dieser Tabelle ist wie folgt in `<linux/swap.h>` definiert:

```
struct swap_info_struct {
    unsigned int  flags;
    kdev_t        swap_device;
    struct inode  * swap_file;
    unsigned char * swap_map;
    unsigned char * swap_lockmap;
    int lowest_bit;
    int highest_bit;
    int cluster_next;
    int cluster_nr;
    int prio;      /* swap priority          */
    int pages;
    unsigned long max;
    int next;      /* next entry on swap list */
};
```

Ist in `flags` das Bit `SWP_USED` für einen Eintrag gesetzt, so zeigt dies an, daß der Eintrag vom Systemkern schon für einen anderen Auslagerungsbereich genutzt wird. Beim Aufruf von **swapon** wird der erste freie Eintrag in der Tabelle `swap_info` gesucht, was durch folgenden Codeausschnitt in der Funktion **swapon** realisiert wird:

```
struct swap_info_struct * p;
.....
p = swap_info;
for (type = 0 ; type < nr_swapfiles ; type++,p++)
    if (!(p->flags & SWP_USED))
        break;
if (type >= MAX_SWAPFILES)
    return -EPERM;
if (type >= nr_swapfiles)
    nr_swapfiles = type+1;
p->flags = SWP_USED;
.....
```

Nachdem alle Initialisierungen für den Auslagerungsbereich abgeschlossen sind, wird `flags` auf `SWP_WRITEOK` gesetzt. Diese Konstante ist ebenso wie die Konstante `SWP_USED` in `<linux/swap.h>` definiert.

Handelt es sich beim Argument `specialfile` um ein Gerät (Swap-Partition), wird die Komponente `swap_device` gesetzt. Handelt es sich dagegen beim Argument `specialfile` um eine Datei (Swap-Datei), wird die Komponente `swap_file` gesetzt, wofür der folgende Codeausschnitt in **swapon** zuständig ist:

```

p->swap_file = swap_inode;
error = -EBUSY;
if (swap_inode->i_count != 1)
    goto bad_swap_2;
error = -EINVAL;
if (S_ISBLK(swap_inode->i_mode)) {
    p->swap_device = swap_inode->i_rdev;
    .....
} else if (!S_ISREG(swap_inode->i_mode))
    goto bad_swap;

```

Die Komponente `swap_map` zeigt auf eine mit **`vmalloc`** allokierte Tabelle, in der für jede Page im Auslagerungsbereich ein Byte vorgesehen ist. Die Zahl in einem solchen Byte zeigt an, wie viele Prozesse auf diese Page verweisen. Der Wert `0x80` wird dabei verwendet, um anzuzeigen, daß die Page zur Zeit nicht benutzt werden kann. Zusätzlich existiert noch die Komponente `swap_lockmap`, die als eine Bittabelle organisiert ist, bei der jeder Page ein Bit zugeordnet ist. Ein gesetztes Bit zeigt dabei an, daß gerade auf die entsprechende Page zugegriffen wird, was bedeutet, daß diese Page momentan nicht gelesen und nicht beschrieben werden darf. Das Initialisieren dieser beiden Tabellen geschieht in der Funktion **`swapon`** im folgenden Codeausschnitt:

```

.....
p->swap_map = (unsigned char *) vmalloc(p->max);
if (!p->swap_map) {
    error = -ENOMEM;
    goto bad_swap;
}
for (i = 1 ; i < p->max ; i++) {
    if (test_bit(i,p->swap_lockmap))
        p->swap_map[i] = 0;
    else
        p->swap_map[i] = 0x80;
}
p->swap_map[0] = 0x80;
memset(p->swap_lockmap,0,PAGE_SIZE);
.....

```

Weitere Komponenten in `swap_info_struct` enthalten die folgenden Informationen:

<code>pages</code>	Anzahl der Pages, die im Auslagerungsbereich beschrieben werden dürfen
<code>lowest_bit</code>	minimale Offset einer freien Page im Auslagerungsbereich
<code>highest_bit</code>	maximale Offset einer freien Page im Auslagerungsbereich
<code>max</code>	entspricht nach Beendigung der Funktion <code>swapon</code> dem Wert <code>highest_bit+1</code> ; dieser Wert wird so häufig benötigt, daß für ihn eine eigene Komponente vorgesehen ist.
<code>prio</code>	enthält die dem Auslagerungsbereich zugeordnete Priorität.
<code>next</code>	Die einzelnen Auslagerungsbereiche sind in einer einfach verketteten Liste entsprechend ihrer Priorität geordnet. Die Komponente <code>next</code> zeigt auf den nächsten Auslagerungsbereich in dieser Liste.

Der folgende Codeausschnitt zeigt die Belegung dieser Komponenten in der Funktion **swapon**:

```
static struct {
    int head;          /* head of priority-ordered swapfile list */
    int next;          /* swapfile to be used next */
} swap_list = {-1, -1};
.....

p->lowest_bit = 0;
p->highest_bit = 0;
for (i = 1 ; i < 8*PAGE_SIZE ; i++) {
    if (test_bit(i,p->swap_lockmap)) {
        if (!p->lowest_bit)
            p->lowest_bit = i;
        p->highest_bit = i;
        p->max = i+1;
        j++;
    }
}
.....
/* insert swap space into swap_list: */
prev = -1;
for (i = swap_list.head; i >= 0; i = swap_info[i].next) {
    if (p->prio >= swap_info[i].prio) {
        break;
    }
    prev = i;
}
p->next = i;
if (prev < 0) {
    swap_list.head = swap_list.next = p - swap_info;
} else {
    swap_info[prev].next = p - swap_info;
}
return 0;
bad_swap:
.....
```

Um unnötige Bewegungen des Schreib-/Lese Kopfes auf einer Festplatte bei nacheinander stattfindenden Auslagerungen von Pages zu vermeiden, werden neu auszulagernde Pages als zusammenhängende Gruppen (Cluster) im Auslagerungsbereich gespeichert. Dazu dienen die beiden Komponenten `cluster_next` und `cluster_nr`.

Das Gegenstück zur Funktion **swapon** ist die ebenfalls in `mm/swapfile.c` definierte Funktion **swapoff**:

```
asmlinkage int sys_swapoff(const char * specialfile)
```

Diese Funktion meldet eine Swap-Partition bzw. Swap-Datei beim Systemkern wieder ab. Eine solche Abmeldung ist jedoch nur erfolgreich, wenn im Hauptspeicher oder in anderen Auslagerungsbereichen genügend Platz vorhanden ist, um die Pages aus diesem Auslagerungsbereich, der abgemeldet werden soll, aufzunehmen.

Die Speichertabelle

Bei der Speicherverwaltung unter Linux gibt es neben den virtuellen Speicherbereichen (VMAs) und den Pagedirectories bzw. Pagetabellen, die den virtuellen Adreßraum organisieren, noch eine dritte Datenstruktur, die Speichertabelle (*memory map*), die für die Organisation des physikalischen Speichers zuständig ist.

Der Systemkern benötigt Information darüber, wie der physikalische Speicher zur Zeit verwendet wird. Da der Speicher lediglich als ein Array von Pages betrachtet wird, unterhält der Kern eine Tabelle (Array), in der zu jeder verfügbaren Page des physikalischen Speichers entsprechende Informationen enthalten sind. Ein Eintrag in dieser Tabelle hat die folgende in `<linux/mm.h>` definierte Datenstruktur (`struct page` bzw. `mem_map_t`), und auf die Tabelle zeigt der ebenfalls in `<linux/mm.h>` deklarierte Zeiger `mem_map`:

```
/* Try to keep the most commonly accessed fields in
 * single cache lines; here (16 bytes or greater).
 * This ordering should be particularly beneficial
 * on 32-bit processors.
 * The first line is data used in page cache lookup,
 * the second line is used for linear searches
 * (eg. clock algorithm scans).
 */
typedef struct page {
    /* these must be first (free area handling) */
    /* Nachfolger in doppelt verketteter Ringliste */
    struct page *next;

    /* Vorgänger in doppelt verketteter Ringliste */
    struct page *prev;

    /* Datei, aus der die Page eingelesen wurde;
     * zu jedem inode existiert eine Liste, in der alle
     * Pages eingetragen sind, die aus dieser Datei
     * eingelesen wurden. */
    struct inode *inode;

    /* Offset in der Datei, von wo Page eingelesen wurde */
    unsigned long offset;

    /* Nachfolger in der page_hash_table (siehe unten) */
    struct page *next_hash;

    /* Anzahl der Nutzer dieser Page */
    atomic_t count;

    /* gesetzte Flags für diese Page (siehe ????) */
    unsigned flags; /* atomic flags, some possibly
                     * updated asynchronously */

    /* age gibt Alter der Page an; dirty z.Z. ungenutzt */
    unsigned dirty:16,
           age:8;
```

```

    /* Warteschlange von Tasks, die auf das Aufheben
       der Sperre für diese Page warten */
    struct wait_queue *wait;

    /* Vorgänger in der page_hash_table (siehe unten) */
    struct page *prev_hash;

    /* Blockpuffer bei blockorientierten Geräten */
    struct buffer_head * buffers;

    /* Nummer der Page im Auslagerungsbereich, bei der
       Sperre aufzuheben ist, wenn Page gelesen wurde. */
    unsigned long swap_unlock_entry;

    /* Nummer der Page */
    unsigned long map_nr; /* page->map_nr == page->mem_map */
} mem_map_t;

extern mem_map_t * mem_map;

```

Diese Datenstruktur ist so aufgebaut, daß zusammengehörige Daten immer in einer Cachezeile (16 Byte) gespeichert werden.

Nun noch einige Erläuterungen zu den beiden Komponenten `next_hash` und `prev_hash`. Sie zeigen auf Einträge in der Hashtabelle `page_hash_table`, die in `<linux/pagemap.h>` wie folgt definiert ist:

```

#define PAGE_HASH_BITS 11
#define PAGE_HASH_SIZE (1 << PAGE_HASH_BITS)

extern struct page * page_hash_table[PAGE_HASH_SIZE];

```

Die Hashfunktion ist ebenfalls in `<linux/pagemap.h>` definiert:

```

/* We use a power-of-two hash table to avoid a modulus,
 * and get a reasonable hash by knowing roughly how the
 * inode pointer and offsets are distributed (ie, we
 * roughly know which bits are "significant")
 */
static inline unsigned long _page_hashfn(struct inode * inode,
                                         unsigned long offset)
{
    #define i (((unsigned long) inode)/ \
              (sizeof(struct inode) & ~ (sizeof(struct inode) - 1)))
    #define o (offset >> PAGE_SHIFT)
    #define s(x) ((x)+((x)>>PAGE_HASH_BITS))
        return s(i+o) & (PAGE_HASH_SIZE-1);
    #undef i
    #undef o
    #undef s
}

#define page_hash(inode,offset) (page_hash_table+_page_hashfn(inode,offset))

```

Wie zu sehen ist, benutzt die Hashfunktion den i-node und das Offset der Datei, zu der die Page gehört. Soll von einer Page aus einer Datei gelesen werden, wird zuerst geprüft, ob die Page bereits in der Hashtabelle vorhanden ist. Ist dies der Fall, braucht sie nicht zeitaufwendig aus dem Filesystem gelesen werden, sondern kann sofort aus dem Speicher gelesen werden. Lesezugriffe finden damit im Pagecache statt.

Tabelle 9.1 zeigt die möglichen Angaben von Konstanten, die in `<linux/mm.h>` definiert sind:

Flag	Bedeutung
PG_locked	Page wird gesperrt.
PG_error	Bei dieser Page ist eine Fehlerbedingung aufgetreten.
PG_referenced	Auf diese Page wurde vor kurzem zugegriffen.
PG_uptodate	Page is <i>uptodate</i> , was bedeutet, daß ihr Inhalt mit dem Inhalt auf der Festplatte übereinstimmt.
PG_free_after	Page soll nach einer E / A-Operation freigegeben werden.
PG_decr_after	Der Zähler <code>nr_async_pages</code> (in <code><linux/swap.h></code> definiert) ist nach dem Lesen dieser Page zu dekrementieren.
PG_swap_unlock_after	Nach dem Lesen aus dem Auslagerungsbereich ist die Sperre für diese Page mit dem Aufruf der Funktion <i>swap_after_unlock_page</i> (in <code>mm/page_io.c</code> definiert) aufzuheben.
PG_reserved	Diese Page ist reserviert.

Tabelle 9.1: Mögliche Werte für die Komponente *flags* in der *page*-Struktur

Die Page-Speicherverwaltung

Zur Reservierung von physikalischen Pages ruft der Systemkern die Funktion `__get_free_pages` auf, die in `mm/page_alloc.c` definiert ist:

```
unsigned long __get_free_pages(int priority,
                               unsigned long order, int dma)
```

Für *priority* können die in `<linux/mm.h>` definierten Konstanten angegeben werden (siehe Tabelle 9.2). Nur bei den beiden Konstanten `GFP_BUFFER` und `GFP_ATOMIC` ist garantiert, daß der aktuelle Prozeß durch den Aufruf von `__get_free_pages` nicht unterbrochen wird.

Konstante	Bedeutung
GFP_BUFFER	Nur dann eine Page reservieren, wenn im physikalischen Speicher noch Pages frei sind. Diese Konstante wird bei der Puffercache-Verwaltung gesetzt, um zu verhindern, daß für den Cache Pages von Prozessen ausgelagert oder im Extremfall sogar der ganze Puffercache geleert wird.
GFP_ATOMIC	Aktueller Prozeß darf zum Auslagern von Pages nicht von der Funktion <code>__get_free_pages</code> unterbrochen werden. Es sollte aber, wenn möglich, eine Page zurückgegeben werden. Interrupthandler verwenden üblicherweise diese Konstante.
GFP_USER	Aktueller Prozeß darf zum Auslagern von Pages unterbrochen werden.
GFP_KERNEL	entspricht der Konstante GFP_USER.
GFP_NOBUFFER	Puffercache wird nicht verkleinert, um eine freie Page zu finden.
GFP_NFS	weitgehend identisch zu GFP_USER, nur mit dem Unterschied, daß die Zahl der für GFP_ATOMIC reservierten Pages (<code>min_free_pages</code>) auf 5 heruntergesetzt wird, was sich positiv auf die Geschwindigkeit der NFS-Operationen auswirkt.

Tabelle 9.2: Mögliche Angaben für den *priority*-Parameter bei der Funktion `__get_free_pages`

Der zweite Parameter `order` legt die Zweierpotenz (2^{order}) von Pages fest, die für den Speicherblock zu reservieren sind. Der maximal erlaubte Wert für `order` muß kleiner als die in `mm/page_alloc.c` definierte Konstante `NR_MEM_LISTS` sein:

```
#define NR_MEM_LISTS 6
```

Folglich können nur Speicherblöcke allokiert werden, die 1, 2, 4, 8, 16 oder 32 Pages umfassen, was Größen von 4, 8, 16, 32, 64 oder 128 Kbytes entspricht.

Der letzte Parameter `dma` legt fest, daß die reservierten Pages DMA-fähig sein sollen.

Falls der angeforderte Speicherblock erfolgreich allokiert werden konnte, liefert `__get_free_pages` diese Adresse als Rückgabewert.

Zur Verwaltung der freien und belegten Pages des physikalischen Speichers unterhält der Systemkern eine Tabelle, die in `mm/page_alloc.c` wie folgt definiert ist:

```
/* The start of this MUST match the start of "struct page" */
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int *map;
};

#define memory_head(x) ((struct page *) (x))

static struct free_area_struct free_area[NR_MEM_LISTS];
```

Als Index für die Tabelle `free_area` wird dabei die `order` verwendet. Die entsprechenden Speicherblöcke dieser Größe liegen in jedem Eintrag dieser Tabelle als doppelt verkettete Ringliste (Komponenten `next` und `prev`) vor. Der Kopf dieser Ringliste ist der eigene Eintrag (siehe auch obiges Makro `memory_head`).

Die Komponente `map` eines jeden Tabelleneintrags zeigt auf eine Bitmap. Jedes Bit dieser Bitmap ist zwei aufeinanderfolgenden Speicherblöcke der jeweiligen Größe (`order`) zugeordnet. Das Bit ist gesetzt, wenn einer der beiden Speicherblöcke (der Größe `order`) frei ist und im anderen zumindest eine Page belegt ist.

Der verwendete Allokierungsalgorithmus ist so ausgelegt, daß niemals zwei aufeinanderfolgende Speicherblöcke frei sind, die zu einem größeren Speicherblock zusammengefaßt werden können. Sollte diese Vorgehensweise dazu führen, daß keine Speicherblöcke für die niedrigeren Ordnungen mehr verfügbar sind, müssen Speicherblöcke der höheren Ordnungen geteilt werden, wofür das in `mm/page_alloc.c` definierte Makro **EXPAND** zuständig ist.

Grundsätzlich versucht die Funktion `__get_free_pages` einen Speicherblock mit der angeforderten Größe (`order`) in der zugehörigen Liste von freien Speicherblöcken zu finden. Sollte dies nicht möglich sein, kann diese Funktion bei Angabe einer der beiden Konstanten `GFP_BUFFER` oder `GFP_ATOMIC` für den Parameter `priority` die gewünschte Speicheranforderung nicht erfüllen und kehrt sofort wieder zum Aufrufer zurück. Sind andere Konstanten für den Parameter `priority` angegeben (siehe Fehler!), wird in diesem Fall die in `mm/vmscan.c` definierte Funktion `try_to_free_page` aufgerufen. Diese Funktion `try_to_free_page`, die als Zustandsautomat implementiert ist, versucht in mehreren Durchläufen freie Pages zu finden, wobei sie mit jedem Durchlauf »aggressiver« wird. Im ersten Durchlauf beispielsweise versucht sie mit der in `mm/filemap.c` definierten Funktion

```
int shrink_mmap(int priority, int dma, int free_buf)
```

Pages aus dem Page- bzw. Puffercache zu entfernen, die nur von einem Benutzer genutzt werden und auf die seit dem letzten Durchlauf nicht mehr zugegriffen wurde. Im nächsten Durchlauf versucht sie mit der in `ipc/shm.c` definierten Funktion

```
int shm_swap(int priority, int dma)
```

Speicherbereiche auszulagern, die für *Shared Memory* (siehe Kapitel 18.4) vorgesehen sind. Im nachfolgenden Durchlauf versucht sie mit der in `mm/vmscan.c` definierten Funktion

```
static int swap_out(unsigned int priority, int dma,
                    int wait, int can_do_io)
```

Pages aus dem Benutzersegment der Prozesse auszulagern oder zu entfernen.

Falls beim Aufruf von `try_to_free_page` für den Parameter `wait` ein von 0 verschiedener Wert übergeben wird, werden diese drei Schritte nochmals wiederholt, jedoch nun mit einer höheren Priorität (Parameter `priority` in allen drei Funktionen). Höhere Priorität

Dieser Routine werden über den Parameter `regs` die Werte der Register (zum Zeitpunkt der Unterbrechung) und über den Parameter `error_code` der Fehlercode übergeben. **`do_page_fault`** sucht dann nach dem virtuellen Speicherbereich (VMA) des gerade aktiven Prozesses, in dem die Adresse (im Benutzersegment) liegt, die den Fehler auslöste.

```
/* get the address */
__asm__("movl %%cr2,%0":"=r" (address));
down(&mm->mmap_sem);
vma = find_vma(mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;
```

Findet sie die Adresse nicht in einem der virtuellen Speicherbereiche, überprüft sie, ob das Flag `VM_GROWSDOWN` für den nächsten virtuellen Speicherbereich gesetzt ist. Dieses Flag zeigt an, daß der Bereich nach unten verlängert werden kann. Ist dieses Flag gesetzt, so verlängert die Funktion **`do_page_fault`** den nächsten virtuellen Speicherbereich mit der Funktion **`expand_stack`**. Sollte das Flag `VM_GROWSDOWN` nicht gesetzt sein oder der Aufruf von **`expand_stack`** nicht erfolgreich sein, wird die Marke `bad_area` angesprungen, wo dann das Signal `SIGSEGV` (*Segmentation Violation*) dem Prozeß geschickt wird, der diesen Fehler auslöste.

```
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4) {
    /* accessing the stack below %esp is always a bug.
     * The "+ 32" is there due to some instructions (like
     * pusha) doing pre-decrement on the stack and that
     * doesn't show up until later..
     */
    if (address + 32 < regs->esp)
        goto bad_area;
}
if (expand_stack(vma, address))
    goto bad_area;
```

An der Marke `good_area` wird dann anhand der Flags des entsprechenden virtuellen Speicherbereichs geprüft, ob die angeforderten Operationen (Schreiben bzw. Lesen) hierfür erlaubt sind.

```
/* Ok, we have a good vm_area for this memory access,
 * so we can handle it..
 */
good_area:
    write = 0;
    handler = do_no_page;
    switch (error_code & 3) {
        default: /* 3: write, present */
            handler = do_wp_page;
    }
#ifdef TEST_VERIFY_AREA
```

```

        if (regs->cs == KERNEL_CS)
            printk("WP fault at %08lx\n", regs->eip);
#endif

        /* fall through */
        case 2:          /* write, not present */
            if (!(vma->vm_flags & VM_WRITE))
                goto bad_area;
            write++;
            break;
        case 1:          /* read, present */
            goto bad_area;
        case 0:          /* read, not present */
            if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
                goto bad_area;
    }
    handler(tsk, vma, address, write);
    up(&mm->mmap_sem);
    /*
     * Did it hit the DOS screen memory VA from vm86 mode?
     */
    if (regs->eflags & VM_MASK) {
        unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
        if (bit < 32)
            tsk->tss.screen_bitmap |= 1 << bit;
    }
    return;

```

Sollten die geforderten Operationen erlaubt sein, wird eine der beiden in `mm/memory.c` definierten Funktionen **`do_no_page`** bzw. **`do_wp_page`** aufgerufen:

```

/* do_no_page() tries to create a new page mapping.
 * It aggressively tries to share with existing pages,
 * but makes a separate copy if the "write_access" parameter
 * is true in order to avoid the next page fault.
 * As this is called only for pages that do not
 * currently exist, we do not need to flush old virtual
 * caches or the TLB.
 */
void do_no_page(struct task_struct * tsk,
                struct vm_area_struct * vma,
                unsigned long address,
                int write_access)
{
    .....
}

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *
 * Note that this routine assumes that the protection checks have been
 * done by the caller (the low-level page fault routine in most cases).

```



```

* Thus we can safely just mark it writable once we've done any necessary
* COW.
*
* We also mark the page dirty at this point even though the page will
* change only once the write actually happens. This avoids a few races,
* and potentially makes it more efficient.
*/
void do_wp_page(struct task_struct * tsk,
                struct vm_area_struct * vma,
                unsigned long address,
                int write_access)
{
    .....
}

```

Die Funktion **do_wp_page** prüft, ob eine schreibgeschützte Page überhaupt unter der entsprechenden Adresse vorhanden ist. Ist diese nur einmal referenziert, wird lediglich ihr Schreibschutz aufgehoben. Ist sie dagegen mehrmals referenziert, wird diese Page kopiert und die Kopie als nicht schreibgeschützte Page in die Pagetabelle des Prozesses eingetragen, der den Fehler auslöste.

Auf eine weitergehende Erläuterung der Funktion **do_no_page** wird hier verzichtet.



9.8 Übung

9.8.1 Ändern des Environment eines Elternprozesses nicht möglich

Es ist immer nur möglich, das Environment des aktuellen Prozesses, aber niemals das des Elternprozesses zu ändern. Ein aktueller Prozess kann zwar Environment-Variablen mit **export** (in *sh* und *ksh*) oder **setenv** (in *csh*) an seine Kindprozesse vererben, aber niemals an seinen Elternprozess. Erklären Sie dieses Phänomen!

9.8.2 Zugriff auf Adresse 0 des Datensegments meist nicht erlaubt

Auf vielen Systemen ist es nicht möglich, auf die Adresse 0 des Datensegments zuzugreifen. Was mag hierfür der Grund sein?

9.8.3 Gefahren bei der Verwendung von lokalen Variablen

Sind die folgenden Programmteile korrekt oder nicht?

a) Eine elegante Allokierungsroutine, oder nicht?

```

void *
allokiere(int groesse)

```

```

{
    char    array[groesse];

    return(array);
}

```

b) Rückgabe eines Zeigers auf eine lokale Variable

```

int
dividiere(int a, int b)
{
    int    *zgr;

    if (b != 0) {
        int    ergeb;

        ergeb = a+b;
        zgr = &ergeb;
    }

    return(*zgr);
}

```

c) Schreiben in eine Struktur über einen Zeiger

```

struct adresse {
    char name[100];
    int  alter;
};

struct adresse *zgr;
.....
void funktion(....)
{
    .....
    strcpy(zgr->name, "Hans Mayer");
    zgr->alter = 10;
    .....
}
.....

```

9.8.4 Eigene Implementierung von `getenv`, `putenv`, `setenv` und `unsetenv`

Erstellen Sie ein Programm `mein_env.c`, das interaktiv das Ändern bzw. Erfragen der Environment-Variablen ermöglicht, wie z.B.:

```

-----
Environment-Liste
-----

```

- 0: Ende
- 1: Gesamte Environment-Liste anzeigen

- 2: Einzelnen Namen lesen (*mein_getenv*)
- 3: Neuen Eintrag schreiben (*mein_putenv*)
- 4: Neuen Eintrag schreiben (*mein_setenv*)
- 5: Eintrag löschen (*mein_unsetenv*)

Deine Wahl:

Dieses Programm soll dabei nicht die vorgegebenen Routinen ***getenv***, ***putenv***, ***setenv*** und ***unsetenv*** verwenden, sondern eigene Funktionen *mein_getenv*, *mein_putenv*, *mein_setenv* und *mein_unsetenv* für das Erfragen, Setzen oder Löschen von Environment-Variablen einsetzen.

Hinweis

Die Environment-Liste (Array von Zeigern auf Strings der Form »name=wert«) und die Environment-Strings selbst sind üblicherweise am Anfang des Adreßraums eines Prozesses (oberhalb des Stacks) untergebracht (siehe auch Abbildung 9.3). Dieser Environment-Speicherbereich kann weder nach oben erweitert werden, da er sich schon an der obersten Stelle des Adreßraums befindet, noch kann er nach unten expandiert werden, denn dort befinden sich die Stack-Daten.

Deshalb ist bei der Realisierung der obigen Routinen folgendes zu beachten:

1. Ändern eines bereits existierenden Namens

- ▶ Ist die Länge des neuen Strings (*wert*) kleiner oder gleich der Länge des existierenden Strings (*wert*), kann der neue String einfach über den alten String kopiert werden.
- ▶ Ist jedoch der neue String (*wert*) länger als der alte String (*wert*), dann muß zuerst mit ***malloc*** neuer Speicherplatz allokiert, der neue String dorthin kopiert, und schließlich in der Environment-Liste der alte Zeiger für *name* auf die Adresse des neu allokierten Speichers gesetzt werden.

2. Hinzufügen eines neuen Namens

Hier muß zuerst ***malloc*** aufgerufen werden, um neuen Speicherplatz für den Eintrag »name=wert« zu allokieren, bevor dieser String in diesen neuen Speicherplatz kopiert wird. Danach sind die beiden folgenden Fälle zu unterscheiden:

- ▶ Fügt man das erste Mal einen neuen Namen hinzu, so muß zunächst mit ***malloc*** Speicherplatz für eine neue Environment-Liste (mit einem Eintrag mehr) allokiert werden. In diesen neuen Speicherplatz werden dann alle Zeiger der alten Environment-Liste kopiert, wobei am Ende der neue Zeiger auf den String »name=wert« angehängt wird. Die ganze Liste muß natürlich wieder mit einem NULL-Zeiger abgeschlossen werden. Schließlich muß der globalen Variablen *environ* die Anfangsadresse dieser neuen Liste zugewiesen werden. Dies bedeutet, daß sich jetzt alle Zeiger der Environment-Liste im Heap befinden, während sich die alten Einträge der Form »name=wert« weiterhin am Anfang des Adreßraums (oberhalb des Stacks) befinden.

- Fügt man nicht das erste Mal einen neuen Namen hinzu, dann muß man den Speicherplatz für die Environment-Liste, den man beim ersten Hinzufügen (durch **malloc** auf dem Heap) allokiert hat, nur mit **realloc** vergrößern, um am Ende den neuen Zeiger mit abschließendem NULL-Zeiger anzuhängen.

3. Löschen eines Namens

Hier muß zuerst in der Environment-Liste der entsprechende Zeiger gefunden werden, und dann müssen alle darauffolgenden Zeiger um eine Stelle nach vorne in der Environment-Liste verschoben werden. Hierbei sollte man nicht vergessen, das neue Ende der Environment-Liste durch einen NULL-Zeiger zu kennzeichnen.

9.8.5 Automatisches Erstellen von Bundesliga-Tabellen

Erstellen Sie ein Programm `bundliga.c`, das von der Datei `tabelle.neu` die Tabelle des letzten Spieltags der Fußball-Bundesliga und von der Datei `ergebnis` die Ergebnisse des neuen Spieltags liest, bevor es daraus die neue Tabelle erstellt. Die alte Tabelle (aus `tabelle.neu`) soll das Programm `bundliga.c` an das Ende der Datei `tabelle.alt` anhängen, bevor es die neue Tabelle in die Datei `tabelle.neu` schreibt; die alte Tabelle in `tabelle.neu` wird dadurch überschrieben. Bei dieser Vorgehensweise muß bei jedem neuen Spieltag nur die Datei `ergebnis` neu erstellt werden. Die alten Tabellen werden in der Datei `tabelle.alt` aufgehoben.

Verwenden Sie bei diesem Programm die Funktion **atexit**, um dem Benutzer den Erfolg oder Mißerfolg bei der Tabellenerstellung mitzuteilen.

Wenn z.B. die folgenden Dateien vorliegen:

Datei `tabelle.neu`:

Bayern Muenchen	37-13 45:19
1.FC Koeln	34-16 43:19
Werder Bremen	32-18 38:25
Hamburger SV	31-17 43:24
Moenchengladbach	28-20 31:29
Borussia Dortmund	26-24 42:28
VfB Stuttgart	26-24 40:37
Karlsruher SC	26-24 37:34
1.FC Kaiserslautern	25-25 35:31
Bayer Uerdingen	25-25 34:33
FC St.Pauli	25-25 27:27
Bayer Leverkusen	24-26 31:33
VfL Bochum	24-26 30:33
1.FC Nuernberg	20-30 27:40
Waldhof Mannheim	18-32 26:44
Eintracht Frankfurt	17-33 16:38
Stuttgarter Kickers	17-33 29:54
Hannover 96	13-37 21:47

Datei ergebnis

Hannover 96-Waldhof Mannheim| 0 : 2
 1.FC Kaiserslautern-Bayer Uerdingen| 2 : 0
 VfB Stuttgart-VfL Bochum| 3 : 1
 Hamburger SV-1.FC Nuernberg| 3 : 2
 Bayer Leverkusen-1.FC Koeln| 0 : 0
 Moenchengladbach-FC St.Pauli| 2 : 2
 Werder Bremen-Stuttgarter Kickers| 4 : 0
 Borussia Dortmund-Bayern Muenchen| 1 : 1
 Karlsruher SC-Eintracht Frankfurt| 1 : 3

dann sollte das Programm bundliga folgendes ausgeben:

	Punkte	Tore
1. (1) Bayern Muenchen	38:14	46-20
2. (2) 1.FC Koeln	35:17	43-19
3. (3) Werder Bremen	34:18	42-25
4. (4) Hamburger SV	33:17	46-26
5. (5) Moenchengladbach	29:21	33-31
6. (7) VfB Stuttgart	28:24	43-38
7. (6) Borussia Dortmund	27:25	43-29
8. (9) 1.FC Kaiserslautern	27:25	37-31
9. (8) Karlsruher SC	26:26	38-37
10. (11) FC St.Pauli	26:26	29-29
11. (10) Bayer Uerdingen	25:27	34-35
12. (12) Bayer Leverkusen	25:27	31-33
13. (13) VfL Bochum	24:28	31-36
14. (14) 1.FC Nuernberg	20:32	29-43
15. (15) Waldhof Mannheim	20:32	28-44
16. (16) Eintracht Frankfurt	19:33	19-39
17. (17) Stuttgarter Kickers	17:35	29-58
18. (18) Hannover 96	13:39	21-49

Neuester Tabellenstand in 'tabelle.neu'

In 'tabelle.alt' wurde der Inhalt von 'tabelle.neu' angehaengt

Auf Wiedersehen, lieber Fussballfan

Die beiden Dateien tabelle.neu und tabelle.alt sollten nach diesem Ablauf die folgenden Inhalte haben:

Datei tabelle.neu:

Bayern Muenchen| 38-14 46:20
 1.FC Koeln| 35-17 43:19
 Werder Bremen| 34-18 42:25
 Hamburger SV| 33-17 46:26
 Moenchengladbach| 29-21 33:31
 VfB Stuttgart| 28-24 43:38
 Borussia Dortmund| 27-25 43:29
 1.FC Kaiserslautern| 27-25 37:31
 Karlsruher SC| 26-26 38:37
 FC St.Pauli| 26-26 29:29

Bayer Uerdingen	25-27 34:35
Bayer Leverkusen	25-27 31:33
VfL Bochum	24-28 31:36
1.FC Nuernberg	20-32 29:43
Waldhof Mannheim	20-32 28:44
Eintracht Frankfurt	19-33 19:39
Stuttgarter Kickers	17-35 29:58
Hannover 96	13-39 21:49

Datei *tabelle.alt*:

```

:
:      [Alte Tabellen]
:
:
-----

```

Bayern Muenchen	37-13 45:19
1.FC Koeln	34-16 43:19
Werder Bremen	32-18 38:25
Hamburger SV	31-17 43:24
Moenchengladbach	28-20 31:29
Borussia Dortmund	26-24 42:28
VfB Stuttgart	26-24 40:37
Karlsruher SC	26-24 37:34
1.FC Kaiserslautern	25-25 35:31
Bayer Uerdingen	25-25 34:33
FC St.Pauli	25-25 27:27
Bayer Leverkusen	24-26 31:33
VfL Bochum	24-26 30:33
1.FC Nuernberg	20-30 27:40
Waldhof Mannheim	18-32 26:44
Eintracht Frankfurt	17-33 16:38
Stuttgarter Kickers	17-33 29:54
Hannover 96	13-37 21:47

10 Die Prozeßsteuerung

*Gewöhnlich zerstreut der Sohn,
was der Vater gesammelt hat,
sammelt etwas anderes oder auf andere Weise.*

Goethe

In diesem Kapitel wird zunächst der die Unix-Prozeßhierarchie beschrieben, bevor dann auf die Kreierung von neuen Prozessen eingegangen wird. Ausführlich beschäftigt sich dieses Kapitel auch mit dem Warten auf die Beendigung von Prozessen und dem Überlagern von Prozessen mit anderen Programmen, bevor es die unterschiedlichen Möglichkeiten zum Ändern der User-IDs und Group-IDs vorstellt. Zum Abschluß stellt dieses Kapitel weitere Informationen vor, die über einen Prozeß erfragt werden können.

10.1 Prozeßkennungen und die Unix-Prozeßhierarchie

10.1.1 Prozeß-IDs

Jedem Prozeß wird in Unix eine eindeutige Kennung in Form einer nicht negativen ganzen Zahl zugewiesen: die sogenannte *Prozeßnummer* oder *Prozeß-ID* (*process identification*). Meist verwendet man die Abkürzung *PID*. Der Kern stellt sicher, daß niemals zwei oder mehrere gleichzeitig ablaufende Prozesse die gleiche PID besitzen.

Da bis auf einige wenige Ausnahmen jeder Prozeß einen Elternprozeß hat, der seinen Start veranlaßt hat, existiert für jeden Prozeß neben der PID noch eine *parent process ID* (*PPID*), die die Prozeßnummer des Elternprozesses ist.

Neben diesen Prozeßnummern sind jedem Prozeß weitere Kennungen zugeteilt: die *reale* und *effektive User-ID* und die *Group-ID*. Die Bedeutung dieser Begriffe ist in Kapitel 5.3 beschrieben.

10.1.2 `getpid` und `getppid` – Erfragen der PID und PPID

Um die PID und die PPID zu erfragen, stehen die beiden Funktionen *getpid* und *getppid* zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
gibt zurück: PID des aufrufenden Prozesses

pid_t getppid(void);
gibt zurück: PPID des aufrufenden Prozesses
```

10.1.3 **getuid** und **geteuid** – Erfragen der realen und effektiven User-ID

Um die reale und effektive User-ID zu erfragen, stehen die beiden Funktionen **getuid** und **geteuid** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getuid(void);
gibt zurück: reale User-ID des aufrufenden Prozesses

pid_t geteuid(void);
gibt zurück: effektive User-ID des aufrufenden Prozesses
```

Die Bedeutung der realen und effektiven User-ID ist in Kapitel 5.3 beschrieben.

10.1.4 **getgid** und **getegid** – Erfragen der realen und effektiven Group-ID

Um die reale und effektive Group-ID zu erfragen, stehen die beiden Funktionen **getgid** und **getegid** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getgid(void);
gibt zurück: reale Group-ID des aufrufenden Prozesses

pid_t getegid(void);
gibt zurück: effektive Group-ID des aufrufenden Prozesses
```

Die Bedeutung der realen und effektiven Group-ID ist in Kapitel 5.3 beschrieben.

Beispiel

Ausgeben seiner PID, PPID, UID, EUID, GID und EGID durch einen Prozeß

```
#include <sys/types.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    printf("    PID/PPID: %d/%d\n", getpid(), getppid());
    printf("    UID/EUID: %d/%d\n", getuid(), geteuid());
    printf("    GID/EGID: %d/%d\n", getgid(), getegid());

    exit(0);
}
```

Programm 10.1 (*procsid.c*): Ausgabe von PID, PPID, UID, EUID, GID und EGID

Nachdem man dieses Programm 10.1 (*procsid.c*) kompiliert und gelinkt hat

```
cc -o procsid procsid.c fehler.c
```

ergeben sich beim Start z.B. folgende Abläufe:

```
$ procsid
    PID/PPID: 166/58
    UID/EUID: 2021/2021
    GID/EGID: 25/25
$ sh      [Starten einer Subshell]
$ procsid
    PID/PPID: 170/167
    UID/EUID: 2021/2021
    GID/EGID: 25/25
$ exit    [Verlassen der Subshell]
$
```

10.1.5 Unix-Prozeßhierarchie

Beim Start eines Unix-Systems werden üblicherweise einige spezielle Prozesse eingerichtet.

Scheduler-Prozeß mit PID 0

Dieser Systemprozeß ist Teil des Kerns und erhält normalerweise die PID 0. Er wird oft auch mit *swapper* bezeichnet.

init-Prozeß mit PID 1

Gewöhnlich wird dieser Prozeß nach dem Booten vom Kern kreiert und erhält die PID 1. Die zu diesem Prozeß gehörige Programmdatei befindet sich entweder in */etc/init* (ältere Unix-Systeme) oder in */sbin/init* (neuere Unix-Systeme). Der *init*-Prozeß ist für die systemspezifische Initialisierung zuständig, indem er die Dateien */etc/rc** liest, und das System beim Start entsprechend den dort gemachten Vorgaben konfiguriert.

Genau wie der Scheduler-Prozeß wird der *init*-Prozeß niemals beendet. Obwohl der *init*-Prozeß mit Superuser-Rechten läuft, ist er doch – anders als der Scheduler-Prozeß – ein Benutzerprozeß und kein Systemprozeß im Kern.

pagedaemon mit PID 2 (auf manchen Systemen)

Auf manchen Systemen, die mit virtuellen Speichern arbeiten, wird dieser spezielle Prozeß kreiert. Er ist dabei für das Paging im virtuellen Speicher zuständig. Genau wie der *swapper* ist der *pagedaemon* ein Systemprozeß im Kern.

10.2 Kreieren von neuen Prozessen

Zum Kreieren von neuen Prozessen werden die beiden Funktionen ***fork*** und ***vfork*** angeboten.

10.2.1 *fork* – Kreieren eines neuen Prozesses

Um durch den Unixkern einen neuen Prozeß kreieren zu lassen, steht die Funktion ***fork*** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
        gibt zurück: 0 im Kindprozeß; Prozeß-ID des Kindprozesses im Elternprozeß; -1 bei Fehler
```

Den Aufrufer von ***fork*** nennt man *Elternprozeß* (*parent process*) und den durch ***fork*** neu kreierten Prozeß nennt man *Kindprozeß* (*child process*). Unmittelbar nach der Durchführung von ***fork*** sind die beiden Prozesse einander sehr ähnlich. So haben sie z.B. die gleichen offenen Dateien, dieselbe User-ID, dasselbe Working-Directory usw. (siehe auch weiter unten).

Das Besondere an der Funktion ***fork*** ist, daß sie nur einmal aufgerufen wird, jedoch zweimal zurückkehrt:

- ▶ Die Rückkehr zum Kindprozeß zeigt sie durch den Rückgabewert 0 an.
- ▶ Die Rückkehr zum Elternprozeß zeigt sie durch die Rückgabe der Prozeß-ID des neuen Kindprozesses an.

Ein typisches Codestück für die Kreierung eines neuen Prozesses sieht deshalb z.B. oft wie folgt aus:

```
switch ( rueckgabe=fork() ) {
    case -1:
        /* Fehlermeldung, daß fork-Aufruf nicht erfolgreich war */
        break;
```

```
case 0:
    /* Code fuer den Kindprozeß */
    break;

default:
    /* Code fuer den Elternprozeß */
    break;
}
```

oder auch wie:

```
if ( (rueckgabe=fork()) == 0 ) {
    /* Code fuer den Kindprozeß */

} else if (rueckgabe > 0) {
    /* Code fuer den Elternprozeß */

} else {
    /* Fehlermeldung, daß fork-Aufruf nicht erfolgreich war */

}
```

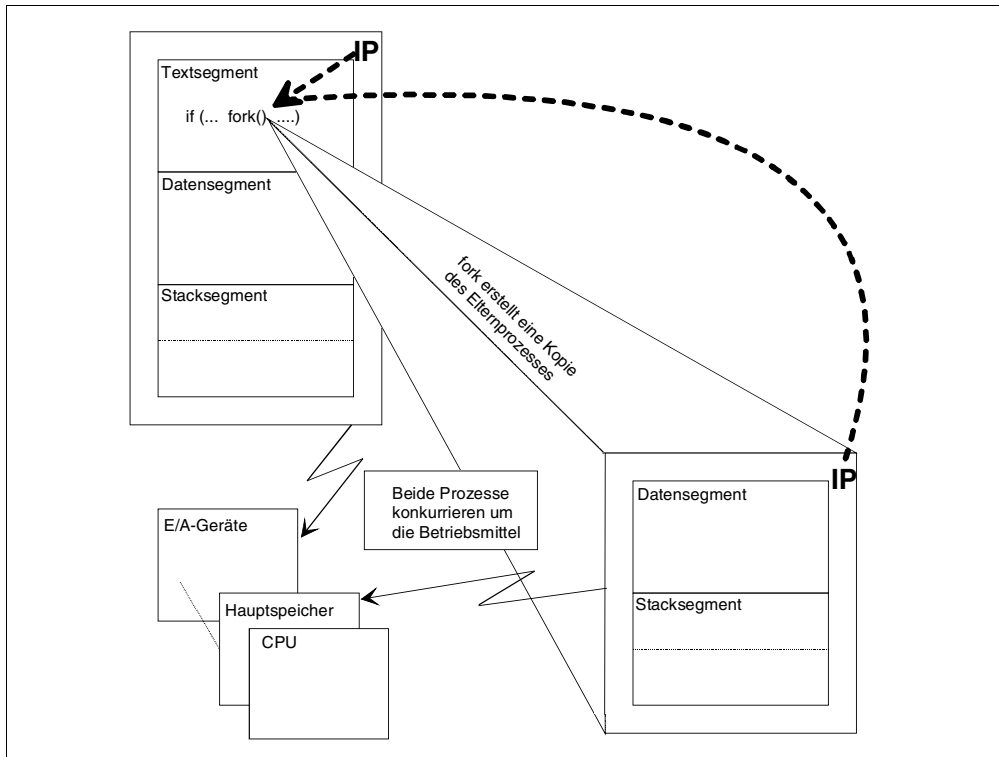
Der Grund für diese Rückgabewert-Regelung ist, daß ein Elternprozeß mehrere Kindprozesse, aber jeder Kindprozeß nur einen Elternprozeß haben kann.

Dies bringt es mit sich, daß es für einen Elternprozeß keine andere Möglichkeit gibt, die Prozeß-IDs seiner Kinder zu erfahren, außer zum Zeitpunkt ihrer Kreierung beim **fork**-Aufruf. Will dagegen ein Kindprozeß die Prozeß-ID seines Elternprozesses erfahren, muß er nur die Funktion **getppid** aufrufen.

Ein **fork**-Aufruf bewirkt, daß für den neuen Kindprozeß eine Kopie des Elternprozesses erstellt wird. Während dabei meist das Datensegment, Stacksegment und der Heap des Elternprozesses wirklich kopiert wird, wird das Textsegment, wenn es nur lesbar ist, meist nicht kopiert, sondern von beiden gleichzeitig benutzt (*shared text segment*). Beide Prozesse fahren dann mit der Ausführung nach dem **fork**-Aufruf fort, arbeiten nun aber mit unterschiedlichen *Instruction Pointer*.

Abbildung 10.1 stellt die Wirkung des **fork**-Aufrufs anschaulich dar.

Viele Unix-Implementierungen machen jedoch nicht immer eine Kopie vom Datensegment, Stacksegment und Heap des Elternprozesses, denn oft ist eine solches zeitaufwendiges Kopieren überflüssig, da sich in vielen Anwendungsfällen der Kindprozeß unmittelbar nach seiner Kreierung durch einen **exec**-Aufruf (siehe Kapitel 10.5) mit dem Code und den Daten eines anderen Programms versorgt.

Abbildung 10.1: Kreieren eines Kindprozesses mit **fork**

Deswegen wenden viele Unix-Implementierungen bei **fork** das COW-Verfahren (*copy-on-write*) an. Bei diesem Verfahren wird für den Kindprozeß zunächst keine Kopie des Datensegments, Stacksegments und Heaps erstellt, sondern die Originale des Elternprozesses, die der Kern als nur-lesbar (*read-only*) einstuft, werden auch zugleich vom Kindprozeß genutzt. Erst wenn einer der beiden Prozesse versucht, in einem der entsprechenden Speicherbereiche zu schreiben, erzeugt der Kern nur für diesen Speicherbereich wirklich eine Kopie.

Beispiel

Demonstrationsprogramm zur Funktion `fork`

```
#include <sys/types.h>
#include "eighdr.h"

int global_var=100;

int
main(void)
{
    int    lokal_var;
```

```

pid_t  pid;

lokal_var=1;

printf("---vor fork-Aufruf---\n");

switch ( pid=fork() ) {
    case -1:
        fehler_meld(FATAL_SYS, "Fehler bei fork");
        break;

    case 0:
        lokal_var++;
        global_var++;
        printf(".....Ich bin der Kindprozess.....\n");
        break;

    default:
        printf(".....Ich bin der Elternprozess.....\n");
        break;
}
printf("%s: global_var=%d, lokal_var=%d\n",
        (pid==0) ? "Kindprozess" : "Elternprozess", global_var, lokal_var);

exit(0);
}

```

Programm 10.2 (forkdemo.c): Demonstrationsbeispiel zu `fork`

Nachdem man dieses Programm 10.2 (forkdemo.c) kompiliert und gelinkt hat

```
cc -o forkdemo forkdemo.c fehler.c
```

ergeben sich beim Start z.B. folgende Abläufe:

```

$ forkdemo
---vor fork-Aufruf---
.....Ich bin der Elternprozess.....
Elternprozess: global_var=100, lokal_var=1
.....Ich bin der Kindprozess.....
Kindprozess: global_var=101, lokal_var=2

$ forkdemo >temp
$ cat temp
---vor fork-Aufruf---
.....Ich bin der Elternprozess.....
Elternprozess: global_var=100, lokal_var=1
---vor fork-Aufruf---
.....Ich bin der Kindprozess.....
Kindprozess: global_var=101, lokal_var=2
$ rm temp
$

```

Beim ersten Aufruf von `forkdemo` wird der Text »---vor fork-Aufruf---« nur einmal ausgegeben, nämlich vom Elternprozeß. Beim zweiten Aufruf dagegen wird dieser Text zweimal, nämlich sowohl vom Eltern- als auch vom Kindprozeß, ausgegeben.

Dies ist darin begründet, daß die Funktion **`printf`** intern mit einem Puffer arbeitet (siehe Kapitel 3.5). Dabei gelten die folgenden Regeln:

- ▶ Ist die Standardausgabe (`stdout`) auf ein Terminal eingestellt, so findet eine Zeilenpufferung statt. Beim ersten Aufruf liegt diese Konstellation vor.
- ▶ Ist die Standardausgabe (`stdout`) nicht auf ein Terminal eingestellt (wie beim zweiten Aufruf), so findet eine Vollpufferung statt.

Da beim ersten Aufruf für **`printf`** eine Zeilenpufferung stattfindet, wird – bedingt durch `\n` – der Puffer sofort geleert, was zur Ausgabe des Textes »---vor fork-Aufruf---« führt. Beim darauffolgenden **`fork`**-Aufruf wird somit nur ein leerer Ausgabe-Puffer an den Kindprozeß weitergereicht.

Da beim zweiten Aufruf für **`printf`** eine Vollpufferung stattfindet, wird der Text »---vor fork-Aufruf---« nicht sofort ausgegeben, sondern verbleibt im Ausgabe-Puffer. Beim darauffolgenden **`fork`**-Aufruf wird somit – bedingt durch das Kopieren des Datenssegments – auch dieser nicht geleerte Ausgabe-Puffer an den Kindprozeß weitergereicht. Beide Prozesse (Eltern- und Kindprozeß) haben somit nun den gleichen Text in ihrem Ausgabe-Puffer stehen. Die folgenden **`printf`** hängen dann weiteren Text an ihren jeweiligen Ausgabe-Puffer an. Wenn sich die beiden Prozesse (mit `exit(0)`) beenden, wird der jeweilige Ausgabe-Puffer wirklich physikalisch auf die Standardausgabe (hier in die Datei `temp` umgelenkt) geschrieben.

Hinweis

Für einen nicht erfolgreichen **`fork`**-Aufruf gibt es zwei Gründe:

1. Es existieren bereits zu viele Prozesse.
2. Das obere Limit von Prozessen (`CHILD_MAX` aus `<limits.h>`) ist für die reelle User-ID bereits ausgeschöpft.

Soll bei der Ausgabe in keinem Fall eine Pufferung stattfinden, so sollte man die entsprechenden Daten mit **`write`** ausgeben, wie z.B.

```
write(STDOUT_FILENO, textarray, strlen(textarray));
```

Es ist nicht festgelegt, in welcher Reihenfolge die beiden Prozesse (Eltern- und Kindprozeß) zur Ausführung kommen. Dies hängt vom Algorithmus ab, den der Scheduler des Kerns verwendet. In den Kapiteln 10.4 und 13 werden wir Mechanismen kennenlernen, die die Synchronisation von Prozessen ermöglichen. Das Programm 10.3 (`mehrproz.c`) demonstriert die Abhängigkeit der Reihenfolge.

```

#include <sys/types.h>
#include "eighdr.h"

int
main(void)
{
    int    var=0;

    if (fork() == -1) {
        fehler_meld(FATAL_SYS, "Fehler bei ersten fork");
    } else {
        var++;
        printf("var = %d\n", var); /* ..... */

        if (fork() == -1) { /* Kind- und Elternprozess erzeugen neues Kind */
            fehler_meld(FATAL_SYS, "Fehler bei zweiten fork");
        } else {
            var++;
            printf("var = %d\n", var); /* Kindern und dessen Enkel ausgeführt */
        }
    }

    exit(0);
}

```

Programm 10.3 (mehrproz.c): Vom Scheduler abhängige Ausführungsreihenfolge

Nachdem man dieses Programm 10.3 (mehrproz.c) kompiliert und gelinkt hat

```
cc -o mehrproz mehrproz.c fehler.c
```

ergeben sich beim Start z.B. folgende Abläufe:

```

$ mehrproz
var = 1
var = 2
var = 1
var = 2
var = 2
var = 2
$ mehrproz >temp [Hier wirkt sich wieder die Vollpufferung von printf aus]
$ cat temp
var = 1
var = 2
var = 1
var = 2
var = 1
var = 2
var = 1
var = 2
$ rm temp
$

```

Abbildung 10.2 erläutert den Ablauf des Programms 10.3 (mehrproz.c).

Jede Ausgabe von `var` wird in Abbildung 10.2 durch einen dick umrandeten Kasten angezeigt, und es ist leicht zu erkennen, daß zweimal eine 1 und viermal eine 2 ausgegeben wird. Aus dieser Abbildung läßt sich jedoch nicht erschließen, in welcher Reihenfolge die einzelnen Werte von `var` ausgegeben werden, da dies immer vom Scheduler-Algorithmus abhängig ist.

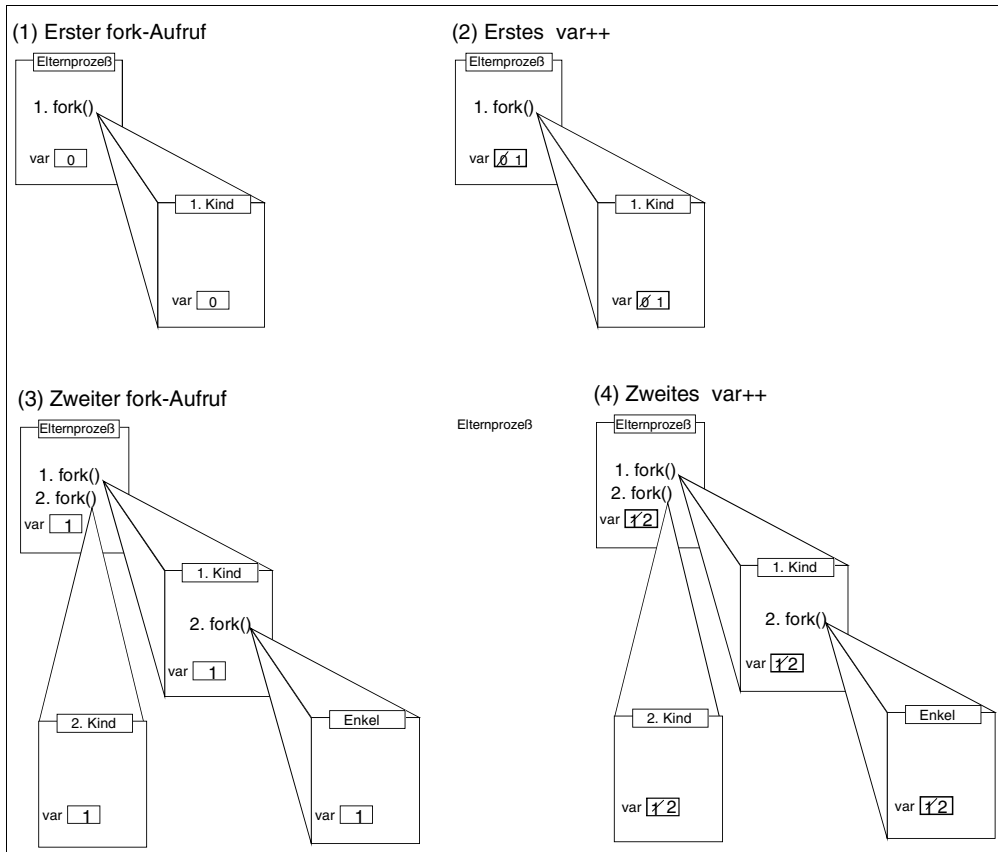


Abbildung 10.2: Erklärung zum Programm 10.3 (*mehrproz.c*)

Es ist zu beachten, daß bei aufeinanderfolgenden **fork**-Aufrufen die auf den ersten **fork** folgenden **fork**-Aufrufe auch bereits von den Kindprozessen ausgeführt werden. So ergeben sich sehr schnell neben Eltern- und Kindprozessen auch Enkel- und Urenkel-Prozesse usw. So führen z.B. im folgenden Programm 10.4 (*mehrfork.c*) die vier aufeinanderfolgenden **fork**-Aufrufe zur Kreierung von 15 Prozessen.

```
#include <sys/types.h>
#include "eighdr.h"
```



```
int
main(void)
{
    char pid[MAX_ZEICHEN];

    fork();
    fork();
    fork();
    fork();
    sprintf(pid, "PID = %d\n", getpid());
    write(STDOUT_FILENO, pid, strlen(pid)); /* Bei write keine Pufferung */

    exit(0);
}
```

Programm 10.4 (**mehrfork.c**): 15 neue Prozesse mit nur vier **fork**-Aufrufen

Nachdem man dieses Programm 10.4 (**mehrfork.c**) kompiliert und gelinkt hat

```
cc -o mehrfork mehrfork.c fehler.c
```

ergibt sich beim Start z.B. folgender Ablauf:

```
$ mehrfork
PID = 441
PID = 442
PID = 443
PID = 444
PID = 445
PID = 447
PID = 448
PID = 450
PID = 451
PID = 453
PID = 454
PID = 446
PID = 449
PID = 452
PID = 455
PID = 456
$
```

Versuchen Sie, die hierbei entstandene Prozeßhierarchie selbst nachzuvollziehen!

10.2.2 Unterschiede zwischen Eltern- und Kindprozeß

In den folgenden Punkten unterscheiden sich Eltern- und Kindprozeß:

- ▶ Rückgabewert von **fork** (0 bei Kind, PID des Kindprozesses bei Elternprozeß)
- ▶ unterschiedliche PIDs (Prozeß-IDs)
- ▶ unterschiedliche PPIDs (Parent Prozeß-IDs)

- ▶ Beim Kindprozeß werden `tms_utime`, `tms_stime`, `tms_cutime` und `tms_ustime` auf 0 gesetzt.
- ▶ Dateisperren des Elternprozesses werden nicht an den Kindprozeß vererbt.
- ▶ Eingeschaltete Zeitschaltuhren des Elternprozesses (mittels **alarm**) werden beim Kindprozeß ausgeschaltet.
- ▶ Hängende (noch nicht zugestellte) Signale des Elternprozesses werden nicht an den Kindprozeß vererbt.

Viele der obigen Punkte sind bisher noch nicht behandelt, sondern werden erst in späteren Kapiteln vorgestellt. Der Vollständigkeit halber wurden sie aber hier mit in die Liste der Unterschiede zwischen Eltern- und Kindprozeß aufgenommen.

10.2.3 Vererbungen eines Elternprozesses an seinen Kindprozeß

Wenn ein Elternprozeß mit **fork** einen neuen Kindprozeß generiert, so erbt der Kindprozeß alle offenen Filedeskriptoren des Elternprozesses. Das Vererben entspricht hierbei einem Duplizieren der Filedeskriptoren (mit **dup**), so daß Eltern- und Kindprozeß für jeden offenen Filedeskriptor den gleichen Dateitabelleneintrag benutzen. Wenn z.B. der Elternprozeß zum Zeitpunkt des **fork**-Aufrufs seine Standardausgabe umgeleitet hat, so gilt diese Umleitung auch für den Kindprozeß.

Abbildung 10.3 zeigt die intern vorliegenden Strukturen nach einem **fork**-Aufruf eines Prozesses, der neben der Standardeingabe, Standardausgabe und Standardfehlerausgabe eine weitere Datei geöffnet hat.

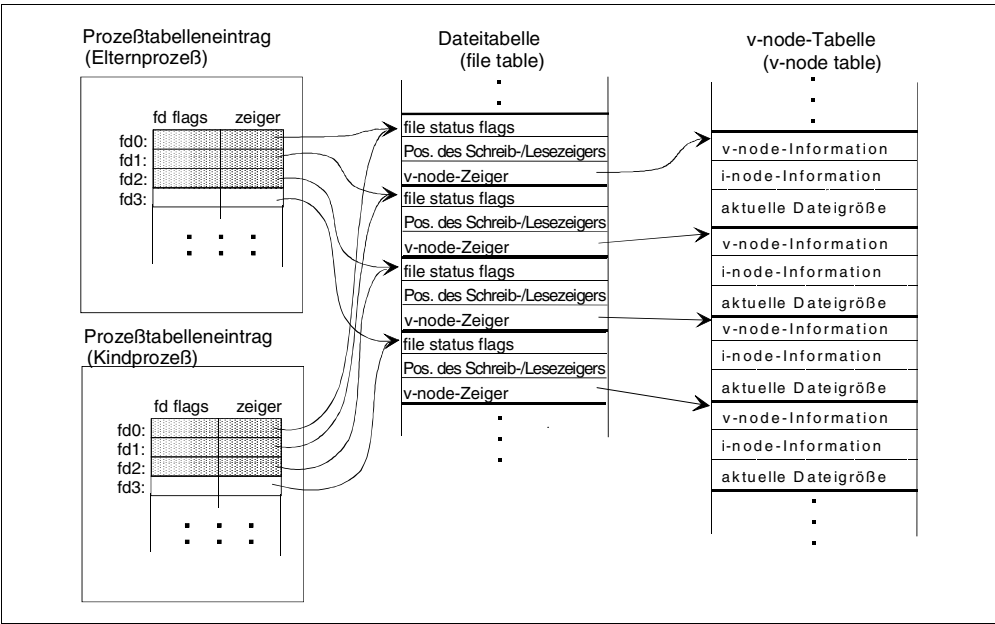


Abbildung 10.3: Vererbung der offenen Filedeskriptoren bei einem **fork**-Aufruf

In Abbildung 10.3 ist zu erkennen, daß Eltern- und Kindprozeß den gleichen Schreib-/Lesezeiger benutzen. Dies bedeutet, daß ein Schreiben bzw. Lesen des Kindprozesses eine neue Positionierung des Schreib-/Lesezeigers nach sich zieht, die dann auch für den Elternprozeß gültig ist. Dies ist wichtig, wenn z.B. ein Eltern- und Kindprozeß in die gleiche Datei schreiben. In diesem Fall ist es sicherlich erwünscht, daß ein Elternprozeß, der auf die Beendigung eines Kindprozesses wartet, nach dessen Beendigung am neuen Ende der Datei, in die der Kindprozeß geschrieben hat, weiterschreibt; andernfalls würde er die vom Kind geschriebenen Daten überschreiben.

Wenn Eltern- und Kindprozeß den gleichen Filedeskriptor zum Schreiben benutzen, dann werden die Ausgaben der beiden Prozesse vermischt in die entsprechende Datei geschrieben, wenn keinerlei Synchronisation zwischen Eltern- und Kindprozeß (wie z.B. ein Warten des Elternprozesses auf Beendigung des Kindprozesses mit **wait**) stattfindet.

Beispiel

Gleichzeitiges Schreiben von Eltern- und Kindprozeß in gleiche Datei (mit und ohne Synchronisation)

```
#include <sys/types.h>
#include "eighdr.h"

int
main(void)
{
    FILE      *dz;
    int        i, status;
    long int   j;

    /*--- Schreiben von Kind- und Elternprozess in gleiche Datei ---*/
    /*   (mit wait vom Elternprozess)                                */

    if ( (dz=fopen("datei1.txt", "w")) == NULL)
        fehler_meld(FATAL_SYS, "kann Datei datei1.txt nicht oeffnen");

    switch ( fork() ) {
        case -1:
            fehler_meld(FATAL_SYS, "Fehler bei fork");
            break;
        case 0:
            for (i=1; i<=10; i++) {
                for (j=1; j<=100000; j++) /*-- Warteschleife ----*/
                    ;
                fprintf(dz, "%2d: Kindprozess schreibt\n", i);
                fflush(NULL);
            }
            exit(0);
        default:
            wait(&status); /*-- Auf Beendigung des Kindes warten ---*/
            for (i=1; i<=10; i++) {
                for (j=1; j<=100000; j++) /*-- Warteschleife ----*/
```

```

        ;
        fprintf(dz, "%2d: Elternprozess schreibt\n", i);
        fflush(NULL);
    }
    fclose(dz);
    break;
}

/*--- Schreiben von Kind- und Elternprozess in gleiche Datei ---*/
/*   (ohne wait vom Elternprozess)                               */

if ( (dz=fopen("datei2.txt", "w")) == NULL)
    fehler_meld(FATAL_SYS, "kann Datei datei2.txt nicht oeffnen");

switch ( fork() ) {
    case -1:
        fehler_meld(FATAL_SYS, "Fehler bei fork");
        break;
    case 0:
        for (i=1; i<=10; i++) {
            for (j=1; j<=100000; j++) /*-- Warteschleife ----*/
                ;
            fprintf(dz, "%2d: Kindprozess schreibt\n", i);
            fflush(NULL);
        }
        exit(0);
    default:
        for (i=1; i<=10; i++) {
            for (j=1; j<=100000; j++) /*-- Warteschleife ----*/
                ;
            fprintf(dz, "%2d: Elternprozess schreibt\n", i);
            fflush(NULL);
        }
        break;
}

exit(0);
}

```

Programm 10.5 (forkerb.c): Schreiben von Eltern- und Kindprozeß in gleiche Datei

Nachdem man dieses Programm 10.5 (forkerb.c) kompiliert und gelinkt hat

```
cc -o forkerb forkerb.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ forkerb
$ cat datei1.txt
1: Kindprozeß schreibt
2: Kindprozeß schreibt
3: Kindprozeß schreibt
4: Kindprozeß schreibt
5: Kindprozeß schreibt

```

```
6: Kindprozeß schreibt
7: Kindprozeß schreibt
8: Kindprozeß schreibt
9: Kindprozeß schreibt
10: Kindprozeß schreibt
1: Elternprozeß schreibt
2: Elternprozeß schreibt
3: Elternprozeß schreibt
4: Elternprozeß schreibt
5: Elternprozeß schreibt
6: Elternprozeß schreibt
7: Elternprozeß schreibt
8: Elternprozeß schreibt
9: Elternprozeß schreibt
10: Elternprozeß schreibt
$ cat datei2.txt
1: Elternprozeß schreibt
2: Elternprozeß schreibt
3: Elternprozeß schreibt
4: Elternprozeß schreibt
5: Elternprozeß schreibt
6: Elternprozeß schreibt
1: Kindprozeß schreibt
2: Kindprozeß schreibt
3: Kindprozeß schreibt
4: Kindprozeß schreibt
5: Kindprozeß schreibt
7: Elternprozeß schreibt
8: Elternprozeß schreibt
9: Elternprozeß schreibt
10: Elternprozeß schreibt
6: Kindprozeß schreibt
7: Kindprozeß schreibt
8: Kindprozeß schreibt
9: Kindprozeß schreibt
10: Kindprozeß schreibt
$
```

Nach einem **fork**-Aufruf bestehen grundsätzlich zwei Möglichkeiten der Handhabung von noch offenen Filedeskriptoren:

1. Wenn der Elternprozeß auf die Beendigung des Kindprozesses wartet, so sind keinerlei besondere Vorkehrungen zu treffen, da die Positionen der Schreib-/Lesezeiger mit Beendigung des Kindprozesses entsprechend der Lese-/Schreibaktionen des Kindprozesses automatisch gesetzt sind.
2. Wenn der Elternprozeß nicht auf die Beendigung des Kindprozesses wartet, so sollten nach dem **fork**-Aufruf sowohl der Eltern- als auch der Kindprozeß die jeweils nicht benötigten Filedeskriptoren schließen. So ist z.B. bei Netzwerk-Servern, wo dieser Fall häufiger auftritt, sichergestellt, daß sich die beiden Prozesse beim Lesen und Schreiben nicht in die Quere kommen.

Neben den offenen Filedeskriptoren erbt ein Kindprozeß weitere Eigenschaften von seinem Elternprozeß:

- ▶ IDs (reale und effektive User-ID und Group-ID, Zusatz-Group-IDs, Prozeßgruppen-ID, Session-ID, Set-User-ID, Set-Group-ID)
- ▶ Working-Directory und Root-Directory
- ▶ Dateikreierungsmaske
- ▶ *close-on-exec*-Flag für offene Filedeskriptoren
- ▶ Signalmaske und Signalhandler
- ▶ Kontrollterminal
- ▶ Environment
- ▶ Ressourcen-Limits
- ▶ Angebundene *Shared-Memory*-Segmente

Einige der obigen Punkte wurden bisher noch nicht behandelt, sondern werden erst in späteren Kapiteln vorgestellt. Der Vollständigkeit halber wurden sie aber hier in die Liste der Eigenschaften, die ein Prozeß vererbt, aufgenommen.

10.2.4 Typische Anwendungen für *fork*

fork wendet man hauptsächlich in den beiden folgenden Fällen an:

1. Ein Programm soll zu einem Zeitpunkt gleichzeitig zwei verschiedene Codestücke ausführen. Dieser Anwendungsfall liegt zum Beispiel bei einem Netzwerk-Server vor, wenn dieser auf eine Anforderung durch einen Client wartet. Bei Eintreffen einer Anforderung, wird mit *fork* ein Kindprozeß generiert, der die Client-Anforderung bedient, während der Elternprozeß sich wieder in den Wartezustand begibt, um auf das Eintreffen neuer Anforderungen zu warten.
2. Ein Prozeß (wie eine Shell) möchte ein anderes Programm ausführen. Bei diesem Anwendungsfall ruft der Kindprozeß unmittelbar nach der Rückkehr aus *fork* die Funktion *exec* (siehe Kapitel 10.5) auf.

10.2.5 *vfork* – Kreieren eines Prozesses mit anschließendem *exec*-Aufruf

Wenn man einen Kindprozeß kreiert, um in ihm sofort anschließend ein anderes Programm mit *exec* zu starten, dann ist es überflüssig, den Adreßraum (Datensegment, Stacksegment, Heap) des Elternprozesses zu kopieren, da der Kindprozeß diesen sowieso nie benutzen wird, wenn er sofort anschließend *exec* aufruft, um ein völlig anderes Programm auszuführen. Diese Überlegungen führten dazu, daß viele Systeme (wie z.B. SVR4, 4.4BSD oder Linux) für diesen Anwendungsfall die Funktion *vfork* zur Verfügung stellen.

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
pid_t vfork(void);
```

gibt zurück: 0 im Kindprozeß; Prozeß-ID des Kindprozesses im Elternprozeß; -1 bei Fehler

Die Funktion **vfork** hat den gleichen Prototyp wie **fork** und kreiert ebenso wie **fork** einen Kindprozeß. Anders als **fork** kopiert **vfork** nicht den Adreßraum des Elternprozesses, sondern läßt dem Kindprozeß den Adreßraum des Elternprozesses mitbenutzen, bis der Kindprozeß **exec** oder **exit** aufruft. Auf Systemen (wie z.B. Linux), die mit dem früher vorgestellten COW-Verfahren arbeiten, ist **vfork** meist identisch zu **fork**.

Hinweis

Ein weiterer Unterschied zwischen **fork** und **vfork** – soweit sie nicht identisch sind – ist, daß bei **vfork** garantiert wird, daß der Kindprozeß zuerst (also vor dem Elternprozeß) abläuft, bis er entweder **exec** oder **exit** aufruft. Es ist darauf hinzuweisen, daß dies zu einem *Deadlock* führen kann, wenn der Kindprozeß auf Aktionen des Elternprozesses wartet, bevor er **exec** oder **exit** aufruft.

Beispiel

Demonstrationsprogramm zu vfork

Das folgende Programm 10.6 (`vfork.c`) ist weitgehend identisch mit dem Programm 10.2 (`forkdemo.c`). Neben einigen kleinen Änderungen wird in diesem Programm **vfork** anstelle von **fork** verwendet.

```
#include <sys/types.h>
#include "eighdr.h"

int global_var=100;

int
main(void)
{
    int    lokal_var, status;
    pid_t  pid;

    lokal_var=1;

    printf("---vor vfork-Aufruf---\n");

    switch ( pid=vfork() ) {
        case -1:
            fehler_meld(FATAL_SYS, "Fehler bei vfork");
            break;
```

```

    case 0:
        lokal_var++;
        global_var++;
        printf(".....Ich bin der Kindprozess.....\n");
        _exit(0); /* Kindprozess beendet sich */

    default:
        break;
}

wait(&status);
printf(".....Ich bin der Elternprozess.....\n");
printf("%s: global_var=%d, lokal_var=%d\n",
        (pid==0) ? "Kindprozess" : "Elternprozess", global_var, lokal_var);

exit(0);
}

```

Programm 10.6 (vfork.c): Demonstrationsbeispiel zu vfork

Nachdem man dieses Programm 10.6 (vfork.c) kompiliert und gelinkt hat

```
cc -o vfork vfork.c fehler.c
```

ergibt sich z.B. auf Systemen, die bei **vfork** nicht mit dem COW-Verfahren arbeiten, folgender Ablauf:

```

$ vfork
---vor vfork -Aufruf ---
.....Ich bin der Kindprozess.....
.....Ich bin der Elternprozess.....
Elternprozess: global_var = 101, lokal_var=2
$

```

Hier wirkt sich (anders als in Programm 10.2 (forkdemo.c)) das Inkrementieren der beiden Variablen `global_var` und `lokal_var` im Kindprozeß auch auf die gleichnamigen Variablen im Elternprozeß aus. Der Grund hierfür ist, daß bei **vfork** (anders als bei **fork**) der Kindprozeß nicht über einen eigenen Adreßraum (Datensegment, Stacksegment, Heap) verfügt, sondern den des Elternprozesses mitbenutzt.

Der Kindprozeß beendet sich im Programm 10.6 (vfork.c) mit `_exit(0)`. Es wurde nicht `exit(0)` verwendet, weil **exit** alle Standard-E/A-Puffer nicht nur leert (wie **_exit**), sondern diese auch alle schließt. Da der Kindprozeß noch im Adreßraum des Elternprozesses arbeitet, führt dies (bei einigen Systemen) dazu, daß auch die Standardausgabe des Elternprozesses geschlossen wird und somit die **printf**-Aufrufe im Elternprozeß dort nicht mehr erfolgreich schreiben können. Es würde sich also beim Start von **vfork** folgender Ablauf ergeben.

```

$ vfork
--- vor vfork Aufruf----
.....Ich bin der Kindprozess.....
$

```


Es sei nochmals darauf hingewiesen, daß die hier genannten Punkte nur für Systeme gelten, die nicht mit dem COW-Verfahren arbeiten.

10.2.6 clone – Ein fork (unter Linux) mit einer gemeinsamen Ressourcennutzung durch Eltern- und Kindprozeß



Auch wenn **fork** die traditionelle Art ist, unter Linux neue Prozesse zu kreieren, stellt Linux zusätzlich den Systemaufruf **clone** zur Verfügung, der es ermöglicht, für den kreierten Kindprozeß festzulegen, welche Ressourcen er sich mit dem Elternprozeß teilen soll.

```
#include <kernel/sched.h>
```

```
#include <linux/unistd.h>
```

```
pid_t clone(int flags);
```

gibt zurück: 0 im Kindprozeß; Prozeß-ID des Kindprozesses im Elternprozeß; -1 bei Fehler

Die Rückgabewerte von **clone** sind die gleichen wie bei **fork**. Anders als **fork** besitzt die Funktion **clone** einen Parameter `flags`. Hier sollte das Signal angegeben werden, das an den Elternprozeß geschickt wird, wenn der Kindprozeß sich beendet, was normalerweise SIGCHLD ist. Zusätzlich können mittels einer bitweisen OR-Verknüpfung folgende Konstantennamen, die in `<linux/sched.h>` definiert sind, angegeben werden:

CLONE_VM

Eltern- und Kindprozeß teilen sich den virtuellen Speicherraum (dieselben Speicherseiten) einschließlich des Stacks. Ist dieses Flag nicht angegeben, werden die Speicherseiten des Kindprozesses mit dem COW-Verfahren erzeugt.

CLONE_FS

Eltern- und Kindprozeß benutzen dieselbe Filesystemstruktur (wie z.B. das Working-Directory). Ansonsten wird diese Struktur für den Kindprozeß kopiert.

CLONE_FILES

Eltern- und Kindprozeß benutzen dieselben Filedeskriptoren. Ansonsten werden die Filedeskriptoren für den Kindprozeß kopiert.

CLONE_SIGHAND

Eltern- und Kindprozeß teilen sich die Signalhandlerrouinen. Ansonsten werden diese für den Kindprozeß kopiert.

Wenn zwei Ressourcen gleichzeitig vom Kind- und Elternprozeß benutzt werden, haben beide das gleiche Bild dieser Ressourcen.

Ist `CLONE_FILES` angegeben, werden nicht nur die offenen Dateien miteinander geteilt, sondern auch die aktuellen Positionen der jeweiligen Schreib-/Lesezeiger.

Wenn `CLONE_SIGHAND` angegeben wurde und einer der beiden Prozesse einen neuen Signalhandler für ein bestimmtes Signal einrichtet, benutzen beide Prozesse diesen neuen Signalhandler.

Wird ein anderes Signal als `SIGCHLD` spezifiziert, das bei der Beendigung des Kindprozesses an den Elternprozeß zu schicken ist, geben die verschiedenen ***wait***-Funktionen keine Informationen über einen solchen Kindprozeß zurück. Möchte man in einem solchen Fall dennoch Informationen erhalten, muß zusätzlich noch `__WCLONE` (definiert in `<linux/wait.h>`) mittels einer OR-Verknüpfung im Parameter `flags` angegeben werden. Dieses Verhalten läßt sich durch die daraus resultierende Flexibilität erklären. Würde ***wait*** Informationen über geklonte Prozesse zurückgeben, würde dies den Entwurf einer Standardbibliothek für Threads um ***clone*** erheblich komplizieren, da ***wait*** sowohl Informationen über Kindprozesse als auch über Threads zurückgeben müßte.

Auch wenn die direkte Benutzung von ***clone*** nicht empfehlenswert ist, gibt es verschiedene Bibliotheken, die ***clone*** benutzen und eine volle POSIX-kompatible Thread-Implementation bieten. Neuere Versionen der Linux-C-Bibliothek (`glibc2`) enthalten eine solche Bibliothek, wodurch ein Standard für die Benutzung von Threads über alle Linux-Plattformen geschaffen wird.



10.3 Warten auf Beendigung von Prozessen

10.3.1 Arten von Beendigungen eines Prozesses

Ein Prozeß kann auf unterschiedlichste Weise beendet werden:

1. Normale Beendigung (siehe auch Kapitel 9.2)
 - ▶ normales Beenden der Funktion `main` (mit oder ohne ***return***)
 - ▶ Aufruf der Funktionen ***exit*** oder ***_exit***
2. Anormale Beendigung (siehe auch Kapitel 13)
 - ▶ Aufruf der Funktion ***abort***
 - ▶ durch interne oder externe Signale

Wie bereits in Kapitel 9.2 genauer beschrieben, hat jeder Prozeß einen Exit-Status, den er bei seiner Beendigung an den aufrufenden Prozeß zurückgibt.

Exit-Status ist dabei eigentlich nicht die richtige Bezeichnung, denn für den Fall einer anormalen Beendigung generiert der Kern, nicht der Prozeß einen Beendigungsstatus für einen Prozeß, um über den Grund für die anormale Beendigung zu informieren.

Man unterscheidet also zwischen Exit-Status (Argument von **exit**, **_exit** oder Rückgabewert von **main**) und Beendigungsstatus.

Der Exit-Status wird vom Kern in den entsprechenden Beendigungsstatus konvertiert, wenn zur Prozeßbeendigung schließlich **_exit** aufgerufen wird (siehe auch Abbildung 9.1)

In jedem Fall kann der Elternprozeß den Beendigungsstatus eines Kindprozesses mit einer der beiden weiter unten beschriebenen Funktionen **wait** und **waitpid** erfahren.

10.3.2 Verwaiste Kindprozesse

Wenn ein Elternprozeß sich beendet, bevor alle seine Kindprozesse beendet sind, so wird der **init**-Prozeß der Elternprozeß von allen dessen Kindprozessen. Der Kern setzt dies um, indem er bei jeder Beendigung eines Prozesses die PID aller aktiven Prozesse überprüft. Besitzt ein noch aktiver Prozeß als PPID die PID des gerade beendeten Prozesses, so erhält er als neue PPID die Nummer 1 (Prozeß-ID von **init**). So ist immer sichergestellt, daß jeder Prozeß einen Elternprozeß hat.

10.3.3 Zombie-Prozesse

Wenn ein Elternprozeß nicht auf die Beendigung eines Kindprozesses wartet, so stellt sich die Frage, wie der Elternprozeß dann nachträglich den Beendigungsstatus des nun nicht mehr existierenden Kindprozesses erfragen kann. Dieses Problem wird dadurch gelöst, daß der Kern sich über jeden beendeten Prozeß eine gewisse Menge an Informationen hält, so daß der Elternprozeß auch nachträglich diese Information mittels einer der beiden Funktionen **wait** oder **waitpid** erfragen kann. Die dabei vom Kern aufgehobene Information umfaßt mindestens die PID, Beendigungsstatus und verbrauchte CPU-Zeit des beendeten Prozesses. In jedem Fall veranlaßt der Kern bei Beendigung eines Prozesses das Schließen aller noch offenen Dateien und die Freigabe des reservierten Speicherplatzes.

Solche Kindprozesse, die sich beendet haben, ohne daß der Elternprozeß auf sie wartete, werden in Unix als *Zombies* bezeichnet und beim Kommando **ps** wird für ihren Zustand **Z** ausgegeben.

Wenn verwaiste Kindprozesse, die – wie zuvor besprochen – als neuen Elternprozeß den **init**-Prozeß zugeordnet bekamen, sich beenden, so ruft **init** automatisch eine der **wait**-Funktionen auf, um ihren Beendigungsstatus zu erfragen. So ist sichergestellt, daß **init**-Kindprozesse niemals Zombies werden und das System nicht unnötig belasten.

10.3.4 **wait** und **waitpid** – Warten auf die Beendigung eines Prozesses

Bei einer normalen oder anormalen Beendigung eines Kindprozesses wird dem Elternprozeß das Signal `SIGCHLD` (siehe Kapitel 13) geschickt. Das Eintreffen eines solchen Signals ist nicht vorhersehbar, da es sich dabei um ein asynchrones Ereignis handelt.

Der Elternprozeß kann unter Verwendung des in Kapitel 13 beschriebenen Signalkonzepts festlegen, ob beim Eintreffen des Signals dieses zu ignorieren ist oder ob für diesen Fall eine eigene bereitgestellte Routine (der sogenannte Signalhandler) anzuspringen ist. Trifft der Elternprozeß keinerlei Vorkehrungen für das Eintreffen des Signals `SIGCHLD`, so wird es einfach ignoriert.

Um auf die Beendigung eines Kindprozesses zu warten, stehen die beiden Funktionen **wait** und **waitpid** zur Verfügung.

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

```
pid_t waitpid(pid_t pid, int *status, int optionen);
```

beide geben zurück: Prozeß-ID (bei Erfolg); 0 (WNOHANG wurde angegeben und kein Kindprozeß aktiv);
-1 (bei Fehler)

Ein Aufruf der Funktionen **wait** oder **waitpid** kann folgendes Verhalten nach sich ziehen:

1. Sofortige Rückkehr von **wait** bzw. **waitpid** mit dem Beendigungsstatus eines Kindprozesses, wenn ein Kindprozeß sich bereits früher beendet hat und der Kern nur auf die Abholung des Beendigungsstatus dieses Zombieprozesses wartet.
2. Sofortige Rückkehr mit Fehler, wenn keine Kindprozesse existieren.
3. Blockierung des aufrufenden Prozesses, wenn alle Kindprozesse immer noch aktiv sind.

wait und **waitpid** unterscheiden sich in den drei folgenden Punkten:

1. **wait** wartet nur auf die nächste Beendigung eines beliebigen Kindprozesses, während **waitpid** auf die Beendigung eines bestimmten Kindprozesses warten kann.
2. **wait** kann den aufrufenden Prozeß blockieren, bis sich ein Kindprozeß beendet, während bei **waitpid** mit einer Option die Blockierung des aufrufenden Prozesses unterbunden werden kann.
3. **waitpid** unterstützt anders als **wait** die Jobkontrolle (siehe Option `WUNTRACED` bei **waitpid**).

Beide Funktionen geben bei Erfolg die Prozeß-ID des Kindprozesses zurück, der sich beendet hat. Bei Fehler geben beide Funktionen -1 zurück. Mögliche Fehlersituationen sind dabei bei **wait** und **waitpid**, daß kein Kindprozeß existiert oder daß **wait** bzw. **waitpid** durch ein Signal unterbrochen wurden.

Bei **waitpid** führt zusätzlich die Nicht-Existenz eines angegebenen Prozesses oder einer Prozeßgruppe (siehe weiter unter) oder aber, daß es sich bei der angegebenen ID nicht um einen Kindprozeß handelt, zu einem Fehler.

Beendigungsstatus

Beide Funktionen schreiben den Beendigungsstatus an die Adresse, die mit dem Argument `status` übergeben wird. Falls der Aufrufer nicht an dem Beendigungsstatus interessiert ist, kann er für `status` einen NULL-Zeiger angeben.

Um den über `status` zurückgegebenen Beendigungsstatus zu interpretieren, schreibt POSIX.1 die in Tabelle 10.1 abgegebenen Makros vor.

Diese drei in `<sys/wait.h>` definierten Makros liefern Information darüber, wie sich der entsprechende Prozeß beendet hat. Abhängig davon, welches Makro TRUE (Wert verschieden von 0) liefert, müssen dann andere Makros aufgerufen werden, um z.B. den Exit-Status, Signalnummer usw. zu erfahren.

Makro	Beschreibung
WIFEXITED(status)	liefert TRUE, wenn status von einem Kindprozeß geliefert wurde, der sich normal beendet hat. Um in diesem Fall den Exit-Status des Kindprozeß (niederwertige 8 Bit von status) zu erfragen, muß WEXITSTATUS(status) aufgerufen werden.
WIFSIGNALED(status)	liefert TRUE, wenn status von einem Kindprozeß geliefert wurde, der sich anormal (durch Eintreffen eines Signals, das er nicht abging) beendet hat. Um in diesem Fall die Nummer des Signals zu erfahren, das den Prozeßabbruch bewirkte, muß
WTERMSIG(status)	aufgerufen werden. Um zu erfahren, ob das Signal zur Generierung einer core-Datei führte, kann in BSD und SVR4 (aber nicht POSIX.1)
WCOREDUMP(status)	aufgerufen werden.
WIFSTOPPED(status)	liefert TRUE, wenn status von einem Kindprozeß geliefert wurde, der angehalten wurde. Um in diesem Fall die Nummer des Signals zu erfahren, das das Anhalten des Prozesses bewirkte, muß
WSTOPSIG(status)	aufgerufen werden.

Tabelle 10.1: Makros zum Erfragen des von **wait** und **waitpid** gelieferten Beendigungsstatus.

Beispiel*Funktion zur Ausgabe des Beendigungsstatus*

Das Programm 10.7 (endestat.c) stellt eine Funktion `print_endestatus` zur Verfügung, die Makros aus der Tabelle 10.1 benutzt, um eine Beschreibung über den Beendigungsstatus auszugeben.

```
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

int
print_endestatus(int status)
{
    if (WIFEXITED(status)) {
        printf("Normale Beendigung; exit-Status=%d\n", WEXITSTATUS(status));

    } else if (WIFSIGNALED(status)) {
        printf("Abnormale Beendigung; Signalnummer=%d", WTERMSIG(status));
#ifdef WCOREDUMP
        if (WCOREDUMP(status))
            printf(" (core-Datei generiert)\n" );
        else
            printf("\n");
#else
        printf("\n");
#endif

    } else if (WIFSTOPPED(status))
        printf("Prozess wurde angehalten; Signalnummer=%d\n", WSTOPSIG(status));
}
```

Programm 10.7 (endestat.c): Ausgeben einer Beschreibung des Beendigungsstatus

Die Funktion `print_endestatus` werden wir in den nachfolgenden Programmen immer dann verwenden, wenn wir eine Beschreibung des Beendigungsstatus ausgeben lassen möchten, wie z.B. im folgenden Programm 10.8 (waitdemo.c).

Beispiel*Demonstrationsbeispiel für verschiedene Beendigungsstatus*

```
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
```

```

pid_t  pid;
int     status;
int     *zgr=NULL;

/*---- 1.Kind kreieren und mit exit beenden -----*/
if ( (pid=fork()) < 0 )
    fehler_meld(FATAL_SYS, "Fehler bei fork");
else if (pid == 0)
    exit(20);

if (wait(&status) != pid)
    fehler_meld(FATAL_SYS, "Fehler bei wait");
print_endestatus(status);

/*---- 2.Kind kreieren und mit abort beenden -----*/
if ( (pid=fork()) < 0 )
    fehler_meld(FATAL_SYS, "Fehler bei fork");
else if (pid == 0)
    abort();

if (wait(&status) != pid)
    fehler_meld(FATAL_SYS, "Fehler bei wait");
print_endestatus(status);

/*---- 3.Kind kreieren und mit illegalem Speicherzugriff beenden -----*/
if ( (pid=fork()) < 0 )
    fehler_meld(FATAL_SYS, "Fehler bei fork");
else if (pid == 0)
    *zgr = 100;

if (wait(&status) != pid)
    fehler_meld(FATAL_SYS, "Fehler bei wait");
print_endestatus(status);

    exit(0);
}

```

Programm 10.8 (*waitdemo.c*): Demonstrationsbeispiel für verschiedene Beendigungsstatus

Nachdem man dieses Programm 10.8 (*waitdemo.c*) kompiliert und gelinkt hat

```
cc -o waitdemo waitdemo.c endestat.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ waitdemo
Normale Beendigung; exit-Status=20
Abnormale Beendigung; Signalnummer=6
Abnormale Beendigung; Signalnummer=11
$

```

Die zu den Signalnummern gehörenden Namen können in den Headerdateien `</usr/include/signal.h>` bzw. `</usr/include/sys/signal.h>` oder unter Linux in `</usr/include/linux/signal.h>` nachgeschlagen werden.

Im obigen Ablaufbeispiel ist 6 die Signalnummer für SIGABRT und 11 die Signalnummer für SIGSEGV (illegaler Speicherzugriff; *segment violation*).

waitpid

Während **wait** nur auf die Beendigung des nächsten Prozesses wartet, kann man mit der neuen POSIX.1-Funktion **waitpid**, die in SVR4, Linux und 4.4BSD (nicht in 4.3BSD) angeboten wird, auf die Beendigung eines bestimmten Prozesses warten.

Das Argument `pid` legt bei **waitpid** fest, auf was zu warten ist

- `pid == -1`
Auf Beendigung eines beliebigen Kindprozesses warten; identisch zu **wait**
- `pid > 0`
Auf Beendigung des Kindprozesses mit der Prozeß-ID `pid` warten.
- `pid == 0`
Auf Beendigung eines Kindprozesses warten, dessen Prozeßgruppen-ID (siehe Kapitel 11.2) gleich der Prozeßgruppen-ID des aufrufenden Prozesses ist.
- `pid < -1`
Auf Beendigung eines Kindprozesses warten, dessen Prozeßgruppen-ID (siehe Kapitel 11.2) gleich dem absoluten Wert von `pid` ist.

waitpid gibt immer die Prozeß-ID des Kindprozesses zurück, der sich beendet hat. Den Beendigungsstatus schreibt **waitpid** an die mittels `status` übergebene Adresse.

Über das Argument `optionen` ist es möglich, noch weiteren Einfluß auf das Verhalten von **waitpid** zu nehmen. Sind keine `optionen` erwünscht, so ist für `optionen` 0 anzugeben, andernfalls ist ein Ausdruck anzugeben, in dem die entsprechenden Konstanten aus Tabelle 10.2 mit bitweisen OR (`|`) verknüpft sind.

Konstante	Beschreibung
WNOHANG	<i>waitpid</i> blockiert den aufrufenden Prozeß nicht, wenn der Kindprozeß mit der Prozeß-ID <code>pid</code> nicht sofort verfügbar ist. In diesem Fall liefert <i>waitpid</i> 0 als Rückgabewert.
WUNTRACED	Falls das System Jobkontrolle anbietet, so liefert <i>waitpid</i> den Status eines angehaltenen Kindprozesses, wenn dieser über das Argument <code>pid</code> spezifiziert ist und dessen Status seit seinem Anhaltezeitpunkt nicht abgefragt wurde. Das Makro <code>WIFSTOPPED</code> liefert dann TRUE, wenn es sich beim Rückgabewert um die PID eines angehaltenen Kindprozesses handelt.

Tabelle 10.2: Mögliche Konstanten für das `optionen`-Argument bei **waitpid**

Konstante	Beschreibung
WNOWAIT	(nur in SVR4) Der Prozeß, dessen Beendigungsstatus durch <i>waitpid</i> geliefert wurde, wird im Wartezustand gehalten, so daß auf diesen Prozeß erneut gewartet werden kann.
WCONTINUED	(nur SVR4) <i>waitpid</i> liefert den Status eines Kindprozesses mit <i>pid</i> , wenn dieser Kindprozeß, nachdem er angehalten wurde, wieder fortgesetzt wird und zwischenzeitlich sein Status nicht abgefragt wurde.

Tabelle 10.2: Mögliche Konstanten für das *optionen*-Argument bei ***waitpid***

Nachfolgend werden zwei Programme angegeben, die jeweils einen 50-m-Lauf zwischen drei Kindprozessen simulieren.

Beispiel*Simulation eines Wettrennens (Warten auf Siegerprozeß)*

Beim ersten Programm 10.9 (*rennen1.c*) wartet der Elternprozeß nur auf das Ende eines Prozesses, den »Siegerprozeß«, und gibt dann sofort den Sieger aus, ohne auf das Ende der beiden anderen Kindprozesse zu warten. Die beiden anderen Kindprozesse sind somit verwaist und erhalten als neuen Elternprozeß den *init*-Prozeß. Dies läßt sich an der in Klammern ausgegebenen parent-PID 1 erkennen.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include "eighdr.h"

static void rennen(int i);

int
main(void)
{
    int    i;
    pid_t  pid[4],
           pid_ende;

    printf("%18s%15s%15s\n", "Laeufer 1", "Laeufer 2", "Laeufer 3");
    printf("-----\n");
    if ((pid[1]=fork()) == 0)
        rennen(1);
    else if ((pid[2]=fork()) == 0)
        rennen(2);
    else if ((pid[3]=fork()) == 0)
        rennen(3);
    else {
        pid_ende=wait(NULL); /* auf Ende eines Kindprozesses warten */
        for (i=1 ; i<=3 ; i++) {
            if (pid_ende==pid[i])
                break;
        }
    }
}
```

```

    }
    printf("Laeufer %d hat gewonnen !!!\n", i);
}

exit(0);
}

static void rennen(int i)
{
    int meter=0;

    srand(time(NULL)+i);
    while (meter<50) {
        sleep(rand()%3+1);
        meter += 5;
        printf("%*s%3d (%4d)\n", i*15-7, " ", meter, getppid());
    }
    printf("%*s\n", i*15, "-----");
    exit(0);
}

```

Programm 10.9 (rennen1.c): Simulation eines Wettrennens zwischen Kindprozessen

Nachdem man dieses Programm 10.9 (rennen1.c) kompiliert und gelinkt hat

`cc -o rennen1 rennen1.c fehler.c`

ergibt sich z.B. der folgende Ablauf:

```

$ rennen1
-----
          5 ( 344)
5 ( 344)
          10 ( 344)
10 ( 344)      15 ( 344)
                10 ( 344)
                15 ( 344)
15 ( 344)      20 ( 344)
                20 ( 344)
20 ( 344)
25 ( 344)      25 ( 344)
                25 ( 344)
30 ( 344)      30 ( 344)
                35 ( 344)
                30 ( 344)
                35 ( 344)
          40 ( 344)
35 ( 344)

```



```
    }

    exit(0);
}

static void rennen(int i) /*--- identisch zur vorherigen Funktion rennen */
{
    int meter=0;
    srand(time(NULL)+getpid()+i);
    while (meter<50) {
        sleep(rand()%3+1);
        meter += 5;
        printf("%s%3d (%4d)\n", i*15-7, " ", meter, getpid());
    }
    printf("%s\n", i*15, "-----");
    exit(0);
}
```

Programm 10.10 (rennen2.c): Simulation eines Wettrennens zwischen Kindprozessen

Nachdem man dieses Programm 10.10 (rennen2.c) kompiliert und gelinkt hat

```
cc -o rennen2 rennen2.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

\$ rennen2			
	Laeufer 1	Laeufer 2	Laeufer 3
		5 (348)	
	5 (348)		
			5 (348)
		10 (348)	
	10 (348)		
		15 (348)	
			10 (348)
			15 (348)
	15 (348)		
		20 (348)	
			20 (348)
	20 (348)		
	25 (348)		
		25 (348)	
			25 (348)
	30 (348)		
		30 (348)	
		35 (348)	
			30 (348)
			35 (348)
		40 (348)	
	35 (348)		
	40 (348)		
			40 (348)
			45 (348)

```

                45 ( 348)
45 ( 348)
                50 ( 348)
                -----
                50 ( 348)
                -----
50 ( 348)
-----
Zieleinlauf:
Laeufer 3
Laeufer 2
Laeufer 1
$

```

10.3.5 Verhindern von Zombies

Wenn man ein Programm erstellt, in dem man einen Kindprozeß kreiert, auf dessen Ende man nicht warten möchte, man aber auch gleichzeitig verhindern möchte, daß dieser Kindprozeß ein **Zombie** wird, so muß man **fork** zweimal aufrufen.

Programm 10.11 (nozombie.c) verdeutlicht diese Technik. Es kreiert zunächst einen Kindprozeß, der dann seinerseits einen Kindprozeß kreiert, bevor er sich beendet. Dies bewirkt, daß der Enkelprozeß nun verwaist ist und als neuen Elternprozeß den `init`-Prozeß erhält. Da der `init`-Prozeß bei Beendigung eines seiner Kindprozesse immer automatisch **wait** aufruft, um dessen Beendigungsstatus zu ermitteln, ist sichergestellt, daß der »Enkelprozeß« niemals ein **Zombie** wird und das System unnötig belastet.

```

#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    pid_t pid;

    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid == 0) { /*----- Kindprozess -----*/
        printf("Elternprozess %d (Grosseltern %d)\n", getpid(), getppid());
        if ( (pid=fork()) < 0)
            fehler_meld(FATAL_SYS, "fork-Fehler");
        else if (pid > 0) {
            sleep(1);
            printf("Elternprozess %d (Enkel %d; Grosseltern %d) ---\n",
                getpid(), pid, getppid());
            exit(0); /* Kindprozess (Elternprozess vom Enkel) beendet sich */
        }

        /*----- Enkelprozess -----*/
        /*          Sobald sein Elternprozess exit aufgerufen hat, wird */
        /*          sein neuer Elternprozess der init-Prozess          */
    }
}

```

```

    printf("Enkel %d (Elternprozess %d)\n", getpid(), getppid());
    sleep(3);
    printf("Enkel %d (Elternprozess %d)\n", getpid(), getppid());
    exit(0); /* Enkelprozess beendet sich */
}

/*- Grosselternprozess wartet auf Beendigung seines leiblichen Kindes -*/
if (waitpid(pid, NULL, 0) != pid)
    fehler_meld(FATAL_SYS, "waitpid-Fehler");

/*..... Code vom Grosselternprozess .....*/

printf("---- Grosselternprozess %d laeuft weiter ----\n", getpid());

exit(0);
}

```

Programm 10.11 (nozombie.c): Vermeiden von Zombies

Nachdem man dieses Programm 10.11 (nozombie.c) kompiliert und gelinkt hat

```
cc -o nozombie nozombie.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ nozombie
Elternprozess 217 (Grosseltern 216)
Enkel 218 (Elternprozess 217)
Elternprozess 217 (Enkel 218; Grosseltern 216) ---
---- Grosseltern-Prozess 216 laeuft weiter ----
Enkel 218 (Elternprozess 1)
$

```

Bei der Ausgabe im Programm 10.11 (nozombie.c) wird der ursprüngliche Prozeß als Großelternprozeß, der mit dem ersten **fork** kreierte Prozeß als Elternprozeß und der mit dem zweiten **fork** kreierte Prozeß als Enkel bezeichnet.

10.3.6 wait3 und wait4 – Warten auf Ende eines Prozesses (Information über benutzte Ressourcen)

Um auf das Ende eines Prozesses zu warten und dann bei Prozeßende zu erfahren, welche Ressourcen vom beendeten Prozeß und allen seinen Kindprozessen benutzt wurden, stehen in BSD-Unix und Linux die beiden Funktionen **wait3** und **wait4**, die nicht Bestandteil von POSIX.1 sind, zur Verfügung.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *status, int optionen, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int optionen, struct rusage *usage);
```

beide geben zurück: Prozeß-ID (bei Erfolg); -1 bei Fehler

Die Struktur `rusage` enthält die Informationen über die benutzten Ressourcen des beendeten Kindprozesses, wie z.B. verbrauchte CPU-Zeit, Anzahl der empfangenen Signale usw. Näheres dazu liefert die Manpage `getrusage(2)`.

Hinweis

wait4 entspricht weitgehend der Funktion **waitpid**. Der einzige Unterschied ist, daß bei **wait4** zusätzlich Informationen über die Ressourcenbenutzung (viertes Argument `usage`) des beendeten Prozesses zurückgegeben werden.

wait3 gleicht andererseits der Funktion **wait4**, erlaubt es dem Aufrufer aber nicht festzulegen, auf welchen Kindprozeß zu warten ist.

SVR4 stellt die Funktion **wait3** in der *BSD compatibility library* zur Verfügung.

10.4 Synchronisationsprobleme zwischen Eltern- und Kindprozessen

Synchronisationsprobleme (*race condition*) zwischen Eltern- und Kindprozessen treten immer dann auf, wenn Eltern- und Kindprozesse voneinander abhängig sind.

Das nachfolgende Programm 10.12 (`forkadd1.c`) demonstriert eine solche Abhängigkeit. Bei diesem Programm soll der Elternprozeß zwei Zahlen einlesen und diese beiden Zahlen in eine Datei `zahlen` schreiben, aus die sie der Kindprozeß dann lesen soll, bevor er sie addiert und das Ergebnis ausgibt.

```
#include <sys/types.h>
#include "eighdr.h"

int
main(void)
{
    pid_t    pid;
    FILE     *fz;
    int      zahl1, zahl2;
```

```

if ( (pid=fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");

else if (pid == 0) {
    if ( (fz=fopen("zahlen", "r")) == NULL)
        fehler_meld(FATAL_SYS, "kann Datei 'zahlen' nicht oeffnen");
    fscanf(fz, "%d %d", &zahl1, &zahl2);
    fclose(fz);
    printf("%d + %d = %d\n", zahl1, zahl2, zahl1+zahl2);
    exit(0);
}

/*----- Elternprozess -----*/
printf("Gib zwei Zahlen (durch Komma getrennt) ein: ");
scanf("%d, %d", &zahl1, &zahl2);
if ( (fz=fopen("zahlen", "w")) == NULL)
    fehler_meld(FATAL_SYS, "kann Datei 'zahlen' nicht oeffnen");
fprintf(fz, "%d %d", zahl1, zahl2);
fclose(fz);

exit(0);
}

```

Programm 10.12 (forkadd1.c): Elternprozeß liest aus Datei zwei vom Kind geschriebene Zahlen

Bei dieser Aufgabenstellung ist eine Synchronisation zwischen Eltern- und Kindprozeß notwendig, denn sonst versucht der Kindprozeß aus der Datei `zahlen` zu lesen, bevor der Elternprozeß die eingelesenen Zahlen überhaupt in die Datei `zahlen` geschrieben hat, wie dies auch im Ablaufbeispiel ersichtlich wird.

Nachdem man dieses Programm 10.12 (`forkadd1.c`) kompiliert und gelinkt hat

```
cc -o forkadd1 forkadd1.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ forkadd1
Gib zwei Zahlen (durch Komma getrennt) ein:
kann Datei 'zahlen' nicht oeffnen: No such file or directory  [Fehlermeldung]
5,10 [hier erfolgt erst Eingabe, obwohl Kind bereits versuchte aus zahlen zu lesen]
$ cat zahlen
5 10
$

```

Es ist hier also eine Synchronisation zwischen Eltern- und Kindprozeß notwendig.

Eine einfache Möglichkeit für den Kindprozeß auf die Beendigung des Elternprozesses zu warten, ist die Angabe der folgenden Schleife

```

while (getppid() != 1)
    sleep (1);

```

Diese sogenannte *Polling*-Methode hat jedoch zwei Nachteile.

Sie verbraucht CPU-Zeit, da **getppid** jede Sekunde erneut aufgerufen wird, um festzustellen, ob sich der Elternprozeß zwischenzeitlich beendet hat, was sich an der parent-PID 1 (init ist neuer Elternprozeß) erkennen läßt.

Der andere Nachteil ist, daß sie nur eingesetzt werden kann, wenn sich der Elternprozeß nach Ausführung der Aktion, auf die der Kindprozeß wartet, beendet. Oft ist dies jedoch nicht der Fall. Ändern wir z.B. die vorherige Aufgabenstellung so, daß der Kindprozeß das Ergebnis der Addition nicht ausgeben, sondern dieses in die Datei `ergebnis` schreiben soll, aus der es dann wiederum der Elternprozeß lesen und dann ausgeben soll, so läßt sich diese *Polling*-Methode nicht einsetzen.

10.4.1 Synchronisation von Eltern- und Kindprozeß mit Signalen

Eine andere und bessere Methode der Synchronisation ist die Verwendung von Signalen. Signale werden in Kapitel 13 vorgestellt. Wir stellen bereits hier eine mögliche Implementierung der Synchronisation zwischen Eltern- und Kindprozeß mit Signalen vor. Dazu werden im Programm 10.13 (`forksync.c`) fünf Funktionen bereitgestellt:

INIT_SYNCH

Synchronisation initialisieren.

HALLO_PAPA

Kindprozeß informiert Elternprozeß, daß seine Aktion abgeschlossen ist.

WARTE_AUF_PAPA

Kindprozeß wartet auf Signal von Elternprozeß, daß dieser die entsprechende Aktion durchgeführt hat.

HALLO_KIND

Elternprozeß informiert Kindprozeß, daß seine Aktion abgeschlossen ist.

WARTE_AUF_KIND

Elternprozeß wartet auf Signal vom Kindprozeß, daß dieser die entsprechende Aktion durchgeführt hat.

```
#include <signal.h>
#include "eighdr.h"

static volatile sig_atomic_t  sflag;
static sigset_t      neu_smaske, alt_smaske, null_smaske;

/*----- Signalhandler fuer die Signale SIGUSR1 und SIGUSR2 -----*/
static void sig_usr(int signr)
{
    INIT_SYNCH();
    sflag = 1;
}
```

```

/*----- Synchronisation initialisieren -----*/
void INIT_SYNCH(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann SIGUSR1-Signalhandler nicht installieren");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann SIGUSR2-Signalhandler nicht installieren");

    sigemptyset(&null_smaske);

    sigemptyset(&neu_smaske);
    sigaddset(&neu_smaske, SIGUSR1);
    sigaddset(&neu_smaske, SIGUSR2);

    if (sigprocmask(SIG_BLOCK, &neu_smaske, &alt_smaske) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
}

/*----- Information von Kind an Elternprozess, dass es fertig ----*/
void HALLO_PAPA(pid_t pid)
{
    kill(pid, SIGUSR2);
}

/*----- Kind wartet auf Signal vom Elternprozess -----*/
void WARTE_AUF_PAPA(void)
{
    while (sflag == 0)
        sigsuspend(&null_smaske); /* Warten auf Signal vom Elternprozess*/

    sflag = 0;

    if (sigprocmask(SIG_SETMASK, &alt_smaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
}

/*----- Information von Elternprozess an Kind, dass er fertig ist */
void HALLO_KIND(pid_t pid)
{
    kill(pid, SIGUSR1);
}

/*----- Elternprozess wartet auf Signal vom Kind -----*/
void WARTE_AUF_KIND(void)
{
    while (sflag == 0)
        sigsuspend(&null_smaske); /* Warten auf Signal vom Elternprozess */

    sflag = 0;
}

```

```

    if (sigprocmask(SIG_SETMASK, &alt_maske, NULL) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
}

```

Programm 10.13 (forksync.c): Funktionen zur Synchronisation von Eltern- und Kindprozeß

In späteren Kapiteln werden wir weitere Möglichkeiten der Synchronisation von Eltern- und Kindprozessen kennenlernen.

Das Programm 10.14 (forkadd2.c) verwendet die Funktionen aus Programm 10.13 (forksync.c), um Eltern- und Kindprozeß zu synchronisieren. Die Additionsaufgabe vom Programm 10.14 (forkadd2.c) muß zeitlich in folgenden Schritten ablaufen.

1. Der Elternprozeß soll zwei Zahlen einlesen und diese in die Datei zahlen schreiben.
2. Der Kindprozeß liest diese beiden Zahlen aus der Datei zahlen und schreibt das Ergebnis in die Datei ergebnis.
3. Der Elternprozeß liest dann das Ergebnis aus der Datei ergebnis und gibt es auf der Standardausgabe aus.

```

#include <sys/types.h>
#include "eighdr.h"

int
main(void)
{
    pid_t    pid;
    FILE     *fz;
    int      zahl1, zahl2, ergeb;

    INIT_SYNCH(); /*----- Synchronisation initialisieren -----*/

    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid == 0) {
        WARTE_AUF_PAPA(); /*-----*/

        if ( (fz=fopen("zahlen", "r")) == NULL)
            fehler_meld(FATAL_SYS, "kann Datei 'zahlen' nicht oeffnen");
        fscanf(fz, "%d %d", &zahl1, &zahl2);
        fclose(fz);

        if ( (fz=fopen("ergebnis", "w")) == NULL)
            fehler_meld(FATAL_SYS, "kann Datei 'ergebnis' nicht oeffnen");
        fprintf(fz, "%d\n", zahl1+zahl2);
        fclose(fz);

        HALLO_PAPA(getppid()); /*-----*/

        exit(0);
    }
}

```

```

/*..... Elternprozess .....*/
printf("Gib zwei Zahlen (durch Komma getrennt) ein: ");
scanf("%d, %d", &zahl1, &zahl2);
if ( (fz=fopen("zahlen", "w")) == NULL)
    fehler_meld(FATAL_SYS, "kann Datei 'zahlen' nicht oeffnen");
fprintf(fz, "%d %d", zahl1, zahl2);
fclose(fz);

HALLO_KIND(pid); /*-----*/

WARTE_AUF_KIND(); /*-----*/

if ( (fz=fopen("ergebnis", "r")) == NULL)
    fehler_meld(FATAL_SYS, "kann Datei 'ergebnis' nicht oeffnen");
fscanf(fz, "%d", &ergeb);
printf("--- %d + %d = %d ---\n", zahl1, zahl2, ergeb);
fclose(fz);

exit(0);
}

```

Programm 10.14 (forkadd2.c): Synchronisiertes Lesen und Schreiben von Eltern- und Kindprozeß

Nachdem man dieses Programm 10.14 (forkadd2.c) kompiliert und gelinkt hat

```
cc -o forkadd2 forkadd2.c forksync.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ forkadd2
Gib zwei Zahlen (durch Komma getrennt) ein: 10,5
--- 10 + 5 = 15 ---
$ cat zahlen
10 5
$ cat ergebnis
15
$

```

10.5 Die exec-Funktionen

Wenn ein Prozeß eine der folgenden sechs **exec**-Funktionen aufruft, so wird er vollständig durch das angegebene neue Programm ersetzt. Das neue Programm beginnt seine Ausführung bei seiner `main`-Funktion.

Es stehen sechs verschiedene **exec**-Funktionen zur Verfügung.

```
#include <unistd.h>

int execl(const char *pfadname, const char *arg0, ... /* NULL */ );

int execv(const char *pfadname, char *const argv[]);

int execle(const char *pfadname, const char *arg0, ...
           /* NULL, char *const envp[] */ );

int execve(const char *pfadname, char *const argv[], char *const envp[]);

int execlp(const char *dateiname, const char *arg0, ... /* NULL */ );

int execvp(const char *dateiname, char *const argv[]);

           alle sechs geben zurück: -1 bei Fehler; keine Rückkehr (bei Erfolg)
```

Ein **exec**-Aufruf bewirkt keine Änderung der Prozeß-ID, da nicht wie bei **fork** ein neuer Prozeß kreiert wird, sondern nur die Segmente (Textsegment, Datensegment, Heap und Stack) des aktuellen Prozesses durch das neue Programm überschrieben werden.

Die sechs **exec**-Funktionen unterscheiden sich nur wenig in ihren Parametern. Nur eine der Funktionen, die Funktion **execve**, stellt unter Linux einen Systemaufruf dar. Die restlichen Funktionen sind in der C-Bibliothek implementiert und rufen ihrerseits **execve** auf.

10.5.1 Unterschiede der exec-Funktionen im Überblick

Die Namen der **exec**-Funktionen unterscheiden sich in nur wenigen Buchstaben. Diese angehängten Buchstaben sind Abkürzungen mit folgenden Bedeutungen:

- l (*list*)
Der Funktion werden die Komandozeilenargumente in Form einer Liste übergeben.
- v (*vector*)
Der Funktion werden die Komandozeilenargumente als Vektor (`argv[]`) übergeben.
- p (*path*)
Der Buchstabe steht für `PATH` und bedeutet, daß diese Funktion einen Dateinamen (und nicht einen Pfadnamen) als Argument erwartet. Bei den beiden letzten **exec**-Funktionen, für die dies zutrifft, bedeutet dies, daß der angegebene `dateiname` eine ausführbare Datei ist, nach der in den `PATH`-Directories gesucht wird.
- e (*environment*)
Die Funktion erwartet die Environment-Liste als Vektor (`envp[]`) und benutzt nicht die aktuelle Environment.

Tabelle 10.3 zeigt die Unterschiede zwischen den sechs **exec**-Funktionen im Überblick.

Funktion	Pfadname	Dateiname	Argliste	argv[]	environ	envp[]
execl	x		x		x	
execlp		x	x		x	
execle	x		x			x
execv	x			x	x	
execvp		x		x	x	
execve	x			x		x
Buchstabe im Namen		p	l	v		e

Tabelle 10.3: Unterschiede zwischen den sechs exec-Funktionen

10.5.2 Interpretation des Dateinamens (bei execlp und execvp)

Wenn eine Funktion als Argument einen Dateinamen erwartet (**execlp** und **execvp**), dann gilt folgendes:

- ▶ Enthält der beim Aufruf angegebene Dateiname einen Slash (/), so wird er als Pfadname interpretiert,
- ▶ andernfalls wird in den PATH-Directories nach einer ausführbaren Datei diesen Namens gesucht. Wird eine solche Datei gefunden, so wird sie für den Fall, daß sie Maschinencode enthält, direkt gestartet. Enthält sie keinen Maschinencode, so wird angenommen, daß es sich um Shellskript handelt und zur Ausführung dieser Datei wird **/bin/sh** aufgerufen.

10.5.3 Unterschiede in der Form der Argumentübergabe

execl, execlp, execle

Hier müssen die Kommandozeilenargumente des neu zu startenden Programms einzeln in Form einer Liste (l) angegeben werden. Das Ende der Argumentliste muß dabei durch einen NULL-Zeiger angezeigt werden.

execv, execvp, execve

Hier müssen die Kommandozeilenargumente des neu zu startenden Programms in einem String-Vektor (v) der Form (char *argv[]) abgelegt werden, und die Adresse dieses Vektors muß beim Aufruf der Funktionen angegeben werden.

10.5.4 Unterschiede bei Benutzung des Environment

Die beiden Funktionen **execle** und **execve** (enden mit e) lassen es zu, daß man die Adresse eines Environment-Vektors (char *envp[]) übergibt, der die Environment-Strings enthält.

Die anderen vier Funktionen übernehmen implizit den `environ`-Vektor (siehe Kapitel 9.2) des aufrufenden Prozesses für das neu zu startende Programm. Diese vier Funktionen verwendet man, um das aktuelle Environment an einen Kindprozeß zu vererben, was in den meisten Fällen auch erwünscht ist. Nur in wenigen Ausnahmen möchte man ein eigenes neues Environment für einen Kindprozeß festlegen. Ein Beispiel hierfür ist das Programm **login**, das eine neue Loginshell startet.

10.5.5 Vererbungen bei exec

Wenn ein Prozeß **exec** aufruft, um sich mit einem neuen Programm zu überlagern, so erbt das neue Programm folgendes vom aufrufenden Prozeß:

- ▶ IDs (PID und PPID, reale User-ID und Group-ID, Zusatz-Group-IDs, Prozeßgruppen-ID, Session-ID)
- ▶ Working-Directory und Root-Directory
- ▶ Dateikreierungsmaske und Dateisperren
- ▶ Kontrollterminal
- ▶ Signalmaske und hängende Signale
- ▶ noch laufende Zeitschaltuhren
- ▶ Ressourcenlimits
- ▶ die Werte von `tms_utime`, `tms_stime`, `tms_cutime` und `tms_ustime`

Vererben von offenen Dateien ist abhängig vom close-on-exec-Flag

Ob offene Dateien weiter vererbt werden, hängt davon ab, ob für die jeweiligen Filedeskriptoren das *close-on-exec*-Flag gesetzt ist oder nicht. Wenn es gesetzt ist, wird der entsprechende Filedeskriptor beim **exec**-Aufruf geschlossen, andernfalls bleibt er auch für das neue mit **exec** gestartete Programm offen. Das *close-on-exec*-Flag kann mit der in Kapitel 4.9 beschriebenen Funktion **fcntl** gesetzt werden. Nach der Voreinstellung ist das *close-on-exec*-Flag nicht gesetzt.

POSIX.1 schreibt vor, daß offene Directories bei einem **exec**-Aufruf geschlossen werden. Die Funktion **opendir** berücksichtigt diese Forderung, indem sie automatisch **fcntl** aufruft, um das *close-on-exec*-Flag für das gerade geöffnete Directory zu setzen.

Vererbung von effektiven IDs ist nicht garantiert

Während bei einem **exec**-Aufruf die realen IDs immer weiter vererbt werden, können sich die effektiven IDs ändern. Wenn nämlich für das neu zu startende Programm das Set-User-ID-Bit gesetzt ist, so wird die effektive User-ID auf die Eigentümer-ID der neuen Programmdatei gesetzt. Das gilt auch für die effektive Group-ID. Nur wenn weder das Set-User-ID- noch das Set-Group-ID-Bit beim neu zu startenden Programm gesetzt sind, wird die entsprechende effektive ID vererbt.

Beispiel*Ausgeben der Kommandozeilenargumente und der Environment-Liste*

In den beiden nachfolgenden Beispielen werden wir bei den **exec**-Aufrufen das folgende Programm 10.15 (`arg_env.c`) verwenden, das lediglich die Kommandozeile Argumente und die aktuelle Environment-Liste ausgibt.

```
#include "eighdr.h"

extern char **environ;
int
main(int argc, char *argv[])
{
    int i;
    char **zgr;

    printf("\n---- Programm %s ----\n", argv[0]);
    printf("      Seine Argumente:\n");
    for (i=1; i<argc; i++)
        printf("%20d : %s\n", i, argv[i]);

    printf("      Environment:\n");
    for (zgr=environ; *zgr != NULL; zgr++)
        printf("%15s%s\n", " ", *zgr);

    exit(0);
}
```

*Programm 10.15 (**arg_env.c**): Ausgabe der Kommandozeilenargumente und der Environment-Liste*

Dieses Programm wird wie folgt kompiliert und gelinkt

```
cc -o arg_env arg_env.c fehler.c
```

Aufgerufen wird das Programm `arg_env` in den beiden folgenden Beispielprogrammen.

Beispiel*Demonstrationsbeispiel zu den Funktionen `execle` und `execlp`*

```
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

char *neu_env[] = {
    "TERM=vt100",
    "VISUAL=emacs",
    "TEMP=/usr/tmp",
    NULL
};

int
main(void)
```



```

{
    pid_t    pid;

    /*----- Demo zu execl -----*/
    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid == 0) {
        if (execl("/home/hh/sysprog/kap10/arg_env",
            "arg_env", "Hallo", "", "Welt", NULL, neu_env) < 0)
            fehler_meld(FATAL_SYS, "execl-Fehler");
    }
    if (waitpid(pid, NULL, 0) < 0)
        fehler_meld(FATAL_SYS, "waitpid-Fehler");

    /*----- Demo zu execlp -----*/
    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid == 0) {
        if (execlp("arg_env", "arg_env", "eins", "zwei", "drei", NULL) < 0)
            fehler_meld(FATAL_SYS, "execlp-Fehler");
    }

    exit(0);
}

```

Programm 10.16 (exec1.c): Demonstrationsbeispiel zu den Funktionen **execl** und **execlp**

Nachdem man dieses Programm 10.16 (exec1.c) kompiliert und gelinkt hat

```
cc -o exec1 exec1.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ exec1
```

```
----- Programm arg_env -----
```

```
Seine Argumente:
```

```
1 : Hallo
```

```
2 :
```

```
3 : Welt
```

```
Environment:
```

```
TERM=vt100
```

```
VISUAL=emacs
```

```
TEMP=/usr/tmp
```

```
----- Programm arg_env -----
```

```
Seine Argumente:
```

```
1 : eins
```

```
2 : zwei
```

```
3 : drei
```

```
Environment:
```

```
HOME=/home/hh
```

```
PATH=/usr/local/bin:/usr/bin:/bin:.....
```

```
SHELL=/bin/tcsh
```

```

        TERM=console
        MAIL=/var/spool/mail/hh
        .....
        .....
$

```

Beispiel

exec-Aufruf mit einer Interpreterdatei

SVR4 und BSD-Unix erlauben sogenannte Interpreterdateien. Dies sind Textdateien, die mit einer Zeile der folgenden Form beginnen.

```
#! pfadname [argumente]1
```

Die häufigste Anwendung finden diese Dateien in der Shellprogrammierung. Wenn man z.B. ein C-Shellskript erstellt hat und die Anwender dieses Skripts arbeiten mit der Bourne-Shell, so gibt man einfach am Anfang des Skripts folgendes an:

```
#! /bin/csh
```

So stellt man sicher, daß dieses Shellskript in jedem Fall von der C-Shell ausgeführt wird. Als *pfadname* sollte man immer den absoluten Pfadnamen angeben, da hier kein explizites Suchen in den PATH-Directories durchgeführt wird.

Solche Interpreterdateien sind eine typische Anwendung eines **exec**-Aufrufs durch den Kern. Er startet hier also mit Hilfe von **exec** das nach *#!* angegebene Programm *pfadname*.

Das folgende Programm 10.17 (*exec2.c*) verdeutlicht die Auswirkungen von Interpreterdateien auf die Kommandozeile des neu mit **exec** gestarteten Programms.

```

#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

char    *neu_env[] = {
        "TERM=vt100",
        "VISUAL=emacs",
        NULL
};

int
main(void)
{
    pid_t  pid;

    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid == 0) {

```

1. Hinweis: Viele Systeme haben ein Limit von 32 Zeichen für diese Zeile (einschließlich *#!*, *pfadname*, *argumente* und Leerzeichen).

```

    if (execle("/home/hh/sysprog/kap10/interpr",
               "interpr", "arg1", "arg 2", "ARG3", NULL, neu_env) < 0)
        fehler_meld(FATAL_SYS, "execle-Fehler");
    }

    if (waitpid(pid, NULL, 0) < 0)
        fehler_meld(FATAL_SYS, "waitpid-Fehler");
    exit(0);
}

```

Programm 10.17 (exec2.c): exec-Aufruf mit einer Interpreterdatei

Nachdem man dieses Programm 10.17 (exec2.c) kompiliert und gelinkt hat

```
cc -o exec2 exec2.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ cat interpr
#!/home/hh/sysprog/kap10/arg_env eins zwei drei vier
$ exec2

----- Programm arg_env -----
    Seine Argumente:
        1 : eins zwei drei vier
        2 : /home/hh/sysprog/kap10/interpr
        3 : arg1
        4 : arg 2
        5 : ARG3
    Environment:
        TERM=vt100
        VISUAL=emacs
$

```

An der obigen Ausgabe wird deutlich, daß der **exec**-Aufruf für die Interpreterdatei `interpr` dazu führt, daß `arg_env` als neues Programm (`argv[0]`) für den **exec**-Aufruf genommen wird. Die in der ersten Zeile der Interpreterdatei angegebenen Argumente werden in `argv[1]` abgelegt, so daß die beim ursprünglichen **exec**-Aufruf vorliegenden Argumente um zwei Positionen nach rechts geschoben werden, also ab `argv[2]` beginnen.

10.6 Die Funktion system

In manchen Situationen ist es erwünscht, daß man von einem Programm aus ein anderes Programm aufruft, wie z.B. den Editor **vi**.

Hierfür steht die ANSI-C-Funktion **system**, die intern die drei Funktionen **fork**, **exec** und **waitpid** aufruft, zur Verfügung.

```
#include <stdlib.h>
```

```
int system(const char *kdozeile);
```

gibt zurück:

- a) bei Angabe von NULL für *kdozeile* 0 (wenn kein Kommandoprozessor verfügbar); verschieden von 0 sonst. So kann festgestellt werden, ob die **system**-Funktion am aktuellen System verfügbar ist (in Unix ist **system** immer verfügbar).
- b) -1, wenn das interne **fork** fehlschlug, oder der interne **waitpid**-Aufruf einen anderen Fehler als EINTR geliefert hat. Hierbei wird *errno* entspr. gesetzt.
- c) Rückgabewert, der einem `exit(127)` durch die Shell entspricht, wenn der interne **exec**-Aufruf mißlang.
- d) Beendigungsstatus der Shell im Format von **waitpid**, wenn alle drei Funktionen (**fork**, **exec** und **waitpid**), die von **system** aufgerufen wurden, erfolgreich ausgeführt werden konnten.

Obwohl **system** eine von ANSI C definierte Funktion ist, ist das Verhalten von **system** doch sehr systemabhängig. Die hier gegebene Beschreibung entspricht dem POSIX.2-Standard².

In der *kdozeile* können bei einem **system**-Aufruf alle in der Shell erlaubten Metazeichen angegeben werden, wie z.B. * für Dateinamenexpansion, | für Pipe oder > für Dateiumlenkung. Der Vorteil der Verwendung von **system** gegenüber einer eigenen Nachbildung eines **system**-Aufrufs mittels **fork**, **exec** und **waitpid** ist, daß **system** die erforderlichen Fehler und Signalbehandlung von sich aus durchführt.

Beispiel

Demonstrationsbeispiel zur Funktion **system**

Das folgende Programm 10.18 (`sysdemo.c`) demonstriert die Anwendung der Funktion **system**. Es verwendet zur Ausgabe des Beendigungsstatus die Funktion `print_endestatus` aus dem Programm `endestat.c`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    int    status;

    printf("---- 1. system-Aufruf-----\n");
    if ( (status = system("echo `ls -a | wc -l` Dateien")) < 0)
        fehler_meld(FATAL_SYS, "system-Fehler");
    print_endestatus(status); /* aus Programm ende_stat.c */
}
```

2. **system** ist nicht durch POSIX.1 standardisiert, da es keine Schnittstelle zum Betriebssystem, sondern eben zur Shell darstellt.

```

printf("\n--- 2. system-Aufruf-----\n");
if ( (status = system("llllllllllll")) < 0)
    fehler_meld(FATAL_SYS, "system-Fehler");
print_endestatus(status); /* aus Programm ende_stat.c */

printf("\n--- 3. system-Aufruf-----\n");
if ( (status = system("pwd; exit 123")) < 0)
    fehler_meld(FATAL_SYS, "system-Fehler");
print_endestatus(status); /* aus Programm ende_stat.c */

exit(0);
}

```

Programm 10.18 (sysdemo.c): Demonstrationsbeispiel zur Funktion system

Nachdem man dieses Programm 10.18 (sysdemo.c) kompiliert und gelinkt hat

```
cc -o sysdemo sysdemo.c endestat.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ sysdemo
---- 1. system-Aufruf-----
48 Dateien
Normale Beendigung; exit-Status=0

---- 2. system-Aufruf-----
sh: llllllllllll: command not found
Normale Beendigung; exit-Status=127

---- 3. system-Aufruf-----
/home/hh/sysprog/kap10
Normale Beendigung; exit-Status=123
$

```

Beispiel

Mögliche Implementierung der Funktion system

```

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
int system(const char *kdozeile) /*--- Version ohne Signalbehandlung ---*/
{
    pid_t pid;
    int status;

    if (kdozeile == NULL)
        return(1); /* In Unix ist immer Kommandoprozessor vorhanden */

    if ( (pid=fork()) < 0)
        status = -1;

```

```

else if (pid == 0) {
    execl("/bin/sh", "sh", "-c", kdozeile, NULL);
    _exit(127);

} else
    while (waitpid(pid, &status, 0) < 0)
        if (errno != EINTR) {
            status = -1;
            break;
        }

return(status);
}

```

Programm 10.19 (**system.c**): Implementierung der Funktion **system** (ohne Signalbehandlung)

Das Programm 10.19 (**system.c**) ist eine Implementierung der Funktion **system** ohne Signalbehandlung.

In der Implementierung hier wird **sh** aufgerufen, um die Shell die angegebene `kdozeile` ausführen zu lassen. Die Angabe der Option **-c** bewirkt, daß die Shell nicht von der Standardeingabe liest, sondern nur die nach **-c** angegebene `kdozeile` ausführt.

Die Verwendung der Shell zur Ausführung der übergebenen `kdozeile` hat einige Vorteile:

- ▶ Die Shell übernimmt für uns die Aufteilung der `kdozeile` in einzelne Wörter (Argumente).
- ▶ In der `kdozeile` sind Shell-Metazeichen erlaubt; sie werden entsprechend von der aufgerufenen Shell interpretiert.

Im Kindprozeß wird **_exit** (und nicht **exit**) aufgerufen. So ist sichergestellt, daß am Ende des Kindprozesses eventuell vom Elternprozeß (bei **fork**) geerbte, aber noch nicht geleerte Puffer geleert werden, was ein Schreiben auf die betreffende Datei bewirken würde.

Beispiel

system-Aufruf in einem set-User-/Group-ID-Programm ist ein Sicherheitsloch

In einem Programm, für das das Set-User-ID-Bit gesetzt ist, sollte niemals **system** verwendet werden, denn dies stellt eine Sicherheitslücke dar.

Das einfache Programm 10.20 (**systprog.c**) verdeutlicht dies, indem es das auf einer Kommandozeile angegebene Programm mit einem **system**-Aufruf ausführen läßt.

```

#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int status;

```

```

if (argc != 2)
    fehler_meld(FATAL, "usage: %s progname", argv[0]);

if ( (status=system(argv[1])) < 0)
    fehler_meld(FATAL_SYS, "system-Fehler");
print_endestatus(status);

exit(0);
}

```

Programm 10.20 (systprog.c): Ausführung des auf Komandozeile angegebenen Programms mittels **system**

Das Programm 10.21 (ausg_uid.c) seinerseits gibt lediglich seine reale und effektive User-ID aus.

```

#include "eighdr.h"

int
main(void)
{
    printf("reale uid = %d, effektive uid = %d\n", getuid(), geteuid());
    exit(0);
}

```

Programm 10.21 (ausg_uid.c): Ausgabe der realen und effektiven User-ID

Nachdem man die beiden Programme einzeln kompiliert und gelinkt hat

```

cc -o ausg_uid ausg_uid.c fehler.c
cc -o systprog systprog.c endestat.c fehler.c

```

ergibt sich z.B. folgender Ablauf:

```

$ systprog ausg_uid
reale uid = 2021, effektive uid = 2021
Normale Beendigung; exit-Status=0
$ su                                [Als Superuser anmelden]
Password:                          [Superuser-Password hier eingeben]
# chown root systprog              [Superuser wird Eigentümer von systprog]
# chmod u+s systprog              [Superuser setzt set-uid-Bit für systprog]
# exit                             [wieder normaler Benutzer werden]
$ systprog ausg_uid
reale uid = 2021, effektive uid = 0  [hier ist das Sicherheitsloch]
Normale Beendigung; exit-Status=0
$

```

An diesem Ablaufbeispiel ist erkennbar, daß ein gesetztes Set-User-ID-Bit bei einem **system**-Aufruf erhalten bleibt. Die Sicherheitslücke besteht nun darin, daß **system** immer die Shell aufruft, um die angegebene Komandozeile auszuführen. Die Shell benutzt immer die in der Shellvariablen IFS (*Input Field Separator*) angegebenen Trennzeichen, um eine Komandozeile in einzelne Wörter zu zerteilen. Ältere Shell-Versionen setzten nun beim Shell-Aufruf die Variable IFS nicht auf ihre Default-Werte zurück. Böswillige Benut-

zer können sich dies zunutze machen, indem sie IFS entsprechend setzen. Beim **system**-Aufruf wird dann ein ganz anderes Programm ausgeführt. Um eine solche Sicherheitslücke zu vermeiden, sollten deshalb Programme, für die das Set-uid-ID oder Set-Group-ID-Bit gesetzt ist, niemals **system** verwenden, sondern einen **system**-Aufruf mittels **fork** (setzt Rechte zurück) und **exec** nachbilden.

10.7 Ändern der User-ID und Group-ID eines Prozesses

10.7.1 setuid und setgid – Ändern der realen und effektiven User-ID und Group-ID

Um die reale und effektive User-ID oder Group-ID zu ändern, stehen die beiden Funktionen **setuid** und **setgid** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Allerdings kann nicht jeder Prozeß nach Belieben die User-IDs bzw. Group-IDs ändern, sondern es gelten dabei die nachfolgenden Regeln, die hier zwar nur für die User-ID beschrieben sind, aber entsprechend auch für die Group-ID³ gelten:

1. Wenn der Prozeß Superuser-Rechte besitzt, so setzt **setuid** die reale User-ID, die effektive User-ID und die saved Set-User-ID auf **uid**. Normalerweise wird **setuid** vom **login**-Programm, das immer mit Superuser-Rechten abläuft, verwendet, um die drei User-IDs zu setzen, die sich dann nicht mehr ändern.
2. Wenn der Prozeß keine Superuser-Rechte besitzt, aber **uid** entweder gleich der realen User-ID oder aber gleich der saved Set-User-ID ist, so setzt **setuid** nur die effektive User-ID auf **uid**. Die reale User-ID und die saved Set-User-ID werden dabei nicht geändert. Diese Regel ist dann wichtig, wenn nach einem **exec**-Aufruf die effektive User-ID auf die reale User-ID oder auf die saved Set-User-ID gesetzt werden soll.
3. Ist keine der beiden vorherigen Bedingungen erfüllt, so wird **errno** auf **EPERM** gesetzt und -1 (für Fehler) zurückgegeben.

Tabelle 10.4 faßt die verschiedenen Änderungsarten der drei User-IDs zusammen:

3. Die Angaben für die saved Set-User-ID sind dabei nur gültig, wenn **_POSIX_SAVED_IDS** gesetzt ist, wie dies z.B. in SVR4 der Fall ist.

ID	<i>exec</i>		<i>setuid(uid)</i>	
	Set-User-ID-Bit nicht gesetzt	Set-User-ID-Bit gesetzt	Superuser	nicht privilegierte Benutzer
reale User-ID	unverändert	unverändert	wird uid	unverändert
effektive User-ID	unverändert	wird User-ID der Programmdatei	wird uid	wird uid
saved Set-User-ID	kopiert von effek- tiver User-ID	kopiert von effek- tiver User-ID	wird uid	unverändert

Tabelle 10.4: Unterschiedliche Arten, um die 3 User-IDs zu ändern

Hinweis

Bei einem **exec**-Aufruf wird der Wert der effektiven User-ID in die saved Set-User-ID kopiert. Diese Sicherungskopie bleibt dann erhalten, wenn **exec** bei gesetztem Set-User-ID-Bit die effektive User-ID auf die User-ID der Programmdatei setzt.

Mit den Funktionen **getuid** und **geteuid** können nur die momentanen Werte der realen User-ID und der effektiven User-ID erfragt werden. Das saved Set-User-ID-Bit kann nicht erfragt werden.

10.7.2 saved Set-User-ID-Bit – Zeitweises Ein-/Ausschalten des Set-User-ID-Mechanismus

Die saved Set-User-ID ermöglicht das zeitweise Ein- und Ausschalten des Set-User-ID-Mechanismus. Dies soll an folgendem Beispiel erläutert werden. Wir haben ein Programm `wahl` geschrieben, das Abstimmungen jeglicher Art erlaubt. Ruft ein anderer Benutzer dieses Programm auf, so soll dessen Stimme in einer Datei festgehalten werden. Diese Datei muß natürlich gegen fremden Zugriff geschützt sein, um manuelle Manipulationen des Wahlergebnisses zu unterbinden.

Um einem anderen Benutzer nun trotzdem die Möglichkeit zu geben, mittels `wahl` seine Stimme abzugeben, muß für die Programmdatei `wahl` das Set-User-ID-Bit gesetzt sein. Für das Programm `wahl` wären nun folgende Schritte möglich:

1. Der Eigentümer der Programmdatei `wahl` ist z.B. der Benutzer `wahlleiter`, der das Set-User-ID-Bit für die Datei `wahl` gesetzt hat. Wenn dann ein anderer Benutzer (z.B. `hans`) mittels **exec** das Programm `wahl` startet, so ergibt sich folgende Konstellation:

```
reale User-ID = hans
effektive User-ID = wahlleiter
saved Set-User-ID = wahlleiter
```

2. `wahl` merkt sich zunächst mittels `eff_pid = geteuid()` die effektive User-ID, bevor es `setuid(geteuid())` aufruft, was folgende Konstellation nach sich zieht:

```
reale User-ID = hans (unverändert)
effektive User-ID = hans
saved Set-User-ID = wahlleiter (unverändert)
```

Nun hat `wahl` die eigene ID auch als effektive User-ID, so daß es keinerlei bevorzugte Rechte für die Wahldateien besitzt, deren Eigentümer der `wahlleiter` ist.

3. Wenn ein Zugriff auf die geschützten Wahldateien erforderlich wird, führt `wahl` den Aufruf `setuid(eff_pid)` aus. Dieser Aufruf kann nur dann erfolgreich sein, wenn `eff_pid` gleich der saved Set-User-ID ist. Da dies hier der Fall ist, ergibt sich folgende Konstellation:

```
reale User-ID:    hans (unverändert)
effektive User-ID: wahlleiter
saved Set-User-ID: wahlleiter (unverändert)
```

Mit dieser effektiven User-ID ist es dem Programm `wahl`, das von `hans` gestartet wurde, möglich, in die geschützten Wahldateien von `wahlleiter` zu schreiben.

4. Nach dem Schreiben in die Wahldateien setzt `wahl` mittels `setuid(geteuid())` die effektive User-ID wieder zurück auf die reale User-ID. Somit ergibt sich dann folgendes Aussehen der IDs:

```
reale User-ID:    hans (unverändert)
effektive User-ID: hans
saved Set-User-ID: wahlleiter (unverändert)
```

Nun hat `wahl` wieder keinerlei bevorzugte Rechte für die Wahldateien, deren Eigentümer der `wahlleiter` ist.

Durch das saved Set-User-ID-Bit ist es möglich zu verhindern, daß ein Prozeß für seine ganze Ausführungszeit mit besonderen Rechten läuft, die ihm mittels der Set-User-ID gewährt wurden. Die saved Set-User-ID ermöglicht uns, nach Belieben diese Sonderrechte ein- und auszuschalten. Ohne saved Set-User-ID ist dies nicht möglich, denn dann ist nach dem Ausschalten des Set-User-ID-Mechanismus ein erneutes Einschalten nicht mehr möglich.

Das zuvor beschriebene Verfahren des Ein- und Ausschaltens funktioniert nicht, wenn das Programm `wahl` dem Superuser gehört, denn ein Aufruf von **`setuid`** mit Superuser-Rechten setzt alle drei User-IDs. Für diesen Spezialfall benötigt man eine eigene Routine, mit der sich nur die effektive User-ID setzen läßt.

10.7.3 **seteuid und setegid – Ändern der effektiven User-ID bzw. Group-ID**

Um nur die effektive User-ID bzw. Group-ID zu ändern, stellen sowohl BSD als auch SVR4 die beiden Funktionen **`seteuid`** und **`setegid`** zur Verfügung

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t uid);
int setegid(gid_t gid);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Ein nicht-privilegierter Benutzer kann seine effektive User-/Group-ID mit **seteuid**/**setegid** entweder auf seine reale User-/Group-ID oder auf seine saved Set-User-ID setzen.

Bei einem privilegierten Benutzer wird bei **seteuid**/**setegid** nur die effektive User-/Group-ID auf uid bzw. gid gesetzt. Darin unterscheiden sich diese beiden Funktionen von den Funktionen **setuid** und **setgid**, die alle drei User-/Group-IDs ändern.

Diese beiden POSIX.1-Funktionen **seteuid** und **setegid** setzen voraus, daß saved Set-User-IDs unterstützt werden.

10.7.4 setreuid und setregid – Vertauschen der realen und effektiven User-/Group-ID

4.4BSD stellt zum Vertauschen der realen und effektiven User- bzw. Group-ID die beiden Funktionen **setreuid** und **setregid** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

int setreuid(uid_t real_uid, uid_t eff_uid);
int setregid(gid_t real_gid, gid_t eff_gid);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Ein nicht-privilegierter Benutzer kann immer die reale und effektive User-ID bzw. Group-ID vertauschen. So ist es einem Programm, für das das Set-User-ID-Bit gesetzt ist, möglich, die Set-User-ID-Sonderrechte ein- und auszuschalten. Nur dem Superuser ist es erlaubt, für die Argumente andere Werte als die realen oder effektiven User-/Group-IDs anzugeben.

Wird für eines der beiden Argumente -1 ausgegeben, so soll hierfür die aktuelle ID verwendet werden.

Somit ist der Aufruf **setreuid(-1, uid)** äquivalent zum Aufruf **seteuid(uid)** und der Aufruf **setregid(-1, gid)** äquivalent zum Aufruf **setegid(gid)**.

Hinweis

SVR4 stellt diese beiden in der *BSD compatibility library* zur Verfügung.

10.7.5 Überblick über die unterschiedlichen Funktionen zum Setzen der User-IDs

Abbildung 10.4 zeigt alle zuvor beschriebenen Funktionen und ihre Auswirkung auf die einzelnen User-IDs im Überblick.

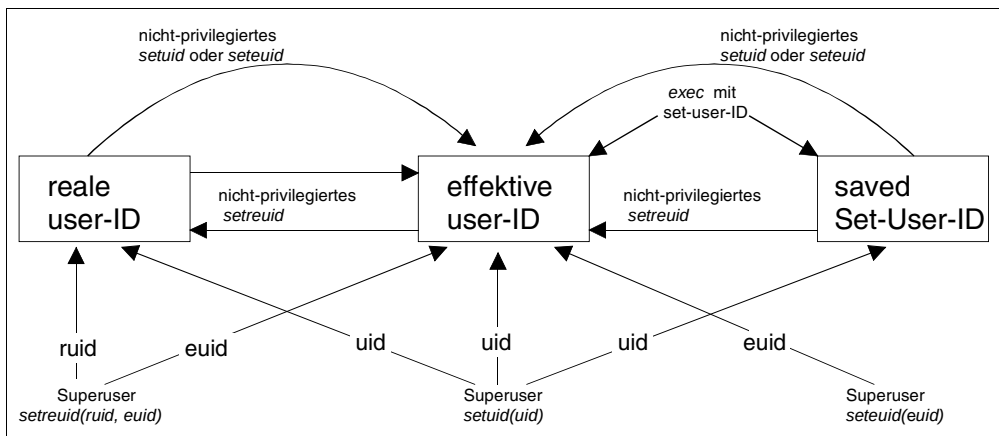


Abbildung 10.4: Zusammenfassung aller Funktionen zum Setzen der unterschiedlichen User-IDs

Die Abbildung 10.4 gilt entsprechend auch für Group-IDs, wobei hierbei nur die entsprechenden Funktionen zum Setzen von Group-IDs einzusetzen sind.

10.7.6 setfsuid und setfsgid – Setzen der User-/Group-IDs für Filesystemzugriffe unter Linux



In manchen Situationen kann es notwendig sein, daß ein Prozeß seine Superuser-Rechte für alle seine Aktionen, außer für Dateizugriffe, die er mit den Rechten eines normalen Benutzers durchzuführen wünscht, behalten möchte. Dazu stehen die beiden Funktionen **setfsuid** und **setfsgid** zur Verfügung:

```
#include <sys/types.h>
#include <unistd.h>

int setfsuid(uid_t fsuid);
```

gibt zurück: vorherige fsuid (bei Erfolg); übergebene fsuid bei Fehler

```
int setfsuid(gid_t fsgid);
```

gibt zurück: vorherige fsgid (bei Erfolg); übergebene fsgid bei Fehler

setfsuid setzt die User-ID, die der Linuxkern für die Prüfung aller Filesystemzugriffe verwendet. Für **fsuid** kann die reale User-ID, effektive User-ID, saved User-ID oder die aktuelle **fsuid** angegeben werden. Wann immer die effektive User-ID geändert wird, wird die **fsuid** des Prozesses auf den neuen Wert der effektiven User-ID gesetzt.

Die Funktion **setfsuid** wird normalerweise nur von Programmen wie dem Linux NFS Server aufgerufen, die eine eigene User-ID für Filesystemzugriffe benötigen, ohne daß sie ihre reale und effektive User-ID ändern. Würden solche Programme wie der Linux NFS Server in solchen Fällen ihre normalen User-IDs (z.B. mit **setreuid**) ändern, wäre dies eine Sicherheitslücke, da der Benutzer, der die Dienste des NFS-Servers in Anspruch nimmt, für eine kurze Zeit der Eigentümer des NFS-Serverprozesses wäre.

Für die Funktion **setfsuid** gilt das gleiche, nur daß diese Funktion auf die **fsgid** und die entsprechenden Group-IDs angewendet wird.



10.8 Informationen zu Prozessen

Hier wird gezeigt, welche unterschiedlichen Informationen man zu Prozessen erfragen kann.

10.8.1 times – Erfragen der von einem Prozeß verbrauchten Zeit

Um zu erfragen, wieviel Zeit (Uhrzeit und CPU-Zeit) ein Prozeß (einschließlich aller schon beendeten Kindprozesse) verbraucht hat, steht die Funktion **times** zur Verfügung.

```
#include <sys/times.h>
```

```
clock_t times(struct tms *cpu_zeit);
```

gibt zurück: seit Programmstart vergangene Uhrzeit (im Datentyp **clock_t**); -1 bei Fehler

Die Struktur **tms** hat folgende Komponenten:

```
struct tms {
    clock_t tms_utime; /* Benutzer-CPU-Zeit */
    clock_t tms_stime; /* System CPU-Zeit */
    clock_t tms_cutime; /* Benutzer-CPU-Zeit der beendet. Kindproz. */
    clock_t tms_cstime; /* System-CPU-Zeit der beendet. Kindproz. */
};
```

Die beiden Komponenten `tms_cutime` und `tms_cstime` enthalten nur die Werte von Kindprozessen, auf deren Beendigung mittels ***wait*** oder ***waitpid*** gewartet wurde.

Als Rückgabewert liefert ***times*** die seit irgendeinem in der Vergangenheit festgelegten Zeitpunkt vergangene Uhrzeit. Dieser zurückgegebene absolute Zeitwert ist meist wenig informativ, weswegen man auch nicht mit dieser absoluten, sondern mit der relativen seit Programmstart vergangenen Zeit arbeitet. Dies erreicht man dadurch, daß man zunächst einmal ***times*** aufruft und sich den dabei zurückgegebenen Wert in einer `clock_t`-Variablen festhält. Die Subtraktion eines von einem späteren ***times***-Aufruf erhaltenen Zeitwerts von diesem Anfangszeitwert liefert dann entsprechend die seit dem vorherigen Aufruf vergangene Uhrzeit. Um den im Datentyp `clock_t` enthaltenen Wert in Sekunden umzurechnen, muß dieser durch den Wert geteilt werden, den der Aufruf `sysconf(_SC_CLK_TCK)` als Rückgabewert liefert. Diese Rückgabewerte sind die am jeweiligen System eingestellten »Uhrticks« pro Sekunde. Anstelle von `sysconf(_SC_CLK_TCK)` könnte auch das von ANSI C vorgeschriebene Makro `CLOCKS_PER_SEC` verwendet werden.

Hinweis

BSD-Unix und SVR4 (im *BSD-compatibility package*) stellen mit ***getrusage*** eine Funktion zur Verfügung, die neben der verbrauchten CPU-Zeit noch viele weiteren Informationen zu einem Prozeß liefert, wie z.B. page-Faults oder Anzahl der empfangenen Signale.

Beispiel

Zeitmessung für die auf Kommandozeile angegebenen Programmen

Das Programm 10.22 (`timekdos.c`) läßt jedes auf der Kommandozeile angegebene Programm ausführen, mißt die entsprechenden von diesem Programm verbrauchten Zeiten und gibt sie aus.

```
#include <sys/times.h>
#include "eighdr.h"

static void print_zeiten(clock_t uhr_zeit,
                        struct tms *start_zeit, struct tms *ende_zeit);
static void kdo_ausfuehr(char *kdozeile);

int
main(int argc, char *argv[])
{
    int i;

    for (i=1; i<argc; i++) {
        printf("\n----- Kommando %d: ", i);
        kdo_ausfuehr(argv[i]);
    }
    exit(0);
}

static void kdo_ausfuehr(char *kdozeile)
{

```

```

struct tms  tmsstart, tmsende;
clock_t     start_uhr, ende_uhr;

printf("%s\n", kdozeile);

if ( (start_uhr = times(&tmsstart)) == -1) /* Startzeiten festhalten */
    fehler_meld(FATAL_SYS, "times-Fehler");

if (system(kdozeile) < 0)                  /* kdozeile ausfuehren    */
    fehler_meld(FATAL_SYS, "system-Fehler");

if ( (ende_uhr = times(&tmsende)) == -1) /* Endezeiten festhalten */
    fehler_meld(FATAL_SYS, "times-Fehler");

print_zeiten(ende_uhr-start_uhr, &tmsstart, &tmsende);
}

static void print_zeiten(clock_t uhr_zeit,
                        struct tms *start_zeit, struct tms *ende_zeit)
{
    double uhr_ticks;

    if ( (uhr_ticks = sysconf(_SC_CLK_TCK)) < 0)
        fehler_meld(FATAL_SYS, "sysconf-Fehler");
    printf("Uhrzeit:          %6.2f\n", uhr_zeit/uhr_ticks);
    printf("    Benutzer CPU-Zeit: %6.2f\n",
           (ende_zeit->tms_utime-start_zeit->tms_utime) / uhr_ticks);
    printf("    System CPU-Zeit: %6.2f\n",
           (ende_zeit->tms_stime-start_zeit->tms_stime) / uhr_ticks);
    printf("Kind-Benutzer CPU-Zeit: %6.2f\n",
           (ende_zeit->tms_cutime-start_zeit->tms_cutime) / uhr_ticks);
    printf("    Kind-System CPU-Zeit: %6.2f\n",
           (ende_zeit->tms_cstime-start_zeit->tms_cstime) / uhr_ticks);
}

```

Programm 10.22 (timekdos.c): Ausführen und Messen der Zeiten für die auf Kommandozeile angegebenen Programme

Nachdem man dieses Programm 10.22 (timekdos.c) kompiliert und gelinkt hat

```
cc -o timekdos timekdos.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ timekdos "find / -name hallo -print >/dev/null" pwd
```

```

----- Kommando 1: find / -name hallo -print >/dev/null
Uhrzeit:          62.50
    Benutzer CPU-Zeit:  0.00
    System CPU-Zeit:   0.00
Kind-Benutzer CPU-Zeit: 1.27
    Kind-System CPU-Zeit: 8.98

```

```

----- Kommando 2: pwd
/home/hh/sysprog/kap10
Uhrzeit:          0.49
  Benutzer CPU-Zeit: 0.00
    System CPU-Zeit: 0.00
Kind-Benutzer CPU-Zeit: 0.03
  Kind-System CPU-Zeit: 0.11
$ timekdos date "sleep 7" "who am i"

----- Kommando 1: date
Tue Jun 27 11:31:32 MET DST 1995
Uhrzeit:          0.17
  Benutzer CPU-Zeit: 0.00
    System CPU-Zeit: 0.01
Kind-Benutzer CPU-Zeit: 0.02
  Kind-System CPU-Zeit: 0.14

----- Kommando 2: sleep 7
Uhrzeit:          7.16
  Benutzer CPU-Zeit: 0.00
    System CPU-Zeit: 0.00
Kind-Benutzer CPU-Zeit: 0.01
  Kind-System CPU-Zeit: 0.14

----- Kommando 3: who am i
hh      tty2      Jun 27 11:25
Uhrzeit:          0.44
  Benutzer CPU-Zeit: 0.00
    System CPU-Zeit: 0.01
Kind-Benutzer CPU-Zeit: 0.03
  Kind-System CPU-Zeit: 0.15
$

```

In beiden Ablaufbeispielen verbrauchen die Kindprozesse die CPU-Zeit. Der Grund hierfür ist, daß zur Ausführung der auf der Kommandozeile angegebenen Kommandos von **system** eine Shell als Kindprozeß kreiert wird.

10.8.2 getlogin – Erfragen des Namens des Prozeßeigentümers

Um den Loginnamen des Benutzers zu erfragen, der das gerade ablaufende Programm gestartet hat, gibt es verschiedene Möglichkeiten:

1. Aufruf von `getpwuid(getuid())`

Der Nachteil dieses Aufrufs ist, daß er den falschen Namen liefern kann, wenn ein Benutzer mehrere Loginnamen (mit der gleichen UID) besitzt. Dieser Fall tritt z.B. dann auf, wenn Benutzer mit unterschiedlichen Shells und Environments arbeiten wollen. Sie lassen sich dann für die verschiedenen Loginshells jeweils einen eigenen Loginnamen einrichten (wie z.B. hh für Bourne-Shell und hhc für C-Shell).

2. Lesen der Environment-Variablen LOGNAME

Beim Anmelden wird immer die Environment-Variable LOGNAME gesetzt. Um den Loginnamen zu einem Prozeß zu erfahren, muß also nur der Inhalt von LOGNAME gelesen werden. Der Nachteil dieser Vorgehensweise ist, daß ein Benutzer die Environment-Variable LOGNAME beliebig verändern kann und somit nicht garantiert ist, daß LOGNAME den wirklichen Loginnamen enthält.

Da keine der beiden Möglichkeiten mit absoluter Sicherheit den richtigen Loginnamen garantieren kann, wurde die Funktion **getlogin** zur Verfügung gestellt.

```
#include <unistd.h>
```

```
char *getlogin(void);
```

gibt zurück: Adresse des Loginnamen-Strings (bei Erfolg); NULL bei Fehler

getlogin liefert als Rückgabewert NULL (für Fehler), wenn der Prozeß nicht einem Terminal zugeordnet ist, an dem der entsprechende Benutzer gerade angemeldet ist. Solche Prozesse werden als Dämonprozesse (*daemons*) bezeichnet.

10.8.3 Buchführung bei Prozessen (process accounting)

Zur Buchführung von Prozessen gibt es keine Vorgaben durch Standards. Um die Buchführung ein- oder auszuschalten, bieten sowohl SVR4 als auch BSD-Unix das Kommando **accton** an, das seinerseits die Funktion **acct** (siehe auch Manpages) zum Ein- und Ausschalten der Buchführung aufruft.

Zum Einschalten der Buchführung muß der Superuser **accton** mit einem Pfadnamen aufrufen. Der Pfadname ist dabei üblicherweise `/var/adm/pacct` (oder `/usr/adm/acct` auf älteren Systemen). Gibt der Superuser beim Aufruf von **accton** keine Argumente an, so wird die Buchführung ausgeschaltet.

Ist die Prozeßbuchführung eingeschaltet, so macht der Kern bei jeder Beendigung eines Prozesses einen entsprechenden Eintrag in seiner »Buchführungsdatei«. Ein solcher Eintrag setzt sich üblicherweise aus 32 Byte binären Daten zusammen, in denen sich der Kommandoname, verbrauchte CPU-Zeit, User-ID, Group-ID usw. befindet.

Die Struktur für einen solchen Eintrag ist in `<sys/acct.h>` wie folgt definiert:

```
typedef u_short comp_t; /* ersten 3 Bits: Exponent zur Basis 8*/
                        /* letzten 13 Bits: Mantisse          */
struct acct {
    char ac_flag; /* mögliche Werte
        AFORK      Prozeß wurde durch fork kreiert,
                   hat aber niemals exec aufgerufen.
        ASU        Prozeß hatte Superuser-Rechte.
        ACOMPAT    Prozeß lief im Kompatibilitätsmodus
                   (nur auf VAX).
        ACORE      Prozeß wurde mit Erzeugung einer
```

```

                                core-Datei beendet (nicht in SVR4).
                                AXSIG   Prozeß wurde durch Signal beendet
                                (nicht SVR4).
char ac_stat; /* Beendigungsstatus (nicht in BSD) */
uid_t ac_uid, /* reale User-ID */
gid_t ac_gid; /* reale Group-ID */
dev_t ac_tty; /* Kontrollterminal */
time_t ac_btime; /* Startzeit (als Kalenderzeit) */
comp_t ac_utime; /* verbrauchte Benutzer-CPU-Zeit (Uhrticks) */
comp_t ac_stime; /* verbrauchte System-CPU-Zeit (Uhrticks) */
comp_t ac_etime; /* Lebensdauer (in Uhrticks) */
comp_t ac_mem; /* durchschnittliche Speicherbenutzung */
comp_t ac_io; /* Anzahl gelesener und geschriebener Bytes */
comp_t ac_rw; /* Anzahl gelesener oder geschriebener Blöcke */
char ac_comm[8]; /* Kommandoname: [8] in SVR4; [10] in BSD */
};

```

Für jeden Prozeß hält sich der Kern diese Information in der Prozeßtabelle und schreibt sie bei Beendigung eines Prozesses in seine »Buchführungsdatei«. Die Einträge in der »Buchführungsdatei« geben somit die Reihenfolge der Beendigung der einzelnen Prozesse wieder, nicht die Reihenfolge ihres Starts.

Hinweis

Nur für Prozesse (nicht für Programme) findet ein Eintrag in der »Buchführungsdatei« statt. Nur bei einem **fork** erzeugt der Kern in der Prozeßtabelle einen neuen Buchführungseintrag, nicht dagegen bei einem **exec**. Bei einem **exec** wird lediglich in dem schon existierenden Eintrag der Kommandoname geändert und das Flag **AFORK** gelöscht. Wenn also ein Prozeß sich mehrmals mit **exec** »überlagert«, so steht nur der Name des zuletzt mit **exec** gestarteten Programms in `ac_comm`, während aber z.B. die CPU-Zeiten von allen **exec**-Programmen addiert wurden.

Beispiel

Erzeugen von 4 Kindprozessen mit unterschiedlichen Aktionen

Das folgende Programm 10.23 (`vierkind.c`) ruft **fork** viermal auf, wobei die einzelnen Kind- und Enkelprozesse unterschiedliche Dinge tun (siehe auch Abbildung 10.5).

```

#include <signal.h>
#include "eighdr.h"

int
main(void)
{
    pid_t pid;

    if ( (pid=fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid > 0) { /* _____ Elternprozess _____ */
        sleep(3);
        exit(3);
    }
}

```

```

}
/*_____ 1. Kind _____*/
if ( (pid=fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");
else if (pid > 0) {
    sleep(6);
    abort(); /* Ende mit core */
}
/*_____ 2. Kind _____*/
if ( (pid=fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");
else if (pid > 0)
    execl("/bin/cp", "cp", "/etc/passwd", "/dev/null", NULL);

/*_____ 3. Kind _____*/
if ( (pid=fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");
else if (pid > 0) {
    sleep(10);
    exit(0); /* Normales Ende */
}
/*_____ 4. Kind _____*/
sleep(8);
kill(getpid(), SIGKILL); /* Ende durch Signal (ohne core) */
}

```

Programm 10.23 (**vierkind.c**): Erzeugen von 4 Kindprozessen mit unterschiedlichen Aktionen

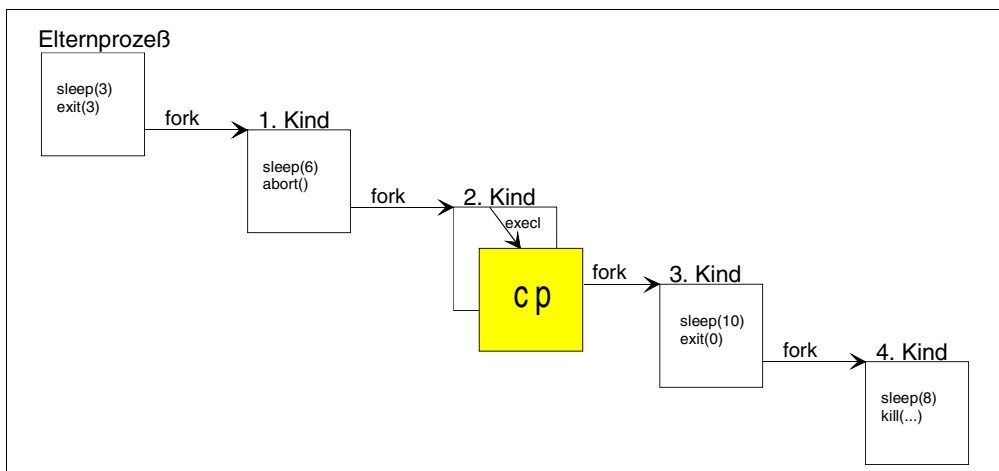


Abbildung 10.5: Prozeßstruktur zum Programm 10.23 (**vierkind.c**)

Beispiel*Ausgeben von Buchführungsinformationen*

Das Programm 10.24 (acctinfo.c) liest die Einträge aus jeder entsprechenden Buchführungsdatei und gibt sie aufbereitet wieder aus.

```
#include <sys/types.h>
#include <sys/acct.h>
#include "eighdr.h"

#define BUCH_DATEI "/var/adm/pacct"

static unsigned long compt_interpret(comp_t comp_inhalt);

int
main(void)
{
    struct acct    ac_eintrag;
    FILE          *dz;

    if ( (dz = fopen(BUCH_DATEI, "r")) == NULL)
        fehler_meld(FATAL_SYS, "kann %s nicht oeffnen", BUCH_DATEI);

    while (fread(&ac_eintrag, sizeof(ac_eintrag), 1, dz) == 1) {
        printf("%-10s lebensdauer=%7lu bytettransfer=%7lu %c %c ",
            ac_eintrag.ac_comm,
            compt_interpret(ac_eintrag.ac_etime),
            compt_interpret(ac_eintrag.ac_io),
            (ac_eintrag.ac_flag & AFORK) ? 'F' : ' ',
            (ac_eintrag.ac_flag & ASU)  ? 'S' : ' ');
#ifdef ACORE
        printf("%c ", (ac_eintrag.ac_flag & ACORE) ? 'C' : ' ');
#else
        printf("  ");
#endif
#ifdef AXSIG
        printf("%c ", (ac_eintrag.ac_flag & AXSIG) ? 'X' : ' ');
#else
        printf("  ");
#endif
        printf("\n");
    }
    if (ferror(dz))
        fehler_meld(FATAL_SYS, "Fehler beim Lesen aus %s", BUCH_DATEI);
    exit(0);
}

static unsigned long compt_interpret(comp_t comp_inhalt)
{
    unsigned long wert;
    int          exponent;
```

```

wert      = comp_inhalt & 0x1fff;      /* 13-Bit Mantisse */
exponent = (comp_inhalt >> 13) & 0x7; /* 3-Bit Exponent */
while (exponent-- > 0)
    wert *= 8;
return(wert);
}

```

Programm 10.24 (acctinfo.c): Ausgeben von Buchführungsinformationen

Nachdem beide Programme kompiliert und gelinkt wurden

```

cc -o vierkind vierkind.c fehler.c
cc -o acctinfo acctinfo.c fehler.c

```

muß die folgende Vorgehensweise gewählt werden:

1. Der Superuser muß die Buchführung mit **accton** einschalten, wie z.B.:
`/usr/lib/acct/accton /var/adm/pacct`
2. Aufruf des Programms `vierkind`.
 Der Aufruf sollte fünf Einträge in der Buchführungsdatei `/var/adm/pacct` vornehmen, nämlich einen für den Elternprozeß und vier für die Kindprozesse.
3. Der Superuser sollte die Buchführung mit **accton** (ohne weitere Argumente) wieder ausschalten.
4. Aufruf des Programms `acctinfo` zur Ausgabe der entsprechenden Buchführungsdaten.

10.9 Übung

10.9.1 Kreieren eines Zombies

Erstellen Sie ein Programm `pszombie.c`, das einen Zombieprozeß kreiert und dann **system** aufruft, um mittels des Kommandos **ps** diesen Zombieprozeß anzuzeigen.

10.9.2 Ausgeben der Ziffern von Zahlen als Wörter

Erstellen Sie ein Programm `zahlwort.c`, bei dem der Elternprozeß immer **eine** Zufallszahl in eine Datei `zahlen` schreibt. Der Kindprozeß soll diese Zahl dann lesen und deren Ziffer als Wörter ausgeben. Danach soll der Elternprozeß die nächste Zufallszahl in die Datei schreiben, der Kindprozeß diese wieder lesen und deren Ziffernörter ausgeben usw. Da das Schreiben und Lesen der Zahl immer abwechselnd durch Eltern- und Kindprozeß erfolgen soll, ist hier eine Synchronisation dieser beiden Prozesse notwendig. Erst wenn der Elternprozeß seine Zahl in die Datei geschrieben hat, kann der Kindprozeß sie lesen. Umgekehrt kann natürlich auch der Elternprozeß erst dann die nächste Zahl in diese Datei schreiben, wenn der Kindprozeß die vorherige Zahl daraus gelesen hat, andernfalls würde die vorherige Zahl überschrieben, ohne daß sie vom Kindprozeß gelesen wurde. Das Ende soll der Elternprozeß dem Kindprozeß mitteilen, indem er die Zahl `-1` schreibt.

Nachdem man dieses Programm `zahlwort.c` (hier für 20 Zahlen ausgelegt) kompiliert und gelinkt hat

```
cc -o zahlwort zahlwort.c forksync.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ zahlwort
747672578 = sieben vier sieben sechs sieben zwei fuenf sieben acht
760587229 = sieben sechs null fuenf acht sieben zwei zwei neun
781049034 = sieben acht eins null vier neun null drei vier
370562387 = drei sieben null fuenf sechs zwei drei acht sieben
553943786 = fuenf fuenf drei neun vier drei sieben acht sechs
711548410 = sieben eins eins fuenf vier acht vier eins null
202904927 = zwei null zwei neun null vier neun zwei sieben
1641403514 = eins sechs vier eins vier null drei fuenf eins vier
726133004 = sieben zwei sechs eins drei drei null null vier
309649901 = drei null neun sechs vier neun neun null eins
565969618 = fuenf sechs fuenf neun sechs neun sechs eins acht
1306626638 = eins drei null sechs sechs zwei sechs sechs drei acht
1169451312 = eins eins sechs neun vier fuenf eins drei eins zwei
215036031 = zwei eins fuenf null drei sechs null drei eins
1674484634 = eins sechs sieben vier vier acht vier sechs drei vier
962495730 = neun sechs zwei vier neun fuenf sieben drei null
1461715828 = eins vier sechs eins sieben eins fuenf acht zwei acht
843414693 = acht vier drei vier eins vier sechs neun drei
287193971 = zwei acht sieben eins neun drei neun sieben eins
1497349177 = eins vier neun sieben drei vier neun eins sieben sieben
---- Kindprozeß fertig ----
---- Elternprozeß fertig ----
$
```

10.9.3 Vorsicht bei Aufruf von `vfork` in einer anderen Funktion als `main`

Das folgende Programm `vforkfal.c` ruft **`vfork`** in einer anderen Funktion als `main` auf. Kann dies zu Problemen führen? Wenn ja, warum?

```
#include <sys/types.h>
#include "eighdr.h"

static void a(void),
           b(void);

int
main(void)
{
    a();
    b();
    _exit(0);
}
```

```

static void a(void)
{
    pid_t  pid;

    if ( (pid = vfork()) < 0)
        fehler_meld(FATAL_SYS, "vfork-Fehler");
    /*- Sowohl Kind als auch Elternprozess
       kehren von dieser Funktion zurueck */
}

static void b(void)
{
    char   zahlen[100]; /* automatic-Variablen auf Stack */
    int    i;

    for (i=0; i<sizeof(zahlen); i++)
        zahlen[i] = i;
}

```

Programm 10.25 (**vforkfal.c**): Aufruf von **vfork** in einer anderen Funktion als *main*

10.9.4 Erfragen der eigenen saved Set-User-ID durch einen Prozeß

Wie kann ein Prozeß seine eigene *saved Set-User-ID* erfahren?

10.9.5 Ausgeben der Prozeßhierarchie in Baumform

Erstellen Sie ein Programm `prozhier.c`, das eine Reihe von Kindprozessen, Enkelprozessen usw. kreierte. Alle Prozesse sollen ihre PID und die PID ihres Elternprozesses in eine gemeinsame Datei schreiben. Am Ende des Programms soll dann der Elternprozeß die PIDs aus dieser Datei lesen, und die beim Programmablauf vorliegende Prozeßstruktur in Baumform ausgeben.

Nachdem man das Programm `prozhier.c` kompiliert und gelinkt hat

```
cc -o prozhier prozhier.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

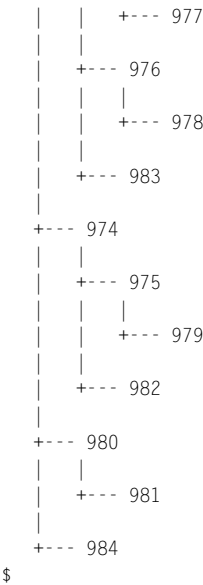
$ prozhier
Prozesshierarchie fuer dieses Programm
=====

```

```

969
|
+--- 970
| |
| | +--- 971
| | |
| | | +--- 972
| | | |
| | | | +--- 973
| | | |
| | |
| |
|

```



11 Attribute eines Prozesses (Kontrollterminal, Prozeßgruppe und Session)

*Nicht die Blumen und Bäume,
nur der Garten ist unser Eigentum.*

Chinesisches Sprichwort

Zunächst wird in Kapitel 11 auf die bei einem Login ablaufenden Prozesse eingegangen. Dabei wird zwischen Terminal-Logins und Netzwerk-Logins unterschieden. Des Weiteren werden Kontrollterminals und die von POSIX.1 eingeführten Sessions vorgestellt. Auch wird ein detaillierter Einblick in die von vielen Shells angebotene Jobkontrolle und die dabei ablaufenden Mechanismen gegeben.

11.1 Loginprozesse

Nachfolgend werden die bei einem Login ablaufenden Prozesse beschrieben. Es wird dabei zwischen Terminal-Logins und Netzwerk-Logins unterschieden.

11.1.1 Terminal-Logins

init-Prozeß und /etc/ttys

Beim Booten eines Systems kreiert der Kern immer den `init`-Prozeß (mit der Prozeß-ID 1). Dieser Prozeß liest unter anderem die Datei `/etc/ttys`, in der sich für jedes Terminal, von dem ein Login möglich ist, eine eigene Zeile befindet. Eine solche Zeile enthält oft neben dem Gerätenamen des Terminals weitere Angaben, wie z.B. die Baudrate.

Für jede in `/etc/ttys` angegebene Zeile kreiert nun `init` mittels **`fork`** einen Kindprozeß, der sich mittels eines **`exec`**-Aufrufs mit dem Programm **`getty`** überlagert. Abbildung 11.1 verdeutlicht dies.

Alle von `init` kreierten und mit **`getty`** überlagerten Kindprozesse haben dabei 0 als reale und als effektive User-ID, was bedeutet, daß sie Superuser-Rechte besitzen. Das Environment dieser Kindprozesse ist leer.

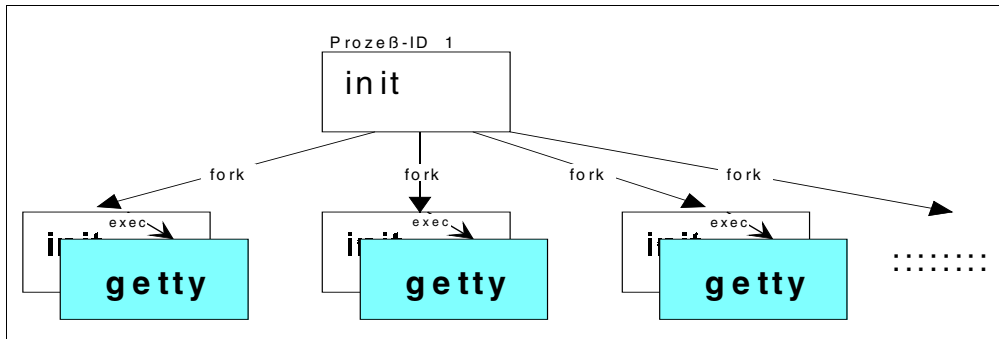


Abbildung 11.1: *init* kreiert für */dev/tty*-Einträge Kindprozesse, die mit **getty** überlagert werden

getty startet das *login*-Programm

getty öffnet nun mit *open* die Gerätedatei des entsprechenden Terminals¹, so daß ein Lesen und Schreiben möglich ist. Dazu richtet **getty** die Filedesktoren 0, 1 und 2 ein. Danach fordert **getty** mit der Ausgabe

login:

zum Anmelden auf. Nachdem der Benutzer hier seinen Loginnamen eingegeben hat, ruft **getty** das *login*-Programm mit einem *exec*-Aufruf auf, wie z.B.

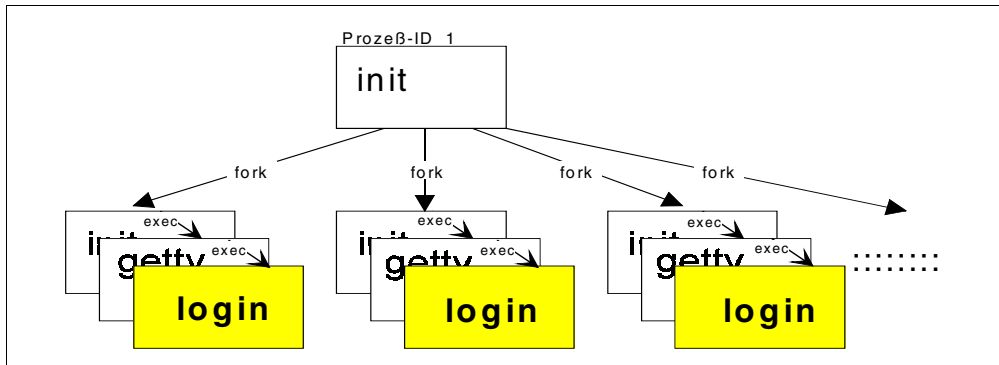
```
execle("/usr/bin/login", "login", "-p", Loginname, NULL, envp);
```

Während *init* noch **getty** mit einem leeren Environment aufrief, kreiert **getty** seinerseits ein Environment für den Loginprozeß (das Argument *envp*). Dieses Environment umfaßt unter anderem die Variable *TERM*, die mit dem entsprechenden Namen des Terminals gesetzt wird. Die Werte für *TERM* und die anderen Environment-Variablen werden dabei aus der Datei *gettytab* gelesen.

Die Option *-p* beim *execle* legt fest, daß das übergebene Environment aufzuheben ist und eventuell erweitert werden kann, aber nicht überschrieben werden darf. Abbildung 11.2 zeigt die Situation, die vorliegt, nachdem *login* aufgerufen wurde.

Alle Prozesse in Abbildung 11.2 haben Superuser-Rechte, da der *init*-Prozeß, der sie kreierte, ebenfalls Superuser-Rechte besitzt. Die Elternprozeß-ID von **getty** und *login* ist jeweils 1, da diese sich durch einen *exec*-Aufruf nicht ändert.

1. Wenn das Gerät ein Modem ist, dann wird *open* im entsprechenden Gerätetreiber verzögert, bis das Modem angewählt wurde und den Anruf beantwortet hat.

Abbildung 11.2: Situation nach dem Aufruf von **login**

Das **login**-Programm

Das **login**-Programm ruft zunächst **getpwnam**, um das zum zuvor eingegebenen Loginnamen gehörende verschlüsselte Paßwort (aus der Paßwortdatei) zu erfahren. Danach ruft **login** die Funktion **getpass** auf, um den Text

Password:

auszugeben und das Paßwort (unsichtbar) einzulesen. Dieses eingegebene Paßwort legt es dann **crypt** zum Verschlüsseln vor. Den verschlüsselten String vergleicht **login** mit der Komponente `pw_passwd` aus der Paßwortdatei (erhalten durch **getpwnam**). Falls ein nicht gültiges Paßwort eingegeben wurde, beendet sich **login** mit `exit(1)`, worauf der Elternprozeß (**init**) wieder mit **fork** einen neuen Kindprozeß kreiert, der sich mittels **exec** mit dem Programm **getty** überlagert, so daß die ganze Prozedur wieder von Anfang an beginnt.

War aber das eingegebene Paßwort richtig, so wechselt **login** mittels **chdir** ins entsprechende Home-Directory, trägt dann mittels **chown** den Benutzer, der sich gerade anmeldet, als neuen Eigentümer und Gruppen-Eigentümer für die Terminalgerätedatei ein und ändert dann die Zugriffsrechte für die Terminalgerätedatei auf 620 (`rw--w----`). Danach setzt **login** mit den Funktionen **setgid** und **initgroups** die Gruppen-IDs, bevor es die entsprechenden Environment-Variablen mit der zu diesem Zeitpunkt verfügbaren Information setzt:

```
HOME
SHELL
USER, LOGNAME
PATH
```

Schließlich ändert **login** mit **setuid** die User-IDs (reale, effektive, saved Set-User-ID), bevor es die entsprechende Loginshell aufruft, wie z.B.:

```
execl("/bin/sh", "-sh", NULL);
```

Das Minuszeichen zeigt bei diesem Aufruf an, daß es sich um eine Loginshell handelt.

Hier wurden nur die wichtigsten Punkte erwähnt, die von **login** ausgeführt werden. **login** führt daneben noch eine Vielzahl anderer Aktionen aus, wie z.B. Ausgabe der aktuellen Tagesmeldung (*message of the day*) aus der Datei */etc/motd* oder das Überprüfen, ob der entsprechende Besitzer neue *mail* erhielt.

Die Loginshell

Nachdem sich das **login**-Programm mit der Loginshell überlagert hat, beginnt die Loginshell die entsprechenden Startup-Dateien (*.profile* bei Bourne- und Korn-Shell, *.cshrc* und *.login* bei C-Shell) zu lesen. Danach gibt die Loginshell das Shell-Promptzeichen aus und wartet auf Eingaben durch den Benutzer.

Die Loginshell hat weiterhin den *init*-Prozeß als Elternprozeß. Wenn sich also die Loginshell beendet, so wird *init* mit dem Signal *SIGCHLD* davon informiert, und es startet die ganze Loginprozedur für dieses Terminal wieder von Beginn an.

Terminal-Login mit *ttymon* (SVR4)

SVR4 bietet eine neue Möglichkeit von Logins, die **ttymon**-Logins. Normalerweise wird in SVR4 das zuvor beschriebene **getty**-Loginverfahren für die Konsole und das **ttymon**-Verfahren für andere Terminals benutzt.

ttymon ist Teil der sogenannten *Service Access Facility* (SAF).

Hierbei ist *init* der Elternprozeß von *sac* (*service access controller*), der ein **fork**, gefolgt von einem anschließenden **exec**, mit dem **ttymon**-Programm durchführt, wenn das System in den Multi-User-Modus überwechselt.

ttymon überwacht alle Terminalports, die in seiner Konfigurationsdatei angegeben sind, und kreiert mittels **fork** einen Kindprozeß, wenn an einem Terminal ein Loginname eingegeben wird. Dieser Kindprozeß von **ttymon** startet dann mit **exec** das **login**-Programm, das das Paßwort einliest. War das eingegebene Paßwort richtig, so startet **login** mit **exec** die Loginshell.

Ein wichtiger Unterschied zum **getty**-Verfahren ist, daß hier **ttymon** und nicht *init* der Elternprozeß der Loginshell ist.

11.1.2 Netzwerk-Logins

Anders als bei Terminal-Logins, wo *init* die angeschlossenen Terminalgerätedateien kennt und für jede einen **getty**-Prozeß startet, ist bei Netzwerk-Logins nicht im voraus bekannt, wie viele Loginanforderungen auftreten werden. Bei Netzwerk-Logins kommen nämlich die Loginanforderungen über die Netzwerktreiber im Kern, so daß man einen Prozeß benötigt, der auf eine Anforderung zu einer Netzwerkverbindung wartet. Üblicherweise heißt dieser Prozeß *inetd* (manchmal auch *Internet superserver* genannt).

Nachfolgend sind die wichtigsten Schritte von Netzwerk-Logins aufgezeichnet.

init startet mittels /etc/rc den Dämonprozeß inetd

Beim Systemstart ruft `init` unter anderem die Shell auf, um das Shellskript `/etc/rc` ausführen zu lassen. Dieses Shellskript startet eine Reihe von Dämonprozessen (siehe Kapitel 16), unter anderem auch den Dämonprozeß `inetd`. Wenn sich das Shellskript beendet, so wird `init` der neue Elternprozeß von `inetd`.

inetd wartet auf TCP/IP-Verbindungsanforderungen

`inetd` wartet auf Verbindungsanforderungen. Trifft eine solche Anforderung ein, so kreiert `inetd` mittels **`fork`** einen Kindprozeß, der sich dann mit **`exec`** mit dem geeigneten Programm zur Behandlung dieser Anforderung überlagert.

inetd startet bei Verbindungsanforderungen die geeigneten Programme

Wenn z.B. eine TCP-Verbindungsanforderung für den TELNET-Server eintrifft, dann überlagert sich der kreierte Kindprozeß mit dem Programm `telnetd`.

TELNET ist ein Programm, das unter Verwendung des TCP-Protokolls Logins an entfernten Stationen in einem Netzwerk ermöglicht. Ein Benutzer kann sich dabei von seiner lokalen Station mit

```
telnet hostname
```

an einer anderen Station (*hostname*) anmelden. Der Aufrufer dieses Kommandos ist der sogenannte Client, der eine TCP-Verbindung zu *hostname* aufbaut. Auf *hostname* wird dann als Programm der sogenannte TELNET-Server gestartet. Nun können der Client und der Server über die TCP-Verbindung miteinander kommunizieren, da nun der Benutzer, der das Client-Programm startete, am Server *hostname* angemeldet wird.

Der `telnetd`-Prozeß öffnet nun ein sogenanntes Pseudoterminal und kreiert mit **`fork`** einen Kindprozeß. Während der Elternprozeß für die Kommunikation in der Netzwerkverbindung zuständig ist, startet der Kindprozeß mit **`exec`** das **`login`**-Programm. Der Eltern- und der Kindprozeß sind über das Pseudoterminal, auf dem beide über die Filedeskriptoren 0, 1 und 2 lesen und schreiben können, miteinander verbunden. Bei einem erfolgreichen Login führt **`login`** die gleichen Schritte wie bei einem Terminal-Login aus, bevor es sich mittels **`exec`** mit der entsprechenden Loginshell überlagert.

Anders als bei Terminal-Logins ist die Loginshell mit einem sogenannten Pseudoterminal verbunden.

SVR4-Besonderheit

Unter SVR4 entsprechen Netzwerk-Logins weitgehend den zuvor beschriebenen Schritten. Es wird derselbe `inetd`-Server benutzt, nur hat dieser nicht `init`, sondern `sac` (*service access controller*) als Elternprozeß.

11.2 Prozeßgruppen

Jeder Prozeß ist Mitglied einer Prozeßgruppe. Zu einer Prozeßgruppe können ein oder mehrere Prozesse gehören.

11.2.1 Prozeßgruppen-ID

Jede Prozeßgruppe hat eine eindeutige Prozeßgruppen-ID.

Prozeßgruppen-IDs ähneln den Prozeß-IDs: Sie sind positive ganze Zahlen und können im Datentyp `pid_t` gespeichert werden.

11.2.2 Prozeßgruppenführer (process group leader)

Jede Prozeßgruppe kann einen Prozeßgruppenführer haben. Ihn erkennt man daran, daß seine Prozeßgruppen-ID gleich seiner Prozeß-ID ist. Ein Prozeßgruppenführer kann eine Prozeßgruppe und auch Prozesse in dieser Prozeßgruppe kreieren.

11.2.3 Lebensdauer einer Prozeßgruppe

Eine Prozeßgruppe hört immer erst dann auf zu existieren, wenn sie keine Mitglieder mehr hat. Dies bedeutet, daß die Prozeßgruppe selbst dann weiterlebt, wenn sich der Prozeßgruppenführer beenden sollte, aber weitere Prozesse in dieser Gruppe vorhanden sind.

Die Lebensdauer einer Prozeßgruppe erstreckt sich somit von ihrer Kreierung (durch den Prozeßgruppenführer) bis zu dem Zeitpunkt, an dem der letzte Prozeß diese Gruppe verläßt. Verlassen einer Gruppe ist dabei durch die Beendigung oder aber dem Wechseln in eine andere Gruppe möglich.

11.2.4 `getpgp`/`getpgid` – Erfragen der Prozeßgruppen-ID

Um die Prozeßgruppen-ID eines Prozesses zu ermitteln, steht die Funktion **`getpgp`** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgp(void);
```

gibt zurück: Prozeßgruppen-ID des aufrufenden Prozesses

Neben der Funktion **getprgrp** steht mit **getpgid** noch eine weitere Funktion zum Erfragen der Prozeßgruppen-ID zur Verfügung:

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
pid_t getpgid(pid_t pid);
```

gibt zurück: Prozeßgruppen-ID (bei Erfolg); -1 bei Fehler

getpgid gibt die Prozeßgruppen-ID des Prozesses mit der Prozeß-ID *pid* zurück. Wird für *pid* der Wert 0 angegeben, liefert **getpgid** die Prozeßgruppen-ID des aktuellen Prozesses zurück. Somit ist der Aufruf **getpgid(0)** äquivalent zum Aufruf **getprgrp(0)**.

Für den Aufruf von **getpgid** sind keine besonderen Rechte erforderlich, da es jeden Prozeß erlaubt sein soll, die Prozeßgruppe zu erfragen, zu der irgendein anderer Prozeß gehört.

11.2.5 setpgid – Setzen der Prozeßgruppen-ID

Um Mitglied einer existierenden Prozeßgruppe zu werden oder eine neue Prozeßgruppe zu kreieren, steht einem Prozeß die Funktion **setpgid** zur Verfügung.

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
int setpgid(pid_t pid, pid_t pgid);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

setpgid setzt die Prozeßgruppen-ID des Prozesses *pid* auf *pgid*. Sind die beiden Argumente *pid* und *pgid* gleich, so wird der Prozeß mit *pid* der Prozeßgruppenführer.

Ein Prozeß kann **setpgid** nur für sich selbst oder aber einen seiner Kindprozesse aufrufen. Hat allerdings ein Kindprozeß **exec** aufgerufen, so darf er dessen Prozeßgruppen-ID nicht mit **setpgid** ändern.

Wird für *pid* der Wert 0 angegeben, so wird hierfür die Prozeß-ID des Aufrufers verwendet. Wird für *pgid* der Wert 0 angegeben, so wird der Prozeß in eine neue Prozeßgruppe eingefügt, die als Prozeßgruppen-ID die angegebene *pid* erhält. Der Prozeß mit der Prozeß-ID *pid* ist dann auch der Prozeßgruppenführer dieser neuen Prozeßgruppe.

Hinweis

Wenn das entsprechende System über keine Jobkontrolle (siehe Kapitel 11.5) verfügt, so liefert diese Funktion immer -1 (für Fehler) und setzt *errno* auf *ENOSYS*. Auf Systemen, die Jobkontrolle anbieten, ist immer die Konstante *_POSIX_JOB_CONTROL* gesetzt.

Manche Systeme bieten zusätzlich die Funktion

```
int setpgp(void);
```

an. Ein Aufruf dieser Funktion ist identisch mit dem Aufruf `setpgid(0, 0)`.

Mit der in Kapitel 11.3 vorgestellten Funktion **setsid** wird ebenfalls eine Prozeßgruppe kreiert.

11.3 Session

Zu einer sogenannten *Session* können eine oder mehrere Prozeßgruppen gehören. Abbildung 11.3 zeigt z.B. eine Session mit vier Prozeßgruppen. Die Zuteilung von Prozessen zu Prozeßgruppen erfolgt üblicherweise entsprechend den auf der Kommandozeile angegebenen Pipelines. Um z.B. die in Abbildung 11.3 gezeigte Konstellation zu erhalten, sind die folgenden Kommandozeilen erforderlich:

```
$ a &
$ b | c &
$ d | e | f
$
```

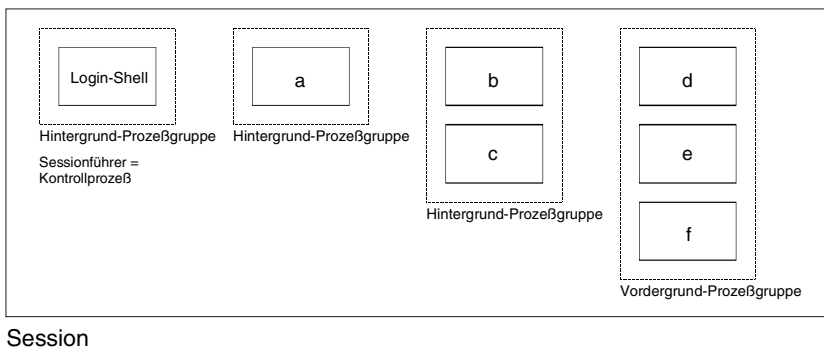


Abbildung 11.3: Eine Session mit vier Prozeßgruppen

11.3.1 setsid – Einrichten einer neuen Session

Um eine neue Session einzurichten, steht die Funktion **setsid** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

gibt zurück: Prozeßgruppen-ID (bei Erfolg); -1 bei Fehler

Ist der aufrufende Prozeß kein Prozeßgruppenführer, so richtet **setsid** eine neue Session ein. Es sind dabei die folgenden Punkte zu beachten:

1. Der aufrufende Prozeß wird der Sessionführer (*session leader*) der neuen Session. Er ist zu diesem Zeitpunkt auch der einzige Prozeß in dieser neuen Session.
2. Der Prozeß wird der Prozeßgruppenführer der neuen Prozeßgruppe. Die neue Prozeßgruppen-ID ist die Prozeß-ID des aufrufenden Prozesses.
3. Der Prozeß hat keinerlei Kontrollterminal (siehe Kapitel 11.4). Falls der Prozeß vor dem Aufruf von **setsid** ein Kontrollterminal hatte, so wird diese Zuordnung aufgehoben.

setsid gibt -1 (für Fehler) zurück, wenn der aufrufende Prozeß bereits ein Prozeßgruppenführer ist. Um dies zu verhindern, kreiert man üblicherweise mittels **fork** einen Kindprozeß, der weiterläuft, während sich der Elternprozeß beendet. Der Kindprozeß kann nämlich kein Prozeßgruppenführer sein, da er zwar die Prozeßgruppen-ID vom Elternprozeß erbt, aber in jedem Fall eine neue Prozeß-ID erhält, die niemals eine Prozeßgruppen-ID sein kann, da sie neu ist.

Hinweis

- POSIX.1 erwähnt nirgends eine Session-ID, sondern kennt nur einen Sessionführer (*session leader*). SVR4 interpretiert diese nicht spezifizierte Vorgabe dahingehend, daß es eine Session-ID einführt, die gleich der Prozeß-ID des Sessionführers ist. BSD-Unix folgt dieser Vorgehensweise nicht.
- SVR4 bietet die Funktion **getsid** an, um die Session-ID eines Prozesses zu erfragen. **getsid** ist nicht Bestandteil von POSIX.1 und wird nicht von BSD-Unix angeboten.

11.4 Kontrollterminals, Sessions und Prozeßgruppen

Nachfolgend sind die Beziehungen zwischen diesen drei Begriffen zusammengefaßt:

- Eine Session kann nur ein einziges Kontrollterminal haben. Das Kontrollterminal ist dabei abhängig vom Login entweder eine Terminalgerätedatei (bei einem Terminal-Login) oder eine Pseudoterminalgerätedatei (bei einem Netzwerk-Login).
- Der Sessionführer, der die Verbindung zum Kontrollterminal einrichtete, wird als der Kontrollprozeß (*controlling process*) bezeichnet.
- Prozeßgruppen in einer Session können in eine *Vordergrund-Prozeßgruppe* und eine oder mehrere *Hintergrund-Prozeßgruppen* aufgeteilt werden. Wenn eine Session einen Kontrollterminal hat, so hat sie eine Vordergrund-Prozeßgruppe und alle anderen Prozeßgruppen dieser Session sind Hintergrund-Prozeßgruppen.

- Wird eine der beiden Programm-Abbruchtasten **INTR** (*Strg-C* oder *DEL*) oder **QUIT** (*Strg-*) gedrückt, so wird allen Prozessen in der Vordergrund-Prozeßgruppe das Signal **SIGINT** oder **SIGQUIT** geschickt.
- Wird eine Modemverbindung unterbrochen, so wird dem Kontrollprozeß (Sessionführer) das Signal **SIGHUP** (*hangup*) geschickt.

11.4.1 /dev/tty – Gerätedatei für das Kontrollterminal

In gewissen Situationen kann es vorkommen, daß ein Programm in jedem Fall auf das Terminal schreiben oder von ihm lesen möchte, unabhängig davon, ob die Standardausgabe oder Standardeingabe umgelenkt ist. Hierzu muß das Programm die Datei `/dev/tty` öffnen, denn diese spezielle Gerätedatei ist immer unabhängig von Umlenkungen auf das Kontrollterminal eingestellt. Falls das Programm kein Kontrollterminal hat, so schlägt natürlich auch der Versuch, die Datei `/dev/tty` zu öffnen, fehl.

11.4.2 **tcgetpgrp** und **tcsetpgrp** – Erfragen und Setzen der Vordergrund-Prozeßgruppen-ID

Zum Erfragen oder Setzen der Vordergrund-Prozeßgruppen-ID stehen die beiden Funktionen **tcgetpgrp** und **tcsetpgrp** zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp(int filedescriptor);
           gibt zurück: Prozeßgruppen-ID der Vordergrund-Prozeßgruppe (bei Erfolg); -1 bei Fehler

int tcsetpgrp(int filedescriptor, pid_t pgrpid);
           gibt zurück: 0 (bei Erfolg); -1 bei Fehler
```

Die Funktion **tcgetpgrp** gibt die Prozeßgruppen-ID der Vordergrund-Prozeßgruppe zurück, die mit dem Terminal verbunden ist, das mit *filedescriptor* geöffnet ist. Wenn der Prozeß einen Kontrollterminal hat, so kann er mit **tcsetpgrp** die Vordergrund-Prozeßgruppen-ID auf *pgrpid* setzen.

Der Wert von *pgrpid* muß die Prozeßgruppen-ID einer Prozeßgruppe in derselben Session sein und *filedescriptor* muß dem Kontrollterminal der Session zugeordnet sein.

Hinweis

Die beiden Funktionen **tcgetpgrp** und **tcsetpgrp** sind nur dann definiert, wenn `_POSIX_JOB_CONTROL` definiert ist. Ansonsten liefert ein Aufruf einer dieser beiden Funktionen -1 (für Fehler).

Diese beiden Funktionen werden meistens nicht direkt, sondern von Jobkontroll-Shells aufgerufen (siehe auch Kapitel 11.5).

11.5 Jobkontrolle und Programmausführung durch die Shell

11.5.1 Allgemeines zur Jobkontrolle

Jobkontrolle durch die Shell ermöglicht es, mehrere Jobs (Prozeßgruppen) von einem Terminal aus zu starten und dann zu steuern, welche Jobs im Vordergrund ablaufen und damit die Möglichkeit des Lesens und Schreibens auf dem Terminal haben sollen bzw. umgekehrt, welche Jobs im Hintergrund ablaufen sollen. Die Konstante `_POSIX_JOB_CONTROL` legt fest, ob ein System über Jobkontrolle verfügt oder nicht.

Sowohl SVR4 als auch BSD-Unix verfügen über die von POSIX.1 vorgeschriebene Form der Jobkontrolle.

Benutzt man die Jobkontrolle von einer Shell aus, so kann man einen Job entweder im Vordergrund oder im Hintergrund starten:

- (1) `ls *.c`
- (2) `cc -o prog *.c &`
- (3) `find / -name "*.c" -print | lp &`

(1) startet einen Job, der nur aus einem Prozeß im Vordergrund besteht. (2) und (3) starten zwei Jobs im Hintergrund.

Ein Job ist eine Sammlung von Prozessen, meist ein einzelner Prozeß oder eine Pipeline von Prozessen. Tabelle 11.1 gibt für die drei gängigen Shells an, ob sie über Jobkontrolle verfügen oder nicht.

	Jobkontrolle
Bourne-Shell sh jsh (Job-Shell)	nein ja
Korn-Shell ksh	ja
C-Shell csh	ja

Tabelle 11.1: Jobkontrolle in den drei wichtigsten Shells

Wenn man einen Hintergrund-Job startet, so erhält dieser eine Jobnummer, die die betreffende Shell ebenso ausgibt, wie eine oder mehrere der Prozeß-IDs. Der nachfolgende Ablauf zeigt dies beispielhaft für die Korn-Shell:

```
$ cc -o prog *.c &
[1] 2387
$ find / -name "*.c"-print | lp &
[2] 2401
$      [Eingabe von Return]
[1] + Done cc -o prog *.c &
[2] + Done find / -name "*.c" -print | lp &
$
```

Der Compileraufruf hat die Jobnummer 1 und der zugehörige Startprozeß die Prozeß-ID 2387. Die **find**-Pipeline hat die Jobnummer 2 und der erste dabei gestartete Prozeß hat die Prozeß-ID 2401. Wenn die Jobs ihre Ausführung beendet haben und man die Return-Taste drückt, so teilt die Shell deren Beendigung mit.

Das Drücken der Return-Taste ist dabei notwendig, da die Shell die Beendigung von Hintergrundprozessen immer nur bei der Ausgabe des Promptzeichens als Mitteilung ausgibt. So wird verhindert, daß eine Mitteilung zu einem beliebigen Zeitpunkt, z.B. während man ein anderes Kommando ausführt, am Bildschirm erscheint.

11.5.2 Tastenkombinationen zur Jobkontrolle

Für die Jobkontrolle existieren spezielle Tastenkombinationen:

- ▶ Programmabbruchtaste (meist *DEL* oder *Strg-C*); generiert das Signal SIGINT.
- ▶ Programmabbruchtaste mit *core*-Datei (meist *Strg-*); generiert das Signal SIGQUIT.
- ▶ Programmsuspendierungstaste (meist *Strg-Z*). Diese Suspendierungstaste hält alle Prozesse der Vordergrundprozeßgruppe an. Dies wird dadurch bewirkt, daß allen Prozessen der Vordergrundprozeßgruppe das Signal SIGTSTP geschickt wird.

11.5.3 Lesen vom Terminal durch Hintergrundprozesse (mit Jobkontrolle)

Nur ein Vordergrund-Job kann direkt von Terminal lesen. Das heißt, daß Hintergrundprozesse niemals die am Terminal eingegebenen Zeichen lesen können. Versucht ein Hintergrundprozeß vom Terminal zu lesen, so erkennt der Terminaltreiber dies und schickt dem betreffenden Hintergrundprozeß das Signal SIGTTIN. Dieses Signal bewirkt normalerweise, daß der Hintergrundprozeß angehalten wird, was die Shell zu einer entsprechenden Mitteilung am Terminal veranlaßt. Nun kann man den entsprechenden Hintergrundprozeß durch das Kommando **fg** in den Vordergrund bringen, so daß er sein gewünschtes Lesen vom Terminal durchführen kann. Nachfolgender Ablauf verdeutlicht dies:

```
$ cat lesstdin      [Skript, das Zeilen vom Terminal liest und dann wieder ausgibt]
while read zeile
do
    echo $zeile
done
$ lesstdin &       [Skript lesstdin im Hintergrund starten]
```

```
[1] 2704
$                               [Eingabe von Return]
[1] + Suspended (tty input) lesstdin
$ fg %1                         [Job mit Jobnummer 1 in Vordergrund bringen]
lesstdin                       [Mitteilung von Shell, daß Job nun im Vordergrund]
eine Zeile                     [Eingabe von 2 Textzeilen, die einfach wieder ausgegeben werden]
eine Zeile
und noch ein bisschen Text
und noch ein bisschen Text
Ctrl-D                         [Eingabeende mittels EOF]
$
```

Das Kommando **fg** verwendet im übrigen die in Kapitel 11.4 vorgestellte Funktion **tcsetgrp**, um den entsprechenden Job in der Vordergrundprozeßgruppe zu plazieren. Danach schickt **fg** dieser Prozeßgruppe das Signal SIGCONT, so daß sie ihre Ausführung nun fortsetzt. Da der entsprechende Job jetzt ein Vordergrundprozeß ist, kann er vom Kontrollterminal lesen.

11.5.4 Lesen vom Kontrollterminal durch Hintergrundprozesse (ohne Jobkontrolle)

Wenn ein Hintergrundprozeß in einer Shell ohne Jobkontrolle von seinem Kontrollterminal lesen will, so lenkt die Shell die Standardeingabe des Hintergrundprozesses automatisch in /dev/null um. Ein Lesen von /dev/null führt zum sofortigen Lesen von EOF. So zieht z.B. in der Bourne-Shell der Aufruf

```
cat > xxx.c &
```

keinerlei Leseaktion nach sich, sondern beendet sich sofort und führt zu einer leeren Datei xxx.c.

11.5.5 Schreiben auf Terminal durch Hintergrundprozesse

Das direkte Schreiben von Hintergrundprozessen auf ein Terminal kann man zulassen oder verbieten. Dazu verwendet man das Kommando **stty**:

```
stty tostop    (Verbieten des direkten Schreibens)
stty -tostop   (Erlauben des direkten Schreibens)
```

Nachfolgender Ablauf verdeutlicht dies:

```
$ ls -l /bin/c* &
[1] 2873
$                               [Nach Prompt schreibt der Hintergrund-Prozeß direkt auf's Terminal]
/bin/cat
/bin/chgrp
/bin/chmod
/bin/chown
/bin/compress
/bin/cp
/bin/cpio
```

```

/bin/csh
/bin/cut
[1] + Done    ls -l /bin/c*
$ stty tostop    [Verbieten der direkten Ausgabe durch Hintergrund-Prozeß]
$ ls -l /bin/c* &
[1] 2947
$
[1] + suspended (tty output) ls -l /bin/c*    [Da Schreiben verboten, wird Job susp.]
$ fg %1    [Bringe suspendierten Hintergrundjob in Vordergrund]
ls -l /bin/c*    [Shell teilt neuen Vordergrundjob mit]
/bin/cat    [Ausgabe des nun im Vordergrund ablaufenden ls]
/bin/chgrp
/bin/chmod
/bin/chown
/bin/compress
/bin/cp
/bin/cpio
/bin/csh
/bin/cut
$

```

11.5.6 Ausführung von Programmen durch eine Shell ohne Jobkontrolle

Hier verwenden wir als Shell die Bourne-Shell **sh**, die über keine Jobkontrolle verfügt.

Komandos im Vorder- und Hintergrund

Hierbei rufen wir zunächst **ps** mit den entsprechenden Optionen auf:

```

ps -jl          (unter SVR4, gibt jedoch niemals TPGID aus)
ps -xj -otpgid (unter 4.4BSD)
ps -xj          (unter Linux)

```

woraus dann z.B. die folgende (etwas gekürzte) Ausgabe resultiert.

PPID	PID	PGID	SID	TPGID	COMMAND
1	100	100	100	100	-sh
100	200	100	100	100	ps

An dieser Ausgabe ist zu erkennen, daß sich sowohl die Shell als auch das **ps**-Kommando in der gleichen Session und Vordergrundprozeßgruppe (SID=100 und TPGID=100) befinden.

TPGID steht dabei für *Terminal-Prozeßgruppen-ID*. Wenn eine Session kein Kontrollterminal hat, dann wird dies durch -1 in der TPGID-Spalte angezeigt.

Wenn wir den obigen **ps**-Aufruf im Hintergrund ausführen, so ergibt sich bis auf eine andere PID für das **ps**-Kommando die gleiche Ausgabe wie oben. Der Hintergrund-Job wird also nicht einer eigenen Prozeßgruppe zugeordnet, und das Kontrollterminal wird dem Hintergrundprozeß nicht entzogen, da die Bourne-Shell keine Jobkontrolle kennt.

Pipes im Vorder- und Hintergrund

Nun wollen wir das Verhalten der Bourne-Shell bei der Angabe einer Pipe auf der Kommandozeile testen. Dazu geben wir z.B. die folgende Kommandozeile an:

```
ps -xj | cat
```

woraus dann z.B. die folgende (gekürzte) Ausgabe resultiert.

PPID	PID	PGID	SID	TPGID	COMMAND
1	100	100	100	100	-sh
100	200	100	100	100	cat
200	201	100	100	100	ps

Hieran läßt sich erkennen, daß der letzte Prozeß in der Pipeline (**cat**) der Kindprozeß der Shell ist, während der erste Prozeß der Pipeline (**ps**) der Kindprozeß des letzten Prozesses (**cat**) ist.

Wenn wir die obige Kommandozeile im Hintergrund ausführen, so ergibt sich bis auf andere PIDs für die Kommandos **ps** und **cat** die gleiche Ausgabe wie oben.

Nun wollen wir das Verhalten der Bourne-Shell bei der Angabe einer Pipeline überprüfen, die sich nicht nur über zwei, sondern über drei Prozesse erstreckt. Dazu geben wir die folgende Kommandozeile ein:

```
ps -xj | cat | lesstdin
```

woraus dann z.B. die folgende (etwas gekürzte) Ausgabe resultiert:

PPID	PID	PGID	SID	TPGID	COMMAND
1	100	100	100	100	-sh
100	200	100	100	100	lesstdin
200	201	100	100	100	ps
200	202	100	100	100	cat

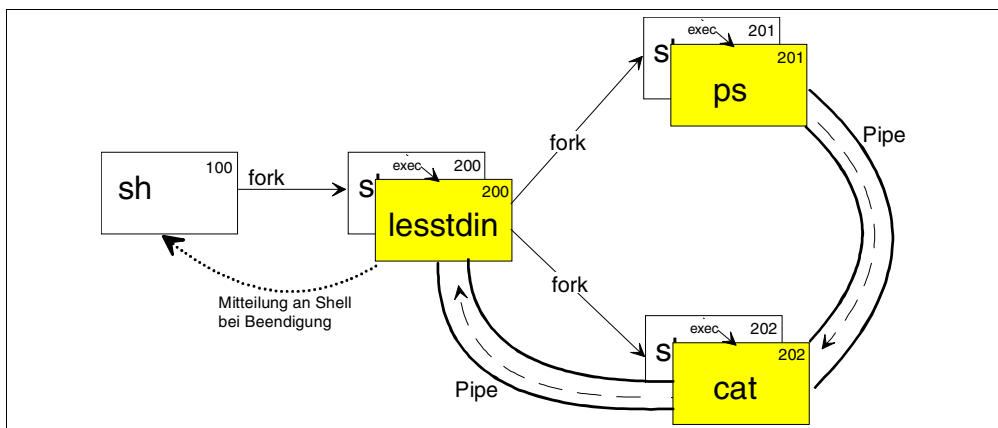


Abbildung 11.4: Prozeßstruktur zur Pipeline »ps -xj | cat | lesstdin«

An dieser Ausgabe läßt sich erkennen, daß der letzte Prozeß in der Pipeline (`lesstdin`) der Kindprozeß der Shell ist, und alle vorherigen Prozesse in der Pipeline Kindprozesse dieses letzten Prozesses sind. Abbildung 11.4 stellt diese Prozeßstruktur dar.

Da der letzte Prozeß in der Pipeline der Kindprozeß der Loginshell ist, wird diese bei dessen Beendigung (bedeutet auch die Beendigung aller Kommandos der Pipe) darüber informiert, und kann dann weitere Kommandos entgegennehmen.

11.5.7 Ausführung von Programmen durch eine Shell mit Jobkontrolle

Hier verwenden wir als Shell die Korn-Shell, die über eine Jobkontrolle verfügt.

Kommandos im Vorder- und Hintergrund

Hierbei geben wir zunächst folgende Kommandozeile ein:

```
ps -xj
```

woraus dann z.B. die folgende (gekürzte) Ausgabe resultiert:

PPID	PID	PGID	SID	TPGID	COMMAND
1	500	500	500	510	-ksh
500	510	510	500	510	ps

Bei dieser und den folgenden Ausgaben sind die Zeilen der Prozesse in der Vordergrundprozeßgruppe fett gedruckt. Im Unterschied zur Bourne-Shell platziert die Korn-Shell den Vordergrundprozeß (**ps**) in seine eigene Prozeßgruppe (510). Das Kommando **ps** ist dabei der Prozeßgruppenführer und der einzige Prozeß in dieser Prozeßgruppe. Diese Prozeßgruppe ist dann die Vordergrundprozeßgruppe, da sie das Kontrollterminal hat. Während das **ps**-Kommando ausgeführt wird, ist die Loginshell eine Hintergrundprozeßgruppe, obwohl beide Prozesse (**ksh** und **ps**) Mitglieder der gleichen Session (500) sind.

Wenn wir das **ps**-Kommando im Hintergrund ausführen

```
ps -xj &
```

so ergibt sich z.B. die folgende Ausgabe:

PPID	PID	PGID	SID	TPGID	COMMAND
1	500	500	500	500	-ksh
500	520	520	500	500	ps

Auch hier wird das **ps**-Kommando in einer eigenen Prozeßgruppe untergebracht, die jedoch keine Vordergrundprozeßgruppe, sondern eben eine Hintergrundprozeßgruppe ist. Die TPGID von 500 zeigt an, daß die Loginshell hier die Vordergrundprozeßgruppe ist.

Pipes im Vorder- und Hintergrund

Nun wollen wir das Verhalten der Korn-Shell bei der Angabe einer Pipe auf der Kommandozeile testen. Dazu geben wir z.B. die folgende Kommandozeile an

```
ps -xj | cat
```

woraus dann z.B. die folgende (gekürzte) Ausgabe resultiert:

PPID	PID	PGID	SID	TPGID	COMMAND
1	500	500	500	510	-ksh
500	510	510	500	510	ps
500	520	510	500	510	cat

Hier werden die beiden Prozesse (**ps** und **cat**) in einer neuen Prozeßgruppe (510) plaziert. Diese Prozeßgruppe ist dabei die Vordergrundprozeßgruppe. Anders als bei der Bourne-Shell, in der der letzte Prozeß der Pipeline zuerst kriert und dann der Elternprozeß von allen anderen Prozessen der Pipeline wurde, ist hier die Korn-Shell der Elternprozeß von allen Prozessen der Pipeline.

Läßt man dagegen die obige Pipeline im Hintergrund ablaufen, wie z.B.

```
(ps -xj | cat) &
```

so ergibt sich die folgende (gekürzte) Ausgabe:

PPID	PID	PGID	SID	TPGID	COMMAND
1	700	700	700	700	-ksh
700	800	800	700	700	cat
800	900	800	700	700	ps

Hier werden die beiden Prozesse (**cat** und **ps**) in einer eigenen Hintergrundprozeßgruppe (800) plaziert.

11.6 Verwaiste Prozeßgruppen

Wenn ein Elternprozeß sich vorzeitig beendet, so werden alle seine Kindprozesse zu sogenannten verwaisten Prozessen (*orphans*), deren neuer Elternprozeß der `init`-Prozeß wird.

Nun kann es jedoch passieren, daß eine ganze Prozeßgruppe verwaist. Wenn z.B. ein Kindprozeß mit dem Signal `SIGTSTP` (durch *Strg-Z* ausgelöst) angehalten wird und sich während dieser Suspendierungszeit der Elternprozeß beendet, so erhält der Kindprozeß als neuen Elternprozeß den `init`-Prozeß (`PID=1`) und wird damit auch automatisch Mitglied einer *verwaisten Prozeßgruppe*.

Eine *verwaiste Prozeßgruppe* ist nach POSIX.1 eine Prozeßgruppe, in der der Elternprozeß jedes Mitglieds entweder selbst Gruppenmitglied oder aber nicht ein Mitglied der Gruppen-Session ist. Anders ausgedrückt: Eine Prozeßgruppe ist solange nicht verwaist, solange sie mindestens einen Prozeß enthält, dessen Elternprozeß zu einer anderen Pro-

zeßgruppe in der gleichen Session gehört. Solange eine Prozeßgruppe nicht verwaist ist, besteht die Möglichkeit, daß einer der Elternprozesse aus einer anderen Prozeßgruppe der gleichen Session einen angehaltenen Prozeß wieder aufweckt.

11.7 Übung

11.7.1 Kreieren einer neuen Session durch einen Kindprozeß

Erstellen Sie ein Programm `kindsess.c`, das mit ***fork*** einen neuen Kindprozeß kreiert, der dann seinerseits eine neue Session anlegt. Überprüfen Sie, ob dieser Kindprozeß dabei der Prozeßgruppenführer wird und ob er nach der Kreierung der Session noch ein Kontrollterminal besitzt oder nicht.

11.7.2 Kontrollterminal für eine verwaiste Prozeßgruppe

Erstellen Sie ein Programm `waisgrp.c`, das einen Kindprozeß erzeugt, der sich selbst suspendiert. In dieser Suspendierungszeit soll sich dann der Elternprozeß beenden, so daß dieser Kindprozeß Mitglied der verwaisten Prozeßgruppe wird. Hat dieser Kindprozeß dann noch ein Kontrollterminal, von dem er lesen kann oder nicht ?

12 Blockierungen und Sperren von Dateien

*Liebe deinen Nachbarn,
reiß aber den Zaun nicht ein.*

Spruchwort

Dieses Kapitel stellt zunächst blockierende und nichtblockierende E/A-Operationen vor, bevor es sich ausführlich mit dem Sperren von Dateien und den dabei möglichen Problemen beschäftigt. In der Übung wird ein umfangreicheres Projekt vorgestellt: die Entwicklung einer einfachen Multiuser-Datenbank.

12.1 Blockierende und nichtblockierende E/A-Operationen

Die Systemaufrufe lassen sich in zwei Kategorien unterteilen, die sogenannten »langsamen« Systemaufrufe, die eine Blockierung nach sich ziehen können, und die übrigen Systemaufrufe, bei denen keine Blockierung möglich ist.

12.1.1 Blockierende E/A-Operationen

Die langsamen Systemaufrufe können dazu führen, daß der aufrufende Prozeß für immer blockiert wird. Zu dieser Kategorie von Systemaufrufen zählen die Funktionen, die Operationen der folgenden Art durchführen:

- ▶ Lesen von Dateien, die den aufrufenden Prozeß für immer blockieren können, wenn keine Daten vorhanden sind (Pipes, Terminalgerätedateien und Netzwerk-Gerätedateien).
- ▶ Schreiben in Dateien, die den aufrufenden Prozeß für immer blockieren können, wenn Schreiben nicht sofort möglich ist (Pipes, Terminalgerätedateien und Netzwerk-Gerätedateien).
- ▶ Öffnen von Dateien, die solange blockiert sind, bis ein bestimmtes Ereignis eintritt (wie z.B. Öffnen einer Terminalgerätedatei, was solange blockiert wird, bis das angeschlossene Modem den Anruf beantwortet, oder Öffnen einer FIFO zum *Nur-Schreiben* (*write-only*), wenn kein anderer Prozeß diese FIFO zum Lesen geöffnet hat).
- ▶ Lesen und Schreiben von Dateien, für die *mandatory record locking* (zwangsweise Satzblockierung) festgelegt wurde.

- bestimmte **ioctl**-Operationen
- einige der Funktionen für Interprozeßkommunikation (siehe Kapitel 17 und 18)

Da E/A-Operationen auf Speichermedien wie Festplatten oder Disketten nur zu einer zeitlich begrenzten Blockierung führen können, werden solche E/A-Routinen nicht den langsamen Systemaufrufen zugeordnet.

12.1.2 Nichtblockierende E/A-Operationen

Bei nichtblockierenden E/A-Operationen wie z.B. **open**, **read** oder **write** ist sichergestellt, daß beim Fehlschlagen der Operation die entsprechende Funktion sich sofort mit einem Fehler beendet.

Es existieren zwei Möglichkeiten, um für einen Filedeskriptor »nichtblockierende E/A« einzustellen:

1. Setzen des Flags `O_NONBLOCK` beim Öffnen der Datei mit **open** (siehe Kapitel 4.2).
2. Bei einem bereits geöffneten Filedeskriptor nachträglich das Flag `O_NONBLOCK` mit der Funktion **fcntl** (siehe Kapitel 4.9) einschalten.

12.2 Sperren von Dateien (record locking)

Für manche Anwendungen, wie z.B. Datenbanksysteme, ist es wichtig, daß zu einem Zeitpunkt nur ein Prozeß in eine Datei schreibt. Hierzu muß es möglich sein, andere Prozesse vom Schreiben in eine Datei auszusperrern. Dazu bieten neuere Unix-Systeme das sogenannte *record locking* an. *Record locking* ermöglicht das Sperren einer Datei oder eines Dateibereichs für andere Prozesse, während ein Prozeß in dieser Datei oder im entsprechenden Dateibereich liest oder schreibt.

12.2.1 Sperren von Dateien oder Dateibereichen mittels fcntl

Zum Sperren von Dateien steht die Funktion **fcntl**, die in Kapitel 4.9 vorgestellt wurde, zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int kdo, ... /* struct flock *flockzgr */);
```

gibt zurück: abhängig von *kdo* (bei Erfolg); -1 bei Fehler

Im Zusammenhang mit *record locking* sind nur die Angaben `F_GETLK`, `F_SETLK` und `F_SETLKW` für `kdo` von Interesse. Das dritte Argument (hier `flockzgr`) ist ein Zeiger auf die Struktur `flock`.

```
struct flock {
    short l_type; /* F_RDLCK (gemeinsame Lesesperre),
                  F_WRLCK (exklusive Schreibsperre) oder
                  F_UNLCK (Sperre aufheben) */
    off_t l_start; /* relatives offset (in Bytes);
                  abhängig von l_whence */
    short l_whence; /* SEEK_SET, SEEK_CUR oder SEEK_END */
    off_t l_len; /* Länge (in Bytes);
                0 bedeutet Sperren bis Dateiende */
    pid_t l_pid; /* wird bei F_GETLK zurückgegeben */
};
```

Festlegen des Dateibereichs

Hierbei gelten die folgenden Regeln:

- ▶ Die Ausgangsposition für das offset (`l_start`) hängt wie bei der Funktion ***lseek*** (siehe Kapitel 4.4) von der Angabe für `l_whence` (`SEEK_SET`, `SEEK_CUR` oder `SEEK_END`) ab.
- ▶ Während eine Sperre am Ende einer Datei beginnen und sich über das Dateiende hinaus erstrecken kann, ist das Sperren eines Bereichs, der vor dem Dateianfang beginnt, nicht möglich.
- ▶ Um eine Datei immer von einer bestimmten Position bis zum aktuellen Dateiende zu sperren, muß für den Parameter `l_len` der Wert 0 angegeben werden. Diese Angabe stellt immer eine Sperre bis zum aktuellen Dateiende sicher, selbst wenn neue Daten ans Ende der Datei geschrieben werden und somit das Dateiende verschoben wird.
- ▶ Um eine ganze Datei zu sperren, muß `l_start` und `l_whence` auf den Dateianfang festgelegt werden und für `l_len` muß 0 ausgegeben werden. Es gibt zwar verschiedene Möglichkeiten, den Dateianfang festzulegen, meist geschieht dies aber durch folgende Angaben:

`l_start==0` und `l_whence==SEEK_SET`

F_RDLCK* und *R_WRLCK

Hier gilt allgemein, daß beliebig viele Prozesse eine gemeinsame Lesesperre (`F_RDLCK`), aber nur ein Prozeß eine exklusive Schreibsperre (`F_WRLCK`) für ein bestimmtes Byte festlegen können. Des weiteren gilt, daß niemals gleichzeitig für ein bestimmtes Byte `F_RDLCK` und `F_WRLCK` festgelegt sein kann.

Um für einen Filedeskriptor eine Lesesperre (`F_RDLCK`) einzurichten, muß dieser zum Lesen geöffnet sein. Ebenso kann nur dann eine Schreibsperre (`F_WRLCK`) für einen Filedeskriptor eingerichtet werden, wenn dieser zum Schreiben geöffnet ist.

Mögliche Angaben für *kdo*

Für den Parameter *kdo* können im Zusammenhang mit *record locking* nur eine der folgenden drei Angaben gemacht werden:

F_GETLK

Mit dieser Angabe für *kdo* kann festgestellt werden, ob die mit *flockzgr* spezifizierte Sperre bereits durch eine andere Sperre blockiert wird. Falls die mit *flockzgr* spezifizierte Sperre nicht mit einer anderen Sperre kollidiert, wird in der Struktur *flock*, auf die *flockzgr* zeigt, lediglich die Komponente *l_type* auf *F_UNLCK* gesetzt.

Ist die über *flockzgr* spezifizierte Sperre nicht erlaubt, so wird die Struktur *flock*, auf die *flockzgr* zeigt, mit den Daten der bereits existierenden Sperre überschrieben.

F_SETLK

Mit dieser Angabe für *kdo* wird die über *flockzgr* spezifizierte Sperre eingerichtet. Falls diese geforderte Sperre nicht möglich ist, weil eine der zuvor beschriebenen Regeln (siehe Unterkapitel »*F_RDLCK* und *F_WRLCK*“) verletzt wird, so beendet sich die Funktion *fcntl* sofort und setzt die Variable *errno* entweder auf *EACCES* oder *EAGAIN*.

F_SETLK wird auch benutzt, um eine zuvor eingerichtete Sperre wieder aufzuheben. Dazu muß die Komponente *l_type* der Struktur, auf die *flockzgr* zeigt, auf *F_UNLCK* gesetzt werden.

F_SETLKW

Bei dieser Angabe für *kdo* handelt es sich um eine blockierende Version zu *F_SETLK* (w steht dabei für *wait*). Wenn hierbei die geforderte Sperre nicht eingerichtet werden kann, weil ein anderer Prozeß momentan einen Teil des angegebenen Dateibereichs bereits gesperrt hat, dann beendet sich *fcntl* nicht wie bei *F_SETLK*, sondern der aufrufende Prozeß wird suspendiert und erst dann wieder durch ein Signal aufgeweckt, wenn die geforderte Sperre eingerichtet werden kann.

Hinweis

Die Überprüfung mit *F_GETLK* auf das Vorliegen einer Sperre und das anschließende Einrichten einer Sperre mit *F_SETLK* und *F_SETLKW*, wenn die Überprüfung entsprechendes zuließ, sind keine atomaren Operationen. Es ist also nicht garantiert, daß zwischen den beiden *fcntl*-Aufrufen nicht ein anderer Prozeß dazwischenkommt und die entsprechende Sperre seinerseits einrichtet. Deshalb ist es wichtig, daß man entweder bei *F_SETLKW* (Warten auf Freiwerden der entsprechenden Sperre), oder aber bei *F_SETLK* den Rückgabewert und die Variable *errno* prüft, um so einen eventuell fehlerhaften *fcntl*-Aufruf zu erkennen.

Wenn man eine Sperre einrichtet oder wieder freigibt, so faßt das System aneinanderliegende Einzelbereiche automatisch zu einem geschlossenen Bereich zusammen oder teilt sie bei Freigabe in entsprechende Einzelbereiche auf. Richtet man z.B. für die Bytes 100 bis 400 eine Lesesperre ein, hebt dann diese Sperre für die Bytes 100 bis 300 auf und richtet dafür eine Schreibsperre für diesen Bereich ein, so liegen zwei Sperrbereiche vor: 100 bis 300 (Schreibsperre) und 301 bis 400 (Lesesperre). Wenn man z.B. eine Sperre für die

Bytes 500 bis 600 einrichtet und dann diese Sperre für das Byte 550 aufhebt, so bleibt weiterhin die Sperre für die Bytes 500 bis 549 und 551 bis 600 bestehen.

Das Sperren von Datei(bereich)en mit der Funktion **fcntl** entspricht dem POSIX.1-Standard, sowohl SVR4 als auch 4.4BSD bieten diese Möglichkeit an. SVR4 bietet daneben die Funktion **lockf** an, die lediglich eine andere Form des Aufrufs von **fcntl** ist. BSD-Unix bietet neben **fcntl** die ältere Funktion **flock** an, die jedoch nur das Sperren ganzer Dateien und nicht wie **fcntl** das Sperren einzelner Bereiche in Dateien zuläßt.

12.2.2 Einrichten, Freigeben und Testen von Sperren

Um nicht bei jeder Sperre, die man einrichten, freigeben oder testen möchte, eine Struktur **flock** allokalieren und die Komponenten dieser Struktur entsprechend setzen zu müssen, empfiehlt es sich, Funktionen wie `sperre_einaus` und `sperre_test` zu erstellen. Diese Funktionen sind im Programm 12.1 (`sperre.c`) angegeben.

```
#include <sys/types.h>
#include <fcntl.h>
#include "eighdr.h"
/*---- Einrichten oder Freigeben einer Sperre -----*/
int sperre_einaus(int fd, int kdo, int sperr_typ,
                  off_t offset, int wie, off_t laenge)
{
    struct flock    sperre;

    sperre.l_type   = sperr_typ; /* F_RDLCK, F_WRLCK oder F_UNLCK      */
    sperre.l_start  = offset;    /* Byte-Offset (abhaengig von wie) */
    sperre.l_whence = wie;       /* SEEK_SET, SEEK_CUR oder SEEK_END */
    sperre.l_len    = laenge;    /* Anzahl von Bytes; 0 bedeutet bis EOF */

    return( fcntl(fd, kdo, &sperre) );
}

/*---- Testen einer Sperre -----*/
*      Wenn bereits eine Sperre vorliegt, die das Einrichten der hier
*      ueber die Argumente spezifizierten Sperre nicht zulaesst, so
*      liefert diese Funktion die Prozess-ID des Prozesses, der diese
*      blockierende Sperre eingerichtet hat; ansonsten liefert diese
*      Funktion als Rueckgabewert 0.                                     */
pid_t sperre_testen(int fd, int sperr_typ, off_t offset, int wie, off_t laenge)
{
    struct flock    sperre;

    sperre.l_type   = sperr_typ; /* F_RDLCK oder F_WRLCK      */
    sperre.l_start  = offset;    /* Byte-Offset (abhaengig von wie) */
    sperre.l_whence = wie;       /* SEEK_SET, SEEK_CUR oder SEEK_END */
    sperre.l_len    = laenge;    /* Anzahl von Bytes; 0 bedeutet bis EOF */

    if (fcntl(fd, F_GETLK, &sperre) < 0)
        fehler_meld(FATAL_SYS, "fcntl-Fehler");
}
```

```

    if (sperre.l_type == F_UNLCK)
        return(0); /* Bereich ist nicht durch anderen Prozess gesperrt */
    else
        return(sperre.l_pid); /* ID des Prozesses, der schon bestehende
                               Sperre eingerichtet hat          */
}

```

Programm 12.1 (**sperre.c**): Funktionen zum Einrichten, Freigeben und Testen von Dateisperren

Daneben ist es empfehlenswert, sich die folgenden Makros für das Einrichten, Freigeben und Testen von Sperren in der eigenen Headerdatei `eighdr.h` (siehe auch Anhang) zu definieren:

```

/*----- Einrichten einer Sperre -----*/
#define lese_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLK, F_RDLCK, offset, wie, laenge)
#define lesewarte_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLKW, F_RDLCK, offset, wie, laenge)
#define schreib_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLK, F_WRLCK, offset, wie, laenge)
#define schreibwarte_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLKW, F_WRLCK, offset, wie, laenge)

/*----- Aufheben einer Sperre -----*/
#define sperre_aufheben(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLK, F_UNLCK, offset, wie, laenge)

/*----- Testen einer Sperre -----*/
#define lesesperre_vorhanden(fd,offset,wie,laenge) \
    sperre_testen(fd, F_RDLCK, offset, wie, laenge)
#define schreibsperre_vorhanden(fd,offset,wie,laenge) \
    sperre_testen(fd, F_WRLCK, offset, wie, laenge)

```

Um sich die Reihenfolge der Argumente beim Aufruf dieser Makros besser merken zu können, entspricht die Reihenfolge der ersten drei Parameter der von der Funktion **lseek**.

12.2.3 Blockierung (Deadlock) durch gegenseitiges Aussperren

Beim gleichzeitigen Ablauf von Prozessen, die Dateibereiche sperren, ist es möglich, daß diese sich gegenseitig so blockieren, daß keiner mehr weiterarbeiten kann und das Programm sich somit in einem Blockadezustand befindet, der niemals wieder aufgehoben werden kann.

Ein solcher Blockadezustand wird mit *Deadlock* bezeichnet. Ein *Deadlock* kann z.B. dann auftreten, wenn ein Prozeß x, der eine Sperre A eingerichtet hat, später suspendiert wird, wenn er versucht, eine andere momentan durch einen Prozeß y blockierte Sperre B mit `F_SETLKW` einzurichten. Versucht nun der andere Prozeß y seinerseits die Sperre A für sich mit `F_SETLKW` einzurichten, so wird auch dieser suspendiert, da diese Sperre A momentan durch Prozeß x blockiert ist. Die beiden Prozesse x und y bleiben »ewig« suspendiert und werden niemals wieder aufgeweckt, da jeder auf die Freigabe einer Sperre des anderen wartet, was niemals geschehen wird (siehe Abbildung 12.1).

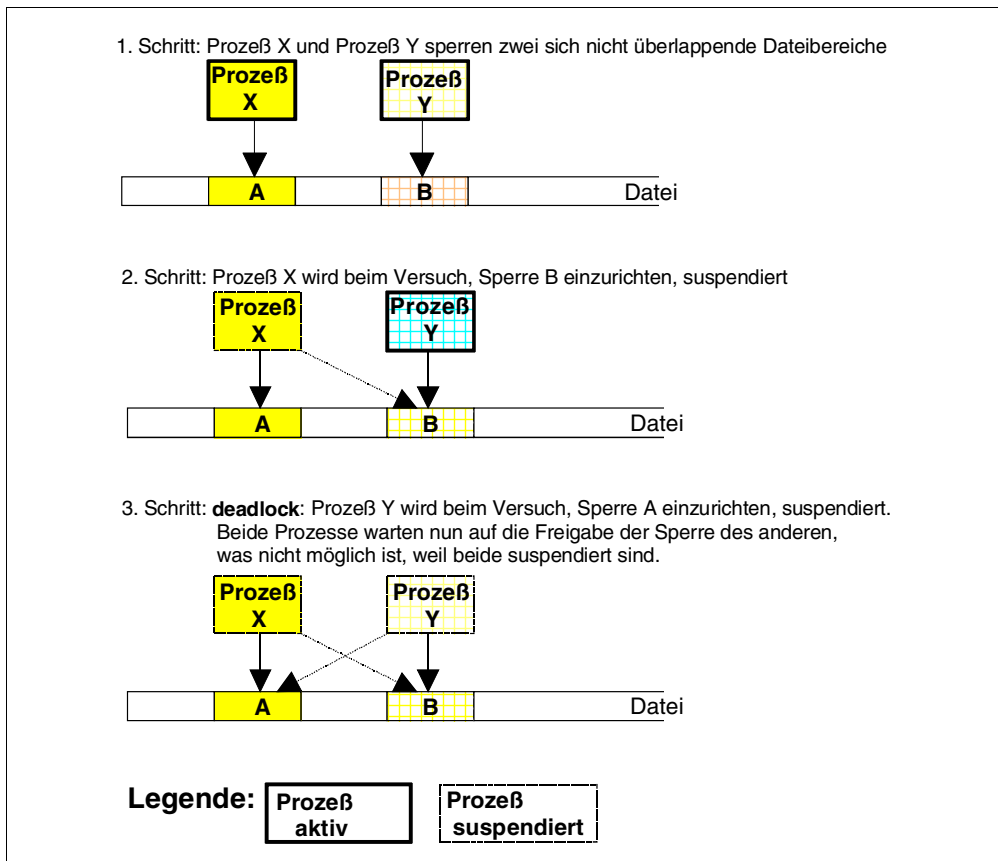


Abbildung 12.1: Deadlock von 2 Prozessen durch gegenseitiges Aussperren

Das nachfolgende Programm 12.2 (`no_dead.c`) zeigt anhand eines Kind- und Elternprozesses eine Technik, mit der *Deadlocks* vermieden werden können. Es verwendet dazu die Synchronisationsroutine **INIT_SYNCH**, **HALLO_KIND**, **WARTE_AUF_KIND**, **HALLO_PAPA** und **WARTE_AUF_PAPA** aus Programm 10.13 (`forksync.c`) in Kapitel 10.4.

In diesem Programm 12.2 (`no_dead.c`) sperrt der Elternprozeß die Bytes 0 bis 19 und der Kindprozeß die Bytes 20 bis zum Ende einer temporären Datei, die mit dem Zeichen `x` gefüllt ist. Danach versucht der Elternprozeß den vom Kindprozeß gesperrten Bereich und der Kindprozeß den vom Elternprozeß gesperrten Bereich zu sperren. Mit der Verwendung der Synchronisationsroutinen aus Programm 10.13 (`forksync.c`) wird sichergestellt, daß jeder Prozeß darauf wartet, bis der andere seine erste Dateisperre eingerichtet hat. Der Kern kann in diesem Fall den Deadlock erkennen.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```

#include    "eighdr.h"

static void
    sperre_bereich(const char *prozess, int fd, off_t von, off_t laenge);

int
main(void)
{
    int    i,
           fd;
    pid_t  pid;

    /*---- Anlegen einer temporaeren Datei, die mit Zeichen X gefuelllt wird */
    if ( (fd = creat("tmpdatei", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0)
        fehler_meld(FATAL_SYS, "creat-Fehler");
    for (i=1; i<100; i++)
        if (write(fd, "X", 1) != 1)
            fehler_meld(FATAL_SYS, "write-Fehler");

    INIT_SYNCH(); /* Synchronisation initialisieren */

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid == 0) { /*--- Kindprozess ---*/
        sperre_bereich("Kindprozess", fd, 20, 0);
        HALLO_PAPA(getppid());
        WARTE_AUF_PAPA();
        sperre_bereich("Kindprozess", fd, 0, 20);

    } else { /*--- Elternprozess ---*/
        sperre_bereich("Elternprozess", fd, 0, 20);
        HALLO_KIND(pid);
        WARTE_AUF_KIND();
        sperre_bereich("Elternprozess", fd, 20, 0);
    }

    exit(0);
}

static void
    sperre_bereich(const char *prozess, int fd, off_t von, off_t laenge)
{
    if (schreibwarte_sperre(fd, von, SEEK_SET, laenge) < 0)
        fehler_meld(FATAL_SYS, "%s: Fehler beim Sperrversuch", prozess);

    if (laenge != 0)
        printf("%s: Bereich %d..%d gesperrt\n", prozess, von, von+laenge-1);
    else
        printf("%s: Bereich %d..EOF gesperrt\n", prozess, von);
}

```

Bei der Angabe des Offsets ist zu beachten, daß die Zählung bei 0 beginnt.

Nachdem man das Programm 12.2 (`no_dead.c`) kompiliert und gelinkt hat

```
cc -o no_dead no_dead.c sperre.c forksync.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ no_dead
Kindprozess: Bereich 20..EOF gesperrt
Elternprozess: Bereich 0..19 gesperrt
Elternprozess: Fehler beim Sperrversuch: Deadlock situation detected/avoided
Kindprozess: Bereich 0..19 gesperrt
$
```

Wenn der Kern einen Deadlock entdeckt, so ist es vom jeweiligen System oder sogar vom Zufall abhängig, für welchen Prozeß er den Fehler meldet.

12.2.4 Sperren für Dämonen

Sperren können von sogenannten Dämonprozessen (*daemons*) verwendet werden, um sicherzustellen, daß immer nur eine Kopie desselben Dämonprozesses abläuft. Dämonprozesse werden in Kapitel 16 beschrieben.

Beim Start schreiben viele Dämonprozesse ihre Prozeß-ID in eine Datei. Diese Prozeß-IDs sind nützlich, wenn ein System herunterzufahren ist und alle noch laufenden Dämonprozesse ordnungsgemäß zu beenden sind.

Um zu verhindern, daß mehrere Kopien desselben Dämonen gleichzeitig ablaufen, muß ein Dämon beim Start eine Sperre auf die Datei mit seiner Prozeß-ID einrichten. Hebt er diese Sperre während seiner Laufzeit niemals auf, so können keine neuen Kopien dieses Dämons gestartet werden. Programm 12.3 (`sperdaem.c`) demonstriert diese Technik.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(void)
{
    int          fd, laenge, wert;
    char         puffer[10];

    if ( (fd = open("pid_daem", O_WRONLY | O_CREAT,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0)
        fehler_meld(FATAL_SYS, "open-Fehler");

    /*--- Ganze Datei fuer Schreiben sperren ---*/
    if (schreib_sperre(fd, 0, SEEK_SET, 0) < 0) {
        if (errno == EACCES || errno == EAGAIN)
```

```

        exit(0); /*--- Ende, wenn Daemon schon laeuft ---*/
    else
        fehler_meld(FATAL_SYS, "schreib_sperre-Fehler");
}

/*--- Datei leeren ---*/
if (ftruncate(fd, 0) < 0)
    fehler_meld(FATAL_SYS, "ftruncate-Fehler");

/*--- Prozess-ID schreiben ---*/
sprintf(puffer, "%d\n", getpid());
laenge = strlen(puffer);
if (write(fd, puffer, laenge) != laenge)
    fehler_meld(FATAL_SYS, "write-Fehler");

/*--- close-on-exec-Flag fuer Filedeskriptor setzen ---*/
if ( (wert = fcntl(fd, F_GETFD, 0)) < 0 )
    fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_GETFD");
wert |= FD_CLOEXEC;
if ( (wert = fcntl(fd, F_SETFD, 0)) < 0 )
    fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_SETFD");

/*--- Restlicher Code des Daemons ---
::::::::::::::::::
:::::::::::::::::: */

exit(0);
}

```

Programm 12.3 (**sperdaem.c**): Einrichten von Sperren für Dämonprozesse

In diesem Programm wird **ftruncate** verwendet, um die Datei mit den Prozeß-IDs zu leeren. Die Verwendung des Flags `O_TRUNC` bei der Funktion **open** anstelle von **ftruncate** wäre hier falsch, da dies die Datei leeren würde, selbst wenn sie gesperrt ist.

Um zu verhindern, daß bei einem **fork** oder **exec** die Datei mit den Prozeß-IDs offen bleibt, was nicht notwendig ist, wird das *close-on-exec*-Flag für diese Datei gesetzt.

12.2.5 Mögliche Probleme beim Sperren bis zum Dateieinde

In bestimmten Situationen ist beim Sperren bis zum Dateieinde (Angabe von 0 für `l_len` in der Struktur `flock`) Vorsicht geboten. Der Grund dafür liegt in der Tatsache, daß die meisten Systeme bei der Angabe von `SEEK_CUR` oder `SEEK_END` (in `l_whence`) diese Angabe unter Verwendung von `l_start` und der aktuellen Position des Schreib-/Lesezeigers oder der momentanen Dateigröße in ein absolutes Offset konvertieren. Oft möchte man jedoch eine Sperre einrichten, die immer bis zum Dateieinde gilt, selbst wenn sich die Größe der Datei nachfolgend ändert.

Beispiel*Demonstrationsprogramm zu Problemen beim Sperren bis Dateiende*

Das folgende Programm 12.4 (`sperfehl.c`) verdeutlicht die Gefahr, die bei Systemen besteht, die `SEEK_CUR` und `SEEK_END` in ein absolutes Offset umrechnen. Programm 12.4 (`sperfehl.c`) beschreibt eine große Datei (1 Megabyte) abwechselnd mit den Buchstaben A und B. Bei jedem Durchlauf der **for**-Schleife sperrt es den Bereich vom aktuellen Dateiende bis zu einem zukünftigen Dateiende und schreibt dann den Buchstaben A. Danach gibt dieses Programm den Bereich vom aktuellen Dateiende bis zu einem zukünftigen Dateiende frei und schreibt dann den Buchstaben B.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int
main(void)
{
    int    i, fd;

    /*---- Anlegen einer temporären Datei -----*/
    if ( (fd = open("tmpdatei", O_RDWR | O_CREAT | O_TRUNC,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0)
        fehler_meld(FATAL_SYS, "open-Fehler");

    for (i=1; i<=500000; i++) {
        /* Sperre von momentanen EOF bis EOF */
        if (schreibwarte_sperre(fd, 0, SEEK_END, 0) < 0)
            fehler_meld(FATAL_SYS, "Fehler bei schreibwarte_sperre");

        if (write(fd, "A", 1) != 1)
            fehler_meld(FATAL_SYS, "write-Fehler");

        if (sperre_aufheben(fd, 0, SEEK_END, 0) < 0)
            fehler_meld(FATAL_SYS, "Fehler bei sperre_aufheben");

        if (write(fd, "B", 1) != 1)
            fehler_meld(FATAL_SYS, "write-Fehler");
    }

    exit(0);
}
```

*Programm 12.4 (**sperfehl.c**): Problemprogramm bei Systemen, die `SEEK_END` in absolutes Offset umrechnen*

Bei Systemen, die `SEEK_END` in ein absolutes Offset konvertieren, wird dieses Programm zu Problemen führen. Nachdem man dieses Programm 12.4 (`sperfehl.c`) kompiliert und gelinkt hat

```
cc -o sperfehl sperfehl.c sperre.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ sperfehl
Fehler bei schreibwarte_sperre: No record locks available
$ wc -c tmpdatei
  122 tmpdatei
$
```

Abbildung 12.2 verdeutlicht das interne Ablaufgeschehen von Programm 12.4 (`sperfehl.c`).

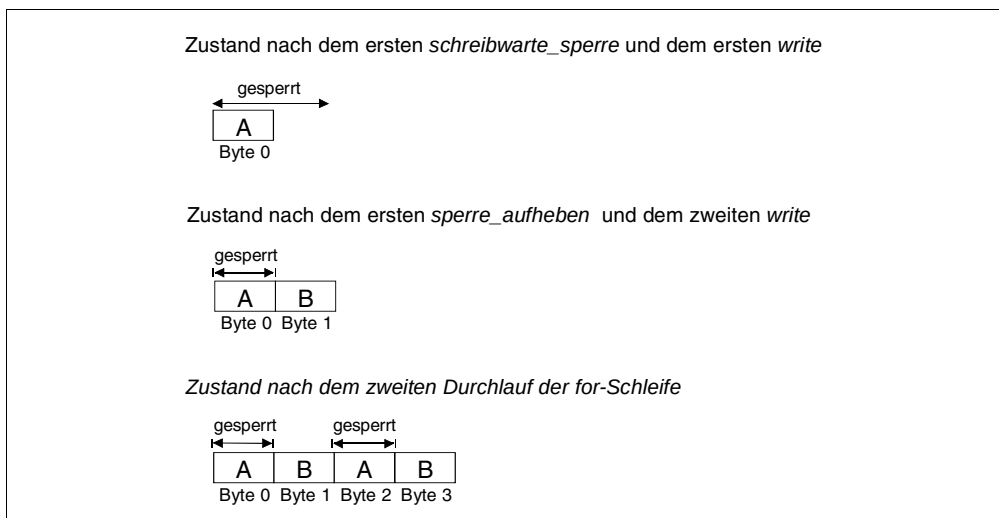


Abbildung 12.2: Eingerichtete Sperren durch Programm 12.4 (`sperfehl.c`)

Dies Abbildung zeigt, daß nur jedes 2. Byte gesperrt wird, was bei einer großen Datei früher oder später dazu führt, daß die vom Kern unterhaltenen internen Listen von `flock` Strukturen nicht mehr ausreichen und der nächste `fcntl`-Aufruf nicht erfolgreich ausgeführt werden kann. `fcntl` setzt hierbei `errno` auf `ENOLCK` und beendet sich mit einem Fehler.

Da wir in Programm 12.4 (`sperfehl.c`) wissen, wie viele Bytes wir jedesmal schreiben, können wir dort dieses Problem beseitigen, indem wir als zweites Argument (was `l_start` ist) für `sperre_aufheben` nicht 0, sondern die negative Zahl der Bytes angeben, die wir schreiben möchten. Da wir in Programm 12.4 (`sperfehl.c`) nur ein Byte schreiben, muß für dieses zweite Argument -1 angegeben werden. So wird jede Sperre (nach dem Schreiben von Buchstabe A) wieder aufgehoben.

12.2.6 Vererbung von Sperren

Für die Vererbung von Sperren auf Dateien oder Dateibereiche gelten die folgenden Regeln:

1. Wenn ein Prozeß sich beendet, so werden alle von diesem Prozeß eingerichteten Sperren freigegeben.
2. Wenn ein Filedeskriptor geschlossen wird, so werden im aktuellen Prozeß alle Sperren, die sich auf diesen Filedeskriptor und der zugehörigen Datei beziehen, freigegeben. Das bedeutet z.B., daß bei den folgenden vier Anweisungen

```
fd1 = open(dateiname,...);
schreib_sperre(fd1,...);
fd2 = dup(fd1);
close(fd2);
```

mit dem `close(fd2)` die Sperre, die für `fd1` eingerichtet wurde, aufgehoben wird. Würde der ***dup***-Aufruf durch ein ***open*** ersetzt, wie in

```
fd1 = open(dateiname,...);
schreib_sperre(fd1,...);
fd2 = open(dateiname,...);
close(fd2);
```

so wird auch hier mit `close(fd2)` die Sperre aufgehoben, die für `fd1` eingerichtet wurde.

3. Eingerichtete Sperren werden bei einem ***fork*** nicht an den Kindprozeß vererbt. Möchte der Kindprozeß die gleichen Sperren wie sein Elternprozeß besitzen, so muß er diese für die vom Elternprozeß geerbten Filedeskriptoren explizit mit ***fcntl*** einrichten. Der Grund dafür liegt in der Tatsache, daß Sperren das gleichzeitige Beschreiben durch verschiedene Prozesse verhindern sollen, und ein Kind- und Elternprozeß sind nun mal verschiedene Prozesse.
4. In SVR4 und BSD-Unix werden Sperren bei einem ***exec***-Aufruf vererbt. POSIX.1 schreibt dies nicht zwingend vor.

12.2.7 Starke Sperren (mandatory locking) in SVR4

In Unix unterscheidet man schwache und starke Sperren. In der englischen Originalliteratur werden diese mit *advisory locking* (wahlfreie Sperre) und *mandatory locking* (zwangswise Sperre) bezeichnet.

Starke Sperren können für eine Datei dadurch eingerichtet werden, daß man für diese das *Set-Group-ID*-Bit einschaltet und das *Group-Execute*-Bit ausschaltet.

Starke Sperren sind nicht Bestandteil von POSIX.1, werden aber von SVR4 unterstützt und bewirken, daß der Kern bei jedem ***open***, ***read*** und ***write*** überprüft, ob der aufrufende Prozeß nicht durch eine Sperre an dieser Aktion gehindert werden muß. Bei schwachen Sperren (*advisory locking*) findet eine solche Überprüfung nicht statt. Hier liegt es in der

Verantwortung des jeweiligen Programms, daß es selbst überprüft, ob Sperren vorliegen oder nicht.

Wenn ein Prozeß mittels **read** oder **write** versucht, einen durch einen anderen Prozeß (mit **fcntl**) gesperrten Bereich einer Datei, für die eine starke Sperre eingerichtet ist, zu lesen oder zu beschreiben, dann hängt das Verhalten des Systems von der vorliegenden Konstellation ab.

Tabelle 12.1 zeigt alle möglichen Konstellationen und die Auswirkung für jede einzelne von diesen Konstellationen.

	Blockierender Filedeskriptor		Nichtblockierender Filedeskriptor	
	read	write	read	write
Lesesperre auf Bereich	erfolgreich	Blockierung	erfolgreich	Fehler EAGAIN
Schreibsperre auf Bereich	Blockierung	Blockierung	Fehler EAGAIN	Fehler EAGAIN

Tabelle 12.1: Auswirkung von **read** und **write** durch andere Prozesse bei starken Sperren

Wenn ein Prozeß mit **open** versucht, eine Datei zu öffnen, für die eine starke Sperre eingerichtet ist, so gelten die in Tabelle 12.2 angegebenen Regeln.

	Flag O_TRUNC oder O_CREAT	
	gesetzt	keines von beiden gesetzt
open	Fehler EAGAIN	erfolgreich

Tabelle 12.2: Auswirkung von **open** auf eine Datei mit einer starken Sperre

Es ist jedoch darauf hinzuweisen, daß starke Sperren in SVR4 nicht unbedingt so »stark« sind, wie man vielleicht erwartet. So verhindern z.B. starke Sperren für eine Datei nicht das Löschen einer Datei mit **unlink**.

Um festzustellen, ob ein System starke Sperren (*mandatory locking*) unterstützt, kann das Programm 12.5 (**sperstar.c**) verwendet werden.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include "eighdr.h"

extern void add_fstatus_flags(int fd, int neuflags);

int
main(void)
{
    int          fd;
```



```

pid_t      pid;
char       puffer[10];
struct stat fstatpuff;

if ( (fd = open("tmpdatei", O_RDWR | O_CREAT | O_TRUNC,
               S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0)
    fehler_meld(FATAL_SYS, "open-Fehler");

if (write(fd, "Hallo", 5) != 5)
    fehler_meld(FATAL_SYS, "write-Fehler");

/*---- set-group-ID einschalten und group-execute ausschalten */
if (fstat(fd, &fstatpuff) < 0)
    fehler_meld(FATAL_SYS, "fstat-Fehler");
if (fchmod(fd, (fstatpuff.st_mode & ~S_IXGRP) | S_ISGID) < 0)
    fehler_meld(FATAL_SYS, "fchmod-Fehler");

INIT_SYNCH();

if ( (pid=fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");

else if (pid > 0) { /*----- Elternprozess -----*/
    if (schreib_sperre(fd, 0, SEEK_SET, 0) < 0) /* Ganze Datei fuer */
        fehler_meld(FATAL_SYS, "schreib_sperre-Fehler");/* Schreiben sperren*/

    HALLO_KIND(pid);

    if (waitpid(pid, NULL, 0) < 0)
        fehler_meld(FATAL_SYS, "waitpid-Fehler");

} else { /*----- Kindprozess -----*/
    WARTE_AUF_PAPA();

    add_fstatus_flags(fd, O_NONBLOCK);

    if (lese_sperre(fd, 0, SEEK_SET, 0) != -1) /* ohne Warten */
        fehler_meld(FATAL_SYS, "Kind: lese_sperre erfolgreich");
    printf("Lese-Sperre-Versuch fuer schon gesperrten Bereich liefert %d\n",
           errno);

    /*-- Versuch, die Datei mit starker Sperre zu lesen */
    if (lseek(fd, 0, SEEK_SET) == -1)
        fehler_meld(FATAL_SYS, "lseek-Fehler");

    if (read(fd, puffer, 3) < 0)
        fehler_meld(WARNUNG_SYS, "Lesen nicht erfolgreich "
                    "(starke Sperren unterstuetzt)");
    else
        printf("%3.3s' erfolgreich gelesen "
              "(starke Sperren nicht unterstuetzt)\n", puffer);
}

```

```

    exit(0);
}

/*----- Hinzufuegen von file status flags -----*/
void add_fstatus_flags(int fd, int neuflags)
{
    int fsflags;
    if ( (fsflags=fcntl(fd, F_GETFL, 0)) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_GETFL");
    fsflags |= neuflags; /*----- Hinzufuegen der neuen Flags */
    if (fcntl(fd, F_SETFL, fsflags) < 0 )
        fehler_meld(FATAL_SYS, "Fehler bei fcntl mit F_SETFL");
}

```

Programm 12.5 (sperstar.c): Feststellen, ob ein System starke Sperren unterstützt oder nicht

Das Programm 12.5 (sperstar.c) kreiert eine Datei und richtet für diese Datei eine starke Sperre ein, bevor es einen Kindprozeß startet. Der Elternprozeß setzt seinerseits eine Schreibsperre auf die ganze Datei, während der Kindprozeß seinen Filedeskriptor zunächst auf »Nicht-Blockieren« setzt, bevor er versucht, eine Lesesperre für die Datei einzurichten. Der dabei aufgetretene und erwartete Fehler (errno) wird ausgegeben.

Danach setzt der Kindprozeß den Schreib-/Lesezeiger auf den Dateianfang und versucht, aus der Datei zu lesen.

Wenn starke Sperren vom jeweiligen System unterstützt werden, so ist dieser Leseversuch nicht erfolgreich und liefert als Fehler entweder EACCES oder EAGAIN. Unterstützt das jeweilige System keine starke Sperren, so kann hier erfolgreich gelesen werden und die dabei gelesenen drei Zeichen werden ausgegeben.

Zunächst soll das Programm 12.5 (sperstar.c) kompiliert und gelinkt werden

```
cc -o sperstar sperstar.c sperre.c forksync.c fehler.c
```

Läßt man dieses Programm 12.5 (sperstar.c) unter SVR4 oder Solaris ablaufen, die starke Sperren unterstützen, so liefert es z.B. die folgende Ausgabe

```

$ sperstar
Lese-Sperre-Versuch fuer schon gesperrten Bereich liefert 11
Lesen nicht erfolgreich (starke Sperren unterstuetzt): No more processes
$

```

Auf diesem System hat EAGAIN die Nummer 11.

Läßt man dieses Programm 12.5 (sperstar.c) auf einem System ablaufen, das keine starken Sperren unterstützt, so liefert es die folgende Ausgabe:

```

$ sperstar
Lese-Sperre-Versuch fuer schon gesperrten Bereich liefert 11
'Hal' erfolgreich gelesen (starke Sperren nicht unterstuetzt)
$

```

Auf diesem System hat EAGAIN die Nummer 11.

12.3 Übung (Multiuser-Datenbankbibliothek)

Eine typische Anwendung für Sperren sind Datenbanken. Deshalb wird hier ein umfangreicheres Projekt vorgestellt, in dem eine einfache Multiuser-Datenbankbibliothek entwickelt werden soll. Diese Bibliothek soll eine Reihe von C-Routinen anbieten, die jedes Programm aufrufen kann, um Datensätze aus einer Datenbank zu erfragen oder dort zu speichern.

12.3.1 Schnittstellen der Bibliotheksdatenbank

Wie bei jedem Projekt müssen vor der eigentlichen Implementierung zuerst die Schnittstellen geklärt sein. Wir wollen hier die folgenden C-Routinen (Schnittstellen) anbieten.

Öffnen und Schließen der Datenbank

Zum Öffnen und Schließen einer Datenbank stellen wir die beiden folgenden Funktionen zur Verfügung.

```
#include "db.h"

DBANK *db_oeffne(const char *pfad, int oflag, int modus);

                                gibt zurück: DBANK-Zeiger (bei Erfolg); NULL bei Fehler

void db_schliesse(DBANK *db);
```

Der von **db_oeffne** gelieferte DBANK-Zeiger entspricht in der Funktionsweise dem FILE-Zeiger von **fopen** oder dem DIR-Zeiger von **opendir**. Dieser Zeiger muß bei den anderen Datenbankfunktionen dann als Argument übergeben werden.

Ist das Öffnen einer Datenbank mit **db_oeffne** erfolgreich, so werden dabei automatisch zwei Dateien kreiert: *pfad.dat* (Datendatei) und *pfad.idx* (Indexdatei).

Die beiden Argumente *oflag* und *modus* entsprechen dem gleichnamigen Argumenten der Funktion **open** (siehe Kapitel 4.2):

oflag

legt fest, wie die Datenbank zu öffnen ist. Z.B. legt `O_RDONLY` fest, daß die Datenbank nur zum Lesen geöffnet werden soll.

modus

legt die Zugriffsrechte für die Datenbank fest, wenn diese neue angelegt wird. Dies setzt wie bei **open** voraus, daß bei *oflag* die Konstante `O_CREAT` angegeben wurde.

db_schliesse gibt alle allokierten internen Puffer frei und schließt die Indexdatei und die Datenbankdatei (Datendatei).

Schreiben eines Datensatzes

Um einen Datensatz in einer mit **db_oeffne** geöffneten Datenbank abzuspeichern, steht die Funktion **db_schreibe** zur Verfügung.

```
#include "db.h"
```

```
int db_schreibe(DBANK *db, const char *schluessel,  
               const char *datensatz, int wie);
```

gibt zurück: 0 (bei Erfolg); verschieden von 0 bei Fehler

Zu jedem *datensatz* muß ein sogenannter Schlüssel angegeben werden. Wenn z.B. eine Datenbank die Daten zu den Studenten einer Universität enthält, so könnte der Schlüssel die Matrikelnummer sein und der *datensatz* könnte den Namen, Adresse, Telefonnummer usw. des jeweiligen Studenten enthalten. Die Schlüssel der Datensätze einer Datenbank müssen alle unterschiedlich sein. Es können also niemals mehrere Studenten die gleiche Matrikelnummer besitzen.

Sowohl *schluessel* als auch *datensatz* müssen Strings sein, die mit `\0` abgeschlossen sind. Zusätzlich ist gefordert, daß diese Strings niemals leer sein dürfen.

Für *wie* ist entweder `DB_EINFUEGE` (um einen Datensatz einzufügen) oder `DB_UEBERSCHREIBE` (um einen existierenden Datensatz zu überschreiben) anzugeben. Diese zwei Konstanten sind in der Headerdatei `db.h` definiert.

Wenn bei der Angabe von `DB_EINFUEGE` der entsprechende Datensatz schon existiert, so liefert **db_schreibe** als Rückgabewert 1.

Wenn bei der Angabe von `DB_UEBERSCHREIBE` der entsprechende Datensatz nicht existiert, so liefert **db_schreibe** als Rückgabewert -1.

Lesen eines Datensatzes

Um einen Datensatz aus einer mit **db_oeffne** geöffneten Datenbank zu lesen, steht die Funktion **db lese** zur Verfügung.

```
#include "db.h"
```

```
char *db lese(DBANK *db, const char *schluessel);
```

gibt zurück: Zeiger auf Datensatz (bei Erfolg); NULL, wenn kein Datensatz gefunden wurde

Löschen eines Datensatzes

Um einen Datensatz in einer mit **db_oeffne** geöffneten Datenbank zu löschen, steht die Funktion **db_loesche** zur Verfügung.

```
#include "db.h"

int db_loesche(DBANK *db, const char *schluessel);

gibt zurück: 0 (bei Erfolg); -1, wenn kein Datensatz gefunden wurde
```

Sukzessives Lesen aus der Datenbank

Um aus einer mit **db_oeffne** geöffneten Datenbank sukzessive zu lesen, stehen die beiden Funktionen **db_anfang** und **db_naechstdatsatz** zur Verfügung.

```
#include "db.h"

void db_anfang(DBANK *db);

char *db_naechstdatsatz(DBANK *db, char *schluessel);

gibt zurück: Zeiger auf Datensatz (bei Erfolg); NULL bei Dateiende
```

Um sukzessive zu lesen, muß zunächst mit **db_anfang** auf den ersten Datensatz der Datenbank positioniert werden. Mit Aufrufen von **db_naechstdatsatz** können dann die Datensätze der Datenbank nacheinander gelesen werden.

Wird bei **db_naechstdatsatz** für *schluessel* ein NULL-Zeiger abgegeben, dann wird der nächste Datensatz in der Datenbank gelesen. Wird dagegen für *schluessel* ein wirklicher Schlüssel angegeben, dann wird der Datensatz mit diesem *schluessel* gelesen. Dieser *schluessel* ist dann auch die neue Position, auf die sich der nächste **db_naechstdatsatz** bezieht.

Die Reihenfolge, in der **db_naechstdatsatz** liest, ist nicht festgelegt. Es ist lediglich garantiert, daß mit **db_naechstdatsatz** jeder Datensatz gelesen wird. Wenn wir z.B. drei Datensätze mit den Schlüsseln A, B und C (in dieser Reihenfolge) in die Datenbank geschrieben haben, so ist nicht festgelegt, in welcher Reihenfolge sie aus dieser Datei durch **db_naechstdatsatz** gelesen werden.

Die Reihenfolge ist dabei von der Implementierung abhängig. So können diese Datensätze z.B. in der Reihenfolge C, A, B gelesen werden.

Der folgende Codeausschnitt zeigt eine typische Verwendung der beiden Funktionen **db_anfang** und **db_naechstdatsatz**.

```
db_anfang(db);
while ( (zgr = db_naechstdatsatz(db, schluessel)) != NULL ) {
    ::::
    ::::
}
```

Er liest sukzessive alle Datensätze einer Datenbank.

12.3.2 Überblick zur Implementierung der Bibliotheksdatenbank

Hier wird ein Überblick über die Implementierung unserer Bibliotheksdatenbank gegeben.

Organisationsstruktur der Indexdatei

Die meisten Datenbankbibliotheken verwenden zwei Dateien zum Speichern der Informationen: eine Indexdatei und eine Datendatei. Die Indexdatei enthält den aktuellen Indexwert (Schlüssel) und einen Zeiger auf den entsprechenden Datensatz in der Datendatei. Um ein schnelles Auffinden eines Schlüssels zu ermöglichen, soll hier für die Indexdatei als Organisationsform eine Hashtabelle mit verketteten Listen verwendet werden.

Speicherung der Schlüssel und Indizes

In dieser Implementierung werden die Schlüssel und Indizes als Strings (mit \0 abgeschlossen) gespeichert. Andere Datenbanksysteme speichern numerische Schlüssel und Indizes oft in einem Binärformat (z.B. 2 oder 4 Byte für ganze Zahlen), um Speicherplatz zu sparen. Diese Vorgehensweise hat jedoch den Nachteil, daß diese Datenbankdateien oft nicht auf andere Systeme portiert werden können, wenn diese mit einem anderen Binärformat arbeiten. Abbildung 12.3 zeigt einen möglichen Aufbau der Indexdatei und Datendatei.

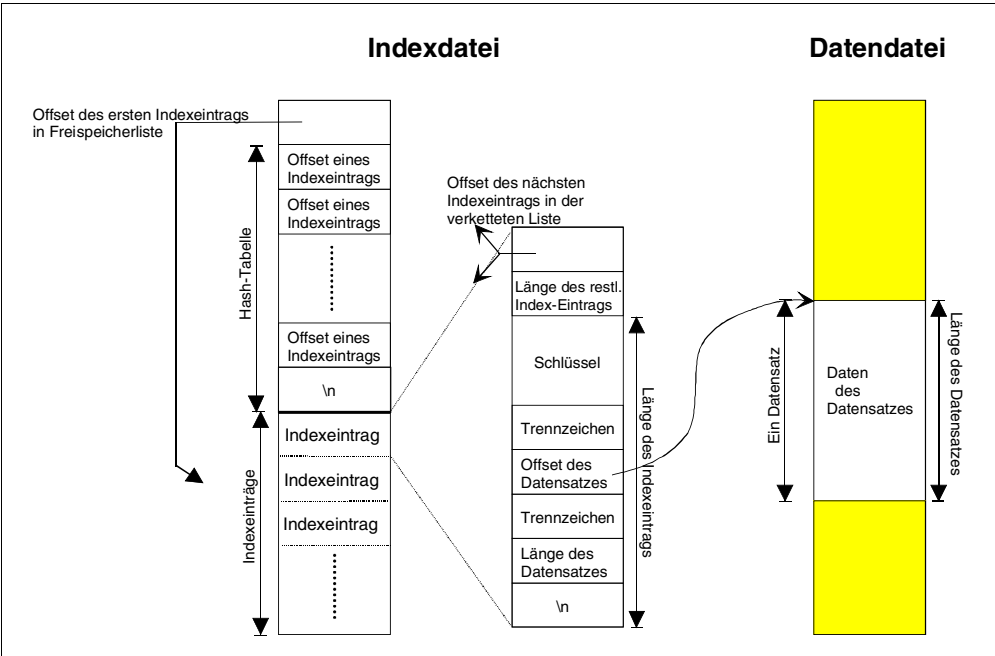


Abbildung 12.3: Möglicher Aufbau der Index- und Datendatei

Die Indexdatei besteht aus drei Teilen:

- ▶ dem Offset des ersten Indexeintrags in der Freispeicherliste
- ▶ einer Hashtabelle, die die Offsets der Indexeinträge enthält
- ▶ einer sequentiellen Liste der Indexeinträge

Um einen Eintrag in der Datenbank zu finden, berechnet die Funktion **db lese** zum übergebenen Schlüssel den Hashwert.

Dieser Hashwert liefert den Offset des ersten Indexeintrags einer möglicherweise verketteten Liste. Das Ende einer verketteten Liste lässt sich hier am Wert 0 als Offset des nächsten Indexeintrags in der verketteten Liste erkennen.

Beispiel

Kreieren einer Datenbank und Schreiben von drei Datensätze

Das Programm 12.6 (dbeinf.c) kreiert eine neue Datenbank und schreibt drei Datensätze in diese Datenbank.

```
#include "db.h"

int
main(void)
{
    DBANK *db;

    if ( (db = db_oeffne("einf", O_RDWR | O_CREAT | O_TRUNC,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == NULL)
        fehler_meld(FATAL_SYS, "kann Datenbank 'einf' nicht oeffnen");

    if (db_schreibe(db, "Eins", "datsatz_one", DB_EINFUEGE) != 0)
        fehler_meld(FATAL, "kann 'datsatz_one' (Schl. Eins) nicht schreiben");

    if (db_schreibe(db, "Zwei", "datsatz_two", DB_EINFUEGE) != 0)
        fehler_meld(FATAL, "kann 'datsatz_two' (Schl. Zwei) nicht schreiben");

    if (db_schreibe(db, "Drei", "datsatz_three", DB_EINFUEGE) != 0)
        fehler_meld(FATAL, "kann 'datsatz_three' (Schl. Drei) nicht schreiben");

    db_schliesse(db);
    exit(0);
}
```

Programm 12.6 (dbeinf.c): Kreieren einer Datenbank und Schreiben von 3 Datensätzen in diese

Zunächst kompilieren und linken wir dieses Programm:

```
cc -o dbeinf dbeinf.c db.c sperre.c fehler.c -lm
```

Da wir alle Einträge in der Datenbank als ASCII-Zeichen hinterlegen, kann man nach dem Start von `dbeinf` sowohl den Inhalt der erzeugten Daten- als auch den der erzeugten Indexdatei lesen.

```
$ dbeinf           [Kreieren der Datenbank einf, mit Schreiben von 3 Datensätzen]
$ ls -l einf*
-rw-r--r--  1 hh      grafik          38 Jun 28 16:51 einf.dat
-rw-r--r--  1 hh      grafik        105 Jun 28 16:51 einf.idx
$ cat einf.dat
datatz_one
datatz_two
datatz_three
$ cat einf.idx    [Zur Demonstration wurde hier nur 5 als Hashtab.-Größe verwendet]
 0    0  59    0    0  82
 0   10Eins:0:12
 0   11Zwei:12:12
37   11Drei:24:14
$
```

Um die Übersichtlichkeit in diesem Beispiel zu wahren, wurde hier (nicht wie im wirklichen Programm) die Hashtabellengröße von 5 angenommen.

Die erste Zeile in der Indexdatei

```
0    0  59    0    0  82
```

gibt als erstes das Offset des ersten Indexeintrags in der Freispeicherliste (0 = Freispeicherliste ist leer) und die fünf Offseteinträge in der Hashtabelle (0,59,0,0,82) an.

Die zweite Zeile

```
0   10Eins:0:12
```

ist der erste Indexeintrag mit folgender Bedeutung:

1. Feld (0)	Kein weiterer Indexeintrag mit gleichem Hashwert (in dieser verketteten Liste)
2. Feld (10)	Länge des restlichen Indexeintrags
3. Feld (Eins)	Schlüssel
4. Feld (:))	Feldtrennzeichen
5. Feld (0)	Offset des zugehörigen Datensatzes in Datendatei <code>einf.dat</code>
6. Feld (:))	Feldtrennzeichen
7. Feld (12)	Länge des zugehörigen Datensatzes in Datendatei <code>einf.dat</code>

Bei den beiden Längenangaben für den Indexeintrag und den Datensatz ist zu beachten, daß immer automatisch am Ende ein Neue-Zeile-Zeichen (`\n`) anhängt wird.

In der vierten Zeile

```
37      11Drei:24:14
```

haben wir im 1.Feld den Wert 37. Dieser Wert ist das Offset des nächsten Indexeintrags (mit gleichem Hashwert) in dieser verketteten Liste. Das Offset 37 hat der erste Indexeintrag (in der 2.Zeile).

Sperren von Einträgen

Wenn mehrere Prozesse gleichzeitig auf dieselbe Datenbank zugreifen, dann müssen Sperren für Dateibereiche eingerichtet sein, um die Konsistenz der Datenbank zu gewährleisten. Bei unserer Datenbank sollen folgende Zugriffsbedingungen gelten:

1. Gleichzeitiges Lesen der Hashtabelle durch mehrere Prozesse ist erlaubt.
2. Das gleichzeitige Schreiben in die Hashtabelle durch mehrere Prozesse soll niemals möglich sein.
3. Ein Prozeß, der auf die Freispeicherliste (mit **db_loesche** oder **db_schreibe**) zugreift, muß immer eine Schreibsperre auf die Freispeicherliste einrichten.
4. Wenn die Funktion **db_schreibe** einen neuen Eintrag an das Ende der Index- oder Datendatei schreibt, dann muß sie für diesen Bereich eine Schreibsperre einrichten.

Die Freispeicherliste

Die Freispeicherliste ist eine Liste von gelöschten Indexeinträgen. Wenn ein Eintrag gelöscht wird, dann wird der entsprechende Index- und Dateneintrag mit Leerzeichen überschrieben und das Offset dieses Eintrags am Anfang der Freispeicherliste eingefügt.

Die gelöschten Einträge in der Freispeicherliste werden beim Schreiben von neuen Datensätzen wieder verwendet, wenn die Länge des neuen Datensatzes und die Länge des zugehörigen Schlüssels genau den entsprechenden Längen des gelöschten Eintrags entsprechen. In diesem Fall muß der entsprechende Eintrag aus der Freispeicherliste entfernt werden.

12.3.3 Die Headerdatei db.h

Die Headerdatei für das hier zu erstellende Datenbankmodul db.c hat z.B. folgendes Aussehen:

```
#ifndef _DB
#define _DB

#include <sys/types.h>
#include <sys/stat.h> /* fuer modus-Argument von open und db_oeffne */
#include <fcntl.h>     /* fuer oflag-Argument von open und db_oeffne */
#include <math.h>
#include <stddef.h>
#include "eighdr.h"
```

```

/*----- Limits -----*/
#define MAX_NAM_LAENG 100

/*----- Konstanten fuer Argument 'wie' bei db_speichere -----*/
#define DB_EINFUEGE 0
#define DB_UEBERSCHREIBE 1

/*----- Festlegung des Trennzeichens -----*/
#define TRENNZ ':' /* Trennzeichen im Index-Eintrag */

/*----- Festlegung der einzelnen Feldgroessen in einem Index-Eintrag */
#define IDX_LAENGE_GR 6 /* Groesse eines Index-Laenge-Felds */
#define IDX_LAENGE_MIN 6 /* Min. Laenge eines Indexeintrags */
#define IDX_LAENGE_MAX 2000 /* Max. Laenge eines Indexeintrags */
#define DAT_LAENGE_MIN 2 /* Min. Laenge eines Datensatzes */
#define DAT_LAENGE_MAX 2000 /* Max. Laenge eines Datensatzes */

/*----- Festlegung der Offset-Position fuer Freispeicherliste -----*/
#define FREI_OFFSET 0

/*----- Festlegung der einzelnen Groessen fuer Hashtabelle -----*/
#define OFFSET_GROESSE 6 /* Laenge eines Index-Offsets */
#define OFFSET_MAX pow(10, OFFSET_GROESSE+1) /* Max. Groesse von */
/* Index-Offset */
#define HASH_GROESSE 997 /* Groesse der Hashtabelle */
#define HASH_ANFANG OFFSET_GROESSE /* Beginn der Hashtabelle */
/* in der Index-Datei */

/*----- Selbstdefinierter Datentyp 'DBANK' -----*/
typedef struct {
    char name[MAX_NAM_LAENG]; /* Name der eroeffneten Datenbank */
    int datfd; /* Filedeskriptor fuer Datendatei */
    int idxfd; /* Filedeskriptor fuer Indexdatei */
    int oflag; /* Art der Eroeffnung (wie O_RDONLY fuer nur-Lesen) */
    char idxpuff[IDX_LAENGE_MAX]; /* Puffer fuer Indexeintrag */
    off_t idxoffset; /* Offset eines Indexeintrags in Indexdatei */
    size_t idxlaenge; /* Laenge eines Indexeintrags */
    /* (ohne IDX_LAENGE_GR Bytes am Anfang; */
    /* mit \n am Ende des Indexeintrags) */
    off_t idxnaechst; /* Offset des naechst. Index-Eintr. in Indexdat. */

    char datpuff[DAT_LAENGE_MAX]; /* Puffer fuer Datensatz */
    off_t datoffset; /* Offset eines Datensatzes in Datendatei */
    size_t datlaenge; /* Laenge eines Datensatzes (einschl. \n am Ende) */

    off_t hash_anfang; /* Beginn der Hashtabelle */
    /* in der Indexdatei */
    unsigned long hash_groesse; /* aktuelle Hashtabelle-Groesse */

    off_t offset_off; /* Offset des Offsets fuer akt. Indexeintrag */
    off_t ketten_off; /* Offset des Beginns der verketteten Liste */

    long leseok_zaehl; /* erfolgreiche Leseoperationen */

```

```

long lesefehl_zaehl; /* aufgetretene Lesefehler */
long loeschok_zaehl; /* erfolgreiche Loeschoperationen */
long loeschfehl_zaehl; /* aufgetretene Loeschfehler */

long schreibfehl_zaehl; /* aufgetretene Schreibfehler */
long einf_anh_zaehl; /* DB_EINFUEGE, kein freier Pl.->Angehaengt */
long einf_einf_zaehl; /* DB_EINFUEGE, freier Platz -> Eingefuegt */
long ueber_anh_zaehl; /* DB_UEBER..., verschied. lang->Angehaengt */
long ueber_einf_zaehl; /* DB_UEBER..., gleich lang->Ueberschrieb. */

long naechstdsatz_zaehl; /* zaehlt db_naechstdsatz hoch */
} DBANK;

/*===== Global aufrufbare Routinen =====*/

extern DBANK *db_oeffne(const char *pfad, int oflag, int modus);
extern void db_schliesse(DBANK *db);
extern int db_schreibe(DBANK *db, const char *schluessel,
                      const char *datensatz, int wie);
extern char *db lese(DBANK *db, const char *schluessel);
extern int db_loesche(DBANK *db, const char *schluessel);
extern void db_anfang(DBANK *db);
extern char *db_naechstdsatz(DBANK *db, char *schluessel);

#endif

```

Programm 12.7 (**db.h**): Headerdatei zum Datenbank-Modul *db.c*

12.3.4 Testen der Datenbank

Zum Testen der erstellten Datenbank kann das folgende Programm 12.8 (*zufalldb.c*) verwendet werden. Dieses Programm erwartet zwei Kommandozeilenargumente:

- die Anzahl der Kindprozesse, die es kreieren soll, und
- die Anzahl der von jedem Kindprozeß zu schreibenden Datenbankeinträge (n).

```

#include <stdlib.h>
#include <sys/wait.h>
#include "db.h"

#define DB_NAME      "test.db"
#define MAX_PROZESSE 1000
#define MAX_EINTRAEGE 10000

static void datenbank_zugriffe(pid_t pid);
static void statistik_wert_update(DBANK *db, const char *schluessel,
                                long wert);
static void statistik_wert_ausgeben(DBANK *db, const char *schluessel,
                                const char *kommentar);

static long    anz_kinder,
               anz_eintraege;
static pid_t   vater_pid,

```

```

static int      pid_benutzt[MAX_PROZESSE];
pid_zahl=0;

/*--- main -----*/
int
main(int argc, char *argv[])
{
    long        i;
    pid_t        pid;
    DBANK        *db;
    char        schluesssel[20], *dsatz;

    if (argc != 3)
        fehler_meld(FATAL, "usage: %s anz_kindprozesse anz_eintraege", argv[0]);

    /*--- Argumente in Zahlen umwandeln */
    if ( (anz_kinder = atol(argv[1])) == 0 ||
        anz_kinder < 0 || anz_kinder > MAX_PROZESSE)
        fehler_meld(FATAL, "%s ist keine gueltige Zahl", argv[1]);
    if ( (anz_eintraege = atol(argv[2])) == 0 ||
        anz_eintraege < 0 || anz_eintraege > MAX_EINTRAEGE)
        fehler_meld(FATAL, "%s ist keine gueltige Zahl", argv[2]);

    /*--- Erzeugen der Kindprozesse, die nun auf die Datenbank zugreifen */
    vater_pid = getpid();
    for (i=1; i<=anz_kinder; i++) {
        if (getpid() == vater_pid)
            if ( (pid = fork()) > 0)
                pid_benutzt[pid_zahl++] = pid;
            else
                datenbank_zugriffe(getpid());
    }

    /*--- Auf das Ende aller Kindprozesse warten */
    for (i=0; i<pid_zahl; i++) {
        if (waitpid(pid_benutzt[i], NULL, 0) < 0)
            fehler_meld(FATAL_SYS, "waitpid-Fehler");
    }

    /*--- Inhalt der jetzt vorhandenen Datenbank lesen */
    if ( (db = db_oeffne(DB_NAME, O_RDONLY, 0)) == NULL)
        fehler_meld(FATAL_SYS, "kann Datenbank %s nicht oeffnen", DB_NAME);

    printf("Inhalt der Datenbank\n"
           "=====\n\n"
           "      Schluesssel:Datsatz\n");
    db_anfang(db);

    while ( (dsatz = db_naechstdatsatz(db, schluesssel)) != NULL)
        printf("%20s:%s\n", schluesssel, dsatz);

    printf("\n\n"
           "Statistik ueber Datenbank-Operationen\n"

```

```

    "=====\\n\\n");
    printf("-----\\n");
    statistik_wert_ausgeben(db, "leseok_zahl", "Erfolgreiches Lesen");
    statistik_wert_ausgeben(db, "lesefehl_zahl", "Fehlerhaftes Lesen");
    printf("-----\\n");
    statistik_wert_ausgeben(db, "loeschok_zahl", "Erfolgreiches Loeschen");
    statistik_wert_ausgeben(db, "loeschfehl_zahl", "Fehlerhaftes Loeschen");
    printf("-----\\n");
    statistik_wert_ausgeben(db, "schreibfehl_zahl", "Fehlerhaftes Schreiben");
    printf("-----\\n");
    statistik_wert_ausgeben(db, "einf_anh_zahl",
        "Bei Einfuegen kein freier Platz gefunden -> Angehaengt");
    statistik_wert_ausgeben(db, "einf_einf_zahl",
        "Bei Einfuegen freier Platz gefunden -> Eingefuegt");
    statistik_wert_ausgeben(db, "ueber_anh_zahl",
        "Bei Ueberschreiben verschieden lang -> Angehaengt");
    statistik_wert_ausgeben(db, "ueber_einf_zahl",
        "Bei Ueberschreiben gleich lang -> Ueberschrieben");
    printf("-----\\n");

    db_schliesse(db);

    exit(0);
}

/*--- datenbank_zugriffe -----
 *      fuehrt eine Vielzahl von zufaelligen Datenbankzugriffen aus */

static void datenbank_zugriffe(pid_t pid)
{
    DBANK      *db;
    int         i, j;
    char        schluesel[20],
               datsatz[50],
               *dsatz;
    long        zaehler=0;

    /*--- Zufallszahlengenerator initialisieren -----*/
    srand(time(NULL));

    /*--- Datenbank oeffnen -----*/
    if ( (db = db_oeffne(DB_NAME, O_RDWR | O_CREAT | O_TRUNC,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == NULL)
        fehler_meld(FATAL_SYS, "kann Datenbank %s nicht oeffnen", DB_NAME);

    /*--- anz_eintraege in Datenbank schreiben -----*/
    for (i=1; i<=anz_eintraege; i++) {
        sprintf(schluesel, "%d", i);
        sprintf(datsatz, "%x%d", i, pid);
        db_schreibe(db, schluesel, datsatz, DB_EINFUEGE);
    }

    /*--- anz_eintraege aus Datenbank lesen -----*/

```

```

for (i=1; i<=anz_eintraege; i++) {
    sprintf(schluesssel, "%d", i);
    dsatz = db lese(db, schluesssel);
}

/*--- 10 x anz_eintraege Schleife -----*/
for (i=1; i<=10; i++) {
    for (j=1; j<=anz_eintraege; j++) {
        /*--- Jedes 100. mal einen zufaellichen Eintrag loeschen */
        if (zaehler % 100 == 0) {
            sprintf(schluesssel, "%d", rand()%anz_eintraege + 1);
            db_loesche(db, schluesssel);
        }
        /*--- Jedes 500. mal einen nicht existierenden Eintrag loeschen */
        if (zaehler % 500 == 0)
            db_loesche(db, "nicht vorhanden");

        /*--- Jedes 10. mal einen zufaellichen Eintrag ueberschreiben */
        if (zaehler % 10 == 0) {
            sprintf(schluesssel, "%d", rand()%anz_eintraege + 1);
            sprintf(datsatz, "ue_%d", zaehler%1000);
            db_schreibe(db, schluesssel, datsatz, DB_UEBERSCHREIBE);
        }
        zaehler++;
    }
}

/*--- Statistik ueber Datenbankzugriffe in Datenbank selbst schreiben */
statistik_wert_update(db, "leseok_zaehl", db->leseok_zaehl);
statistik_wert_update(db, "lesefehl_zaehl", db->lesefehl_zaehl);
statistik_wert_update(db, "loeschok_zaehl", db->loeschok_zaehl);
statistik_wert_update(db, "loeschfehl_zaehl", db->loeschfehl_zaehl);
statistik_wert_update(db, "schreibfehl_zaehl", db->schreibfehl_zaehl);
statistik_wert_update(db, "einf_anh_zaehl", db->einf_anh_zaehl);
statistik_wert_update(db, "einf_einf_zaehl", db->einf_einf_zaehl);
statistik_wert_update(db, "ueber_anh_zaehl", db->ueber_anh_zaehl);
statistik_wert_update(db, "ueber_einf_zaehl", db->ueber_einf_zaehl);

/*--- Datenbank schliessen -----*/
db_schliesse(db);

exit(0);
}

/*--- statistik_wert_update -----
* schreibt Statistik ueber eine Art des Datenbank-Zugriffs
* in die Datenbank selbst */

static void statistik_wert_update(DBANK *db, const char *schluesssel,
    long wert)
{
    char *dsatz,
        datsatz[50];
    long zahl;

```

```

    dsatz = db lese(db, schluesssel);
    zahl = (dsatz==NULL) ? 0 : atol(dsatz);
    zahl += wert;
    sprintf(datsatz, "%ld", zahl);
    if (dsatz == NULL)
        db_schreibe(db, schluesssel, datsatz, DB_EINFUEGE);
    else
        db_schreibe(db, schluesssel, datsatz, DB_UEBERSCHREIBE);
}

/*--- statistik_wert_ausgeben -----
 *      liest Statistik ueber eine Art des Datenbankzugriffs
 *      aus der Datenbank                                     */

static void statistik_wert_ausgeben(DBANK *db, const char *schluesssel,
                                   const char *kommentar)
{
    char    *dsatz;
    long    zahl;

    dsatz = db lese(db, schluesssel);
    zahl = atol(dsatz);
    printf("%54s: %10ld\n", kommentar, zahl);
}

```

Programm 12.8 (**zufalldb.c**): Datenbanktest mittels gleichzeitiger Zugriffe durch Kindprozesse

Dieses Programm läßt dann jeden Kindprozeß die Datenbank öffnen, n Datensätze dorthin schreiben und diese wieder lesen.

Zusätzlich läßt es jeden Kindprozeß zu Testzwecken noch existierende und nicht existierende Datensätze löschen und Datensätze überschreiben.

Bevor sich jeder Kindprozeß beendet, schreibt er die Anzahl seiner erfolgreichen bzw. fehlgeschlagenen Operationen (Lesen, Löschen, Schreiben,...) in die Datenbank. Dazu liest er zunächst die eventuell schon von anderen Prozessen geschriebenen Statistikwerte zu diesen Operationen, addiert seine Werte und überschreibt die alten Werte in der Datenbank mit den neuen Werten. Alle Prozesse verwenden dabei für die einzelnen Statistikwerte die gleichen Schlüssel. Somit befindet sich am Ende des Programms die Gesamtstatistik zu den Datenbankzugriffen der einzelnen Prozesse in der Datenbank selbst. Sie muß also nach der Beendigung der Kindprozesse nur noch vom Elternprozeß aus der Datenbank gelesen und auf der Standardausgabe ausgegeben werden.

Vor dieser Statistikausgabe gibt der Elternprozeß jedoch mittels **db_anfang** und **db_naechstdatsatz** zunächst den Inhalt der gesamten Datenbank aus.

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -c zufalldb zufalldb.c db.c sperre.c fehler.c -lm
```

kann man seine Datenbank testen, wie z.B.:

\$ zufalldb 20 31 [20 Kindprozesse mit jeweils 31 Einträgen]

Inhalt der Datenbank

=====

Schluessel:Datensatz
1:1_225
5:ue_90
8:ue_20
10:a_225
11:b_225
12:c_225
18:ue_190
19:ue_110
21:15_225
22:16_225
24:18_225
25:ue_230
27:1b_225
29:1d_225
30:ue_300
31:1f_225
lesefehl_zahl:1
einf_anh_zahl:36
loeschok_zahl:80
leseok_zahl:620
einf_einf_zahl:76
ueber_einf_zahl:789
schreibfehl_zahl:40
ueber_anh_zahl:455
loeschfehl_zahl:20
20:ue_70
16:ue_80
3:ue_100
7:ue_240
6:ue_140
15:ue_150
2:ue_180
13:ue_260
9:ue_270
23:ue_280
4:ue_290

Statistik ueber Datenbank-Operationen

=====

-----	Erfolgreiches Lesen:	620
	Fehlerhaftes Lesen:	1
-----	Erfolgreiches Loeschen:	80
	Fehlerhaftes Loeschen:	20
-----	Fehlerhaftes Schreiben:	40


```

-----
Bei Einfuegen kein freier Platz gefunden -> Angehaengt:      36
    Bei Einfuegen freier Platz gefunden -> Eingefuegt:      76
    Bei Ueberschreiben verschieden lang -> Angehaengt:     455
    Bei Ueberschreiben gleich lang -> Ueberschrieben:      789
-----

```

\$ zufalldb 10 5 [10 Kindprozesse mit jeweils 5 Einträgen]

Inhalt der Datenbank

=====

```

        Schluessel:Datensatz
            1:ue_30
            3:ue_40
            5:5_247
lesefehl_zahl:1
schreibfehl_zahl:0
einf_anh_zahl:10
einf_einf_zahl:9
leseok_zahl:50
ueber_anh_zahl:23
ueber_einf_zahl:134
            4:ue_10
loeschok_zahl:10
loeschfehl_zahl:10

```

Statistik ueber Datenbank-Operationen

=====

```

-----
                                Erfolgreiches Lesen:      50
                                Fehlerhaftes Lesen:        1
-----
                                Erfolgreiches Loeschen:     10
                                Fehlerhaftes Loeschen:     10
-----
                                Fehlerhaftes Schreiben:    0
-----
Bei Einfuegen kein freier Platz gefunden -> Angehaengt:    10
    Bei Einfuegen freier Platz gefunden -> Eingefuegt:      9
    Bei Ueberschreiben verschieden lang -> Angehaengt:     23
    Bei Ueberschreiben gleich lang -> Ueberschrieben:     134
-----

```

\$

Erstellen Sie nun das Programm db.c, das die zuvor beschriebene Aufgabenstellung erfüllt.

13 Signale

*Das Schicksal mischt die Karten,
und wir spielen.*

Schopenhauer

Signale sind sogenannte *Interrupts* (Unterbrechungen), die von der Hardware oder Software erzeugt werden, wenn während einer Programmausführung besondere Ausnahmesituationen auftreten, wie z.B. Division durch 0 oder Drücken der Programmabbruchtaste (*Strg-C* oder *DELETE*) durch den Benutzer. Das Signalkonzept wurde zwar schon in den ersten Unix-Versionen angeboten, war aber dort noch äußerst unzuverlässig. Mit 4.3BSD und SVR3 wurde das Signal-Modell sicherer; es wurden sogenannte *reliable signals* (zuverlässige Signale) eingeführt. POSIX.1 standardisierte später die zuverlässigen Signalaroutinen, die wir hier beschreiben.

In diesem Kapitel wird zunächst das Signalkonzept und die Funktion **signal** vorgestellt, bevor ein Überblick über die unterschiedlichen Arten von Signalen gegeben wird. Bevor das neue zuverlässige Signalkonzept behandelt wird, wird kurz auf die Schwäche des alten Signalkonzeptes eingegangen. Daneben werden die Routinen **kill** und **raise** behandelt, die das Schicken von Signalen ermöglichen. Ein weiteres Unterkapitel beschäftigt sich mit dem Einrichten einer Zeitschaltuhr und dem Suspendieren eines Prozesses, bevor kurz auf die anormale Beendigung eines Prozesses und auf die nicht standardisierten zusätzlichen Argumente eingegangen wird, die einige Systeme für Signalhandler anbieten.

13.1 Das Signalkonzept und die Funktion signal

Signale sind asynchrone Ereignisse, die zu nicht vorhersagbaren Zeitpunkten bei der Ausführung eines Prozesses auftreten können. Einige solcher möglichen asynchronen Ereignisse sind z.B.:

- ▶ Drücken der Programmabbruchtaste (meist *Strg-C* oder *DELETE*) durch den Benutzer.
- ▶ Illegitime Hardware-Operationen, wie z.B. Division durch 0 oder Zugriff auf unerlaubte Speicheradressen. Solche Ereignisse werden üblicherweise von der Hardware entdeckt, die den Kern darüber informiert. Der Kern schickt dann seinerseits dem betreffenden Prozeß das entsprechende Signal, wie z.B. *SIGFPE* bei Division durch 0.

- ▶ Signale von der Funktion **kill**. Mit der **kill**-Funktion kann ein Prozeß einem anderen Prozeß Signale schicken, soweit er die dazu nötigen Rechte besitzt.
- ▶ Softwaresignale, um den entsprechenden Prozeß über das Eintreten von bestimmten Ereignissen zu informieren. Solche Softwaresignale sind z. B. das Schreiben in einer Pipe, zu der kein Leser existiert (SIGPIPE) oder der Ablauf einer zuvor eingerichteten Zeitschaltuhr (SIGALRM).

13.1.1 Das Signalkonzept

Bei asynchronen Ereignissen wie den Signalen kommt man mit dem üblichen Konzept des Überprüfens von Variablen, wie z.B. der Überprüfung der Variablen `errno`, um das Auftreten eines Fehlers zu entdecken, nicht aus. Bei Signalen braucht man ein anderes Konzept, das man als *Signalkonzept* bezeichnet. Bei diesem Signalkonzept richtet ein Prozeß sogenannte **Signalhandler** ein, indem er dem Kern sagt: *Wenn dieses bestimmte Signal auftritt, dann tue bitte folgendes!* Solche Signalhandler lassen sich mit der Funktion **signal** einrichten.

13.1.2 signal – Einrichten von Signalhandlern

Mit der ANSI C-Funktion **signal** kann man dem Kern mitteilen, was zu tun ist, wenn ein bestimmtes Signal auftritt.

```
#include <signal.h>
```

```
void (*signal(int signr, void (*sighandler)(int)))(int);
```

gibt zurück: Adresse des zuvor eingerichteten Signalhandlers

Das Argument `signr` legt die Nummer des Signals fest, für das ein Signalhandler einzurichten ist. Üblicherweise gibt man hierfür den symbolischen Signalnamen aus `<signal.h>` (siehe Kapitel 13.2) an.

Das zweite Argument `sighandler` gibt die Adresse der Funktion an, die aufzurufen ist, wenn das Signal `signr` auftritt. Es bestehen hierbei grundsätzlich drei verschiedene Möglichkeiten der Angabe:

1. Signal ignorieren (Angabe: `SIG_IGN`)

Dies ist für alle Signale außer `SIGKILL` und `SIGSTOP` möglich. Diese beiden Signale `SIGKILL` und `SIGSTOP` können nicht ignoriert werden, damit der Superuser immer die Möglichkeit hat, einen Prozeß zu beenden (`SIGKILL`) oder anzuhalten (`SIGSTOP`).

Auch ist darauf hinzuweisen, daß das Ignorieren von bestimmten Hardwaresignalen, wie z.B. Division durch 0 oder illegitimer Speicherzugriff zu einem undefinierten Verhalten des jeweiligen Prozesses führen kann, der solche »ernstzunehmende« Signale ignoriert.

2. Default-Aktion einrichten (Angabe: `SIG_DFL`)

Zu jedem Signal gibt es eine voreingestellte Aktion (*Default-Aktion*), mit der Prozesse auf das Eintreffen dieses Signals reagieren (siehe auch Tabelle 13.1). In den meisten Fällen ist die Default-Aktion die Beendigung des betreffenden Prozesses.

3. Signal abfangen (Angabe: *Adresse einer Funktion*)

Hierbei gibt man die Adresse einer eigenen Funktion an, die aufzurufen ist, wenn ein bestimmtes Signal auftritt. In dieser eigenen Funktion kann man die entsprechenden Reaktionen auf das Signal festlegen.

So schreibt man sich z.B. üblicherweise eine Funktion ***cleanup***, die aufgerufen wird, wenn ein Abbruchsignal geschickt wird. In dieser Funktion ***cleanup*** löscht man dann z.B. alle noch vorhandenen temporären Dateien und schließt alle noch offenen Dateien, bevor man das Programm verläßt.

Ein anderes Beispiel ist das Abfangen des Signals `SIGCHLD`, das geschickt wird, wenn ein Kindprozeß sich beendet. Für diesen Fall ist es sinnvoll, in der entsprechenden »Abfangfunktion« die Funktion ***waitpid*** aufzurufen, um die PID des Kindprozesses und seinen Beendigungsstatus zu erfahren.

Die zwei Signale `SIGKILL` und `SIGSTOP` können nicht abgefangen werden. Der Betriebssystemkern führt für diese beiden Signale immer die Standardaktionen aus, was das Beenden bzw. das Anhalten des jeweiligen Prozesses ist.

Übliche Definitionen für die Konstanten `SIG_IGN` und `SIG_DFL` in `<signal.h>` sind:

```
#define SIG_DFL (void (*)(void)) 0
#define SIG_IGN (void (*)(void)) 1
```

Der Rückgabewert der Funktion ***signal*** ist entweder die Adresse des bisher eingerichteten Signalhandlers oder `SIG_ERR`, wobei `SIG_ERR` anzeigt, daß die Einrichtung des Signalhandlers nicht erfolgreich war. `SIG_ERR` ist z.B. wie folgt in `<signal.h>` definiert:

```
#define SIG_ERR (void (*)(void)) -1
```

Deklaration der signal-Funktion

Unter Verwendung von ***typedef*** läßt sich die komplexe Deklaration der Funktion ***signal*** etwas vereinfachen. Dazu geben wir in unserer Headerdatei `sighdr.h` folgende Zeile an:

```
typedef void sigfunk(int);
```

Mit dieser Typdefinition läßt sich dann der komplexe Prototyp der ***signal***-Funktion

```
void (*signal(int signr, void (*sighandler)(int)))(int);
```

vereinfachen zu:

```
sigfunk *signal(int signr, sigfunk *sighandler);
```

Beispiel*Abfangen der Signale SIGFPE und SIGINT*

Das folgende Programm 13.1 (intcatch.c) demonstriert das Abfangen des Signals SIGFPE, das hier bei einer Division durch 0 gesendet wird. Zusätzlich fängt es viermal das Signal SIGINT ab, welches beim Drücken der Programmabbruchtaste (meist *Strg-C*) geschickt wird. Nach dem vierten Drücken der Programmabbruchtaste beendet es sich mit dem Aufruf der **exit**-Funktion.

```
#include <signal.h>
#include "eighdr.h"

static void ctrlc_faenger(int sig);
static void null_division(int sig);

/*----- main -----*/
int
main(void)
{
    long int i, j;
    double wert;

    if (signal(SIGINT, ctrlc_faenger) == SIG_ERR)
        fehler_meld(FATAL_SYS, "Signalhandler 'Ctrlc_faenger' konnte "
                    "nicht installiert werden");
    printf(".....Signalhandler ctrlc_faenger installiert....\n");

    if (signal(SIGFPE, null_division) == SIG_ERR)
        fehler_meld(FATAL_SYS, "Signalhandler 'null_division' konnte "
                    "nicht installiert werden");
    printf(".....Signalhandler null_division installiert....\n");

    /* Erzeugen einer Division durch 0 */
    wert = wert / 0;

    /* Warte-Schleife */
    while (1)
        ;

    printf("---- Programmende ---\n");
    exit(0);
}

/*----- Signalhandler-Routinen -----*/
void ctrlc_faenger( int sig )
{
    static int i=1;
    /* Fuer die Dauer dieser Funktionsausfuehrung muessen weitere */
    /* SIGINT-Signale ignoriert werden.                               */
    signal(SIGINT, SIG_IGN);

    printf(" %d. Ctrl-c gedrueckt", i);
```

```

    if (i<=3) {
        if (signal(SIGINT, ctrlc_faenger) == SIG_ERR)
            fehler_meld(FATAL_SYS, "Signalhandler 'Ctrlc_faenger' konnte "
                        "nicht installiert werden");
        printf("\n");
    } else {
        printf(" (Programmende)\n");
        exit(0);
    }
    i++;
}

void null_division( int sig )
{
    /* Fuer die Dauer dieser Funktionsausfuehrung muessen weitere */
    /* SIGFPE-Signale ignoriert werden.                               */
    signal(SIGFPE, SIG_IGN);

    /* Text "Division durch 0 aufgetreten" ausgeben */
    printf("Division durch 0 aufgetreten\n");
}

```

Programm 13.1 (intcatch.c): Abfangen des Signals SIGFPE und vielmaliges Abfangen des Signals SIGINT

Nachdem man dieses Programm 13.1 (intcatch.c) kompiliert und gelinkt hat

```
cc -o intcatch intcatch.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ intcatch
....Signalhandler ctrlc_faenger installiert....
....Signalhandler null_division installiert....
Division durch 0 aufgetreten
Ctrl-c      [1. Versuch, das Programm mit Ctrl-C abubrechen]
1. Ctrl-c gedrueckt
Ctrl-c      [2. Versuch, das Programm mit Ctrl-C abubrechen]
2. Ctrl-c gedrueckt
Ctrl-c      [3. Versuch, das Programm mit Ctrl-C abubrechen]
3. Ctrl-c gedrueckt
Ctrl-c      [4. erfolgreicher Versuch, das Programm mit Ctrl-C abubrechen]
4. Ctrl-c gedrueckt (Programmende)
$

```

Beispiel

Abfangen der Signale SIGUSR1, SIGUSR2 und SIGTERM

Das folgende Programm 13.2 (sigusr.c) fängt die beiden benutzerdefinierten Signale SIGUSR1, SIGUSR2 sowie das Signal SIGTERM ab und gibt deren Signalnummer aus. Zusätzlich versucht das Programm, das Signal SIGKILL abzufangen, das niemals abgefangen werden kann. Dieses Programm 13.2 (sigusr.c) ruft die Funktion **pause** auf, die den Prozeß solange anhält, bis er ein Signal empfängt. Die Funktion **pause** wird in Kapitel 13.6 beschrieben.

```

#include <signal.h>
#include "eighdr.h"

static void sig_handler(int signr);

/*----- main -----*/
int
main(void)
{
    long int i, j;
    double wert;

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        fehler_meld(FATAL_SYS,
                    "kann Signalhandler fuer SIGUSR1 nicht installieren");
    if (signal(SIGUSR2, sig_handler) == SIG_ERR)
        fehler_meld(FATAL_SYS,
                    "kann Signalhandler fuer SIGUSR2 nicht installieren");
    if (signal(SIGTERM, sig_handler) == SIG_ERR)
        fehler_meld(FATAL_SYS,
                    "kann Signalhandler fuer SIGTERM nicht installieren");
    if (signal(SIGKILL, sig_handler) == SIG_ERR)
        fehler_meld(WARNUNG_SYS,
                    "kann Signalhandler fuer SIGKILL nicht installieren");

    while (1)
        pause();
    printf("---- Programmende ---\n");
    exit(0);
}

/*----- Signalhandler-Routinen -----*/
void sig_handler( int signr )
{
    if (signr == SIGUSR1)
        printf(".....Signal SIGUSR1 (%d) wurde geschickt.....\n", SIGUSR1);
    else if (signr == SIGUSR2)
        printf(".....Signal SIGUSR2 (%d) wurde geschickt.....\n", SIGUSR2);
    else if (signr == SIGTERM)
        printf(".....Signal SIGTERM (%d) wurde geschickt.....\n", SIGTERM);
    else
        fehler_meld(FATAL_SYS, "....Signal %d wurde geschickt....", signr);
}

```

Programm 13.2 (sigusr.c): Abfangen der Signale SIGUSR1, SIGUSR2 und SIGTERM

Nachdem wir dieses Programm 13.2 (sigusr.c) kompiliert und gelinkt haben

```
cc -o sigusr sigusr.c fehler.c
```

rufen wir das Programm sigusr im Hintergrund auf und schicken ihm mit dem **kill**-Kommando nacheinander die Signale SIGUSR1, SIGUSR2 und SIGTERM, bevor wir ihm das Signal SIGKILL schicken, das ihn schließlich beendet, da das Signal SIGKILL niemals von einem Prozeß abgefangen werden.


```

$ sigusr &
[1] 188
kann Signalhandler fuer SIGKILL nicht installieren: Invalid argument
$ kill -USR1 188
.....Signal SIGUSR1 (10) wurde geschickt.....
$ kill -USR2 188
.....Signal SIGUSR2 (12) wurde geschickt.....
$ kill -TERM 188
.....Signal SIGTERM (15) wurde geschickt.....
$ kill -KILL 188
$           [Eingabe von Return]
[1] Killed      sigusr      [Ausgabe, dass Job durch SIGKILL beendet wurde]
$

```

Hinweis

In SVR4 wird bei der Funktion ***signal*** immer noch das alte unzuverlässige Signalkonzept von SVR2 verwendet, um Kompatibilität zu Anwendungen zu wahren, die für das alte Signalkonzept ausgelegt wurden. Dieses alte Signalkonzept wird in Kapitel 13.3 beschrieben. Neu erstellte Programme sollten niemals diese unzuverlässigen Signale benutzen.

BSD-Unix bietet zwar auch die Funktion ***signal*** an, aber dort entspricht sie der neuen zuverlässigen Funktion ***sigaction*** (siehe Kapitel 13.4).

13.1.3 Signale und Kindprozesse

Wenn ein Prozeß mit ***fork*** einen Kindprozeß erzeugt, so erbt der Kindprozeß alle eingerichteten Signalhandler des Elternprozesses, da bei der Kreierung des Kindprozesses immer automatisch die Adressen der Signalhandler-Routinen mitkopiert werden.

13.1.4 Signale und die *exec*-Funktion

Wenn ein Prozeß ein neues Programm mit der ***exec***-Funktion startet, so wird außer bei den zu ignorierenden Signalen bei allen anderen Signalen die Default-Aktion eingerichtet. Das bedeutet, daß für alle Signale, für die eine Funktion als Signalhandler eingerichtet ist, im neuen Programm in jedem Fall wieder die Default-Aktion eingerichtet wird. Dies ist auch einsichtig, da die Adressen der Signalhandlerfunktionen für das neue Programm keine Gültigkeit mehr haben.

Wenn ein Programm im Hintergrund (mit Angabe von ***&***) gestartet wird, so werden die Programmabbruchsignale *SIGINT* (meist *Strg-C*) und *SIGQUIT* (*Strg-*) von einer Shell (ohne Jobkontrolle) ignoriert. Wenn dies nicht getan würde, würde beim Drücken einer dieser beiden Programmabbruchtasten nicht nur der Vordergrundprozeß, sondern auch alle momentan laufenden Hintergrundprozesse beendet.

Dies ist auch der Grund, warum es unter Unix üblich ist, daß interaktive Programme z.B. folgenden Codeausschnitt (oder einen ähnlichen) enthalten:

```

int sigint_handler(int signr),
    sigquit_handler(int signr);

```

```

.....

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sigint_handler);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sigquit_handler);

```

Dadurch ist sichergestellt, daß der Prozeß nur dann die Signale SIGINT und SIGQUIT abfängt, wenn diese momentan nicht ignoriert werden.

13.1.5 Begriffe rund um das Signalkonzept

Hier werden die Begriffe geklärt, die im Zusammenhang mit Signalen benutzt werden.

Generieren (Erzeugen) eines Signals für einen Prozeß ***Schicken eines Signals an einen Prozeß***

Diese Ausdrucksform benutzt man, wenn das Ereignis eintritt, das das entsprechende Signal auslöst. Das Ereignis kann dabei ein Hardwarefehler (Division durch 0), das Eintreten einer mit der Software gesetzten Bedingung (z.B. Ablauf einer eingerichteten Zeitschaltuhr), das Eintreten eines Ereignisses am Terminal (z.B. Drücken der Programmabbruchtaste) oder das Aufrufen der Funktion **kill** (siehe Kapitel 13.5) sein. Wenn ein Signal erzeugt wird, so setzt der Kern üblicherweise ein bestimmtes Flag in der Prozeßtabelle.

Zustellen eines Signals an einen Prozeß

Diese Ausdrucksform besagt, daß die für ein Signal eingerichtete Aktion gestartet wird.

Hängen eines Signals

Für die Zeitspanne zwischen der Erzeugung und der Zustellung eines Signal verwendet man diese Ausdrucksform.

Blockieren eines Signals

Ein Prozeß hat immer die Möglichkeit, die Zustellung eines Signals zu blockieren. Wenn ein Signal generiert wird, das für einen Prozeß blockiert ist, und der betreffende Prozeß hat für dieses Signal entweder die Default-Aktion oder einen eigenen Signalhandler eingerichtet, so bleibt das Signal solange hängen bis der Prozeß

- ▶ entweder die Blockierung für dieses Signal aufhebt
- ▶ oder aber explizit angibt, daß dieses Signal zu ignorieren ist.

Wie mit einem blockiertem Signal zu verfahren ist, wird nämlich immer erst bei der Zustellung und nicht bei der Generierung eines Signals festgelegt. So ist es einem Prozeß immer möglich, die für ein Signal eingerichtete Aktion zu ändern, bevor es zugestellt wird. Zum Blockieren von Signalen oder zum Erfragen von hängenden Signalen steht die Funktion **sigpending** (siehe Kapitel 13.4) zur Verfügung.

Signalmaske

Jeder Prozeß hat eine Signalmaske, die die Menge aller Signale (siehe Kapitel 13.4) enthält, die momentan blockiert sind. Bei einer Signalmaske ist jedem Signal ein Bit zugeordnet. Ist dieses Bit gesetzt, so ist das zugehörige Signal momentan blockiert. Mit der Funktion **sigprocmask** (siehe Kapitel 13.4) kann die momentane Signalmaske erfragt oder geändert werden. Für Signalmasken, die eine Menge von Signalen definieren, hat POSIX.1 einen eigenen Datentyp `sigset_t` eingeführt.

Warteschlange für blockierte Signale der gleichen Art

Wenn ein blockiertes Signal mehr als einmal generiert wird, bevor der Prozeß die Blockierung aufhebt, dann läßt POSIX.1 dem jeweiligen System folgende beide Möglichkeiten offen:

- ▶ Das Signal wird nur einmal zugestellt, was für die meisten Unix-Implementierungen zutrifft.
- ▶ Die Signale werden in eine Warteschlange eingereiht.

Reihenfolge der Zustellung von Signalen

Wenn mehrere Signale für die Zustellung an einen Prozeß anstehen, so schreibt POSIX.1 keine feste Reihenfolge für die Zustellung vor. POSIX.1 schlägt lediglich vor, daß Signale, die sich auf den momentanen Prozeßzustand beziehen (wie `SIGSEGV`) vor anderen Signalen zugestellt werden sollten.

13.2 Signalnamen und Signalnummern

13.2.1 Signalnamen

Zu jedem Signal gibt es einen symbolischen Namen, der immer mit `SIG` beginnt und für eine Nummer steht, wie z.B. der Name `SIGINT` für das Signal, das generiert wird, wenn der Benutzer die Programmabbruchtaste (*Strg-C*) drückt. Alle symbolischen Namen sind in `<signal.h>` (bzw. `<sys/signal.h>` oder `<linux/signal.h>`) definiert. Kein Signal hat die Nummer 0, da diese Nummer für spezielle Anwendungsfälle der Funktion **kill** (siehe Kapitel 13.5) vorgesehen ist.

Während in älteren Unix-Systemen 15 verschiedene Signale angeboten wurden, stellen SVR4 und 4.4BSD inzwischen mehr als 30 Signale zur Verfügung. Die Tabelle 13.1 zeigt die meisten Signale von SVR4 und BSD-Unix im Überblick.

Name	Beschreibung	ANSIC	POSIX.1	SVR4	BSD	Default Aktion
SIGABRT	anormale Beendigung (abort)	x	x	x	x	Beendigung mit core
SIGALRM	Ablauf einer »Zeitschaltuhr«		x	x	x	Beendigung
SIGBUS	Hardwarefehler			x	x	Beendigung mit core
SIGCHLD	Statusänderung in Kindprozeß		j	x	x	Ignorieren
SIGCONT	Fortsetzen von angehalt. Prozeß		j	x	x	Fortsetzen / Ignorier.
SIGEMT	Hardwarefehler			x	x	Beendigung mit core
SIGFPE	Arithmetischer Fehler	x	x	x	x	Beendigung mit core
SIGHUP	Verbindungsunterbrechung		x	x	x	Beendigung
SIGILL	Unerlaubter Hardwarebefehl	x	x	x	x	Beendigung mit core
SIGINFO	Statusanforderung von Tastatur				x	Ignorieren
SIGINT	Unterbrechungstaste am Terminal	x	x	x	x	Beendigung
SIGIO	Asynchrone E / A			x	x	Beendigung / Ignorier.
SIGIOT	Hardwarefehler			x	x	Beendigung mit core
SIGKILL	Beendigung		x	x	x	Beendigung
SIGPIPE	Schreiben in Pipe ohne Leser		x	x	x	Beendigung
SIGPOLL	wählbares Ereignis (poll)			x		Beendigung
SIGPROF	Profiling-Zeitalarm (setitimer)			x	x	Beendigung
SIGPWR	Stromausfall			x		Ignorieren
SIGQUIT	Unterbrechungstaste am Terminal		x	x	x	Beendigung mit core
SIGSEGV	Unerlaubte Speicher adressierung	x	x	x	x	Beendigung mit core
SIGSTOP	Prozeß anhalten		j	x	x	Prozeß anhalten
SIGSYS	Unerlaubter Systemaufruf			x	x	Beendigung mit core

Tabelle 13.1: Überblick über die Signalnamen

Name	Beschreibung	ANSIC	POSIX.1	SVR4	BSD	Default Aktion
SIGTERM	Beendigung	x	x	x	x	Beendigung
SIGTRAP	Hardwarefehler			x	x	Beendigung mit core
SIGTSTP	Terminal-Stoppzeichen		j	x	x	Prozeß anhalten
SIGTTIN	Lesewunsch von Hintergr.-Prozeß		j	x	x	Prozeß anhalten
SIGTTOU	Schreibwunsch von Hintergr.-Prozeß		j	x	x	Prozeß anhalten
SIGURG	dringendes Ereignis			x	x	Ignorieren
SIGUSR1	benutzerdefiniertes Signal		x	x	x	Beendigung
SIGUSR2	benutzerdefiniertes Signal		x	x	x	Beendigung
SIGV- TALRM	Virtueller Zeitalarm (<i>setitimer</i>)			x	x	Beendigung
SIG- WINCH	Änderung der Window-Größe			x	x	Ignorieren
SIGXCPU	Überschreitung des CPU-Limits (<i>setrlimit</i>)			x	x	Beendigung mit core
SIGXFSZ	Überschreitung des Dateigrößelimits (<i>setrlimit</i>)			x	x	Beendigung mit core

Tabelle 13.1: Überblick über die Signalnamen

In eigenen Spalten zeigt die Tabelle 13.1, welche Signale jeweils von ANSI-C und POSIX.1 vorgeschrieben sind. Bei der POSIX.1-Spalte zeigt ein *x* an, daß dieses Signal in jedem Fall vorgeschrieben ist, während ein *j* anzeigt, daß es sich bei diesem Signal um ein »Jobkontrollsignal« handelt, welches nur dann existieren muß, wenn Jobkontrolle vorhanden ist.

Die letzte Spalte *Default-Aktion* beschreibt kurz die voreingestellte Reaktion des Prozesses, an den dieses Signal geschickt wird. So bedeutet z.B. »Beendigung mit core«, daß vom aktuellen Zustand des Prozesses ein Speicherabbild (*core image*) in der Datei *core* im Working-Directory des Prozesses hinterlegt wird.

Diese Datei *core* kann den meisten Unix-Debuggern vorgelegt werden, um nachträglich den Zustand des Prozesses zum Zeitpunkt seiner Beendigung zu untersuchen. In den folgenden Fällen wird kein Speicherabbild in der Datei *core* hinterlegt:

- Wenn der Prozeß mit Set-User-ID-Bit lief und der Aufrufer nicht der Besitzer der Programmdatei ist.

- ▶ Wenn der Prozeß mit Set-Group-ID-Bit lief und der Aufrufer nicht der Gruppeneigentümer der Programmdatei ist.
- ▶ Wenn der Benutzer keine Schreibrechte im aktuellen Working-Directory hat.
- ▶ Wenn die Datei `core` zu groß ist (siehe auch `RLIMIT_CORE` in Kapitel 9.5).

Die Zugriffsrechte für die Datei `core` sind üblicherweise 644 (`rw-r--r--`), wenn sie nicht schon existiert.

Hinweis

Das Anlegen der Datei `core` ist zwar typisch für Unix, aber nicht Bestandteil von POSIX.1.

BSD-Unix legt eine Datei `core.prog`, wobei `prog` die ersten 16 Zeichen des entsprechenden Programmnamens sind. So können dort mehrere `core`-Dateien für unterschiedliche Programme im gleichen Directory liegen.

Beschreibung der einzelnen Signale

Nachfolgend sind die Signale aus der Tabelle 13.1 ausführlicher beschrieben.

SIGABRT

Dieses Signal wird beim Aufruf der ***abort***-Funktion (siehe Kapitel 13.7) erzeugt. Es signalisiert, das ein Prozeß anormal beendet wurde.

Unter Linux z.B. wird ***abort*** immer dann aufgerufen, wenn die beim Aufruf der ***assert***-Funktion angegebene Bedingung nicht erfüllt ist.

SIGALRM

Dieses Signal zeigt an, daß eine zuvor mit der ***alarm***-Funktion eingerichtete Zeitschaltuhr abgelaufen ist (siehe auch Kapitel 13.6). Es wird auch generiert, wenn eine mit ***setitimer*** eingerichtete Intervall-Zeitschaltuhr abgelaufen ist.

SIGBUS

Dieses Signal wird bei einem Hardwarefehler (implementierungsdefiniert) geschickt.

SIGCHLD

Dieses Signal wird immer dann an den Elternprozeß geschickt, wenn sich einer seiner Kindprozesse beendet. Normalerweise wird dieses Signal ignoriert, wenn der Elternprozeß es nicht abfängt. Üblicherweise fängt man dieses Signal mit der ***wait***-Funktion ab, um die ID des beendeten Kindprozesses und den Beendigungsstatus dieses Kindprozesses zu erfahren. Dieses Signal löste das alte Signal `SIGCLD` von früheren Unix-Versionen ab.

SIGCONT

Dieses Signal wird an einen angehaltenen Prozeß geschickt, wenn er seine Ausführung fortsetzen soll. Wird dieses Signal an einen nicht angehaltenen Prozeß geschickt, so wird es von diesem ignoriert.

Viele Editoren fangen dieses Signal ab und frischen das Terminal-Fenster auf, wenn sie wieder gestartet, also in den Vordergrund gebracht werden.

SIGEMT

Dieses Signal wird bei einem Hardwarefehler (implementierungsdefiniert) geschickt. EMT stammt von dem Befehl *emulator trap* der PDP-11.

SIGFPE

Dieses Signal wird bei einem arithmetischen Fehler, wie z.B. Division durch 0 oder *Overflow*, geschickt (FPE steht für *floating point error*).

SIGHUP

Dieses Signal wird dem Kontrollprozeß (Sessionführer) eines Terminals geschickt, wenn eine Verbindung zum Terminal unterbrochen wird. Der Kontrollprozeß ist dabei der Prozeß, auf den die Komponente `s_leader` der `session`-Struktur zeigt. Wenn das Flag `CLOCAL` (siehe Kapitel 20) für ein Terminal (lokales Terminal) gesetzt ist, so wird dieses Signal nicht generiert und Statusänderungen von Modemanschlüssen werden ignoriert.

Das Signal `SIGHUP` wird auch geschickt, wenn der Kontrollprozeß (*session leader*) beendet wird. In diesem Fall wird das Signal an jeden Prozeß geschickt, der momentan im Vordergrund arbeitet.

Üblicherweise wird dieses Signal benutzt, um Dämonprozesse (siehe Kapitel 16) zu veranlassen, ihre Logdateien zu schließen und neu zu öffnen sowie ihre Konfigurationsdateien erneut zu lesen. `SIGHUP` ist hierfür besonders geeignet, da ein Dämonprozeß üblicherweise kein Kontrollterminal besitzt und deshalb dieses Signal normalerweise nicht empfangen würde.

SIGILL

Dieses Signal zeigt an, daß der Prozeß einen illegalen Hardwarebefehl ausgeführt hat.

SIGINFO

Dieses Signal wird in BSD-Unix generiert, wenn die Statusanforderungstaste (üblicherweise *Strg-T*) gedrückt wird. Dieses Signal wird dabei allen Prozessen geschickt, die momentan im Vordergrund arbeiten, und bewirkt, daß Statusinformation über alle diese Prozesse am Terminal ausgegeben werden.

SIGINT

Dieses Signal wird allen Prozessen geschickt, die momentan im Vordergrund arbeiten, wenn die Unterbrechungstaste (üblicherweise *DELETE* oder *Strg-C*) gedrückt wird.

SIGIO

Dieses Signal zeigt asynchrone E/A-Anforderungen an (siehe auch Kapitel 15.2). In SVR4 ist dieses Signal identisch zum Signal `SIGPOLL` und die Default-Aktion ist dort die Beendigung des Prozesses. In BSD-Unix ist die Default-Aktion das Ignorieren dieses Signals.

SIGIOT

Dieses Signal zeigt einen implementierungsspezifischen Hardwarefehler an. IOT steht dabei für *Input/Output-Trap*-Befehl (PDP-11).

SIGKILL

Dieses Signal beendet den Prozeß, an den es geschickt wird, in jedem Fall, da es niemals abgefangen oder ignoriert werden kann.

SIGPIPE

Dieses Signal wird einem in eine Pipe schreibenden Prozeß geschickt, wenn der aus der Pipe lesende Prozeß sich vorzeitig beendet. Diese Situation wird mit »*broken pipe*« bezeichnet.

SIGPOLL

Dieses Signal zeigt an, daß ein spezielles Ereignis an einem wählbaren Gerät aufgetreten ist. Dieses Signal wird bei der **poll**-Funktion in Kapitel 15.1 genauer beschrieben. Unter BSD-Unix sind die Signale SIGIO und SIGURG mit diesem Signal vergleichbar.

SIGPROF

Dieses Signal wird geschickt, wenn eine *Profiling*-Zeitschaltuhr, die mit der Funktion **setitimer** (siehe auch Manpage `setitimer(2)`) eingestellt wurde, abgelaufen ist.

Profiler werden normalerweise eingesetzt, um die Ausführungszeit einzelner Programmteile zu ermitteln. Unter Unix wird dazu der Profiler **prof** angeboten und unter Linux dessen GNU-Variante **gprof**.

SIGPWR

Dieses Signal wird unter SVR4 nur in Systemen angeboten, die über eine nicht unterbrechbare Stromversorgung verfügen. In solchen Systemen wird dieses Signal üblicherweise geschickt, wenn nach einem Stromausfall auf Batterie umgeschaltet wurde und die Batterie beginnt, an Ladung zu verlieren. Die meisten Systeme sind so konfiguriert, daß dieses Signal dem `init`-Prozeß geschickt wird, welcher daraufhin ein **shutdown** des Systems veranlaßt. Viele SVR4-Implementierungen von `init` stellen dazu in der Datei `inittab` zwei Einträge `powerfail` und `powerwait` zur Verfügung.

SIGQUIT

Dieses Signal wird allen Prozessen geschickt, die momentan im Vordergrund arbeiten, wenn die Unterbrechungstaste QUIT (meist *Strg-^*) gedrückt wird. SIGQUIT verhält sich wie Signal SIGINT, legt jedoch eine `core`-Datei an.

SIGSEGV

Dieses Signal zeigt an, daß der Prozeß versuchte, auf eine unerlaubte Adresse im Speicher zuzugreifen (Lesen oder Schreiben). SEGV ist dabei die Abkürzung für *segmentation violation*.

SIGSTOP

Dieses Signal hält einen Prozeß an. Das Signal SIGSTOP ist zwar dem interaktiven Terminalstoppsignal SIGTSTP ähnlich, kann aber nicht wie dieses abgefangen oder ignoriert werden.

SIGSYS

Dieses Signal zeigt an, daß ein unerlaubter Systemaufruf stattfand. Ein unerlaubter Systemaufruf liegt dann vor, wenn ein Prozeß einen Maschinenbefehl ausführt, den der Kern fälschlicherweise als Systemaufruf interpretiert, und diesen Fehler dann erst bei den falschen oder fehlenden Argumenten erkennt.

SIGTERM

Dieses Signal ist das voreingestellte Signal, das das **kill**-Kommando einem Prozeß schickt, dem es mitteilen möchte, daß er sich beenden soll.

SIGTRAP

Dieses Signal zeigt einen implementierungsdefinierten Hardware-Fehler an.

Wenn die Ausführung eines Prozesses auf einen Breakpoint trifft, wird dieses Signal an den Prozeß geschickt. Es wird gewöhnlich von einem Debugger abgefangen, der den Breakpoint gesetzt hat.

SIGTSTP

Dieses Signal wird allen Prozessen geschickt, die momentan im Vordergrund arbeiten, wenn die Terminalstopptaste (meist *Strg-Z*) gedrückt wird.

SIGTTIN

Dieses Signal wird generiert, wenn ein Hintergrundprozeß versucht, von seinem Kontrollterminal zu lesen. *SIGTTIN* wird nicht generiert, wenn der lesende Prozeß dieses Signal ignoriert oder blockiert oder aber die Prozeßgruppe des lesenden Prozesses verwaist ist. In diesen Spezialfällen führt die Leseoperation zu einem Fehler, wobei die Variable `errno` auf `EIO` gesetzt wird.

SIGTTOU

Dieses Signal wird generiert, wenn ein Hintergrundprozeß versucht, auf das Kontrollterminal zu schreiben. Dieses Signal *SIGTTOU* wird nicht generiert, wenn der schreibende Prozeß dieses Signal ignoriert oder blockiert oder aber die Prozeßgruppe des schreibenden Prozesses verwaist ist. In diesen beiden Spezialfällen führt die Schreiboperation zu einem Fehler, wobei die Variable `errno` auf `EIO` gesetzt wird.

Anders als beim Signal *SIGTTIN* kann ein Hintergrundprozeß das Schreiben jedoch zulassen oder auch verbieten. Ist Schreiben durch einen Hintergrundprozeß erlaubt, so gelten die beiden zuvor erwähnten Spezialfälle nicht.

Neben Schreiboperationen kann dieses Signal *SIGTTOU* auch von den Terminalroutinen *tcsetattr*, *tcsendbreak*, *tcdrain*, *tcflush*, *tcflow* und *tcsetpgrp* (siehe auch Kapitel 20) generiert werden.

SIGURG

Dieses Signal zeigt an, daß ein dringendes Ereignis eingetreten ist, auf das sofort reagiert werden muß. Solche dringenden Ereignisse treten z.B. bei Netzwerkverbindungen auf.

SIGUSR1

Dieses benutzerdefinierte Signal ist für die Verwendung in Anwenderprogrammen reserviert.

SIGUSR2

Dieses zweite benutzerdefinierte Signal ist ebenfalls für die Verwendung in Anwenderprogrammen reserviert.

SIGVTALRM

Dieses Signal zeigt an, daß eine zuvor mit der Funktion **setitimer** eingerichtete virtuelle Zeitschaltuhr abgelaufen ist.

SIGWINCH

Dieses Signal wird allen Vordergrundprozessen geschickt, die einem Terminal oder Pseudoterminal zugeordnet sind, wenn die Fenstergröße dieses Terminals bzw. Pseudoterminals mit der **ioctl**-Funktion (siehe auch Kapitel 20) geändert wird.

SIGXCPU

Dieses Signal wird Prozessen geschickt, die das für sie festgelegte CPU-Zeitlimit überschreiten (siehe auch Kapitel 9.5).

SIGXFSZ

Dieses Signal wird Prozessen geschickt, die das für sie festgelegte Dateigrößenlimit überschreiten (siehe auch Kapitel 9.5).

13.2.2 sys_siglist und psignal – Signalbeschreibungen

Einige Systeme (wie BSD und SVR4) stellen das folgende Array zur Verfügung.

```
extern char *sys_siglist[];
```

Dieses Array enthält Kurzbeschreibungen zu allen Signalen. Als Arrayindex ist dabei die Signalnummer anzugeben.

Daneben stellen diese Systeme normalerweise die Funktion **psignal** zur Verfügung.

```
#include <signal.h>

void psignal(int signr, const char *string);
```

Diese Funktion **psignal** ist ähnlich zur Funktion **perror**. Sie gibt den angegebenen *string* (normalerweise der Programmname) auf die Standardfehlerausgabe aus. Danach gibt sie einen Doppelpunkt mit Leerzeichen aus, bevor sie eine kurze Beschreibung des Signals, gefolgt von einem Neue-Zeile-Zeichen, ausgibt.

Beispiel*Kurzbeschreibungen zu den ersten 10 Signalen*

Das folgende Programm 13.3 (psignal.c) gibt Kurzbeschreibungen zu den ersten 10 Signalen einmal mit **psignal** und einmal mit `sys_siglist` aus.

```
#include <signal.h>
#include "eighdr.h"

int
main(void)
{
    int i;
    char text[10];

    fprintf(stderr, "----- Ausgabe mit psignal -----\\n");
    for (i=1; i<=10; i++) {
        sprintf(text, "%2d", i);
        psignal(i, text);
    }

    fprintf(stderr, "\\n");
    fprintf(stderr, "----- Ausgabe mittels sys_siglist ---\\n");
    for (i=1; i<=10; i++) {
        sprintf(text, "%2d", i);
        printf(stderr, "%s: %s\\n", text, sys_siglist[i]);
    }

    exit(0);
}
```

Programm 13.3 (**psignal.c**): Beschreibungen zu ersten 10 Signalen mit **psignal** und `sys_siglist`

Nachdem man dieses Programm 13.3 (psignal.c) kompiliert und gelinkt hat

```
cc -o psignal psignal.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ psignal
----- Ausgabe mit psignal -----
1: Hangup
2: Interrupt
3: Quit
4: Illegal instruction
5: Trace/breakpoint trap
6: IOT trap/Abort
7: Unused signal
8: Floating point exception
9: Killed
10: User defined signal 1

----- Ausgabe mittels sys_siglist ---
```

```

1: Hangup
2: Interrupt
3: Quit
4: Illegal instruction
5: Trace/breakpoint trap
6: IOT trap/Abort
7: Unused signal
8: Floating point exception
9: Killed
10: User defined signal 1
$

```

13.3 Probleme mit der signal-Funktion

Das Signalkonzept mit der Funktion **signal** wurde zwar schon in den ersten Unix-Versionen angeboten, war aber dort noch äußerst unzuverlässig. Diese Unzuverlässigkeit und Mangelhaftigkeit mit der **signal**-Funktion konnte zum Teil leider bis heute nicht beseitigt werden, was an der damaligen Konzipierung der Funktion **signal** liegt. Deswegen wurde eine neue Funktion **sigaction** (siehe Kapitel 13.4) eingeführt, die die nachfolgend aufgelisteten Probleme, die bei der **signal**-Funktion auftreten können, nicht kennt.

13.3.1 Erfragen des aktuellen Signalstatus ohne Änderung nicht möglich

Mit der Funktion **signal** ist es nicht möglich, lediglich den momentan eingerichteten Signalhandler zu erfragen, ohne einen neuen (eventuell auch den gleichen) Signalhandler einzurichten.

Um z.B. für das Signal SIGINT nur dann einen neuen Signalhandler einzurichten, wenn es momentan nicht ignoriert wird, muß man die Funktion **signal** aufrufen, um über den Rückgabewert zu erfahren, welcher Signalhandler zur Zeit für das Signal SIGINT eingerichtet ist. Bei einem Aufruf von **signal** wird aber immer ein neuer Signalhandler eingerichtet. Deswegen sieht z.B. der Code für diese Aufgabenstellung wie folgt aus:

```

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sighandler);

```

13.3.2 Zeitspanne zwischen Auftreten eines Signals und Aufruf der signal-Funktion

In früheren Unix-Versionen wurde nach dem Abfragen eines Signals durch einen Signalhandler automatisch wieder die Default-Aktion für dieses Signal vom Kern eingerichtet. Eine typische Vorgehensweise wie z.B. früher das SIGINT-Signal abgefangen wurde, zeigt der folgende Code-Auszug:

```

int signal_handler();
.....
signal(SIGINT, signal_handler); /* Einrichten eines */

```



```

    sig_int_flag = 1; /* Flag setzen, um anzuzeigen,
                       daß Signal eingetreten ist    */
}

```

Hier ruft der Prozeß die Funktion **pause** auf, um auf die Ankunft des Signals `SIGINT` zu warten. Wenn das Signal auftritt, so wird die Variable `sig_int_flag` auf 1 gesetzt und die **while**-Schleife verlassen.

Diese Technik ist jedoch nicht ganz frei von Problemen. Tritt nämlich das Signal `SIGINT` genau in der Zeitspanne nach der Überprüfung von `sig_int_flag` (in der **while**-Schleife) und vor dem Aufruf von **pause** auf, so bleibt der Prozeß in der **while**-Schleife hängen, es sei denn, daß irgendwann später nochmals das Signal `SIGINT` geschickt wird. Solche Fehler sind oft schwer auffindbar, da sie nur selten auftreten und es nicht einfach ist, mit Debuggen wieder die gleiche Situation herzustellen, die zum Fehler führte.

13.4 Das neue Signalkonzept

Die in Kapitel 13.3 genannten Schwächen waren der Grund dafür, daß man neue Konzepte und Funktionen schuf, um diese Schwächen zu beseitigen.

13.4.1 Signalmengen

Um Signalmengen repräsentieren zu können, benötigt man einen eigenen Datentyp. POSIX.1 schreibt hierfür den Datentyp `sigset_t` und die folgenden fünf Funktionen zur Manipulation von Signalmengen vor.

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);
```

```
int sigfillset(sigset_t *set);
```

```
int sigaddset(sigset_t *set, int signr);
```

```
int sigdelset(sigset_t *set, int signr);
```

alle vier geben zurück: 0 (bei Erfolg); -1 bei Fehler

```
int sigismember(const sigset_t *set, int signr);
```

gibt zurück: 1 (wenn TRUE); 0 bei FALSE

sigemptyset und sigfillset – Initialisieren der Signalmenge

sigemptyset entfernt alle Signale aus der Signalmenge, auf die `set` zeigt. **sigfillset** fügt alle vorhandenen Signale zu der Signalmenge hinzu, auf die `set` zeigt. Da jede Signalmengenvariable (wie auch jede andere Variable) initialisiert werden muß, muß eine dieser beiden Funktionen für jede Signalmenge aufgerufen werden.

sigaddset und sigdelset – Hinzufügen und Löschen einzelner Signale in Signalmenge

Nachdem man eine Signalmenge initialisiert hat, kann man mit ***sigaddset*** ein einzelnes Signal zu dieser Signalmenge hinzufügen oder mit ***sigdelset*** ein einzelnes Signal aus dieser Signalmenge entfernen.

sigismember – Prüfen, ob Signal in Signalmenge vorhanden ist

Mit der Funktion ***sigismember*** kann man erfragen, ob das Signal `signr` in der Signalmenge enthalten ist, auf die `set` zeigt.

13.4.2 sigaction – Einrichten und Erfragen von Signalhandlern

Um für ein Signal einen Signalhandler neu einzurichten oder den momentan eingerichteten Signalhandler zu erfragen oder auch beides, steht die Funktion ***sigaction*** zur Verfügung.

```
#include <signal.h>
```

```
int sigaction(int signr, const struct sigaction *neu_handler,
              struct sigaction *alt_handler);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Anders als in früheren Unix-Systemen, bleibt ein mit ***sigaction*** installierter Signalhandler solange installiert, bis er explizit durch einen weiteren ***sigaction***-Aufruf geändert wird.

Das Argument `signr` spezifiziert dabei das Signal, zu dem ein Signalhandler einzurichten oder zu erfragen ist.

Wenn `neu_handler` kein NULL-Zeiger ist, so bedeutet dies, daß für `signr` ein neuer Signalhandler einzurichten ist.

Wenn `alt_handler` kein NULL-Zeiger ist, so liefert diese Funktion den Signalhandler, der momentan für das Signal `signr` eingerichtet ist.

13.4.3 Struktur sigaction

Die Struktur `sigaction` ist wie folgt definiert:

```
struct sigaction {
    void      (*sa_handler)(); /* Adresse des Signalhandlers
                               oder SIG_IGN oder SIG_DFL */
    sigset_t  sa_mask; /* zusätzlich zu blockierende Signale */
    int       sa_flags; /* Signalooptionen; siehe Tabelle 13.2 */
};
```

sa_mask

Wenn man mit **sigaction** einen neuen Signalhandler einrichtet (für `sa_handler` ist weder `SIG_IGN` noch `SIG_DFL` angegeben), dann gibt `sa_mask` die Menge von Signalen an, die zur Signalmaske des Prozesses hinzuzufügen sind, bevor der entsprechende Signalhandler aufgerufen wird. Diese so modifizierte Signalmaske wird nach der Rückkehr vom Signalhandler wieder auf ihren vorherigen Wert gesetzt.

So können bestimmte Signale für die Dauer der Ausführung der Signalhandlerroutine blockiert werden. Zu dieser temporären Signalmaske wird vor dem Aufruf des Signalhandlers immer automatisch das aktuell aufgetretene Signal hinzugefügt. So ist sichergestellt, daß der Signalhandler nicht durch ein gleiches Signal unterbrochen wird, sondern dieses Signal solange blockiert wird, bis der gerade arbeitende Signalhandler sich beendet hat. Dabei ist zu beachten, daß üblicherweise gleiche Signale nicht in einer Warteschlange eingereiht werden. Dies bedeutet, daß bei mehrfachem Auftreten des gleichen Signals während einer Blockierung nach dem Beenden der Blockierung der Signalhandler nur einmal aufgerufen wird. Die anderen überschüssigen Signale sind verloren.

sa_flags

Über die Strukturkomponente `sa_flags` von `neu_handler` können Optionen für den Signalhandler gesetzt werden. Die möglichen Optionen sind in Tabelle 13.2 zusammengefaßt:

Option	POSIX.1	SVR4	BSD	Beschreibung
SA_NOCLDSTOP	x	x	x	Wenn für <code>signr SIGCHLD</code> angegeben ist, so soll dieses Signal nicht generiert werden, wenn ein Kindprozeß anhält, sondern nur, wenn ein Kindprozeß sich beendet (siehe auch Option <code>SA_NOCLDWAIT</code>).
SA_RESTART		x	x	Systemaufrufe, die durch dieses Signal unterbrochen werden, werden automatisch wieder neu gestartet.
SA_ONSTACK		x	x	Wenn mit der Funktion sigaltstack ein alternativer Stack deklariert wurde, wird dieses Signal dem Prozeß auf dem alternativen Stack geschickt.
SA_NOCLDWAIT		x		Wenn für <code>signr SIGCHLD</code> angegeben ist, so werden bei Beendigung von Kindprozessen keine Zombieprozesse generiert. Wenn der aufrufende Prozeß danach wait aufruft, so wird er solange blockiert, bis alle seine Kindprozesse beendet sind; in diesem Fall liefert wait -1 als Rückgabewert und setzt <code>errno</code> auf <code>ECHILD</code> .

Tabelle 13.2: Mögliche Optionsangaben für die Komponente `sa_flags`

Ein Aufruf von **sigaction**, bei dem die Optionen `SA_NODEFER` und `SA_RESETHAND` gesetzt sind, entspricht einem Aufruf der früheren unzuverlässigen **signal**-Funktion.

Option	POSIX.1	SVR4	BSD	Beschreibung
<code>SA_NODEFER</code>		x		Während der Ausführung der Signalhandlerroutine wird nicht automatisch das Signal blockiert; entspricht dem früheren unzuverlässigen Signalkonzept.
<code>SA_RESETHAND</code>		x		Beim Eintritt in die Signalhandlerroutine wird für Signal wieder <code>SIG_DFL</code> eingestellt; entspricht dem früheren unzuverlässigen Signalkonzept.
<code>SA_SIGINFO</code>		x		Stellt einem Signalhandler zusätzliche Information zur Verfügung (siehe auch Kapitel 13.8).

Tabelle 13.2: Mögliche Optionsangaben für die Komponente `sa_flags`



Unter Linux sind noch die folgenden Optionsangaben für die Komponente `sa_flags` möglich:

`SA_NOMASK`

Wenn der Signalhandler des Prozesses aufgerufen wird, wird das Signal nicht automatisch blockiert. Die Verwendung dieses Flags führt zu unzuverlässigen Signalen, weshalb es benutzt werden sollte, um unzuverlässige Signale in Anwendungen zu emulieren, die von diesem Verhalten abhängig sind. Somit entspricht dieses Flag dem SVR4-Flag `S_NODEFER`.

`SA_ONESHOT`

Wenn dieses Signal einem Prozeß geschickt wird, wird der Signalhandler auf `SIG_DFL` zurückgesetzt. Dieses Flag erlaubt es, das Verhalten der ANSI-C-Funktion **signal** in einer Bibliothek zu emulieren. Somit entspricht dieses Flag dem SVR4-Flag `SA_RESETHAND`.

Unter Linux wird in der Struktur `struct sigaction` eine zusätzliche Komponente angeboten:

```
void (*sa_restorer)(void);
```

Sie ist für zukünftige Erweiterungen reserviert. Zukünftige Linux-Versionen werden diese Komponente dazu verwenden, um einem Prozeß die Möglichkeit zu geben, einen alternativen Speicherbereich festzulegen, der als Stack während der Ausführung des Signalhandlers benutzt werden soll. Dazu muß allerdings auch noch ein neues `sa_flags`-Flag angeboten werden.



13.4.4 Nachbildung der `signal`-Funktion mit `sigaction`

SVR4 bietet im Gegensatz zu BSD-Unix immer noch die alte unzuverlässige Funktion **`signal`** an. Deshalb sollte man unter SVR4 entweder mit der neuen Funktion **`sigaction`** oder aber mit der folgenden Implementierung der **`signal`**-Funktion arbeiten.

```
#include <signal.h>
#include "eighdr.h"

sigfunk *signal(int signr, sigfunk *sighandler)
{
    struct sigaction neu_handler, alt_handler;

    neu_handler.sa_handler = sighandler;
    sigemptyset(&neu_handler.sa_mask);
    neu_handler.sa_flags = 0;

    if (signr == SIGALRM) {
#ifdef SA_INTERRUPT
        neu_handler.sa_flags |= SA_INTERRUPT; /* Solaris */
#endif
    } else {
#ifdef SA_RESTART
        neu_handler.sa_flags |= SA_RESTART; /* SVR4, BSD */
#endif
    }

    if (sigaction(signr, &neu_handler, &alt_handler) < 0)
        return(SIG_ERR);

    return(alt_handler.sa_handler);
}
```

Programm 13.4 (**`signal.c`**): Implementierung der **`signal`**-Funktion mittels **`sigaction`**

Lediglich für das Signal `SIGALRM` wird der automatische Start einer unterbrochenen Systemroutine verboten. Dies ist sinnvoll, wenn man mit `SIGALRM` eine Zeitschaltuhr für E/A Operationen einrichten möchte.

13.4.5 `sigprocmask` – Erfragen oder Ändern einer Signalmaske

Die Signalmaske eines Prozesses ist die Menge aller Signale, die momentan für diesen Prozeß blockiert ist. Blockiert bedeutet dabei, daß diese Signale nicht dem Prozeß zuge-
stellt werden können.

Zum Erfragen oder Ändern der Signalmaske eines Prozesses steht die Funktion **`sigprocmask`** zur Verfügung.

```
#include <signal.h>
```

```
int sigprocmask(int wie, const sigset_t *smenge, sigset_t *alt_smenge);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Für die Funktion **sigprocmask** sind folgende Fälle zu unterscheiden:

- ▶ Signalmaske ohne Ändern erfragen (`smenge == NULL, alt_smenge != NULL`).
In diesem Fall schreibt **sigprocmask** die aktuelle Signalmaske an die Adresse `alt_smenge`. Das Argument `wie` hat in diesem Fall keinerlei Bedeutung.
- ▶ Signalmaske ohne Erfragen ändern (`smenge != NULL, alt_smenge == NULL`).
In diesem Fall legt das Argument `wie` fest, wie die momentane Signalmaske zu modifizieren ist (siehe Tabelle 13.3).
- ▶ Signalmaske mit Erfragen ändern (`smenge != NULL, alt_smenge != NULL`).
In diesem Fall wird die aktuelle Signalmaske an die Adresse `alt_smenge` geschrieben, bevor die Signalmaske entsprechend dem Argument `wie` (siehe Tabelle 13.3) modifiziert wird.

wie-Argument	Beschreibung
SIG_BLOCK	Zur aktuellen Signalmaske des Prozesses werden die Signale aus *smenge hinzugefügt (entspricht bitweises OR ()).
SIG_UNBLOCK	Aus der aktuellen Signalmaske des Prozesses werden die Signale aus *smenge entfernt (entspricht <code>alte_signalmaske & ~(*smenge)</code>).
SIG_SETMASK	Die neue Signalmaske wird mit den Signalen besetzt, die in *smenge angegeben sind.

Tabelle 13.3: Mögliche Angaben für `wie` bei **sigprocmask** und deren Wirkung

Wenn nach dem Aufruf von **sigprocmask** irgendwelche nicht blockierten Signale hängen, so wird mindestens eines dieser Signale dem Prozeß zugestellt, bevor **sigprocmask** sich beendet.

Beispiel

Ausgeben der aktuellen Signalmaske

Das Programm 13.5 (`pr_mask.c`) enthält eine Funktion `print_mask`, mit der man sich die aktuelle Signalmaske ausgeben lassen kann.

```
#include <errno.h>
#include <signal.h>
#include "eighdr.h"
```

```
#define ausgab(sigmask,name) \
```

```

        if (sigismember(&sigmask, name)) printf("%s, ", #name);

void print_smask(char *string)
{
    sigset_t    sigmaske;
    int         alt_errno=errno;

    if (sigprocmask(0, NULL, &sigmaske) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
    printf("%s: ", string);

    ausgab(sigmaske, SIGABRT)
    ausgab(sigmaske, SIGALRM)
    ausgab(sigmaske, SIGBUS)
    ausgab(sigmaske, SIGCHLD)
    ausgab(sigmaske, SIGCONT)
    ausgab(sigmaske, SIGFPE)
    ausgab(sigmaske, SIGHUP)
    ausgab(sigmaske, SIGILL)
    ausgab(sigmaske, SIGINT)
    ausgab(sigmaske, SIGIO)
    ausgab(sigmaske, SIGIOT)
    ausgab(sigmaske, SIGKILL)
    ausgab(sigmaske, SIGPIPE)
    ausgab(sigmaske, SIGPOLL)
    ausgab(sigmaske, SIGPROF)
    ausgab(sigmaske, SIGPWR)
    ausgab(sigmaske, SIGQUIT)
    ausgab(sigmaske, SIGSEGV)
    ausgab(sigmaske, SIGSTOP)
    ausgab(sigmaske, SIGSYS)
    ausgab(sigmaske, SIGTERM)
    ausgab(sigmaske, SIGTRAP)
    ausgab(sigmaske, SIGTSTP)
    ausgab(sigmaske, SIGTTIN)
    ausgab(sigmaske, SIGTTOU)
    ausgab(sigmaske, SIGURG)
    ausgab(sigmaske, SIGUSR1)
    ausgab(sigmaske, SIGUSR2)
    ausgab(sigmaske, SIGVTALRM)
    ausgab(sigmaske, SIGWINCH)
    ausgab(sigmaske, SIGXCPU)
    ausgab(sigmaske, SIGXFSZ)

    printf("\b \n");
    errno = alt_errno;
}

```

Programm 13.5 (pr_smask.c): Ausgeben der Signalmaske eines Prozesses

13.4.6 sigpending – Erfragen von blockierten Signalen, die momentan hängen

Um die Menge von Signalen zu erfragen, deren Zustellung blockiert ist und die momentan hängen, steht die Funktion ***sigpending*** zur Verfügung.

```
#include <signal.h>
```

```
int sigpending(sigset_t *smenge);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Menge der momentan hängenden Signale schreibt die Funktion ***sigpending*** an die Adresse smenge.

Beispiel

Blockieren von Signalen und Erfragen von hängenden Signalen

Das Programm 13.6 (sigproc.c) demonstriert die Anwendung der Funktionen ***sigprocmask*** und ***sigpending***.

```
#include <signal.h>
#include "eighdr.h"

static void sig_int(int);

int
main(void)
{
    sigset_t blockmaske, sigmaske, haengmaske;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_int nicht installieren");

    sigemptyset(&blockmaske); /* blockmaske mit SIGINT setzen */
    sigaddset(&blockmaske, SIGINT);

                                /* Signal SIGINT blockieren */
    if (sigprocmask(SIG_BLOCK, &blockmaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigprocmask");

    sleep(10); /* Falls SIGINT hier generiert wird, wird es blockiert */

    /* Erfragen von haengenden Signalen und Ausgabe, ob SIGINT haengt */
    if (sigpending(&haengmaske) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigpending");
    if (sigismember(&haengmaske, SIGINT))
        printf("--- SIGINT haengt ---\n");

                                /* Blockierung fuer SIGINT wieder aufheben */
}
```

```

    if (sigprocmask(SIG_UNBLOCK, &blockmaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigprocmask");
    printf("----- Blockierung fuer SIGINT wieder aufgehoben -----\\n");

    sleep(10);    /* Eintreffen von SIGINT beendet den Prozess          */

    exit(0);
}

static void sig_int(int signr)
{
    printf("SIGINT abgefangen; SIG_DFL wird nun fuer SIGINT installiert\\n");

    if (signal(SIGINT, SIG_DFL) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann SIG_DFL nicht fuer SIGINT installieren");
}

```

Programm 13.6 (sigproc.c): Signale blockieren und Erfragen von hängenden Signalen

Nachdem man dieses Programm 13.6 (sigproc.c) kompiliert und gelinkt hat

cc -o sigproc sigproc.c fehler.c

ergibt sich z.B. der folgende Ablauf:

```

$ sigproc
Ctrl-C          [Generiere Signal SIGINT einmal (bevor 10 Sek. vorbei sind)]
--- SIGINT haengt --- [Ausgabe nach Rueckkehr aus sleep]
SIGINT abgefangen; SIG_DFL wird nun fuer SIGINT installiert [Nach sigprocmask-Rueckkehr]
----- Blockierung fuer SIGINT wieder aufgehoben -----
Ctrl-C          [Erneutes Generieren von SIGINT bewirkt Programmabbruch,]
                  [da Signalhandler nun auf SIG_DFL eingerichtet]

$ sigproc
Ctrl-C Ctrl-C Ctrl-C [Generiere Signal SIGINT dreimal (bevor 10 Sek. vorbei sind)]
--- SIGINT haengt --- [Ausgabe nach Rueckkehr aus sleep]
SIGINT abgefangen; SIG_DFL wird nun fuer SIGINT installiert [SIGINT nur 1mal generiert]
----- Blockierung fuer SIGINT wieder aufgehoben -----
Ctrl-C          [Erneutes Generieren von SIGINT bewirkt Programmabbruch,]
                  [da Signalhandler nun auf SIG_DFL eingerichtet]

$

```

Beim zweiten Ablaufbeispiel wird sofort nach dem Programmstart das Signal SIGINT mehrmals generiert, während der Prozeß mit sleep(10) für 10 Sekunden angehalten ist. Trotzdem wird das Signal SIGINT nach der Aufhebung der Blockierung nur einmal zugestellt. Dies zeigt, daß Signale, die üblicherweise nicht sofort zugestellt werden können, nicht in einer Warteschlange eingereiht werden, sondern verlorengehen.

13.4.7 Erlaubte Systemaufrufe in Signalhandlern (Reentrant-Funktionen)

Wenn ein Prozeß für ein Signal einen eigenen Signalhandler eingerichtet hat, dann wird beim Eintreffen dieses Signals die normale Ausführung des Prozesses kurzzeitig unterbrochen und der Code des eingerichteten Signalhandlers ausgeführt. Nach der Rückkehr aus dem Signalhandler setzt der Prozeß seine Ausführung an der unterbrochenen Stelle wieder fort.

Nun existieren aber Funktionen, die vollständig ausgeführt sein müssen, bevor sie »schadlos« wieder aufgerufen werden können. Man denke dabei nur an eine Speicherallokierung mit **malloc**. Wird **malloc** zu einem Zeitpunkt durch ein Signal unterbrochen, in dem es gerade seine verkettete Liste von allokierten Speicherbereichen ändert, und im betreffenden Signalhandler wird dann erneut **malloc** aufgerufen, so führt dies zwangsläufig zu einer inkonsistenten Speicherverwaltung mit wahrscheinlich schlimmen Folgen für den entsprechenden Prozeß.

Im Gegensatz zu solchen Funktionen, die während ihrer Ausführung nicht erneut aufgerufen werden dürfen, existieren aber auch Funktionen, die problemlos während ihrer Ausführung wieder aufgerufen werden dürfen. Solche Funktionen sind *reentrant*. Signalhandler sollten also grundsätzlich nur *Reentrant-Funktionen* aufrufen.

POSIX.1 benennt die Funktionen, die in jedem Fall reentrant sein müssen (siehe Tabelle 13.4).

_exit	fstat	read	sysconf
access	getegid	rename	tcdrain
alarm	geteuid	rmdir	tcflow
cfgetispeed	getgid	setgid	tcflush
cfgetospeed	getgroups	setpgid	tcgetattr
cfsetispeed	getpgrp	setsid	tcgetpgrp
cfsetospeed	getpid	setuid	tcsendbreak
chdir	getppid	sigaction	tcsetattr
chmod	getuid	sigaddset	tcsetpgrp
chown	kill	sigdelset	time
close	link	sigemptyset	times
creat	lseek	sigfillset	umask
dup	mkdir	sigismember	uname
dup2	mkfifo	sigpending	unlink
execle	open	sigprocmask	utime
execve	pathconf	sigsuspend	wait
fcntl	pause	sleep	waitpid
fork	pipe	stat	write

Tabelle 13.4: Reentrant-Funktionen (nach POSIX.1), die in Signalhandlern aufgerufen werden dürfen

SVR4 garantiert zusätzlich zu den Funktionen aus Tabelle 13.4, daß die Funktionen **abort**, **exit**, **longjmp** und **signal** reentrant sind.

13.5 Senden von Signalen mit den Funktionen **kill** und **raise**

Zum Senden von Signalen stehen die beiden Funktionen **kill** und **raise** zur Verfügung.

13.5.1 **raise** – Senden eines Signals an den eigenen Prozeß

Mit der Funktion **raise** kann sich ein Prozeß selbst ein Signal schicken.

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int raise(int signr);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die Funktion **raise** ist Bestandteil von ANSI C, aber nicht von POSIX1.

13.5.2 **kill** – Senden eines Signals an einen anderen Prozeß oder Prozeßgruppe

Um anderen Prozessen ein Signal zu schicken, steht die Funktion **kill** zur Verfügung.

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill(pid_t pid, int signr);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Man unterscheidet vier mögliche Angaben für das Argument *pid*:

pid > 0

Das Signal *signr* wird an den Prozeß geschickt, dessen Prozeß-ID *pid* ist.

pid == 0

Das Signal *signr* wird an alle Prozesse geschickt, deren Prozeßgruppen-ID gleich der Prozeßgruppen-ID des Senders ist, soweit der Sender die entsprechenden Rechte zum Senden dieses Signals besitzt. Üblicherweise können keine Signale an folgende Systemprozesse geschickt werden: *Swapper* (PID=0), *init* (PID>1) und *Pagedaemon* (PID=2).


```
pid < -1
```

Das Signal *signr* wird allen Prozessen geschickt, deren Prozeßgruppen-ID gleich dem absoluten Wert von *pid* ist, soweit der Sender die entsprechenden Rechte zum Senden dieses Signals besitzt. Üblicherweise können keine Signale an folgende Systemprozesse geschickt werden: *Swapper* (PID=0), *init* (PID>1) und *Pagedaemon* (PID=2).

```
pid == -1
```

Diese spezielle Angabe wird zwar von POSIX.1 nicht unterstützt, aber von SVR4 und BSD-Unix für sogenannte *broadcast signals* benutzt. *Broadcast-Signale* sollten nur für administrative Zwecke benutzt werden, wie z.B. von einem Superuser-Prozeß, um ein *shutdown* des Systems zu veranlassen. Wenn nämlich der **kill**-Aufrufer der Superuser ist, so wird das Signal an alle Prozesse (außer *swapper*, *init* und *Pagedaemon*) geschickt. Falls der Aufrufer nicht der Superuser ist, so wird das Signal allen Prozessen geschickt, deren reale User-ID oder saved Set-User-ID gleich der realen User-ID oder effektiven User-ID des Aufrufers ist.

Es ist noch darauf hinzuweisen, daß BSD-Unix niemals ein *broadcast-Signal* an den Senderprozeß schickt.

Benötigte Rechte zum Senden von Signalen

Damit ein Prozeß anderen Prozessen ein Signal schicken kann, muß er entsprechende Rechte besitzen. Nachfolgend sind die dabei geltenden Regeln aufgelistet:

- ▶ Der Superuser kann allen Prozessen Signale schicken.
- ▶ Bei Nicht-Superuser-Prozessen muß die reale oder effektive User-ID des Senders gleich der realen oder effektiven User-ID des Empfängers sein. Falls – wie in SVR4 – `_POSIX_SAVED_IDS` unterstützt wird, dann wird beim Empfänger anstelle der effektiven User-ID die saved Set-User-ID zur Prüfung auf Berechtigung herangezogen.
- ▶ Das Signal `SIGCONT` kann jeder Prozeß an alle Mitglieder der gleichen Session schicken.

Senden des Null-Signals

Wird beim Aufruf von **kill** für das Argument *signr* die 0 (in POSIX.1 als *Null-Signal* definiert) angegeben, so sendet **kill** kein Signal, sondern führt lediglich eine Prüfung durch, ob an den betreffenden Prozeß oder die Prozeßgruppe ein Signal geschickt werden kann.

Das Nullsignal wird meist geschickt, um zu überprüfen, ob ein bestimmter Prozeß noch existiert. Falls der betreffende Prozeß nämlich nicht mehr existiert, so liefert **kill** als Rückgabewert -1 und setzt *errno* auf `ESRCH`.

Das folgende Programm 13.7 (*kill0.c*) gibt beim Aufruf die Prozeß-IDs aller Prozesse aus, an die es Signale schicken kann.

```
#include <sys/types.h>
#include <signal.h>
#include "eighdr.h"
```

```

/*----- main -----*/
int
main(void)
{
    long    i, max_kind;

    max_kind = sysconf(_SC_CHILD_MAX);
    printf(" An folgende Prozesse kann ein Signal geschickt werden:\n");
    for (i=1 ; i<=max_kind ; i++)
        if (kill(i, 0) != -1)
            printf("PID %d\n", i);
    exit(0);
}

```

Programm 13.7 (**kill0.c**). Ermitteln aller Prozesse, an die das Senden von Signalen möglich ist

13.6 Einrichten einer Zeitschaltuhr und Suspendieren eines Prozesses

Zum Einrichten einer Zeitschaltuhr (*timer*) wird die Funktion **alarm** und zum Suspendieren eines Prozesses die Funktion **pause** angeboten.

13.6.1 alarm und setitimer – Einrichten von Zeitschaltuhren

Zum Einrichten einer Zeitschaltuhr steht die Funktion **alarm** zur Verfügung.

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int sekunden);
```

gibt zurück: 0 oder Anzahl der Sekunden, bis eine zuvor eingerichtete Zeitschaltuhr abläuft

Wenn die mit **alarm** eingerichtete Zeitschaltuhr abgelaufen ist, wird das Signal **SIGALRM** generiert. Wird dieses Signal ignoriert oder nicht abgefangen, so beendet sich der Prozeß, was die voreingestellte Default-Aktion für dieses Signal ist.

Das Argument *sekunden* gibt an, in wieviel Sekunden die Zeitschaltuhr ablaufen und das Signal **SIGALRM** generiert werden soll. Es ist dabei zu beachten, daß diese mit *sekunden* angegebene Zeit für den Ablauf der Zeitschaltuhr nicht immer genau eingehalten werden kann, denn zwischen dem Generieren des Signals durch den Kern und der Zustellung an den Prozeß vergeht weitere Zeit, welche vor allen Dingen durch Verzögerungen beim Prozessor-Scheduling nicht ganz unerheblich sein kann.

Wenn bei einem **alarm**-Aufruf eine zuvor mit **alarm** eingerichtete Zeitschaltuhr noch nicht abgelaufen ist, so wird diese alte Zeitschaltuhr durch die neue ersetzt. Als Rückgabewert wird in diesem Fall die Anzahl der Sekunden geliefert, die für den Ablauf der alten Zeitschaltuhr verbleiben.

Ist bei einem **alarm**-Aufruf eine zuvor eingerichtete Zeitschaltuhr noch nicht abgelaufen und wurde für das Argument `sekunden` der Wert 0 angegeben, so wird diese noch laufende Zeitschaltuhr ausgeschaltet, und als Rückgabewert wird wieder die Anzahl der Sekunden geliefert, bis diese alte Zeitschaltuhr abgelaufen wäre.

Typische Anwendung für die Funktion **alarm**

Eine typische Anwendung für **alarm** ist das Festlegen einer oberen Zeitgrenze für eine Aktion, die blockiert werden kann. Wenn man z.B. von einem Peripheriegerät liest, das blockiert werden kann, so ist es sinnvoll, die Leseoperation nach Ablauf einer gewissen Zeitspanne abubrechen, da dann angenommen werden kann, daß das Peripheriegerät blockiert ist. Das Programm 13.8 (`alarmread.c`) zeigt diese Technik anhand des Lesens einer Zeile von der Standardeingabe und dem Schreiben der gelesenen Zeile auf die Standardausgabe.

```
#include <signal.h>
#include "eighdr.h"

static void sig_alarm(int signr);

int
main(void)
{
    int    n;
    char   zeile[MAX_ZEICHEN];

    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_alarm nicht installieren");

    alarm(60); /* 60 Sek. fuer folgende Leseoperation vorgeben */
               /* Ist die Leseoperation in dieser Zeit nicht abgeschlossen, */
               /* so wird SIGALRM geschickt und Leseoperation abgebrochen */

    if ( (n=read(STDIN_FILENO, zeile, MAX_ZEICHEN)) < 0)
        fehler_meld(FATAL_SYS, "Lesefehler aufgetreten");

    alarm(0); /* Zeitschaltuhr wieder ausschalten */

    write(STDOUT_FILENO, zeile, n);

    exit(0);
}

static void sig_alarm(int signr)
{
    return; /* keinerlei Aktion; nur Rueckkehr, um read abubrechen */
}
```

Programm 13.8 (alarmread.c): Abbruch von **read** nach Ablauf einer Zeitschaltuhr

Bei diesem Programm 13.8 (`alarmread.c`) besteht jedoch das Problem, daß, wenn unterbrochene Systemaufrufe automatisch wieder gestartet werden, die Funktion **read** nicht abgebrochen wird, wenn der `SIGALRM`-Signalhandler sich beendet, sondern wieder von neuem gestartet wird. In diesem Fall hat das Einrichten einer Zeitschaltuhr zum automatischen Abbruch der Leseoperation nach einer bestimmten Zeit keinerlei Auswirkung. Deswegen ist das folgende Programm 13.9 (`alarmrea2.c`), das die Funktion **longjmp** verwendet, dem vorherigen Programm vorzuziehen, da es auch beim automatischen Neustart von unterbrochenen Systemroutinen funktioniert.

```
#include <setjmp.h>
#include <signal.h>
#include "eighdr.h"

static void    sig_alrm(int signr);
static jmp_buf progzust;

int
main(void)
{
    int    n;
    char   zeile[MAX_ZEICHEN];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_alrm nicht installieren");

    if (setjmp(progzust) != 0)
        fehler_meld(FATAL, "Timer fuer read abgelaufen");

    alarm(60); /* 60 Sek. fuer folgende Leseoperation vorgeben          */
               /* Ist die Leseoperation in dieser Zeit nicht abgeschlossen, */
               /* so wird SIGALRM geschickt und Leseoperation abgebrochen */

    if ( (n=read(STDIN_FILENO, zeile, MAX_ZEICHEN)) < 0)
        fehler_meld(FATAL_SYS, "Lesefehler aufgetreten");

    alarm(0); /* Zeitschaltuhr wieder ausschalten */

    write(STDOUT_FILENO, zeile, n);

    exit(0);
}

static void sig_alrm(int signr)
{
    longjmp(progzust, 1);
}
```

Programm 13.9 (`alarmrea2.c`): Abbruch von **read** nach Ablauf einer Zeitschaltuhr (unter Verwendung von **longjmp**)

Neben der Funktion **alarm** bieten viele Unix-Systeme, wie auch Linux, noch sogenannte Intervalltimer an. Ist ein Intervalltimer einmal aktiviert, schickt er ständig nach bestimmten Regeln ein Signal zu einem Prozeß. Systeme, die Intervalltimer anbieten, stellen jedem Prozeß automatisch drei Intervalltimer zur Verfügung:

ITIMER_REAL

läuft in Echtzeit und schickt nach dem Ablauf seiner Zeitschaltuhr das Signal SIGALRM; dieser Intervalltimer sollte nicht zusammen mit den Funktionen **alarm** und **sleep** verwendet werden, um Konflikte zu vermeiden.

ITIMER_VIRTUAL

läßt seine Zeitschaltuhr nur laufen, wenn der Prozeß im Benutzermodus läuft, also nicht beim Aufruf von Systemfunktionen. Nach Ablauf seiner Zeitschaltuhr schickt er das Signal SIGVTALRM.

ITIMER_PROF

läßt seine Zeitschaltuhr laufen, wenn der Prozeß im Benutzer- oder im Systemmodus läuft. Nach Ablauf seiner Zeitschaltuhr schickt er das Signal SIGPROF. Zusammen mit ITIMER_VIRTUAL können so die beiden Zeiten ermittelt werden, die der Prozeß im Benutzer- und die er im Systemmodus verbringt.

Wenn einer der obigen Timer abläuft, wird dem Prozeß das entsprechende Signal geschickt und der Timer wird neu gestartet.

Nach Ablauf eines Timers wird dessen Signal innerhalb eines Taktes der Systemuhr dem entsprechenden Prozeß zugestellt. Typische Taktwerte sind 1 ms oder 10 ms. Wird der Prozeß gerade ausgeführt, wenn das Signal auftritt, wird dieses sofort zugestellt, ansonsten unmittelbar danach, was von der aktuellen Systemlast abhängig ist. Da der Timer ITIMER_VIRTUAL nur während der Ausführung des Prozesses läuft, wird dessen Signal immer sofort zugestellt.

Um Timer zu setzen oder abzufragen, stehen die beiden folgenden in `<sys/time.h>` bzw. `<linux/time.h>` definierten Strukturen zur Verfügung:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
};

struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
```

Die Komponente `it_value` enthält die verbleibende Zeit bis zum Schicken des nächsten Signals und die Komponente `it_interval` enthält die gesamte Intervallzeit zwischen zwei Signalen.

Nach jedem Ablauf eines Timers wird der Wert von `it_interval` wieder in die Komponente `it_value` geschrieben, um den Timer erneut zu starten.

Für das Arbeiten mit Intervalltimern stehen die beiden Funktionen **getitimer** und **setitimer** zur Verfügung.

```
#include <sys/time.h>
```

```
int getitimer(int which, struct itimerval *wert);
```

```
int setitimer(int which, const struct itimerval *neu,  
             struct itimerval *alt);
```

beide geben zurück 0 (bei Erfolg); -1 bei Fehler

Über den Parameter *which* wird bei beiden Funktionen der entsprechende Timer ausgewählt: `ITIMER_REAL`, `ITIMER_VIRTUAL` oder `ITIMER_PROF`.

Die Funktion **getitimer** schreibt die aktuellen Werte des entsprechenden Timers in die Struktur, auf die der Parameter *wert* zeigt.

Die Funktion **setitimer** setzt den über *which* ausgewählten Timer auf die mit dem Parameter *neu* festgelegten Werte. Wird für den Parameter *alt* kein NULL-Zeiger angegeben, so schreibt **setitimer** die vorherigen Werte des Timers in die Struktur, auf die der Parameter *alt* zeigt.

Setzt man die Komponente `it_value` eines Timers auf 0, wird dieser sofort abgeschaltet. Wird dagegen die Komponente `it_interval` eines Timers auf 0 gesetzt, so wird dieser erst abgeschaltet, nachdem er abgelaufen ist.

13.6.2 pause – Suspendieren eines Prozesses (bis Eintreffen eines Signals)

Um einen Prozeß zu suspendieren, bis ein Signal eintrifft, steht die Funktion **pause** zur Verfügung.

```
#include <unistd.h>
```

```
int pause(void);
```

gibt zurück: -1, wobei `errno` auf `EINTR` gesetzt wird.

Ein mit **pause** suspendierter Prozeß bleibt so lange suspendiert, bis er ein Signal empfängt. Nur der Aufruf eines Signalhandlers und seine anschließende Beendigung bewirkt eine Rückkehr aus der **pause**-Funktion.

13.6.3 **sleep**, **usleep**, **select** und **nanosleep** – Suspendieren eines Prozesses (für eine bestimmte Zeit)

Um einen Prozeß für eine bestimmte Zeit oder bis zum Eintreffen eines Signals zu suspendieren, steht die Funktion **sleep** zur Verfügung.

```
#include <unistd.h>

unsigned int sleep(unsigned int sekunden);
```

gibt zurück: 0 oder Anzahl der nicht geschlafenen Sekunden.

Die Funktion **sleep** suspendiert den aufrufenden Prozeß bis entweder

- ▶ die als Argument angegebenen *sekunden* vergangen sind (Rückgabewert 0), oder
- ▶ ein Signal durch den Prozeß abgefangen wurde und sich der entsprechende Signalhandler beendet. In diesem Fall wird die Anzahl der nicht »geschlafenen« *sekunden* als Rückgabewert geliefert.

Neben **sleep** werden auf den meisten Unix-Systemen, so auch unter Linux, noch die folgenden drei Funktionen zum Suspendieren eines Prozesses angeboten:

```
#include <unistd.h>

void usleep(unsigned long usec);
```

Die Funktion **usleep** suspendiert den aufrufenden Prozeß für mindestens *usec* Mikrosekunden. Bei **usleep**, das meist unter Zuhilfenahme der Funktion **select** implementiert ist, werden keine Signale benutzt.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(0, NULL, NULL, NULL, struct timeval *timeout);
```

Die Funktion **select** wird zwar erst in Kapitel 15.1.5 genauer beschrieben, kann aber in dieser Form des Aufrufs auch dazu verwendet werden, um die Ausführung eines Prozesses für eine bestimmte Zeit zu suspendieren.

Die Struktur `timeval` ist in `<sys/time.h>` bzw. `<linux/time.h>` wie folgt definiert:

```
struct timeval {
    long tv_sec;    /* Sekunden */
    long tv_usec;   /* Mikrosekunden */
};
```

Man muß also nur die beiden Komponenten `tv_sec` und `tv_usec` des übergebenen Zeigers `timeout` vor dem Aufruf von ***select*** entsprechend setzen, um den aufrufenden Prozeß dann so lange zu suspendieren.

Mit der Funktion ***nanosleep*** steht dann noch eine dritte Möglichkeit für die Suspendierung eines Prozesses zur Verfügung.

```
#include <time.h>

int nanosleep(const struct timespec *req, struct timespec *rem);

gibt zurück: 0 (bei Erfolg); -1 bei Fehler oder frühzeitigem Abbruch durch Signal
```

Die Funktion ***nanosleep*** suspendiert einen Prozeß für die Zeitdauer, die über den Parameter `req` festgelegt ist. Die Struktur `timespec` ist in `<sys/time.h>` bzw. `<linux/time.h>` wie folgt definiert:

```
struct timespec {
    long tv_sec;    /* Sekunden */
    long tv_nsec;   /* Nanosekunden */
};
```

Kehrt die Funktion ***nanosleep*** aufgrund des Empfangs eines Signals früher zurück, liefert sie -1 als Rückgabewert, setzt die globale Variable `errno` auf `EINTR` und schreibt die noch verbleibende Zeit an den Speicherplatz, auf den `rem` zeigt, wenn für diesen Parameter nicht NULL angegeben wurde.

Zu ***nanosleep*** ist noch anzumerken, daß nicht alle Rechner über die Fähigkeit verfügen, Zeiten im Nanosekunden-Bereich zu messen, woraus dann natürlich eine gewisse Ungenauigkeit resultiert, da die angegebenen Nanosekunden dann zum nächstmöglichen Zeittakt aufgerundet werden.

13.6.4 Mögliche Implementierungen für `sleep`

sleep1 – Implementierung von `sleep` mit `alarm` und `pause`

Das Programm 13.10 (`sleep1.c`) enthält eine mögliche Realisierung von ***sleep*** (hier ***sleep1*** genannt) unter Verwendung der Funktionen ***alarm*** und ***pause***.

```
#include <signal.h>
#include <unistd.h>
#include "eighdr.h"
```



```

static void sig_alm(int signr)
{
    return; /* keinerlei Aktionen; nur Rueckkehr, um pause wieder aufzuwecken */
}

unsigned int sleep(unsigned int sekunden)
{
    sigfunk      *alt_sighandler;
    unsigned int  alt_schaltzeit,
                  rest_zeit,
                  sleep_zeit;

    if ( (alt_sighandler=signal(SIGALRM, sig_alm)) == SIG_ERR)
        return(sekunden);

    alt_schaltzeit = alarm(sekunden); /* Laeuft noch eine andere Schaltuhr ? */

    if (alt_schaltzeit == 0) {
        rest_zeit = 0;           /* keine andere Schaltuhr momentan laufend */
        sleep_zeit = sekunden;   /* --> Schaltuhr mit vorgeg. Zeit einrichten */

    } else if (alt_schaltzeit < sekunden) {
        rest_zeit = 0;           /* alte Schaltuhr laeuft frueher ab */
        sleep_zeit = alt_schaltzeit; /* --> alte Schaltuhr wieder einrichten */

    } else if (alt_schaltzeit >= sekunden) {
        rest_zeit = alt_schaltzeit-sekunden; /* alte Schaltuhr laeuft spaeter ab */
        sleep_zeit = sekunden;           /* --> Uhr mit vorg. Zeit einricht. */
    }

    alarm(sleep_zeit);

    pause();    /* Auf Abfangen eines Signals warten */

    if ( signal(SIGALRM, alt_sighandler) == SIG_ERR )
        fehler_meld(WARNUNG, "kann alten Signalhandler nicht mehr einrichten");

    return( alarm(rest_zeit) );
}

```

Programm 13.10 (**sleep1.c**): Einfache Implementierung von **sleep**

In früheren Systemen war **sleep** ähnlich umgesetzt. Dabei konnte jedoch eine *race condition* (siehe Kapitel 10.4) zwischen dem zweiten Aufruf von **alarm** und dem Aufruf von **pause** auftreten. Wenn nämlich an einem überlasteten System die mit **alarm** eingerichtete Zeitschaltuhr ablief und der Signalhandler aufgerufen wurde, bevor **pause** aufgerufen wurde, so wurde der Prozeß für immer durch den nun erst folgenden **pause**-Aufruf suspendiert, wenn nicht weitere Signale abgefangen wurden. Dieses Problem ist im folgenden Programm 13.11 (**sleep2.c**) unter Verwendung der Funktionen **setjmp** und **longjmp** behoben.

sleep2 – Implementierung von sleep mit alarm, pause, setjmp und longjmp

Im folgenden Programm 13.11 (sleep2.c) ist die *race condition* aus Programm 13.10 (sleep1.c) behoben. Die geänderten oder neu hinzugekommenen Zeilen sind im folgenden Listing fett hervorgehoben.

```
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include "eighdr.h"

static jmp_buf progzust;

static void sig_alrm(int signr)
{
    longjmp(progzust, 1);
}

unsigned int sleep(unsigned int sekunden)
{
    sigfunk      *alt_sighandler;
    unsigned int  alt_schaltzeit,
                 rest_zeit,
                 sleep_zeit;

    if ( (alt_sighandler=signal(SIGALRM, sig_alrm)) == SIG_ERR)
        return(sekunden);

    alt_schaltzeit = alarm(sekunden); /* Laeuft noch eine andere Schaltuhr ? */

    if (alt_schaltzeit == 0) {
        rest_zeit = 0;          /* keine andere Schaltuhr momentan laufend */
        sleep_zeit = sekunden;  /* --> Schaltuhr mit vorgeg. Zeit einrichten */
    } else if (alt_schaltzeit < sekunden) {
        rest_zeit = 0;          /* alte Schaltuhr laeuft frueher ab */
        sleep_zeit = alt_schaltzeit; /* --> alte Schaltuhr wieder einrichten */
    } else if (alt_schaltzeit >= sekunden) {
        rest_zeit = alt_schaltzeit-sekunden; /* alte Schaltuhr laeuft spaeter ab*/
        sleep_zeit = sekunden;          /* --> Uhr mit vorg. Zeit einricht.*/
    }

    if (setjmp(progzust) == 0) {
        alarm(sleep_zeit); /* Zeitschaltuhr starten */
        pause();           /* Auf Abfangen eines Signals warten */
    }

    if ( signal(SIGALRM, alt_sighandler) == SIG_ERR )
        fehler_meld(WARNUNG, "kann alten Signalhandler nicht mehr einrichten");

    return( alarm(rest_zeit) );
}
```

Programm 13.11 (sleep2.c): Verbesserte (aber noch nicht vollkommene) Implementierung von **sleep**

Sogar wenn die Funktion **pause** niemals aufgerufen wurde, ist hier beim Auftreten des Signals **SIGALRM** sichergestellt, daß die Funktion **sleep2** sich beendet.

Selbst Funktion **sleep2** ist nicht perfekt. Wenn nämlich während der Ausführung von **sleep2** ein anderes Signal auftritt, dessen Signalhandler sich nicht vor Ablauf der **sleep2**-Funktion beendet, so führt der **longjmp**-Aufruf zwangsweise zur »gewaltsamen« Beendigung des anderen Signalhandlers. Unter Verwendung der nachfolgend vorgestellten Funktionen werden wir später eine zuverlässige Implementierung von **sleep** zeigen.

13.6.5 sigsetjmp und siglongjmp – setjmp und longjmp für Signalhandler

Wenn ein Signal abgefangen wird, dann wird die entsprechende für dieses Signal eingerichtete Signalhandlerroutine aufgerufen, wobei das aktuelle Signal automatisch zur Signalmaske des Prozesses hinzugefügt wird. So wird verhindert, daß ein erneutes Auftreten des gleichen Signals die Ausführung des Signalhandlers unterbricht.

Bei einem Aufruf von **longjmp** im Signalhandler verhalten sich die einzelnen Systeme unterschiedlich. Bei einigen Systemen bleibt die aktuelle Signalmaske erhalten und bei anderen wiederum nicht. Dies ist der Grund, warum POSIX.1 die zwei Funktionen **sigsetjmp** und **siglongjmp** einführte, die man immer bei nicht-lokalen Sprüngen aus Signalhandlern (anstelle von **setjmp** und **longjmp**) verwenden sollte.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf progzust, int erhalte_smask);

    gibt zurück: 0 (bei direktem Aufruf); verschieden von 0 bei Rückkehr von einem siglongjmp-Aufruf

void siglongjmp(sigjmp_buf progzust, int wert);
```

Der einzige Unterschied zwischen diesen beiden Funktionen und den in Kapitel 8.1 beschriebenen Funktionen **setjmp** und **longjmp** ist das zusätzliche Argument **erhalte_smask** bei der Funktion **sigsetjmp**. Ist **erhalte_smask** verschieden von 0, so wird auch die aktuelle Signalmaske des Prozesses in **progzust** hinterlegt. Wurde mit **sigsetjmp** diese Signalmaske in **progzust** hinterlegt, so wird bei **siglongjmp** diese Signalmaske für den Prozeß wiederhergestellt.

Beispiel

*Demonstrationsprogramm zu den Funktionen **sigsetjmp** und **siglongjmp***

```
#include <signal.h>
#include <setjmp.h>
#include <time.h>
#include "eighdr.h"

extern void print_smask(char *string);
```

```

static void          sig_usr1(int), sig_alrm(int);
static sigjmp_buf    progzust;
static volatile sig_atomic_t  sprg_moegl=0;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_usr1 nicht installieren");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_alrm nicht installieren");

    print_smask("Am Anfang von main"); /* Aus Programm pr_smask.c */

    if (sigsetjmp(progzust, 1)) {
        print_smask("Am Ende von main");
        exit(0);
    }
    sprg_moegl = 1; /* nun ist Aufruf von sigsetjmp problemlos moeglich */

    while (1)
        pause();
}

static void sig_usr1(int signr)
{
    time_t  zeit;

    if (sprg_moegl == 0)
        return; /* Unerwartetes Signal ---> ignorieren */

    print_smask("Am Anfang von sig_usr1");

    alarm(4); /* SIGALRM in 4 Sekunden */

    zeit = time(NULL);

    while (1) /* Aktives Warten fuer 5 Sekunden */
        if (time(NULL) > zeit+5)
            break;

    print_smask("Am Ende von sig_usr1");

    sprg_moegl = 0;
    siglongjmp(progzust, 1); /* Nicht-lokaler Sprung zurueck zu main */
}

static void sig_alrm(int signr)
{
    print_smask("In sig_alrm");
    return;
}

```

Programm 13.12 (sigjmp.c): Beispiel zu *sigsetjmp* und *siglongjmp*

Das Programm 13.12 (sigjmp.c) zeigt unter anderem eine Technik, die man bei **siglongjmp** immer verwenden sollte. Bei dieser Technik wird die Variable `prg_moegl` erst nach dem Aufruf von **sigsetjmp** auf einen Wert verschieden von 0 gesetzt. Mit einer Überprüfung dieser Variablen in den entsprechenden Signalhandlern ist es möglich, **siglongjmp** erst dann aufzurufen, wenn `prg_moegl` verschieden von 0 ist. So ist sichergestellt, daß nur dann ein nicht-lokaler Sprung im Signalhandler stattfindet, wenn zuvor mit **sigsetjmp** die `sigjmp_buf`-Variable initialisiert wurde.

Der im Programm 13.12 (sigjmp.c) verwendete Datentyp `sigatomic_t` ist von ANSI C definiert. Für Variablen dieses Datentyps ist garantiert, daß beim Schreiben von Daten in diese Variablen niemals eine Unterbrechung stattfindet.

Nachdem man dieses Programm 13.12 (sigjmp.c) kompiliert und gelinkt hat

```
cc -o sigjmp sigjmp.c pr_mask.c signal.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ sigjmp &                                [sigjmp im Hintergrund starten]
[1] 1223                                   [Jobsteuerung gibt Prozeß-ID aus]
Am Anfang von main:
$ kill -USR1 1223                          [Schicken des Signals SIGUSR1 an Prozeß mit PID 1223]
Am Anfang von sig_usr1: SIGUSR1
In sig_alm: SIGUSR1,SIGALRM
Am Ende von sig_usr1: SIGUSR1
Am Ende von main:
                                           [Eingabe von Return]
[1] + Done                                sigjmp
$
```

Die Abbildung 13.1 verdeutlicht den Ablauf dieses Programms sigjmp.

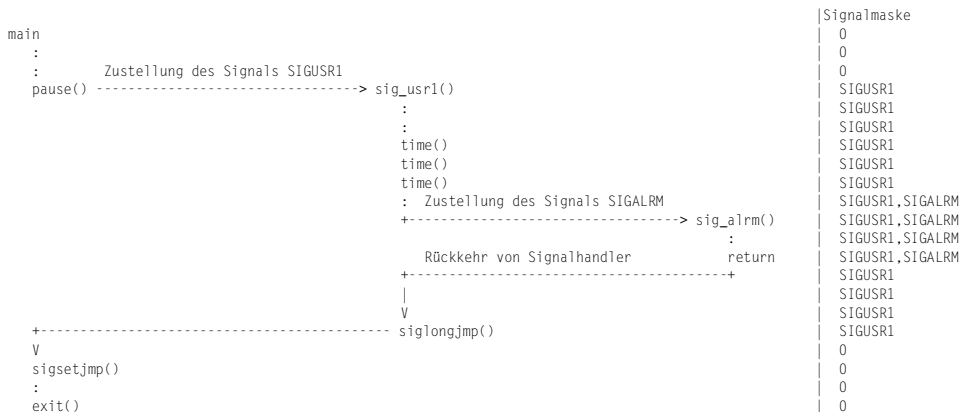


Abbildung 13.1: Erklärung des Ablaufbeispiels zu sigjmp

13.6.6 sigsuspend – Suspendieren eines Prozesses während der Änderung der Signalmaske

Manchmal ist es notwendig, daß man Signale blockiert, damit kritische Ausschnitte »ungestört« ausgeführt werden können, ohne daß sie durch diese Signale unterbrochen werden. Möchte man z.B. sicherstellen, daß ein kritischer Codeabschnitt nicht vom Benutzer durch Drücken einer Unterbrechungstaste SIGINT (*Strg-c* bzw. *DELETE*) oder SIGQUIT (*Strg-*) unterbrochen wird, bevor **pause** aufgerufen wird, so bietet sich das folgende Codestück an:

```

sigset_t neumaske, altmaske;
.....
.....
1  sigemptyset(&neumaske);
2  sigaddset(&neumaske, SIGINT);
3  sigaddset(&neumaske, SIGQUIT);
4  if (sigprocmask(SIG_BLOCK, &neumaske, &altmaske) < 0)
5      fehler_meld(FATAL_SYS, "sigprocmask-Fehler (SIG_BLOCK)");
.....
.....
6  /* kritischer Codeabschnitt */
.....
.....
7  if (sigprocmask(SIG_SETMASK, &altmaske, NULL) < 0)
8      fehler_meld(FATAL_SYS, "sigprocmask-Fehler (SIG_SETMASK)");
9  pause(); /* Auf Zustellung eines Signals warten*/
.....

```

Bei diesem Codeausschnitt tritt allerdings ein Problem auf, wenn ein Signal während der Aufhebung der Blockierung (Zeile 7) und dem Aufruf von **pause** (Zeile 9) eintrifft. Dieses Signal geht dann verloren. Das ist der Grund, warum eine eigene Funktion **sigsuspend** zur Verfügung gestellt wird, bei der das Setzen der Signalmaske und das Suspendieren des Prozesses eine einzige atomare Operation ist.

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *signalmaske);
```

gibt zurück: -1, wobei errno auf EINTR gesetzt wird

Die Funktion **sigsuspend** setzt die Signalmaske auf den Wert, auf den *signalmaske* zeigt. **sigsuspend** suspendiert den Prozeß, bis ein Signal eintrifft, das entweder abgefangen wird oder aber den Prozeß beendet. Wenn ein Signal abgefangen wird, so beendet auch **sigsuspend** sich nach Beendigung des Signalhandlers, und die Signalmaske wird auf den Wert zurückgesetzt, der vor dem Aufruf von **sigsuspend** vorlag.

Die Funktion **sigsuspend** beendet sich immer mit dem Rückgabewert -1 und dem Setzen von *errno* auf EINTR (Anzeige, daß ein Systemaufruf unterbrochen wurde).

13.6.7 Schützen eines kritischen Codeausschnitts vor Unterbrechung durch Signale

Das Programm 13.13 (`sigkrit.c`) zeigt die richtige Vorgehensweise, um einen kritischen Codeabschnitt vor der Unterbrechung durch bestimmte Signale zu schützen.

```
#include <signal.h>
#include "eighdr.h"

static void sig_int(int);
static void sig_quit(int);

int
main(void)
{
    sigset_t neumaske, altmaske, nullmaske;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_int nicht installieren");
    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_quit nicht installieren");

    sigemptyset(&nullmaske);

    sigemptyset(&neumaske);
    sigaddset(&neumaske, SIGINT);
    sigaddset(&neumaske, SIGQUIT);

    if (sigprocmask(SIG_BLOCK, &neumaske, &altmaske) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigprocmask");

    /* ..... */
    /* ..... Kritischer Codeabschnitt ..... */
    /* ..... */
    print_smask("Im kritischen Codeabschnitt");

    sigsuspend(&nullmaske); /* pause mit Zulassung aller Signale aufrufen */
    print_smask("Nach Rueckkehr von sigsuspend");

    if (sigprocmask(SIG_SETMASK, &altmaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigprocmask");

    /* ..... */

    exit(0);
}

static void sig_int(int signr)
{
    print_smask("In sig_int");
}

static void sig_quit(int signr)
```

```
{
    print_smask("In sig_quit");
}
```

Programm 13.13 (sigkrit.c): Schützen eines kritischen Codeabschnitts vor Unterbrechung durch Signale

Bei der Verwendung von ***sigsuspend*** ist zu beachten, daß diese Funktion die Signalmaske immer auf den Wert vor dem Aufruf zurücksetzt. In Programm 13.13 (sigkrit.c) werden die Signale SIGINT und SIGQUIT für die Dauer der Ausführung des kritischen Codeabschnitts blockiert, bevor mit dem Aufruf von ***sigsuspend*** die ***pause***-Funktion mit Zulassung aller Signale nachgebildet wird. Mit dem letzten ***sigprocmask*** wird dann die Signalmaske wieder auf den Wert zurückgesetzt, den sie vor dem kritischen Codeabschnitt hatte.

Nachdem man dieses Programm 13.13 (sigkrit.c) kompiliert und gelinkt hat

```
cc -o sigkrit sigkrit.c pr_smask.c signal.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ sigkrit
Im kritischen Codeabschnitt: SIGINT,SIGQUIT
Ctrl-\          [QUIT-Signal schicken]
In sig_quit: SIGQUIT
Nach Rueckkehr von sigsuspend: SIGINT,SIGQUIT
$
```

Beispiel

Abfangen mehrerer Signale, Programmfortsetzung nur bei bestimmtem Signal

Das folgende Programm 13.14 (sigmehr.c) fängt zwar die beiden Signale SIGUSR1 und SIGUSR2 ab, setzt die Programmausführung aber nur beim Empfang des Signals SIGUSR1 fort.

```
#include <signal.h>
#include "eighdr.h"

static void sig_usr(int);

volatile sig_atomic_t usr1_flag=0;

int
main(void)
{
    sigset_t neumaske, altmaske, nullmaske;

    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_usr nicht installieren");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhandler sig_usr nicht installieren");

    sigemptyset(&nullmaske);
```



```

sigemptyset(&neumaske);
sigaddset(&neumaske, SIGUSR1);

    /* Blockieren von SIGUSR1 und Aufheben der momentanen Signalmaske */
    if (sigprocmask(SIG_BLOCK, &neumaske, &altmaske) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigprocmask(SIG_BLOCK,...)");

while (usr1_flag==0)
    sigsuspend(&nullmaske); /* pause mit Zulassung aller Signale aufrufen*/

usr1_flag = 0; /* SIGUSR1 wurde abgefangen und ist nun blockiert */

    /* Signalmaske auf ursprgl. Wert zuruecksetzen */
    if (sigprocmask(SIG_SETMASK, &altmaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "Fehler bei sigprocmask");

/* ..... */

exit(0);
}

static void sig_usr(int signr)
{
    if (signr == SIGUSR1)
        usr1_flag = 1;
    else if (signr == SIGUSR2)
        printf("--- SIGUSR2 abgefangen ---\n");
}

```

Programm 13.14 (sigmehr.c): Mit **sigsuspend** auf Eintreffen bestimmter Signale warten

```

$ sigmehr &
[1] 1292
$ kill -USR2 1292
--- SIGUSR2 abgefangen ---
$ kill -USR2 1292
--- SIGUSR2 abgefangen ---
$ kill -USR2 1292
--- SIGUSR2 abgefangen ---
$ kill -USR1 1292
                                     [Eingabe von Return]
[1] + Done                          sigmehr
$

```

13.6.8 Synchronisation von Prozessen mit Signalen

Das Programm 13.15 (forksync.c) zeigt nochmals die bereits in Kapitel 10.4 vorgestellten Routinen: **INIT_SYNCH**, **HALLO_KIND**, **WARTE_AUF_KIND**, **HALLO_PAPA** und **WARTE_AUF_PAPA**, die eine Synchronisation von Eltern- und Kindprozessen mit Signalen ermöglichen. Es werden dabei die beiden benutzerdefinierten Signale SIGUSR1 (wird vom Kindprozeß an Elternprozeß geschickt) und SIGUSR2 (wird vom Elternprozeß an Kindprozeß geschickt) verwendet.

```

#include <signal.h>
#include "eighdr.h"

static volatile sig_atomic_t  sflag;
static sigset_t      neu_smaske, alt_smaske, null_smaske;

/*----- Signalhandler fuer die Signale SIGUSR1 und SIGUSR2 -----*/
static void sig_usr(int signr)
{
    INIT_SYNCH();
    sflag = 1;
}
/*----- Synchronisation initialisieren -----*/
void INIT_SYNCH(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann SIGUSR1-Signalhandler nicht installieren");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann SIGUSR2-Signalhandler nicht installieren");

    sigemptyset(&null_smaske);

    sigemptyset(&neu_smaske);
    sigaddset(&neu_smaske, SIGUSR1);
    sigaddset(&neu_smaske, SIGUSR2);

    if (sigprocmask(SIG_BLOCK, &neu_smaske, &alt_smaske) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
}

/*----- Information von Kind an Elternprozess, dass es fertig ----*/
void HALLO_PAPA(pid_t pid)
{
    kill(pid, SIGUSR2);
}

/*----- Kind wartet auf Signal vom Elternprozess -----*/
void WARTE_AUF_PAPA(void)
{
    while (sflag == 0)
        sigsuspend(&null_smaske); /* Warten auf Signal vom Elternprozess*/

    sflag = 0;

    if (sigprocmask(SIG_SETMASK, &alt_smaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
}

/*----- Information von Elternprozess an Kind, dass er fertig ist */
void HALLO_KIND(pid_t pid)
{
    kill(pid, SIGUSR1);
}

```

```

/*----- Elternprozess wartet auf Signal vom Kind -----*/
void WARTEN_AUF_KIND(void)
{
    while (sflag == 0)
        sigsuspend(&null_smaske); /* Warten auf Signal vom Elternprozess */

    sflag = 0;

    if (sigprocmask(SIG_SETMASK, &alt_smaske, NULL) < 0)
        fehler_meld(FATAL_SYS, "sigprocmask-Fehler");
}

```

Programm 13.15 (**forksync.c**): Funktionen zur Synchronisation von Eltern- und Kindprozeß

13.6.9 sleep3 – Eine zuverlässige Implementierung von sleep

Das Verstellen einer Zeitschaltuhr bei gleichzeitiger Benutzung von **sleep** und anderen Zeitfunktionen wie **alarm** oder **setitimer** wird von den unterschiedlichen Systemen auch unterschiedlich gehandhabt. Das nachfolgende Programm 13.16 (sleep3.c) ist eine Implementierung von **sleep**, die den Vorgaben von POSIX.1 entspricht.

```

#include <signal.h>
#include <stddef.h>
#include "eighdr.h"

static void sig_alm(int signr);

unsigned int sleep(unsigned int sekunden)
{
    struct sigaction neuaktion, altaktion;
    sigset_t neumaske, altmaske, suspendmaske;
    unsigned int rest_schlafzeit;

    neuaktion.sa_handler = sig_alm;
    sigemptyset(&neuaktion.sa_mask);
    neuaktion.sa_flags = 0;
    /* Eigenen Handler einrichten und vorherige Info. merken */
    sigaction(SIGALRM, &neuaktion, &altaktion);

    /* SIGALRM blockieren; alte Signalmaske merken */
    sigemptyset(&neumaske);
    sigaddset(&neumaske, SIGALRM);
    sigprocmask(SIG_BLOCK, &neumaske, &altmaske);
    alarm(sekunden);

    suspendmaske = altmaske;
    sigdelset(&suspendmaske, SIGALRM); /* Blockierung von SIGALRM aufheben */
    sigsuspend(&suspendmaske); /*Auf Eintreffen von erwartet. Signale warten*/

    rest_schlafzeit = alarm(0);
    sigaction(SIGALRM, &altaktion, NULL); /*Vorherige Aktion wieder einrichten*/

    /* Signalmaske wieder zuruecksetzen */
}

```

```

    sigprocmask(SIG_SETMASK, &altmaske, NULL);

    return(rest_schlafzeit);
}

static void sig_alm(int signr)
{
    return; /* keinerlei Aktion; nur Rueckkehr, um sigsuspend aufzuwecken */
}

```

Programm 13.16 (**sleep3.c**): Implementierung eines zuverlässigen **sleep**

In dieser Implementierung werden keine nicht-lokalen Sprünge – wie in Programm 13.11 (**sleep2.c**) – verwendet, so daß diese Funktion beim Auftreten des Signals SIGALRM keinerlei Auswirkungen auf andere Signalhandler hat.

13.7 Anormale Beendigung mit Funktion **abort**

Der Aufruf der Funktion **abort** bewirkt eine anormale Programmbeendigung.

```
#include <stdlib.h>
```

```
void abort(void);
```

abort kehrt niemals zurück

Die Funktion **abort** schickt dem aufrufenden Prozeß das Signal SIGABRT. Dieses Signal sollte niemals von einem Prozeß ignoriert werden.

ANSI C schreibt vor, daß nach der Rückkehr aus einem eventuellen Signalhandler, der das Signal SIGABRT abgefangen hat, die Funktion **abort** niemals zum Aufrufer zurückkehrt. Das Abfangen des Signals SIGABRT wurde zugelassen, um dem Benutzer für den Fall einer anormalen Beendigung eines Prozesses noch Aufräumarbeiten (*cleanup*) durchführen zu lassen.

POSIX.1 legt zusätzlich fest, daß **abort** das Blockieren oder das Ignorieren des Signals SIGABRT durch einen Prozeß aufhebt.

Während ANSI C für die Funktion **abort** nicht vorschreibt, ob noch nicht geleerte Ausgabepuffer geleert und damit wirklich geschrieben werden oder ob temporäre Dateien automatisch gelöscht werden, legt POSIX.1 sehr wohl fest, daß bei einer Beendigung eines Prozesses durch **abort** alle noch offenen Standard-E/A-Streams mit **fclose** ordnungsgemäß zu schließen sind.

Laut POSIX.1 hat dagegen ein **abort**, das keine Beendigung eines Prozesses nach sich zieht, keinerlei Auswirkung auf offene E/A-Streams.

13.7.1 Mögliche Implementierung von abort

Das Programm 13.17 (abort.c) zeigt eine mögliche Implementierung von **abort** entsprechend den Anforderungen, die POSIX.1 an diese Funktion stellt.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include "eighdr.h"

void abort(void)
{
    struct sigaction  aktion;
    sigset_t          sigmaske;

    /*-- Falls Aufrufer das SIGABRT ignoriert, so wird SIG_DFL eingerichtet */
    sigaction(SIGABRT, NULL, &aktion);
    if (aktion.sa_handler == SIG_IGN) {
        aktion.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &aktion, NULL);
    }

    if (aktion.sa_handler == SIG_DFL)
        fflush(NULL); /* alle offenen Standard-E/A-Streams flushen */

    /*-- SIGABRT darf nicht blockiert sein ---> aus Signalmaske entfernen */
    sigfillset(&sigmaske);
    sigdelset(&sigmaske, SIGABRT);
    sigprocmask(SIG_SETMASK, &sigmaske, NULL);

    kill(getpid(), SIGABRT); /*-- Senden des Signals SIGABRT an Prozess */

    /*---- Hierhin gelangt man nur, wenn SIGABRT vom aufrufenden Prozess */
    /*---- abgefangen wurde, und der Signalhandler sich beendet hat      */

    fflush(NULL); /* alle offenen Standard E/A-Streams flushen */

    /*-- SIG_DFL wieder einstellen */
    aktion.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &aktion, NULL);
    sigprocmask(SIG_SETMASK, &sigmaske, NULL);

    kill(getpid(), SIGABRT); /*-- Erneutes Senden von SIGABRT an Prozess */

    exit(1); /*-- Dieses exit sollte niemals erreicht werden */
}
```

*Programm 13.17 (abort.c): Implementierung von **abort** entsprechend POSIX.1-Vorgabe*

Bei dieser Implementierung ist zu berücksichtigen, daß der aufrufende Prozeß für SIGABRT (wie für jedes Signal) drei mögliche Reaktionen auf dieses Signal festgelegt haben kann:

1. SIG_IGN

Da das Ignorieren des Signals SIGABRT nicht erlaubt ist, stellt die Funktion **abort** die Default-Aktion (SIG_DFL) ein.

2. SIG_DFL

Bei SIG_DFL und SIG_IGN (siehe Punkt 1) werden alle Standard-E/A-Puffer mit `fflush(NULL)` geleert und auf die entsprechenden Streams geschrieben. Hierbei ist zu beachten, daß **fflush** die entsprechenden Dateien nicht schließt. Dies geschieht erst dann, wenn der aufgerufene Prozeß sich beendet und dann das System automatisch die Dateien schließt.

3. Einen eigenen Signalhandler

Fängt ein Prozeß das Signal SIGABRT (erster **kill**-Aufruf) durch einen eigenen Signalhandler ab, dann kehrt er zur **abort**-Funktion zurück, wo hier nun mit **fflush** alle Standard-E/A-Puffer geleert und die darin enthaltenen Daten auf die entsprechenden Streams geschrieben werden, bevor für den Prozeß die Default-Signalbehandlung für SIGABRT eingerichtet wird. Dann wird dem Prozeß das Signal SIGABRT erneut geschickt, was durch die zwischenzeitliche Einrichtung der Default-Signalbehandlung zu seinem Abbruch führt.

13.8 Zusätzliche Argumente für Signalhandler

SVR4 und BSD-Unix bieten die Möglichkeit, Signalhandler mit mehr als einem Argument (die Signalnummer) aufzurufen.

13.8.1 Zusätzliche Argumente für Signalhandler in SVR4

Beim Aufruf von **sigaction**, kann man die Komponente `sa_flags` der Struktur **sigaction** auf den Wert `SA_SIGINFO` (siehe auch Tabelle 13.2) setzen. Dies bewirkt, daß der Signalhandler neben der Signalnummer als erstes Argument mit zwei zusätzlichen Argumenten aufgerufen wird, wobei hier nur das zweite Argument vorgestellt wird.

Das zweite Argument ist dabei ein NULL-Zeiger oder ein Zeiger auf eine `siginfo`-Struktur:

```
struct siginfo {
    int    si_signo; /* Signalnummer */
    int    si_errno; /* wenn ungleich 0: errno-Wert aus <errno.h> */
    int    si_code;  /* zusätzliche Info (vom System abhängig) */
    pid_t  si_pid;   /* PID des Sender-Prozesses */
    uid_t  si_uid;   /* reale User-ID des Sender-Prozesses */
    /* ... weitere Komponenten... */
}
```

Falls hierbei der Wert von `si_code` kleiner oder gleich 0 ist, so wurde das entsprechende Signal von einem Benutzerprozeß durch einen **kill**-Aufruf generiert. In diesem Fall enthalten die Komponenten `si_pid` und `si_uid` die Prozeß-ID und Benutzer-ID des Prozesses, der das Signal geschickt hat.

Handelt es sich beim geschickten Signal um `SIGFPE` (*floating point error*), so gibt der Wert von `si_code` mehr Information über den aufgetretenen Hardwarefehler. Hat `si_code` den Wert `FPE_INTDIV`, so ist eine Ganzzahldivision durch 0 aufgetreten, während der Wert `FPE_FLTDIV` auf eine Gleitpunktdivision durch 0 hinweist usw.

Mehr Information zur `siginfo`-Struktur findet sich in der SVR4-Manpage `siginfo(5)`.

13.8.2 Zusätzliche Argumente für Signalhandler in BSD

BSD-Unix ruft einen Signalhandler immer mit drei Argumenten auf:

```
sighandler(int signr, int code, struct sigcontext *sigconzgr);
```

Neben dem Argument `signr`, das die Signalnummer ist, stellt das Argument `code` für bestimmte Signale weitere Informationen zur Verfügung. Zum Beispiel zeigt der `code`-Wert `FPE_INTDIV_TRAP` beim Signal `SIGFPE` an, daß eine Ganzzahldivision durch 0 aufgetreten ist. Das 3. Argument `sigconzgr` ist hardwareabhängig.

13.9 Übung

13.9.1 Implementierung der Funktion `raise`

Geben Sie eine mögliche Implementierung für die Funktion `raise` an.

13.9.2 Nicht-lokaler Sprung unmittelbar nach `alarm`

In Programm 13.9 (`alarmrea2.c`) wurde eine Technik gezeigt, um für E/A-Operationen eine Zeitschaltuhr einzurichten. Oft wird für diese Aufgabenstellung auch folgender Codeausschnitt benutzt:

```
.....
signal(SIGALRM, sig_alarm);
alarm(60);
if (setjmp(progzust) != 0) {
    /*.... Reaktion auf Ablauf der Zeitschaltuhr ....*/
}
.....
```

Ist dieser Code absolut richtig oder birgt er etwa irgendetwelche Gefahren in sich?

13.9.3 Umständliche Beendigung bei der `abort`-Implementierung

Bei der Implementierung der `abort`-Funktion in Programm 13.17 (`abort.c`) wurde nach dem Senden des Signals `SIGABRT` (erster *kill*-Aufruf) dafür Sorge getragen, daß der aufrufende Prozeß eventuell dieses Signal abfängt und die Ausführung der `abort`-Funktion nach diesem ersten *kill*-Aufruf fortgesetzt wird. Warum wurde an dieser Stelle die erforder-

derliche Beendigung des Prozesses so umständlich umgesetzt (Einrichtung der Default-Aktion und erneutes Schicken des Signals mit **kill**)? Hätte hier nicht auch ein einfaches **_exit** ausgereicht?

13.9.4 Aufruf einer Nicht-Reentrant-Funktion im Signalhandler

Erstellen Sie ein Programm `nonreent.c`, das in einer Endlosschleife immer wieder eine Nicht-Reentrant-Funktion (wie z.B. **getpwnam**) aufruft. Zudem soll diese Nicht-Reentrant-Funktion in einem Signalhandler aufgerufen werden. Dieser Signalhandler soll jede Sekunde (`alarm(1)`) aktiviert werden. Starten Sie dann dieses Programm `nonreent.c` und versuchen Sie das Ablaufgeschehen zu erklären.

13.9.5 Implementierung der Signalmengenfunktionen

Erstellen Sie ein Programm `sigmenge.c`, das mögliche Implementierungen zu den Funktionen **sigemptyset**, **sigfillset**, **sigaddset**, **sigdelset** und **sigismember** enthält.

Bei dieser Implementierung soll angenommen werden, daß nicht mehr Signale vorhanden sind, als der **int**- bzw. der **long**-Datentyp an Bits zur Verfügung hat. So kann dann eine Signalmenge (Datentyp `sigset_t`) durch diesen Datentyp realisiert werden, wobei jeweils ein Bit immer ein Signal repräsentiert. Dies entspricht im übrigen auch den meisten Systemen.

13.9.6 Implementierung der Funktion `system` mit Signalhandler

Im Programm 10.19 (`system.c`) des Kapitels 10.6 wurde eine mögliche Implementierung der Funktion **system** gezeigt. Diese Implementierung fing jedoch keinerlei Signale ab. POSIX.2 verlangt aber, daß die Funktion **system** die beiden Signale `SIGINT` und `SIGQUIT` ignoriert und das Signal `SIGCHLD` blockiert.

Die Gründe für diese Vorschrift sind, daß ein mit **system** gestarteter Prozeß die volle Kontrolle über eventuell ankommende Signale haben sollte.

Wird z.B. während der Ausführung von **system** eines der beiden Signale `SIGINT` oder `SIGQUIT` geschickt, so sollte dieses Signal nur dem gerade ausführenden Prozeß und nicht dem **system**-Aufrufer geschickt werden. Dies ist der Grund, warum für den **system**-Aufrufer die beiden Signale `SIGINT` und `SIGQUIT` (in **system**) ignoriert werden sollten.

Das Signal `SIGCHLD` andererseits sollte von der Funktion **system** blockiert werden, da der durch **system** kreierte Kindprozeß nicht explizit vom **system**-Aufrufer, sondern implizit in der Funktion **system** kreiert wurde. Um zu verhindern, daß das Signal `SIGCHLD` dem **system**-Aufrufer geschickt wird, was diesen irrtümlicherweise denken läßt, daß einer seiner eigenen explizit kreierte Kindprozesse sich beendet hat, sollte in der Funktion **system** (für den Aufrufer) das Signal `SIGCHLD` blockiert werden.

Erstellen Sie ein Programm `system2.c`, das das Programm 10.19 (`system.c`) dahingehend erweitert, daß die von POSIX.2 vorgegebenen Vorschriften (Ignorieren von `SIGINT` und `SIGQUIT`, Blockieren von `SIGCHLD`) eingehalten werden.

13.9.7 Warten auf das Ende aller Kindprozesse (Signal SIGCHLD)

Erstellen Sie ein Programm `sigkind.c`, das x Kindprozesse kreiert. Die Anzahl x der Kindprozesse soll dabei auf der Kommandozeile angegeben werden. Bei jedem Start eines Kindprozesses soll der Elternprozeß eine globale Variable n um 1 hochzählen. Bei Beendigung eines Kindprozesses, was dem Elternprozeß mit dem Signal `SIGCHLD` mitgeteilt wird, soll dieser in einem explizit hierfür eingerichteten Signalhandler den Status des gerade beendeten Kindprozesses erfragen und die Variable n wieder um 1 dekrementieren. Wenn $n == 0$ wird, soll der Elternprozeß sich beenden.

Nachdem man dieses Programm `sigkind.c` kompiliert und gelinkt hat

```
cc -o sigkind sigkind.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ sigkind 5
--- Kind 2482 startet (n=1)
--- Kind 2483 startet (n=2)
--- Kind 2484 startet (n=3)
--- Kind 2485 startet (n=4)
--- Kind 2486 startet (n=5)
..... Hauptprogramm ..... (5 Sek. schlafen)
--- Kind 2482 beendet (noch n=4 Kinder)
..... Hauptprogramm ..... (wieder aufgewacht)
--- Kind 2483 beendet (noch n=3 Kinder)
--- Kind 2484 beendet (noch n=2 Kinder)
--- Kind 2485 beendet (noch n=1 Kinder)
--- Kind 2486 beendet (noch n=0 Kinder)
..... Hauptprogramm beendet sich .....
$
```

13.9.8 Kindprozeß nur für gewisse Zeit ausführen lassen

Erstellen Sie ein Programm `sigkind2.c`, das einen Kindprozeß kreiert und anschließend auf das Signal `SIGCHLD` wartet.

Wenn der Kindprozeß sich nicht innerhalb einer Wartezeit von 10 Sekunden beendet (durch Schicken von `SIGCHLD` angezeigt), so soll der Elternprozeß ihn mit dem Signal `SIGTERM` gewaltsam beenden.

Falls der Elternprozeß aber innerhalb von 10 Sekunden das Signal `SIGCHLD` empfängt, so soll er, wenn der Kindprozeß nur angehalten wurde, ihn gewaltsam durch das Schicken des Signals `SIGKILL` beenden. Andernfalls soll der Elternprozeß den Beendigungsstatus des Kindprozesses auswerten und ausgeben.

14 STREAMS in System V

*Oder ob ein Knopf der Hose
Abgerissen oder lose -
Wie und wo und wann es sei,
Hinten, vorne, einerlei -
Alles machte Meister Böck,
Denn das ist sein Lebenszweck.*

Wilhelm Busch

STREAMS werden von SVR4 vollständig unterstützt und sind dort die allgemeine Schnittstelle zu Kommunikationstreibern. Das Verständnis von STREAMS ist wichtig, um die Terminalschnittstelle in SVR4 zu verstehen. Zudem werden STREAMS benötigt, um die im nächsten Kapitel beschriebene Funktion ***poll***, die Implementierung von Stream Pipes in Kapitel 19.2 und die Terminalschnittstelle von SVR4 (in Kapitel 20) zu verstehen.

14.1 Allgemeines zu STREAMS

STREAMS wurden 1984 von Dennis Ritchie als Erweiterung zum traditionellen E/A-System und zur Anpassung an Netzwerkprotokolle entwickelt. Seit SVR4 werden STREAMS vollständig unterstützt.

Ein STREAM stellt eine *Vollduplex*-Verbindung zwischen einem Benutzerprozeß und einem Gerätetreiber zur Verfügung. Ein STREAM muß nicht direkt mit dem aktuellen physikalischen Gerät kommunizieren, sondern kann auch für Pseudoterminalgerätetreiber verwendet werden (siehe auch Kapitel 20).

Abbildung 14.1 zeigt das grundsätzliche Aussehen eines sogenannten einfachen STREAMS.

Unterhalb des *STREAM-Kopfes* kann man *Steuerungsmodule* eintragen, über die die Kommunikation zwischen STREAM-Kopf und Gerätetreiber stattfindet. Das Eintragen eines solchen Moduls erfolgt mit der Funktion ***ioctl*** (siehe Kapitel 14.2).

Abbildung 14.2 zeigt einen STREAM mit einem solchen Steuermodul. Die Vollduplex-Eigenschaft wird dort durch die zwei eingehenden und ausgehenden Pfeile hervorgehoben.

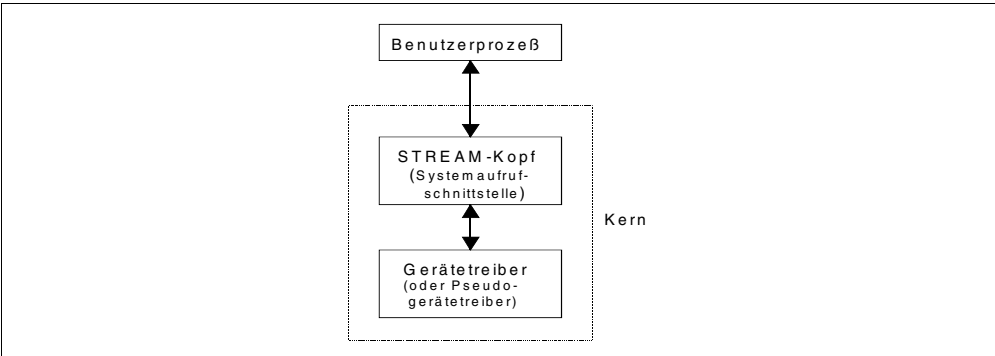


Abbildung 14.1: Ein einfacher STREAM

In einem STREAM können beliebig viele Steuermodule eingetragen werden, wobei jedes neue Modul nach dem Stackprinzip (LIFO) unter dem STREAM-Kopf eingeordnet wird, und somit die bereits vorhandenen Module weiter nach unten verschoben werden. In Abbildung 14.2 ist zusätzlich die Richtung (abwärts oder aufwärts) angegeben. Während Daten, die in einen STREAM-Kopf geschrieben werden, abwärts geschickt werden, werden die vom Gerätetreiber gelesenen Daten aufwärts geschickt.

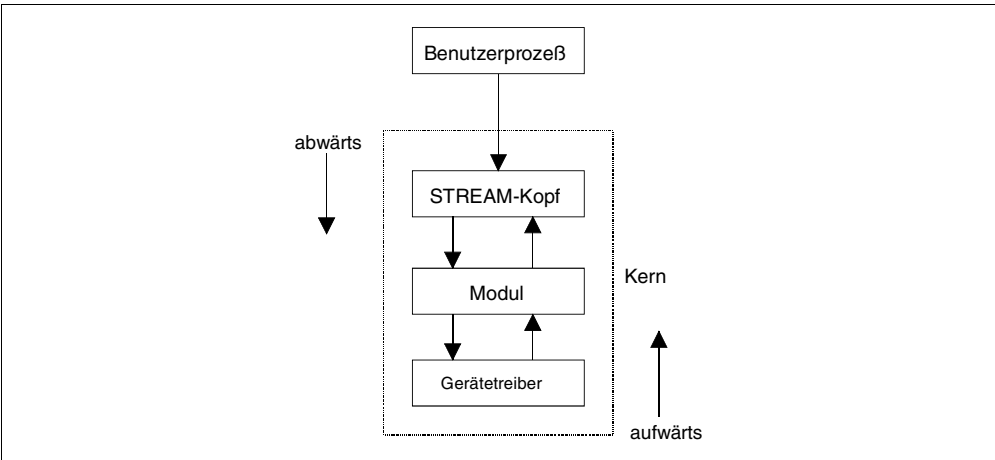


Abbildung 14.2: Ein STREAM mit einem Steuermodul

STREAM-Module werden normalerweise beim Generieren des Kerns in den Kern gelinkt. Die meisten Systeme erlauben deshalb auch nur die Eintragung von bereits im Kern vorhandenen Modulen in einen STREAM. Die Eintragung anderer Module ist dort nicht möglich. Abbildung 14.3 zeigt ein auf einem STREAM basierendes Terminalsystem.

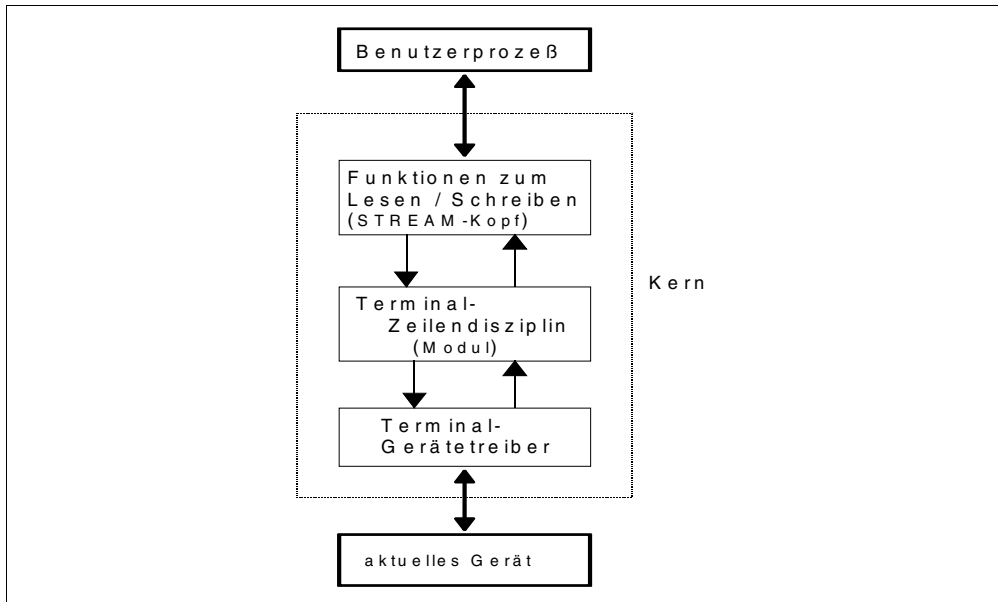


Abbildung 14.3: Auf einem STREAM basierendes Terminalsystem

Der Zugriff auf einen STREAM erfolgt mit den folgenden Funktionen:

- **open, close, read, write** (siehe Kapitel 4)
- **ioctl, getmsg, putmsg, poll, getpmsg, putpmsg** (werden später in diesem Kapitel beschrieben)

Öffnet man einen STREAM, so wird der dabei angegebene Pfadname im Directory `/dev` als zeichenorientierte Gerätedatei angelegt.

Hinweis

STREAMS dürfen nicht mit dem in Kapitel 3.3 erwähnten *Streams* der Standard-E/A-Funktionen verwechselt werden.

14.2 STREAM-Messages

Vor der Einführung von STREAMS mußte beim Hinzufügen eines neuen zeichenorientierten Geräts ein neuer Gerätetreiber für dieses Gerät geschrieben werden. Jeder spätere Zugriff auf das neue Gerät mittels **read** oder **write** bedeutete einen direkten Zugriff auf den Gerätetreiber. Mit dem neuen STREAMS-Konzept ist es nun möglich, zwischen STREAM-Kopf und Gerätetreiber beliebig viele Steuermodule einzutragen, die die entsprechenden Operationen an den zwischen STREAM-Kopf und Gerätetreiber fließenden Daten vornehmen.

Jede Ein- und Ausgabe erfolgt bei STREAMS über sogenannte *Messages* (Nachrichten oder Botschaften). Der STREAM-Kopf und ein Benutzerprozeß tauschen unter Verwendung von ***read***, ***write***, ***ioctl***, ***getmsg***, ***putmsg***, ***getpmsg*** und ***putpmsg*** untereinander Nachrichten aus.

Diese Nachrichten werden im STREAM entsprechend abwärts oder aufwärts weitergeleitet (siehe auch Abbildung 14.2).

Zwischen dem Benutzerprozeß und dem STREAM-Kopf besteht eine *Message* aus den folgenden Komponenten

1. *Message*-Typ (siehe auch Tabelle 14.1)
2. optionale Kontrollinformation
3. optionale Daten

14.2.1 Daten und Kontrollinformationen

Der Inhalt der Kontrollinformation und der Daten ist über die Struktur `strbuf` festgelegt.

```
struct strbuf {
    int  maxlen; /* Puffer-Größe */
    int  len;    /* Momentane Anzahl der Bytes im Puffer */
    char *buf;   /* Puffer-Adresse */
};
```

Wenn eine Message mit ***putmsg*** oder ***putpmsg*** geschickt wird, so gibt `len` die Anzahl der Datenbytes im Puffer an. Empfängt man eine Message mit ***getmsg*** oder ***getpmsg***, so gibt `maxlen` die Puffer-Größe an, und `len` wird vom Kern auf die Anzahl der im Puffer gespeicherten Daten gesetzt. Später werden wir sehen, daß `len == 0` auf eine leere Message hinweist und bei `len == -1` keinerlei Kontrollinformation bzw. Daten vorhanden sind.

Die Komponente *Kontrollinformation* wird z.B. für Anwendungen benötigt, die eine verbindungslose Netzwerknachricht (*datagram*) schicken. Um sie zu schicken, muß neben den eigentlichen Daten die Zieladresse angegeben werden, die als Kontrollinformation mitgegeben wird.

14.2.2 Message-Typen

Es gibt über 25 verschiedene *Message*-Typen, von denen aber nur wenige zwischen Benutzerprozeß und STREAM-Kopf benutzt werden. Der Rest wird vom Kern beim Weiterleiten der Message (auf- und abwärts) benutzt. Diese restlichen Typen sind nur für Personen von Interesse, die Steuermodule schreiben.

Die drei wichtigsten Message-Typen sind

- `M_DATA` (Benutzerdaten für E/A)
- `M_PROTO` (Protokoll-Kontrollinformation)
- `M_PCPROTO` (Protokoll-Kontrollinformation mit hoher Priorität)

14.2.3 Message-Prioritäten

Jede Message in einem STREAM hat eine Warteschlangenpriorität.

- ▶ *hochpriore* Messages (höchste Priorität) Prioritätswert: >255
- ▶ Messages unterschiedlicher Priorität; Prioritätswert: 1-255
- ▶ *normale* Messages (niedrigste Priorität); Prioritätswert: 0

Jedes STREAM-Modul hat zwei Eingabewarteschlangen: Eine nimmt Messages vom darüberliegenden Modul (abwärts laufende Messages) und die andere Messages vom darunterliegenden Modul (aufwärts laufende Messages) auf. In der jeweiligen Eingabewarteschlange werden die Messages entsprechend ihrer Priorität angeordnet.

Tabelle 14.1 zeigt, welche Argumente für **write**, **putmsg** und **putpmsg** die Messages unterschiedlicher Priorität generieren.

14.2.4 putmsg und putpmsg – Schicken einer Message an einen STREAM

Um eine Message (Kontrollinformation oder Daten oder beides) an einen STREAM zu schicken, stehen die beiden Funktionen **putmsg** und **putpmsg** zur Verfügung.

```
#include <stropts.h>

int putmsg(int fd, const struct strbuf *ktrlzgr,
           const struct strbuf *datzgr, int flag);

int putpmsg(int fd, const struct strbuf *ktrlzgr,
            const struct strbuf *datzgr, int band, int flag);

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler
```

Bei **putpmsg** kann im Unterschied zu **putmsg** der Prioritätswert (*band*) für die Message festgelegt werden kann.

Das Senden einer Message mittels **write** ist ebenso möglich. Dies entspricht einem **putmsg** ohne jegliche Kontrollinformation (*flag* == 0).

Die Funktionen **putmsg** und **putpmsg** können oben genannte Arten von Messages generieren: normale, band-Prioritäts- und hochpriore Messages. Welche Art generiert wird, hängt von den Argumenten beim Aufruf einer der beiden Funktionen ab. Tabelle 14.1 zeigt alle möglichen Argumentkombinationen und die daraus resultierenden Message-Arten.

Funktion	Kontrollinfo	Daten	band	flag	generierte Message-Art
write	-	1	-	-	M_DATA (normal)
putmsg	1	0	-	0	keine Message wird geschickt; Rückgabewert 0
putmsg	0	1	-	0	M_DATA (normal)
putmsg	1	1 oder 0	-	0	M_PROTO (normal)
putmsg	1	1 oder 0	-	RS_HIPRI	M_PCPROTO (hoch-prior)
putmsg	0	1 oder 0	-	RS_HIPRI	Fehler EINVAL
putpmsg	1 oder 0	1 oder 0	0-255	0	Fehler EINVAL
putpmsg	0	0	0-255	MSG_BAND	keine Message wird geschickt; Rückgabewert 0
putpmsg	0	1	0	MSG_BAND	M_DATA (normal)
putpmsg	0	1	1-255	MSG_BAND	M_DATA (band-Priorität)
putpmsg	1	1 oder 0	0	MSG_BAND	M_PROTO (normal)
putpmsg	1	1 oder 0	1-255	MSG_BAND	M_PROTO (band-Priorität)
putpmsg	1	1 oder 0	0	MSG_HIPRI	M_PCPROTO (hoch-prior)
putpmsg	0	1 oder 0	0	MSG_HIPRI	Fehler EINVAL
putpmsg	1 oder 0	1 oder 0	!=0	MSG_HIPRI	Fehler EINVAL

Tabelle 14.1: Von **write**, **putmsg** und **putpmsg** generierte Message-Arten

Die einzelnen Bezeichnungen in Tabelle 14.1 haben folgende Bedeutung:

-	nicht möglich	
0	für Kontrollinfo:	ktrlzgr == NULL oder ktrlzgr->len == -1
	für Daten:	datzgr == NULL oder datzgr->len == -1
1	für Kontrollinfo:	ktrlzgr != NULL und ktrlzgr->len >= 0
	für Daten:	datzgr != NULL und datzgr->len >= 0

14.2.5 getmsg und getpmsg – Lesen einer Message aus einem STREAM

Um eine Message aus einem STREAM zu lesen, stehen die beiden Funktionen **getmsg** und **getpmsg** zur Verfügung.


```
#include <stropts.h>

int getmsg(int fd, struct strbuf *ktrlzgr,
           struct strbuf *datzgr, int *flagzgr);

int getpmsg(int fd, struct strbuf *ktrlzgr,
            struct strbuf *datzgr, int *bandzgr, int *flagzgr);

           beide geben zurück: nichtnegativen Wert (bei Erfolg); -1 bei Fehler
```

Um festzulegen, welche Art von Message zu lesen ist, müssen vor dem Aufruf an die Adressen, auf die *flagzgr* und *bandzgr* zeigen, die entsprechenden Werte geschrieben werden. Bei der Rückkehr aus der entsprechenden Funktion steht dort dann die Art der gelesenen Message.

Falls **flagzgr* == 0 ist, liefert **getmsg** die nächste Message aus der Lesewarteschlange des STREAM-Kopfes. Falls es sich dabei um ein hochpriorie Message handelt, dann schreibt **getmsg** an die Adresse *flagzgr* den Wert RS_HIPRI. Sollen nur hoch-priorie Messages gelesen werden, so muß beim Aufruf von **getmsg** **flagzgr* == RS_HIPRI sein.

Für **getpmsg** müssen andere Konstanten als für **getmsg** benutzt werden. Zusätzlich kann bei **getpmsg** über *bandzgr* eine bestimmte Bandpriorität spezifiziert werden.

Welche Art von Message dem Aufrufer durch eine von diesen beiden Funktionen zurückgegeben wird, hängt von vielen Faktoren ab:

1. Werte an den Adressen *flagzgr* und *bandzgr*.
2. Message-Arten, die sich in der STREAMS-Warteschlange befinden.
3. *ktrlzgr* und *datzgr* ungleich NULL.
4. Werte von *ktrlzgr*->*maxlen* und *datzgr*->*maxlen*.

Nähere Details hierzu finden sich in der Manpage zu **getmsg(2)**.

Beispiel

Demonstrationsprogramm zu **getmsg**

Das folgende Programm 14.1 (*streamcp.c*) demonstriert die Anwendung von **getmsg** anhand des Kopierens der Standardeingabe auf die Standardausgabe.

```
#include <stropts.h>
#include "eighdr.h"

#define PUFFGROESSE 8192

int
main(void)
{
    int                n, flag;
```

```

char          ktrlpuffer[PUFFGROESSE], datpuffer[PUFFGROESSE];
struct strbuf  ktrl, dat;

ktrl.buf      = ktrlpuffer;
ktrl.maxlen   = PUFFGROESSE;
dat.buf       = datpuffer;
dat.maxlen    = PUFFGROESSE;

while (1) {
    flag = 0;
    if ( (n = getmsg(STDIN_FILENO, &ktrl, &dat, &flag)) < 0)
        fehler_meld(FATAL_SYS, "getmsg-Fehler");
    fprintf(stderr, "--- flag=%d, ktrl.len=%d, dat.len=%d---\n",
              flag, ktrl.len, dat.len);
    if (dat.len > 0) {
        if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
            fehler_meld(FATAL_SYS, "write-Fehler");
    } else
        exit(0);
}
}

```

Programm 14.1 (**streamcp.c**): Kopieren der Standardeingabe auf Standardausgabe mit **getmsg**

Nachdem man Programm 14.1 (**streamcp.c**) kompiliert und gelinkt hat

```
cc -o streamcp streamcp.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ echo Pipetest | streamcp      [erfordert Pipe-Implementierung mit STREAMS]
--- flag=0, ktrl.len=-1, dat.len=9---
Pipetest
--- flag=0, ktrl.len=0, dat.len=0--- [zeigt einen STREAMS-hangup an]
$ streamcp < /etc/passwd
getmsg-Fehler: Not a stream device
$ streamcp      [erfordert, dass Terminals mit STREAMS implementiert sind]
eine einfache Eingabe
--- flag=0, ktrl.len=-1, dat.len=21---
eine einfache Eingabe
und noch ne Eingabe
--- flag=0, ktrl.len=-1, dat.len=19---
und noch ne Eingabe
Ctrl-D      [Eingabe von EOF]
--- flag=0, ktrl.len=-1, dat.len=0--- [EOF ist nicht dasselbe wie ein hangup]
$

```

Wenn die Pipe geschlossen wird, so entspricht dies einem *STREAMS-hangup* (Kontrollinfolänge und Datenlänge sind beide 0). Bei einem Terminal jedoch entspricht die Eingabe von EOF (*Strg-D*) nicht einem *hangup*, da hierbei nur die Datenlänge auf 0 gesetzt wird, während die Kontrollinfolänge -1 bleibt.

14.2.6 ioctl – Ausführen der unterschiedlichsten Operationen auf STREAMS

Die Funktion **ioctl** ist eine Art von Lückenbüßer für alle Arten von E/A-Operationen, für die keine eigene Funktion vorgesehen ist. Der Hauptanwender für **ioctl** war früher die Terminal-Ein-/Ausgabe, bis POSIX.1 hierfür eigene neue Funktionen zur Verfügung gestellt hat (siehe Kapitel 20). Heute sind die Operationen auf STREAMS eine der Hauptanwendungen für **ioctl**.

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* BSD */
```

```
int ioctl(int fd, int operation, ...);
```

gibt zurück: -1 (bei Fehler); anderer Wert sonst

Unter SVR4 gibt es fast 30 verschiedene Operationen, die man mit **ioctl** auf einen STREAM durchführen kann. Diese Operationen sind in der Manpage `streamio(7)` dokumentiert. Um STREAM-Operationen mit **ioctl** durchzuführen, muß

```
#include <stropts.h>
```

angegeben werden. Das Argument `operation` legt die durchzuführende Operation fest. Hierfür wird üblicherweise eine Konstante angegeben, die mit `I_` beginnt. Das 3. Argument ist von der `operation`-Angabe abhängig; Entweder eine ganze Zahl oder ein Zeiger auf eine ganze Zahl oder auf eine Struktur.

Hinweis

Die Funktion **ioctl** ist nicht Bestandteil von POSIX.1, wird aber sowohl von SVR4 und BSD-Unix angeboten.

Im obigen Prototyp wurden nur die Headerdateien angegeben, die für die Funktion **ioctl** selbst benötigt werden. Normalerweise benötigt man abhängig vom E/A-Gerät, auf das man die **ioctl**-Operation anwenden will, weitere Headerdateien. Bei Terminal-E/A benötigt man z.B. zusätzlich die Headerdatei `<termios.h>`.

Die Tabelle 14.2 zeigt weitere Headerdateien und definierte Konstanten (Anfangsbuchstaben) für die Verwendung von **ioctl** in BSD.

E/A auf	Konstanten (Anfangsbuchst.)	Headerdatei
Terminal	TIO....	<ioctl.h>
Datei	FIO...	<ioctl.h>
Magnetband	MTIO....	<mtio.h>
Socket	SIO...	<ioctl.h>

Tabelle 14.2: Weitere **ioctl**-Operationen in BSD-Unix

Welche und wie viele Operationen bei den in Tabelle 14.2 angegebenen Ein-/Ausgaben für **ioctl** zur Verfügung stehen, ist von der jeweiligen Kategorie abhängig. So werden z.B. für ein Magnetband Operationen wie Zurücksputzen, Vorwärtsspulen um eine bestimmte Anzahl von Dateien oder Einträgen usw. angeboten.

14.2.7 isastream – Überprüfen, ob Filedeskriptor ein STREAM ist

Um festzustellen, ob ein Filedeskriptor ein STREAM ist oder nicht, stellt SVR4 die Funktion **isastream** zur Verfügung.

```
int isastream(int fd);
```

gibt zurück: 1 (wenn *fd* ein STREAM ist); 0 sonst

Beispiel

Demonstrationsprogramm zur Funktion isastream

```
#include <sys/types.h>
#include <sys/fcntl.h>
#include <unistd.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int i, fd;

    for (i=1; i<argc; i++) {
        if ( (fd = open(argv[i], O_RDONLY)) < 0)
            fehler_meld(WARNUNG_SYS, "...%s: kann nicht oeffnen", argv[i]);
        else if (isastream(fd))
            fprintf(stderr, "%s: STREAM\n", argv[i]);
        else
            fprintf(stderr, "...%s: kein STREAM\n", argv[i]);
        close(fd);
    }
    exit(0);
}
```

Programm 14.2 (isstream.c): Demonstrationsbeispiel zur Funktion isastream

Nachdem man das Programm 14.2 (isstream.c) kompiliert und gelinkt hat

```
cc -o isstream isstream.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ isstream /dev/stdin /etc/passwd /dev/null
/dev/stdin: STREAM
.../etc/passwd: kein STREAM
.../dev/null: kein STREAM
```

```
$ isstream /dev/nichts /dev/tty /dev/fd0
.../dev/nichts: kann nicht oeffnen: No such file or directory
/dev/tty: STREAM
.../dev/fd0: kein STREAM
$
```

Wie wir an den beiden obigen Programmabläufen erkennen können, sind `/dev/stdin` und `/dev/tty` STREAMS.

Beispiel

Mögliche Implementierung der Funktion `isastream`

```
#include <stropts.h>
#include <unistd.h>

int isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

Programm 14.3 (`isastream.c`): Mögliche Implementierung der Funktion `isastream`

Im Programm 14.3 (`isastream.c`) wurde `I_CANPUT` beim `ioctl`-Aufruf angegeben, um zu prüfen, ob band 0 (3.Argument) beschreibbar ist.

14.2.8 Ausgeben der Steuermodule eines STREAMS

Um alle Steuermodule eines STREAMS zu erhalten, muß beim Aufruf von `ioctl` als Argument für Operation `I_LIST` angegeben werden. In diesem Fall muß das dritte Argument ein Zeiger auf die Struktur `str_list` sein:

```
struct str_list {
    int          sl_nmods;    /* Anzahl der Array-Einträge */
    struct str_mlist *sl_modlist; /* Zgr. auf 1.Element des Arrays */
}
```

Die Struktur `str_mlist` besteht lediglich aus einer Komponente

```
struct str_mlist {
    char l_name[FMNAMESZ +1]; /* Modulname + abschließendes \0 */
}
```

Die Konstante `FMNAMESZ` ist in der Headerdatei `<sys/conf.h>` definiert (meist 8).

Vor dem Aufruf von `ioctl` muß die Komponente `sl_modlist` der Struktur `str_list` auf die Adresse des ersten Elements eines Arrays gesetzt werden, dessen Elemente Strukturen des Datentyps `str_mlist` sind. `sl_nmods` muß in diesem Fall auf die Anzahl der Elemente dieses Arrays gesetzt werden.

Falls für das dritte Argument bei einem `ioctl`-Aufruf der Wert 0 angegeben wird, so gibt `ioctl` die Anzahl der Steuermodule und nicht die Modulnamen zurück. Üblicherweise

ruft man **ioctl** zunächst auf diese Art auf (3. Argument == 0), um vorab die Anzahl der im STREAM vorhandenen Module zu ermitteln. Kennt man diese Anzahl, so kann man den benötigten Speicherplatz (Anzahl von `str_mlist`-Strukturen) allokieren, bevor man die Modulnamen mit dem nächsten **ioctl**-Aufruf erfragt.

Programm 14.4 (`streamod.c`) ermittelt alle Module zu dem auf der Kommandozeile angegebenen STREAM und gibt diese aus.

```
#include <sys/conf.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stropts.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int          fd, i, modzahl;
    struct str_list liste;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s dateiname", argv[0]);

    if ( (fd = open(argv[1], O_RDONLY)) < 0)
        fehler_meld(FATAL_SYS, "kann Datei %s nicht oeffnen", argv[1]);

    if (!isastream(fd))
        fehler_meld(FATAL, "%s ist kein STREAM", argv[1]);

    /*--- Anzahl der Module erfragen -----*/
    if ( (modzahl = ioctl(fd, I_LIST, NULL)) < 0)
        fehler_meld(FATAL_SYS, "ioctl-Fehler");
    printf("--- %d Module ---\n", modzahl);

    /*--- Speicherplatz fuer die Modulnamen allokieren -----*/
    if ( (liste.sl_modlist = calloc(modzahl, sizeof(struct str_mlist))) == NULL)
        fehler_meld(FATAL_SYS, "calloc-Fehler");
    liste.sl_nmods = modzahl;

    /*--- Modulnamen erfragen -----*/
    if (ioctl(fd, I_LIST, &liste) < 0)
        fehler_meld(FATAL_SYS, "ioctl-Fehler");

    /*--- Modulnamen ausgeben -----*/
    for (i=1; i<=modzahl; i++)
        printf("%15s: %s\n", (i==modzahl) ? "Treiber" : "Modul",
               liste.sl_modlist++);

    exit(0);
}
```

Programm 14.4 (**streamod.c**): Ausgeben der Modulnamen zu einem STREAM

Programm 14.4 (`streamod.c`) verwendet zum Erfragen der Module die Operation `I_LIST` beim ***ioctl***-Aufruf.

Nachdem man dieses Programm 14.4 (`streamod.c`) kompiliert und gelinkt hat

```
cc -o streamod streamod.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```
$ streamod /dev/tty
--- 6 Module ---
    Modul: ttcompat
    Modul: ldterm
    Modul: emap
    Modul: ansi
    Modul: char
    Treiber: cmux
$ streamod /dev/null
/dev/null ist kein STREAM
$
```

14.2.9 Schreibmodus für STREAMS

Es ist zu unterscheiden, ob man mit ***ioctl*** den »Schreibmodus« für einen STREAM erfragen oder setzen will:

1. Wird für das `operation`-Argument `I_GWROPT` angegeben, so muß als drittes Argument ein ***int***-Zeiger angegeben werden. An diese Adresse wird von ***ioctl*** der momentan für den STREAM eingestellte Schreibmodus geschrieben.
2. Wird für das `operation`-Argument `I_SWROPT` angegeben, so muß als drittes Argument der neu einzustellende Schreibmodus in Form einer ganzen Zahl angegeben wird.

Um einen Schreibmodus für einen STREAM neu zu setzen, sollte man zuerst den momentan eingestellten Schreibmodus erfragen, diesen modifizieren und dann den modifizierten als neuen Schreibmodus setzen. Ein direktes absolutes Setzen des Schreibmodus ist nicht anzuraten, da dies eventuell zum unbeabsichtigten Ausschalten von vorher gesetzten Bits (Eigenschaften) führen kann.

Momentan sind nur zwei Werte für den Schreibmodus definiert:

SNDZERO

Ein ***write*** von 0 Bytes in eine Pipe oder eine FIFO bewirkt, daß eine Message der Länge 0 STREAM-abwärts geschickt wird. Voreinstellung ist, daß bei einem solchen ***write*** von 0 Bytes keinerlei Message geschickt wird.

SNDPIPE

Bewirkt, daß das Signal `SIGPIPE` dem Prozeß, der entweder ***write*** oder ***putmsg*** aufgerufen hat, geschickt wird, wenn ein Fehler im STREAM auftritt.

14.2.10 Lesemodus für STREAMS

Es ist zu unterscheiden, ob man mit **ioctl** den Lesemodus für einen STREAM erfragen oder setzen möchte:

1. Wird für das *operation*-Argument **I_GRDOPT** angegeben, so muß als drittes Argument ein **int**-Zeiger angegeben werden. An diese Adresse wird dann von **ioctl** der momentan für den STREAM eingestellte Lesemodus geschrieben.
2. Wird für das *operation*-Argument **I_SRDOPT** angegeben, so muß als drittes Argument der neu einzustellende Lesemodus in Form einer ganzen Zahl angegeben werden.

Die verschiedenen Arten von Lesemodi sind durch die folgenden drei Konstanten spezifiziert:

RNORM

Normaler byteweiser Modus: In diesem Modus liest **read** die Daten Byte für Byte aus dem STREAM, bis die geforderte Anzahl von Bytes gelesen wurde oder aber keine weiteren Daten mehr vorhanden sind. Dieser Modus ist die Voreinstellung.

RMSGN

Nondiscard-Modus: In diesem Modus liest **read** die geforderte Anzahl von Bytes oder aber bis zum Message-Ende. Wenn mit diesem **read** nicht alle Daten der Message gelesen wurden, so verbleiben die restlichen Daten im STREAM, damit das nächste **read** dort weiterlesen kann.

RMSGD

Discard-Modus: Dieser Modus unterscheidet sich vom *Nondiscard-Modus* darin, daß bei einem **read**, das nicht alle Daten einer Message liest, die restlichen Daten nicht im STREAM verbleiben, sondern weggeworfen werden.

Zusätzlich können die folgenden drei Konstanten benutzt werden, um Einfluß auf das Verhalten eines **read** im Lesemodus zu nehmen, wenn es Messages vorfindet, die Protokollinformation enthalten:

RPROTNORM

Protokoll-Normal-Modus: **read** liefert Fehler EBADMSG. Dies ist die Voreinstellung.

RPROTDAT

Protokoll-Daten-Modus: **read** liefert den Kontrollteil als Daten.

RPROTDIS

Protokoll-Discard-Modus: **read** ignoriert die Kontrollinformation, liefert aber eventuelle Daten der entsprechenden Message.

14.3 Übung

14.3.1 Anzahl der verschiedenen Arten von Informationen bei **getmsg**

Wie viele verschiedene Arten von Informationen kann **getmsg** zurückliefern?

15 Fortgeschrittene Ein- und Ausgabe

*Mancher gibt sich viele Müh'
Mit dem lieben Federvieh;
Einesteils der Eier wegen,
Welche diese Vögel legen,
Zweitens: Weil man dann und wann
Einen Braten essen kann;
Drittens aber nimmt man auch
Ihre Federn in Gebrauch.*

Wilhelm Busch

Dieses Kapitel beschäftigt sich mit den folgenden Formen der Ein- und Ausgabe: *E/A-Multiplexing, asynchrone E/A, gleichzeitiges Lesen und Schreiben aus mehreren nicht zusammenhängenden Puffer* und dem sogenannten *Memory Mapped I/O*. Alle diese Formen der Ein- und Ausgabe sind Voraussetzung zum Verständnis der Kapitel 17, 18 und 19, die sich mit Interprozeßkommunikation beschäftigen.

15.1 E/A-Multiplexing

Wenn man von einem Filedeskriptor liest und auf einen anderen schreibt, verwendet man meist den folgenden Code:

```
while ( (n = read(lese_fd, puffer, BUFSIZ)) > 0)
    if (write(schreib_fd, puffer, n) != n)
        fehler_meld("write-Fehler");
```

Diese Form der abwechselnd blockierenden Ein- und Ausgabe kann man jedoch nicht verwenden, wenn man gleichzeitig von zwei Filedeskriptoren lesen muß. In diesem Fall darf man nämlich kein blockierendes **read** für einen der beiden Filedeskriptoren durchführen, da Daten dann eventuell im anderen Filedeskriptor ankommen, während man mit **read** auf dem einen Filedeskriptor blockiert ist. Für Anwendungen dieser Art benötigt man andere Vorgehensweisen oder Funktionen.

Zum Diskutieren dieser unterschiedlichen Techniken und ihrer Schwächen wollen wir als Beispiel ein *Modem-Terminalkommunikationsprogramm* (MTK) heranziehen. Ein solches MTK-Programm muß vom Terminal lesen und auf das Modem schreiben, während es gleichzeitig auch vom Modem lesen und auf das Terminal schreiben muß. Abbildung 15.1 verdeutlicht dies.

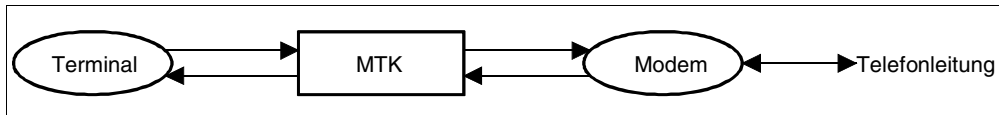


Abbildung 15.1: Modem-Terminalkommunikation

Der Prozeß MTK hat also zwei Ein- und zwei Ausgaben. Ein blockierendes **read** auf eine der beiden Eingabekanäle ist dabei nicht möglich, da nicht vorhersagbar ist, in welchem Kanal als nächstes Daten zur Bearbeitung anstehen werden. Nachfolgend wollen wir unterschiedliche Lösungsansätze für dieses Problem diskutieren.

15.1.1 Aufteilen der Kommunikation auf mehrere Prozesse

Ein möglicher Lösungsansatz ist das Aufteilen des Prozesses in zwei Prozesse (mit **fork**). Jeder Prozeß ist für die Kommunikation in einer Richtung zuständig (siehe Abbildung 15.2).

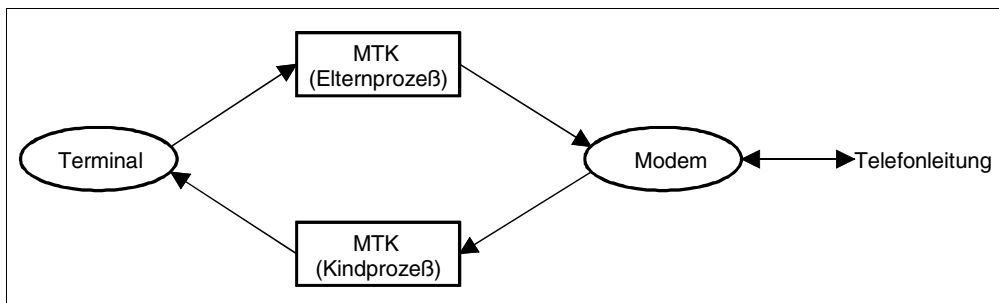


Abbildung 15.2: Modem-Terminalkommunikation mit zwei Prozessen

Bei dieser Vorgehensweise kann jeder Prozeß ein blockierendes **read** für seinen Eingabekanal durchführen.

Diese Technik führt allerdings zu einem etwas komplexeren Code, da sie die Beendigung der beiden Prozesse berücksichtigen muß. Wenn der Kindprozeß ein EOF empfängt (Modemverbindung wurde abgebrochen), so beendet der Kindprozeß sich und der Elternprozeß wird davon über das Signal `SIGCHLD` informiert. Sollte der Elternprozeß sich beenden (Benutzer gibt EOF am Terminal ein), so muß der Elternprozeß den Kindprozeß darüber informieren, daß er sich beenden muß. Diese Information kann auch über ein Signal (wie z.B. `SIGUSR1`) erfolgen.

15.1.2 Polling

Bei dieser Methode verwendet man nur einen Prozeß, der jedoch nicht-blockierende E/A-Operationen ausführt. Dazu setzt man beide Filedescriptoren auf »Nicht-Blockieren« und führt ein **read** auf den ersten Filedeskriptor durch. Sind dort Daten angekommen, so werden sie gelesen und verarbeitet. Sind dagegen keinerlei Daten zum Lesen vorhanden,

so kehrt **read** sofort (wegen »Nicht-Blockieren“) zurück. Nun führt man die gleichen Operationen für den zweiten Filedeskriptor durch. Man läßt eine gewisse Zeit verstreichen und versucht danach, vom ersten Filedeskriptor zu lesen usw.

Diese Art von Schleifen nennt man *Polling*. Das Problem der Polling-Technik ist die Vergeudung von wertvoller CPU-Zeit. Meistens werden nämlich keine neuen Daten angekommen sein, so daß die entsprechenden **read**-Aufrufe eigentlich nicht notwendig wären.

15.1.3 Asynchrone E/A

Bei dieser Technik weist der entsprechende Prozeß den Kern an, ihm ein Signal zu schicken, wenn ein Filedeskriptor für E/A bereit ist. Für diese Technik bietet SVR4 das Signal `SIGPOLL` und BSD-Unix das Signal `SIGIO` an. Während `SIGPOLL` nur funktioniert, wenn der Filedeskriptor sich auf einen `STREAM` bezieht, funktioniert `SIGIO` nur bei Filedeskriptoren, die auf Terminals oder Netzwerke eingestellt sind.

Ein großer Nachteil dieser Technik ist jedoch, daß pro Prozeß nur ein Signal (`SIGPOLL` oder `SIGIO`) zur Verfügung steht. Hat man also wie in unserem MTK-Programm mehrere Filedeskriptoren, so weiß man beim Empfang des Signals nicht, in welchem der beiden Filedeskriptoren Daten angekommen sind. Man muß also jeden der beiden Filedeskriptoren mit einem nicht-blockierenden **read** überprüfen. Asynchrone E/A wird in Kapitel 15.2 beschrieben.

15.1.4 E/A-Multiplexing

Diese Technik ist die beste aller hier vorgestellten Techniken. Hierbei erstellt man eine Liste von allen Filedeskriptoren, die von Interesse sind, und ruft dann eine Funktion auf, die erst zurückkehrt, wenn einer der Filedeskriptoren aus der Liste für die Ein-/Ausgabe bereit ist. Bei der Rückkehr aus der Funktion wird dem Aufrufer mitgeteilt, welche Filedeskriptoren für eine Ein-/Ausgabe bereit sind.

15.1.5 select – E/A-Multiplexing in SVR4 und BSD

Für E/A-Multiplexing steht sowohl in SVR4 als auch in BSD die Funktion **select** zur Verfügung.

```
#include <sys/types.h> /* für Datentyp fd_set */
#include <sys/time.h> /* für Datentyp struct timeval */
#include <unistd.h>

int select(int maxfd, fd_set *lesefds, fd_set *schreib_fds,
           fd_set *exceptfds, struct timeval *timeout);
```

gibt zurück: Anzahl von bereiten Filedeskriptoren (bei Erfolg);
0 bei Ablauf der Zeitschaltuhr;
-1 bei Fehler

Für das erste Argument `maxfd` muß immer `maxfd+1` angegeben werden. `maxfd` ist dabei die Nummer des größten Filedeskriptors aus allen drei Deskriptormengen, der für den Aufrufer von Interesse ist.

Die mittleren drei Argumente `lesefds`, `schreibfds` und `exceptfds` sind Zeiger auf sogenannte *Deskriptormengen*. Diese drei Mengen legen fest, welche Deskriptoren und welche Operationen (Lesen, Schreiben oder Exception (Ausnahme)) von Interesse sind. Eine Deskriptormenge hat den Datentyp `fd_set`, der für jeden möglichen Filedeskriptor ein Bit vorsieht. Für den Datentyp `fd_set` sind nur die folgenden Operationen möglich:

1. Deklaration einer Variablen von diesem Typ
2. Zuweisen einer Variablen dieses Typs an eine andere Variable dieses Typs
3. Anwendung der folgenden Makros auf Variablen dieses Typs:

```
FD_ZERO(fd_set *fdzgr)      /* Alle Bits in *fdzgr löschen */
FD_SET(int fd, fd_set *fdzgr) /* Bit für fd in *fdzgr setzen */
FD_CLR(int fd, fd_set *fdzgr) /* Bit für fd in *fdzgr löschen */
FD_ISSET(int fd, fd_set *fdzgr) /* Prüfen, ob Bit für fd gesetzt*/
```

Nach dem Deklarieren einer Deskriptormenge, wie z.B.

```
fd_set lesemenge;
int      fd;
```

muß man zunächst immer alle Bits dieser Menge löschen:

```
FD_ZERO(&lesemenge);
```

Danach kann man für jeden gewünschten Filedeskriptor das entsprechende Bit setzen, wie z.B.:

```
FD_SET(fd, &lesemenge);
FD_SET(STDIN_FILENO, &lesemenge);
```

Nach der Rückkehr aus ***select*** kann man überprüfen, ob ein bestimmtes Bit gesetzt ist oder nicht, wie z.B.:

```
if (FD_ISSET(fd, &lesemenge)) {
    .....
}
```

Abbildung 15.3 zeigt das Aussehen der einzelnen Deskriptormengen nach dem folgenden Codestück:

```
fd_set lesemenge,
       schreibmenge;

FD_ZERO(&lesemenge);
FD_ZERO(&schreibmenge);
FD_SET(0, &lesemenge);
FD_SET(3, &lesemenge);
```

```
FD_SET(4, &lesemenge);

FD_SET(1, &schreibmenge);
FD_SET(2, &schreibmenge);

select(5, &lesemenge, &schreibmenge, NULL, NULL);
```

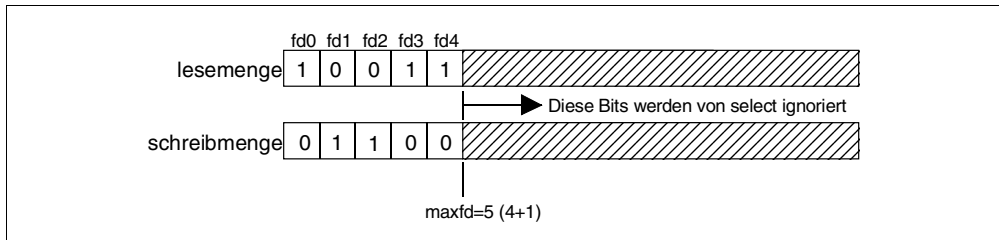


Abbildung 15.3: Beispiel für Deskriptormengen bei **select**

Man muß auf den maximalen Filedeskriptor (`maxfd`) 1 addieren, da Filedeskriptor-Nummern mit 0 beginnen und als erstes Argument die Anzahl der Filedeskriptoren anzugeben ist.

Der Aufrufer von **select** kann durch die Angabe von `NULL` für einen oder mehrere der drei mittleren `fdset`-Zeiger festlegen, daß er nicht an dieser Art von Operation (Lesen, Schreiben, Exception) interessiert ist.

Das letzte Argument `timeout` legt fest, wie lange **select** darauf warten soll, ob einer der spezifizierten Filedeskriptoren bereit für die E/A wird. Die Struktur `timeval` hat folgendes Aussehen:

```
struct timeval {
    long tv_sec; /* Sekunden */
    long tv_usec; /* und Mikrosekunden */
};
```

Es gibt hier drei unterschiedliche Möglichkeiten:

1. Ewiges Warten (`timeout == NULL`)

In diesem Fall kehrt **select** nur dann zurück, wenn entweder einer der spezifizierten Filedeskriptoren fertig ist oder aber ein Signal abgefangen wird. Wenn ein Signal abgefangen wird, so liefert **select** als Rückgabewert -1, wobei `errno` auf `EINTR` gesetzt wird.

2. Kein Warten (`timeout->tv_sec == 0 && timeout->tv_usec == 0`)

Nach dem Überprüfen aller spezifizierten Filedeskriptoren kehrt **select** sofort wieder zurück. Mit dieser Aufrufform kann man *Polling* nachbilden, um den Status von mehreren Deskriptoren herauszufinden, ohne daß man blockiert.

3. (Mikro)Sek. warten (`timeout->tv_sec != 0 || timeout->tv_usec != 0`)
select kehrt hierbei zurück, wenn einer der spezifizierten Filedeskriptoren für E/A bereit ist oder aber die mit `timeout` festgelegte Zeitschaltuhr abgelaufen ist. Falls die Zeitschaltuhr abläuft, bevor ein Filedeskriptor bereit ist, liefert **select** 0 als Rückgabewert. Diese Art des Wartens kann wie die erste Möglichkeit durch ein Signal abgebrochen werden.

Die Funktion **select** hat drei mögliche Rückgabewerte

-1	deutet auf einen Fehler hin, z.B. beim Auftreten eines Signals, das abgefangen wurde.
0	zeigt an, daß kein Filedeskriptor für E/A bereit ist. Dies tritt auf, wenn die Zeitschaltuhr abläuft, bevor ein Filedeskriptor bereit ist.
>0	Der Rückgabewert ist die Anzahl von Filedeskriptoren, die für E/A bereit sind. In diesem Fall zeigen die gesetzten Bits in den drei übergebenen Deskriptormengen an, welche Filedeskriptoren für E/A bereit sind. Es sollte jedoch unbedingt darauf geachtet werden, daß diese gesetzten Bits nur dann eine Aussagekraft haben, wenn der Rückgabewert >0 ist.

Filedeskriptoren, die für E/A bereit sind

Ein Filedeskriptor ist bereit für die E/A, wenn einer der folgenden Punkte zutrifft:

- 1. Ein Filedeskriptor aus der Lesedeskriptormenge (`lesefds`) ist bereit für E/A, wenn ein **read** auf diesen Filedeskriptor nicht blockiert ist.
- 2. Ein Filedeskriptor aus der Schreibdeskriptormenge (`schreibfds`) ist bereit für E/A, wenn ein **write** auf diesen Filedeskriptor nicht blockiert ist.
- 3. Ein Filedeskriptor aus der Exception-Deskriptormenge (`exceptfds`) ist bereit, wenn eine *Exception* für diesen Deskriptor vorhanden ist. Eine *Exception* tritt auf, wenn entweder *out-of-band*-Daten in einer Netzwerkverbindung ankommen oder aber bestimmte Bedingungen bei einem Pseudoterminal, der in *Packet-Modus* arbeitet, auftreten.

Es ist wichtig, darauf hinzuweisen, daß ein Filedeskriptor auch dann bereit für E/A ist, wenn EOF als nächstes zu lesendes Datum ansteht. Ein nachfolgendes **read** auf diesen Filedeskriptor liefert 0 als Rückgabewert, um anzuzeigen, daß ein EOF gelesen wurde.

Hinweis

Beim Rückgabewert -1 ist nicht garantiert, daß die drei `fdset`-Strukturen, auf die die drei Zeiger `lesefds`, `schreibfds` und `exceptfds` zeigen, noch die gleichen Bitmuster enthalten, die sie vor dem **select**-Aufruf hatten. Während einige Systeme (wie z.B. auch Linux) diese Werte nur bei einem Rückgabewert größer als 0 verändert (aktualisiert) haben, gilt dies nicht für alle Unix-Systeme.

Der Wert, auf den der Parameter `timeout` zeigt, enthält unter Linux nach einem `select`-Aufruf die Zeitspanne, die noch übrig war, bevor die übergebene Zeit abgelaufen wäre, was jedoch nicht für die meisten anderen Unix-Systeme gilt. Aus Portabilitätsgründen sollte man deshalb den Wert, auf den `timeout` zeigt, vor jedem `select`-Aufruf neu initialisieren.

15.1.6 `delay` – Ein `sleep` für Mikrosekunden mit `select`

Wenn beim Aufruf von **`select`** für alle drei Argumente `lesefds`, `schreibfds` und `exceptfds` `NULL` angegeben wird, dann verhält sich **`select`** wie die Funktion **`sleep`**. Anders als die Funktion **`sleep`**, die das Suspendieren eines Prozesses nur in Sekundenangaben zulässt, ermöglicht **`select`** abhängig von der Systemuhr das Anhalten eines Prozesses für Mikrosekunden.

Das nachfolgende Programm 15.1 (`delay.c`) zeigt eine mögliche Implementierung einer eigenen **`sleep`**-Funktion mit Mikrosekunden als Argument.

```
#include <sys/types.h>
#include <sys/times.h>
#include <sys/time.h>
#include <stddef.h>
#include "eighdr.h"

void delay(long mikrosek)
{
    struct timeval timeout;

    timeout.tv_sec = mikrosek / 1000000L;
    timeout.tv_usec = mikrosek % 1000000L;
    select(0, NULL, NULL, NULL, &timeout);
}

int
main(int argc, char *argv[])
{
    clock_t start, ende;
    struct tms puffer;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s mikrosek", argv[0]);

    if ( (start = times(&puffer)) == -1)
        fehler_meld(WARNUNG_SYS, "times-Fehler");
    delay(atol(argv[1]));
    if ( (ende = times(&puffer)) == -1)
        fehler_meld(WARNUNG_SYS, "times-Fehler");

    printf("...%lg Sek. gewartet\n",
           (double)(ende-start)/sysconf(_SC_CLK_TCK));
}
```

Programm 15.1 (**`delay.c`**): Implementierung einer **`sleep`**-Funktion mit Mikrosekunden als Argument

Nachdem man dieses Programm 15.1 (`delay.c`) kompiliert und gelinkt hat

```
cc -o delay delay.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ delay 1900000
...1.9 Sek. gewartet
$ delay 60000
...0.06 Sek. gewartet
$ delay 10500000
...10.51 Sek. gewartet
$
```

Hinweis

Die Funktion ***select*** wird zwar von SVR4 und BSD-Unix angeboten, ist aber nicht Bestandteil von POSIX.1.

BSD-Unix liefert die Summe aller fertigen Filedeskriptoren in den einzelnen Deskriptormengen als Rückgabewert. Falls der gleiche Filedeskriptor in zwei Deskriptormengen bereit ist (z.B. in der Lese- und Schreibdeskriptormenge), so wird er also zweimal gezählt. In SVR4 wird dagegen ein solcher Filedeskriptor nur einmal gezählt.

Ob für einen Filedeskriptor die Blockierung ein- oder ausgeschaltet ist, hat keinerlei Auswirkung auf den ***select***-Aufruf. Wenn man z.B. von einem nicht-blockierenden Filedeskriptor lesen möchte und man ruft ***select*** mit einer Zeitschaltuhr von 10 Sekunden auf, so wird ***select*** für 10 Sekunden blockieren.

15.1.7 poll – E/A-Multiplexing für STREAMS in SVR4

Um E/A-Multiplexing für STREAMS durchzuführen, stellt SVR4 die Funktion ***poll***, die nicht Bestandteil von POSIX.1 ist, zur Verfügung.

```
#include <stropts.h>
```

```
#include <poll.h>
```

```
int poll(struct pollfd fdarray[], unsigned long nfds, int timeout);
```

gibt zurück: Anzahl von bereiten Filedeskriptoren (bei Erfolg);

0 bei Ablauf der Zeitschaltuhr;

-1 bei Fehler

Obwohl ***poll*** eigentlich nur für STREAMS vorgesehen ist, kann ***poll*** für jede Art von Filedeskriptor verwendet werden. Anstelle von einzelnen Deskriptormengen für jede Operation (Lesen, Schreiben, Exception) muß bei ***poll*** die Adresse eines Arrays übergeben werden, dessen Elemente den Datentyp `struct pollfd` haben.

```
struct pollfd {
    int fd;          /* zu prüfender Fildeskriptor oder < 0 für Ignorieren */
    short events;    /* Ereignisse, die für fd von Interesse sind */
    short revents;   /* Ereignisse, die bei fd eingetreten sind */
}
```

Die Anzahl der Elemente des Arrays wird über das Argument `nfds` festgelegt.

Die Komponente `events` muß für jedes Arrayelement mit einem oder mehreren Werten aus Tabelle 15.1 besetzt werden. Über diese Werte teilt man dem Kern mit, an was man für den betreffenden Fildeskriptor interessiert ist. Der Kern seinerseits setzt dann die Komponente `revents`, um dem Aufrufer von ***poll*** mitzuteilen, welche Ereignisse für diesen Fildeskriptor aufgetreten sind. Die Komponente `events` wird vom Kern nur gelesen und niemals modifiziert.

Name	Angabe in events möglich	Vorkommen in revents möglich	Beschreibung
POLLIN	x	x	Daten, die nicht hochprior sind, können ohne Blockierung gelesen werden.
POLLRDNORM	x	x	Normale Daten (Band-Priorität 0) können ohne Blockierung gelesen werden.
POLLRDBAND	x	x	Daten, die nicht die Priorität 0 haben (also keine normale Daten sind) können ohne Blockierung gelesen werden.
POLLPRI	x	x	Hochpriore Daten können ohne Blockierung gelesen werden.
POLLOUT	x	x	Normale Daten können ohne Blockierung geschrieben werden.
POLLWRNORM	x	x	identisch mit POLLOUT
POLLWRBAND	x	x	Daten, die nicht die Priorität 0 haben (also keine normale Daten sind), können ohne Blockierung geschrieben werden.
POLLERR		x	Ein Fehler ist aufgetreten.
POLLHUP		x	Eine Verbindungsunterbrechung ist aufgetreten.
POLLNVAL		x	Dem Fildeskriptor ist keine offene Datei zugeordnet.

Tabelle 15.1: Mögliche Werte für die ***poll***-Argumente `events` und `revents`

Die letzten drei Werte in Tabelle 15.1 werden beim Auftreten der entsprechenden Exception durch den Kern gesetzt.

Das letzte Argument `timeout` legt fest, wie lange **poll** warten soll, ob einer der spezifizierten Filedeskriptoren für E/A bereit wird. Es gibt drei unterschiedliche Möglichkeiten:

1. Ewiges Warten (`timeout == INFTIM`)
Die Konstante `INFTIM` ist in `<stropts.h>` definiert und ihr Wert ist meist -1.
In diesem Fall kehrt **poll** zurück, wenn entweder einer der spezifizierten Filedeskriptoren bereit ist oder ein Signal abgefangen wird. Wenn ein Signal abgefangen wird, so liefert **poll** als Rückgabewert -1, wobei `errno` auf `EINTR` gesetzt wird.
2. Kein Warten (`timeout == 0`)
Nach dem Überprüfen aller spezifizierten Filedeskriptoren kehrt **poll** sofort wieder zurück. Mit dieser Aufrufform kann man Polling realisieren, um den Status von mehreren Deskriptoren zu erfragen, ohne daß man blockiert.
3. timeout Millisekunden warten (`timeout > 0`)
poll kehrt hierbei zurück, wenn einer der spezifizierten Filedeskriptoren für E/A bereit ist oder aber die mit `timeout` angegebenen Millisekunden abgelaufen sind. Falls die Millisekunden ablaufen, bevor ein Filedeskriptor bereit ist, liefert **poll** 0 als Rückgabewert.

15.1.8 delay2 – Ein sleep für Millisekunden mit poll

Wird beim Aufruf von **poll** für das Argument `nfds` der Wert 0 angegeben, dann verhält sich **poll** wie die Funktion **sleep**. Anders als die Funktion **sleep**, die das Suspendieren eines Prozesses nur in Sekundenangaben zuläßt, ermöglicht **poll** abhängig von der Taktrate der Systemuhr das Anhalten eines Prozesses für Millisekunden. Das nachfolgende Programm 15.2 (`delay2.c`) zeigt eine mögliche Implementierung einer eigenen **sleep**-Funktion mit Millisekunden als Argument.

```
#include <sys/types.h>
#include <sys/times.h>
#include <poll.h>
#include <stropts.h>
#include "eighdr.h"
void delay(long millisek)
{
    struct pollfd    leer;
    int              timeout;

    if ( (timeout = millisek) <= 0)
        timeout = 1;

    poll(&leer, 0, timeout);
}

int
main(int argc, char *argv[])
{
    clock_t    start, ende;
    struct tms puffer;
```

```

if (argc != 2)
    fehler_meld(FATAL, "usage: %s millisek", argv[0]);

if ( (start = times(&puffer)) == -1)
    fehler_meld(WARNUNG_SYS, "times-Fehler");;
delay(atoi(argv[1]));
if ( (ende = times(&puffer)) == -1)
    fehler_meld(WARNUNG_SYS, "times-Fehler");;

printf("...%lg Sek. gewartet\n",
       (double)(ende-start)/sysconf(_SC_CLK_TCK));
}

```

Programm 15.2 (**delay2.c**): Implementierung einer **sleep**-Funktion mit Millisekunden als Argument

Nachdem man dieses Programm 15.2 (delay2.c) kompiliert und gelinkt hat

```
cc -o delay2 delay2.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ delay2 350
...0.35 Sek. gewartet
$ delay2 12345
...12.35 Sek. gewartet
$

```

Hinweis

Die Funktion **poll** wird nur von SVR4 angeboten und ist nicht Bestandteil von POSIX.1.

Ob für einen Filedeskriptor Blockierung ein- oder ausgeschaltet ist, hat wie bei **select** keine Auswirkung auf einen **poll**-Aufruf.

Zwischen einem EOF und einer Verbindungsunterbrechung besteht ein Unterschied. Wenn EOF als nächstes zu lesendes Datum ansteht, wird in revents **POLLIN** gesetzt. Ein nachfolgendes **read** auf diesen Filedeskriptor liefert dann 0 als Rückgabewert, um anzuzeigen, daß ein EOF gelesen wurde. Wird dagegen eine Verbindung unterbrochen, z.B. durch Beenden der Modemverbindung, so wird in revents **POLLUP** gesetzt.

15.2 Asynchrone E/A

Die beiden im vorherigen Kapitel beschriebenen Funktionen **select** und **poll** sind eine synchrone Form der Mitteilung über anstehende Ein-/Ausgaben. Ruft man **select** oder **poll** nicht auf, so erfährt man nichts über anstehende E/A-Anforderungen.

Eine Form der asynchronen Kommunikation sind Signale, die in Kapitel 13 vorgestellt wurden. SVR4 und BSD-Unix ermöglichen asynchrone E/A mit jeweils einem Signal: SIGPOLL (SVR4) bzw. SIGIO (BSD). Dieses jeweilige Signal teilt dem betreffenden Prozeß mit, daß bei einem Filedeskriptor irgend etwas ansteht.

Ein Nachteil dieser asynchronen E / A in SVR4 und BSD-Unix ist, daß in beiden Systemen nur ein Signal pro Prozeß angeboten wird. Das bedeutet, daß bei mehreren aktiven Filedeskriptoren das Auftreten des Signals keinerlei Auskunft darüber gibt, auf welchen der Filedeskriptoren sich dieses Signal bezieht.

Nachfolgend werden die Besonderheiten der asynchronen E / A für die beiden Systeme SVR4 und BSD-Unix im einzelnen beschrieben.

15.2.1 SVR4 – Asynchrone E/A nur für STREAMS

Im SVR4 wird die asynchrone E / A mit dem Signal SIGPOLL nur für STREAMS angeboten. Zum Einschalten von asynchroner E / A für einen STREAM muß man **ioctl** aufrufen. Für das zweite Argument (*operation*) muß man dabei I_SETSIG angeben. Für das dritte Argument müssen eine oder mehrere mit | (bitweiser OR) verknüpfte Konstanten aus Tabelle 15.2 angegeben werden. Diese Konstanten sind in <stropts.h> definiert.

Konstante	Bedeutung
S_INPUT	Eine Message, die nicht hochprior ist, ist angekommen.
S_RDNORM	Eine normale Message (Priorität 0) ist angekommen.
S_RDBAND	Eine nicht-normale Message (Priorität 1-255) ist angekommen.
S_BANDURG	Wenn zusammen mit S_RDBAND angegeben, so wird das Signal SIGURG anstelle von SIGPOLL generiert, wenn eine nicht-normale Message (Priorität 1-255) angekommen ist.
S_HIPRI	Eine hochpriore Message ist angekommen.
S_OUTPUT	Die Schreibwarteschlange ist nicht mehr voll.
S_WRNORM	identisch mit S_OUTPUT
S_WRBAND	Eine nicht-normale Message (Priorität 1-255) kann geschickt werden.
S_MSG	Eine Signal-Message ist angekommen, die das Signal SIGPOLL enthält.
S_ERROR	Eine M_ERROR-Message ist angekommen.
S_HANGUP	Eine M_HANGUP-Message ist angekommen.

Tabelle 15.2: Mögliche Angaben für das 3. Argument von **ioctl**, um das Signal SIGPOLL zu generieren

»Angekommen« in Tabelle 15.2 bedeutet »in Lesewarteschlange des STREAM-Kopfes angekommen».

Vor dem **ioctl**-Aufruf, der die Bedingungen einrichtet, die das Signal SIGPOLL generieren sollen, sollte immer zuerst ein Signalhandler eingerichtet werden, der SIGPOLL abfängt. Sonst wird durch dieses Signal der betreffende Prozeß beendet (Default-Aktion).

15.2.2 BSD-Unix – Asynchrone E/A nur für Terminals und Netzwerkverbindungen

In BSD-Unix ist asynchrone E/A eine Kombination aus den beiden Signalen SIGIO und SIGURG. Während SIGIO für die allgemeine asynchrone E/A verwendet wird, wird SIGURG benutzt, um dem Prozeß mitzuteilen, daß *out-of-band*-Daten in einer Netzwerkverbindung angekommen sind.

Um das Signal SIGIO zu empfangen, muß ein Prozeß die folgenden Schritte ausführen:

1. Einrichten eines Signalhandlers für SIGIO (mit **signal** oder **sigaction**).
2. Setzen der Prozeß-ID oder Prozeßgruppen-ID, um das Signal für den Filedescriptor zu empfangen:

```
fcntl(fd, F_SETOWN, id); /* Setzen der Prozeß-ID */
fcntl(fd, F_SETOWN, -id); /* Setzen der Prozeßgruppen-ID */
```

3. Einschalten der asynchronen E/A für den Filedescriptor:

```
fcntl(fd, F_SETFL, O_ASYNC | alt_flags);
```

Dieser 3. Schritt kann nur auf Deskriptoren angewendet werden, die sich auf ein Terminal oder ein Netzwerk beziehen. Dies ist der Grund, warum in BSD-Unix die asynchrone E/A nur für Terminals und Netzwerke möglich ist.

Für das Signal SIGURG müssen nur die ersten beiden Schritte durchgeführt werden. Dieses Signal wird nur für Deskriptoren generiert, die sich auf Netzwerkverbindungen beziehen, bei denen *out-of-band*-Daten möglich sind.

15.3 Memory Mapped I/O

Memory Mapped I/O (E/A über Speicherabbild) ermöglicht die Herstellung einer Beziehung zwischen einer Datei (auf Festplatte) und einem Puffer (im Hauptspeicher). Ist diese spezielle Beziehung hergestellt, dann werden beim Lesen aus diesem Puffer die entsprechenden Bytes aus der zugehörigen Datei gelesen. Umgekehrt werden beim Schreiben von Daten in diesen Puffer diese direkt in die zugehörige Datei geschrieben. Mit *Memory Mapped I/O* ist also eine Ein-/Ausgabe ohne **read** oder **write** möglich.

15.3.1 mmap – Einrichten von Memory Mapped I/O

Zum Memory Mapped IO stellen sowohl SVR4 als auch BSD-Unix die Funktion **mmap** zur Verfügung.

```
#include <sys/types.h>
#include <sys/mman.h>
```

```
caddr_t mmap(addr_t adr, size_t laenge, int schutz,
              int flag, int fd, off_t offset);
```

gibt zurück: Anfangsadresse des zugeordneten Speicherbereichs (bei Erfolg); -1 bei Fehler

adr

adr legt die Anfangsadresse des *mapped*-Speicherbereichs fest. Der Datentyp *caddr_t* ist meist definiert als `char *`. Normalerweise gibt man für *adr* den Wert 0 an, um die Wahl der Anfangsadresse dem System zu überlassen. Der Rückgabewert von ***mmap*** ist die vom System festgelegte Anfangsadresse.

laenge

laenge legt die Anzahl von Bytes fest, die dieser mapped-Speicherbereich umfassen soll.

fd

fd spezifiziert den Filedeskriptor der Datei, die dem ausgewählten mapped-Speicherbereich direkt zugeordnet werden soll.

offset

offset legt das Offset des Dateibereichs fest, der dem mapped-Speicherbereich zugeordnet werden soll.

schutz

schutz legt die Schutzart für den mapped-Bereich fest. Für *schutz* sind dabei die folgenden Angaben möglich:

PROT_READ	Bereich darf gelesen werden
PROT_WRITE	Bereich darf beschrieben werden
PROT_EXEC	Bereich darf ausgeführt werden
PROT_NONE	Auf diesen Bereich ist keinerlei Zugriff möglich (nicht in BSD-Unix)

Die Angabe von *schutz* muß dabei mit dem bei ***open*** für die Datei festgelegten Öffnungsmodus übereinstimmen. So kann z.B. PROT_WRITE nicht für eine Datei verwendet werden, die als »nur-lesbar« geöffnet wurde.

flag

flag spezifiziert zusätzliche Forderungen für den mapped-Bereich:

MAP_FIXED

Der Rückgabewert muß gleich dem Argument `adr` sein. Um Portabilität zu bewahren, sollte diese Angabe nicht verwendet werden. Wenn `MAP_FIXED` nicht angegeben ist und `adr` ungleich 0 ist, dann ist die `adr`-Angabe an den Kern lediglich ein Vorschlag, den dieser auch ignorieren darf.

MAP_SHARED

Jedes Schreiben in den mapped-Bereich wird auf die Originaldatei (und nicht auf eine Kopie) durchgeführt. Dies bedeutet, daß alle anderen Prozesse, die den mapped-Bereich auch benutzen, diese durch das Schreiben bedingten Änderungen sofort kennen.

MAP_PRIVATE

Jedes Schreiben in den mapped-Bereich wird auf eine Kopie (und nicht auf die Originaldatei selbst) durchgeführt. Andere Prozesse, die diesen mapped-Bereich auch benutzen, erfahren also – anders als bei `MAP_SHARED` – nichts von den vorgenommenen Änderungen. Ein Anwendungsfall hierfür ist z.B. ein Debugger, der es dem Benutzer ermöglicht, die Instruktionen eines Programms zu verändern. In diesem Fall wird die Kopie und nicht die Originalprogrammdatei modifiziert.

MAP_ANONYMOUS

Anstelle einer Datei wird ein anonymer mapped-Bereich eingerichtet. Ein so eingerichteter mapped-Bereich kann von keinem anderen Prozeß mitbenutzt werden. Mittels anonymer mapped-Bereiche kann ein Prozeß neuen Speicher für sich allokalieren. Diese Vorgehensweise wird oft von **malloc** sowie einigen speziellen Anwendungen verwendet. Bei `MAP_ANONYMOUS` hat der Parameter `fd` keine Bedeutung.

MAP_DENYWRITE (Linux)

Ein Schreiben in den mapped-Bereich von außerhalb, also z.B. mit **write**, ist nicht erlaubt. Dieses Flag macht Sinn bei mapped-Bereichen, die ausführbar sind. Ist `MAP_DENYWRITE` für einen Speicherbereich gesetzt, geben alle Schreibzugriffe, die nicht intern über den mapped-Speicher stattfinden, `ETXTBSY` zurück.

MAP_GROWSDOWN (Linux)

Dieses Flag legt fest, daß bei Zugriffen unmittelbar vor einem mapped-Bereich nicht das Signal `SIGSEGV` generiert wird, sondern automatisch ein neuer anonymer mapped-Bereich allokiert und der entsprechende Prozeß nicht abgebrochen, sondern normal fortgesetzt wird. `MAP_GROWSDOWN` wird benutzt, um den Stack eines Prozesses automatisch wachsen zu lassen.

MAP_LOCKED (Linux)

Der mapped-Speicherbereich wird gesperrt, was bedeutet, daß er nicht ausgelagert wird, was für Echtzeitanwendungen wichtig ist. `MAP_LOCKED` kann nur vom Superuser (`root`) gesetzt werden.

`MAX_SHARED` oder aber `MAX_PRIVATE` muß immer angegeben sein.

BSD-Unix bietet weitere implementierungsspezifische Flags an, deren Namen mit `MAP_` beginnen. Genauer hierzu läßt sich in der Manpage zu `mmap(2)` finden.

Die Abbildung 15.4 verdeutlicht das *Memory Mapped I/O*, wie es von der Funktion **`mmap`** eingerichtet wird.

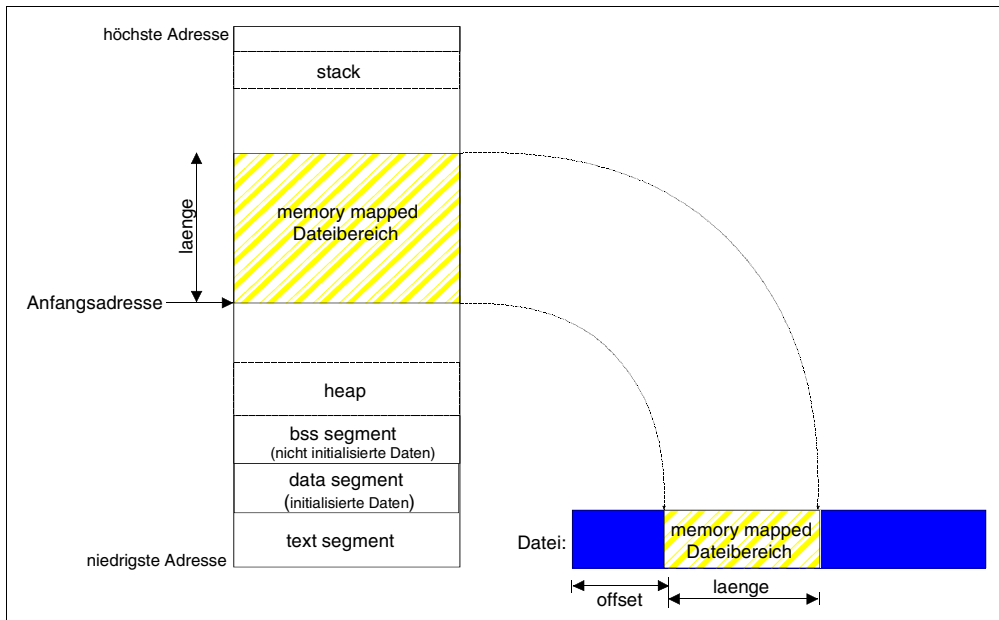


Abbildung 15.4: Memory Mapped I/O mit der Funktion **`mmap`**

In der Abbildung 15.4 ist der mapped-Speicherbereich zwischen Heap und Stack eingezeichnet. Dies ist von der jeweiligen Implementierung abhängig und nicht allgemeingültig.

Hinweis

Die Werte für `offset` und `adr` müssen normalerweise, wenn `MAP_FIXED` angegeben ist, ein Vielfaches der *Page*-Größe sein, die das System für virtuelle Speicherung (*virtual memory*) benutzt. In SVR4 kann man diesen Wert mit dem Aufruf `sysconf(SC_PAGESIZE)` erfragen. In BSD-Unix wird die *Page*-Größe durch die Konstante `NBPG` in der Headerdatei `<sys/param.h>` definiert.

Unter Linux kann die Größe einer Page mit der in `<unistd.h>` deklarierten Funktion **`getpagesize`** ermittelt werden.

Zwei Signale können im Zusammenhang mit Memory Mapped I/O generiert werden:

SIGSEGV

Dieses Signal wird normalerweise generiert, wenn versucht wird, auf einen unerlaubten Speicherbereich zuzugreifen. So wird es z.B. generiert, wenn man in einen mapped-Speicherbereich schreiben möchte, der beim **mmap**-Aufruf als »nur-lesbar« eingerichtet wurde.

SIGBUS

Dieses Signal kann generiert werden, wenn man versucht, auf einen nicht mehr gültigen Bereich des mapped-Speicherbereichs zuzugreifen. Diese Situation kann z.B. dann auftreten, wenn auf einen mapped-Bereich zugegriffen wird, der nicht mehr zur Datei gehört, da diese Datei z.B. zwischenzeitlich von einem anderen Prozeß verkleinert wurde.

Ein mapped-Speicherbereich wird bei einem **fork** an den Kindprozeß vererbt, da dieser Bestandteil des Adreßraums vom Elternprozeß ist. Bei einem **exec** wird dagegen ein mapped-Speicherbereich nicht vererbt.

Ob der bei einem **mmap** angegebene `schutz` für einen mapped-Speicherbereich auch wirksam wird, hängt von der darunterliegenden Hardware ab. Manche Architekturen können z.B. nicht das Ausführrecht für einen mapped-Bereich setzen, wenn kein Lese-recht gewährt wird. Bei solchen Hardwarearchitekturen ist die Angabe von `PROT_EXEC` äquivalent zur Angabe von `PROT_EXEC | PROT_READ`.

Beispiel

Einfache Nachbildung des Kommandos `cat` mittels `mmap`

Das Programm 15.3 (`cat_mmap.c`) bildet das Unix-Kommando **cat** nach. Es öffnet die auf der Kommandozeile angegebenen Dateien, bildet sie in einen mapped-Speicherbereich ab und gibt dann die gesamte Datei mit einem Aufruf der Funktion **write** auf die Standardausgabe aus.

```
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    int          i, fd;
    struct stat  attribut;
    void         *map_addr;

    for (i=1; i<argc; i++) {
        if ( (fd = open(argv[i], O_RDONLY)) < 0 ) {
            fehler_meld(WARNUNG_SYS, "kann Datei '%s' nicht oeffnen", argv[i]);
```

```

        continue;
    }

    /* fstat fuer die Groesse der Datei */
    if (fstat(fd, &attribut) == -1) {
        fehler_meld(WARNUNG_SYS, "Fehler bei fstat auf Datei '%s'", argv[i]);
        continue;
    }

    /* hier wird MAP_SHARED spezifiziert, aber MAP_PRIVATE waere
       auch moeglich, da kein Schreiben im mapped-Bereich stattfindet */
    map_addr = mmap(NULL, attribut.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (map_addr == ((caddr_t) -1)) {
        fehler_meld(WARNUNG_SYS, "Fehler bei mmap auf Datei '%s'", argv[i]);
        continue;
    }

    close(fd);

    if (write(STDOUT_FILENO, map_addr, attribut.st_size) != attribut.st_size)
        fehler_meld(WARNUNG_SYS, "Fehler bei write fuer Datei '%s'", argv[i]);
}

exit(0);
}

```

Programm 15.3 (cat-mmap.c): Nachbildung des Kommandos cat mit Memory Mapped I/O

Dieses Programm 15.3 (cat_mmap.c) verdeutlicht auch, daß mapped-Speicherbereiche bestehenbleiben, wenn die zugehörige Datei geschlossen wird.

Typische Anwendungen für Memory Mapped I/O

1. Schnellere und einfachere Dateizugriffe

Memory Mapped I/O kann in vielen Anwendungsfällen die Ausführungsgeschwindigkeit von Programmen erhöhen, da man bei der Ein- und Ausgabe im Speicher arbeitet und nicht zeitaufwendig auf die externen Medien zugreifen muß. Zudem können Programme vereinfacht werden, da man auf die Dateien (mapped-Bereiche) mit Zeigern zugreifen kann, und nicht mehr mühsam mit **read**, **write** und **lseek** arbeiten muß.

2. Dynamisches Laden

Ausführbare Dateien können in den Speicherbereich des Programms abgebildet werden, wodurch neue Programmteile, die auszuführen sind, dynamisch geladen werden können. Auf diese Art ist z.B. auch das dynamische Laden unter Linux, das in einem späteren Kapitel vorgestellt wird, realisiert.

3. Shared Memory für verwandte Prozesse

Wird für die spezielle Gerätedatei `/dev/zero` mit der Funktion **mmap** ein mapped-Speicherbereich eingerichtet, so wird ein namenloser Speicherbereich der geforderten Größe angelegt, der mit 0 initialisiert wird. Ist beim **mmap**-Aufruf das Flag `MAP_SHARED` angegeben, so können alle verwandten Prozesse auf diesen mapped-Speicherbereich zugreifen. In Kapitel 18.4.7 wird diese Technik näher erläutert.

4. Realisierung von Just-In-Time Compilern

Da mapped-Speicherbereiche das Ausführrecht besitzen können, ist es möglich, dort hin Anweisungen zu schreiben, die anschließend ausgeführt werden. Diese Eigenschaft wird von Just-In-Time-Compilern genutzt.

5. Kommunikation zwischen Prozessen, die nicht gleichzeitig ablaufen

Mit mapped-Speicherbereichen können sich Prozesse, die nicht gleichzeitig, sondern versetzt ablaufen, Speicherbereiche teilen. Da ein mapped-Speicherbereich auch nach dem Beenden eines Prozesses noch erhalten bleiben kann, ist es möglich, daß Prozesse, die nicht zur gleichen Zeit ablaufen, über einen mapped-Speicherbereich Informationen austauschen.

15.3.2 munmap – Aufheben von Memory Mapped I/O

Um für einen mapped-Speicherbereich das Memory Mapped I/O wieder aufzuheben, steht die Funktion ***munmap*** zur Verfügung.

```
#include <sys/types.h>
#include <sys/mman.h>

int munmap(caddr_t adr, size_t laenge);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

munmap hebt lediglich das Memory Mapped I/O für den spezifizierten Speicherbereich auf und bewirkt nicht, daß die veränderten Inhalte des mapped-Speicherbereichs automatisch in die zugehörige Datei geschrieben werden, um diese so zu aktualisieren. Für diesen Zweck bieten einige Systeme die Funktion ***msync*** an, die ähnlich der Funktion ***fsync*** ist, nur eben für Memory Mapped I/O.

Das automatische Aktualisieren ist dagegen für `MAP_SHARED` immer garantiert.

Hinweis

Ein Schließen des Filedeskriptors, der bei ***mmap*** angegeben wurde, bewirkt nicht die Aufhebung des Memory Mapped I/O für den betreffenden mapped-Speicherbereich. Dagegen wird bei der Beendigung des Prozesses, der das Memory Mapped I/O eingerichtet hat, dieses automatisch aufgehoben.

15.3.3 msync – Aktualisieren der einem mapped-Bereich zugeordneten Datei

Wird in einen einer Datei zugeordneten mapped-Speicherbereich geschrieben, so werden normalerweise die dadurch bedingten Änderungen im mapped-Bereich nicht sofort in die zugehörige Datei übernommen. Möchte ein Prozeß, daß die durch die Schreibaktivitäten stattgefundenen Änderungen auch in der Datei (auf der Festplatte) durchgeführt werden, muß er die Funktion ***msync*** aufrufen.

```
#include <sys/types.h>
#include <sys/mman.h>

int msync(caddr_t adr, size_t laenge, int flags);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die beiden ersten Parameter `adr` und `laenge` legen den mapped-Speicherbereich fest, dessen Inhalt mit der zugehörigen Datei (auf der Festplatte) synchronisiert werden soll.

Der Parameter `flags` legt die Art der Synchronisation fest. Für `flags` können die folgenden Konstanten (eventuell mit bitweisem OR (`|`) verknüpft) angegeben werden:

MS_ASYNC	Geänderter mapped-Speicherbereich sollte baldmöglichst synchronisiert werden. Es kann entweder nur MS_ASYNC oder aber MS_SYNC angegeben werden.
MS_SYNC	Geänderter mapped-Speicherbereich ist vor der Rückkehr der Funktion msync zu synchronisieren. Es kann entweder nur MS_ASYNC oder aber MS_SYNC angegeben werden.
MS_INVALIDATE	Bei diesem Flag wird dem Systemkern die Entscheidung überlassen, ob die Änderungen überhaupt jemals in der Datei (auf der Festplatte) durchgeführt werden. Dieses Flag, das für besondere Situationen vorgesehen ist, teilt dem Systemkern mit, daß die Änderungen nicht unbedingt in die Datei übernommen werden sollen.

15.3.4 Sperren von Speicherbereichen

In den meisten heutigen Unix-Systemen können bei Mangel an Hauptspeicher Speicherbereiche auf die Festplatte ausgelagert werden, die bei Bedarf wieder einzulagern sind, was natürlich Zeit kostet. Bei speziellen Programmen, die für zeitkritische Aufgabenstellungen entwickelt wurden, sind diese durch das erneute Einlagern bedingten Zeitverzögerungen oft nicht annehmbar. Für solche zeitkritischen Anwendungen wird die Möglichkeit angeboten, Bereiche im Hauptspeicher (RAM) zu sperren, um so bestimmte Zugriffszeiten garantieren zu können. Aus Sicherheitsgründen dürfen momentan nur Superuser-Prozesse, die mit `root`-Rechten ablaufen, Speicherbereiche sperren. Die maximale Größe an Speicher, die ein solcher Prozeß sperren kann, wird durch das Ressourcenlimit `RLIMIT_MEMLOCK` (siehe auch Kapitel 9.5) definiert. Zum Sperren von Speicherbereichen bzw. zum Aufheben von eingerichteten Sperren stehen die folgenden Funktionen zur Verfügung.

```
#include <sys/mman.h>

int mlock(caddr_t adr, size_t laenge);

int mlockall(int flags);

int munlock(caddr_t adr, size_t laenge);

int munlockall(void);
```

alle geben zurück: 0 (bei Erfolg); -1 bei Fehler

mlock

sperrt ab der Adresse *adr* den Speicherbereich von *laenge* Bytes. Da immer nur ganze Pages (Speicherseiten) gesperrt werden können, sperrt **mlock** in Wirklichkeit alle Pages zwischen der Page, die die erste Adresse (*adr*) enthält, und der Page, die die letzte Adresse (wird durch *laenge* ermittelt) enthält. Bei der Rückkehr von **mlock** ist garantiert, daß sich alle diese gesperrten Pages im Hauptspeicher (RAM) befinden.

mlockall

sperrt den gesamten Adreßraum des aufrufenden Prozesses. Für den Parameter *flags* kann eine oder beide (mit bitweisem OR (|) verknüpft) der folgenden Konstanten angegeben werden:

MCL_CURRENT

Alle Pages, die sich gerade im Adreßraum des Prozesses befinden, werden gesperrt. Bei der Rückkehr von **mlockall** ist garantiert, daß sich alle diese gesperrten Pages im Hauptspeicher (RAM) befinden.

MCL_FUTURE

Alle Pages, die in Zukunft zum Adreßraum des Prozesses hinzugefügt werden, werden im Hauptspeicher (RAM) gesperrt.

munlock

hebt die Sperren für die Pages, die durch die beiden Parameter *adr* und *laenge* spezifiziert sind, wieder auf.

munlockall

hebt alle vorhandenen Sperren im Adreßraum des aufrufenden Prozesses wieder auf.

15.3.5 Laufzeitverbesserungen durch Memory Mapped I/O

Das Programm 15.3 (*mmap.c*) kopiert eine Datei. Es kopiert diese Datei zweimal: Einmal unter Verwendung von **mmap** und **memcpy** und einmal unter Verwendung von **read** und **write**. Bei beiden Kopiervorgängen werden die benötigten Zeiten gemessen und ausgegeben.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
```



```

DATEIEN_OEFFNEN
normal_copy(von_fd, nach_fd);
DATEIEN_SCHLIESSEN

exit(0);
}

/*---- Kopieren mit mmap (Memory Mapped I/O) -----*/
static void mmap_copy(int von_fd, int nach_fd)
{
    struct stat    statpuff;
    caddr_t        quell_adr, ziel_adr;
    struct tms      start_zeit, ende_zeit;
    clock_t         uhr_start, uhr_ende;

    /*---- Quelldateigroesse wird gebraucht */
    if (fstat(von_fd, &statpuff) < 0)
        fehler_meld(FATAL_SYS, "fstat-Fehler");

    /*---- Groesse der Zieldatei setzen */
    if (lseek(nach_fd, statpuff.st_size-1, SEEK_SET) == -1)
        fehler_meld(FATAL_SYS, "Fehler bei lseek");
    if (write(nach_fd, "", 1) != 1)
        fehler_meld(FATAL_SYS, "write-Fehler");

    STOPPUHR_EIN

    /*---- Memory Mapped I/O */
    if ( (quell_adr = mmap(0, statpuff.st_size, PROT_READ,
                          MAPFILE | MAP_SHARED, von_fd, 0)) == (caddr_t) -1)
        fehler_meld(FATAL_SYS, "mmap-Fehler fuer Quelldatei");
    if ( (ziel_adr = mmap(0, statpuff.st_size, PROT_READ | PROT_WRITE,
                          MAPFILE | MAP_PRIVATE, nach_fd, 0)) == (caddr_t) -1)
        fehler_meld(FATAL_SYS, "mmap-Fehler fuer Zieldatei");

    /*---- Kopieren der Datei mit memcpy */
    memcpy(ziel_adr, quell_adr, statpuff.st_size);

    STOPPUHR_AUS

    if (munmap(quell_adr, statpuff.st_size) == -1 ||
        munmap(ziel_adr, statpuff.st_size) == -1)
        fehler_meld(FATAL_SYS, "munmap-Fehler");

    zeit_ausgabe("mmap/memcpy", uhr_ende-uhr_start, &start_zeit, &ende_zeit);
}

/*---- Normales Kopieren mit read und write -----*/
static void normal_copy(int von_fd, int nach_fd)
{
    char            puffer[PUFF_GROESSE];
    ssize_t         n;
    struct tms      start_zeit, ende_zeit;

```

```

clock_t      uhr_start, uhr_ende;

if (lseek(von_fd, 0L, SEEK_SET) == -1) /* Auf Dateianfang setzen */
    fehler_meld(FATAL_SYS, "Fehler bei lseek");

STOPPUHR_EIN

while ( (n = read(von_fd, puffer, PUFF_GROESSE)) > 0)
    if (write(nach_fd, puffer, n) != n)
        fehler_meld(FATAL_SYS, "Fehler bei write");
    if (n < 0)
        fehler_meld(FATAL_SYS, "Fehler bei read");

STOPPUHR_AUS

zeit_ausgabe("read/write", uhr_ende-uhr_start, &start_zeit, &ende_zeit);
}

/*---- Ausgeben der verbrauchten Zeiten -----*/
static void zeit_ausgabe(char *wovon, clock_t realzeit,
                        struct tms *start_zeit, struct tms *ende_zeit)
{
    static long    ticks=0;

    if (ticks == 0)
        if ( (ticks = sysconf(_SC_CLK_TCK)) < 0)
            fehler_meld(FATAL_SYS, "Fehler bei sysconf");

    fprintf(stderr, "| %13s | %10.21f | %10.21f | %10.21f |\n", wovon,
            (ende_zeit->tms_utime - start_zeit->tms_utime) / (double)ticks,
            (ende_zeit->tms_stime - start_zeit->tms_stime) / (double)ticks,
            realzeit / (double)ticks);
    fprintf(stderr, "+-----+-----+-----+-----+\n");
    return;
}

```

Programm 15.4 (**mmap.c**): Kopieren einer Datei mit Memory Mapped I/O und mit **read/write**

In der Funktion `mmap_copy`, die das Kopieren mit Memory Mapped I/O durchführt, wird zunächst mit **fstat** die Größe der zu kopierenden Datei ermittelt. Diese Größe wird benötigt, um die Größe `n` der Zielfeile festzulegen: **lseek** auf das Byte `n-1` und anschließendes Schreiben (**write**) eines Bytes. Wenn man die Größe der Zielfeile nicht auf diese Art festlegt, dann würde zwar der **mmap**-Aufruf für die Zielfeile erfolgreich sein, aber der erste Zugriff auf diesen mapped-Speicherbereich würde das Signal `SIGBUS` generieren.

Nach dem Festlegen der Größe der Zielfeile, wird **mmap** zweimal aufgerufen: einmal für die Quelldatei und einmal für die Zielfeile. Das Umkopieren erfolgt dann mit einem Aufruf von **memcpy**.

Nachdem man dieses Programm 15.3 (`mmap.c`) kompiliert und gelinkt hat

```
cc -o mmap mmap.c fehler.c
```

ergibt sich z.B. auf einem 486er, der unter SOLARIS 2.1 läuft, für das Kopieren von 4 Megabyte folgende Ausgabe.

```
$ mmap quelldatei zieldatei
+-----+-----+-----+-----+
|           | UserCPU | SystemCPU | Gebrauchte |
|           |   (Sek) |   (Sek)   |   Uhrzeit  |
+-----+-----+-----+-----+
| mmap/memcpy |    0.48 |    1.35   |    1.83    |
+-----+-----+-----+-----+
| read/write  |    0.06 |    2.22   |    3.63    |
+-----+-----+-----+-----+
$
```

Memory Mapped I/O ist also beim Kopieren von normalen Dateien schneller als das Kopieren mit elementaren oder Standard-E/A-Funktionen.

Memory Mapped I/O kann jedoch nicht verwendet werden, um zwischen zwei verschiedenen Gerädateien zu kopieren. Auch ist Vorsicht angesagt, wenn sich die Größe der Datei ändert, für die Memory Mapped I/O eingerichtet wurde.

Nichtsdestoweniger kann Memory Mapped I/O in vielen Fällen die Ablaufgeschwindigkeit von Programmen erhöhen und Programme auch vereinfachen, da man bei der E/A im Speicher arbeitet und sich nicht mühsam mit den einzelnen Dateioperationen abgeben muß. In Kapitel 18.4 werden wir nochmals auf Memory Mapped I/O zurückkommen, wenn das sogenannte *shared Memory* vorgestellt wird.

15.4 Weitere read- und write-Funktionen

Zu den elementaren Funktionen **read** und **write** existieren weitere Varianten, die sich im Namen und ihrer Funktionsweise etwas von diesen unterscheiden. Hier werden zwei **read** / **write**-Varianten vorgestellt.

15.4.1 readv und writev – Gleichzeitiges Lesen und Schreiben mit mehreren Puffern

Um mit nur einem Funktionsaufruf aus mehreren nicht zusammenhängenden Puffern zu lesen oder in solche Puffer zu schreiben, stehen die beiden POSIX.1 Funktionen **readv** und **writev** zur Verfügung.

```
#include <sys/types.h>
#include <sys/uio.h>
```

```
ssize_t readv(int fd, const struct iovec iov[], int iovanzahl);
ssize_t writev(int fd, const struct iovec iov[], int iovanzahl);
```

beide geben zurück: Anzahl der gelesenen / geschriebenen Bytes (bei Erfolg); -1 bei Fehler

Bei beiden Funktionen ist das zweite Argument die Adresse eines Arrays, dessen Elemente den Datentyp `struct iovec` haben:

```
struct iovec {
    void *iov_base; /* Anfangsadresse des Puffers */
    size_t iov_len; /* Größe des Puffers */
}
```

Das Argument `iovanzahl` legt fest, wie viele Elemente das Array `iov` hat. Abbildung 15.5 verdeutlicht die einzelnen Argumente dieser beiden Funktionen.

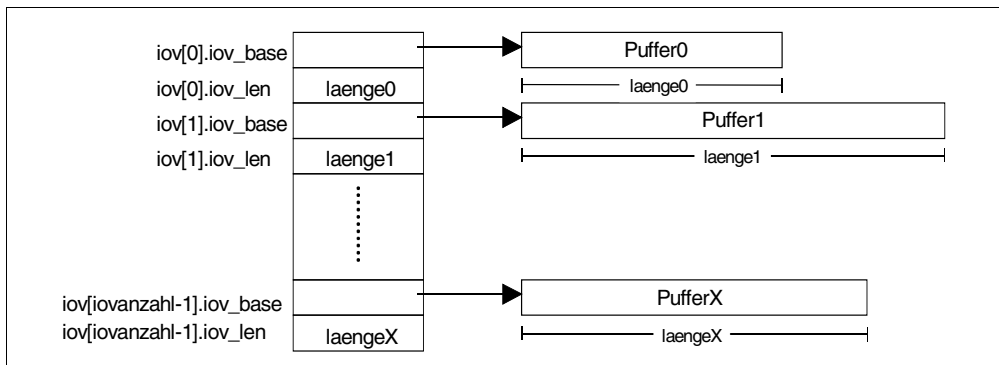


Abbildung 15.5: Array `iov` bei **readv** und **writev**

writev

Beim Schreiben mit **writev** werden nacheinander die Daten aus den Puffern `iov[0]`, `iov[1]`, ..., `iov[iovanzahl-1]` gesammelt und dann mit einem Schreibzugriff in die entsprechende Datei geschrieben. Es liegt nahe, daß ein **writev**-Aufruf somit schneller ist als mehrere **write**-Aufrufe, mit denen die einzelnen Puffer in eine Datei geschrieben werden.

Die von **writev** zurückgegebene Anzahl von geschriebenen Bytes sollte normalerweise gleich der Summe aller Pufferlängen sein. Um z.B. einen Vor- und Nachnamen einer Person mit Leerzeichen getrennt in eine Datei zu kopieren, kann das folgende Codestück angegeben werden:

```
char vorname[..], nachname[..], leerzeichen[] = " ";
....
....
iov[0].iov_base = vorname;
iov[0].iov_len = strlen(vorname);
iov[1].iov_base = leerzeichen;
iov[1].iov_len = 1;
iov[2].iov_base = nachname;
iov[2].iov_len = strlen(nachname);

if (writev(fd, &iov[0], 3) !=
    iov[0].iov_len + iov[1].iov_len + iov[2].iov_len)
    fehler_meld(FATAL_SYS, "writev-Fehler");
```

readv

Beim Lesen mit **readv** werden mit einem Lesezugriff die entsprechenden Daten aus der Datei gelesen und dann nacheinander auf die einzelnen Puffer verteilt.

Es ist offensichtlich, daß ein **readv**-Aufruf schneller ist als mehrere aufeinanderfolgende **read**-Aufrufe, um die einzelnen Puffer mit Daten aus der entsprechenden Datei zu füllen.

readv liefert 0 als Rückgabewert, wenn beim Lesen das Dateiende erreicht wurde und keine weiteren Daten mehr vorhanden sind.

Hinweis

In BSD-Unix legt die Konstante `UIO_MAXIOV` den maximalen Wert für `iovanzahl` fest. Momentan ist dieser Wert auf 1024 festgelegt.

In SVR4 legt die Konstante `UIO_MAX` den maximalen Wert für `iovanzahl` fest.

writev könnte man auch selbst nachbilden, indem man zunächst einen Puffer allokiert, dessen Größe gleich der Summe aller einzelnen Puffergrößen ist. Nachdem man dann die einzelnen Puffer in diesen großen Puffer kopiert hat, schreibt man den Puffer mit einem **write** in die entsprechende Datei. Ein Nachbilden von **readv** ist entsprechend auch möglich. Beide Nachbildungen sind aber umständlicher als ein direkter Aufruf von **writev** und **readv**.

15.4.2 Besonderes Lesen und Schreiben auf speziellen Geräten

Einige Geräte, wie z.B. Terminals, Netzwerke oder STREAMS in SVR4 haben zwei Besonderheiten:

1. Ein **read**-Aufruf kann bei solchen Geräten eventuell weniger Bytes liefern als angefordert wurden. Selbst wenn dabei nicht EOF aufgetreten ist, bedeutet dies nicht zwangsläufig einen Fehler, und das Lesen von diesem Gerät sollte fortgesetzt werden.
2. Beim Schreiben mit **write** auf solche Geräte kann es eventuell vorkommen, daß weniger Bytes geschrieben werden als gefordert. Ein solches unvollständiges Schreiben kann nur bei nicht-blockierenden Filedeskriptoren oder beim Abfangen eines Signals auftreten. Es weist jedenfalls nicht zwangsläufig auf einen Fehler hin und das Schreiben der restlichen Daten sollte auf jeden Fall fortgesetzt werden.

Beim Lesen und Schreiben auf externe Speichermedien (wie z.B. Festplatte oder Diskette) kann ein solches unvollständiges, aber fehlerfreies Lesen/Schreiben niemals auftreten.

Für Geräte, bei denen ein solches unvollständiges, aber richtiges Lesen/Schreiben möglich ist, empfiehlt es sich, eigene Funktionen zu schreiben, die mit dieser besonderen Form der Ein-/Ausgabe umgehen können, wie z.B. **readspez** und **writespez**.

```
#include "eighdr.h"
```

```
ssize_t readspez(int fd, void *puffer, size_t bytezahl);
```

```
ssize_t writespez(int fd, void *puffer, size_t bytezahl);
```

beide geben zurück: Anzahl der gelesenen bzw. geschriebenen Bytes (bei Erfolg); -1 bei Fehler

Diese beiden Funktionen rufen **read** und **write** solange auf, bis alle geforderten Bytes (bytezahl) gelesen oder geschrieben wurden.

Während man **writespez** bei jedem Schreiben auf diese speziellen Geräte verwenden sollte, sollte **readspez** nur dann aufgerufen werden, wenn man in jedem Fall genau bytezahl Bytes von einem solchen Gerät lesen möchte. Bei anderen Lesevorgängen, bei denen unbekannt ist, wie viele Bytes zu lesen sind, sollte man **read** verwenden. Programm 15.4 (readwrit.c) zeigt eine mögliche Implementierung der beiden Funktionen **readspez** und **writespez**.

```
#include "eighdr.h"
```

```
ssize_t readspez(int fd, void *puffer, size_t bytezahl)
```

```
{
    size_t    noch_zulesen = bytezahl, n;
    char      *zgr = puffer;

    while (noch_zulesen > 0) {
        if ( (n = read(fd, zgr, noch_zulesen)) < 0)
            return(n); /* Fehler: Rueckgabe n */

        else if (n == 0)
            return(bytezahl - noch_zulesen); /* EOF: Rueckgabe >=0 */

        noch_zulesen -= n;
        zgr += n;
    }
    return(bytezahl - noch_zulesen); /* geforderte Bytes wurden gelesen */
}
```

```
ssize_t writespez(int fd, void *puffer, size_t bytezahl)
```

```
{
    size_t    noch_zuschreiben = bytezahl, n;
    char      *zgr = puffer;

    while (noch_zuschreiben > 0) {
        if ( (n = write(fd, zgr, noch_zuschreiben)) <= 0)
            return(n); /* Fehler: Rueckgabe n */

        noch_zuschreiben -= n;
        zgr += n;
    }
}
```

```

    }
    return(bytezahl);
}

```

Programm 15.5 (**readwrit.c**): Mögliche Implementierung der Funktionen **readspez** und **writespez**

15.5 Übung

15.5.1 Gegenüberstellung der Signalmengen- und Deskriptormengenfunktionen

Wenn man die Signalmengenfunktionen (aus Kapitel 13.4) und die Deskriptormengenfunktionen (aus Kapitel 15.1) einander gegenüberstellt, so lassen sich Ähnlichkeiten und auch kleine Unterschiede erkennen. Erstellen Sie eine kleine Tabelle, bei der sie die Funktionen, die ähnliches in der jeweiligen Menge leisten, paarweise gruppieren. Was sind die kleinen Unterschiede zwischen diesen beiden Funktionsgruppen?

15.5.2 Ändern der Limits für Deskriptormengen

Die Headerdatei `<sys/types.h>` enthält meist eine Konstante, die eine maximale Anzahl von Filedeskriptoren festlegt, die der Datentyp `fd_set` aufnehmen kann. Wie könnte man dieses Limit wenn nötig erhöhen oder erniedrigen?

15.5.3 Ermitteln der Kapazität einer Pipe mit `select` oder `poll`

Erstellen Sie ein Programm `pipgroes.c`, das unter Verwendung der Funktion ***select*** oder der Funktion ***poll*** die Kapazität einer Pipe ermittelt und ausgibt. Um eine Pipe einzurichten, muß folgender Code verwendet werden.

```

int  fd[2]; /* 2 Filedeskriptoren:
            einer zum Lesen aus Pipe (fd[0]) und
            einer zum Schreiben in die Pipe (fd[1]) */
.....

if (pipe(fd) < 0)
    fehler_meld(FATAL_SYS, "kann keine Pipe einrichten");
..... /* Zum Lesen aus der Pipe muß nun Filedeskriptor fd[0] und
        zum Schreiben in die Pipe muß Filedeskriptor fd[1]
        verwendet werden                                     */

```

15.5.4 Zahlenwurzeln in den mapped-Speicherbereich schreiben und wieder lesen

Erstellen Sie ein Programm `zahlmmap.c`, das zu einem ganzzahligen Zahlenbereich, der auf der Kommandozeile anzugeben ist, alle dazwischenliegenden Quadratwurzeln berechnet und als Strings in einen memory-mapped-Speicherbereich (über Array) schreibt. Danach soll der Benutzer sich Teilbereiche aus diesem memory-mapped-Speicherbereich ausgeben lassen können.

Nachdem man dieses Programm `zahlmmap.c` kompiliert und gelinkt hat.

```
cc -o zahlmmap zahlmmap.c fehler.c -lm
```

ergibt sich z.B. der folgende Ablauf:

```
$ zahlmmap 100000 200000
```

```
Ausgabe von? 100010
```

```
Ausgabe bis? 100019
```

```
-----
100010: 316.243577
100011: 316.245158
100012: 316.246739
100013: 316.248320
100014: 316.249901
100015: 316.251482
100016: 316.253063
100017: 316.254644
100018: 316.256225
100019: 316.257806
-----
```

```
Ausgabe von? 199100
```

```
Ausgabe bis? 199107
```

```
-----
199100: 446.206230
199101: 446.207350
199102: 446.208471
199103: 446.209592
199104: 446.210712
199105: 446.211833
199106: 446.212953
199107: 446.214074
-----
```

```
Ausgabe von? 0
```

```
$
```

15.5.5 E/A-Multiplexing für das Lesen aus zwei Pipes

Erstellen Sie ein Programm `ea_multi.c`, das abwechselnd die anstehenden Daten aus zwei Pipes `pipe1` und `pipe2` liest. Diese beiden Pipes sollten Sie zunächst mit den beiden folgenden Kommandos anlegen.

```
$ mknod pipe1 p
```

```
$ mknod pipe2 p
```

```
$
```

Daß es sich dabei um zwei Pipes handelt, kann man mit dem Kommando `ls` prüfen:

```
$ ls -l pipe*
```

```
prw-r--r--  1 hh      users      0 Dec 29 12:54 pipe1
```

```
prw-r--r--  1 hh      users      0 Dec 29 12:54 pipe2
```

```
$
```


Das Programm `ea_multi` soll dem Benutzer über Optionen festlegen lassen, wie aus den beiden Pipes zu lesen ist:

```
$ ea_multi
usage: ea_multi -b | -n | -s
    -b   blockierendes Lesen aus zwei Pipes
    -n   nichtblockierendes Lesen aus zwei Pipes
    -s   select verwenden zum Lesen aus zwei Pipes
$
```

Zum Testen des Programms `ea_multi` sollten entweder drei X-Terminals oder aber drei virtuelle Konsolen verwendet werden, wobei auf den einzelnen X-Terminals (virtuellen Konsolen) folgende Kommandos einzugeben sind:

X-Terminal (bzw. virtuelle Konsole) 1:

```
$ cat > pipe1
  [Hier kann nun Text eingegeben werden]
$
```

X-Terminal (bzw. virtuelle Konsole) 2:

```
$ cat > pipe2
  [Hier kann nun Text eingegeben werden]
$
```

X-Terminal (bzw. virtuelle Konsole) 3:

```
$ ea_multi -b oder ea_multi -n oder ea_multi -s
  [Hier sollte der an den beiden X-Terminals (virtuellen Konsolen)
   eingegebene Text erscheinen]
$
```

Bei der Angabe der Option **-b** sollte unabhängig davon, wie viele Zeilen an den jeweiligen X-Terminals (virtuellen Konsolen) 1 oder 2 eingegeben werden, immer nur eine Zeile am dritten X-Terminal (virtuelle Konsole) ausgegeben werden, da hier immer aufgrund des blockierenden Lesens (**read**) abwechselnd eine Zeile aus der nächsten Pipe gelesen werden muß.

Diese alternierende Ausgabe von Zeilen der beiden Pipes sollte bei den Optionen **-n** und **-s** nicht gelten. Bei diesen beiden Optionen sollte der vollständige Text (auch eventuell mehrere Zeilen), der in einem der beiden X-Terminals (virtuellen Konsole) 1 bzw. 2 eingegeben wird, sofort am dritten X-Terminal (virtuelle Konsole) erscheinen, da hier ein nachfolgendes **read** auf die andere Pipe nicht blockiert wird und somit sofort wieder ein **read** auf die aktuelle Pipe (aktuelles X-Terminal bzw. aktuelle virtuelle Konsole) erfolgt.

16 Dämonprozesse

*Dämonen, weiß ich, wird man schwerlich los,
Das geistig-strenge Band ist nicht zu trennen ...*

Goethe

Dämonprozesse (*daemons*) sind Prozesse, die ständig im Hintergrund ablaufen. Sie werden üblicherweise beim Booten des Systems gestartet und werden erst beim »Herunterfahren« oder Absturz dieses Systems beendet. Dämonprozesse sind für ständig anfallende Aufgaben zuständig, wie z.B. Überprüfen, ob neue Post (*mail*) angekommen ist.

Dieses Kapitel gibt zunächst einen Überblick über typische Unix-Dämonen und deren Besonderheiten, und zeigt dann, wie ein Dämonprozeß zu schreiben ist. Da ein Dämonprozeß im Hintergrund abläuft und somit auch kein Kontroll-Terminal besitzt, wird zusätzlich gezeigt, wie ein Dämonprozeß trotzdem das Auftreten von Fehlern melden kann.

16.1 Typische Unix-Dämonen

Hier werden einige Unix-Dämonen, die auf den meisten Systemen ablaufen, kurz vorgestellt.

16.1.1 syslogd – Dämon für Fehlermeldungen

Dieser Dämon ermöglicht es jedem Programm, Systemmeldungen (für den Administrator) auf der Konsole auszugeben. Diese Meldungen werden zusätzlich in der Log-Datei eingetragen. Darauf wird genauer in Kapitel 16.3 eingegangen.

16.1.2 sendmail – Mail-Dämon

sendmail ist der Standard-Mail-Dämon. Er ist dafür verantwortlich, um *mail* (elektronische Post) zu empfangen oder zu senden.

16.1.3 update – Dämon zum regelmäßigen Schreiben des Puffer-Caches auf Festplatte

Der **update**-Dämon ist für den regelmäßigen *Update* der Festplatte zuständig. Er schreibt unter Verwendung der **sync**-Funktion (siehe Kapitel 5.11) in regelmäßigen Abständen (meist 30 Sekunden) den Inhalt des Puffer-Cache auf Festplatte.

16.1.4 cron – Dämon zum regelmäßigen Ausführen von Kommandos

Der **cron**-Dämon prüft in bestimmten Zeitabständen (Voreinstellung ist meist eine Minute) den Inhalt der sogenannten *crontab*-Dateien. *crontab*-Dateien legen die Zeitpunkte fest, zu denen entsprechende Kommandos automatisch ablaufen sollen. Der Benutzer kann unter Verwendung des Kommandos **crontab**, seine eigene *crontab*-Dateien anlegen¹.

16.1.5 inetd – Netz-Dämon

Der **inetd**-Dämon überprüft ständig die Netzwerkanschlüsse auf das Ankommen neuer Netzanforderungen (siehe auch Kapitel 11.1).

16.1.6 lpd – Drucker-Dämon

Der **lpd**-Dämon steuert das Druckersystem, indem er Druckaufträge entgegennimmt oder auf Anforderung wieder löscht.

16.2 Besonderheiten von Dämonen

Um die Unterschiede von Dämonprozessen gegenüber anderen Prozessen zu erfahren, empfiehlt sich der Aufruf des **ps**-Kommandos, das den Status von verschiedenen Prozessen ausgibt. **ps** verfügt über eine Vielzahl von Optionen. Für die Ausgabe von Dämonprozessen empfehlen sich die folgenden Aufrufe

```
ps -axj (4.4BSD, SOLARIS, Linux)
ps -efjc (SVR4)
```

Die Ausgabe hierfür kann z.B. sein

```
$ ps -axj
PPID  PID  PGID  SID  TTY  TPGID  STAT  UID  TIME  COMMAND
    0     1     1     1  ?    -1  S      0   0:05  init auto ro
    1     6     2     2  ?    -1  S      0   0:00  bdflush (daemon)
    1     7     2     2  ?    -1  S      0   0:00  update (bdflush)
    1    48    20    20  ?    -1  S      0   0:00  /usr/sbin/rpc.mountd
    1    52    20    20  ?    -1  S      0   0:00  /usr/sbin/rpc.pcnfsd /var
```

1. Siehe auch Band »Linux-Unix-Grundlagen« in dieser Buchreihe.

```

1  32  32  32  ?  -1 S    0  0:01 /usr/sbin/syslogd
1  36  36  36  ?  -1 S    0  0:00 /usr/sbin/klogd
1  38  38  38  ?  -1 S    0  0:00 /usr/sbin/rpc.portmap
1  40  40  40  ?  -1 S    0  0:00 /usr/sbin/inetd
1  43  43  43  ?  -1 S    0  0:00 /usr/sbin/lpd
43 45  45  43  ?  -1 S    0  0:00 /usr/sbin/lpd
1  46  46  20  ?  -1 S    0  0:00 /usr/sbin/crond
1  50  50  50  ?  -1 S    0  0:00 /usr/sbin/rpc.nfsd
1  57  57  57 v01 57 S    0  0:01 -tcsh
1  58  58  58 v02 396 S   2021 0:01 -tcsh
1  59  59  59 v03 59 S    0  0:00 /sbin/agetty 38400 tty3
1  60  60  60 v04 60 S    0  0:00 /sbin/agetty 38400 tty4
58 396 396 58 v02 396 R   2021 0:00 ps -axj
$

```

An dieser Ausgabe, die unter Linux erhalten wurde, ist erkennbar, daß alle Dämonprozesse Superuser-Rechte (UID==0) haben und ihr Elternprozeß der `init`-Prozeß ist. Keinem der Dämonprozesse ist ein Kontrollterminal zugeordnet, was an dem Fragezeichen erkennbar ist. Auch zeigt diese Ausgabe, daß die Terminalvordergrundprozeßgruppe von Dämonprozessen gleich -1 ist.

16.3 Schreiben von eigenen Dämonen

Beim Schreiben von Dämonen gilt es, die folgenden Regeln zu beachten:

1. Aufrufen von `fork` und anschließendes Beenden des Elternprozesses

Als erstes muß **`fork`** aufgerufen werden, und dann muß der Elternprozeß sich mit **`exit`** beenden. So erreicht man beim Aufruf des Dämons in einer Shell, daß diese Shell annimmt, das Kommando habe sich beendet, und sie so den Benutzer mit der Ausgabe des Promptzeichens weitere Kommandos eingeben läßt.

Da der Kindprozeß die Prozeßgruppen-ID vom Elternprozeß erbt, seinerseits aber eine neue Prozeß-ID erhält, ist sichergestellt, daß der Kindprozeß nicht ein Prozeßgruppenführer ist, was die Voraussetzung für den nächsten Schritt (Aufruf von **`setsid`**) ist.

2. Aufrufen von `setsid`

Mit einem **`setsid`**-Aufruf wird eine neue Session kreiert, was folgende Konsequenzen hat:

- ▶ Der Prozeß wird Sessionführer der neuen Session.
- ▶ Der Prozeß wird Prozeßgruppenführer der neuen Prozeßgruppe.
- ▶ Der Prozeß hat kein Kontrollterminal.

3. Wechseln ins Root-Directory oder in ein spezielles Directory

Da das vom Elternprozeß geerbte Directory eventuell ein montiertes Filesystem sein könnte, empfiehlt es sich, ins Root-Directory zu wechseln. Der Grund dafür liegt in der Tatsache, daß beim Herunterfahren eines Systems ein Filesystem, auf dem noch ein Prozeß (in diesem Fall der Dämon) läuft, nicht demontiert werden kann.

Manche Dämonprozesse wechseln in ein spezielles Directory, in dem sie ihre Aktionen durchführen, wie z.B. der **lpd**-Dämon, der meist ins Spool-Directory wechselt.

4. Setzen der Dateikreierungsmaske auf 0

Der Kindprozeß (Dämon) erbt die Dateikreierungsmaske von seinem Elternprozeß. Damit der Dämon für seine kreierten Dateien auch genau die Zugriffsrechte erhält, die er fordert, ohne daß diese durch die geerbte Dateikreierungsmaske umgeändert werden, sollte er die Dateikreierungsmaske löschen (auf 0 setzen).

5. Schließen von nicht mehr benötigten Filedeskriptoren

Der Dämon sollte die vom Elternprozeß geerbten offenen, aber nicht benötigten Filedeskriptoren schließen. Es hängt natürlich vom jeweiligen Dämon ab, welche Filedeskriptoren zu schließen sind. Wenn ein Dämon alle geerbten Filedeskriptoren schließen möchte, kann er mit `sysconf(_SC_OPEN_MAX)` die Nummer des höchsten Filedeskriptors ermitteln und dann entsprechend alle Filedeskriptoren schließen.

16.3.1 Umwandeln eines normalen Prozeß in einen Dämonprozeß

Das nachfolgende Programm 16.1 (`daemon.c`) zeigt eine Funktion `daemonisierung`, die den Aufrufer dieser Funktion zu einem Dämonprozeß macht.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "eighdr.h"

int daemonisierung(void)
{
    pid_t    pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0); /* Elternprozess beendet sich */

    /*---- Ab hier wird nur vom Kindprozess ausgeführt */
    setsid(); /* Kind wird Session-Fuehrer */

    chdir("/"); /* Wechseln zum Root-Directory */

    umask(0); /* Dateikreierungsmaske loeschen */

    return(0);
}
```

Programm 16.1 (**daemon.c**): Dämonisierung eines normalen Prozesses

Die Routine `daemonisierung` wollen wir mit dem folgenden einfachen Programm 16.2 (`daemtest.c`) testen.

```
#include "eighdr.h"

int
main(void)
{
    if (daemonisierung() != 0)
        fehler_meld(FATAL_SYS, "Daemonisierung nicht moeglich");
    printf(".... Ich bin jetzt ein Daemon .....\n");

    sleep(20);

    printf(".... Ich verabschiede mich nun als Daemon ....\n");
}
```

Programm 16.2 (**`daemtest.c`**): Testen der Routine `daemonisierung`

Nachdem man dieses Programm 16.2 (`daemtest.c`) kompiliert und gelinkt hat

```
cc -o daemtest daemtest.c daemon.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ daemtest
$ .... Ich bin jetzt ein Daemon .....
ps -xj
PPID  PID  PGID  SID  TTY  TPGID  STAT  UID  TIME  COMMAND
    1   58   58   58  v02   303   S    2021  0:02  -tcsh
    1  302  302  302  ?     -1    S    2021  0:00  daemtest
   58  303  303   58  v02   303   R    2021  0:00  ps -xj
$ .... Ich verabschiede mich nun als Daemon .... [Nach 20 Sekunden]
```

16.4 Fehlermeldungen von Dämonen

Da Dämonen kein Kontrollterminal haben, können sie ihre Meldungen nicht einfach auf die Standardfehlerausgabe ausgeben. Auch ist es sicher nicht erwünscht, daß alle Dämonen ihre Meldungen an der Systemkonsole ausgegeben, was sehr störend für den Systemadministrator wäre.

Ebenso ist es nicht klug, daß jeder Dämon seine Fehlermeldung in eine eigene Log-Datei schreibt. Die Überprüfung der einzelnen Log-Dateien durch den Systemadministrator würde dann sehr aufwendig sein.

Was man braucht, ist eine zentrale Einrichtung, die für Meldungen von Dämonen verantwortlich ist. Nachfolgend wird für SVR4 und BSD-Unix diese zentrale Einrichtung und ihre Struktur vorgestellt.

16.4.1 log – STREAMS-Gerätetreiber in SVR4

SVR4 stellt einen STREAMS-Gerätetreiber mit dem Namen **log** zur Verfügung (siehe auch Manpage `log(7)`). **log** bietet Schnittstellen für:

- ▶ *error-logging* (normale Eintragung von Fehlermeldungen)
- ▶ *STREAMS event tracing* (Mitverfolgung von Ereignissen)
- ▶ *console logging* (Fehlermeldungen auf Konsole, in Dateien oder als mail)

Abbildung 16.1 veranschaulicht die Gesamtstruktur der **log**-Einrichtung unter SVR4.

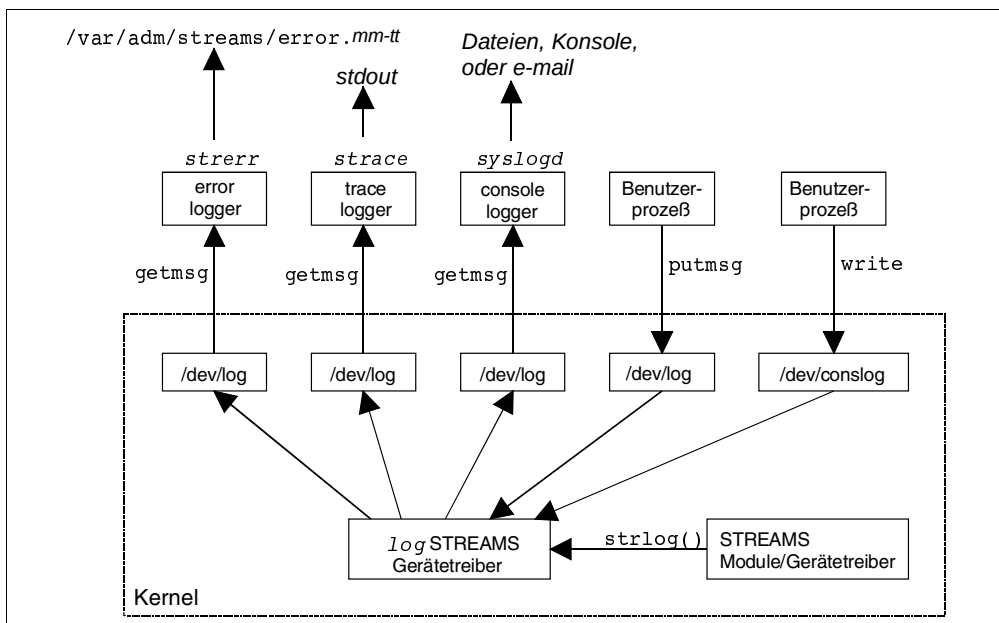


Abbildung 16.1: log-Einrichtung unter SVR4

Das Generieren von Messages ist auf drei verschiedene Arten möglich:

1. Routinen im Kern können **strlog** aufrufen, um **log**-Messages zu erzeugen. Diese Möglichkeit wird normalerweise von STREAMS-Modulen oder -Gerätetreibern benutzt.
2. Ein Benutzerprozeß (Dämon) kann mit **putmsg** eine Message in den STREAM `/dev/log` schreiben. Diese Message kann er an einen der drei Logger (*error*, *trace*, *console*) schicken.
3. Ein Benutzerprozeß (Dämon) kann mit **write** eine Message in den STREAM `/dev/conslog` schreiben. Diese Message kann nur an den *console logger* geschickt werden.

Das Lesen von **log**-Messages ist auf drei verschiedene Arten möglich:

1. Der normale Error Logger ist **strerr**, der als Dämon im Hintergrund abläuft. Er schreibt die Messages an das Ende der Datei `/var/adm/streams/error.mm-tt`, wobei `mm` der Monat und `tt` der Tag des Monats ist. **strerr** ist im übrigen selbst ein Dämon.
2. Der normale Trace Logger ist **strace**. Er kann sogenannte *trace*-Messages auf seine Standardausgabe schreiben.
3. Der Console Logger ist **syslogd**, der von BSD übernommen wurde. Dieses Programm **syslogd** ist ein Dämon, der eine Konfigurationsdatei liest und die **log**-Messages entweder in bestimmte Dateien oder auf die Konsole schreibt oder aber als Mail an bestimmte Benutzer schickt. **syslogd** wird im nächsten Abschnitt noch genauer beschrieben.

Neben der eigentlichen Message enthält eine **log**-Message weitere Information. Wenn z.B. eine Message vom **log**-Gerätetreiber »aufwärts« geschickt wird, so enthält sie zusätzliche Information über den Urheber (wenn durch **strlog** generiert), über Priorität, über Entstehungszeit usw. Die Manpage `log(7)` enthält hierzu eine genaue Beschreibung. Während bei der Verwendung von **putmsg** einige dieser Informationen vom Aufrufer festgelegt werden können, kann bei einem **write** auf den Console Logger (über `/dev/console`) nur die reine Message ohne jegliche Zusatzinformationen geschickt werden.

Eine andere in Abbildung 16.1 nicht gezeigte Möglichkeit ist der Aufruf der BSD-Funktion **syslog** durch einen Dämon in SVR4. In diesem Fall wird die Message ähnlich wie bei einem **putmsg** (auf `/dev/log`) zum Console Logger geschickt. Die Funktion **syslog** ist im nächsten Abschnitt beschrieben.

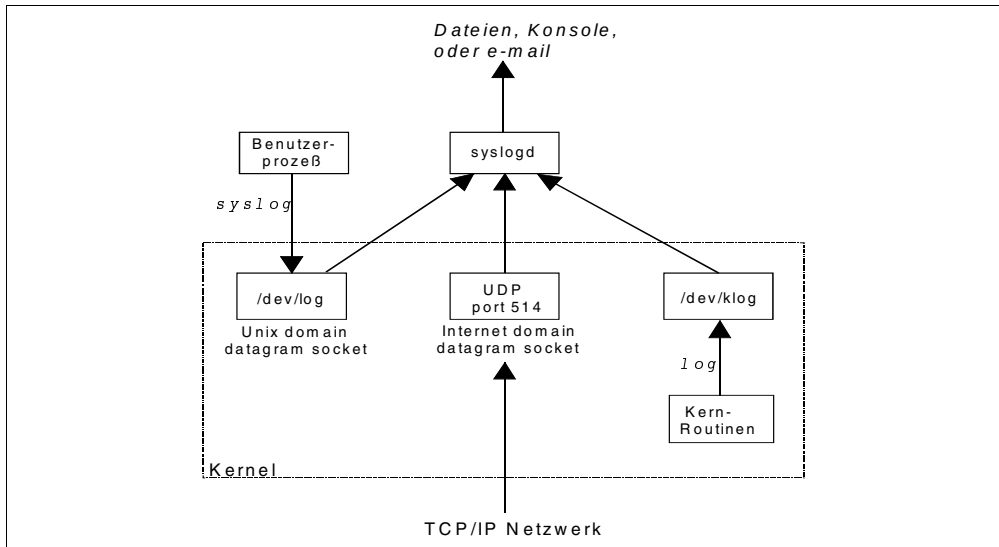
Hinweis

Wenn der entsprechende Logger zum Zeitpunkt, an dem eine *Log Message* für diesen generiert wird, nicht läuft, so geht diese Message verloren.

Die Funktion **syslog** und der Dämon **syslogd** befinden sich in der Standard-C-Bibliothek und können somit in SVR4 von allen Benutzerprozessen verwendet werden.

16.4.2 syslog – Error-Logging in BSD

BSD stellt zum Error-Logging die **syslog**-Einrichtung zur Verfügung. Die meisten Dämonen verwenden diese Einrichtung zum Error-Logging. Abbildung 16.2 zeigt die Struktur dieser Einrichtung.

Abbildung 16.2: Die **syslog**-Einrichtung von BSD-Unix

Es gibt drei Möglichkeiten, um Log-Messages zu generieren:

1. Die Kernroutinen können die Funktion **log** aufrufen. Diese Messages können von jedem Benutzerprozeß aus der Gerätedatei `/dev/klog` mit **read** gelesen werden, nachdem er diese Datei mit **open** geöffnet hat.
2. Die meisten Benutzerprozesse (Dämonen) rufen die Funktion **syslog** auf, um Log-Messages zu generieren. Ein solcher Aufruf bewirkt, daß die Message an das *Unix Domain Datagram Socket* `/dev/log` geschickt wird. Die Funktion **syslog** wird weiter unten genauer beschrieben.
3. In einem TCP/IP-Netzwerk kann jeder Benutzerprozeß Log-Messages auf das *UDP port 514* schicken. Dies ist jedoch nicht mit der Funktion **syslog** möglich, dazu ist vielmehr Netzwerkprogrammierung notwendig.

In Kapitel 19.7 dieses Buches und in »*Unix Network Programming*, Stevens, W.R., 1990, Prentice-Hall« sind *Unix Domain Datagram Sockets* und *UDP Sockets* detailliert beschrieben.

Der Dämon **syslogd** liest beim Systemstart eine Konfigurationsdatei, normalerweise `/etc/syslog.conf`. Diese Konfigurationsdatei legt fest, wohin die verschiedenen Message-Klassen geschickt werden sollen. So könnten z.B. dringende Messages dem Systemadministrator auf der Systemkonsole ausgegeben werden und zur Sicherheit noch über Mail zugeschickt werden, während Warnungen nur in einer Log-Datei geschrieben werden.

Zur Verwendung der **syslog**-Einrichtung stehen drei Funktionen zur Verfügung.

```
#include <syslog.h>

void openlog(char *kennung, int option, int facility);

void syslog(int priorität, char *format, ...);

void closelog(void);
```

openlog

Der Aufruf von **openlog** ist optional. Wird **openlog** nicht aufgerufen, so wird diese Funktion automatisch beim ersten Aufruf von **syslog** aufgerufen.

Das Argument *kennung* erlaubt die Angabe eines Strings, der zu jeder Log-Message hinzugefügt wird. Normalerweise wird für *kennung* der Programmname (z.B. **cron**, **inetd** usw.) angegeben.

Für *option* ist eine der folgenden vier Konstanten anzugeben:

LOG_CONS

Wenn Log-Message nicht über das *Unix Domain Datagram* an den **syslogd**-Dämon geschickt werden kann, so wird sie statt dessen auf der Konsole ausgegeben.

LOG_NDELAY

Sofortiges Öffnen des *Unix Domain Datagram Sockets* für den **syslogd**-Dämon. Normalerweise geschieht dieses Öffnen erst dann, wenn die erste Log-Message ansteht.

LOG_PERROR

Neben dem Schicken der Log-Message an den **syslogd**-Dämon, wird die Message zusätzlich noch auf der Standardfehlerausgabe ausgegeben.

LOG_PID

Die Log-Message soll zusätzlich noch die Prozeß-ID enthalten. Diese Option ist für Dämonen gedacht, die mittels **fork** einen Kindprozeß zur Erledigung von bestimmten Aufgaben kreieren.

Das Argument *facility* macht es möglich, in der Konfigurationsdatei für Messages von unterschiedlichen Einrichtungen auch unterschiedliche Aktionen festzulegen.

Für *facility* ist eine der folgenden Konstanten anzugeben:

LOG_AUTH

Authorisierungsprogramm: **login**, **su**, **getty**,...

LOG_CRON

cron oder **at**

LOG_DAEMON

Systemdämonen: **ftpd**, **routed**,...

LOG_KERN
vom Kern generierte Messages

LOG_LOCAL0
für lokale Zwecke reserviert

.....
.....

LOG_LOCAL7
für lokale Zwecke reserviert

LOG_LPR
Druckersystem: **lpd, lpc**,...

LOG_MAIL
Mailsystem

LOG_NEWS
Newssystem des Usenet-Netzes

LOG_SYSLOG
syslogd-Dämon selbst

LOG_USER
Messages von anderen Benutzerprozessen (Voreinstellung)

LOG_UUCP
UUCP-System

Wird **openlog** nicht aufgerufen oder für *facility* der Wert 0 angegeben, so ist bei einem späteren **syslog**-Aufruf trotzdem noch eine nachträgliche Festlegung von *facility* möglich. Die entsprechende *facility*-Angabe ist dort dann als Teil des Arguments *priorität* anzugeben.

syslog

Mit einem **syslog**-Aufruf wird eine Log-Message generiert. Das Argument *priorität* ist dabei eine Kombination aus *facility* (siehe oben) und einem sogenannten *level*.

Nachfolgend sind die verschiedenen *level*-Arten (von der höchsten bis zur niedrigsten Priorität) angegeben:

LOG_EMERG
System ist überhaupt nicht benutzbar (höchste Priorität)

LOG_ALERT
höchste Alarmstufe; Aktion muß sofort ausgeführt werden

LOG_CRIT
kritische Notfallsituation (z.B. Hardwarefehler)

LOG_ERR

Fehlersituation

LOG_WARNING

Warnung

LOG_NOTICE

normale, aber signifikante Situation

LOG_INFO

normale Information-Message

LOG_DEBUG

Debug-Message

(niedrigste Priorität)

Das angegebene *format*-Argument und eventuell weitere Argumente werden zum Formatieren der Funktion **vsprintf** übergeben. Sollte in *format* *%m* vorkommen, so wird diese Formatangabe vor dem **vsprintf**-Aufruf durch die Fehlermeldung ersetzt, deren Nummer momentan in *errno* steht (entspricht dem Rückgabe-String von **strerror**).

closelog

Der Aufruf von **closelog** ist optional. **closelog** schließt den Deskriptor, der zur Kommunikation mit dem *syslogd*-Dämon verwendet wurde.

Beispiel

Um z.B. in einem eigenen *mail*-Dämon eine **log**-Message zu generieren, können die beiden folgenden Anweisungen angegeben werden.

```
openlog("meinmail", LOG_PID, LOG_MAIL);
syslog(LOG_ERR, "kann %s nicht oeffnen: %m", dateiname);
```

Mit **openlog** wird als *kennung* der Programmname festgelegt. Zusätzlich wird dabei festgelegt, daß immer die Prozeß-ID mit auszugeben und als Einrichtung das Mailsystem zu benutzen ist.

Mit dem **syslog**-Aufruf wird die Log-Message generiert, die als normale Fehlersituation (LOG_ERR) zu behandeln ist. Ohne den **openlog**-Aufruf, muß man **syslog** wie folgt aufrufen:

```
syslog(LOG_ERR | LOG_MAIL, "kann %s nicht oeffnen: %m", dateiname);
```

Das Argument *priorität* ist dabei eine Kombination von *level* und *facility*.

Hinweis

Das sowohl von SVR4 als auch von BSD-Unix angebotene Programm **logger** ermöglicht nicht interaktiv ablaufenden Shellskripts das Generieren von Log-Messages. Dieses **logger**-Programm ermöglicht über Optionen die Angabe von *kennung*, *facility* und *level*.

Eine typische Anwendung eines Dämonprozesses ist ein Serverprozeß. Ein *Server* ist im allgemeinen ein Prozeß, der darauf wartet, daß ein sogenannter *Client*prozeß Dienstleistungen von ihm anfordert. In Abbildung 16.2 ist z.B. **syslogd** der Server, der als Dienstleistung das Loggen von Fehlermeldungen anbietet. In Abbildung 16.2 ist die Kommunikation einkanalig: Der Client fordert zwar eine Dienstleistung an, erhält aber niemals eine Antwort vom Server zurück. Bei anderen Anwendungen ist dagegen oft eine zweikanalige Kommunikation notwendig, bei der auch der Server eine Antwort zurück an den Client schicken muß.

Im nächsten Kapitel 17 werden bei der Interprozeßkommunikation hierfür Beispiele gegeben.

16.5 Übung

16.5.1 Schließen der Filedesriptoren 0, 1 und 2 durch einen Dämonprozeß

Testen Sie das folgende Programm 16.3 (*daemclo.c*) auf Ihrem System und versuchen Sie, das Verhalten zu erklären.

```
#include "eighdr.h"

#define MAX_PUFFER 100

int
main(void)
{
    char    *zgr,
           puffer[MAX_PUFFER];

    daemonisierung();
    close(0);
    close(1);
    close(2);

    zgr = getlogin();
    sprintf(puffer, "Loginname: %s\n",
            (zgr == NULL) ? "(keiner)" : zgr);
    write(3, puffer, strlen(puffer));

    exit(0);
}
```

Programm 16.3 (*daemclo.c*): Dämon schließt alle 3 Filedesriptoren und ruft dann **getlogin** auf

Nachdem man das Programm 16.3 (`daemclo.c`) kompiliert und gelinkt hat.

```
cc -o daemclo daemclo.c daemon.c
```

sollte man es wie folgt in der Bourne- oder Korn-Shell aufrufen:

```
$ daemclo 3>tmpdatei
$ cat tmpdatei
????
$
```

16.5.2 Dämon zur Überwachung von neuen Anmeldungen

Erstellen Sie ein Programm `werdaem.c`, das als Dämonprozeß ablaufen soll. Dieser Dämon soll Anmeldungen am System überwachen und neu angemeldete Benutzer immer am aktuellen Terminal ausgeben. Um das System nicht zu überlasten, sollte die Überprüfung auf neue Anmeldungen im Minutentakt erfolgen.

17 Pipes und FIFOs

*Du hast das nicht, was andre haben
und andern mangeln deine Gaben.
Aus dieser Unvollkommenheit
entspringet die Geselligkeit.*

Christian Fürchtegott Gellert

Würden keine weiteren Funktionen zur Verfügung gestellt, so wären Dateien die einzige Möglichkeit der Kommunikation zwischen unterschiedlichen Prozessen. In diesem Kapitel werden andere Techniken der Interprozeßkommunikation (*Interprocess Communication* bzw. im folgenden *IPC*) vorgestellt, die wesentlich schneller und eleganter sind als die Kommunikation über Dateien.

17.1 Überblick über die unterschiedlichen Arten der Interprozeßkommunikation

Tabelle 17.1 gibt einen Überblick über die verschiedenen IPC-Arten in den unterschiedlichen Systemen und Standards.

<i>IPC-Art</i>	<i>POSIX.1</i>	<i>XPG3</i>	<i>SVR4</i>	<i>BSD</i>
Pipes (halbduplex)	x	x	x	x
FIFOs (benannte Pipes)	x	x	x	x
STREAM Pipes			x	x
Benannte STREAM Pipes			x	x
Message-Queues		x	x	
Semaphore		x	x	
Gemeinsamer Speicher (Shared Memory)		x	x	
Sockets			x	x
STREAMS			x	

Tabelle 17.1: Überblick über die unterschiedliche IPC-Arten

Tabelle 17.1 verdeutlicht, daß Pipes (Halbduplex) und FIFOs (benannte Pipes) die einzige Art der IPC sind, die von allen Systemen und Standards unterstützt wird. Von den in Tabelle 17.1 angegebenen IPCs sind nur die letzten beiden (Sockets und STREAMS) für eine netzweite Kommunikation zwischen Prozessen auf verschiedenen Rechnern vorgesehen. Die anderen IPCs erlauben nur die Kommunikation zwischen Prozessen, die auf dem gleichen Rechner ablaufen. Obwohl in Tabelle 17.1 angegeben ist, daß *Message-Queues*, *Semaphore* und *Shared Memory* von BSD-Unix nicht unterstützt werden, werden diese drei heute auf fast allen von BSD abstammenden Systemen (wie z.B. Solaris oder Ultrix) unterstützt.

17.2 Pipes

Pipes wurden schon in älteren Unix-Systemen angeboten und waren die erste Form der IPC. Sie werden heute von allen Unix-Systemen angeboten.

Pipes besitzen grundsätzlich die beiden folgenden Eigenschaften:

1. Pipes können nur zwischen Prozessen eingerichtet werden, die einen gemeinsamen Vorfahren (Elternprozeß, Großelternprozeß usw.) haben. Sie können z.B. niemals zwischen Prozessen eingerichtet werden, die unterschiedliche Elternprozesse haben. Normalerweise wird eine Pipe von einem Elternprozeß eingerichtet, der dann seinerseits mittels **fork** einen Kindprozeß kreiert, der diese Pipe erbt, so daß dann eine Kommunikation zwischen dem Eltern- und Kindprozeß über die Pipe möglich ist.
2. Pipes sind halbduplex, was bedeutet, daß die Daten immer nur in eine Richtung fließen können. So kann z.B. ein Prozeß, der eine Pipe zum Schreiben eingerichtet hat, in diese Pipe nur schreiben, während der Prozeß auf der anderen Pipe-Seite nur aus dieser lesen kann. Soll die Kommunikation auch in umgekehrter Form stattfinden, muß hierfür eine weitere Pipe eingerichtet werden.

In diesem und späteren Kapiteln werden wir Konstruktionen kennenlernen, die diese Einschränkungen nicht haben: *FIFOs* und *benannte Stream Pipes* (haben Einschränkung 1 nicht) und *Stream Pipes* (haben Einschränkung 2 nicht).

17.2.1 pipe – Einrichten einer Pipe

Um eine Pipe einzurichten, steht die Funktion **pipe** zur Verfügung.

```
#include <unistd.h>
```

```
int pipe(int fd[2]);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Bei erfolgreichem Aufruf von **pipe** gilt folgendes:

fd[0]	geöffneter Filedescriptor zum Lesen aus der Pipe
fd[1]	geöffneter Filedescriptor zum Schreiben in die Pipe

Mit dem **pipe**-Aufruf richtet ein Prozeß eine Pipe im Kern ein (siehe Abbildung 17.1).

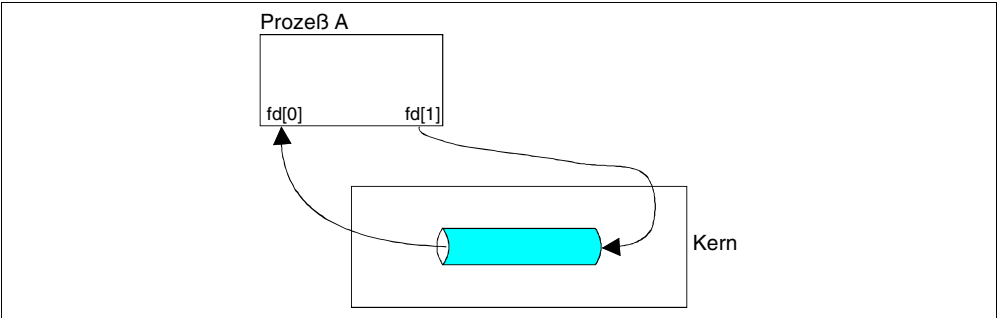


Abbildung 17.1: Eine im Kern eingerichtete Pipe nach **pipe**-Aufruf

Eine so eingerichtete Pipe ist für den Prozeß zunächst nutzlos, da kein weiterer Prozeß existiert, mit dem über diese Pipe kommuniziert werden kann. Normalerweise ruft deswegen der Prozeß, der die Pipe eingerichtet hat, nun **fork** auf, um einen Kindprozeß zu kreieren, mit dem er kommunizieren möchte. Daraus resultiert dann die in Abbildung 17.2 gezeigte Konstellation.

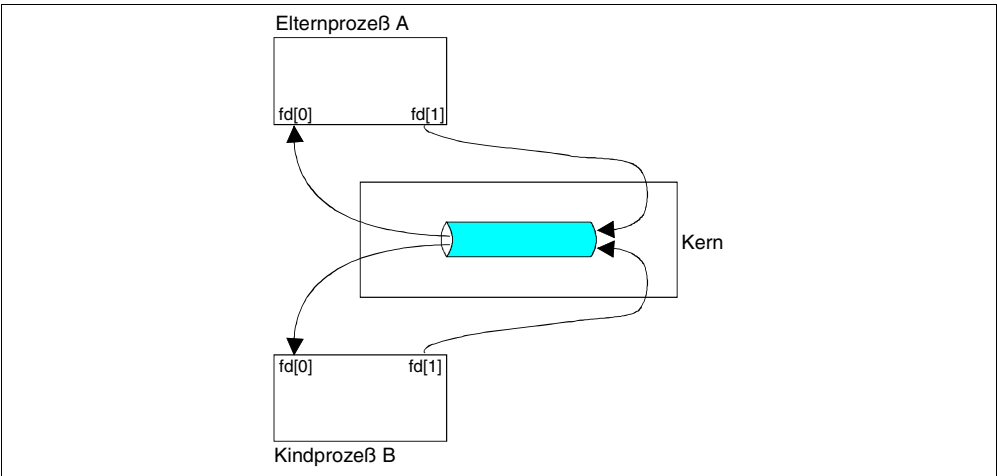


Abbildung 17.2: Konstellation nach **pipe**- und **fork**-Aufruf

Nun hängt es von den weiteren Operationen des Eltern- und Kindprozesses ab, in welche Richtung die Daten fließen. Hierfür gibt es zwei Möglichkeiten:

1. Elternprozeß schreibt und Kindprozeß liest.

Für diesen Fall muß der Elternprozeß die Leseseite der Pipe (`fd[0]`) und der Kindprozeß die Schreibseite der Pipe (`fd[1]`) schließen. Es ergibt sich dann die in Abbildung 17.3 gezeigte Konstellation.

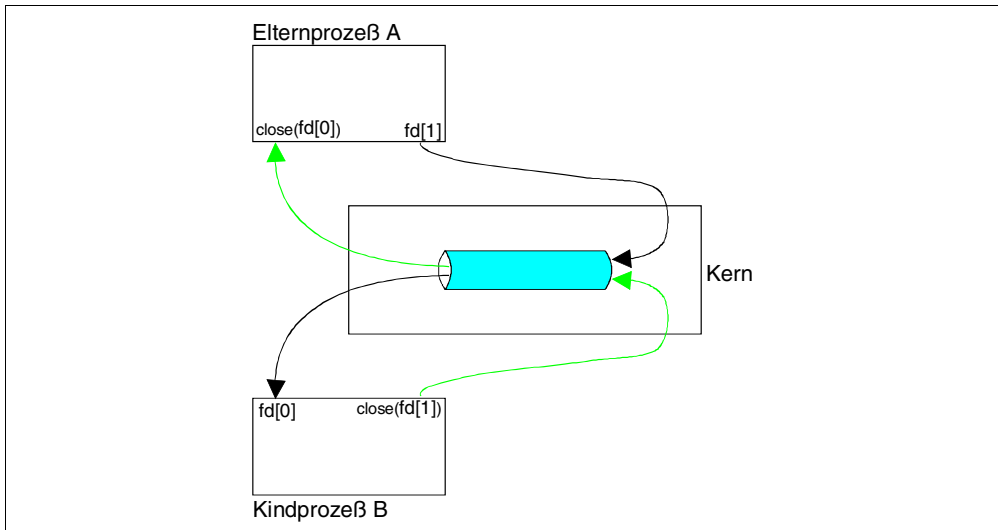


Abbildung 17.3: Pipe, in der Daten vom Eltern- zum Kindprozeß fließen

2. Elternprozeß liest, Kindprozeß schreibt

In diesem Fall muß der Elternprozeß die Schreibseite der Pipe (`fd[1]`) und der Kindprozeß die Leseseite der Pipe (`fd[0]`) schließen. Es ergibt sich dann die in Abbildung 17.4 gezeigte Konstellation.

Wenn eine Seite einer Pipe geschlossen wird, so gelten die beiden folgenden Regeln:

1. Beim Lesen aus einer Pipe, deren Schreibseite geschlossen wurde, nachdem alle Daten aus dieser Pipe gelesen wurden, liefert **read** 0 (für EOF).
2. Beim Schreiben in eine Pipe, deren Leseseite geschlossen wurde, wird das Signal **SIGPIPE** (gebrochene Pipe) generiert. Sowohl beim Ignorieren als auch beim Abfangen des Signals (nach der Rückkehr aus dem Signalhandler) liefert **write** einen Fehler als Rückgabewert, wobei `errno` auf `EPIPE` gesetzt wird.

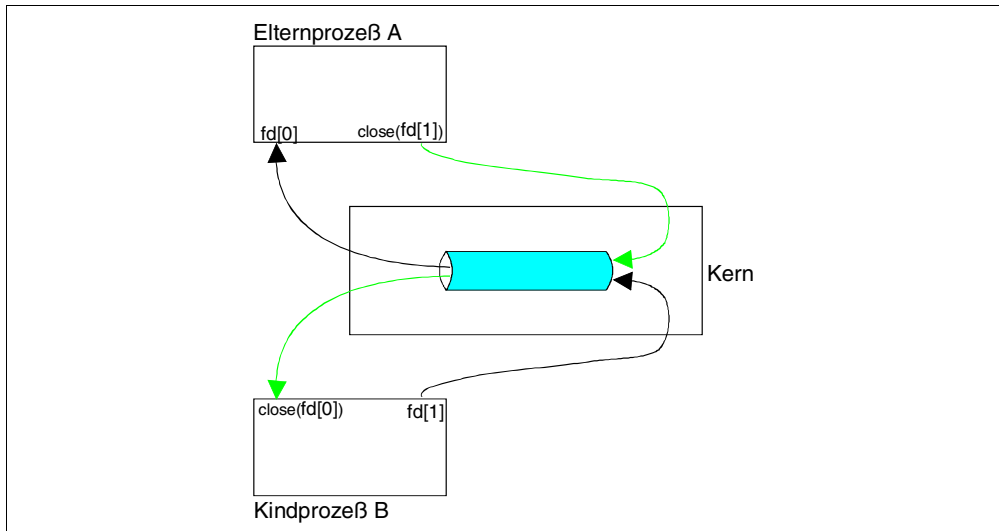


Abbildung 17.4: Pipe, in der Daten vom Kind- zum Elternprozeß fließen

Hinweis

Beim Schreiben in eine Pipe legt die Konstante `PIPE_BUF` die vom Kern verwendete Puffergröße für die Pipe fest. Wenn mehrere Prozesse gleichzeitig in dieselbe Pipe schreiben, dann ist sichergestellt, daß keinerlei Mischen der von den unterschiedlichen Prozessen geschriebenen Daten stattfindet, solange mit einem **write** nicht mehr als `PIPE_BUF` Bytes auf einmal geschrieben werden.

Die Funktion **fstat** (siehe Kapitel 5.1) klassifiziert einen Pipe-Filedeskriptor (Lese- oder Schreibseite) bei der Dateiarart (Komponente `st_mode` in Struktur `stat`) als FIFO. Mit dem Makro `S_ISFIFO` kann geprüft werden, ob es sich um eine Pipe handelt.

Beispiel

Eingegebene Zeile in Großschreibung ausgeben (Eltern-Kind-Pipe)

Das folgende Programm 17.1 (`gross1.c`) liest eine Textzeile ein und gibt diese wieder in Großschreibung aus. Dabei ist der Elternprozeß für das Einlesen der Zeile zuständig. Die gelesene Zeile schickt der Elternprozeß dann über eine Pipe an den Kindprozeß. Dieser liest diese Zeile aus der Pipe und gibt sie dann in Großschreibung aus.

```
#include <ctype.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    int    fd[2], i, n;
```

```

pid_t    pid;
char     zeile[MAX_ZEICHEN];

if (pipe(fd) < 0)
    fehler_meld(FATAL_SYS, "kann keine Pipe einrichten");

if ( (pid = fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");

else if (pid > 0) { /*----- Elternprozess: schreibt in die Pipe -----*/
    close(fd[0]); /* Leseseite der Pipe schliessen */
    fgets(zeile, MAX_ZEICHEN, stdin);
    write(fd[1], zeile, strlen(zeile));
    if (waitpid(pid, NULL, 0) < 0)
        fehler_meld(FATAL_SYS, "waitpid-Fehler");

} else { /*----- Kindprozess: liest aus der Pipe -----*/
    close(fd[1]); /* Schreibseite der Pipe schliessen */
    n = read(fd[0], zeile, MAX_ZEICHEN);
    for (i=0; i<n; i++)
        zeile[i] = toupper(zeile[i]);
    write(STDOUT_FILENO, zeile, n);
}

exit(0);
}

```

Programm 17.1 (gross1.c): Ausgabe einer eingelesenen Zeile in Großschreibung (Eltern-Kind-Pipe)

Nachdem man dieses Programm 17.1 (gross1.c) kompiliert und gelinkt hat

```
cc -o gross1 gross1.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ gross1
Das ist eine Zeile
DAS IST EINE ZEILE
$ gross1
Hallo Kind
HALLO KIND
$

```

Beispiel

Eingegebene Zeile in Großschreibung ausgeben (Kind-Kind-Pipe)

Das folgende Programm 17.2 (gross2.c) leistet das gleiche wie das Programm 17.1 (gross1.c). Hier wird nun jedoch eine Pipe zwischen zwei Kindprozessen (mit gleichem Elternprozeß) verwendet. Ein Kindprozeß liest die Zeile von der Standardeingabe, schickt sie dann über eine Pipe an den anderen Kindprozeß, der diese Zeile aus der Pipe liest, in Großbuchstaben umwandelt und auf der Standardausgabe ausgibt.

```

#include <ctype.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    int      fd[2], i, n;
    pid_t     pid1, pid2;
    char      zeile[MAX_ZEICHEN];

    if (pipe(fd) < 0)
        fehler_meld(FATAL_SYS, "kann keine Pipe einrichten");

    /*----- Kind 1: Schreiber -----*/
    if ( (pid1 = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid1 > 0)
        close(fd[1]); /* Elternprozess: schliesst Schreibseite der Pipe */

    else {
        /*----- Kind1: schreibt in die Pipe -----*/
        close(fd[0]); /* Leseseite der Pipe schliessen */
        fgets(zeile, MAX_ZEICHEN, stdin);
        write(fd[1], zeile, strlen(zeile));
        exit(0);
    }

    /*----- Kind 2: Leser -----*/
    if ( (pid2 = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid2 > 0)
        close(fd[0]); /* Elternprozess: schliesst Leseseite der Pipe */

    else {
        /*----- Kind2: liest aus der Pipe -----*/
        close(fd[1]); /* Schreibseite der Pipe schliessen */
        n = read(fd[0], zeile, MAX_ZEICHEN);
        for (i=0; i<n; i++)
            zeile[i] = toupper(zeile[i]);
        write(STDOUT_FILENO, zeile, n);
        exit(0);
    }

    /*----- Elternprozess wartet auf Beendigung der beiden Kindprozesse -*/
    if (waitpid(pid1, NULL, 0) < 0)
        fehler_meld(FATAL_SYS, "waitpid-Fehler");
    if (waitpid(pid2, NULL, 0) < 0)
        fehler_meld(FATAL_SYS, "waitpid-Fehler");

    exit(0);
}

```

Programm 17.2 (gross2.c): Ausgabe einer eingelesenen Zeile in Großschreibung (Kind-Kind-Pipe)

Sollen zwei Kindprozesse über eine Pipe kommunizieren, so schließt der Elternprozeß nach dem Kreieren des »Schreib-Kinds« die Schreibseite seiner Pipe und das »Schreib-Kind« die Leseseite seiner Pipe. Abbildung 17.5 veranschaulicht dies.

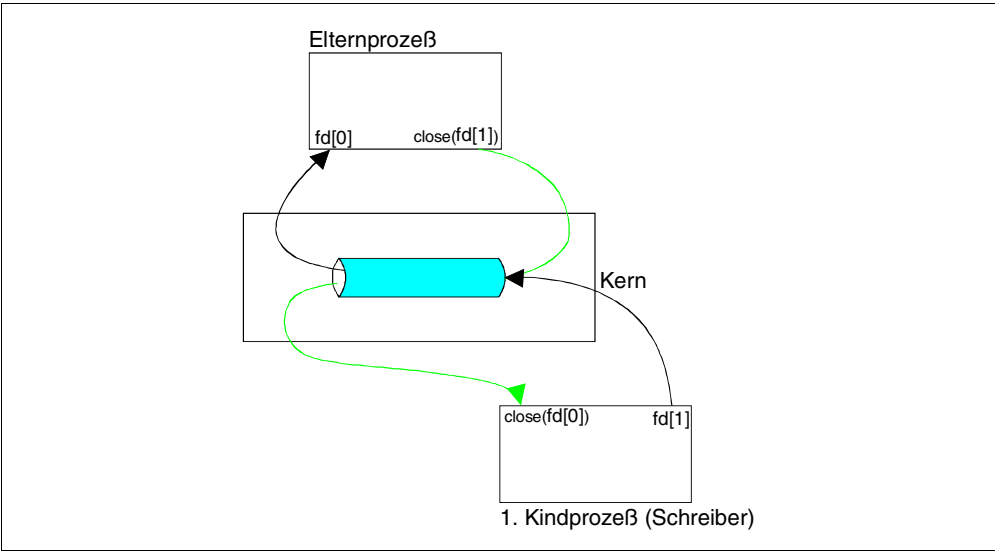


Abbildung 17.5: Herstellung einer Pipe-Verbindung zwischen Schreib-Kind und Elternprozeß

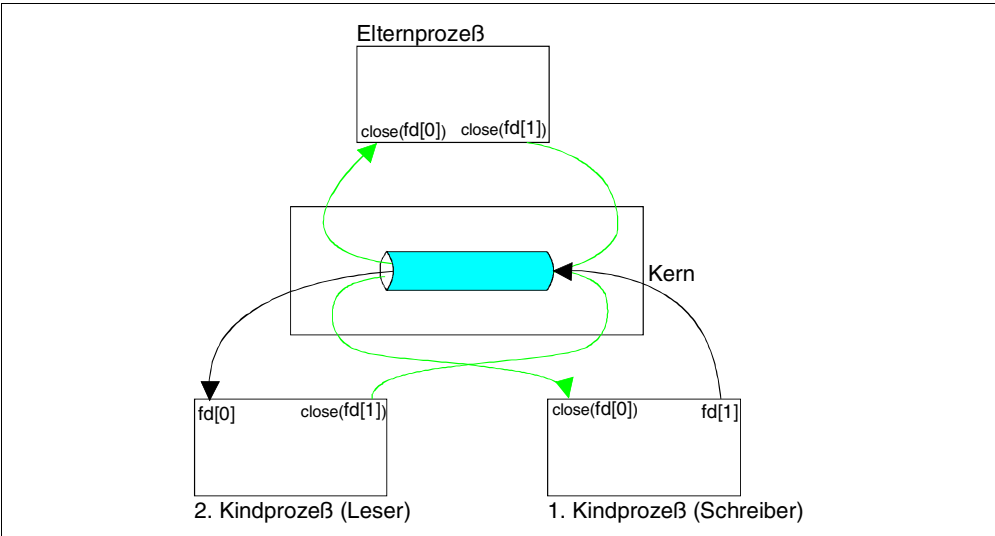


Abbildung 17.6: Endgültige Herstellung einer Pipe-Verbindung zwischen Schreib- und Lesekind

Nach dem Kreieren des »Lese-Kinds« schließt der Elternprozeß die Leseseite seiner Pipe und das »Lese-Kind« die Schreibseite seiner Pipe, so daß sich die in Abbildung 17.6 gezeigte Konstellation ergibt.

17.2.2 Zugriff auf eine Pipe mit Standard-E/A-Funktionen

Auf eine Pipe kann nicht nur mit den elementaren E/A-Funktionen wie z.B. **read** oder **write**, sondern auch mit Standard-E/A-Funktionen wie z.B. **fscanf** oder **fprintf** zugegriffen werden. Dazu muß allerdings zuerst den mit dem **pipe**-Aufruf erhaltenen Filedeskriptoren unter Verwendung der Funktion **fdopen** (siehe Kapitel 4.10) ein Dateizeiger (vom Typ `FILE *`) zugeteilt werden.

Das nachfolgende Programm 17.3 (`einmalei.c`) demonstriert die dabei notwendige Vorgehensweise, indem es das Einmaleins auf der Standardausgabe ausgibt. Der Elternprozeß ermittelt mit zwei ineinander geschachtelten **for**-Schleifen alle Kombinationen des Einmaleins (bis 100) und reicht jedes gefundene Zahlenpaar über eine Pipe an den Kindprozeß weiter. Der Kindprozeß seinerseits liest jedes Zahlenpaar aus der Pipe, berechnet das Produkt und gibt es aus.

```
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    int      fd[2], i, j;
    pid_t    pid;
    FILE      *schreib_dz,
              *lese_dz;

    if (pipe(fd) < 0)
        fehler_meld(FATAL_SYS, "kann keine Pipe einrichten");

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid > 0) { /*----- Elternprozess: schreibt in die Pipe -----*/
        close(fd[0]); /* Leseseite der Pipe schliessen */
        if ( (schreib_dz = fdopen(fd[1], "w")) == NULL)
            fehler_meld(FATAL_SYS, "fdopen-Fehler");
        for (i=1; i<=10; i++)
            for (j=1; j<=10; j++)
                fprintf(schreib_dz, "%d %d ", i, j);
        fclose(schreib_dz); /* Schreibseite schliessen, um Kindprozess */
                          /* das Ende des Schreibens in Pipe mitzuteilen */
        if (waitpid(pid, NULL, 0) < 0)
            fehler_meld(FATAL_SYS, "waitpid-Fehler");
    } else { /*----- Kindprozess: liest aus der Pipe -----*/
        close(fd[1]); /* Schreibseite der Pipe schliessen */
        if ( (lese_dz = fdopen(fd[0], "r")) == NULL)
```

```

        fehler_meld(FATAL_SYS, "fdopen-Fehler");
        while (fscanf(lese_dz, "%d %d", &i, &j) != EOF)
            printf("%3d * %3d = %3d\n", i, j, i*j);
    }

    exit(0);
}

```

Programm 17.3 (**einmalei.c**): Zugriff auf Pipe mit Standard-E/A-Routine

Im obigen Programm 17.3 (**einmalei.c**) ist die Verwendung von Standard-E/A-Funktionen sehr hilfreich, da dadurch die umständliche Konvertierung von Zahlen in Zeichenketten wegfällt. Diese Umwandlungen übernehmen dort die beiden Funktionen **fprintf** und **fscanf**. Beim **fprintf** des Elternprozesses ist nur darauf zu achten, daß nach dem letzten %d ein Leerzeichen angegeben ist, um die einzelnen Zahlen voneinander zu trennen. Vergißt man dieses Leerzeichen, so wird beim nächsten **fprintf** die erste Zahl direkt an die letzte Zahl des vorherigen **fprintf**-Aufrufs angehängt, was dazu führt, daß **fscanf** diese beide Zahlen als eine Zahl liest und somit nicht das gewünschte Ergebnis liefert.

Nachdem man dieses Programm 17.3 (**einmalei.c**) kompiliert und gelinkt hat

```
cc -o einmalei einmalei.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ einmalei
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
.....
.....
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
$

```

17.2.3 Leseseite einer Pipe in die Standardeingabe eines anderen Programms umleiten

Oft ist es sehr nützlich, wenn man die Leseseite einer Pipe direkt in die Standardeingabe eines bereits existierenden Programms umlenken kann. Typische Beispiele hierfür sind Programme, die mehr Zeilen ausgeben, als auf den Bildschirm passen. In diesen Fällen ist es meist mühsam, die seitenweise Ausgabe vom Programm aus zu organisieren. Viel angenehmer wäre es, wenn man vom Programm aus seine Daten direkt an die entsprechenden Unix-Kommandos, wie z.B. **more**, über eine Pipe weiterleiten könnte. Dies ist auch möglich, man muß dazu nur die folgende Vorgehensweise wählen:

1. Pipe einrichten,
2. Kindprozeß kreieren,
3. Standardeingabe des Kindprozesses auf die Leseseite der Pipe einrichten,
4. mit **exec** den Kindprozeß mit entsprechendem Programm (wie z.B. **more**) überlagern.

Das Programm 17.4 (`prim_fak.c`) demonstriert diese Vorgehensweise anhand einer Primfaktorzerlegung. Der Elternprozeß führt dabei die Primfaktorzerlegung für alle Zahlen aus einem Zahlenbereich durch, dessen Start- und Endwert auf der Kommandozeile angegeben ist. Sämtliche berechnete Ergebnisse schickt er über eine Pipe an den Kindprozeß, der sich mittels **exec** mit dem Unix-Programm **more** überlagert.

```
#include <math.h>
#include <sys/wait.h>
#include "eighdr.h"

int
main(char argc, char *argv[])
{
    long        von, bis, i,
               teiler, zahl, wurzel;
    int         fd[2];
    pid_t       pid;
    FILE        *schreib_dz;

    if (argc != 3)
        fehler_meld(FATAL, "usage: %s von bis", argv[0]);
    else if ( (von = atol(argv[1])) <= 0 || (bis = atol(argv[2])) <= 0 )
        fehler_meld(FATAL, "Argumente muessen positive Zahlen sein");

    if (pipe(fd) < 0)
        fehler_meld(FATAL_SYS, "kann keine Pipe einrichten");

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid > 0) { /*----- Elternprozess: schreibt in die Pipe -----*/
        close(fd[0]); /* Leseseite der Pipe schliessen */
        if ( (schreib_dz = fdopen(fd[1], "w")) == NULL)
            fehler_meld(FATAL_SYS, "fdopen-Fehler");

        for (i=von; i<=bis; i++) { /*-- Primfaktorzerlegung fuer von..bis */
            teiler = 2;
            wurzel = sqrt(zahl=i);
            fprintf(schreib_dz, "%ld = ", zahl);
            while (teiler <= wurzel) { /* Faktoren zu einer Zahl */
                while (zahl % teiler == 0) {
                    fprintf(schreib_dz, "%ld * ", teiler);
                    zahl /= teiler;
                }
                teiler++;
            }
        }
    }
}
```

```

    }
    if (zahl != 1)
        fprintf(schreib_dz, "%ld\n", zahl);
    else
        fprintf(schreib_dz, "\b\b\b  \n");
}
fclose(schreib_dz); /* Schreibseite schliessen, um Kindprozess */
/* das Ende des Schreibens in Pipe mitzuteilen */
if (waitpid(pid, NULL, 0) < 0)
    fehler_meld(FATAL_SYS, "waitpid-Fehler");
exit(0);

} else { /*----- Kindprozess: liest aus der Pipe -----*/
    close(fd[1]); /* Schreibseite der Pipe schliessen */
    if (fd[0] != STDIN_FILENO) {
        if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
            fehler_meld(FATAL_SYS, "Fehler bei dup2 fuer stdin");
        close(fd[0]); /* Nach dup2 wird fd[0] nicht mehr benoetigt */
    }

    if (execl("/usr/bin/more", "more", NULL) < 0)
        fehler_meld(FATAL_SYS, "execl-Fehler fuer more");
}
}

```

Programm 17.4 (**primfak.c**): Leseseite einer Pipe auf Programm **more** einstellen

Nach dem Einrichten der Pipe und dem Kreieren eines Kindprozesses schließt der Elternprozeß die Leseseite seiner Pipe und der Kindprozeß die Schreibseite seiner Pipe.

Der Kindprozeß ruft dann **dup2** auf, um seine Standardeingabe auf die Leseseite der Pipe einzustellen. Diese Einstellung der Standardeingabe bleibt auch für das Programm **more** erhalten, mit dem sich der Kindprozeß mittels **execl** überlagert. Somit werden alle vom Elternprozeß in die Pipe geschriebenen Daten direkt an das Programm **more** weitergeleitet, welches für die entsprechende seitenweise Ausgabe sorgt. Die Abfrage

```
if (fd[0] != STDIN_FILENO)
```

ist aus folgendem Grund notwendig: Sollte nämlich ein Filedeskriptor bereits auf den entsprechenden Wert (hier `STDIN_FILENO`) eingestellt sein, dann dupliziert **dup2** diesen Filedeskriptor nicht. Dies hat hier die fatale Auswirkung, daß mit dem nachfolgenden **close** der nicht duplizierte Filedeskriptor (nämlich `STDIN_FILENO`) geschlossen wird und keine Standardeingabe mehr verfügbar ist.

Nachdem man dieses Programm 17.4 (**primfak.c**) kompiliert und gelinkt hat

```
cc -o primfak primfak.c fehler.c -lm
```

ergibt sich z.B. der folgende Ablauf:

```

$ primfak 125 153
125 = 5 * 5 * 5
126 = 2 * 3 * 3 * 7

```

```

127 = 127
128 = 2 * 2 * 2 * 2 * 2 * 2 * 2
129 = 3 * 43
130 = 2 * 5 * 13
131 = 131
132 = 2 * 2 * 3 * 11
133 = 7 * 19
134 = 2 * 67
135 = 3 * 3 * 3 * 5
136 = 2 * 2 * 2 * 17
137 = 137
138 = 2 * 3 * 23
139 = 139
140 = 2 * 2 * 5 * 7
141 = 3 * 47
142 = 2 * 71
143 = 11 * 13
144 = 2 * 2 * 2 * 2 * 3 * 3
145 = 5 * 29
146 = 2 * 73
147 = 3 * 7 * 7
--More--                [Zum Weiterblättern Leertaste drücken]
148 = 2 * 2 * 37
149 = 149
150 = 2 * 3 * 5 * 5
151 = 151
152 = 2 * 2 * 2 * 19
153 = 3 * 3 * 17
$

```

17.2.4 Synchronisation von Prozessen über Pipes

In Kapitel 10.4 benutzten wir zur Synchronisation von Prozessen die eigenen Funktionen **INIT_SYNCH**, **HALLO_PAPA**, **WARTE_AUF_PAPA**, **HALLO_KIND** und **WARTE_AUF_KIND**.

Eine mögliche Implementierung dieser Funktionen unter Verwendung von Signalen wurde im Programm 13.15 (forksync.c) in Kapitel 13.6 gezeigt. Das folgende Programm 17.5 (pipesync.c) zeigt eine mögliche Implementierung dieser Funktionen unter Verwendung von Pipes.

```

#include "eighdr.h"

static int ek_pipe[2],
          ke_pipe[2];

/*----- Synchronisation initialisieren -----*/
void INIT_SYNCH(void)
{
    if (pipe(ek_pipe) < 0 || pipe(ke_pipe) < 0)
        fehler_meld(FATAL_SYS, "kann Pipe nicht einrichten");
}

```

```

/*----- Info von Kind an Elternprozess, dass es fertig ist -----*/
void HALLO_PAPA(pid_t pid)
{
    if (write(ke_pipe[1], "k", 1) != 1)
        fehler_meld(FATAL_SYS, "write-Fehler");
}

/*----- Kind wartet auf Zeichen vom Elternprozess -----*/
void WARTE_AUF_PAPA(void)
{
    char    zeich;

    if (read(ek_pipe[0], &zeich, 1) != 1)
        fehler_meld(FATAL_SYS, "read-Fehler");
    if (zeich != 'e')
        fehler_meld(FATAL_SYS, "WARTE_AUF_PAPA: Synchronisation inkonsistent");
}

/*----- Info von Elternprozess an Kind, dass er fertig ist -----*/
void HALLO_KIND(pid_t pid)
{
    if (write(ek_pipe[1], "e", 1) != 1)
        fehler_meld(FATAL_SYS, "write-Fehler");
}

/*----- Elternprozess wartet auf Zeichen vom Kind -----*/
void WARTE_AUF_KIND(void)
{
    char    zeich;

    if (read(ke_pipe[0], &zeich, 1) != 1)
        fehler_meld(FATAL_SYS, "read-Fehler");
    if (zeich != 'k')
        fehler_meld(FATAL_SYS, "WARTE_AUF_KIND: Synchronisation inkonsistent");
}

```

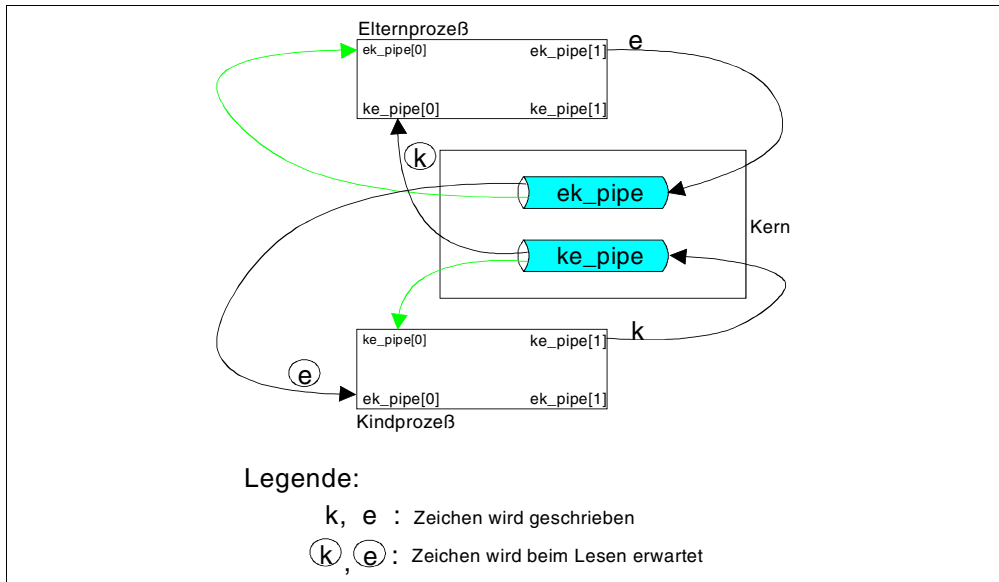
Programm 17.5 (**pipesync.c**): Funktionen zur Synchronisation von Eltern- und Kindprozeß

INIT_SYNCH kreiert zwei Pipes, die nach dem entsprechenden **fork** des aufrufenden Programms sowohl dem Eltern- als auch dem Kindprozeß zur Verfügung stehen.

Wird **HALLO_KIND** aufgerufen, so schickt der Elternprozeß über die Eltern-Kind-Pipe (**ek_pipe**) das Zeichen »e« an den Kindprozeß.

Wird **HALLO_PAPA** aufgerufen, so schickt der Kindprozeß über die Kind-Eltern-Pipe (**ke_pipe**) das Zeichen »k« an den Elternprozeß.

Die Funktion **WARTE_AUF_KIND** liest aus der Kind-Eltern-Pipe ein Zeichen und die Funktion **WARTE_AUF_PAPA** liest aus der Eltern-Kind-Pipe ein Zeichen. Diese Leseoperation blockiert den Aufrufer der jeweiligen Funktion so lange, bis dieses Zeichen vom jeweils anderen Prozeß geschickt wird. Abbildung 17.7 veranschaulicht diese Art der Synchronisation.



In Abbildung 17.7 ist erkennbar, daß sowohl der Eltern- wie der Kindprozeß einen zusätzlichen Lesekanal besitzen (Elternprozeß: `ek_pipe[0]`, Kindprozeß: `ke_pipe[0]`). Solange jedoch keiner dieser beiden Prozesse von diesem zusätzlichen Kanal liest, führt dies zu keinerlei Komplikationen.

17.2.5 popen und pclose – Einrichten und Schließen einer Pipe zu einem anderen Programm

Um zu einem anderen Programm eine Pipe einzurichten oder wieder zu schließen, stehen die beiden Funktionen ***popen*** und ***pclose*** zur Verfügung.

```
#include <stdio.h>

FILE *popen(const char *kdozeile, const char *typ);
                                     gibt zurück: Dateizeiger (bei Erfolg); NULL bei Fehler

int pclose(FILE *dz);
                                     gibt zurück: Beendigungsstatus von kdozeile oder -1 bei Fehler
```

popen

Mit **popen** kann zu einem schon existierenden Programm eine Pipe eingerichtet werden. **popen** nimmt dem Programmierer in diesem Fall viel Arbeit ab, indem es die folgenden Aktionen automatisch ausführt:

1. Einrichten einer Pipe mit **pipe**
2. Kreieren eines Kindprozesses mit **fork**
3. Schließen der nicht benutzten Seiten der Pipe in Eltern- und Kindprozeß
4. Überlagern des Kindprozesses durch ein Shellprogramm mit einem **exec**-Aufruf, um die entsprechende `kdozeile` ausführen zu lassen
5. Warten auf die Beendigung des Kommandos `kdozeile`

Die Funktion **popen** richtet also zwischen dem aufrufenden Prozeß und dem Programm `kdozeile`, das gestartet wird, eine Pipe ein. Für `typ` ist entweder `»r«` (für Lesen aus der Pipe) oder `»w«` (für Schreiben in die Pipe) anzugeben. Der Rückgabewert ist ein Dateizeiger für diese Pipe.

Wird für `typ` `»r«` angegeben, so kann aus dieser Pipe gelesen werden, wobei die gelesenen Daten direkt aus der Standardausgabe von `kdozeile` stammen.

Wird für `typ` `»w«` angegeben, so kann in die Pipe geschrieben, wobei die geschriebenen Daten direkt an die Standardeingabe von `kdozeile` weitergeleitet werden.

pclose

Eine mit **popen** eingerichtete Pipe sollte mit **pclose** wieder geschlossen werden. **pclose** wartet auf die Beendigung von `kdozeile` und liefert den Beendigungsstatus von `kdozeile` als Rückgabewert. Der Begriff *Beendigungsstatus* wurde in Kapitel 10.3 beschrieben. Falls die Shell, die zur Ausführung von `kdozeile` notwendig ist, nicht gestartet werden kann, so liefert **pclose** als Beendigungsstatus das gleiche, wie wenn `exit(127)` aufgerufen worden wäre.

Hinweis

Die beim **popen**-Aufruf angegebene `kdozeile` wird von der Bourne-Shell so ausgeführt, als ob man

```
sh -c kdozeile
```

direkt aufgerufen hätte. Das bedeutet, daß Shell-Metazeichen in der `kdozeile` expandiert werden. Somit sind z.B. auch Aufrufe der folgenden Form möglich:

```
dz = popen("grep Hans *.txt", "r");  
dz = popen("cat >text", "w");
```

Da **popen** und **pclose** mit der Shell kommunizieren, sind sie nicht Bestandteil von POSIX.1, sondern von POSIX.2.

Beispiel

Weiterleiten einer Ausgabe an more mit popen

Hier soll das Programm 17.4 (primfak.c) unter Verwendung von **popen** realisiert werden. Programm 17.6 (primfak2.c) ist eine mögliche Implementierung zu dieser Primfaktorzerlegung.

```
#include <math.h>
#include <sys/wait.h>
#include "eighdr.h"

#define PAGER "${PAGER:-/usr/bin/more}" /* Voreinstellung ist more, wenn */
                                        /* Environment-Variable PAGER */
                                        /* nicht einen anderen Programm */
                                        /* wie z.B. pg fuer die */
                                        /* seitenweise Ausgabe vorgibt */

int
main(char argc, char *argv[])
{
    long        von, bis, i,
               teiler, zahl, wurzel;
    FILE        *schreib_dz;

    if (argc != 3)
        fehler_meld(FATAL, "usage: %s von bis", argv[0]);
    else if ( (von = atol(argv[1])) <= 0 || (bis = atol(argv[2])) <= 0 )
        fehler_meld(FATAL, "Argumente muessen positive Zahlen sein");

    if ( (schreib_dz = popen(PAGER, "w")) == NULL)
        fehler_meld(FATAL_SYS, "popen-Fehler");

    for (i=von; i<=bis; i++) { /*-- Primfaktorzerlegung fuer von..bis */
        teiler = 2;
        wurzel = sqrt(zahl=i);
        fprintf(schreib_dz, "%ld = ", zahl);
        while (teiler <= wurzel) { /* Faktoren zu einer Zahl */
            while (zahl % teiler == 0) {
                fprintf(schreib_dz, "%ld * ", teiler);
                zahl /= teiler;
            }
            teiler++;
        }
        if (zahl != 1)
            fprintf(schreib_dz, "%ld\n", zahl);
        else
            fprintf(schreib_dz, "\b\b\b  \n");
    }

    if (pclose(schreib_dz) == -1)
        fehler_meld(FATAL_SYS, "pclose-Fehler");

    exit(0);
}
```

Programm 17.6 (**primfak2.c**): Primfaktorzerlegung mit Weiterleitung an **more** mittels **popen**

Das Programm 17.6 (`primfak2.c`) ist wesentlich kürzer als das Programm 17.4 (`primfak.c`), das zwar das gleiche leistet, aber nicht **`popen`** verwendet, sondern die Funktionsweise von **`popen`** durch Erzeugung eines Kindprozesses und Verwendung anderer Funktionen nachbildet.

17.2.6 Transformationen mittels Filterprogramme

Des öfteren benötigt man Programme, in denen die Eingabe zunächst in eine andere Form umgewandelt werden muß, bevor man sie weiterverarbeitet. Existiert nun ein Programm, das diese Umwandlung durchführt, so kann man es mit **`popen`** zwischen dem eigentlichen Programm und seiner Eingabe schalten. Solche dazwischengeschaltete Programme nennt man *Filterprogramme* oder auch nur *Filter*. Abbildung 17.8 veranschaulicht sie.

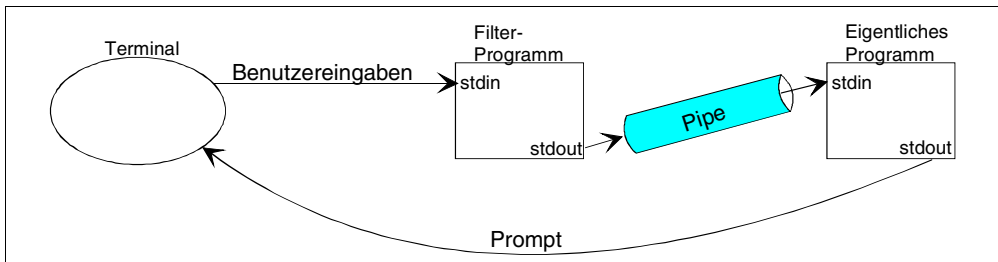


Abbildung 17.8: Transformation der Eingabe mit einem Filterprogramm

Programm 17.7 (`grosklei.c`) zeigt ein einfaches Filterprogramm, das die Standardeingabe auf die Standardausgabe kopiert, wobei es jedoch alle Großbuchstaben in Kleinbuchstaben umwandelt.

```
#include <ctype.h>
#include "eighdr.h"

int
main(void)
{
    int    zeich;

    while ( (zeich=getchar()) != EOF) {
        zeich = tolower(zeich);
        if (putchar(zeich) == EOF)
            fehler_meld(FATAL_SYS, "Fehler bei Ausgabe");
        if (zeich == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Programm 17.7 (**`grosklei.c`**): Filterprogramm zum Umwandeln von Groß- und Kleinbuchstaben

Nachdem man dieses Programm 17.7 (`grosklei.c`) kompiliert und gelinkt hat

```
cc -o grosklei grosklei.c fehler.c
```

kann **grosklei** von anderen Programmen benutzt werden, indem sie mit **popen** eine Pipe zu diesem Filterprogramm einrichten. Das Programm 17.8 (`trafo.c`) verdeutlicht dies.

```
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
    char    zeile[MAX_ZEICHEN];
    FILE    *dz_ein;

    if ( (dz_ein = popen("grosklei", "r")) == NULL)
        fehler_meld(FATAL_SYS, "popen-Fehler");

    while (1) {
        fprintf(stdout, "Gib ein> ");
        fflush(stdout); /* notwendig, da stdout zeilengepuffert ist */
        if (fgets(zeile, MAX_ZEICHEN, dz_ein) == NULL) /*aus Pipe Zeile lesen */
            break;
        if (fputs(zeile, stdout) == EOF)
            fehler_meld(FATAL_SYS, "Pipe-Schreibfehler");
    }
    if (pclose(dz_ein) == -1)
        fehler_meld(FATAL_SYS, "pclose-Fehler");
    putchar('\n');
    exit(0);
}
```

*Programm 17.8 (**trafo.c**): Verwendung des Filters `grosklei`*

Nachdem man dieses Programm 17.8 (`trafo.c`) kompiliert und gelinkt hat

```
cc -o trafo trafo.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ trafo
Gib ein> Hallo
hallo
Gib ein> ICH HOERE AUF
ich hoere auf
Gib ein> Ctrl-D
$
```

Das Programm 17.9 (`zahlwort.c`) ist ein weiteres Beispiel für ein Filterprogramm. Es filtert alle Zahlen aus der Eingabe heraus und formt sie in die entsprechende Wortdarstellung um.

```

#include <ctype.h>
#include "eighdr.h"

static const char *einer_wort[] = {
    "null", "ein", "zwei", "drei", "vier",
    "fuenf", "sechs", "sieben", "acht", "neun", NULL
};

static const char *zehner_wort[] = {
    "zehn", "elf", "zwoelf", "dreizehn", "vierzehn",
    "fuenfzehn", "sechzehn", "siebzehn", "achtzehn", "neunzehn", NULL
};

static const char *zig_wort[] = {
    "zwanzig", "dreissig", "vierzig", "fuenfzig", "sechzig",
    "siebzig", "achtzig", "neunzig", "achtzehn", "neunzehn", NULL
};

static void zahl_in_woerter(unsigned long zahl,
                           char *einheit, char *plural, char *fall);

int
main(void)
{
    int          zeich;
    unsigned long zahl;

    while ( (zeich=getchar()) != EOF) {
        if (isdigit(zeich)) {
            ungetc(zeich, stdin);
            fscanf(stdin, "%lu", &zahl);
            fprintf(stdout, "==");
            zahl_in_woerter(zahl/1000000000, "milliarde", "n", "e");
            zahl_in_woerter(zahl/1000000%1000, "million", "en", "e");
            zahl_in_woerter(zahl/1000%1000, "tausend", "", "");
            zahl_in_woerter(zahl%1000, "", "", "s");
            fprintf(stdout, "==");
            fflush(stdout);
        } else
            if (putchar(zeich) == EOF)
                fehler_meld(FATAL_SYS, "Fehler bei Ausgabe");
            if (zeich == '\n') {
                fflush(stdout);
            }
    }
    exit(0);
}

static void zahl_in_woerter(unsigned long zahl,
                           char *einheit, char *plural, char *fall)
{
    int    hundert = zahl/100,
           zehner  = zahl%100/10,

```

```

        einer    = zahl%10;

    if (zahl > 0) {
        if (hundert > 0)
            fprintf(stdout, "%shundert", einer_wort[hundert]);
        if (zehner == 1)
            fprintf(stdout, "%s", zehner_wort[einer]);
        else if (einer > 0) {
            fprintf(stdout, "%s", einer_wort[einer]);
            if (zehner > 1)
                fprintf(stdout, "und%s", zig_wort[zehner-2]);
            else if (einer == 1)
                fprintf(stdout, "%s", fall);
        }
        fprintf(stdout, "%s%s-", einheit, (zahl>1) ? plural : "");
    }

    return;
}

```

Programm 17.9 (zahlwort.c): Filter zum Umwandeln von Zahlen in Wortform

Nachdem man dieses Programm 17.9 (zahlwort.c) kompiliert und gelinkt hat

```
cc -o zahlwort zahlwort.c fehler.c
```

kann dieses Filterprogramm **zahlwort** von anderen Programmen benutzt werden. Im Programm 17.8 (trafo.c) muß dazu anstelle von **grosklei** beim **popen**-Aufruf das Programm **zahlwort** angegeben werden. Das entsprechende Programm trafo2.c wird hier nicht aufgelistet.

Nachdem man dieses Programm trafo2.c kompiliert und gelinkt hat

```
cc -o trafo2 trafo2.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ trafo2
Gib ein> Hans ist ==neununddreissig== Jahre alt
Gib ein> Er ist am 2.April 1956 geboren.
Er ist am ==zwei==.April ==eintausend-neunhundertsechsfuenfzig== geboren.
Gib ein> 333 v. Chr. war bei Issos Keilerei.
==dreihundertdreissig== v. Chr. war bei Issos Keilerei.
Gib ein> Ctr1-D
$

```

17.2.7 Koprozesse in der Korn-Shell

Die Korn-Shell bietet anders als die Bourne- oder C-Shell sogenannte Koprozesse an. Wird in der Kornshell ein Programm mit der folgenden Angabe gestartet:

```
programm |&
```

so wird *programm* als Koprozeß im Hintergrund gestartet. Dieser Prozeß läuft dann parallel zur Vatershell ab, die nicht auf die Beendigung des Prozesses wartet. Anders als beim Metazeichen **&** wird hier zusätzlich eine »Zweiwege-Pipe« eingerichtet, über die die Vatershell und der Koprozeß (Kindprozeß zur Vatershell) miteinander kommunizieren können.

»Zweiwege-Pipe« bedeutet, daß der Elternprozeß über die Pipe in die Standardeingabe von *programm* schreiben oder aber aus dessen Standardausgabe lesen kann. Dazu muß in der Korn-Shell der Elternprozeß die beiden Builtin-Kommandos **print** und **read** verwenden.

17.2.8 Koprozesse in C

Koprozesse können auch sehr nützlich für C-Programme sein. Während man mit **popen** nur eine »Einwege-Pipe« zu der Standardeingabe oder -ausgabe eines anderen Prozesses einrichten kann, kann bei einem Koprozeß eine »Zweiwege-Pipe« zu einem anderen Prozeß eingerichtet werden: eine zum Schreiben in die Standardeingabe und eine zum Lesen aus der Standardausgabe dieses Prozesses. Abbildung 17.9 verdeutlicht dies.

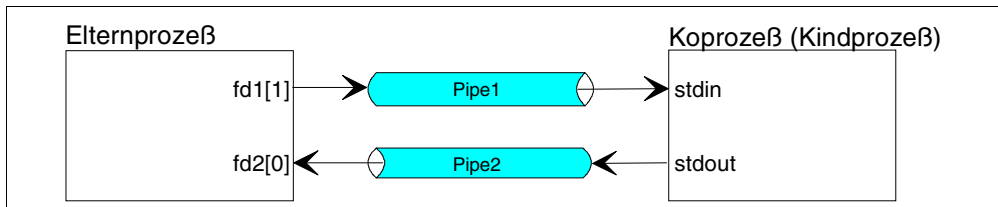


Abbildung 17.9: »Zweiwege-Pipe« zwischen Elternprozeß und Koprozeß (Kindprozeß)

Beispiel

Umwandeln von arabischen in römische Zahlen (Koprozeß in C)

Das nachfolgende Programm 17.10 (*romzahl.c*) liest eine Zahl von seiner Standardeingabe, wandelt diese Zahl in die entsprechende römische Zahl um und schreibt den String dann auf seine Standardausgabe.

```
#include "eighdr.h"

#define UNGUELTIG "ungueltige Eingabe\n"

static void block_ausgabe(long int wert,
                          char einer, char fuenfer, char zehner);

static char romzahl[MAX_ZEICHEN];

/*----- main -----*/
int
main(void)
{
```

```

int      n, laenge =strlen(UNGUELTIG);
long int  i, zahl;
char     zeile[MAX_ZEICHEN];
while ( (n = read(STDIN_FILENO, zeile, MAX_ZEICHEN)) > 0) {
    zeile[n] = '\0';
    if ( (zahl = atol(zeile)) > 0) {
        strcpy(romzahl, ".....");
        for (i=1 ; i<=zahl/1000 ; i++) /*--- Alle Tausender (M) ---*/
            strcat(romzahl, "M");
        zahl %= 1000; /*--- fuer Zahlenbereich 100..900 ---*/
        block Ausgabe(zahl/100, 'C', 'D', 'M');
        zahl %= 100; /*--- fuer Zahlenbereich 10..90 -----*/
        block Ausgabe(zahl/10, 'X', 'L', 'C');
        zahl %= 10; /*--- fuer Zahlenbereich 1..9 -----*/
        block Ausgabe(zahl, 'I', 'V', 'X');
        strcat(romzahl, "\n");
        n = strlen(romzahl);
        if (write(STDOUT_FILENO, romzahl, n) != n)
            fehler_meld(FATAL_SYS, "write-Fehler");
    } else {
        if (write(STDOUT_FILENO, UNGUELTIG, laenge) != laenge)
            fehler_meld(FATAL_SYS, "write-Fehler");
    }
}

/*----- block_Ausgabe -----*/
*
* Diese Funktion gibt immer den entspr. Block einer Zahl aus.
* wert      ist die auszugebende Zehnerpotenz
* einer     enthaelt immer entspr. roemische Zehnerpotenz-Zeichen
*          I=1, X=10, C=100
* fuenfer   enthaelt immer entspr. roemische Zeichen von 5 * Zehnerpotenz
*          V=5, L=50, D=500
* zehner    enthaelt immer entspr. roemische Zeichen der naechsten
*          Zehnerpotenz, die groesser als der wert ist.
*          X=10, C=100, M=1000
*/
static void block_Ausgabe(long int wert,
                          char einer, char fuenfer, char zehner)
{
    long int  i;

    if (wert==9) {
        sprintf(romzahl, "%s%c%c", romzahl, einer, zehner);
    } else if (wert>4) {
        sprintf(romzahl, "%s%c", romzahl, fuenfer);
        for (i=wert ; i>=6 ; i--)
            sprintf(romzahl, "%s%c", romzahl, einer);
    } else if (wert==4) {
        sprintf(romzahl, "%s%c%c", romzahl, einer, fuenfer);
    } else {
        for (i=wert ; i>=1 ; i--)

```

```

        sprintf(romzahl, "%s%c", romzahl, einer);
    }
}

```

*Programm 17.10 (**romzahl.c**): Filterprogramm zum Umwandeln von arabischen in römische Zahlen*

Nachdem man dieses Programm 17.10 (**romzahl.c**) kompiliert und gelinkt hat

```
cc -o romzahl romzahl.c fehler.c
```

kann dieses Filterprogramm **romzahl** von anderen Programmen als Koprozeß gestartet werden, indem sie mit **fork** einen Kindprozeß kreieren und diesen dann mit einem **exec**-Aufruf mit dem Programm **romzahl** überlagern.

Das Programm 17.11 (**romkomm.c**) zeigt die dazu erforderlichen Maßnahmen, indem es Zahlen von seiner Standardeingabe liest und diese an das als Koprozeß gestartete Programm **romzahl** weiterleitet. Die entsprechende römische Zahl erhält es von **romzahl** aus der Lesepipe zurück.

```

#include <signal.h>
#include "eighdr.h"

static void sig_pipe(int signr);    /* eigener Signalhandler */

int
main(void)
{
    int    n, pipe1[2], pipe2[2];
    pid_t  pid;
    char    zeile[MAX_ZEICHEN];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        fehler_meld(FATAL_SYS, "signal-Fehler");

    if (pipe(pipe1) < 0 || pipe(pipe2) < 0)
        fehler_meld(FATAL_SYS, "pipe-Fehler");

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid > 0) { /*----- Elternprozess -----*/
        close(pipe1[0]);
        close(pipe2[1]);

        while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
            n = strlen(zeile);
            if (write(pipe1[1], zeile, n) != n)
                fehler_meld(FATAL_SYS, "Fehler beim Schreiben in Pipe1");
            if ( (n = read(pipe2[0], zeile, MAX_ZEICHEN)) < 0)
                fehler_meld(FATAL_SYS, "Fehler beim Lesen aus Pipe2");
            if (n == 0) {
                fehler_meld(WARNUNG, "Kind hat Pipe geschlossen");
                break;
            }
        }
    }
}

```



```

        zeile[n] = '\0';
        if (fputs(zeile, stdout) == EOF)
            fehler_meld(FATAL_SYS, "fputs-Fehler");
    }
    if (ferror(stdin))
        fehler_meld(FATAL_SYS, "fgets-Fehler (in stdin)");
    exit(0);

} else { /*----- Kindprozess -----*/
    close(pipe1[1]);
    close(pipe2[0]);
    if (pipe1[0] != STDIN_FILENO) {
        if (dup2(pipe1[0], STDIN_FILENO) != STDIN_FILENO)
            fehler_meld(FATAL_SYS, "dup2-Fehler (bei stdin)");
        close(pipe1[0]);
    }
    if (pipe2[1] != STDOUT_FILENO) {
        if (dup2(pipe2[1], STDOUT_FILENO) != STDOUT_FILENO)
            fehler_meld(FATAL_SYS, "dup2-Fehler (bei stdout)");
        close(pipe2[1]);
    }
    if (execl("./romzahl", "romzahl", NULL) < 0)
        fehler_meld(FATAL_SYS, "execl-Fehler");
}

}

static void sig_pipe(int signr)
{
    printf(".....SIGPIPE abgefangen.....\n");
    exit(1);
}

```

Programm 17.11 (romkomm.c): Starten des Koprozesses *romzahl* und Kommunizieren mit diesem

Im Programm 17.11 (romkomm.c) werden zwei Pipes eingerichtet, wobei jeder Prozeß die entsprechenden Enden der beiden Pipes schließt.

Um eine »Zweiwege-Pipe« zu einem Koprozeß einzurichten, benötigt man zwei Pipes: eine für die Standardeingabe des Koprozesses und eine für seine Standardausgabe. Der Kindprozeß ruft dann **dup2** auf, um die Pipe-Deskriptoren auf seine Standardeingabe und Standardausgabe einzurichten, bevor er sich über **execl** mit dem Programm **romzahl** überlagert.

Nachdem man dieses Programm 17.11 (romkomm.c) kompiliert und gelinkt hat

```
cc -o romkomm romkomm.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ romkomm
7
.....VII
1295

```

```

.....MCCXCV
acht
ungueltige Eingabe
15999
.....MMMMMMMMMMMMMCMXCIX
Ctrl-D
$

```

17.2.9 Eventuelle Probleme mit Standard E/A-Pufferung bei Koprozessen

Im Programm 17.10 (romzahl.c), das als Koprozeß verwendet werden kann, wurden **read** und **write** benutzt, um von der Standardeingabe zu lesen oder auf die Standardausgabe zu schreiben. Würde man statt dessen Standard-E/A-Funktionen benutzen, wie dies im Programm 17.12 (romzahl2.c) geschehen ist, ist dieses Programm nicht mehr als Koprozeß verwendbar.

```

#include "eighdr.h"

static void block_ausgabe(long int wert,
                          char einer, char fuenfer, char zehner);

static char romzahl[MAX_ZEICHEN];

/*----- main -----*/
int
main(void)
{
    long int i, zahl;
    char zeile[MAX_ZEICHEN];

    while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
        if ( (zahl = atol(zeile)) > 0) {
            strcpy(romzahl, ".....");
            for (i=1 ; i<=zahl/1000 ; i++) /*--- Alle Tausender (M) ---*/
                strcat(romzahl, "M");
            zahl %= 1000; /*--- fuer Zahlenbereich 100..900 ---*/
            block_ausgabe(zahl/100, 'C', 'D', 'M');
            zahl %= 100; /*--- fuer Zahlenbereich 10..90 -----*/
            block_ausgabe(zahl/10, 'X', 'L', 'C');
            zahl %= 10; /*--- fuer Zahlenbereich 1..9 -----*/
            block_ausgabe(zahl, 'I', 'V', 'X');
            strcat(romzahl, "\n");
            if (printf("%s", romzahl) == EOF)
                fehler_meld(FATAL_SYS, "printf-Fehler");
        } else {
            if (printf("ungueltige Eingabe\n") == EOF)
                fehler_meld(FATAL_SYS, "printf-Fehler");
        }
    }
}

```

```

/*----- block_ausgabe -----*/
*
* Diese Funktion gibt immer den entspr. Block einer Zahl aus.
* wert      ist die auszugebende Zehnerpotenz
* einer     enthaelt immer entspr. roemische Zehnerpotenz-Zeichen
*           I=1, X=10, C=100
* fuenfer   enthaelt immer entspr. roemische Zeichen von 5 * Zehnerpotenz
*           V=5, L=50, D=500
* zehner    enthaelt immer entspr. roemische Zeichen der naechsten
*           Zehnerpotenz, die groesser als der wert ist.
*           X=10, C=100, M=1000
*/
static void block_ausgabe(long int wert,
                          char einer, char fuenfer, char zehner)
{
    long int i;

    if (wert==9) {
        printf(romzahl, "%s%c%c", romzahl, einer, zehner);
    } else if (wert>4) {
        printf(romzahl, "%s%c", romzahl, fuenfer);
        for (i=wert ; i>=6 ; i--)
            printf(romzahl, "%s%c", romzahl, einer);
    } else if (wert==4) {
        printf(romzahl, "%s%c%c", romzahl, einer, fuenfer);
    } else {
        for (i=wert ; i>=1 ; i--)
            printf(romzahl, "%s%c", romzahl, einer);
    }
}

```

Programm 17.12 (**romzahl2.c**): Realisierung von *romzahl* mit Standard-E/A-Funktionen

Bei dem Programm 17.12 (*romzahl2.c*) besteht das Problem in der voreingestellten Standard-E/A-Pufferung. Das erste **fgets** auf die Standardeingabe (*stdin*) bewirkt das Anlegen eines Puffers, bei dem für die Standardeingabe die Vollpufferung voreingestellt ist, wenn sie nicht auf dem Terminal (hier Pipe) eingestellt ist. Dasselbe gilt auch für die Standardausgabe.

Während **romzahl** beim Lesen aus seiner Standardeingabe blockiert ist, ist *romkomm.c* beim Lesen aus der Pipe blockiert, und es liegt somit ein *Deadlock* vor. Dieses Problem kann beseitigt werden, indem man vor der **while**-Schleife die folgenden Codezeilen einfügt, um Zeilenpufferung einzustellen.

```

if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    fehler_meld(FATAL_SYS, "setvbuf-Fehler");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    fehler_meld(FATAL_SYS, "setvbuf-Fehler");

```

Verwendet man fertige Programme, zu denen man nicht die Quelldateien besitzt, als Koprozesse, so kann man diese Technik leider nicht anwenden. In diesem Fall muß man einen Trick anwenden, indem man den aufgerufenen Koprozess glauben läßt, seine Stan-

dardeingabe und -ausgabe sei auf ein Terminal (Pseudoterminal) eingestellt. Das bewirkt, daß die Standard-E / A mit Zeilenpufferung abläuft und man somit das Problem der Vollpufferung vermeidet.

17.3 Benannte Pipes (FIFOs)

Während normale Pipes nur zwischen Prozessen verwendet werden können, wenn ein gemeinsamer Vorfahre die entsprechende Pipe kreiert hat, können die sogenannten *FIFOs* zwischen beliebigen Prozessen zum Austauschen von Daten benutzt werden. FIFOs werden oft auch benannte Pipes (*named pipes*) genannt.

17.3.1 mkfifo – Kreieren einer benannten Pipe

Um eine benannte Pipe zu kreieren, steht die POSIX.1-Funktion **mkfifo** zur Verfügung.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pfadname, mode_t modus);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

mkfifo legt im Dateisystem eine Datei mit dem Namen *pfadname* an. Diese Datei ist keine normale Datei, sondern eine *FIFO*. Eine *FIFO* ist eine der verschiedenen Dateiartern, die in Kapitel 5.2 vorgestellt wurden, als die Komponente *st_mode* der *stat*-Struktur beschrieben wurde. Um zu überprüfen, ob eine Datei eine FIFO ist, kann das vordefinierte Makro *S_ISFIFO* verwendet werden.

Das Argument *modus* entspricht genau dem gleichnamigen Argument der Funktion **open** (siehe Kapitel 4.2). Für Eigentümer und Gruppen der neuen FIFO gelten die gleichen Regeln, die in Kapitel 5.3 vorgestellt wurden.

Nachdem man eine FIFO mit **mkfifo** kreiert hat, kann man diese mit **open** öffnen. Dann können die für normale Dateien angebotenen elementaren E / A-Funktionen (**read**, **write**, **close**, **unlink** usw.) für die FIFO verwendet werden.

17.3.2 Regeln für FIFO-Zugriffe

FIFOs sind eine spezielle Dateiartern, und es gelten die folgenden Regeln für Zugriffe auf FIFOs.

1. **open** für eine FIFO ohne *O_NONBLOCK*

Wenn beim Öffnen der FIFO mit **open** *O_NONBLOCK* nicht angegeben wird, was normalerweise der Fall ist, so wird ein **open** für »Nur-Lesen« (*modus* enthält *O_RDONLY*) so lange blockiert, bis ein anderer Prozeß diese FIFO zum Schreiben öffnet. Umgekehrt

wird ein **open** für »Nur-Schreiben« (modus enthält `O_WRONLY`) so lange blockiert, bis ein anderer Prozeß diese FIFO zum Lesen öffnet.

2. **open** für eine FIFO mit `O_NONBLOCK`

Ein **open** für »Nur-Lesen«, bei dem `O_NONBLOCK` gesetzt ist, kehrt sofort (ohne jegliche Blockierung) zurück. Ein **open** für »Nur-Schreiben«, bei dem `O_NONBLOCK` gesetzt ist, führt zu einem Fehler, wobei `errno` auf `ENXIO` gesetzt wird, wenn kein anderer Prozeß die FIFO zum Lesen geöffnet hat.

3. Schreiben in eine FIFO ohne Leser

Wenn in eine FIFO geschrieben wird, die momentan kein anderer Prozeß zum Lesen geöffnet hat, so wird wie bei Pipes das Signal `SIGPIPE` generiert.

4. Schließen einer FIFO durch letzten Schreiber

Wenn der letzte Prozeß, der eine FIFO zum Schreiben geöffnet hat, die FIFO schließt, so wird für den Leseprozeß ein `EOF` generiert.

5. Gleichzeitiges Schreiben in eine FIFO durch mehrere Prozesse

Wenn mehrere Prozesse gleichzeitig in dieselbe FIFO schreiben, dann ist sichergestellt, daß keinerlei Mischen der unterschiedlichen Daten stattfindet, solange mit einem **write** nicht mehr als `PIPE_BUF` Bytes auf einmal geschrieben werden.

17.3.3 **mkfifo** – Kommando zum Kreieren von FIFOs auf Shell-Ebene

Sowohl SVR4 als auch BSD-Unix bieten das Kommando **mkfifo** an. Dieses Kommando ermöglicht das Anlegen einer FIFO auf Shellebene. Auf diese FIFO kann dann mit `E/A`-Umlenkung zugegriffen werden.

Während Pipes auf Shellebene nur für *lineares Pipelining* verwendet werden können, können FIFOs auch für *nicht-lineares Pipelining* verwendet werden. Abbildung 17.10 veranschaulicht lineares und nicht-lineares Pipelining. FIFOs erlauben nicht-lineares Pipelining, da sie einen Namen besitzen.

Nehmen wir z.B. eine Anwendung, bei der am Monatsende alle Kunden aus einer Datei herauszufiltern sind, die ihre Rechnung nicht bezahlt haben. Für diese Kunden sollen zum einen Mahnungen geschrieben werden, zum anderen sollen für sie zugleich auch Adreßetiketten gedruckt werden. Die gefundenen zahlungssäumigen Kunden sollen also zugleich an zwei Programme **mahndruck** und **ettiketdruck** weitergeleitet werden.

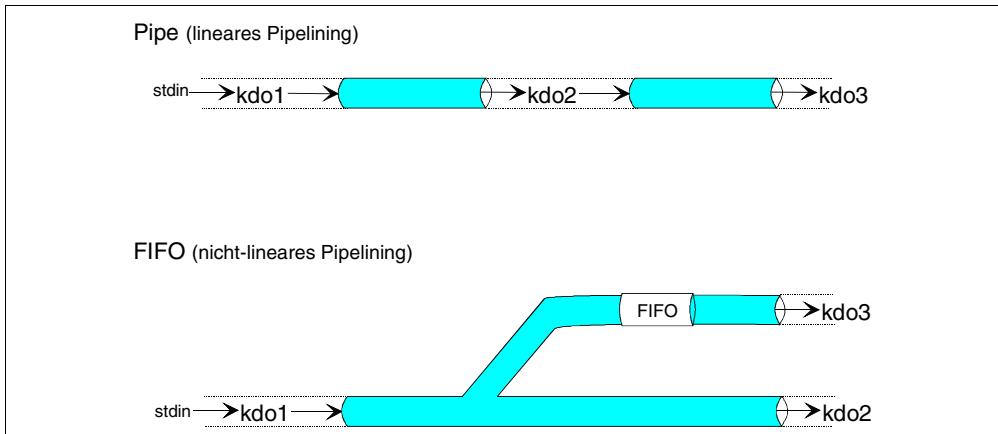


Abbildung 17.10: Lineares Pipelining (bei Pipe) und nicht-lineares Pipelining (bei FIFO)

Mit Pipes könnte diese Aufgabenstellung nur mittels einer temporären Datei gelöst werden, in der die herausgefilterten zahlungssäumigen Kunden zwischengespeichert werden. Wenn z.B. das Programm zum Herausfiltern der zahlungssäumigen Kunden den Namen **schuldner** hat, so sind die folgenden Kommandozeilen möglich:

```
$ schuldner <kundendatei >nicht_bezahlt
$ mahn_druck <nicht_bezahlt
$ ettiket_druck <nicht_bezahlt
```

oder

```
$ schuldner < kunden_datei | tee nicht_bezahlt | mahn_druck
$ ettiket_druck < nicht_bezahlt
```

Mit der Verwendung von FIFOs kommt man ohne eine temporäre Datei aus:

```
$ mkfifo nicht_bezahlt
$ mahn_druck <nicht_bezahlt &
$ schuldner <kundendatei | tee nicht_bezahlt | ettiket_druck
```

Hierbei wird zunächst die FIFO `nicht_bezahlt` kreiert und dann `mahn_druck`, das aus der FIFO `nicht_bezahlt` lesen soll, im Hintergrund gestartet. Danach wird `schuldner` aufgerufen, wobei die von diesem Programm herausgefilterten Kunden über eine Pipe an das **tee**-Kommando weitergeleitet werden.

Das **tee**-Kommando liest diese Kunden aus der Pipe und leitet sie zum einen an das Kommando `ettiket_druck` weiter, zum anderen schreibt es sie in die FIFO `nicht_bezahlt`, aus der sie nun auch das im Hintergrund ablaufende Programm `mahn_druck` liest. Abbildung 17.11 verdeutlicht dies.

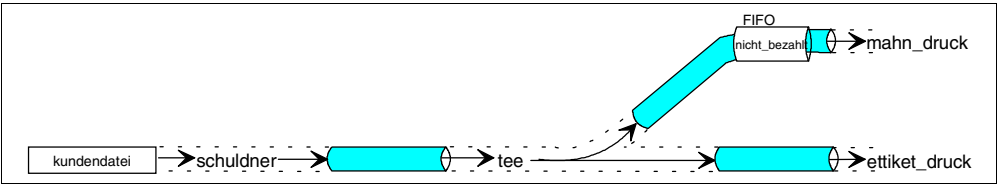


Abbildung 17.11: Verwendung von FIFO und **tee**-Kommando für nicht-lineares Pipelining

17.3.4 Verwendung von FIFOs zur Client-Server-Kommunikation

FIFOs werden oft zum Datenaustausch zwischen einem Client und einem Server verwendet. Dabei wird die folgende Vorgehensweise gewählt:

1. Schicken von Anforderungen durch Clients

Der Server kreiert eine FIFO, deren Name allen Clients bekannt ist. Jeder Client schreibt seine Anforderungen in diese FIFO, aus der sie dann der Server liest. Um ein Vermischen der einzelnen Client-Anforderungen zu vermeiden, sollten die Clients nie mehr als PIPE_BUF Bytes auf einmal in die FIFO schreiben. Abbildung 17.12 verdeutlicht dieses Schicken von Client-Anforderungen an den Server.

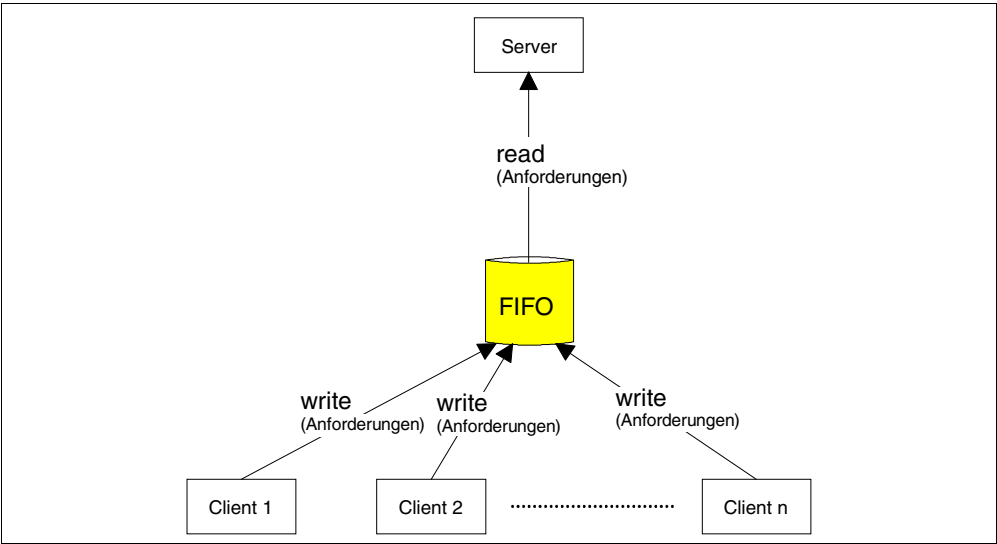


Abbildung 17.12: Schicken von Client-Anforderungen an den Server über eine FIFO

2. Antworten des Servers an die Clients

Für die Antwort des Servers auf die Client-Anforderung kann nicht eine einzige FIFO verwendet werden, da der einzelne Client nicht weiß, welche Antwort für ihn gedacht ist und wann er aus dieser FIFO lesen sollte. Deshalb muß für die Antworten eines

Servers zu jedem einzelnen Client eine eigene FIFO eingerichtet werden. Um eindeutige Namen für diese Server-Client-FIFOs zu garantieren, wird die Prozeß-ID der Clients in den FIFO-Namen verwendet. So kreiert z.B. der Server eine FIFO mit dem Namen `/tmp/server001.nnnn`, wobei `nnnn` für die Prozeß-ID des jeweiligen Clients steht. Jeder Client schickt dazu bei einer Anforderung seine Prozeß-ID mit.

Abbildung 17.13 zeigt die vollständige Struktur für die Client-Server-Kommunikation über FIFOs.

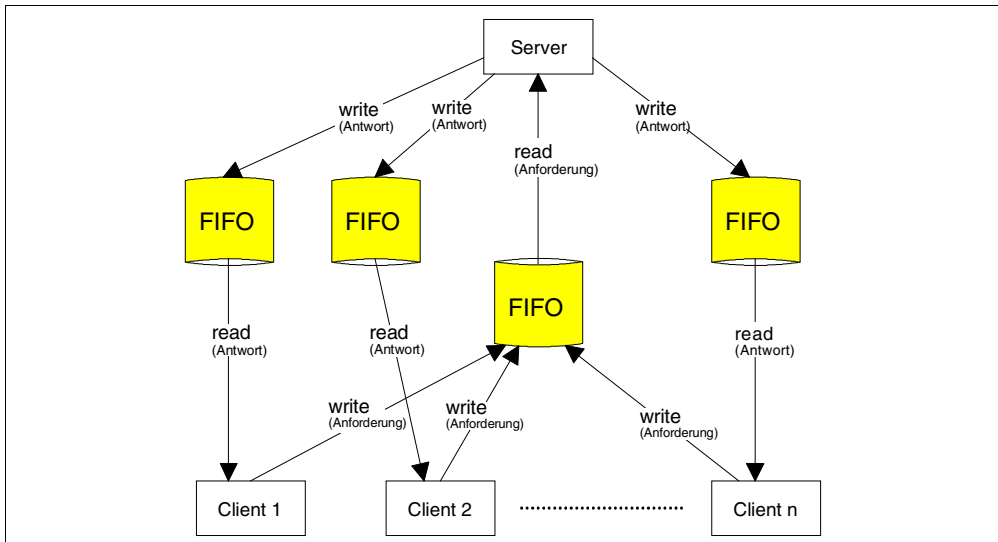


Abbildung 17.13: Struktur einer Client-Server-Kommunikation mit FIFOs

Bei dieser Kommunikationstechnik sind folgende Punkte zu beachten:

- Wenn ein Client sich vorzeitig beendet, ohne den Server darüber zu informieren, so verbleibt die client-spezifische Pipe im Dateisystem.
- Der Server muß das Signal `SIGPIPE` abfangen, da es möglich ist, daß ein Client eine Anforderung schickt, sich aber dann beendet, ohne die Antwort des Servers abzuwarten. Dies führt zu der sinnlosen Konstellation: FIFO mit Schreiber (Server), aber ohne Leser (Client hat sich beendet).
- Der Server sollte die gemeinsame FIFO, aus der er die Anforderungen aller Clients liest, nicht zum »Nur-Lesen« (`O_RDONLY`), sondern zum gleichzeitigen Lesen und Schreiben (`O_RDWR`) öffnen. So verhindert man, daß bei Beendigung des letzten Clients der Server ein EOF aus der FIFO liest, was eine besondere Behandlung dieses Falles durch Server notwendig machen würde.

Hinweis

Wenn die Clients nur Anforderungen an den Server schicken, aber keine Antworten von ihm erwarten, so reicht eine FIFO aus. Der Drucker-Spooler von System V verwendet diese Form der Client-Server-Realisierung. Dabei ist das **lp**-Kommando der Client und der Server der **lpsched**-Dämonprozeß. Hierbei wird nur eine einzige FIFO benutzt, da nur Daten vom Client zum Server fließen und keine in umgekehrter Richtung (vom Server **lpsched** zum Client **lp**).

17.4 Übung

17.4.1 Hexadump für Dateien (mit Eltern-Kind-Pipe)

Erstellen Sie ein Programm `hexd2.c`, das für alle auf der Kommandozeile angegebenen Dateien einen Hexadump durchführt. Dabei soll immer der Elternprozeß den Inhalt der jeweiligen Datei lesen und über eine Pipe an einen Kindprozeß schicken. Dieser Kindprozeß soll die Daten aus der Pipe lesen und sie dann in hexadezimaler und entsprechender ASCII-Darstellung ausgeben. Nicht darstellbare Zeichen soll der Kindprozeß durch einen Punkt bei der ASCII-Ausgabe anzeigen. Für jede auszugebende Datei soll der Elternprozeß immer einen neuen Kindprozeß kreieren.

Nachdem man dieses Programm `hexd2.c` kompiliert und gelinkt hat.

```
cc -o hexd2 hexd2.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ hexd2 hexd2.c fehler.c
---hexd2.c---
000000 23 69 6e 63 6c 75 64 65 20 20 20 3c 63 74 79 70 |#include <ctyp|
000010 65 2e 68 3e 0a 23 69 6e 63 6c 75 64 65 20 20 20 |e.h>.#include |
000020 3c 6c 69 6d 69 74 73 2e 68 3e 0a 23 69 6e 63 6c |<limits.h>.#incl|
000030 75 64 65 20 20 20 3c 73 79 73 2f 77 61 69 74 2e |ude <sys/wait.|
000040 68 3e 0a 23 69 6e 63 6c 75 64 65 20 20 20 22 65 |h>.#include "e|
000050 69 67 68 64 72 2e 68 22 0a 0a 73 74 61 74 69 63 |ighdr.h"..static|
000060 20 76 6f 69 64 20 20 68 65 78 5f 61 75 73 67 61 |void hex_ausga|
000070 62 65 28 46 49 4c 45 20 2a 64 7a 2c 20 63 68 61 |be(FILE *dz, cha|
000080 72 20 2a 64 61 74 65 69 6e 61 6d 65 29 3b 0a 0a |r *dateiname);..|
.....
.....
000910 6d 74 20 2b 3d 20 6e 3b 0a 20 20 20 20 20 20 7d |mt += n;. }|
000920 0a 20 20 20 20 20 20 66 66 6c 75 73 68 28 73 74 |. fflush(st|
000930 64 6f 75 74 29 3b 0a 20 20 20 20 20 20 63 6c 6f |dout);. clo|
000940 73 65 28 66 64 5b 30 5d 29 3b 0a 20 20 20 20 20 |se(fd[0]);. |
000950 20 65 78 69 74 28 30 29 3b 0a 20 20 20 7d 0a 7d |exit(0);. }|
000960 0a |.
---fehler.c---
000000 23 69 6e 63 6c 75 64 65 20 20 20 3c 65 72 72 6e 6f |#include <errno|
000010 2e 68 3e 0a 23 69 6e 63 6c 75 64 65 20 20 3c 73 |.h>.#include <s|
```

```

000020 74 64 61 72 67 2e 68 3e 0a 23 69 6e 63 6c 75 64 |tdarg.h>.#includ|
000030 65 20 20 3c 73 79 73 6c 6f 67 2e 68 3e 0a 23 69 |e <syslog.h>.#i|
000040 6e 63 6c 75 64 65 20 20 22 65 69 67 68 64 72 2e |nclude "eighdr.|
000050 68 22 0a 0a 69 6e 74 20 20 64 65 62 75 67 3b 20 |h"..int debug; |
000060 2f 2a 20 41 75 66 72 75 66 65 72 20 76 6f 6e 20 |/* Aufrufer von |
000070 6c 6f 67 5f 6d 65 6c 64 20 6f 64 65 72 20 6c 6f |log_meld oder lo|
000080 67 5f 6f 70 65 6e 20 6d 75 73 73 20 64 65 62 75 |g_open muss debu|
.....
000a80 6e 75 6e 67 2c 20 69 6e 74 20 6f 70 74 69 6f 6e |nung, int option|
000a90 2c 20 69 6e 74 20 66 61 63 69 6c 69 74 79 29 0a |, int facility).|
000aa0 7b 0a 20 20 20 69 66 20 28 64 65 62 75 67 3d 3d ||. if (debug==|
000ab0 30 29 0a 20 20 20 20 20 20 6f 70 65 6e 6c 6f 67 |0). openlog|
000ac0 28 6b 65 6e 6e 75 6e 67 2c 20 6f 70 74 69 6f 6e |(kennung, option|
000ad0 2c 20 66 61 63 69 6c 69 74 79 29 3b 0a 7d 0a |, facility);. |
$

```

17.4.2 Starten eines Koprozesses ohne Signalhandler

Entfernen Sie im Programm 17.11 (*romkomm.c*) den Signalhandler *sig_pipe*, starten Sie dieses Programm und beenden Sie dann den Kindprozeß. Wenn man nun eine Zahl eingibt, so beendet sich der Elternprozeß (*romkomm*). Wie kann man nun nachträglich feststellen, daß der Elternprozeß durch das Signal *SIGPIPE* beendet wurde?

17.4.3 Lesen und Schreiben in einer Pipe mit Standard-E/A-Funktionen

Was muß im Programm 17.11 (*romkomm.c*) geändert werden, damit man anstelle von *read* und *write* die Standard-E/A-Funktionen verwenden kann, um aus der Pipe zu lesen bzw. in sie zu schreiben?

17.4.4 Implementierung von *popen* und *pclose*

Erstellen Sie ein Programm *popen.c*, das mögliche Implementierungen der beiden Funktionen *popen* und *pclose* enthält.

17.4.5 Parallele Matrizenmultiplikation durch mehrere Kindprozesse

Erstellen Sie ein Programm *matmult.c*, das eine Multiplikation von zwei Matrizen durchführt. Für dieses Programm soll folgendes gelten:

- Die Deklarationen aller Matrizen können modulglobal sein.
- Für jedes Element der Ergebnismatrix ist ein Kindprozeß zu erzeugen, dem über eine Pipe der Zeilen- und Spaltenindex des zu berechnenden Elements der Ergebnismatrix mitgeteilt wird. Nachdem der jeweilige Kindprozeß diese Indizes aus der Pipe gelesen hat, muß er – unter Zugriff auf die modulglobalen Eingabematrizen – dieses Element berechnen und dem Elternprozeß über eine zweite Pipe dieses Ergebniselement zukommen lassen.

- Der Elternprozeß gibt zunächst die beiden Eingabematrizen aus, und wartet dann auf die Ankunft aller Ergebnisse (aus den Antwort-Pipes), bevor er die vollständige Ergebnismatrix ausgibt.

17.4.6 Kein Schließen der Schreibseite einer Pipe

Was passiert, wenn man im Programm 17.4 (`primfak.c`) das ***fclose*** vor dem ***waitpid*** (im Elternprozeß) entfernt?

17.4.7 Gleichzeitiges Schreiben der Standardausgabe und -fehlerausgabe in Pipe

Was ist zu tun, wenn man mit ***popen*** eine Pipe (mit »r« für typ) zu einem Programm einrichtet, das sowohl auf die Standardausgabe als auch auf die Standardfehlerausgabe schreibt und man alle diese Ausgaben aus der Pipe lesen möchte?

18 Message-Queues, Semaphore und Shared Memory

*Der eine geht zum Nächsten,
weil er sich sucht,
und der andre,
weil er sich verlieren möchte.*

Nietzsche

In diesem Kapitel werden drei Methoden der Interprozeßkommunikation (IPC) vorgestellt:

- ▶ Austausch von Nachrichten zwischen Prozessen (*Message-Queues*)
- ▶ Synchronisation über *Semaphore*
- ▶ Austausch von Daten über gemeinsame Speicherbereiche (*Shared Memory*)

Zunächst wird auf die allen drei Methoden zugrundeliegenden Strukturen und Eigenschaften eingegangen, bevor die Methoden und die zugehörigen Funktionen im einzelnen vorgestellt werden.

18.1 Allgemeine Strukturen und Eigenschaften

Die drei Methoden verwenden unterschiedliche Objekte zur Interprozeßkommunikation. Die Objekte sind dabei Nachrichtenwarteschlangen (*Message-Queues*), gemeinsame Hauptspeicherbereiche (*Shared Memory*) oder sogenannte *Semaphore*. Während sich die Objekte für jede einzelne Methode unterscheiden, ist die Verwaltung dieser Objekte jedoch weitgehend vereinheitlicht.

18.1.1 Kennungen und Schlüssel

Kennung (Identifiers)

Jedem Objekt wird intern vom Kern eine eindeutige Kennung in Form einer nichtnegativen Zahl zugeteilt. Diese Zuteilung der Kennung erfolgt beim Einrichten eines Objekts. Anders als bei Filedeskriptoren, bei denen immer nur kleine Nummern verwendet werden, können diese Kennungszahlen auch sehr groß werden. Der Grund dafür liegt in der Tatsache, daß Kennungszahlen von gelöschten Objekten nicht wieder neu vergeben werden, sondern bei jeder Neueinrichtung eines solchen Objekts, unabhängig davon, ob kleinere Nummern frei geworden sind, wird immer weiter hochgezählt. Wird der maximal mögliche Wert erreicht, so beginnt die Zählung wieder bei 0.

Schlüssel (Key)

Immer wenn ein neues Objekt mit einer der Funktionen **msgget**, **semget** oder **shmget** eingerichtet wird, muß ein sogenannter Schlüssel angegeben werden. Dieser Schlüssel hat den Datentyp `key_t`, der meist in `<sys/types.h>` als **long** definiert ist. Der Schlüssel, der vom Kern mit der entsprechenden Kennung verbunden wird, bietet auch nicht verwandten Prozessen die Möglichkeit, ein bestimmtes (über den Schlüssel spezifiziertes) Objekt gemeinsam zu benutzen. Alle Prozesse, die den Schlüssel kennen, können nämlich so auf das gleiche Objekt zugreifen, obwohl ihnen dessen Kennung eventuell nicht bekannt ist.

Objekte, denen kein Schlüssel zugeordnet ist, werden als *private Objekte* bezeichnet. Beim Einrichten eines solchen Objekts muß anstelle eines Schlüssels die Konstante `IPC_PRIVATE` angegeben werden. Die Angabe von `IPC_PRIVATE` bewirkt in jedem Fall, daß ein neues Objekt angelegt wird.

18.1.2 Kommunikationsmöglichkeiten von nicht verwandten Prozessen

Es existieren verschiedene Möglichkeiten für nicht verwandte Prozesse, ein gemeinsames Objekt zur Kommunikation zu verwenden:

1. Der Prozeß A (Server) kreiert ein neues Objekt, indem er als Schlüssel die Konstante `IPC_PRIVATE` angibt. Die zurückgegebene Kennung speichert er dann an einer vereinbarten Stelle (wie z.B. in einer Datei) ab, aus der sie die Prozesse (Clients) B, C usw. lesen können, um dann unter Angabe dieser Kennung auf das Objekt zuzugreifen. Der Nachteil dieser Vorgehensweise ist die nicht sehr effiziente Kommunikation (Austausch der Kennung) über Dateien.

Der Schlüssel `IPC_PRIVATE` kann im übrigen auch bei Eltern-Kind-Beziehungen benutzt werden. Der Elternprozeß kreiert ein neues Objekt (mit `IPC_PRIVATE`) und merkt sich die zurückgegebene Kennung in einer Variablen. Beim Kreieren des Kindprozesses mit **fork** wird diese Kennung an den Kindprozeß vererbt, der diese Kennung z.B. auch einem anderen Programm verfügbar machen kann, indem er sie als Argument bei einem **exec**-Aufruf angibt.

2. Clients und der Server können einen gemeinsamen Schlüssel vereinbaren (z.B. in einer Headerdatei). Der Server kreiert dann ein neues Objekt mit diesem Schlüssel, über den die Clients auf dieses Objekt zugreifen können. Der Nachteil dieser Methode ist allerdings, daß der vereinbarte Schlüssel eventuell vorher schon an ein anderes Objekt vergeben wurde. In diesem Fall schlägt das Einrichten des Objekts (mit **msgget**, **semget** oder **shmget**) fehl. Der Server muß diesen Fehler erkennen, das bereits existierende Objekt löschen und dann erst kann er ein neues Objekt mit dem nun frei gewordenen Schlüssel kreieren.

18.1.3 Einrichten eines neuen Objekts

Um ein neues Objekt einzurichten, stehen die Funktionen **msgget**, **semget** und **shmget** zur Verfügung. Alle drei Funktionen haben neben dem Schlüssel noch ein weiteres gemeinsames Argument `flag`. Ein neues Objekt kann auf zwei verschiedene Arten kreiert werden:

1. Angabe von `IPC_PRIVATE` als Schlüssel oder
2. Angabe eines noch nicht existierenden Schlüssels, wobei im aktuellen `flag`-Argument `IPC_CREAT` gesetzt ist. Um sicherzustellen, daß beim Einrichten eines Objekts wirklich ein neues Objekt angelegt und nicht ein bereits existierendes Objekt mit der gleichen Kennung angesprochen wird, muß im aktuellen `flag`-Argument sowohl `IPC_CREAT` als auch `IPC_EXCL` gesetzt sein. Falls in diesem Fall das Objekt bereits existiert, kehrt die entsprechende Funktion mit einem Fehler zurück, wobei `errno` auf `EEXIST` gesetzt wird.

18.1.4 Herstellen einer Verbindung zu einem existierenden Objekt

Um zu einem bereits existierenden Objekt eine Verbindung herzustellen, was Clients meist tun, muß beim Aufruf der entsprechenden Funktion (**msgget**, **semget** oder **shmget**) der gleiche Schlüssel verwendet werden, der beim Kreieren des Objekts angegeben wurde. Im `flag`-Argument darf dabei `IPC_CREAT` nicht gesetzt sein.

Zum Anzeigen von existierenden IPC-Objekte und ihres Status, steht das Kommando **ipcs** zur Verfügung.

18.1.5 Löschen von Objekten

Ein Objekt existiert so lange, bis es explizit gelöscht wird oder ein Shutdown für das System erfolgt. Ein Objekt kann von jedem Benutzer mit den entsprechenden Zugriffsrechten oder aber immer vom Objekteinrichter bzw. -Eigentümer gelöscht werden.

Zum Löschen von IPC-Objekten können die Funktionen **msgctl**, **semctl** und **shmctl** verwendet werden. Dazu muß dort für das `kdo`-Argument `IPC_RMID` ausgegeben werden.

Darüber hinaus steht zum Löschen eines Objekts das Kommando **ipcrm** zur Verfügung.

18.1.6 Zugriffsrechte

Zu jedem Objekt existiert die Struktur `ipc_perm` (definiert in `<sys/ipc.h>`), die Zugriffsrechte für dieses Objekt und dessen Eigentumsverhältnisse festlegt:

```
struct ipc_perm {
    uid_t  uid; /* effektive User-ID des Eigentümers          */
    gid_t  gid; /* effektive Group-ID des Eigentümers          */
    uid_t  cuid; /* effektive User-ID des Objekt-Einrichters    */
    gid_t  cgid; /* effektive Group-ID des Objekt-Einrichters   */
    mode_t mode; /* Zugriffsmodus                               */
    ulong  seq; /* Kennung                                       */
    key_t  key; /* Schlüssel                                    */
}
```

Bis auf die Komponente `seq` werden alle anderen Komponenten beim Einrichten des Objekts initialisiert. Nach dem Einrichten eines Objekts sind die User-ID und Group-ID des Eigentümers und des Objekteinrichters identisch. Später können dann die Komponenten `uid`, `gid` und `mode` mit den Funktionen ***msgctl***, ***semctl*** oder ***shmctl*** vom Objekteinrichter oder Superuser geändert werden. Dazu muß bei diesen Funktionen für das `kdo`-Argument `IPC_SET` angegeben werden.

Die Zugriffsrechte in der `mode`-Komponente sind ähnlich zu der Komponente `st_mode` in der `stat`-Struktur, die Zugriffsrechte für Dateien enthält (siehe Tabelle 5.2). Anders als dort existieren für die IPC-Objekte keine Ausführrechte. Tabelle 18.1 zeigt die sechs möglichen Zugriffsrechte für jede der drei Objektarten. Während bei *Message-Queues* und *Shared Memory* die Begriffe *read* und *write* verwendet werden, werden bei Semaphore die Begriffe *read* und *alter* (Ändern) benutzt.

Zugriffsrecht	Message-Queues	Semaphore	Shared Memory
user-read	MSG_R	SEM_R	SHM_R
user-write (alter)	MSG_W	SEM_A	SHM_W
group-read	MSG_R >> 3	SEM_R >> 3	SHM_R >> 3
group-write (alter)	MSG_W >> 3	SEM_A >> 3	SHM_W >> 3
other-read	MSG_R >> 6	SEM_R >> 6	SHM_R >> 6
other-write (alter)	MSG_W >> 6	SEM_A >> 6	SHM_W >> 6

Tabelle 18.1: Zugriffsrechte für Message Queues, Semaphore und Shared Memory

18.1.7 Limits

Für alle drei Objektarten (Message-Queues, Semaphore und Shared Memory) sind Limits festgelegt. In SVR4 sind diese minimalen und maximalen Limits in der Datei `/etc/conf/cf.d/mtune` angegeben.

Die meisten Limits können nur durch eine Neukonfigurierung des Kerns geändert werden. Auf die einzelnen Limits wird in den nachfolgenden Kapiteln bei der Vorstellung der einzelnen Objektarten noch genauer eingegangen.

18.2 Message-Queues

Message-Queues (Nachrichtwarteschlangen) werden im Kern in Form einer verketteten Liste verwaltet. Zu jeder Message-Queue existiert eine Kennung (*Message-Queue Identifier*).

Um eine Message-Queue einzurichten oder aber eine bereits existierende zu öffnen, muß die Funktion ***msgget*** verwendet werden. Um Messages zu schicken, steht die Funktion ***msgsnd*** zur Verfügung, die die entsprechende Message am Ende der betreffenden Message-Queue anhängt.

Jede Message setzt sich aus den folgenden 3 Komponenten zusammen:

- Message-Typ (Datentyp **long**)
- Länge der Message (Datentyp **size_t**)
- Message-String

Um Messages aus einer Message-Queue zu empfangen, steht die Funktion **msgrcv** zur Verfügung. Die Messages müssen dabei nicht in der Reihenfolge aus der Message-Queue gelesen werden, in der sie dort eingetragen wurden. Unter Angabe eines entsprechenden Message-Typs kann auch eine Message von einer beliebigen Stelle in der Warteschlange empfangen werden.

18.2.1 msqid_ds – Status einer Message-Queue

Zu jeder Message-Queue existiert eine **msqid_ds**-Struktur, die den momentanen Status der Message-Queue festlegt:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* in Kapitel 18.1 beschrieben */
    struct msg *msg_first; /* Adr. der 1. Message in queue */
    struct msg *msg_last; /* Adr. der letzten Message in queue */
    ulong msg_cbytes; /* Anzahl der Bytes in Message Queue */
    ulong msg_qnum; /* Anzahl der Messages in Message Queue */
    ulong msg_qbytes; /* max.Anzahl von Bytes in Message Queue*/
    pid_t msg_lspid; /* PID des letzten msgsnd-Aufrufers */
    pid_t msg_lrpid; /* PID des letzten msgrcv-Aufrufers */
    time_t msg_stime; /* Zeitpunkt des letzten msgsnd-Aufrufs */
    time_t msg_rtime; /* Zeitpunkt des letzten msgrcv-Aufrufs */
    time_t msg_ctime; /* Zeitpunkt der letzten Änderung */
}
```

Eine Message Queue ist als lineare Liste realisiert, auf deren erstes Element **msg_first** und auf deren letztes Element **msg_last** zeigt. Der Zeiger **msg_last** wird benutzt, um Sendoperationen schneller ausführen zu können, denn so kann eine neue Nachricht sofort am Ende der Message Queue eingehängt werden, ohne daß zuerst alle Elemente der Warteschlange durchlaufen werden müssen, um das Ende zu finden.

Eine Nachricht wird im Systemkern in der Struktur **msg** gespeichert, die unter Linux z.B. das folgende Aussehen hat:

```
struct msg {
    struct msg *msg_next; /* Naechste Nachricht in Message Queue */
    long msg_type; /* Typ der Nachricht */
    char *msg_spot; /* Adresse des Textes der Nachricht */
    time_t msg_stime; /* msgsnd time */
    short msg_ts; /* Laenge der Nachricht */
};
```

Da Linux die Nachricht direkt hinter dieser Struktur speichert, ist die Komponente **msg_spot** eigentlich überflüssig.



Unter Linux enthält die Struktur `msqid_ds` noch zwei weitere Komponenten:

```
struct wait_queue *wwait;  
struct wait_queue *rwait;
```

In die Warteschlange `wwait` wird ein Prozeß eingetragen, wenn die Message Queue voll ist, was bedeutet, daß ein Senden der Nachricht nicht mehr möglich ist, da in diesem Fall die maximal erlaubte Anzahl von Bytes in der Message Queue überschritten würde.

Die Warteschlange `rwait` enthält Prozesse, die darauf warten, daß Nachrichten in die Warteschlange geschrieben werden.



18.2.2 Limits einer Message-Queue

Für Message-Queues sind die folgenden Limitkonstanten definiert:

MSGMAX

Maximale Anzahl von Bytes, die eine geschickte Message enthalten kann (typischer Wert: 2048).

MSGMNB

Maximale Anzahl von Bytes in einer Message-Queue (typischer Wert: 4096).

MSGMNI

Maximale Anzahl von Message-Queues im System (typischer Wert: 50).

MSGTQL

Maximale Anzahl von Messages im System.

18.2.3 msgget – Öffnen oder Kreieren einer Message-Queue

Um eine existierende Message-Queue zu öffnen oder eine neue Message-Queue zu kreieren, steht die Funktion ***msgget*** zur Verfügung.

```
#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/msg.h>  
  
int msgget(key_t schlüssel, int flag);
```

gibt zurück: Kennung der Message-Queue (bei Erfolg); -1 bei Fehler

In Kapitel 18.1 wurde ausführlich beschrieben, wann ein neues Objekt (hier Message-Queue) eingerichtet wird und wann ein bereits existierendes geöffnet wird.

Wenn eine neue Message-Queue eingerichtet wird, so werden die folgenden Komponenten der `msgid_ds`-Struktur initialisiert:

`msg_perm`

(siehe Zugriffsrechte in Kapitel 18.1). Die Komponente `mode` der Struktur `ipc_term` wird mit den entsprechenden im `flag`-Argument angegebenen Zugriffsrechten gesetzt. Die Rechte können dabei mit den in Tabelle 18.1 angegebenen Konstanten spezifiziert werden.

```
msg_qnum = 0
msg_lspid = 0
msg_lrpid = 0
msg_stime = 0
msg_rtime = 0
msg_ctime = momentane Zeit
msg_qbytes = MSGMNB
```

Bei erfolgreichem Aufruf liefert ***msgget*** die Kennung der entsprechenden Message-Queue (nichtnegativer ***int***-Wert) als Rückgabewert. Dieser Wert kann dann bei den nachfolgenden drei Funktionen benutzt werden.

18.2.4 msgsnd – Senden von Messages

Um Messages zu senden (in der Message-Queue einzutragen), steht die Funktion ***msgsnd*** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int kennung, const void *puffer, size_t mlaenge, int flag);

gibt zurück: 0 (bei Erfolg); -1 bei Fehler
```

Eine Message setzt sich aus den folgenden drei Komponenten zusammen:

- ▶ Message-Typ (Datentyp ***long***)
- ▶ Länge der Message (Datentyp `size_t`)
- ▶ Message-String

Eine mit ***msgsnd*** geschickte Message wird immer am Ende der betreffenden Message-Queue angehängt.

kennung

ist die Message-Queue, an die die entsprechende Message zu schicken ist.

puffer und mlaenge

`puffer` enthält die Adresse des Message-Typs (Datentyp **long**). Direkt auf den Message-Typ folgt der eigentliche Message-Text, dessen Länge über `mlaenge` spezifiziert ist. Bei einem leeren Message-Text muß für `mlaenge` der Wert 0 angegeben werden.

Wenn z.B. bekannt ist, daß der längste mögliche Message-Text niemals größer als 256 Byte ist, könnte man sich folgende Struktur definieren:

```
struct meine_mesg {
    long mtype;      /* Message-Typ */
    char mtext[256]; /* Message-Text */
}
```

In diesem Fall würde man als Argument für `puffer` die Adresse einer Variablen dieses Typs (`struct meine_mesg`) angeben. Diese Variable würde den Message-Typ und Message-Text enthalten. Der Message-Typ ist nur von Interesse, wenn man Messages (mit **msgrcv**) in einer anderen Reihenfolge empfangen möchte, als sie an die Message-Queue gesendet wurden.

flag

Wenn eine Message-Queue voll ist, wird **msgsnd** normalerweise solange blockiert, bis einer der folgenden Fälle zutrifft:

1. Es ist genug Platz für die einzutragende Message vorhanden.
2. Die Message-Queue wird gelöscht. In diesem Fall beendet sich **msgsnd** mit einem Fehler, wobei `errno` auf `EIDRM` gesetzt wird.
3. Ein Signal unterbricht den Wartezustand. In diesem Fall beendet sich **msgsnd** mit einem Fehler, wobei `errno` auf `EINTR` gesetzt wird.

Soll ein **msgsnd**-Aufruf bei einer vollen Message-Queue nicht so lange blockiert werden, bis einer der obigen drei Fälle eintritt, so muß im `flag`-Argument die Konstante `IPC_NOWAIT` gesetzt werden. **msgsnd** kehrt dann bei einer vollen Message-Queue sofort mit einem Fehler zurück, wobei `errno` auf `EAGAIN` gesetzt wird.

Bei einem erfolgreichen Senden einer Message mit **msgsnd** werden in der Struktur `msgid_ds` dieser Message-Queue die entsprechenden Komponenten aktualisiert.

18.2.5 msgrcv – Empfangen von Messages

Um Messages zu empfangen, steht die Funktion **msgrcv** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgrcv(int kennung, void *puffer, size_t maxlaenge, long typ, int flag);
```

gibt zurück: Länge der empfangenen Message (bei Erfolg); -1 bei Fehler

kennung

ist die Message-Queue, von der eine Message zu empfangen ist.

puffer und maxlaenge

puffer gibt die Adresse an, an die der Message-Typ (Datentyp **long**) und direkt daran anschließend der eigentliche Message-Text zu schreiben ist. Das Argument *maxlaenge* legt die maximale Länge des Message-Textes (*Puffergröße* - *sizeof(long)*) fest.

Wenn die empfangene Message mehr als *maxlaenge* Bytes hat, dann kehrt die Funktion **msgrcv** mit einem Fehler zurück, wobei *errno* auf **E2BIG** gesetzt wird. In diesem Fehlerfall verbleibt die betreffende Message in der Message-Queue. Ist dagegen beim Aufruf von **msgrcv** im *flag*-Argument **MSG_NOERROR** gesetzt, so werden die überzähligen Bytes einfach abgeschnitten, ohne daß der Aufrufer darüber informiert wird.

typ

Dieses Argument legt den Typ der zu empfangenden Message fest:

typ == 0

Erste Message aus der Message-Queue (FIFO-Prinzip).

typ > 0

Erste Message aus der Message-Queue, die den Typ *typ* hat. Ist jedoch das Flag **MSG_EXCEPT** gesetzt, wird die erste Nachricht empfangen, die nicht den Typ *typ* hat

typ < 0

Erste Message aus der Message-Queue, deren Typ der kleinste Wert ist, der kleiner oder gleich dem absoluten Betrag von *typ* ist.

Der Message-Typ kann z.B. benutzt werden, um Prioritäten an die Messages zu vergeben.

Client-Server-Anwendungen, bei denen nur eine Message-Queue für die Kommunikation zwischen Server und vielen Clients existiert, benutzen die Prozeß-ID als Message-Typ zur Identifikation des entsprechenden Clients.

flag

Wenn keine Message bzw. keine Message des geforderten typ in der Message-Queue ist, so wird **msgrcv** normalerweise solange blockiert, bis einer der folgenden Fälle zutrifft:

1. Eine Message des geforderten Typs ist verfügbar.
2. Die Message-Queue wird gelöscht. In diesem Fall beendet sich **msgrcv** mit einem Fehler, wobei `errno` auf `EIDRM` gesetzt wird.
3. Ein Signal unterbricht den Wartezustand. In diesem Fall beendet sich **msgrcv** mit einem Fehler, wobei `errno` auf `EINTR` gesetzt wird.

Soll ein **msgrcv**-Aufruf beim Nichtvorhandensein der geforderten Message nicht blockiert werden, so muß im `flag`-Argument `IPC_NOWAIT` gesetzt werden. **msgrcv** kehrt dann beim Nichtvorhandensein der geforderten Message sofort zurück, wobei `errno` auf `ENMSG` gesetzt wird.

Konnte eine Message erfolgreich empfangen werden, so werden in der Struktur `msqid_ds` dieser Message-Queue die entsprechenden Komponenten aktualisiert.

18.2.6 msgctl – Abfragen/Ändern des Status oder Löschen einer Message Queue

Um den Status einer Message-Queue zu erfragen oder zu ändern oder aber eine Message-Queue zu löschen, steht die Funktion **msgctl** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgctl(int kennung, int kdo, struct msqid_ds *puffer);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Das `kdo`-Argument legt die durchzuführende Aktion fest:

IPC_STAT

Abfragen des Status der Message-Queue

msgctl schreibt diese Statusinformation an die Adresse `puffer`.

IPC_SET

Setzen der Eigentümer-UID/GID, der Zugriffsrechte und maximalen Größe der Message-Queue

Im übergebenen `puffer` befinden sich die zu setzenden Werte, wobei jedoch nur die folgenden Komponenten relevant sind: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` und `msg_qbytes`.

IPC_SET kann jedoch nur von einem Prozeß verwendet werden, dessen effektive User-ID gleich msg_perm.cuid oder gleich msg_perm.uid ist, oder aber von einem Superuser-Prozeß. Ein Erhöhen des Wertes von msg_qbytes ist nur dem Superuser gestattet.

IPC_RMD

Löschen der Message-Queue mit allen ihren Daten

Dieses Löschen erfolgt sofort. Andere Prozesse, die die Message-Queue noch benutzen, erhalten bei ihrem nächsten Zugriff auf diese Message-Queue einen Fehler, wobei errno auf EIDRM gesetzt wird.

IPC_RMID kann jedoch nur von einem Prozeß ausgeführt werden, dessen effektive User-ID gleich msg_perm.cuid oder gleich msg_perm.uid ist, oder aber von einem Superuser-Prozeß.



Unter Linux kann für kds noch IPC_INFO angegeben werden, um Informationen über die entsprechende Message Queue zu erfragen. Diese Informationen werden in die einzelnen Komponenten der Struktur msginfo (an Adresse puffer) eingetragen:

```
struct msginfo {
    int msgpool; /* Anzahl der benutzten Message Queues;          */
                  /* von Linux ignoriert                                     */
    int msgmap; /* Anzahl der Eintraege in einer Message-Map          */
    int msgmax; /* Maximale Groesse einer Nachricht in Bytes              */
    int msgmnb; /* Voreingestellte maximale Groesse einer Message*       */
    int msgmni; /* Maximale Anzahl von Message-Queue-Kennungen           */
    int msgssz; /* Groesse eines Message-Segments;                         */
                  /* von Linux ignoriert                                     */
    int msgtql; /* Max. Anzahl von Segmenten; von Linux ignoriert*       */
    ushort msgseg;

};
```

Zum Setzen dieser Komponenten sind in <sys/msg.h> bzw. <linux/msg.h> eigene Konstanten definiert, wie z.B.:

```
#define MSGMAP MSGMNB /* number of entries in message map */
#define MSGMAX 4056 /* <= 4056 */ /* max size of message (bytes) */
#define MSGMNB 16384 /* ? */ /* default max size of a message queue */
#define MSGMNI 128 /* <= 1K */ /* max # of msg queue identifiers */

/* unused */
#define MSGPOOL (MSGMNI*MSGMNB/1024) /* size in kilobytes of message pool */
#define MSGSSZ 16 /* message segment size */
#define MSGTQL MSGMNB /* number of system message headers */
#define __MSGSEG ((MSGPOOL*1024)/ MSGSSZ) /* max no. of segments */
#define MSGSEG (__MSGSEG <= 0xffff ? __MSGSEG : 0xffff)
```

Unter Linux 2.0 werden Message Queues verwendet, um mit dem `kernel`-Dämon, der für das automatische Laden von Kernmodulen zuständig ist, zu kommunizieren. Über Messages fordern die Kernroutinen die entsprechenden Kernmodule an.



18.2.7 Client-Server-Implementierung mit Message-Queues

Nachfolgend wird eine Client-Server-Implementierung auf Basis von Message-Queues gezeigt. Dabei ist das Programm 18.1 (`mqdivser.c`) der Serverprozeß, der eine Division mit beliebiger Genauigkeit für ganze Zahlen durchführt. Die Zahlen und die geforderte Genauigkeit erhält er dabei von dem Programm 18.2 (`mqdivcli.c`), das die Client-Implementierung darstellt. Bei jedem Start von `mqdivcli.c` wird ein neuer Clientprozeß zum Serverprozeß (`mqdivser.c`) eingerichtet. Jeder dieser Clientprozesse richtet eine private Message-Queue zum Server ein. Während alle Clients ihre Berechnungswünsche über ein und dieselbe Message-Queue an den Server schicken, empfangen sie die vom Server berechneten Ergebnisse über ihre privaten Message-Queues. Abbildung 18.1 verdeutlicht dies.

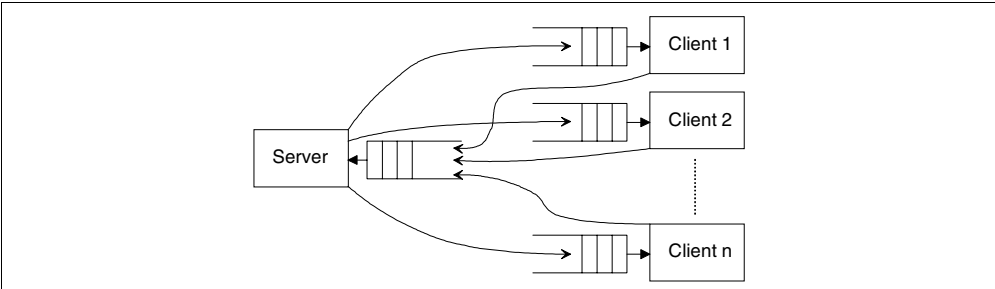


Abbildung 18.1: Client-Server-Modell mit Message-Queues

Jede Clientanforderung (Message) setzt sich aus folgenden Daten zusammen.

Message-Typ	client_kennung	genauigkeit	divident	divisor

Als Message-Typ wird dabei immer die Client-Nummer geschickt, wobei die Client-Nummer 1000 den Server darüber informiert, daß es sich nun beenden soll.

Die vom Server und den Clients gemeinsam benutzten Konstanten und Strukturen sind in der Headerdatei `mq.h` definiert.

```
#ifndef MQ
#define MQ

/*--- Vereinbarung Server-Schlüssel zwischen Server und Clients */
```



```

#define SERVER_KEY          10001

/*--- Maximale Laenge einer Nachricht -----*/
#define MAX_LAENGE          200

/*---- Datentypen fuer Client-Anforderungen und Serverantwort -----*/
typedef struct {
    long mtyp;
    char nachricht[MAX_LAENGE];
} mqu_anforderung;

typedef struct {
    long mtyp;
    char ergebnis[MAX_LAENGE];
} mqu_antwort;

#endif

```

Programm 18.1 Headerdatei **mq.h**: Gemeinsame Konstanten und Strukturen im Server und den Clients

Das Programm 18.1 (`mqdivserver.c`) ist der Server, der eine Message-Queue für alle Client-Anforderungen einrichtet. Als Schlüssel für diese Message-Queue wird dabei die in der Headerdatei `mq.h` definierte Konstante `SERVER_KEY` benutzt. Dann liest der Server aus dieser Message-Queue nacheinander alle von den Clients anstehenden Anforderungen, berechnet das entsprechende Ergebnis und schickt dieses an den entsprechenden Client über dessen private Message-Queue zurück. Die Kennung dieser Message-Queue hat ihm der Client in seiner Anforderung mitgeschickt. Eine solche Vorgehensweise nennt man auch *verbindungsloses Protokoll*. Bei der anderen Protokollart, dem *verbindungsorientierten Protokoll*, meldet sich zu Beginn jeder Client beim Server an und teilt ihm seine Kennung mit. Der Server vergibt dann eine Nummer an diesen Client. Unter Bezugnahme auf diese Clientnummer gibt später der Client seine Anforderungen an den Server ab. Am Ende meldet der Client sich wieder beim Server ab. Diese Vorgehensweise beim verbindungsorientiertem Protokoll entspricht in etwa einem *open* und *close* für eine Message-Queue.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include "eighdr.h"
#include "mq.h"

int
main(void)
{
    int                server_kennung, client_kennung;
    mqu_anforderung    anforderung;
    mqu_antwort        antwort;
    int                genauigkeit, dividend, divisor, quotient, i;

    /*--- Einrichten der Server-Message Queue -----*/

```

```

/*    zum Empfangen von Client-Anforderungen -----*/
if ( (server_kennung =
      msgget(SERVER_KEY, S_IRWXU|S_IWGRP|S_IWOTH | IPC_CREAT)) == -1)
    fehler_meld(FATAL_SYS, "Server: kann Message Queue nicht einrichten");

while (1) {

    /*--- Empfangen von Client-Anforderungen -----*/
    if (msgrcv(server_kennung, &anforderung, MAX_LAENGE, 0, 0) == -1)
        fehler_meld(FATAL_SYS, "Server: kann nicht aus Message Queue lesen");

    sscanf(anforderung.nachricht, "%d %d %d %d",
           &client_kennung, &genauigkeit, &divident, &divisor);

    /*--- Bei message-Type 1000 ist Message Queue zu loeschen -----*/
    /*    und Server beendet sich -----*/
    if (anforderung.mtyp == 1000) {
        if (msgctl(server_kennung, IPC_RMID, NULL) == -1)
            fehler_meld(FATAL_SYS,
                       "Server: kann Message Queue nicht loeschen");
        fprintf(stderr, "---- Server: Message Queue entfernt\n");
        break;
    }

    /*--- Berechnen des Ergebnisses -----*/
    quotient = divident/divisor;
    sprintf(antwort.ergebnis,
           "%5d/%5d = %d.", divident, divisor, quotient);
    divident=divident%divisor*10;
    for (i=1 ; i<=genauigkeit ; i++) {
        sprintf(antwort.ergebnis, "%s%d",
               antwort.ergebnis, quotient=divident/divisor);
        divident = divident%divisor*10;
    }
    antwort.mtyp = 1; /*-- Message-Type hier uninteressant; muss > 0 */

    /*--- Senden des Ergebnisses an Clients -----*/
    if (msgsnd(client_kennung, &antwort, MAX_LAENGE, 0) == -1)
        fehler_meld(FATAL_SYS,
                   "Server: kann nicht in Message Queue schreiben");
    }

    fprintf(stderr, "---- Server: Ende ----\n");
    exit(0);
}

```

Programm 18.2 (mqdivser.c): Server für Division mit beliebiger Genauigkeit

Das Programm 18.2 (mqdivcli.c) ist die Client-Implementierung, die alle ihre Anforderungen an die allgemein verfügbare Message-Queue des Servers schickt, während sie die Antworten des Servers über eine eigens eingerichtete private Message-Queue empfängt. Das Programm 18.2 (mqdivcli.c) erhält die Client-Nummer über die Kommandozeile und ermittelt die zu dividierenden Zahlen und die Genauigkeit zufällig, bevor es diese

dann zusammen mit der Message-Queue-Kennung und mit der Client-Nummer (als Message-Typ) an den Server schickt. Das berechnete Ergebnis empfängt dieses Programm dann wieder aus seiner privaten Message-Queue vom Server.

```
#include <time.h>
#include <limits.h>
#include <stddef.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include "eighdr.h"
#include "mq.h"

static void delay(long mikrosek);

int
main(int argc, char *argv[])
{
    int          client_nr,
                server_kennung, client_kennung;
    mq_anforderung anforderung;
    mq_antwort     antwort;
    int            i, anzahl;

    /*--- Testen und Umwandeln des Kommandozeilenarguments ---*/
    if (argc != 2)
        fehler_meld(FATAL, "usage: %s client_nr", argv[0]);
    if ( (client_nr = atol(argv[1])) == 0)
        fehler_meld(FATAL, "Argument muss eine Clientnummer sein");

    /*--- Zufallszahlen-Generator initialisieren -----*/
    srand(time(NULL)+client_nr);

    /*--- Oeffnen der Server-Message Queue -----*/
    if ( (server_kennung = msgget(SERVER_KEY, 0)) == -1)
        fehler_meld(FATAL_SYS,
                    "Client%d: kann Server-Message Queue nicht oeffnen", client_nr);
    /*--- Einrichten der Client-Message Queue -----*/
    /* zum Empfangen von Server-Antworten -----*/
    if ( (client_kennung =
          msgget(IPC_PRIVATE, S_IRWXU|S_IWGRP|S_IWOTH | IPC_CREAT)) == -1)
        fehler_meld(FATAL_SYS,
                    "Client%d: kann Message Queue nicht einrichten", client_nr);

    anzahl = rand()%10+1;  /*-- Anzahl der Berechnungen --*/

    for (i=1; i<=anzahl; i++) {
        anforderung.mtyp = client_nr; /*-- Message-Typ */
        sprintf(anforderung.nachricht, "%d %d %d %d",
                client_kennung,
                rand()%45+1,          /* Genauigkeit */
                rand()%SHRT_MAX+1,    /* Divident */

```

```

        rand()%SHRT_MAX+1); /* Divisor      */

/*--- Senden von Anforderungen an Server -----*/
if (msgsnd(server_kennung, &anforderung, MAX_LAENGE, 0) == -1)
    fehler_meld(FATAL_SYS,
        "Client%d: kann nicht in Message Queue schreiben", client_nr);

/*--- Abbruch bei Client mit Nummer 1000 -----*/
if (client_nr == 1000)
    break;

/*--- Empfangen von Server-Antworten -----*/
if (msgrcv(client_kennung, &antwort, MAX_LAENGE, 0, 0) == -1)
    fehler_meld(FATAL_SYS,
        "Client%d: kann nicht aus Message Queue lesen", client_nr);

/*--- Ausgabe des vom Server gelieferten Ergebnisses --*/
printf("....Client%d: %s\n", client_nr, antwort.ergebnis);

delay(rand()%1000000); /* Ein bisschen warten */
}

/*--- Client beendet sich mit Loeschen der Message Queue -*/
delay(1000); /*-- Um Server noch Lesen der Ende-Kennung zu ermoeglichen */
if (msgctl(client_kennung, IPC_RMID, NULL) == -1)
    fehler_meld(FATAL_SYS,
        "Client%d: kann Message Queue nicht loeschen", client_nr);
fprintf(stderr, "--- Client%d: Ende ---\n", client_nr);

exit(0);
}

static void delay(long mikrosek)
{
    struct timeval timeout;

    timeout.tv_sec = mikrosek / 1000000;
    timeout.tv_usec = mikrosek % 1000000;
    select(0, NULL, NULL, NULL, &timeout);
}

```

Programm 18.3 (mqdivcli.c): Client für Division mit beliebiger Genauigkeit

Nachdem man das Programm 18.1 (mqdivser.c) und das Programm 18.2 (mqdivcli.c) kompiliert und gelinkt hat

```

cc -o mqdivser mqdivser.c fehler.c
cc -o mqdivcli mqdivcli.c fehler.c

```

läßt sich dieses Client-Server-Modell mit folgendem Bourne-Shellskript testen:

```

$ cat mqtest
#!/bin/sh

```

```

if [ $# -lt 1 ]
then
    echo "usage: $0 clientzahl"
    exit 1
fi

#..... Starten des Servers im Hintergrund
mqdivser &

#..... Starten der Clients im Hintergrund.....
i=1
while [ $i -le $1 ]
do
    mqdivcli $i &
    eval pid$i=$!
    i=`expr $i + 1`
done

#..... Auf Beendigung aller Clients warten .....
i=1
while [ $i -le $1 ]
do
    eval wait \${pid$i} 2>/dev/null
    i=`expr $i + 1`
done

#..... Ende-Meldung an Server .....
mqdivcli 1000
$ mqtest 3
....Client1: 7798/26181 = 0.297849585577327069248691799396
....Client2: 9056/25592 = 0.353860581431697405439199749921850578
....Client3: 10315/25003 = 0.4125
....Client2: 9621/ 4449 = 2.1625084
....Client3: 16737/ 2306 = 7.2580225498
....Client1: 2503/ 6591 = 0.3797
....Client2: 24106/31728 = 0.759770549672213817448
....Client3: 29995/14077 = 2.13078070611636001
....Client1: 18218/16613 = 1.096611087702401733582134
....Client1: 30046/ 7007 = 4.28
....Client2: 25653/ 2873 = 8.928994082840
....Client3: 21261/31506 = 0.6748238430775090458960
....Client2: 12466/ 3613 = 3.450318295045668419595903681151
....Client1: 17480/25376 = 0.68883
....Client3: 7451/14614 = 0.5098
....Client2: 29262/26254 = 1.1145730174449607678829892587796145349
....Client2: 5136/28918 = 0.177605643543813541738
....Client3: 18058/13235 = 1.36441254250094446543256516811484699659992444
....Client1: 7699/ 6508 = 1.183005531653349723417332513829
....Client3: 20884/21117 = 0.9889662357342425533930008997490
....Client1: 22156/ 3950 = 5.6091139240506329113
....Client3: 14957/19331 = 0.773731312399772386322487196730639904816098
....Client2: 19189/23753 = 0.8078558497873952763861406980170925777796488
....Client3: 16544/25271 = 0.65466344822128131059317003680
....Client2: 17674/28245 = 0.62573906886174544167109222
....Client1: 23424/28177 = 0.83131632182276324661958334812080775100

```

```

....Client1: 18803/31221 = 0.602254892540277377406232984209
....Client1: 24331/21306 = 1.14197
....Client2: 7445/27292 = 0.272790561336655430162685035907958
--- Client3: Ende ---
--- Client1: Ende ---
--- Client2: Ende ---
---- Server: Message Queue entfernt
---- Server: Ende ----
--- Client1000: Ende ---
$

```

Hinweise zu Messages Queues

Wie schon früher erwähnt, werden Message-Queues nicht automatisch vom Kern gelöscht. Das Löschen liegt in der Verantwortung des Prozesses, der die Message-Queue angelegt hat. Beendet sich ein solcher Prozeß (freiwillig oder unfreiwillig), so verbleibt die Message-Queue so lange im System, bis sie entweder explizit mit dem Kommando **ipcrm** gelöscht oder das System neu gebootet wird. Dies ist ein erheblicher Nachteil von Message-Queues gegenüber Pipes oder FIFOs (hier bleibt nur der Name, aber nicht die Daten erhalten).

Wird eine zweikanalige Kommunikation zwischen einem Client und einem Server benötigt, so kann man entweder Message-Queues oder *Stream Pipes* (siehe Kapitel 19.2) verwenden, die ähnlich zu Pipes sind, nur daß sie vollduplex arbeiten.

Zeitvergleiche zeigen, daß Message-Queues, die ursprünglich entwickelt wurden, um eine schnellere Interprozeßkommunikation zu ermöglichen, nicht schneller sind als Stream Pipes. Wegen der Nachteile von Message-Queues sollten deshalb heute bei Neuentwicklungen Stream Pipes benutzt werden (siehe auch Kapitel 19).

18.3 Semaphore

Ein *Semaphor* ist eine nichtnegative Zählvariable (vom Datentyp **unsigned short**), deren Wert beim Eintritt in einen kritischen Programmabschnitt dekrementiert und beim Verlassen wieder inkrementiert wird.

Semaphore werden zur Synchronisation benutzt, wenn mehrere Prozesse auf eine gemeinsame Ressource (wie z.B. einen gemeinsamen Datenbereich) zugreifen.

18.3.1 Synchronisation von kritischen Abschnitten mit Semaphore

Ein Prozeß, der auf einen gemeinsamen Datenbereich (*Shared Data Object*) zugreifen möchte, muß folgende Schritte durchführen:

1. Überprüfen des Semaphors, das für die Synchronisation dieses Bereichs zuständig ist.
 - a) Ist der Wert des Semaphors positiv, so kann der Prozeß auf diesen Bereich zugreifen. Um anzuzeigen, daß auf diesen gemeinsamen Bereich gerade zugegriffen wird, dekrementiert der Prozeß den Wert des Semaphors um 1, bevor er zugreift.

- b) Ist der Wert des Semaphors gleich 0, so wartet der Prozeß solange, bis der Bereich für einen Zugriff frei wird, also der Wert des Semaphors positiv wird und somit die Sperre für diesen Bereich aufgehoben ist.
- 2. Wenn ein Prozeß mit dem Zugriff auf den gemeinsamen Bereich fertig ist, inkrementiert er das Semaphore wieder um 1, um nun eventuell anderen wartenden Prozessen den Zugriff auf diesen Bereich zu gestatten.

Da die Überprüfung des Semaphorewerts und das Dekrementieren dieses Werts *eine atomare Operation* sein muß, sind Semaphore normalerweise im Kern implementiert.

Wenn, was häufig der Fall ist, nur eine kritische Ressource über Semaphore zu synchronisieren ist, könnten sogenannte *binäre Semaphore* verwendet werden. Binäre Semaphore können nur die beiden Werte 0 und 1 annehmen. Das heißt, daß immer nur ein Prozeß den kritischen Bereich zu einem Zeitpunkt benutzen darf. Allgemeine Semaphore dagegen können Werte von 0, 1, 2,...,n annehmen, wobei der aktuelle Wert immer anzeigt, wieviele Prozesse den kritischen Bereich noch benutzen dürfen.

18.3.2 Eigenschaften vom System V-Semaphore

Nachfolgend sind die besonderen Eigenschaften und teilweise auch Schwächen der System V Semaphore kurz zusammengefaßt:

1. Ein Semaphore ist nicht nur ein nichtnegativer Wert, sondern eine Menge von einem oder mehreren Semaphorewerten. Beim Kreieren des Semaphors muß die Anzahl der Werte dieser Menge festgelegt werden. Semaphoremenge erlauben eine detailliertere Aufteilung von kritischen Bereichen oder Ressourcen. Es muß z.B. beim Zugriff auf ein Speichersegment nicht das ganze Speichersegment gesperrt werden, sondern eventuell nur einzelne voneinander unabhängige Speicherbereiche.
2. Das Kreieren eines Semaphors (*semget*) ist unabhängig von seiner Initialisierung (*semctl*). Dies ist eine große Schwäche, da das Kreieren und Initialisieren eines Semaphors somit niemals eine atomare Operation sein kann.
3. Da Semaphore (wie auch Message-Queues) immer weiter existieren, selbst wenn sie kein Prozeß benutzt, muß der Anwender darauf achten, daß der Prozeß, der ein Semaphore eingerichtet hat, dieses vor seiner Beendigung wieder freigibt. Weiter unten bei der Vorstellung des *undo*-Zählers wird darauf nochmals eingegangen.

18.3.3 semid_ds – Status eines Semaphors

Zu jedem Semaphore existiert eine *semid_ds*-Struktur:

```
struct semid_ds {
    struct ipc_perm sem_perm; /* in Kapitel 18.1 beschrieben */
    struct sem *sem_base; /* Adr. des 1. Semaphors in Menge */
    ushort sem_nsems; /* Anzahl der Semaphore in Menge */
    time_t sem_otime; /* Zeitpkt. des letzten semop-Aufrufs */
    time_t sem_ctime; /* Zeitpkt. der letzten Änderung */
}
```

Für einen Benutzerprozeß ist die Komponente `sem_base` nicht von Interesse, da sie einen Speicherbereich im Kern adressiert. Dieser Speicherbereich enthält ein Array von `sem_nsems` Elementen, für jeden Semaphorwert eines. Der Datentyp dieser Elemente ist die Struktur `sem`:

```
struct sem {
    ushort semval; /* Semaphorwert; immer >= 0 */
    pid_t sempid; /* PID des letzten zugreifenden Prozesses */
    ushort semncnt; /* Anz. d. Prozesse, die warten, bis semval>0 */
    ushort semzcnt; /* Anz. d. Prozesse, die warten, bis semval==0 */
}
```



Unter Linux enthält die Struktur `semid_ds` noch drei weitere Komponenten:

```
struct sem_queue *sem_pending; /* Operationen, die noch auszuführen sind */
struct sem_queue **sem_pending_last; /* letzte auszuführende Operation */
struct sem_undo *undo; /* Adresse einer Struktur, die die
                        rückgängig zu machenden Operationen enthält */
```

In der Struktur `sem_queue` befindet sich unter anderem eine Warteschlange von Prozessen, die gerade blockiert sind und auf die Freigabe des Semaphors warten.

Die Struktur `sem_undo` enthält Informationen über Semaphor-Operationen eines Prozesses, die durchzuführen sind, wenn dieser das Semaphor wieder freigibt. Für jeden Aufruf einer Semaphor-Operation kann nämlich ein Prozeß diese Undo-Operationen einrichten. Die entsprechenden `sem_undo`-Strukturen werden dann dynamisch allokiert.

```
/* Each task has a list of undo requests. They are executed automatically
 * when the process exits.
 */
```

```
struct sem_undo {
    struct sem_undo *proc_next; /* Lineare Liste aller undo-
                                Strukturen eines Prozesses */
    struct sem_undo *id_next; /* Lineare Liste aller undo-
                                Strukturen fuer eine Semaphormenge */
    int semid; /* Kennung des Semaphors */
    short *semadj; /* Werte, auf die die Semaphore
                   zurueckgesetzt werden */
};
```

In einer `sem_undo`-Struktur werden alle rückgängig zu machenden Operationen eines Prozesses für ein Semaphor gespeichert. Der Systemkern legt für ein Semaphor maximal eine `sem_undo`-Struktur je Prozeß an. Beendet sich der Prozeß, dann werden die Semaphore entsprechend den `semadj`-Werten zurückgesetzt.



18.3.4 Limits von Semaphormengen

Die wichtigsten Limitkonstanten für Semaphormengen sind:

SEMVMX

maximaler Wert eines Semaphors (typischer Wert: 32767)

SEMMNI

maximale Anzahl von Semaphormengen (typischer Wert: 10)

SEMMNS

maximale Anzahl von Semaphoren (typischer Wert: 60)

SEMMSL

maximale Anzahl von Semaphoren pro Semaphormenge (typischer Wert: 25)

SEMOPN

maximale Anzahl von Operationen pro **semop**-Aufruf (typischer Wert: 10)

18.3.5 semget – Öffnen oder Kreieren einer Semaphormenge

Um eine existierende Semaphormenge zu öffnen oder eine neue Semaphormenge zu kreieren, steht die Funktion **semget** zur Verfügung.

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
```

```
#include <sys/sem.h>
```

```
int semget(key_t schlüssel, int nsems, int flag);
```

gibt zurück: Semaphore-ID (Erfolg); -1 bei Fehler

In Kapitel 18.1 wurde ausführlich beschrieben, wann ein neues Objekt (hier *Semaphorobjekt*) eingerichtet wird bzw. wann ein bereits existierendes geöffnet wird.

Wenn eine neue Semaphormenge eingerichtet wird, so werden die folgenden Komponenten der `semid_ds`-Struktur initialisiert:

`sem_perm`

siehe Zugriffsrechte in Kapitel. 18.1. Die Komponente `mode` der Struktur `ipc_perm` wird mit den entsprechenden im `flag`-Argument angegebenen Zugriffsrechten gesetzt. Die Rechte können dabei mit den in Tabelle 18.1 angegebenen Konstanten spezifiziert werden.

`sem_otime = 0`

`sem_ctime = momentane Zeit`

`sem_nsems = nsems (Argument)`

`nsems` ist die Anzahl der Semaphore in der Menge. Wenn eine neue Semaphormenge erzeugt wird (normalerweise im Server), muß das `nsems`-Argument angegeben sein. Wird dagegen mit ***semget*** eine bereits existierende Semaphormenge geöffnet (normalerweise in einem Client), so kann für das `nsems`-Argument der Wert 0 angegeben werden.

18.3.6 **semctl** – Abfragen/Ändern des Status oder Löschen einer Semaphormenge

Um den Status einer Semaphormenge zu erfragen oder zu ändern oder aber eine ganze Semaphormenge zu löschen, steht die Funktion ***semctl*** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int kennung, int semnum, int kdo, union semun arg);

gibt zurück: entsprechender Wert (für alle GETxxx-kdos, außer GETALL); -1 bei Fehler; 0 sonst
```

Das letzte Argument `arg` hat als Datentyp die folgende union:

```
union semun {
    int    val;           /* für kdo=SETVAL                */
    struct semid_ds *buf; /* für kdo=IPC_STAT und kdo=IPC_SET */
    ushort *array;       /* für kdo=GETALL und kdo=SETALL   */
}

struct seminfo *__buf; /* unter Linux; Puffer fuer IPC_INFO */
void *__pad;          /* unter Linux                */
```

Das Argument `kennung` legt die Semaphormenge fest, auf die ***semctl*** anzuwenden ist.

Das Argument `semnum` spezifiziert einen bestimmten Semaphorwert aus der Menge. Der Wert von `semnum` muß zwischen 0 und `nsems-1` liegen. `semnum` ist bei den fünf `kdo`-Angaben von Wichtigkeit, die sich auf einen bestimmten Semaphorwert beziehen.

Die folgenden Angaben sind für das `kdo`-Argument möglich:

IPC_STAT

Abfragen der Struktur `semid_ds` des Semaphorobjekts. ***mcgctl*** legt diese Struktur in `arg.buf` ab.

IPC_SET

Setzen der Eigentümer UID/GID und der Zugriffsrechte in der Struktur `semid_ds`. Die zu setzenden Werte befinden sich dabei in den Komponenten `sem_perm.uid`, `sem_perm.gid` und `sem_perm.mode` von `arg.buf`.

`IPC_SET` kann nur von einem Prozeß verwendet werden, dessen effektive User-ID gleich `msg_perm.cuid` oder gleich `msg_perm.uid` oder aber von einem Superuser-Prozeß ist.

IPC_RMID

Löschen der Semaphormenge. Dieses Löschen erfolgt sofort. Andere Prozesse, die diese Semaphormenge noch benutzen, erhalten bei ihrem nächsten Zugriff auf diese Semaphormenge einen Fehler, wobei `errno` auf `EIDRM` gesetzt wird. `IPC_RMID` kann nur von einem Prozeß verwendet werden, dessen effektive User-ID gleich `sem_perm.cuid` oder gleich `sem_perm.uid` ist, oder aber von einem Superuser-Prozeß.

GETVAL

Abfragen des Werts einer Semaphorvariablen. Der Rückgabewert ist `semval` der Semaphorvariablen `semnum`.

SETVAL

Setzen des Werts einer Semaphorvariablen. Der Wert der Variablen `semnum` (`semval`) wird auf `arg.val` gesetzt.

GETPID

Abfragen der PID des Prozesses, der zuletzt auf die Semaphorvariable `semnum` zugegriffen hat. Der Rückgabewert ist `sempid` der Semaphorvariablen `semnum`.

GETNCNT

Abfragen der Prozesse, die warten, bis der Wert einer Semaphorvariablen größer als 0 wird. Der Rückgabewert ist `semncnt` der Semaphorvariablen `semnum`.

GETZCNT

Abfragen der Prozesse, die warten, bis der Wert einer Semaphorvariablen gleich 0 wird. Der Rückgabewert ist `semzcnt` der Semaphorvariablen `semnum`.

GETALL

Abfragen der Werte aller Semaphorvariablen. Diese Werte legt ***msgctl*** im Array `arg.array` ab.

SETALL

Setzen der Werte aller Semaphorvariablen. Die zu setzenden Werte sind dabei im `arg.array` angegeben.

Für die `kdo`-Angaben `GETVAL`, `GETPID`, `GETNCNT`, `GETZCNT` und `IPC_STAT` ist Leserecht erforderlich. Die `kdo`-Angaben `SETVAL` und `SETALL` sind nur dem Eigentümer oder Einrichter des Semaphorobjekts oder aber dem Superuser vorbehalten.

Unter Linux kann für `kdo` noch `IPC_INFO` angegeben werden, um Informationen über das entsprechende Semaphor zu erfragen. Diese Informationen werden in die einzelnen Komponenten der Struktur `seminfo` (Komponente `__buf` in Union `semun`) eingetragen:

```
struct seminfo {
    int semmap; /* maximale Einträge in einer Semaphormenge;
                  von Linux ignoriert */
    int semmni; /* maximale Anzahl von Semaphorkennungen */
}
```

```

int semmns; /* maximale Anzahl von Semaphoren im System */
int semmnu; /* maximale Anzahl von sem_undo-Strukturen
             im System */
int semmsl; /* maximale Anzahl von Semaphoren je Kennung */
int semopm; /* maximale Anzahl von Operationen bei einem
             semop-Aufruf */
int semume; /* maximale Anzahl der sem_undo-Einträge
             fuer einen Prozeß; von Linux ignoriert */
int semusz; /* Groesse der sem_undo-Struktur;
             von Linux ignoriert */
int semvmx; /* maximaler Wert eines Semaphors */
int semaem; /* maximaler Wert fuer eine sem_undo-Struktur;
             von Linux ignoriert */
};

```

Zum Setzen dieser Komponenten sind in `<sys/sem.h>` bzw. `<linux/sem.h>` eigene Konstanten definiert, wie z.B.:

```

#define SEMMNI 128 /* ? max # of semaphore identifiers */
#define SEMMNS (SEMMNI*SEMSL) /* ? max # of semaphores in system */
#define SEMMNU SEMMNS /* num of undo structures system wide */
#define SEMMSL 32 /* <= 512 max num of semaphores per id */
#define SEMOPM 32 /* ~ 100 max num of ops per semop call */
#define SEMVMX 32767 /* semaphore maximum value */
/* unused */

#define SEMMAP SEMMNS /* # of entries in semaphore map */
#define SEMUME SEMOPM /* max num of undo entries per process */
#define SEMUSZ 20 /* sizeof struct sem_undo */
#define SEMAEM (SEMVMX >> 1) /* adjust on exit max value */

```

18.3.7 semop – Durchführen von Operationen auf Semaphormengen

Um Operationen auf Semaphormengen durchzuführen, steht die Funktion **semop** zur Verfügung.

Die Funktion **semop** führt eine ganze Reihe von Operationen, die in einem Array übergeben werden, auf eine Semaphormenge aus. Diese Operationen sind dabei eine atomare Operation, was bedeutet, daß entweder alle Operationen erfolgreich ausgeführt werden oder aber keine der Operationen.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

```

```
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Das Argument `semid` legt die Semaphormenge fest, auf die **semop** anzuwenden ist.

Das Argument `semoparray` ist die Adresse eines Arrays von Semaphoreoperationen. Die Elemente dieses Arrays haben den Datentyp `struct sembuf`:

```
struct sembuf {
    ushort sem_num; /* Nr. d. Semaphorevar. in Menge (0,1,...,nsems-1) */
    short sem_op; /* Operation */
    short sem_flg; /* IPC_NOWAIT, SEM_UNDO */
}
```

`nops` gibt die Anzahl der Operationen (Elemente) im Array `semoparray` an.

Für jede im Array `semoparray` angegebene Semaphorevariable `sem_num` wird die zugehörige Operation `sem_op` durchgeführt.

Für `sem_op` sind die folgenden Fälle zu unterscheiden:

sem_op > 0 Dekrementieren einer Semaphorevariablen (Ressource freigeben)

Der Wert von `sem_op` wird auf den Wert der entsprechenden Semaphorevariablen (`semval`) addiert. Diese Operation wird zur Freigabe von Ressourcen benötigt. Wenn in der `sem_flg`-Komponente `SEM_UNDO` gesetzt ist, so wird der `sem_op`-Wert zusätzlich noch vom sogenannten *undo*-Zähler (siehe weiter unten) des aufrufenden Prozesses subtrahiert. Der aufrufende Prozeß muß dazu Änderungsrechte (*alter*-) für die entsprechende Semaphoremenge besitzen.

sem_op < 0 Setzen einer Semaphorevariablen (Ressource anfordern)

Falls der Wert der entsprechenden Semaphorevariablen (`sem_val`) größer oder gleich dem absoluten Wert von `sem_op` ist, dann ist die angeforderte Ressource verfügbar und der absolute Wert von `sem_op` wird von `sem_val` subtrahiert. Durch die Subtraktion ist sichergestellt, daß `semval` ≥ 0 ist. Diese Operation wird zur Anforderung von Ressourcen benötigt. Wenn in der `sem_flg`-Komponente `SEM_UNDO` gesetzt ist, so wird der absolute Wert von `sem_op` zusätzlich auf den sogenannten *undo*-Zähler (siehe weiter unten) des aufrufenden Prozesses addiert.

Falls der Wert der entsprechenden Semaphorevariablen (`semval`) kleiner als der absolute Wert von `sem_op` ist, dann ist die angeforderte Ressource momentan nicht frei. Hierbei sind nun zwei Fälle zu unterscheiden:

1. Wenn in der `sem_flg`-Komponente `IPC_NOWAIT` gesetzt ist, so beendet der ***semop***-Aufruf sich mit einem Fehler, wobei `errno` auf `EAGAIN` gesetzt wird.
2. Wenn in der `sem_flg`-Komponente `IPC_NOWAIT` nicht gesetzt ist, so wird der `semcnt`-Wert dieses Semaphors inkrementiert und der aufrufende Prozeß wird so lange suspendiert, bis einer der folgenden Fälle eintritt:
 - Der Wert der Semaphorevariablen (`semval`) wird größer oder gleich dem absoluten Wert von `sem_op`. Dieses Ereignis tritt z.B. dann ein, wenn ein anderer Prozeß die betreffende Ressource wieder freigibt. Tritt dieses Ereignis ein, so wird der `semn-`

cnt-Wert dieses Semaphors wieder dekrementiert (Suspendierung wird aufgehoben) und der absolute Wert von `sem_op` wird vom Wert der Semaphorevariablen (`semval`) subtrahiert.

Wenn in der `sem_flg`-Komponente `SEM_UNDO` gesetzt ist, so wird der absolute Wert von `sem_op` zusätzlich noch auf den sogenannten *undo*-Zähler (siehe weiter unten) des aufrufenden Prozesses addiert.

- ▶ Das Semaphore wird gelöscht. In diesem Fall beendet der *semop*-Aufruf sich mit einem Fehler, wobei `errno` auf `ERMID` gesetzt wird.
- ▶ Vom aufrufenden Prozeß wurde ein Signal abgefangen. In diesem Fall wird der `semcnt`-Wert für dieses Semaphore dekrementiert (Suspendierung wird aufgehoben) und der *semop*-Aufruf beendet sich mit einem Fehler, wobei `errno` auf `EINTR` gesetzt wird.

sem_op == 0 Warten, bis Semaphorevariable gleich 0 ist

Wenn der Wert der Semaphorevariablen gleich 0 ist, kehrt ***semop*** sofort zurück.

Ist der Wert der Semaphorevariablen ungleich 0, so ist zu unterscheiden, ob `IPC_NOWAIT` im `sem_flg` gesetzt ist oder nicht.

1. Ist `IPC_NOWAIT` gesetzt, so beendet der ***semop***-Aufruf sich mit einem Fehler, wobei `errno` auf `EAGAIN` gesetzt wird.
2. Ist `IPC_NOWAIT` nicht gesetzt, so wird der `semcnt`-Wert dieses Semaphors um 1 inkrementiert und der aufrufende Prozeß wird solange suspendiert, bis einer der folgenden Fälle eintritt:
 - ▶ Der Wert der Semaphorevariablen wird 0. In diesem Fall wird der `semcnt`-Wert für dieses Semaphore dekrementiert (Suspendierung wird aufgehoben).
 - ▶ Das Semaphore wird gelöscht. In diesem Fall beendet der ***semop***-Aufruf sich mit einem Fehler, wobei `errno` auf `ERMID` gesetzt wird.
 - ▶ Vom aufrufenden Prozeß wurde ein Signal abgefangen. In diesem Fall wird der `semcnt`-Wert für dieses Semaphore dekrementiert (Suspendierung wird aufgehoben) und der ***semop***-Aufruf beendet sich mit Fehler, wobei `errno` auf `EINTR` gesetzt wird.

Der undo-Zähler (Flag SEM_UNDO)

Um bei Prozessen, die sich freiwillig oder auch unfreiwillig vorzeitig beenden, sicherzustellen, daß die von diesen Prozessen gesetzten Semaphore wieder zurückgesetzt werden, muß `SEM_UNDO` in der `sem_flg`-Komponente gesetzt sein.

Ist `SEM_UNDO` beim Setzen einer Semaphorevariablen (`sem_op < 0`) spezifiziert, so merkt sich der Kern in einem sogenannten *undo*-Zähler, wie viele Ressourcen durch diese spezielle Semaphorevariable belegt werden (Absolutwert von `sem_op`). Wenn sich dann später der

Prozeß – freiwillig oder unfreiwillig – beendet, so kann der Kern über den Wert im *undo*-Zähler für diesen Prozeß herausfinden, wie viele Semaphore zurückgesetzt werden müssen, und diese auch entsprechend richtig wieder zurücksetzen.

Wenn mit ***semctl*** der Wert einer Semaphors (mit SETVAL- oder SETALL für *kdo*) gesetzt wird, so wird der Wert des *undo*-Zählers dieses Semaphors für alle Prozesse auf 0 gesetzt.

18.3.8 Realisierung der P- und V-Operationen von Dijkstra

Der Holländer Dijkstra hat die sogenannten P- und V-Operationen zur Synchronisation von kritischen Programmabschnitten eingeführt:

P-Operation

Die *P-Operation* (holländisch: *Paseer=Betreten*) muß beim Betreten eines kritischen Abschnitts ausgeführt werden. Sie entspricht dem Überprüfen und Setzen des Semaphors (bei Eintritt in kritischen Abschnitt), das für die Synchronisation dieses Abschnitts zuständig ist; siehe auch Punkte 1, 1a und 1b im Unterkapitel »*Synchronisation von kritischen Abschnitten mit Semaphore*« in diesem Kapitel.

V-Operation

Diese *V-Operation* (holländisch: *Verlaat=Verlassen*) muß beim Verlassen eines kritischen Abschnitts ausgeführt werden. Sie entspricht dem Zurücksetzen des Semaphors, um anderen Prozessen das Betreten der kritischen Bereichs zu erlauben.

Eine mögliche Realisierung der P- und V-Operationen zeigt das Programm 18.3 (pv.c).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "eighdr.h"
#include "pv.h"

void pv(int id, int operation)
{
    static struct sembuf semaphore;

    semaphore.sem_op = operation;
    semaphore.sem_flg = SEM_UNDO;

    if (semop(id, &semaphore, 1) == -1)
        fehler_meld(FATAL_SYS, "semop-Fehler");
}
```

Programm 18.4 (pv.c): Funktion *pv* zur Nachbildung von P- und V-Operationen

Programme, die diese Funktion *pv* benutzen möchten, müssen das Programm *pv.c* dazu linken und die folgende Headerdatei *pv.h* zum Bestandteil ihres Programmes machen. (#include "pv.h")

```
#ifndef PV
#define PV
```

```

/*---- Makros fuer die P- und V-Operationen -----*/
#define P(id) pv(id, -1)
#define V(id) pv(id, 1)

extern void pv(int id, int operation);

#endif

```

Programm 18.5 Headerdatei **pv.h**: Makros für die P- und V-Operation

Im Kapitel 18.4 befindet sich ein Beispiel, in dem diese P- und V-Operationen verwendet werden.

Hinweise zu Semaphoren

Wenn eine Ressource von mehreren Prozessen gleichzeitig genutzt werden soll, so können zur Synchronisation der Zugriffe entweder Semaphore oder Dateisperren (*Record Locking* siehe Kapitel 12) benutzt werden. Während man bei Semaphoren mit den P- und V-Operationen arbeitet, benutzt man beim Dateisperren eine leere Datei, bei der das erste Byte als Sperrbyte benutzt wird, das von den Prozessen beim Zugriff auf die Ressource schreibgesperrt und bei Beendigung des Zugriffs wieder freigegeben wird.

Dateisperren haben den Vorteil, daß sie bei vorzeitiger Beendigung eines zugreifenden Prozesses automatisch vom Kern freigegeben werden. Dieser Vorteil und der doch wesentlich einfachere Code bei Dateisperren macht deren Verwendung lukrativer als die Verwendung von Semaphore, obwohl bei letzteren die Synchronisation etwas schneller ist.

18.4 Shared Memory

Shared Memory ist die schnellste Form der Interprozeßkommunikation, da von zwei oder mehreren Prozessen ein bestimmter Speicherbereich gemeinsam benutzt wird, und somit das Kopieren zwischen Server und Clients nicht notwendig ist.

Bei Verwendung von Shared Memory muß lediglich darauf geachtet werden, daß die Zugriffe der einzelnen Prozesse auf den gemeinsamen Speicherbereich synchronisiert werden. Wenn z.B. ein Server Daten in den gemeinsamen Speicherbereich schreibt, sollte den Clients ein Zugriff auf diesen Bereich so lange verwehrt sein, bis der Server seinen Schreibvorgang beendet hat. Zur Synchronisation der Zugriffe auf den gemeinsamen Speicherbereich werden meist Semaphore verwendet, obwohl auch andere Synchronisationsmethoden denkbar wären, wie z.B. Sperren der Speicherbereiche (siehe Kapitel 12).

18.4.1 shmid_ds – Status eines Shared-Memory-Segments

Zu jedem Shared-Memory-Segment existiert eine `shmid_ds`-Struktur im Kern:

```

struct shmid_ds {
    struct ipc_perm shm_perm; /* in Kapitel 18.1 beschrieben */

```



```

struct anon_map *shm_map; /* Adresse */
int shm_segsz; /* Größe des Segments in Bytes */
ushort shm_lkcnt; /* wie oft Segment gesperrt ist */
pid_t shm_lpid; /* PID des letzten shmop-Aufrufers */
pid_t shm_cpid; /* PID des Einrichters des Shared Memory */
ulong shm_nattch; /* wieoft Segment an andere Prozesse
angebunden (attached) ist */
ulong shm_cnattch; /* nur für shminfo benötigt */
time_t shm_atime; /* letzter attach-Zeitpunkt */
time_t shm_dtime; /* letzter detach-Zeitpunkt */
time_t shm_ctime; /* letzter Änderungs-Zeitpunkt */
}

```



Unter Linux fehlen die Komponenten `sh_map` und `shm_lkcnt`. Dafür sind dort andere Komponenten enthalten:

```

unsigned short shm_npages; /* Anzahl der Pages (Speicherseiten) */
unsigned long *shm_pages; /* Array von Pagetableneinträgen */
struct vm_area_struct *attaches; /* attach-Deskriptoren */

```

Die Moduskomponente der `ipc_perm`-Struktur wird zur Speicherung zweier zusätzlicher Flags verwendet: `SHM_LOCKED` verhindert das Auslagern von Pages des Shared Memory und `SHM_DEST` legt fest, daß das Shared Memory-Segment bei der letzten detach-Operation automatisch wieder freigegeben wird.

Im Array `shm_pages` werden die Pagetableneinträge der Pages gehalten, aus denen das Shared Memory besteht. Nach dem Kreieren eines Shared Memory sind noch keine Pages reserviert. Dies erfolgt erst, wenn auf das Shared Memory zugegriffen wird. Im Array `shm_pages` können sich auch Einträge von gerade ausgelagerten Pages befinden.



18.4.2 Limits

Für Shared Memory sind die folgenden Limitkonstanten definiert:

`SHMMAX`

maximale Größe (in Bytes) eines Shared-Memory-Segments (typischer Wert: 131072)

`SHMMIN`

minimale Größe (in Bytes) eines Shared-Memory-Segments (typischer Wert: 1)

`SHMMNI`

maximale Anzahl von Shared-Memory-Segmenten (typischer Wert: 100)

`SHMSEG`

maximale Anzahl von Shared-Memory-Segmenten pro Prozeß (typischer Wert: 6)

18.4.3 shmget – Öffnen oder Kreieren eines Shared-Memory-Segments

Um ein existierendes Shared-Memory-Segment zu öffnen oder ein neues Shared-Memory-Segment zu kreieren, steht die Funktion **shmget** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget(key_t schlüssel, int groesse, int flag);
```

gibt zurück: Kennung des Shared-Memory-Segments (bei Erfolg); -1 bei Fehler

In Kapitel 18.1 wurde ausführlich beschrieben, wann ein neues Objekt (hier Shared-Memory-Segment) eingerichtet und wann ein bereits existierendes geöffnet wird. Wenn ein neues Shared-Memory-Segment eingerichtet wird, so werden die folgenden Komponenten der `shmid_ds`-Struktur initialisiert:

`shm_perm`

siehe Zugriffsrechte in Kapitel 18.1. Die Komponente `mode` der Struktur `ipc_perm` wird mit den entsprechenden im `flag`-Argument angegebenen Zugriffsrechten gesetzt. Die Rechte können dabei mit den in Tabelle 18.1 angegebenen Konstanten spezifiziert werden.

```
shm_lpid      = 0
shm_nattach   = 0
shm_atime     = 0
shm_dtime     = 0
shm_segsz     = groesse (nur beim Kreieren)
shm_ctime     = momentane Zeit
```

Das Argument `groesse` legt die minimale Größe eines Shared-Memory-Segments fest. Wenn ein neues Segment kreiert wird (typischerweise im Server), muß seine `groesse` angegeben werden. Öffnet man dagegen ein bereits existierendes Segment (typischerweise im Client), so kann für das `groesse`-Argument der Wert 0 angegeben werden.

Hinweis

Üblicherweise initialisiert die Funktion **shmget** nur die zugehörige Struktur `shmid_ds` und reserviert noch keinen Speicher für das Shared Memory.

18.4.4 shmctl – Abfragen/Ändern des Status oder Löschen eines Shared-Memory-Segments

Um den Status eines Shared-Memory-Segments zu erfragen oder zu ändern oder aber ein Shared-Memory-Segment zu löschen, steht die Funktion **shmctl** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmctl(int kennung, int kdo, struct shmids *puffer);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Das *kdo*-Argument legt die durchzuführende Aktion fest:

IPC_STAT

Abfragen des Status des Shared-Memory-Segments

shmctl schreibt diese Statusinformation an die Adresse *puffer*.

IPC_SET

Setzen der Eigentümer-UID/GID und der Zugriffsrechte

Im übergebenen *puffer* befinden sich dabei die zu setzenden Werte, wobei jedoch nur die folgenden Komponenten relevant sind: *shm_perm.uid*, *shm_perm.gid* und *shm_perm.mode*. IPC_SET kann jedoch nur von einem Prozeß verwendet werden, dessen effektive User-ID gleich *shm_perm.cuid* oder gleich *shm_perm.uid* ist, oder aber von einem Superuser-Prozeß.

IPC_RMID

Löschen des Shared-Memory-Segments

In der Struktur *shmids* existiert eine Komponente *shm_nattch*, die die Anzahl der Prozesse enthält, an die dieses Segment angebunden (*attached*) ist. Das entsprechende Shared-Memory-Segment wird so lange nicht wirklich gelöscht, wie *shm_nattch* != 0 ist. Das bedeutet, daß das Segment erst dann gelöscht wird, wenn der letzte Prozeß, der es benutzt, sich beendet oder aber die Anbindung dieses Segment aufhebt (*detached*).

Unabhängig davon, ob *shm_nattch* == 0 ist oder nicht, wird die *kennung* des Shared-Memory-Segments sofort gelöscht, so daß keinerlei neue Anbindungen für dieses Segment mit **shmat** mehr möglich sind.

IPC_RMID kann nur von einem Prozeß ausgeführt werden, dessen effektive User-ID gleich *shm_perm.cuid* oder *shm_perm.uid* ist, oder aber von einem Superuser-Prozeß.

SHM_LOCK

Sperren des Shared-Memory-Segments

SHM_LOCK kann nur vom Superuser ausgeführt werden.

SHM_UNLOCK

Aufheben einer Sperre für ein Shared-Memory-Segment.

SHM_UNLOCK kann nur vom Superuser ausgeführt werden.



Unter Linux kann für `kdo` noch `IPC_INFO` angegeben werden, um Informationen über das entsprechende Shared Memory zu erfragen. Diese Informationen werden in die einzelnen Komponenten der Struktur `shminfo` (an Adresse `puffer`) eingetragen:

```
struct shminfo {
    int shmmx; /* Maximale Anzahl eines Segments in Bytes */
    int shmmin; /* Maximale Groesse eines Segments */
    int shmmni; /* Maximale Anzahl von Shared Memories im System */
    int shmseg; /* Maximale Anzahl von Segmenten, die je Prozess
                fuer Shared Memory zur Verfuegung stehen */
    int shmll; /* Maximale Anzahl von Pages, die im System
                fuer Shared Memory zur Verfuegung stehen */
};
```

Zum Setzen dieser Komponenten sind in `<asm/shmparam.h>` z.B. die folgenden Konstanten definiert:

```
/* _SHM_ID_BITS + _SHM_IDX_BITS must be <= 24 on the i386 and
 * SHMMAX <= (PAGE_SIZE << _SHM_IDX_BITS).
 */
#define SHMMAX 0x2000000 /* max shared seg size (bytes) */
#define SHMMIN 1 /* really PAGE_SIZE */ /* min shared seg size (bytes) */

#define SHMMNI (1<<_SHM_ID_BITS) /* max num of segs system wide */

#define SHMALL /* max shm system wide (pages) */ \
    (1<<(_SHM_IDX_BITS+_SHM_ID_BITS))
#define SHMLBA PAGE_SIZE /* attach addr a multiple of this */
#define SHMSEG SHMMNI /* max shared segs per process */
```



18.4.5 shmat – Anbinden eines Shared-Memory-Segments an einen Prozeß

Nachdem ein Shared-Memory-Segment kreiert wurde, können es Prozesse, die dessen kennung kennen, an ihren Adreßraum anbinden (*attach*). Zum Anbinden eines Shared-Memory-Segments steht die Funktion **shmat** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
void *shmat(int kennung, void *adr, int flag);
```

gibt zurück: Adresse des Shared-Memory-Segments (bei Erfolg); -1 bei Fehler

Die Adresse, an die das Shared-Memory-Segment *kennung* im aufrufenden Prozeß angebunden wird, hängt vom *adr*-Argument und der Angabe des SHM_RND-Flag ab:

adr == NULL (empfehlenswert)

Das Shared-Memory-Segment wird an die erste verfügbare Adresse (vom Kern festgelegt) angebunden.

adr != NULL und SHM_RND nicht in *flag* gesetzt (nicht empfehlenswert)

Das Shared-Memory-Segment wird an die angegebene Adresse *adr* angebunden. Aus Portabilitätsgründen ist diese Aufrufform nicht empfehlenswert.

adr != NULL und SHM_RND in *flag* gesetzt (nicht empfehlenswert)

Das Shared-Memory-Segment wird an die Adresse *adr* - (*adr* % SHMLBA) angebunden. Diese berechnete Adresse ist die nächstniedrigere Adresse (zu *adr*), die durch SHMLBA teilbar ist. Die Konstante SHMLBA steht für »*low boundary address multiple*« und hat als Wert immer eine Zweierpotenz (2^x). Aus Portabilitätsgründen ist diese Aufrufform nicht empfehlenswert.

Wenn im *flag*-Argument SHM_RDONLY gesetzt ist, wird das Shared-Memory-Segment zum »Nur-Lesen« angebunden, ansonsten ist sowohl Lesen als auch Schreiben in diesem Speicherbereich möglich.

Bei erfolgreichem **shmat**-Aufruf wird in der Struktur *shmid_ds* der Wert der Komponente *shm_nattch* um 1 inkrementiert und die Zeit des letzten Anbindevorgangs *shm_atime* auf die momentane Zeit gesetzt.



Unter Linux ist **shmat** wie folgt deklariert:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmat(int kennung, char *adr, int flag, ulong *radr);
```

Der Parameter *adr* kann auch hier benutzt werden, um die Adresse festzulegen, an der das Shared-Memory-Segment einzublenden ist. Wird für *adr* der NULL-Zeiger angegeben, sucht sich die Funktion selber einen freien Speicherbereich, dessen Adresse sie über den Parameter *radr* zurückgibt. Über den Parameter *flag* können die folgenden Flags gesetzt werden:

SHM_RND

Adresse wird auf eine Page-Grenze abgerundet. Unter Linux ist das Einblenden eines Segments grundsätzlich nur an einer Page-Grenze möglich.

SHM_RDONLY

legt fest, daß das Segment nur lesbar sein soll.



18.4.6 shmdt – Loslösen eines angebunden Shared-Memory-Segments

Benötigt ein Prozeß ein mit **shmat** angebundenes Shared-Memory-Segment nicht mehr, so kann er diese Anbindung wieder aufheben. Zum Loslösen (*detach*) eines angebundenen Shared-Memory-Segments steht die Funktion **shmdt** zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(void *adr);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Für das *adr*-Argument, das die Adresse des loszulösenden Shared-Memory-Segments festlegt, sollte eine von einem vorherigen **shmat**-Aufruf erhaltene Adresse angegeben werden. Es ist zu beachten, daß ein Loslösen eines Shared-Memory-Segments durch einen Prozeß nicht das Löschen dieses Segments nach sich zieht. Ein Shared-Memory-Segment existiert immer so lange, bis ein Prozeß (meist der Server) dieses explizit mit einem **shmctl**-Aufruf (*kdo* == *IPC_RMID*) löscht.

Bei erfolgreichem **shmdt**-Aufruf wird in der Struktur *shm_id_ds* der Wert der Komponente *shm_nattch* um 1 dekrementiert und die Zeit des letzten Loslösevorgangs *shm_dtime* auf die momentane Zeit gesetzt.

18.4.7 Shared Memory zwischen verwandten Prozessen

SVR4

Wenn Prozesse verwandt sind, also einen gemeinsamen Vorfahren besitzen, so bietet SVR4 zur Kommunikation zwischen diesen Prozessen eine eigene Technik an. Diese Technik verwendet die spezielle Datei */dev/zero*, die aufgrund ihrer besonderen Eigenschaften zur Kommunikation mit Shared Memory gut geeignet ist.

Die Datei */dev/zero* liefert bei jedem Lesezugriff die geforderte Anzahl von Bytes, die alle mit dem Wert 0 besetzt sind. Andererseits können in die Datei */dev/zero* beliebig viele Daten geschrieben werden, denn alle dorthin geschriebenen Daten werden sofort weggeworfen.

Für Interprozeßkommunikation ist die Datei wegen ihrer Besonderheiten im Zusammenhang mit Memory Mapped I/O (siehe Kapitel 15.3) sehr nützlich. Wird nämlich für `/dev/zero` mit der Funktion ***mmap*** ein Memory Mapped I/O eingerichtet, so gelten die folgenden Besonderheiten:

- ▶ Es wird ein namenloser Speicherbereich eingerichtet, dessen Größe mit dem zweiten Argument beim ***mmap***-Aufruf festgelegt wird. Dabei ist zu beachten, daß immer nur ganze Speicherseiten allokiert werden.
- ▶ Dieser Mapped-Speicherbereich wird mit 0 initialisiert.
- ▶ Wenn das Flag `MAP_SHARED` beim ***mmap***-Aufruf angegeben ist, so können alle verwandten Prozesse auf den Mapped-Speicherbereich zugreifen.

Programm 18.4 (`ek_mmap.c`) verdeutlicht diese Technik, indem es die Datei `/dev/zero` öffnet und dann mit ***mmap*** für diese geöffnete Datei Memory Mapped I/O einrichtet. Nach dem ***mmap***-Aufruf kann `/dev/zero` wieder geschlossen werden.

Nach dem ***fork***-Aufruf können sowohl Eltern- als auch Kindprozeß auf den Mapped-Speicherbereich zugreifen, da `MAP_SHARED` beim ***mmap***-Aufruf angegeben wurde.

In diesem Program 18.4 (`ek_mmap.c`) schreibt der Elternprozeß zwei Zufallszahlen in den gemeinsamen Speicherbereich, aus dem sie der Kindprozeß den liest, addiert und das Additionsergebnis nun seinerseits in den gemeinsamen Speicherbereich schreibt. Nun liest der Elternprozeß das vom Kindprozeß geschriebene Ergebnis und gibt es am Terminal aus. Dieser Vorgang wird hundertmal wiederholt. Die Synchronisation zwischen den einzelnen Zugriffen von Eltern- und Kindprozeß wird dabei mit den Funktionen ***INIT_SYNCH***, ***HALLO_PAPA***, ***WARTE_AUF_PAPA***, ***HALLO_KIND*** und ***WARTE_AUF_KIND*** aus Programm 17.5 (`pipesync.c`) durchgeführt.

```
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <time.h>
#include "eighdr.h"

#define ADDITIONEN 100

typedef struct {
    int zahl1;
    int zahl2;
} mmap_typ;

int
main(void)
{
    int      fd, i;
    pid_t    pid;
    caddr_t  adr;
    mmap_typ *mmap_adr;
```

```

if ( (fd = open("/dev/zero", O_RDWR)) < 0)
    fehler_meld(FATAL_SYS, "kann /dev/zero nicht oeffnen");

if ( (adr = mmap(0, sizeof(mmap_typ), PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0)) == (caddr_t)-1)
    fehler_meld(FATAL_SYS, "mmap-Fehler");
mmap_adr = (mmap_typ *)adr;

close(fd); /* Nach dem Mapping kann /dev/zero geschlossen werden */

INIT_SYNC();

if ( (pid = fork()) < 0)
    fehler_meld(FATAL_SYS, "fork-Fehler");

else if (pid > 0) { /*----- Elternprozess -----*/
    srand(time(NULL));
    for (i=1; i<=ADDITIONEN; i++) {
        mmap_adr->zahl1 = rand()%10000;
        mmap_adr->zahl2 = rand()%10000;
        printf("%5d: %5d + %5d = ", i, mmap_adr->zahl1, mmap_adr->zahl2);
        HALLO_KIND(pid);
        WARTEN_AUF_KIND();
        printf("%5d\n", mmap_adr->zahl1);
    }
    mmap_adr->zahl1 = -1; /* Abbruch-Kriterium */
} else { /*----- Kindprozess -----*/
    while (1) {
        WARTEN_AUF_PAPA();
        if (mmap_adr->zahl1 < 0)
            break;
        mmap_adr->zahl1 = mmap_adr->zahl1 + mmap_adr->zahl2;
        HALLO_PAPA(getppid());
    }
}
exit(0);
}

```

Programm 18.6 (ek_mmap.c): IPC zwischen Eltern- und Kindprozeß mit Memory Mapped I/O auf /dev/zero

Nachdem man dieses Programm 18.4 (ek_mmap.c) kompiliert und gelinkt hat

```
cc -o ek_mmap ek_mmap.c pipesync.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ ek_mmap
1: 8327 + 9173 = 17500
2: 1753 + 1353 = 3106
3: 1341 + 4050 = 5391
4: 5970 + 621 = 6591
5: 4975 + 6465 = 11440
6: 6041 + 4324 = 10365

```



```

7: 2484 + 239 = 2723
8: 4835 + 711 = 5546
9: 3833 + 6339 = 10172
10: 4683 + 7340 = 12023
.....
.....
91: 8112 + 3911 = 12023
92: 4293 + 7078 = 11371
93: 8568 + 5566 = 14134
94: 7684 + 5986 = 13670
95: 2915 + 507 = 3422
96: 8611 + 9784 = 18395
97: 4945 + 2681 = 7626
98: 3948 + 3432 = 7380
99: 720 + 7803 = 8523
100: 2650 + 5192 = 7842
$

```

Diese Technik hat den Vorteil, daß keine neue Datei für den **mmap**-Aufruf angelegt werden muß und **mmap** einen Mapped-Speicherbereich der angegebenen Größe automatisch erzeugt.

BSD-Unix

BSD-Unix bietet eine ähnliche Technik an, die dort mit *Anonymous Memory Mapping* bezeichnet wird. Dazu muß beim **mmap**-Aufruf das Flag `MAP_ANON` gesetzt und als Filedeskriptor -1 angegeben werden. Um diese Technik für das Programm 18.4 (`ek_mmap.c`) anzuwenden, müssen dort die folgenden Änderungen vorgenommen werden:

- Das Öffnen (**open**) und Schließen (**close**) der Datei `/dev/zero` entfernen.
- Den Aufruf von **mmap** wie folgt ändern:

```

if ( (adr = mmap(0, sizeof(mmap_typ),
                 PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == (caddr_t)-1)
    fehler_meld (FATAL_SYS, "mmap-Fehler");

```

Da -1 für das Filedeskriptorargument angegeben wird, ist der allokierte Speicherbereich mit keiner Datei verknüpft. Man bezeichnet einen solchen Speicherwert als *anonym*.

Der Nachteil der hier vorgestellten Techniken ist, daß sie nur zwischen verwandten Prozessen benutzt werden kann. Im nachfolgenden Beispiel wird eine Kommunikation zwischen nicht verwandten Prozessen gezeigt.

18.4.8 Client-Server-Implementierung mit Shared Memory und Semaphoren

Nachfolgend wird eine Client-Server-Implementierung auf Basis von Shared Memory und unter Zuhilfenahme von Semaphoren gezeigt. Dabei ist das Programm 18.5 (`smdiv-ser.c`) der Serverprozeß, der eine Division mit beliebiger Genauigkeit für ganze Zahlen

durchführt. Die Zahlen und die geforderte Genauigkeit erhält er dabei von dem Programm 18.6 (`smdivcli.c`), das die Client-Implementierung darstellt. Bei jedem Start von `smdivcli.c` wird ein neuer Clientprozeß zum Serverprozeß (`smdivser.c`) eingerichtet.

Der Serverprozeß richtet zwei Shared Memories ein: Ein Shared Memory, in das die Clients ihre Anforderungen schreiben und aus dem der Server diese liest. Das andere Shared Memory benutzt der Server zum Schreiben seiner Antworten an die Clients, die sie daraus lesen. Abbildung 18.2 verdeutlicht dies.

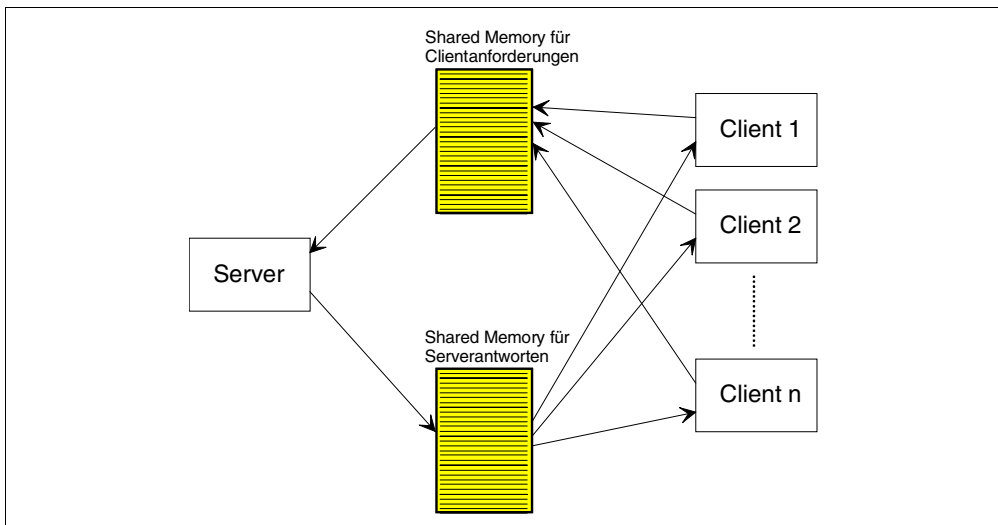


Abbildung 18.2: Client-Server-Modell mit Shared Memory und Semaphoren

Jede Client-Anforderung, die ins Shared Memory geschrieben wird, setzt sich aus den folgenden Daten zusammen:

► *Prozeß-ID des Clients*

Diese PID benötigt der Server zum Schicken des Signals `SIGUSR1` an den entsprechenden Client, um ihm mitzuteilen, daß eine Antwort für ihn im Shared Memory bereitliegt.

► *Clientnummer*

Schickt ein Client mit der Nummer 1000 eine Anforderung an den Server, so bedeutet dies, daß der Server sich beenden soll. Dies ist die einzige Verwendung der Client-Nummer.

► *Genauigkeit, Divident und Divisor*

Dies sind die eigentlichen Daten der jeweiligen Client-Anforderung.

► *Flag (gelesen oder ungelesen)*

Dieses Flag verhindert, daß noch nicht gelesene Anforderungen durch neue überschrieben werden. Es ist notwendig, da das Shared Memory in Form eines Ringpuffers implementiert ist.

Am Anfang des Shared Memory (für Client-Anforderungen) befindet sich ein Eintrag, der immer die Nummer des zuletzt geschriebenen Satzes (Anforderung) angibt.

Jede Server-Antwort, die in das andere Shared Memory (für Server-Antworten) geschrieben wird, setzt sich aus folgenden Daten zusammen:

► *Prozeß-ID des Clients*

Diese PID ist für die Clients notwendig, damit sie beim Lesen im Shared Memory die für sie bestimmte Antwort identifizieren können.

► *Server-Antwort*

Die Antwort des Servers ist immer der String mit der maximalen Länge MAX_ANTWORT.

► *Flag (gelesen oder ungelesen)*

Dieses Flag verhindert, daß noch nicht gelesene Antworten durch neue überschrieben werden. Es ist notwendig, da auch das Shared Memory für Server-Antworten in Form eines Ringpuffers implementiert ist.

Am Anfang des Shared Memory (für Server-Antworten) befindet sich ein Eintrag, der immer die Nummer des zuletzt geschriebenen Satzes (Antwort) angibt.

Die vom Server und den Clients gemeinsam benutzten Konstanten und Strukturen sind in der Headerdatei `sm.h` definiert.

```
#ifndef SM
#define SM

/*---- Vereinbarer Schluessel zwischen Server und Clients -----*/
#define SHM_READKEY 10001
#define SHM_WRITEKEY 10002
#define SEM_READKEY 20001
#define SEM_WRITEKEY 20002

/*--- Maximale Laenge einer Antwort und des gesamten Shared Memory ---*/
#define MAX_ANTWORT 100
#define SHM_MAXSAETZE 1000

/*---- Datentypen fuer Client-Anforderungen und Serverantwort -----*/
typedef struct {
    pid_t pid;
    int client_nr;
    int genauigkeit;
    int dividend;
    int divisor;
    char ungelesen;
} anforder_satz;

typedef struct {
    long satznr;
    anforder_satz anforderung[SHM_MAXSAETZE];
} anforder_shm;
```

```

typedef struct {
    pid_t  pid;
    char   ergebnis[MAX_ANTWORT];
    char   ungelesen;
} antwort_satz;

typedef struct {
    long    satznr;
    antwort_satz antwort[SHM_MAXSAETZE];
} antwort_shm;

#endif

```

*Programm 18.7 Headerdatei **sm.h**: Gemeinsame Konstanten und Strukturen im Server und den Clients*

Das Programm 18.5 (`smdivser.c`) ist der Server, der zwei Shared Memories und zwei Semaphore zur Synchronisation der Zugriffe auf die beiden shared memories einrichtet. Als Schlüssel für diese werden die in der Headerdatei `sm.h` definierten Konstanten benutzt:

`SHM_READKEY`

Shared Memory für Client-Anforderungen.

`SHM_WRITEKEY`

Shared Memory für Server-Antworten.

`SEM_READKEY`

Semaphor zur Synchronisation der Zugriffe auf das Shared Memory für die Client-Anforderungen.

`SEM_WRITEKEY`

Semaphor zur Synchronisation der Zugriffe auf das Shared Memory für die Server-Antworten.

Nach dem Einrichten liest der Server nacheinander die jeweiligen Client-Anforderungen aus dem entsprechenden Shared Memory, berechnet das entsprechende Ergebnis und schreibt dieses in das Shared Memory für Server-Antworten. Danach schickt der Server mit **kill** dem entsprechenden Clientprozeß das Signal `SIGUSR1`, um diesen zu informieren, daß sein angefordertes Ergebnis nun im Shared Memory steht. Der Server beendet sich immer erst dann, wenn ein Client mit Client-Nummer 1000 eine Anforderung schickt. Vor seiner Beendigung löscht der Server jedoch noch alle von ihm eingerichteten Shared Memories und Semaphore.

```

#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include "eighdr.h"
#include "sm.h"

```

```

#include    "pv.h"

static void beende_server(int exit_wert);

static int    shm_anfordid, shm_antwortid,
              sem_anfordid, sem_antwortid;

int
main(void)
{
    int            lese_satznr, schreib_satznr;
    anforder_shm   *shm_anford;
    antwort_shm    *shm_antwort;
    anforder_satz  anforderung;
    int            client_nr, genauigkeit, dividend, divisor, quotient, i;
    pid_t          pid;
    char           ergebnis[MAX_ANTWORT];

    /*--- Einrichten und Anbinden (attach) eines Shared Memory zum -----*/
    /* Lesen der Client-Anforderungen -----*/
    if ( (shm_anfordid =
          shmget(SHM_READKEY, SHM_MAXSAETZE*sizeof(anforder_satz)+1,
                S_IRWXU|S_IRWXG|S_IRWXO | IPC_CREAT | IPC_EXCL)) == -1)
        fehler_meld(FATAL_SYS, "shmget-Fehler (Lese-Shared Memory)");
    if ( (shm_anford =
          (anforder_shm *)shmat(shm_anfordid, NULL, 0)) == (void *)-1)
        fehler_meld(FATAL_SYS, "Server: shmat-Fehler (anforder_shm)");

    /*--- Einrichten und Anbinden (attach) eines Shared Memory zum -----*/
    /* Schreiben der Antworten an die Clients -----*/
    if ( (shm_antwortid =
          shmget(SHM_WRITEKEY, SHM_MAXSAETZE*MAX_ANTWORT+1,
                S_IRWXU|S_IRWXG|S_IRWXO | IPC_CREAT | IPC_EXCL)) == -1)
        fehler_meld(FATAL_SYS, "shmget-Fehler (Schreib-Shared Memory)");
    if ( (shm_antwort =
          (antwort_shm *)shmat(shm_antwortid, NULL, 0)) == (void *)-1)
        fehler_meld(FATAL_SYS, "Server: shmat-Fehler (antwort_shm)");

    /*--- Einrichten und Setzen eines Semaphors fuer Lese-Shared Memory ---*/
    if ( (sem_anfordid = semget(SEM_READKEY, 1,
          S_IRWXU|S_IRWXG|S_IRWXO | IPC_CREAT | IPC_EXCL)) == -1)
        fehler_meld(FATAL_SYS, "semget-Fehler (Lese-Shared Memory)");
    if (semctl(sem_anfordid, 0, SETVAL, (int)1) == -1)
        fehler_meld(FATAL_SYS, "semctl-Fehler (Lese-Shared Memory)");

    /*--- Einrichten und Setzen eines Semaphors fuer Schreib-Shared Memory -*/
    if ( (sem_antwortid = semget(SEM_WRITEKEY, 1,
          S_IRWXU|S_IRWXG|S_IRWXO | IPC_CREAT | IPC_EXCL)) == -1)
        fehler_meld(FATAL_SYS, "semget-Fehler (Schreib-Shared Memory)");
    if (semctl(sem_antwortid, 0, SETVAL, (int)1) == -1)
        fehler_meld(FATAL_SYS, "semctl-Fehler (Schreib-Shared Memory)");

    /*--- Noch keine Saetze im Lese- und Schreib-Shared Memory vorhanden ---*/
    shm_anford->satznr = -1;

```

```

lese_satznr = 0;
shm_antwort->satznr = -1;

while (1) {
    P(sem_anfordid);

    /*--- Lesen einer Client-Anforderung -----*/
    if (shm_anford->anforderung[lese_satznr].ungelesen == 1) {
        anforderung = shm_anford->anforderung[lese_satznr];
        shm_anford->anforderung[lese_satznr].ungelesen = 0;
        lese_satznr = ++lese_satznr%SHM_MAXSAETZE;

        V(sem_anfordid);

        /*--- Bei Clientnr. 1000 sind shared memories und -----*/
        /*   Semaphore zu loeschen, und Server beendet sich -----*/
        if (anforderung.client_nr == 1000)
            beende_server(0);
        pid      = anforderung.pid;
        client_nr = anforderung.client_nr;
        genauigkeit = anforderung.genauigkeit;
        dividend   = anforderung.divident;
        divisor    = anforderung.divisor;

        /*--- Berechnen des Ergebnisses -----*/
        quotient = dividend / divisor;
        sprintf(ergebnis, "%5d/%5d = %d.", dividend, divisor, quotient);
        dividend=dividend%divisor*10;
        for (i=1 ; i<=genauigkeit ; i++) {
            sprintf(ergebnis, "%s%d", ergebnis, quotient=dividend/divisor);
            dividend = dividend%divisor*10;
        }

        P(sem_antwortid);

        /*--- Schreiben des Ergebnisses fuer Client -----*/
        schreib_satznr = shm_antwort->satznr;
        schreib_satznr = ++schreib_satznr%SHM_MAXSAETZE;
        if (shm_antwort->antwort[schreib_satznr].ungelesen == 1) {
            fehler_meld(WARNUNG, "Server: Ueberlauf des Shared Memory");
            beende_server(1);
        }
        shm_antwort->antwort[schreib_satznr].pid = pid;
        strcpy(shm_antwort->antwort[schreib_satznr].ergebnis, ergebnis);
        shm_antwort->antwort[schreib_satznr].ungelesen = 1;
        shm_antwort->satznr = schreib_satznr;

        /*--- Client mit Signal darueber informieren, -----*/
        /*   dass Ergebnis im Shared Memory liegt -----*/
        if (kill(pid, SIGUSR1) == -1)
            fehler_meld(FATAL_SYS,
                "kann Signal SIGUSR1 nicht Prozess %d schicken", pid);
    }
}

```

```

        V(sem_antwortid);

    } else
        V(sem_anfordid);
    }
    exit(0);
}

/*----- beende_server -----*/
static void beende_server(int exit_wert)
{
    if (shmctl(shm_antwortid, IPC_RMID, NULL) == -1)
        fehler_meld(FATAL_SYS, "kann Schreib-Shared Memory nicht loeschen");
    if (shmctl(shm_anfordid, IPC_RMID, NULL) == -1)
        fehler_meld(FATAL_SYS, "kann Lese-Shared Memory nicht loeschen");
    if (semctl(sem_antwortid, 0, IPC_RMID, (int)0) == -1)
        fehler_meld(FATAL_SYS, "kann Schreib-Semaphor nicht loeschen");
    if (semctl(sem_anfordid, 0, IPC_RMID, (int)0) == -1)
        fehler_meld(FATAL_SYS, "kann Lese-Semaphor nicht loeschen");
    fprintf(stderr, "---- Alle shared memories und Semaphore geloescht\n");
    fprintf(stderr, "---- Server: Ende ----\n");
    exit(exit_wert);
}

```

Programm 18.8 (**smdivser.c**): Server für Division mit beliebiger Genauigkeit

Das Programm 18.6 (**smdivcli.c**) ist die Client-Implementierung, die alle ihre Anforderungen in das dafür vom Server eingerichtete Shared Memory schreibt und die Antworten des Servers aus dem anderen eigens dafür eingerichteten Shared Memory liest.

Das Programm 18.6 (**smdivcli.c**) erhält seine Client-Nummer über die Kommandozeile und stellt dann mit **shmat** eine Verbindung (*attach*) zu den beiden vom Server eingerichteten Shared Memorys her. Mittels **semget** stellt es dann noch eine Beziehung zu den beiden vom Server eingerichteten Semaphoren her, bevor es dann die zu dividierenden Zahlen und die Genauigkeit zufällig ermittelt.

Diese und weitere Informationen schreibt das Programm 18.6 (**smdivcli.c**) in das Shared Memory für Client-Anforderungen. Danach suspendiert es seine Ausführung so lange, bis es vom Server das Signal **SIGUSR1** empfängt, das ihm mitteilt, daß der Server seine Anforderung bearbeitet hat und eine Antwort hierzu im entsprechenden Shared Memory für Server-Antworten bereitliegt. Da eventuell mehrere Antworten in diesem Shared Memory liegen, identifiziert das Client-Programm die für ihn gedachte Antwort mittels der vom Server dorthin geschriebenen PID.

```

#include <signal.h>
#include <time.h>
#include <limits.h>
#include <stddef.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/time.h>

```

```

#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "eighdr.h"
#include "sm.h"
#include "pv.h"

static void sig_usr1(int signr);
static void beende_client(int client_nr, int exit_wert);
static void delay(long mikrosek);

static anforder_shm      *shm_anforder;
static antwort_shm       *shm_antwort;

int
main(int argc, char *argv[])
{
    int                client_nr,
                      shm_anforderid, shm_antwortid,
                      sem_anforderid, sem_antwortid;
    int                lese_satznr, schreib_satznr, startnr;
    anforder_satz      anforderung;
    antwort_satz        antwort;
    int                genauigkeit, dividend, divisor, quotient, i, anzahl;
    pid_t              pid = getpid();
    struct sigaction    sa;

    /*--- Testen und Umwandeln des Kommandozeilenarguments ---*/
    if (argc != 2)
        fehler_meld(FATAL, "usage: %s client_nr", argv[0]);
    if ( (client_nr = atol(argv[1])) == 0)
        fehler_meld(FATAL, "Argument muss eine Clientnummer sein");

    /*--- Zufallszahlengenerator initialisieren -----*/
    srand(time(NULL)+client_nr);

    /*--- Oeffnen und Anbinden (attach) eines Shared Memory zum -----*/
    /*   Schreiben der Client-Anforderungen (Server liest sie von dort) ---*/
    if ( (shm_anforderid = shmget(SHM_READKEY, 0, 0)) == -1)
        fehler_meld(FATAL_SYS,
                    "Client%d: shmget-Fehler (Lese-shm)", client_nr);
    if ( (shm_anforder =
          (anforder_shm *)shmat(shm_anforderid, NULL, 0)) == (void *)-1)
        fehler_meld(FATAL_SYS,
                    "Client%d: shmat-Fehler (anforder_shm)", client_nr);

    /*--- Oeffnen und Anbinden (attach) eines Shared Memory zum -----*/
    /*   Lesen der Serverantworten (Server schreibt sie dorthin) -----*/
    if ( (shm_antwortid = shmget(SHM_WRITEKEY, 0, 0)) == -1)
        fehler_meld(FATAL_SYS,
                    "Client%d: shmget-Fehler (Schreib-shm)", client_nr);
    if ( (shm_antwort =
          (antwort_shm *)shmat(shm_antwortid, NULL, 0)) == (void *)-1)
        fehler_meld(FATAL_SYS,

```



```

        "Client%d: shmat-Fehler (antwort_shm)", client_nr);

/*--- Oeffnen des Semaphors fuer Lese-Shared Memory -----*/
if ( (sem_anforderid = semget(SEM_READKEY, 0, 0)) == -1)
    fehler_meld(FATAL_SYS,
        "Client%d: semget-Fehler (Lese-shm)", client_nr);

/*--- Oeffnen des Semaphors fuer Scheib-Shared Memory -----*/
if ( (sem_antwortid = semget(SEM_WRITEKEY, 0, 0)) == -1)
    fehler_meld(FATAL_SYS,
        "Client%d: semget-Fehler (Schreib-shm)", client_nr);

anzahl = rand()%10+1;  /*-- Anzahl der Berechnungen --*/

for (i=1; i<=anzahl; i++) {
    anforderung.pid      = pid;
    anforderung.client_nr = client_nr;
    anforderung.genauigkeit = rand()%45+1;
    anforderung.divident   = rand()%SHRT_MAX+1;
    anforderung.divisor    = rand()%SHRT_MAX+1;
    anforderung.ungelesen  = 1;

    /*--- Schreiben einer Client-Anforderung -----*/
    while (1) {
        P(sem_anforderid);

        schreib_satznr = shm_anforder->satznr;
        schreib_satznr = ++schreib_satznr%SHM_MAXSAETZE;
        if (shm_anforder->anforderung[schreib_satznr].ungelesen == 0)
            break;

        V(sem_anforderid);
    }

    shm_anforder->satznr = schreib_satznr;
    shm_anforder->anforderung[schreib_satznr] = anforderung;

    V(sem_anforderid);

    /*--- Abbruch bei Client mit Nummer 1000 -----*/
    if (client_nr == 1000) {
        delay(1000);
        beende_client(client_nr, 0);
    }

    /*--- Warten auf Server-Antwort (Server schickt Signal SIGUSR1) ----*/
    sa.sa_handler = sig_usr1;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGUSR1, &sa, NULL) == -1)
        fehler_meld(FATAL_SYS, "sigaction-Fehler");
    sigsuspend(&sa.sa_mask);
    if (errno != EINTR)
        fehler_meld(FATAL_SYS, "sigsuspend-Fehler");
}

```

```

/*--- Lesen von Server-Antworten -----*/
P(sem_antwortid);

startnr = lese_satznr = shm_antwort->satznr;
while (shm_antwort->antwort[lese_satznr].pid != pid) {
    lese_satznr = (lese_satznr>0) ? --lese_satznr : SHM_MAXSAETZE-1;
    if (lese_satznr == startnr) {
        fehler_meld(WARNUNG, "Synchronisation inkonsistent");
        beende_client(client_nr, 1);
    }
}
antwort = shm_antwort->antwort[lese_satznr];
shm_antwort->antwort[lese_satznr].ungelesen = 0;

V(sem_antwortid);

/*--- Ausgabe des vom Server gelieferten Ergebnisses --*/
printf("....Client%d: %s\n", client_nr, antwort.ergebnis);

delay(rand()%100000);
}

beende_client(client_nr, 0);
}

/*----- sig_usr1 -----*/
static void sig_usr1(int signr)
{
    return;
}

/*----- beende_client -----*/
static void beende_client(int client_nr, int exit_wert)
{
    if (shmdt((char *)shm_antwort) == -1)
        fehler_meld(FATAL_SYS, "kann Schreib-Shared Memory nicht loeschen");
    if (shmdt((char *)shm_anforder) == -1)
        fehler_meld(FATAL_SYS, "kann Lese-Shared Memory nicht loeschen");
    fprintf(stderr, "--- Client%d: Ende ---\n", client_nr);

    exit(exit_wert);
}

/*----- delay -----*/
static void delay(long mikrosek)
{
    struct timeval timeout;

    timeout.tv_sec = mikrosek / 1000000;
    timeout.tv_usec = mikrosek % 1000000;
    select(0, NULL, NULL, NULL, &timeout);
}

```

Programm 18.9 (smdivcli.c): Client für Division mit beliebiger Genauigkeit

Nachdem man die Programme 18.5 (smdivser.c) und 18.6 (smdivcli.c) kompiliert und gelinkt hat

```
cc -o smdivser smdivser.c pv.c fehler.c
cc -o smdivcli smdivcli.c pv.c fehler.c
```

läßt sich dieses Client-Server-Modell mit dem folgenden Bourne-Shellskript smtest testen:

```
$ cat smtest
#!/bin/sh

if [ $# -lt 1 ]
then
    echo "usage: $0 clientzahl"
    exit 1
fi

#..... Starten des Servers im Hintergrund
smdivser &
sleep 1 # Sicherstellen, dass Server seine Initialisierungen gemacht hat

#..... Starten der Clients im Hintergrund.....
i=1
while [ $i -le $1 ]
do
    smdivcli $i &
    eval pid$i=$!
    i=`expr $i + 1`
done

#..... Auf Beendigung aller Clients warten .....
i=1
while [ $i -le $1 ]
do
    eval wait \${pid$i} 2>/dev/null
    i=`expr $i + 1`
done

#..... Ende-Meldung an Server .....
smdivcli 1000
$ smtest 3
....Client1: 31270/10295 = 3.03739679456046624
....Client1: 31234/11292 = 2.7660290471130003542330853701
....Client2: 1020/ 4337 = 0.235185612174314041964491584044270232879870878
....Client3: 2279/ 1356 = 1.680678466076696165191740412979351
....Client1: 4118/ 1096 = 3.7572
....Client2: 12701/ 9822 = 1.2931174913459580533496232
....Client3: 19819/ 9086 = 2.18126788465771516618974
....Client1: 21858/25343 = 0.8624
....Client2: 15895/ 354 = 44.901129
....Client3: 21782/16366 = 1.3309299767
....Client1: 30182/16017 = 1.8843728538427920334644
```

```

....Client2: 13073/14646 = 0.892598661750
....Client3: 8679/25682 = 0.3379409703294135
....Client1: 14056/16719 = 0.8407201387642801602966684610323583946408
....Client2: 20151/12024 = 1.675898203592814371257485029
....Client3: 15136/10027 = 1.5095242844
....Client1: 21573/24430 = 0.88305362259516987310
....Client2: 24415/22862 = 1.06792931502055813139707812089930889
....Client3: 13211/ 9551 = 1.38320594702125431891948487069
....Client1: 9703/ 4782 = 2.02906733584274362191551652028439
....Client2: 20301/25973 = 0.781619373965271
....Client3: 3282/26747 = 0.122705350
....Client1: 15077/14572 = 1.03465550370573702992039527861652484216
....Client2: 1236/ 8483 = 0.1457031710479783095602970647176706353
....Client3: 29769/10333 = 2.8809639020613568179618697377334752733959159
--- Client1: Ende ---
....Client2: 12818/15037 = 0.8524306710115049544457
....Client3: 11688/31652 = 0.369265765196512068747630481486162
--- Client3: Ende ---
....Client2: 24856/ 6697 = 3.71151261758996565626399880543526
--- Client2: Ende ---
---- Alle shared memories und Semaphore geloescht
---- Server: Ende ----
--- Client1000: Ende ---
$

```

18.5 Übung

18.5.1 Adresse von angebundenem (attached) Shared Memory

Erstellen Sie ein Programm `sharadr.c`, das die Adresse ausgibt, an der der Kern Shared-Memory-Segmente plazierte, die mit einer Adresse von 0 angebunden wurden. Zusätzlich sollte dieses Programm `sharadr.c` noch anzeigen, an welchen Adressen sich der Stack, der Heap und nicht initialisierte Daten befinden.

18.5.2 Unerlaubtes Lesen von Messages durch fremde Prozesse

Was passiert, wenn ein fremder Prozeß eine nicht für ihn gedachte Message aus einer Message-Queue liest, die für den Server und seine Clients eingerichtet wurde? Welche Kenntnisse muß ein fremder Prozeß haben, um aus einer nicht für ihn eingerichteten Message-Queue zu lesen?

18.5.3 Kreieren von Message-Queues mit und ohne `IPC_PRIVATE`

Erstellen Sie ein Programm `msgqpriv.c`, das folgendes tut:

1. In einer Schleife, die es fünf Mal durchläuft, führt es jedesmal die folgenden Schritte durch: Kreieren einer Message-Queue, Ausgeben der Kennung dieser Message-Queue und anschließendes Löschen dieser Message-Queue.

2. In einer weiteren Schleife, die es wieder fünf Mal durchläuft, führt es nun jedesmal die folgenden Schritte durch: Kreieren einer Message-Queue mit dem Schlüssel `IPC_PRIVATE` und Eintragen einer Message in diese Message-Queue.

Starten Sie dann dieses Programm und lassen Sie sich nach dessen Beendigung die noch existierenden Message-Queues mit dem Kommando **ipcs** anzeigen.

18.5.4 Wortstatistik zu einer Textdatei (Vorsicht mit internen Zeigern)

Erstellen Sie ein Programm `wortstat.c`, das eine Wortstatistik zu den auf der Kommandozeile angegebenen Textdateien erstellt. Für das Speichern und Zählen der einzelnen Wörter soll dabei ein Binärbaum verwendet werden, der in einem Shared Memory unterzubringen ist. Während der Elternprozeß die entsprechenden Textdateien liest, die Wörter herausfiltert und in Form eines Binärbaums im Shared Memory ablegt, soll der Kindprozeß dem Benutzer die vom Elternprozeß erstellte Wortstatistik (aus dem Binärbaum im Shared Memory) ausgeben. Der Benutzer soll dabei über Eingabe von Anfangsbuchstaben wählen können, welche Wörter er ausgegeben haben möchte.

Nachdem man dieses Programm `wortstat.c` kompiliert und gelinkt hat

```
cc -o wortstat wortstat.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ cat eingabe.txt
```

```
Dies ist ein sehr schoenes Programm, da es
eine Wortstatistik zu einem beliebigen Text erstellt.
Das Programm laesst den Benutzer waehlen,
welchen Anfangs-Buchstabenbereich es von der
Wortstatistik ausgeben soll.
Vielen Spass mit den vielen Offsets im shared memory,
wenn der binaere Baum waechst.
Ein Mann und ein Speicher.
```

```
Ein wirklich dummer Text.
```

```
$ wortstat eingabe.txt
```

```
Buchstabenbereich (a-z, sonst=Ende):
```

```
von: a
```

```
bis: z
```

```
anfangs      : 1
ausgeben     : 1
baum         : 1
beliebigen   : 1
benutzer     : 1
binaere      : 1
buchstabenbereich : 1
da           : 1
das          : 1
den          : 2
der          : 2
```

```

dies      : 1
dummer    : 1
ein       : 4
eine      : 1
einem     : 1
erstellt  : 1
es        : 2
im        : 1
ist       : 1
laesst    : 1
mann      : 1
memory    : 1
mit       : 1
offsets   : 1
programm   : 2
schoenes  : 1
sehr      : 1
shared    : 1
soll      : 1
spass     : 1
speicher  : 1
text      : 2
und       : 1
vielen    : 2
von       : 1
waechst   : 1
waehlen   : 1
welchen   : 1
wenn      : 1
wirklich  : 1
wortstatistik : 2
zu        : 1
Buchstabenbereich (a-z, sonst=Ende):
    von: e
    bis: e
ein       : 4
eine      : 1
einem     : 1
erstellt  : 1
es        : 2
Buchstabenbereich (a-z, sonst=Ende):
    von: u
    bis: w
und       : 1
vielen    : 2
von       : 1
waechst   : 1
waehlen   : 1
welchen   : 1
wenn      : 1
wirklich  : 1
wortstatistik : 2
Buchstabenbereich (a-z, sonst=Ende):

```

von: 0

\$

Bei diesem Programm `wortstat.c` sollten Sie beachten, daß es gefährlich ist, die Adressen der Knoten des Binärbaums einfach im Shared Memory abzulegen, da es möglich ist, daß die Prozesse (hier Eltern- und Kindprozeß) das Shared-Memory-Segment an verschiedenen Adressen anbinden. Anstelle der Adressen (Zeiger) sollten deshalb im Shared Memory die Offsets zum Beginn des Shared Memory verwendet werden.

19 Stream Pipes, Client-Server-Realisierungen und Netzwerkprogrammierung

*Immer strebe zum Ganzen und, kannst
du selber kein Ganzes werden,
als dienendes Glied schließ an ein Ganzes dich an!*

Schiller

In den beiden vorherigen Kapiteln wurden die klassischen Formen der Interprozeßkommunikation (Pipes, FIFOs, Message-Queues, Semaphore und Shared Memory) vorgestellt.

In diesem Kapitel werden neuere Formen der Interprozeßkommunikation: *Stream Pipes* und *benannte Stream Pipes* vorgestellt. Diese beiden Methoden erlauben z.B. den Austausch von Filedeskriptoren zwischen verschiedenen Prozessen oder die Kommunikation von Clients mit einem Server, der als Dämonprozeß abläuft.

19.1 Client-Server-Eigenschaften der klassischen IPC-Methoden

Bevor Stream Pipes vorgestellt werden, sollen in diesem Kapitel nochmals die Eigenschaften und Schwächen der klassischen Formen der Interprozeßkommunikation hervorgehoben werden.

19.1.1 Client-Server-Realisierung mit Pipes

Bei der einfachsten Form einer Client-Server-Realisierung richtet ein Clientprozeß sich mit **fork** und **exec** einen eigenen Server ein. Vor dem **fork**-Aufruf richtet dabei der Client zwei Pipes ein, um den Datenaustausch in beide Richtungen zu ermöglichen.

Dann ist es z.B. möglich, daß der Client bestimmte Dateien nicht selbst öffnen darf, sondern diese Dateien nur vom Server geöffnet werden dürfen. Mit einer solchen Vorgehensweise kann man dann z.B. die von Unix vorgegebenen Benutzergruppen *user*, *group*, *others* um eigene erweitern. Der Server, bei dem das Set-User-ID Bit gesetzt ist, würde dabei anhand der realen User-ID feststellen, ob der betreffende Client Zugriff auf die geforderte Datei hat oder nicht. Der Server unterhält in diesem Fall zusätzliche Benutzergruppen.

Der Nachteil dieser Client-Server-Realisierung ist, daß der Server nur Daten aus den geöffneten Dateien an die Clients zurückliefern kann. Die Rückgabe eines geöffneten File-

deskriptors ist nicht möglich, da der Server ein Kindprozeß vom Client ist und die Weitergabe eines Filedeskriptors nur von einem Eltern- zu einem Kindprozeß und nicht umgekehrt möglich ist.

19.1.2 Client-Server-Realisierung mit FIFOs

In Kapitel 17.3 wurde eine Client-Server-Realisierung mit FIFOs gezeigt. Da der Server dort als Dämonprozeß im Hintergrund läuft und somit keine Verwandtschaft zu den Clients hat, können hierbei keine normalen Pipes, sondern müssen FIFOs verwendet werden. Es wurde in diesem Beispiel auch gezeigt, daß zwar für die Client-Anforderungen eine FIFO ausreicht, aber für die Server-Antworten je Client eine eigene FIFO eingerichtet werden mußte.

19.1.3 Client-Server-Realisierung mit Message-Queues, Shared Memory und Semaphoren

Für eine Client-Server-Realisierung mit Message-Queues gibt es grundsätzlich zwei Möglichkeiten.

1. Kommunikation über eine Message-Queue

Bei dieser Vorgehensweise wird der Message-Typ verwendet, um den Empfänger der Message festzulegen. So könnten z.B. alle Messages, die die Clients an den Server schicken, als Message-Typ den Wert 1 haben. Die Prozeß-ID des sendenden Clients muß dabei in der Message selbst enthalten sein. Bei den Antworten des Servers wird diese Prozeß-ID dann vom Server als Message-Typ angegeben. Somit können die Clients die für sie gedachten Server-Antworten identifizieren und lesen.

2. Kommunikation über Client-spezifische Message-Queues

Bei dieser Vorgehensweise richtet jeder Client für Server-Antworten seine eigene Message-Queue (mit Schlüssel `IPC_PRIVATE`) zum Server ein. Der Server seinerseits richtet für die Client-Anforderungen eine eigene Message-Queue ein, die allen Clients über einen vereinbarten Schlüssel bekannt ist. Während alle Clients ihre Anforderungen über ein und dieselbe Message-Queue an den Server schicken, empfangen sie die für sie speziell gedachten Server-Antworten über ihre privaten Message-Queues. Damit der Server die Kennung der jeweiligen Message-Queue kennt, muß zumindest jeweils die erste Client-Anforderung diese Kennung beinhalten. In Kapitel 18.2 wurde dazu ein Beispiel gegeben. Diese zweite Vorgehensweise hat den Nachteil, daß man hierbei sehr verschwenderisch mit einer nur begrenzt im System verfügbaren Ressource umgeht, denn die Anzahl von möglichen Message-Queues in einem System ist nicht unendlich.

Eine dieser beiden Techniken kann auch für Client-Server-Realisierungen benutzt werden, wenn diese mit Shared Memory und Semaphoren implementiert wurden.

Bei allen diesen Formen der Interprozeßkommunikation mit Message-Queues, Shared Memory und Semaphoren besteht das Problem in der korrekten Identifizierung des Clients durch den Server. Wenn der Client einen privilegierten Zugriff vom Server fordert, muß der Server unbedingt wissen, wer der Client wirklich ist und ob dieser dazu auch die Rechte hat. Dies ist z.B. der Fall, wenn der Server ein Set-User-ID-Programm ist. Eine solche Identifizierung des Clients durch den Server ist jedoch bei diesen Formen der IPC nicht möglich.

In diesem Kapitel wird unter anderem auch eine Methode vorgestellt, mit der der Server leicht und elegant die effektive User-ID und die effektive Group-ID eines Clients erfragen kann.

19.2 Stream Pipes

Eine Stream Pipe unterscheidet sich von einer normalen Halbduplex-Pipe (siehe Kapitel 17.2) nur darin, daß sie im Vollduplex-Betrieb arbeitet, also anders als die normale Pipe eine »Zwei-Wege-Pipe« ist (siehe Abbildung 19.1).

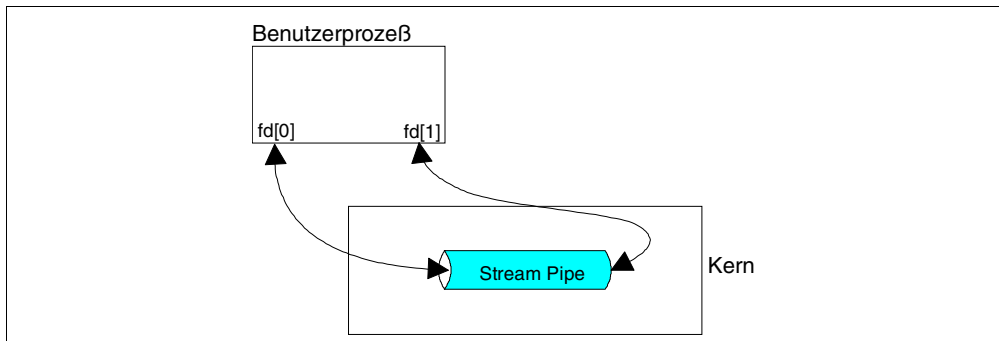


Abbildung 19.1: Eine Stream Pipe

Nachfolgend werden mögliche Realisierungen einer Stream Pipe unter SVR4 und BSD-Unix vorgestellt. Dazu wird jeweils eine Funktion **stream_pipe** angegeben, die den gleichen Prototyp wie die Funktion **pipe** hat. Anders als bei der **pipe**-Funktion sind aber die in das Argument **fd** geschriebenen Filedeskriptoren (nach der Rückkehr aus **stream_pipe**) gleichzeitig zum Lesen und zum Schreiben geöffnet.

19.2.1 stream_pipe – Realisierung einer Stream Pipe in SVR4

Das folgende Programm 19.1 (`spipesv.c`) zeigt die Realisierung einer Stream Pipe unter SVR4. Die Funktion **stream_pipe** ruft dazu nur die **pipe**-Funktion auf, die unter SVR4 ein Vollduplex-Pipe einrichtet.

```
#include "eighdr.h"

int
stream_pipe(int fd[2])
{
    return( pipe(fd) );
}
```

Programm 19.1 (**spipesv.c**): Realisierung einer Stream Pipe in SVR4

Hinweis

In SVR4 ist eine Pipe lediglich eine Verbindung zwischen STREAM-Köpfen. Abbildung 19.2 verdeutlicht dies.

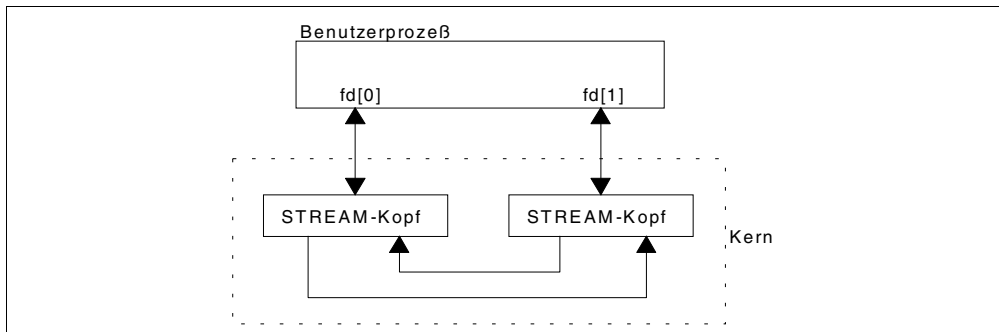


Abbildung 19.2: Realisierung einer Pipe in SVR4

19.2.2 stream_pipe – Realisierung einer Stream Pipe in BSD/Linux

Das folgende Programm 19.2 (**spipebsd.c**) zeigt die Realisierung einer Stream Pipe unter BSD-Unix. Die Funktion **stream_pipe** kreiert dabei mit dem **socketpair**-Aufruf zwei *Unix Domain Stream Sockets*, die miteinander verbunden sind.

```
#include <sys/types.h>
#include <sys/socket.h>
#include "eighdr.h"

int
stream_pipe(int fd[2])
{
    return( socketpair(AF_UNIX, SOCK_STREAM, 0, fd) );
}
```

Programm 19.2 (**spipebsd.c**): Realisierung einer Stream Pipe in BSD-Unix

Hinweis

Die Funktion **stream_pipe** aus Programm 19.2 (**spipebsd.c**) kann ab 4.2BSD benutzt werden.

Seit 4.2BSD werden auch normale Pipes (bei einem **pipe**-Aufruf) mit einem **socketpair**-Aufruf eingerichtet. Da jedoch in BSD-Unix die Funktion **pipe** die Leseseite des ersten Filedeskriptors und die Schreibseite des zweiten Filedeskriptors schließt, muß zum Einrichten einer Vollduplex-Pipe **socketpair** direkt aufgerufen werden.

19.2.3 Kommunikation mit einem Koprozeß über Stream Pipe

In Kapitel 17.2 wurde das Koprozeß-Programm 17.10 (`romzahl.c`) entwickelt, das Zahlen von der Standardeingabe liest, diese in die entsprechende römische Darstellung umwandelt und dann die römische Zahl (String) auf seine Standardausgabe schreibt.

Dieses Filterprogramm `romzahl` kann von anderen Programmen als Koprozeß gestartet werden, indem sie mit **fork** einen Kindprozeß kreieren und diesen mit einem **exec**-Aufruf mit dem Programm `romzahl` überlagern.

Während im Programm 17.11 (`romkomm.c`) zwei einfache Pipes eingerichtet wurden, um mit dem Koprozeß zu kommunizieren, soll hier ein Programm 19.3 (`romkomm3.c`) entwickelt werden, bei dem diese Kommunikation zum Koprozeß `romzahl` über eine Stream Pipe erfolgt.

```
#include <signal.h>
#include "eighdr.h"

static void sig_pipe(int signr);    /* eigener Signalhandler */

int
main(void)
{
    int    n, spipe[2];
    pid_t  pid;
    char   zeile[MAX_ZEICHEN];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        fehler_meld(FATAL_SYS, "signal-Fehler");

    if (stream_pipe(spipe) < 0)
        fehler_meld(FATAL_SYS, "pipe-Fehler");

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid > 0) { /*----- Elternprozess -----*/
        close(spipe[1]);
        while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
            n = strlen(zeile);
            if (write(spipe[0], zeile, n) != n)
                fehler_meld(FATAL_SYS, "Fehler beim Schreiben in Stream Pipe");
            if ( (n = read(spipe[0], zeile, MAX_ZEICHEN)) < 0)
                fehler_meld(FATAL_SYS, "Fehler beim Lesen aus Stream Pipe");
            if (n == 0) {
                fehler_meld(WARNUNG, "Kind hat Stream Pipe geschlossen");
            }
        }
    }
}
```

```

        break;
    }
    zeile[n] = '\0';
    if (fputs(zeile, stdout) == EOF)
        fehler_meld(FATAL_SYS, "fputs-Fehler");
}
if (ferror(stdin))
    fehler_meld(FATAL_SYS, "fgets-Fehler (in stdin)");
exit(0);

} else { /*----- Kindprozess -----*/
    close(spiped[0]);
    if (spiped[1] != STDIN_FILENO) {
        if (dup2(spiped[1], STDIN_FILENO) != STDIN_FILENO)
            fehler_meld(FATAL_SYS, "dup2-Fehler (bei stdin)");
    }
    if (spiped[1] != STDOUT_FILENO) {
        if (dup2(spiped[1], STDOUT_FILENO) != STDOUT_FILENO)
            fehler_meld(FATAL_SYS, "dup2-Fehler (bei stdout)");
    }
    if (execl("./romzahl", "romzahl", NULL) < 0)
        fehler_meld(FATAL_SYS, "execl-Fehler");
}
}

static void sig_pipe(int signr)
{
    printf(".....SIGPIPE abgefangen.....\n");
    exit(1);
}

```

Programm 19.3 (romkomm3.c): Kommunizieren mit Koprozeß (romzahl) über eine Stream Pipe

In Programm 19.3 (romkomm3.c) benutzt der Elternprozeß `spiped[0]` zum Lesen und Schreiben aus der eingerichteten Stream Pipe. Der Kindprozeß dupliziert `spiped[1]` sowohl auf die Standardeingabe als auch auf die Standardausgabe. Abbildung 19.3 zeigt die daraus resultierende Konstellation.

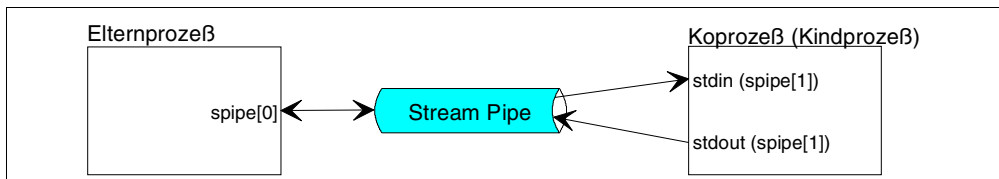


Abbildung 19.3: Stream Pipe zwischen Eltern- und Koprozeß (Kindprozeß)

Dieses Programm 19.3 (romkomm3.c) kompiliert und linkt man.

```

cc -o romkomm3 romkomm3.c spipesv.c fehler.c (in SVR4)
cc -o romkomm3 romkomm3.c spipedbsd.c fehler.c (in BSD oder Linux)

```

Als Koprozeß wird hierbei das Programm 17.10 (romzahl.c)¹ verwendet. Es ergibt sich dann z.B. folgender Ablauf:

```
$ romkomm3
7
.....VII
1295
.....MCCXCV
acht
ungueltige Eingabe
15999
.....MMMMMMMMMMMMMCMXCIX
Ctrl-D
$
```

19.3 Austausch von Filedeskriptoren zwischen Prozessen

Wenn zwei Prozesse die gleiche Datei öffnen, so ergibt sich die in Abbildung 19.4 gezeigte Konstellation.

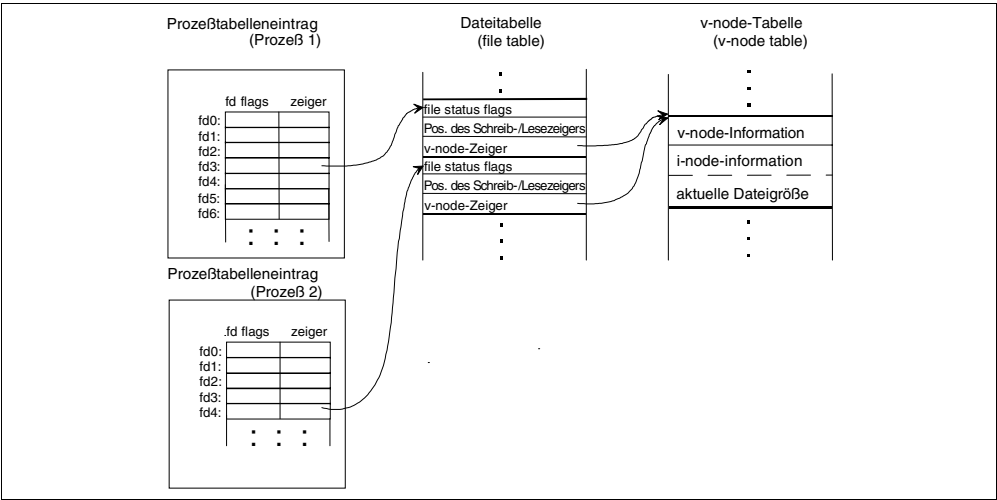


Abbildung 19.4: Zwei Prozesse haben zu einem Zeitpunkt die gleiche Datei geöffnet

Obwohl beide Prozesse für diese Datei den gleichen v-node-Tabelleneintrag benutzen, hat doch jeder einzelne Prozeß seinen eigenen Dateitabelleneintrag für diese Datei.

1. Es sollte zuvor kompiliert und gelinkt werden: cc -o romzahl.c fehler.c

Wenn aber nun – wie in vielen Client-Server-Anwendungen – gefordert ist, daß zwei Prozesse auch den gleichen Dateitableneintrag für eine Datei benutzen (siehe Abbildung 19.5), so muß der entsprechende Filedeskriptor von einem Prozeß an den anderen Prozeß weitergeleitet werden.

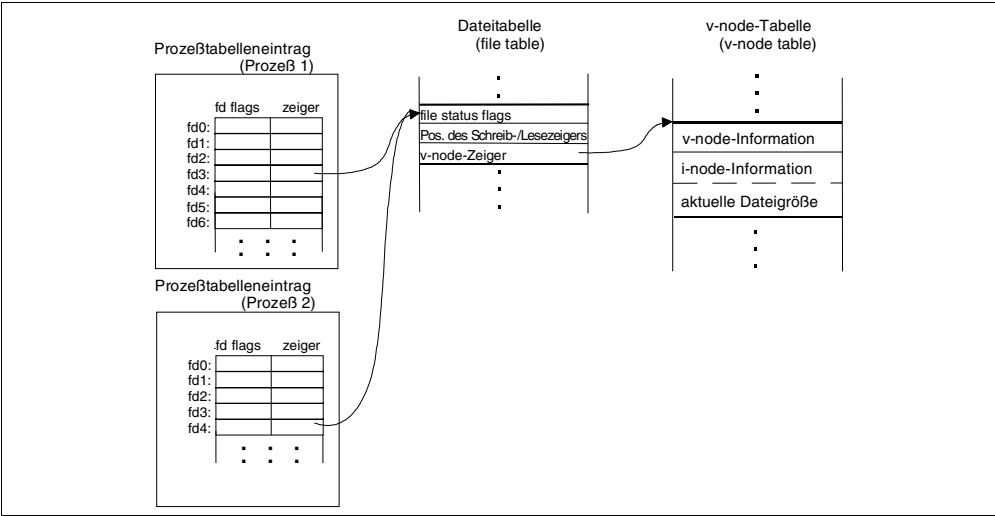


Abbildung 19.5: Zwei Prozesse benutzen den gleichen Dateitableneintrag für eine Datei

In Abbildung 19.5 ist erkennbar, daß beim Schicken des entsprechenden Filedeskriptors eigentlich nur die Adresse des entsprechenden Dateitableneintrags geschickt und diese dann dem ersten freien Filedeskriptor im Empfängerprozeß zugeordnet werden muß. Im Prinzip ist hierbei das gleiche Verhalten gefordert, das für das Vererben von Filedeskriptoren an Kindprozesse bei einem *fork*-Aufruf gilt.

Normalerweise schließt der Senderprozeß nach dem Schicken eines Filedeskriptors diesen anschließend. Dieses Schließen im Senderprozeß bewirkt nicht das Schließen der zugehörigen Datei, da noch ein offener Filedeskriptor (der geschickte) für diese Datei existiert.

19.3.1 send_fd, empfang_fd und send_fehl – Eigene Funktionen zum Austausch von Filedeskriptoren

Hier werden die drei Funktionen *send_fd*, *empfang_fd* und *send_fehl* beschrieben, die den Austausch von Filedeskriptoren zwischen Prozessen ermöglichen. Diese Funktionen wurden vom Buch »Advanced Programming in the UNIX Environment, W. Richard Stevens« in abgeänderter Form übernommen


```
#include "eighdr.h"

int send_fd(int spipe_fd, int fd);

int send_fehl(int spipe_fd, int status, const char *fehlmeld);

                                                    beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

int empfang_fd(int spipe_fd,
               ssize_t (*benutzerfunk)(int, const void *, size_t));

                                                    gibt zurück: Filedeskriptor (bei Erfolg); <0 bei Fehler
```

Um einen Filedeskriptor über die Stream Pipe *spipe_fd* zu schicken, muß der Senderprozeß (normalerweise ein Server) **send_fd** aufrufen. Mit **send_fehl** ist es ihm möglich, über die Stream Pipe *spipe_fd* eine Fehlermeldung an den Empfängerprozeß zu schicken. Das *status*-Argument spezifiziert dabei einen Fehlerstatus und muß einen Wert zwischen -1 und -255 haben.

Der Empfängerprozeß (normalerweise ein Client) kann mit **empfang_fd** einen von einem anderen Prozeß geschickten Filedeskriptor aus der entsprechenden Stream Pipe *spipe_fd* lesen. Wenn der Aufruf von **empfang_fd** erfolgreich war, liefert diese Funktion den entsprechenden Filedeskriptor als Rückgabewert, ansonsten liefert diese Funktion als Rückgabewert den entsprechenden Fehlerstatus, was der *status*-Wert (-1 bis -255) vom **send_fehl**-Aufruf des Senderprozesses ist.

Wurde beim **send_fehl**-Aufruf eine Fehlermeldung (*fehlmeld*) angegeben, so wird die beim **empfang_fd** angegebene *benutzerfunk* zur Abarbeitung dieser Fehlermeldung aufgerufen. Das erste Argument von *benutzerfunk* ist dabei die Konstante `STDERR_FILENO`, das zweite Argument die Adresse und das dritte Argument die Länge der Fehlermeldung. Oft wird für das *benutzerfunk*-Argument die Unix-Funktion **write** angegeben.

Die drei Funktionen **send_fd**, **send_fehl** und **empfang_fd** benutzen ein gemeinsames Protokoll.

send_fd

```
-----
| 0 | 0 | Filedeskriptor |
-----
Byte1 Byte2 .....
```

send_fehl

```
-----
| fehlmeld | 0 | status (1-255) |
-----
Byte1.... ByteX .....
```

empfang_fd

Diese Funktion liest solange aus der Stream Pipe, bis sie ein 0-Byte liest. Der bis dahin gelesene String wird dann der Funktion *benutzerfunk* als Argument übergeben. Das nächste gelesene Byte ist das Statusbyte. Hat dieses Statusbyte den Wert 0, so wurde ein Filedeskriptor geschickt. Bei einem Statuswert verschieden von 0, wurde kein Filedeskriptor geschickt.

Nachfolgend werden mögliche Implementierungen der Funktionen ***send_fd***, ***send_fehl*** und ***empfang_fd*** in den Systemen SVR4, 4.3BSD und neueren BSD-Systemen gezeigt.

19.3.2 Austausch von Filedeskriptoren in SVR4

Unter SVR4 können mittels der beiden ***ioctl***-Kommandos ***I_SENDFD*** und ***I_RECVFD*** Filedeskriptoren über Stream Pipes ausgetauscht werden. Der entsprechende Filedeskriptor wird dabei beim Senden als drittes Argument zu ***ioctl*** angegeben. Beim Empfangen eines Filedeskriptors wird als drittes Argument beim ***ioctl***-Aufruf die Adresse einer Variablen angegeben, deren Datentyp die Struktur *strrecvfd* ist:

```
struct strrecvfd {
    int    fd; /* Neuer Filedeskriptor          */
    uid_t  uid; /* effektive User-ID des Senders */
    gid_t  gid; /* effektive Group-ID des Senders */
    char fill[8];
}
```

Die Funktion ***empfang_fd*** liest aus der Stream Pipe solange, bis sie das erste 0-Byte liest. Das nächste Byte muß dann der Statuswert sein. Ist dieser Statuswert gleich 0, so wird zum Lesen des anschließenden Filedeskriptors ***ioctl*** mit dem ***I_RECVFD***-Kommando aufgerufen.

Programm 19.4 (*svr4.c*) zeigt eine mögliche Implementierung der Funktionen ***send_fd***, ***send_fehl*** und ***empfang_fd*** unter SVR4.

```
#include <sys/types.h>
#include <stropts.h>
#include "eighdr.h"

/*----- send_fd -----
 *      sendet einen Filedeskriptor an anderen Prozess
 *      wenn fd<0, so wird -fd als Fehler-Status geschickt          */
int send_fd(int spipefd , int fd)
{
    char protokoll[2] = { 0, 0 };

    if (fd < 0) {
        protokoll[1] = -fd; /* Status != 0 bedeutet Fehler */
        if (protokoll[1] == 0)
            protokoll[1] = 1; /* Abfangen von Ueberlaufen */
    }
    if (write(spipefd, protokoll, 2) != 2)
        return(-1);
}
```

```

    if (fd >= 0)
        if (ioctl(spipedfd, I_SENDFD, fd) < 0)
            return(-1);
    return(0);
}

/*----- empfang_fd -----
 *      empfaengt einen Filedeskriptor von einem anderen Prozess.
 *      Zusätzlich empfangene Daten werden von
 *      (*benutzerfunk)(STDERR_FILENO, puffer, byte_gelesen)
 *      verarbeitet. */
int empfang_fd(int spipedfd,
               ssize_t (*benutzerfunk)(int, const void *, size_t))
{
    int          neuafd, byte_gelesen, flag=0, status=-1;
    char         *zgr, puffer[MAX_ZEICHEN];
    struct strbuf daten;
    struct strrecvfd empfangfd;

    while (1) {
        daten.buf      = puffer;
        daten.maxlen = MAX_ZEICHEN;
        flag          = 0;
        if (getmsg(spipedfd, NULL, &daten, &flag) < 0)
            fehler_meld(FATAL_SYS, "getmsg-Fehler");
        if ( (byte_gelesen = daten.len) == 0) {
            fehler_meld(WARNUNG, "Verbindung abgebrochen (durch Server)");
            return(-1);
        }

        /* Durchlaufen des ganzen Puffers, wobei die eigentlichen Daten
         * mit einem 0-Byte abgeschlossen sind, dem dann der
         * Status folgt. Ein Statuswert von 0 bedeutet dabei, dass
         * ein Filedeskriptor empfangen wird. */
        for (zgr=puffer; zgr < &puffer[byte_gelesen]; ) {
            if (*zgr++ == 0) {
                if (zgr != &puffer[byte_gelesen-1])
                    fehler_meld(DUMP, "message inkonsistent");
                status = *zgr & 0xff;
                if (status == 0) {
                    if (ioctl(spipedfd, I_RECVFD, &empfangfd) < 0)
                        return(-1);
                    neuafd = empfangfd.fd;
                } else
                    neuafd = -status;
                byte_gelesen -= 2;
            }
        }
        if (byte_gelesen > 0)
            if ( (*benutzerfunk)(STDERR_FILENO, puffer, byte_gelesen)
                != byte_gelesen)
                return(-1);
        if (status >= 0)

```

```

        return(neufd);
    }
}

/*----- send_fehl -----
 *      sendet mit dem beschriebenen Protokoll einen Fehler.
 *      Diese Routine wird benutzt, wenn beabsichtigt war, einen
 *      Filedeskriptor zu schicken, aber ein Fehler aufgetreten ist.*/
int send_fehl(int spipefd, int status, const char *fehlmeld)
{
    int    n;

    if ( (n=strlen(fehlmeld)) > 0)
        if (writespz(spipefd, fehlmeld, n) != n) /*Senden der Fehlermeldung */
            return(-1);

    if (status >= 0)
        status = -1; /* Status muss negativ sein */

    if (send_fd(spipefd, status) < 0)
        return(-1);

    return(0);
}

```

Programm 19.4 (*svr4.c*): Die Funktionen **send_fd**, **send_fehl** und **empfang_fd** für SVR4

19.3.3 Austausch von Filedeskriptoren in 4.3BSD

Um Filedeskriptoren in 4.3BSD (z.B. SunOS) oder von BSD abstammenden Systemen auszutauschen, verwendet man die Funktionen **sendmsg** und **recvmsg** (siehe auch `sendmsg(2)` und `recvmsg(2)`). Bei beiden Funktionen muß als zweites Argument ein Zeiger auf die Struktur `msghdr` angegeben werden. Diese Struktur `msghdr`, die in `<sys/socket.h>` definiert ist, enthält alle notwendigen Informationen:

```

struct msghdr {
    caddr_t    msg_name;    /* Optionale Adresse */
    int        msg_namelen; /* Größe der Adresse */
    struct iovec *msg_iov;   /* Adresse der zu lesenden/schreibenden
                             Puffer */
    int        msg_iovlen;  /* Anzahl der Elemente
                             im Array msg_iov */
    caddr_t    msg_accrights; /* geschickte/empfangene
                             Zugriffsrechte */
    int        msg_accrightslen; /* Größe des
                             Zugriffsrechte-Puffers */
}

```

Die ersten beiden Komponenten dieser Struktur werden normalerweise zum Senden von Datagrammen in einer Netzwerkverbindung benutzt. So kann für jedes Datagramm eine Zieladresse spezifiziert werden.

Die nächsten beiden Komponenten ermöglichen die Angabe eines Arrays von Lese- oder Schreibpuffern (siehe auch Funktionen **readv** und **writev** in Kapitel 15.4).

Die beiden letzten Komponenten ermöglichen das Senden oder Empfangen von Zugriffsrechten. Filedescriptoren sind dabei die einzigen zu schickenden bzw. zu empfangenden Zugriffsrechte. Zum Senden oder Empfangen eines Filedescriptors muß sich in `msg_accrights` die Adresse des entsprechenden Filedescriptors befinden. Die Komponente `msg_accrightslen` gibt dabei die Größe dieses Filedescriptors (`sizeof(int)`) an. Beim Senden bzw. Empfangen eines Filedescriptors muß der Wert dieser Komponente größer als 0 sein.

Beim Empfangen eines Filedescriptors (**empfang_fd**) wird solange aus der Stream Pipe gelesen, bis das erste 0-Byte gelesen wird. Das nächste Byte ist dann der Statuswert. Ist dieser Statuswert gleich 0, so befindet sich der entsprechende Filedescriptor in der Komponente `msg_accrights`, wenn `msg_accrightslen` gleich `sizeof(int)` ist, andernfalls liegt ein Fehler vor.

Programm 19.5 (bsd4_3.c) zeigt eine mögliche Implementierung der Funktionen **send_fd**, **send_fehl** und **empfang_fd** unter 4.3BSD.

```
#include <sys/types.h>
#include <sys/socket.h> /* struct msghdr */
#include <sys/uio.h>    /* struct iovec */
#include <errno.h>
#include <stddef.h>
#include "eighdr.h"

/*----- send_fd -----
 *      sendet einen Filedescriptor an anderen Prozess
 *      wenn fd<0, so wird -fd als Fehler-Status geschickt
 */
int send_fd(int spipecfd , int fd)
{
    struct iovec    iov[1];
    struct msghdr   message;
    char            protokoll[2] = { 0, 0 };

    iov[0].iov_base = protokoll;
    iov[0].iov_len  = 2;
    message.msg_iov  = iov;
    message.msg_iovlen = 1;
    message.msg_name  = NULL;
    message.msg_namelen = 0;

    if (fd < 0) {
        message.msg_accrights = NULL;
        message.msg_accrightslen = 0;
        protokoll[1] = -fd; /* Status != 0 bedeutet Fehler */
        if (protokoll[1] == 0)
            protokoll[1] = 1; /* Abfangen von Ueberläufen */
    } else {
        message.msg_accrights = (caddr_t) &fd;
        message.msg_accrightslen = sizeof(int);
    }
}
```

```

    }

    if (sendmsg(spipedfd, &message, 0) != 2)
        return(-1);

    return(0);
}

/*----- empfang_fd -----
 *      empfaengt einen Filedeskriptor von einem anderen Prozess.
 *      Zusaetzlich empfangene Daten werden von
 *      (*benutzerfunk)(STDERR_FILENO, puffer, byte_gelesen)
 *      verarbeitet. */
int empfang_fd(int spipedfd,
               ssize_t (*benutzerfunk)(int, const void *, size_t))
{
    int          neuafd, byte_gelesen, status=-1;
    char         *zgr, puffer[MAX_ZEICHEN];
    struct iovec  iov[1];
    struct msghdr message;

    while (1) {
        iov[0].iov_base = puffer;
        iov[0].iov_len  = sizeof(puffer);
        message.msg_iov  = iov;
        message.msg_iovlen = 1;
        message.msg_name  = NULL;
        message.msg_namelen = 0;
        message.msg_accrightright = (caddr_t) &neuafd;
        message.msg_accrightrightlen = sizeof(int);

        if ( (byte_gelesen = recvmsg(spipedfd, &message, 0)) < 0)
            fehler_meld(FATAL_SYS, "recvmsg-Fehler");
        else if (byte_gelesen == 0) {
            fehler_meld(WARNUNG, "Verbindung abgebrochen (durch Server)");
            return(-1);
        }

        /* Durchlaufen des ganzen Puffers, wobei die eigentlichen Daten
           mit einem 0-Byte abgeschlossen sind, dem dann der
           Status folgt. Ein Statuswert von 0 bedeutet dabei, dass
           ein Filedeskriptor empfangen wird. */
        for (zgr=puffer; zgr < &puffer[byte_gelesen]; ) {
            if (*zgr++ == 0) {
                if (zgr != &puffer[byte_gelesen-1])
                    fehler_meld(DUMP, "message inkonsistent");
                status = *zgr & 0xff;
                if (status == 0) {
                    if (message.msg_accrightrightlen != sizeof(int))
                        fehler_meld(DUMP, "message inkonsistent");
                } else
                    neuafd = -status;
                byte_gelesen -= 2;
            }
        }
    }
}

```

```

    }
}
if (byte_gelesen > 0)
    if ( (*benutzerfunk)(STDERR_FILENO, puffer, byte_gelesen)
        != byte_gelesen)
        return(-1);
if (status >= 0)
    return(neufd);
}
}

/*----- send_fehl -----
 *      sendet mittels dem beschriebenen Protokoll einen Fehler.
 *      Diese Routine wird benutzt, wenn beabsichtigt war, einen
 *      Filedeskriptor zu schicken, aber ein Fehler aufgetreten ist.*/
int send_fehl(int spipefd, int status, const char *fehlmeld)
{
    int    n;

    if ( (n=strlen(fehlmeld)) > 0)
        if (writespez(spipefd, fehlmeld, n) != n) /*Senden der Fehlermeldung */
            return(-1);

    if (status >= 0)
        status = -1; /* Status muss negativ sein */

    if (send_fd(spipefd, status) < 0)
        return(-1);

    return(0);
}

```

Programm 19.5 (**bsd4_3.c**): Die Funktionen **send_fd**, **send_fehl** und **empfang_fd** für 4.3BSD

19.3.4 Austausch von Filedeskriptoren in neueren BSD-Systemen und in Linux

Unter neueren BSD-Systemen und auch unter Linux haben die beiden Komponenten **msg_accrights** und **msg_accrightslen** eine andere Bedeutung. Deswegen wurde dort die Struktur **msg_hdr** verändert:

```

struct msghdr {
    caddr_t    msg_name;      /* optionale Adresse          */
    int        msg_namelen;   /* Größe der Adresse          */
    struct iovec *msg_iov;     /* Adressen der zu
                               lesenden/schreibenden Puffer */
    int        msg_iovlen;    /* Anzahl der Elemente
                               im Array msg_iov          */
    caddr_t    msg_control;    /* Kontrolldaten              */
    u_int      msg_controllen; /* Größe der Kontrolldaten    */
    int        msg_flags;      /* Flags bei empfangener Message */
}

```

Die Komponente `msg_control` ist in neueren BSD-Systemen ein Zeiger auf die Struktur `cmsghdr` (Kopf der Kontrollmessage):

```
struct cmsghdr {
    u_int cmsg_len; /* Anzahl der Daten */
    int cmsg_level; /* Protokoll-Level */
    int cmsg_type; /* Protokoll-Typ */
    /*.....Daten der Kontrollmessage.....*/
}
```

Zum Schicken eines Filedesktors wird dabei `cmsg_len` auf `sizeof(struct cmsghdr) + sizeof(int)` gesetzt. Die Addition von `sizeof(int)` ist notwendig für den Filedesktor. `cmsg_level` wird auf `SOL_SOCKET` und `cmsg_type` auf `SCM_RIGHTS` gesetzt (SCM = *Socket-level Control Message*). Der entsprechende Filedesktor wird unmittelbar nach der `cmsg_type`-Komponente eingetragen. Zur Ermittlung dieser Adresse wird das Makro `MSG_DATA` verwendet.

Um einen Filedesktor zu empfangen (**empfang_fd**), wird Speicherplatz allokiert, der groß genug ist, um die Struktur `cmsghdr` und einen Filedesktor aufzunehmen. Vor dem Aufruf der Funktion **recvmsg** zum Empfangen des entsprechenden Filedesktors wird die Adresse dieses allokierten Speicherplatzes der Komponente `msg_control` zugewiesen.

Programm 19.6 (bsd4_4.c) zeigt eine mögliche Implementierung der Funktionen **send_fd**, **send_fehl** und **empfang_fd** unter neueren BSD-Systemen.

```
#include <sys/types.h>
#include <sys/socket.h> /* struct msghdr */
#include <sys/uio.h> /* struct iovec */
#include <errno.h>
#include <stddef.h>
#include "eighdr.h"

/* Groesse des Kontrollpuffers zum Senden/Empfangen eines Filedeskr. */
#define KONTROLLAENGE (sizeof(struct cmsghdr) + sizeof(int))

static struct cmsghdr *cmzgr = NULL; /* beim erstmal malloc hierfuer */

/*----- send_fd -----
 *      sendet einen Filedesktor an anderen Prozess
 *      wenn fd<0, so wird -fd als Fehler-Status geschickt
 */
int send_fd(int spipecd , int fd)
{
    struct iovec    iov[1];
    struct msghdr   message;
    char            protokoll[2] = { 0, 0 };

    iov[0].iov_base = protokoll;
    iov[0].iov_len  = 2;
    message.msg_iov  = iov;
    message.msg_iovlen = 1;
    message.msg_name  = NULL;
```



```

message.msg_namelen = 0;

if (fd < 0) {
    message.msg_control = NULL;
    message.msg_controllen = 0;
    protokoll[1] = -fd; /* Status != 0 bedeutet Fehler */
    if (protokoll[1] == 0)
        protokoll[1] = 1; /* Abfangen von Ueberlaufen */
} else {
    if (cmzgr == NULL && (cmzgr = malloc(KONTROLLAENGE)) == NULL)
        return(-1);
    cmzgr->cmsg_level = SOL_SOCKET;
    cmzgr->cmsg_type = SCM_RIGHTS;
    cmzgr->cmsg_len = KONTROLLAENGE;
    message.msg_control = (caddr_t) cmzgr;
    message.msg_controllen = KONTROLLAENGE;
    *(int *)CMSG_DAT(cmzgr) = fd; /* zu schickender Filedeskriptor */
}

if (sendmsg(spipedfd, &message, 0) != 2)
    return(-1);

return(0);
}

/*----- empfang_fd -----*/
*   empfaengt einen Filedeskriptor von einem anderen Prozess.
*   Zusaetzlich empfangene Daten werden von
*   (*benutzerfunk)(STDERR_FILENO, puffer, byte_gelesen)
*   verarbeitet. */
int empfang_fd(int spipedfd,
               ssize_t (*benutzerfunk)(int, const void *, size_t))
{
    int          neuafd, byte_gelesen, status=-1;
    char         *zgr, puffer[MAX_ZEICHEN];
    struct iovec  iov[1];
    struct msghdr msghdr;

    while (1) {
        iov[0].iov_base = puffer;
        iov[0].iov_len = sizeof(puffer);
        message.msg_iov = iov;
        message.msg_iovlen = 1;
        message.msg_name = NULL;
        message.msg_namelen = 0;
        if (cmzgr == NULL && (cmzgr = malloc(KONTROLLAENGE)) == NULL)
            return(-1);
        message.msg_control = (caddr_t) cmzgr;
        message.msg_controllen = KONTROLLAENGE;

        if ( (byte_gelesen = recvmg(spipedfd, &message, 0)) < 0)
            fehler_meld(FATAL_SYS, "recvmg-Fehler");
        else if (byte_gelesen == 0) {

```

```

    fehler_meld(WARNUNG, "Verbindung abgebrochen (durch Server)");
    return(-1);
}

/* Durchlaufen des ganzen Puffers, wobei die eigentlichen Daten
   mit einem 0-Byte abgeschlossen sind, dem dann der
   Status folgt. Ein Statuswert von 0 bedeutet dabei, dass
   ein Filedeskriptor empfangen wird. */
for (zgr=puffer; zgr < &puffer[byte_gelesen]; ) {
    if (*zgr++ == 0) {
        if (zgr != &puffer[byte_gelesen-1])
            fehler_meld(DUMP, "message inkonsistent");
        status = *zgr & 0xff;
        if (status == 0) {
            if (message.msg_controllen != KONTROLLAENGE)
                fehler_meld(DUMP, "message inkonsistent");
            neuofd = *(int *)CMSG_DAT(cmzgr);
        } else
            neuofd = -status;
        byte_gelesen -= 2;
    }
}
if (byte_gelesen > 0)
    if ( (*benutzerfunk)(STDERR_FILENO, puffer, byte_gelesen)
        != byte_gelesen)
        return(-1);
if (status >= 0)
    return(neuofd);
}
}

/*----- send_fehl -----
*      sendet mittels dem beschriebenen Protokoll einen Fehler.
*      Diese Routine wird benutzt, wenn beabsichtigt war, einen
*      Filedeskriptor zu schicken, aber ein Fehler aufgetreten ist.*/
int send_fehl(int spipefd, int status, const char *fehlmeld)
{
    int    n;

    if ( (n=strlen(fehlmeld)) > 0)
        if (writespz(spipefd, fehlmeld, n) != n) /*Senden der Fehlermeldung */
            return(-1);

    if (status >= 0)
        status = -1; /* Status muss negativ sein */

    if (send_fd(spipefd, status) < 0)
        return(-1);

    return(0);
}

```

Programm 19.6 (**bsd4_4.c**): Die Funktionen **send_fd**, **send_fehl** und **empfang_fd** für neuere BSD-Systeme

19.4 Client-Server-Realisierung mit verwandten Prozessen

Hier wird ein Server entwickelt, der für das Öffnen von Dateien zuständig ist. Die Clients starten diese Server mit einem **fork** und einem anschließenden **exec**-Aufruf. Der Server öffnet dann die entsprechende Datei und schreibt den zugehörigen Filedescriptor in eine Stream Pipe, aus der ihn der Client liest.

Die vom Server zu öffnenden Dateien müssen dabei nicht unbedingt reguläre Dateien sein, sondern können auch Netzwerk- oder Modemverbindungen sein. Bei dieser Form der IPC wird auch nur ein Minimum an Information zwischen einem Client (schickt Dateiname und Öffnungsmodus) und dem Server (schickt entsprechenden Filedescriptor zurück) ausgetauscht. Ein Schicken des ganzen Dateiinhalts durch den Server wird bei dieser Methode vermieden, da dies zu einem nicht unerheblichen Datenverkehr in der Stream Pipe führen würde.

19.4.1 Client

Das Programm 19.7 (`opencli.c`) zeigt die Client-Realisierung. Der Clientprozeß kreiert dabei eine Stream Pipe und ruft dann den Server mit einem **fork** und **exec** auf. Danach sendet er seine Anforderung über die Stream Pipe an den Server und wartet auf die Antwort des Servers. Zur Kommunikation zwischen Client und Server wird das folgende Protokoll verwendet.

1. Die Client-Anforderung an den Server hat die folgende Form:

```
open  dateiname  modus\0
```

Der `modus` ist ein ganzzahliger Wert, der dem Öffnungsmodus bei der Funktion **open** (2. Argument) entspricht. Der Anforderungsstring ist immer mit einem 0-Byte abgeschlossen.

2. Die Server-Antwort ist entweder ein

- Filedescriptor (mit **send_fd** geschickt) oder eine
- Fehlermeldung (mit **send_fehl** geschickt).

Im Programm 19.7 (`opencli.c`) besteht die `main`-Funktion aus einer Schleife, die einen Dateinamen von der Standardeingabe liest. Zum Öffnen der Datei dieses Namens wird die Funktion **server_open** aufgerufen, die den vom Server gelieferten Filedescriptor zurückgibt. Unter Benutzung dieses Filedescriptors gibt das Programm 19.7 (`opencli.c`) den Inhalt und die Byteanzahl der betreffenden Datei auf der Standardausgabe aus.

```
#include <sys/types.h>
#include <sys/uio.h> /* struct iovec */
#include <fcntl.h>
#include <errno.h>
#include "eighdr.h"
```

```

#define PUFF_GROESSE 8192

static int server_open(char *name, int openflag);
/*----- main -----
*/
int
main(int argc, char *argv[])
{
    int    n, fd;
    long   zeichzahl;
    char   puffer[PUFF_GROESSE], zeile[MAX_ZEICHEN];

    while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
        zeile[strlen(zeile) - 1] = '\0'; /* \n mit \0 ueberschreiben */
        zeichzahl = 0;
        if ( (fd = server_open(zeile, O_RDONLY)) >= 0) {
            while ( (n = read(fd, puffer, PUFF_GROESSE)) > 0) {
                zeichzahl += n;
                if (write(STDOUT_FILENO, puffer, n) != n)
                    fehler_meld(FATAL_SYS, "write-Fehler");
            }
            if (n < 0)
                fehler_meld(FATAL_SYS, "read-Fehler");
            fprintf(stderr, "---- %s: %ld Zeichen ---\n", zeile, zeichzahl);
            close(fd);
        }
    }
    exit(0);
}

/*----- server_open -----
*          sendet den Dateinamen und open-Flags an den entspr.
*          open-Server und empfaengt dann den Filesdeskriptor
*          fuer die von diesem Server geoeffnete Datei          */
static int server_open(char *name, int openflag)
{
    pid_t    pid;
    char      puffer[10];
    struct iovec  iov[3];
    static int  fd[2] = {-1, -1};

    if (fd[0] < 0) {
        if (stream_pipe(fd) < 0)
            fehler_meld(FATAL_SYS, "stream_pipe-Fehler");
        if ( (pid = fork()) < 0)
            fehler_meld(FATAL_SYS, "fork-Fehler");
        else if (pid == 0) { /*----- Kindprozess -----*/
            close(fd[0]);
            if (fd[1] != STDIN_FILENO)
                if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                    fehler_meld(FATAL_SYS, "dup2-Fehler (stdin)");
            if (fd[1] != STDOUT_FILENO)
                if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)

```

```

        fehler_meld(FATAL_SYS, "dup2-Fehler (stdout)");
        if (exec1("./openser", "openser", NULL) < 0)
            fehler_meld(FATAL_SYS, "exec1-Fehler");
    }
    close(fd[1]); /*----- Elternprozess -----*/
}
sprintf(puffer, " %d", openflag);
iov[0].iov_base = "open ";
iov[0].iov_len = strlen("open ");
iov[1].iov_base = name;
iov[1].iov_len = strlen(name);
iov[2].iov_base = puffer;
iov[2].iov_len = strlen(puffer)+1; /* +1 wegen abschl. \0 */
if (writev(fd[0], &iov[0], 3) !=
    iov[0].iov_len + iov[1].iov_len + iov[2].iov_len)
    fehler_meld(FATAL_SYS, "writev-Fehler");

return( empfang_fd(fd[0], write) );
}

```

Programm 19.7 (**opencli.c**): Client, der zum Öffnen einer Datei einen Server benutzt

Die Funktion **server_open** startet nach dem Kreieren einer Stream Pipe den Server (**openser**) mit einem **fork** und **exec1**. Der Kindprozeß schließt dabei die eine Seite, und der Elternprozeß die andere Seite der Pipe.

Der Kindprozeß dupliziert mit **dup2**-Aufrufe seine Pipe-Seite auf die Standardein- und Standardausgabe, bevor er sich mit **exec1** mit dem Serverprozeß (**openser**) überlagert.

Der Elternprozeß (Client) schickt dann mit einem **writev** seine Anforderung (*open datei-name open-modus*) über die Stream Pipe an den Serverprozeß. Anschließend wartet der Elternprozeß mit **empfang_fd** auf die Antwort des Serverprozesses. Falls der Server eine Fehlermeldung schickt, so wird diese mit **write** auf der Standardfehlerausgabe ausgegeben.

19.4.2 Server

Das Programm 19.8 (**openser.c**) zeigt die Server-Realisierung. Dieser Serverprozeß, der vom Client mit einem **exec1**-Aufruf gestartet wird, liest in seiner **main**-Funktion die Client-Anforderungen aus der Stream Pipe (seine Standardeingabe) und ruft zur Bearbeitung dieser Anforderung die Funktion **anforderung** auf.

Die Funktion **anforderung** überprüft zunächst, ob die Anforderung dem vereinbarten Protokoll entspricht. Dazu ruft sie unter anderem die Funktion **puffer_argv** auf, die den vom Client gelieferten String in einzelne Wörter aufteilt, welche sie in dem übergebenen String-Array **argv** hinterlegt. Die Anzahl der Wörter schreibt sie dabei in das Argument **argc**.

Wenn die geschickte Anforderung dem vereinbarten Protokoll entspricht, dann öffnet die Funktion `anforderung` die entsprechende Datei und schickt den zugehörigen Filedeskriptor mittels eines `send_fd`-Aufrufs an den Client zurück. Bei einem Fehler wird mit `send_fehl` dem Client eine Fehlermeldung geschickt.

```
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include "eighdr.h"

#define MAX_ARGC 100

static void anforderung(char *puffer, int byte_gelesen, int fd);
static int puffer_argv(char *puffer, int *argc, char *argv[]);

char fehl_meldung[MAX_ZEICHEN];

/*----- main -----*/
int
main(void)
{
    int byte_gelesen;
    char puffer[MAX_ZEICHEN];

    /* Lesen der vom Client geschriebenen Argumente (Anforderung) */
    while ( (byte_gelesen = read(STDIN_FILENO, puffer, MAX_ZEICHEN)) != 0 ) {
        if (byte_gelesen < 0)
            fehler_meld(FATAL_SYS, "Fehler beim Lesen aus Stream Pipe");

        anforderung(puffer, byte_gelesen, STDIN_FILENO);
    }
    exit(0);
}

/*----- anforderung -----*/
static void anforderung(char *puffer, int byte_gelesen, int fd)
{
    char *argv[MAX_ARGC];
    int argc, neu fd;

    if (puffer[byte_gelesen-1] != '\0') {
        sprintf(fehl_meldung, "Anforderung ohne abschl. \0: %*.s\n",
                byte_gelesen, byte_gelesen, puffer);
        send_fehl(STDOUT_FILENO, -1, fehl_meldung);
        return;
    }
    if (puffer_argv(puffer, &argc, argv) < 0) {
        send_fehl(STDOUT_FILENO, -1, fehl_meldung);
        return;
    }
    if ( (neu fd = open(argv[1], atoi(argv[2]))) < 0 ) {
        sprintf(fehl_meldung, "kann %s nicht oeffnen: %s\n",
                argv[1], strerror(errno));
    }
}
```

```

        send_fehl(STDOUT_FILENO, -1, fehl_meldung);
        return;
    }
    if (send_fd(STDOUT_FILENO, neufd) < 0)
        fehler_meld(FATAL_SYS, "send_fd-Fehler");
    close(neufd);
}

/*----- puffer_argv -----
 *      legt die im 'puffer' enthaltenen Argumente im Array argv ab.
 *      Die Anzahl der Argumente wird dabei in 'argc' abgelegt.      */
static int puffer_argv(char *puffer, int *argc, char *argv[])
{
    char    *zgr;

    if (strtok(puffer, " \\t\\n") == NULL)
        return(-1);

    argv[*argc=0] = puffer;
    while ( (zgr = strtok(NULL, " \\t\\n")) != NULL) {
        if (++*argc >= MAX_ARGC-1)
            return(-1);
        argv[*argc] = zgr;
    }
    argv[++*argc] = NULL;

    if (*argc != 3 || strcmp(argv[0], "open")) {
        strcpy(fehl_meldung, "Falsches Protokoll (erwartet: open name flag)\\n");
        return(-1);
    }
    return(0);
}

```

Programm 19.8 (openser.c): Server, der vom Client geschickte Dateinamen öffnet

Nachdem man die beiden Programme 19.7 (opencli.c) und 19.8 (openser.c) kompiliert und gelinkt hat

```

cc -o opencli opencli.c readwrit.c svr4.c spivesv.c fehler.c (SVR4)
cc -o openser openser.c readwrit.c svr4.c fehler.c (SVR4)
cc -o opencli opencli.c readwrit.c bsd4_3.c spipebsd.c fehler.c -lsocket -lnsl (4.3BSD)
cc -o openser openser.c readwrit.c bsd4_3.c fehler.c -lsocket -lnsl (4.3BSD)

cc -o opencli opencli.c readwrit.c bsd4_4.c spipebsd.c fehler.c -lsocket -lnsl (neues BSD/
Linux2)
cc -o openser openser.c readwrit.c bsd4_4.c fehler.c -lsocket -lnsl (neues BSD/
Linux3)

```

2. Unter Linux muß -lsocket weggelassen werden.

3. Unter Linux muß -lsocket weggelassen werden.

muß man `openser` im Hintergrund starten. Startet man dann `opencli` im Vordergrund, so kann man Dateinamen interaktiv eingeben. Zu diesen Dateien wird dann deren Inhalt und die Anzahl der Bytes in dieser Datei ausgegeben.

Hinweis

Diese Methode, den Server als eigenes ausführbares Programm zu realisieren, hat einige Vorteile:

1. Wenn der Server geändert werden muß, so ist von dieser Änderung nur ein Programm betroffen. Wenn man dagegen eine solche Aufgabenstellung mit eigens dafür entwickelten Bibliotheksfunktionen löst, so müssen bei Änderungen alle Programme, die diese Funktionen verwenden, neu gelinkt werden.
2. Jeder Client kann leicht die vom entsprechenden Server angebotenen Dienste in Anspruch nehmen. Da diese entsprechenden Dienste nicht in jedem Programm eigens realisiert sind, was eine allgemeine Benutzung durch andere Programme unmöglich macht, sondern eben in einem eigenen Programm der Allgemeinheit zur Verwendung angeboten werden, erfüllt diese Methode die Forderungen nach *Wiederverwendbarkeit* und *Modularität* von Software.
3. Der Server kann ein Set-User-ID-Programm sein, das bevorzugte Zugriffsrechte besitzt, die der Client nicht hat. Bei Verwendung von Bibliotheksfunktionen besteht diese Möglichkeit nicht.
4. Der Server übernimmt alle die ihm zugeteilten Aufgaben, die dem Client verborgen bleiben. So wird z.B. ein Server, der für das Öffnen von Dateien jeglicher Art zuständig ist, alle anfallenden Arbeiten (wie z.B. Übersetzen eines Netzwerknamens in eine Netzwerkadresse, Anwählen eines Modems, Einrichten von Dateisperren usw.) übernehmen und dem anfordernden Client nur den entsprechenden geöffneten Filedeskriptor zurückgeben. Der Client kann dann einfach unter Verwendung dieses Filedeskriptors und der E/A-Funktionen auf die entsprechende Datei (normale Datei, Gerät, Netzwerkverbindung usw.) zugreifen, ohne daß er sich mit den oft mühsamen Öffnungsarbeiten herumschlagen muß.

19.5 Benannte Stream Pipes

Während Stream Pipes nur zum Datenaustausch zwischen verwandten Prozessen (wie Eltern- und Kindprozeß) verwendet werden können, können die in diesem Kapitel vorgestellten *benannten Stream Pipes* auch zum Datenaustausch zwischen Prozessen verwendet werden, die in keinem Verwandtschaftsverhältnis stehen.

Um eine *benannte Stream Pipe* einzurichten, muß mit einem **`stream_pipe`**-Aufruf eine unbenannte Stream Pipe eingerichtet werden, bevor einer der beiden Seiten dieser Stream Pipe ein Dateiname zugeteilt wird. Ein Server, der als Dämonprozeß abläuft, würde z.B. nur eine Seite einer Stream Pipe kreieren und dieser Seite dann einen Namen zuteilen.

Clients könnten dann mit diesem Server kommunizieren, indem sie ihre Daten an diese benannte Seite der Stream Pipe schicken.

Eine noch bessere Methode ist die folgende Vorgehensweise: Der Server kreiert eine Stream Pipe, deren einer Seite er einen Namen zuordnet, und Clients, die Anforderungen schicken möchten, stellen eine Verbindung zu dieser benannten Seite her. Bei jeder dieser Verbindungsanforderungen durch einen Client kreiert der Server eine neue Stream Pipe zur privaten Kommunikation mit diesem speziellen Client. So wird der Server immer darüber informiert, wenn ein Client eine Verbindung anfordert oder aber diese wieder aufhebt. Sowohl SVR4 als auch BSD-Unix unterstützen diese Form der Interprozeßkommunikation.

19.5.1 **serv_initverbind, serv_bereit und cli_verbind** -Eigene Funktionen für Client-Server-Verbindungen

Hier werden drei Funktionen beschrieben, die die Verbindungen zwischen einem Server und einem Client über Stream Pipes herstellen. Die Funktionen wurden vom Buch »*Advanced Programming in the UNIX Environment*, W. Richard Stevens« in etwas abgeänderter Form übernommen.

```
#include "eighdr.h"

int serv_initverbind(const char *name);
    gibt zurück: Filedeskriptor der benannten Pipe, an der Clients Verbindung anfordern (bei Erfolg);
    < 0 bei Fehler

int serv_bereit(int initfd, uid_t *uidzgr);
    gibt zurück: neuer Filedeskriptor (bei Erfolg); < 0 bei Fehler

int cli_verbind(const char *name);
    gibt zurück: Filedeskriptor (bei Erfolg); < 0 bei Fehler
```

serv_initverbind

Diese Funktion, die der Server zu Beginn aufruft, richtet eine Stream Pipe ein und teilt einer Seite dieser Stream Pipe einen Namen (im Dateisystem) zu. Clients, die eine Verbindung zum Server herstellen wollen, rufen die Funktion **cli_verbind** mit diesem Namen auf. Der Rückgabewert dieser Funktion ist der Filedeskriptor für die Server-Seite der benannten Stream Pipe.

serv_bereit

Nachdem ein Server **serv_initverbind** aufgerufen hat, ruft er **serv_bereit** auf, um auf Verbindungsanforderungen von Clients zu warten. Das Argument *initfd* ist dabei der von **serv_initverbind** zurückgegebene Filedeskriptor. Die Funktion **serv_bereit** kehrt immer erst dann zurück, wenn ein Client eine Verbindungsanforderung schickt. In diesem Fall wird eine neue eigene Stream Pipe für die Kommunikation mit diesem Client eingerichtet

und der Filedeskriptor für diese Stream Pipe wird als Rückgabewert geliefert, wobei zuvor die effektive User-ID des Clients nach `*uidzgr` geschrieben wird.

cli_verbind

Jeder Client, der eine Verbindung zum Server wünscht, ruft ***cli_verbind*** unter Angabe des mit dem Server vereinbarten Namens (der benannten Stream Pipe) auf. Der von ***cli_verbind*** zurückgegebene Filedeskriptor bezeichnet dabei die Stream Pipe, die zur privaten Kommunikation mit dem Server eingerichtet wurde.

Mit diesen drei Funktionen ist es möglich, einen Server-Dämonprozeß zu entwickeln, der eine beliebige Anzahl von Clients bedienen kann. Die einzige Einschränkung für die Anzahl von Clients ist dabei die maximale Anzahl von Filedeskriptoren, die am jeweiligen System gleichzeitig geöffnet sein dürfen.

Da diese drei Funktionen mit normalen Filedeskriptoren arbeiten, kann der Server unter Verwendung der E/A-Multiplex-Funktionen ***select*** oder ***poll*** die einzelnen Clients bedienen.

Nachfolgend werden mögliche Realisierungen der obigen drei Funktionen in SVR4 und BSD-Unix gezeigt.

19.5.2 serv_initverbind, serv_bereit und cli_verbind – Realisierung in SVR4

In SVR4 empfiehlt sich die folgende Vorgehensweise. Zuerst richtet der Server eine normale Stream Pipe ein und trägt den in SVR4 vorhandenen Steuermodul `connld` an der einen Seite der Stream Pipe ein. Abbildung 19.6 veranschaulicht die daraus resultierende Konstellation.

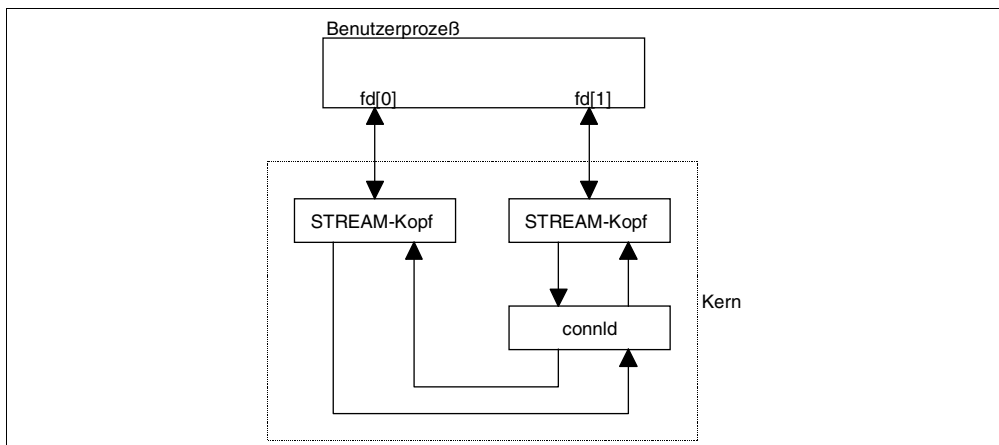


Abbildung 19.6: Stream Pipe in SVR4 nach Eintragung des Moduls `connld`

Nach dieser Eintragung des Steuermoduls `connld` ordnet man mit der in SVR4 angebotenen Funktion ***fattach*** dieser Stream Pipe einen Namen zu.

```
int fattach(int fd, const char *pfadname);
```

```
int fdetach(const char *pfadname);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Neben ***fattach*** bietet SVR4 die Funktion ***fdetach*** an, mit der ein mit ***fattach*** einer Stream Pipe zugeordneter Name wieder von dieser »gelöst« werden kann.

Nachdem mit ***fattach*** der Stream Pipe ein Name zugeordnet wurde, wird bei jedem anschließenden Öffnen dieses Pfadnamens mit ***open*** die benannte Seite der Stream Pipe angesprochen.

Wenn ein anderer Prozeß mit ***open*** die benannte Seite dieser Stream Pipe (Seite mit dem Modul `connld`) öffnet, so geschieht folgendes:

1. Eine neue Pipe wird eingerichtet.
2. Ein Filedeskriptor dieser neuen Pipe wird dem Aufrufer von ***open*** (Client) als Rückgabewert von ***open*** geliefert.
3. Der andere Deskriptor wird an den Server auf der anderen Seite der benannten Stream Pipe (nicht der `connld`-Seite) weitergeleitet. Der Server kann diesen neuen Deskriptor mit einem ***ioctl***-Aufruf erfragen, wenn er dabei als zweites Argument `I_RECVFD` angibt.

Abbildung 19.7 zeigt die Client-Server-Konstellation, nachdem der Server mit ***fattach*** seiner Stream Pipe den Namen `/tmp/opensd` zugeordnet hat und der Client seinerseits

```
fd = open ("/tmp/opensd", O_RDWR);
```

aufgerufen hat. Dieses ***open*** des Clients bewirkt, daß zwischen Client und Server eine neue Pipe eingerichtet wird, da der mit ***open*** geöffnete Dateiname ein benannter STREAM mit dem `connld`-Modul ist. ***open*** liefert dabei den Deskriptor für die eine Pipe-Seite als Rückgabewert an den Client (`fd`)

Den Deskriptor (`client_fd`) für die andere Seite dieser Pipe, die sich im »Server« befindet, kann der Server aus der Stream Pipe `fd[0]` mit einem entsprechenden ***ioctl***-Aufruf (2. Argument `I_RECVFD`) erfragen. Nachdem der Server den Modul `connld` in `fd[1]` eingetragen hat und mit ***fattach*** `fd[0]` einen Namen zugeordnet hat, verwendet er `fd[1]` nicht wieder.

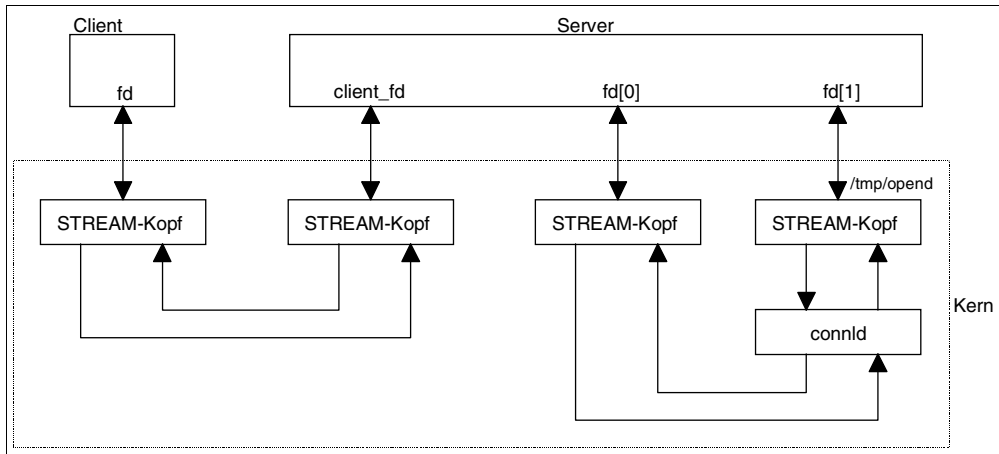


Abbildung 19.7: Client-Server-Verbindung über eine benannte Stream-Pipe in SVR4

Nachdem ein Server ***serv_initverbind*** aufgerufen hat, ruft er ***serv_bereit*** auf, um auf Verbindungsanforderungen von Clients zu warten. In Abbildung 19.7 wäre z.B. das erste Argument für ***serv_bereit*** der Deskriptor `fd[0]` und der Rückgabewert von ***serv_bereit*** wäre `client_fd`.

Jeder Client, der eine Verbindung zum Server wünscht, ruft ***cli_verbind*** auf und erhält als Rückgabewert `fd` in Abbildung 19.7.

Programm 19.9 (`svr4_cs.c`) zeigt eine mögliche Realisierung der drei Funktionen ***serv_initverbind***, ***serv_bereit*** und ***client_verbind***.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include "eighdr.h"

/*----- serv_initverbind -----*/
int serv_initverbind(const char *name)
{
    int fd[2], hilf_fd;

    unlink(name);
    if ( (hilf_fd =
        creat(name, S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)) < 0)
        return(-1);
    if (close(hilf_fd) < 0)
        return(-2);
    if (pipe(fd) < 0)
        return(-3);

    /*--- Modul 'connld' in STREAM eintragen und fattach auf fd[1] ---*/
    if (ioctl(fd[1], I_PUSH, "connld") < 0)
```

```

        return(-4);
    if (fattach(fd[1], name) < 0)
        return(-5);

    return(fd[0]); /* Client-Verbindungsanforderung kommt ueber fd[0] */
}

/*----- serv_bereit -----*/
int serv_bereit(int initfd, uid_t *uidzgr)
{
    struct strrecvfd  empfang;

    if (ioctl(initfd, I_RECVFD, &empfang) < 0)
        return(-1);
    if (uidzgr != NULL)
        *uidzgr = empfang.uid;

    return(empfang.fd);
}

/*----- cli_verbind -----*/
int cli_verbind(const char *name)
{
    int  fd;

    if ( (fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0)
        return(-2);

    return(fd);
}

```

Programm 19.9 (**svr4_cs.c**): Realisierung von *serv_initverbind*, *serv_bereit* und *cli_verbind* in SVR4

19.5.3 *serv_initverbind*, *serv_bereit* und *cli_verbind* – Realisierung in BSD-Unix und SVR4

Unter BSD-Unix werden *Unix Domain Sockets* verwendet, um eine Verbindung zwischen dem Client und dem Server herzustellen. Hier wird zunächst eine kurze Einführung in das Berkeley Socket API gegeben, das unter BSD-Unix entwickelt wurde und sich als Standard-API etabliert hat, weswegen auch SVR4 und Linux das Berkeley Socket API anbieten. Erst werden die Grundlagen der Sockket-Programmierung und die zugehörigen Funktionen kurz vorgestellt, bevor dann eine Socket-Realisierung der in Kapitel 19.5.1 beschriebenen Funktionen ***serv_initverbind***, ***serv_bereit*** und ***cli_verbind*** gegeben wird.

Grundlagen der Socket-Programmierung

Das Berkeley Socket API wurde als abstrakter Vermittler zwischen verschiedenen Netzwerkprotokollen entworfen, was die Schnittstelle zwar verkompliziert, aber den Vorteil hat, daß jederzeit neue Protokolle ohne Änderung der Schnittstelle hinzugefügt werden können.

Das wichtigste Protokoll ist TCP/IP. Neben der Verwendung von Sockets zur Netzwerkprogrammierung, worauf in Kapitel 19.7 noch näher eingegangen wird, werden Sockets aber auch von vielen Anwendungen zur Interprozeßkommunikation (IPC) auf einem lokalen Rechner benutzt. Diese Benutzung von Sockets zur IPC wird hier näher erläutert. Zunächst sollen jedoch einige Begriffe aus der Netzwerkprogrammierung vorgestellt werden, die in diesem Zusammenhang benötigt werden.

Verbindungsorientierte und verbindungslose Protokolle

Bei verbindungsorientierten Protokollen (*connection-oriented*) wird zuerst – ähnlich wie beim Telefon – eine Verbindung zwischen zwei Endpunkten aufgebaut, bevor eine Kommunikation stattfindet. Andere Benutzer haben keine Möglichkeit, sich in eine so eingerichtete Verbindung zwischen zwei Teilnehmern hineinzudrängen.

Protokolle, die ohne eine solche Verbindung zwischen zwei Endpunkten arbeiten, nennt man verbindungslose Protokolle (*connection-less*).

Sequencing

Protokolle, die sicherstellen, daß die Daten in der gleichen Reihenfolge empfangen werden, in der sie gesendet werden, bieten das sogenannte *Sequencing* an.

Streaming-Protokolle und paketbasierte Protokolle

Streaming-Protokolle arbeiten mit einzelnen Bytes, wobei größere Bytefolgen in Blöcken zusammengefaßt werden können. Paketbasierte Protokolle dagegen erlauben nur das Versenden und Empfangen von ganzen Datenpaketen. In den meisten Fällen ist eine Maximalgröße für die Pakete festgelegt.

Fehlerkontrolle (error control)

Hierzu zählt man Protokolle, die Daten, welche während der Übertragung beschädigt wurden, automatisch verwerfen und erneut anfordern können.

Die einzelnen hier aufgezählten Eigenschaften sind voneinander unabhängig. Von allen denkbar möglichen Kombinationen der obigen Eigenschaften haben sich zwei Protokollarten durchgesetzt, die hauptsächlich von Anwendungen benutzt werden:

Datagram-Protokolle

Diese Protokolle sind paketorientiert und bieten weder Sequencing noch Fehlerkontrolle. Ein oft benutztes Datagram-Protokoll ist UDP, was zur TCP/IP-Protokollfamilie gehört. Auf UDP baut z.B. das NFS-Protokoll auf.

Stream-Protokolle

Stream-Protokolle sind Streaming-Protokolle mit Sequencing und Fehlerkontrolle, wie z.B. das TCP-Protokoll.

Hier wird auf Stream-Protokolle näher eingegangen, da sie für die meisten Anwendungen einfacher zu benutzen sind. Mehr Informationen zu den einzelnen Protokollen finden sich in der entsprechenden Fachliteratur, wobei hier besonders »*TCP/IP Illustrated, Volume I/II*« von Gary R. Wright und W. Richard Stevens hervorzuheben ist.

Nach dieser Klärung der wichtigsten grundlegenden Begriffe aus der Netzwerkprogrammierung werden nun die Grundlagen der Socket-Programmierung und die dazugehörigen Funktionen kurz vorgestellt.

Sockets sind mit Hilfe des Filesystems implementiert und werden mit der Funktion **socket** angelegt.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int typ, int protokoll);
```

gibt zurück: Filedeskriptor (bei Erfolg); -1 bei Fehler

domain
legt die zu benutzende Protokollfamilie fest (siehe dazu Tabelle 19.1).

Adresse	Parameter domain	Protokollart
AF_UNIX	PF_UNIX	Unix Domain
AF_INET	PF_INET	TCP / IP
AF_AX25	PF_AX25	AX.25 (Amateurradio)
AF_IPX	PF_IPX	Novell IPX
AF_APPLETALK	PF_APPLETALK	AppleTalk DDS
AF_NETROM	PF_NETROM	NetROM (Amateurradio)

Tabelle 19.1: Protokoll- und Adreßfamilien

typ
Hierfür kann man SOCK_STREAM für ein Streaming-Protokoll oder SOCK_DGRAM für ein Datagram-Protokoll angeben. Es sind zwar noch weitere Angaben möglich, aber diese sind nur für sehr spezifische Anwendungen von Interesse und können mit **man socket** nachgeschlagen werden.

protokoll
Dieser Parameter wählt das zu benutzende Protokoll aus der mit den ersten beiden Parametern festgelegten Protokollfamilie aus. Üblicherweise gibt man hier 0 an und läßt den Systemkern das Standardprotokoll für die entsprechende Protokollfamilie auswählen. Für PF_INET ist das ICP das Standard-Stream-Protokoll und UDP das Standard-Datagram-Protokoll. Weitere Protokollnummern können bei Bedarf in / etc/protocols nachgeschlagen werden.

Ein mit **socket** kreierter Socket ist nicht initialisiert. Für den Socket wird bei seiner Erzeugung lediglich ein bestimmtes Protokoll festgelegt, er wird aber noch nicht mit einer Ressource verbunden, so daß ein lesender oder schreibender Zugriff auf ihn noch nicht möglich ist.

Es ist nun die Aufgabe einer Seite, üblicherweise des Server-Prozesses, eine Verbindung vorzubereiten und darauf zu warten, daß irgend jemand sich mit ihm verbindet. Client-Prozesse erzeugen dagegen einen Socket, teilen dem System die gewünschte Adresse mit und versuchen dann eine Verbindung aufzubauen. Die Verbindung ist hergestellt, wenn der Server, der auf einen Client warten, den Verbindungsversuch akzeptiert. Danach können der Server- und Client-Prozeß über den Socket miteinander kommunizieren. Ist ein Socket richtig initialisiert, kann auf ihn wie auf jeden anderen Filedescriptor mit den elementaren E/A-Funktionen, wie z.B. **read** oder **write**, zugegriffen werden.

Die Reihenfolge, in der Server- und Clientprozeß die entsprechenden Funktionen aufrufen müssen, um eine Verbindung herzustellen, ist in Abbildung 19.8 veranschaulicht.

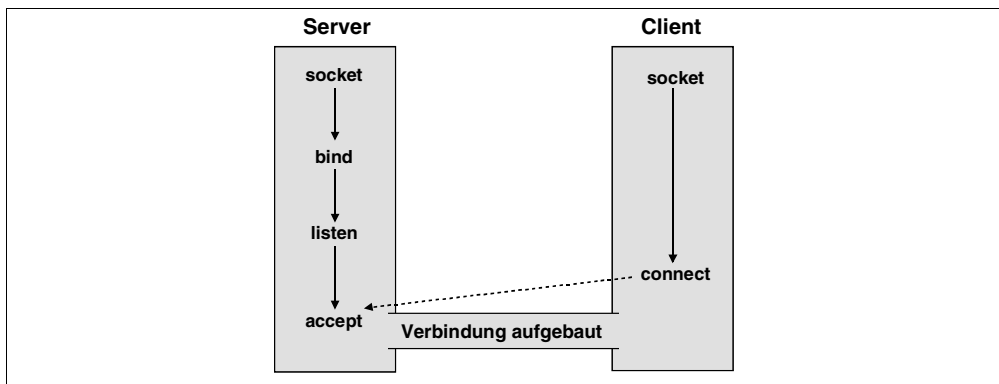


Abbildung 19.8: Schrittfolge zur Herstellung einer Socket-Verbindung

Nachfolgend werden diese Funktionen näher vorgestellt.

bind – Verknüpfen eines Sockets mit einer Adresse (Server)

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *adr, int adrlaenge);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

`sockfd` gibt den Filedescriptor des zu bindenden Sockets an, die letzten beiden Parameter spezifizieren die Adresse.

Die Struktur `sockaddr`, die als Grundform für jede Protokollfamilie verwendet werden kann, ist in der Headerdatei `<sys/socket.h>` bzw. `<linux/socket.h>` wie folgt definiert:

```
struct sockaddr
{
    unsigned short sa_family;    /* address family, AF_XXX    */
    ...
};
```



```
char          sa_data[14]; /* 14 bytes of protocol address */
};
```

listen / accept – Warten auf Verbindungen (Server)

Nachdem ein Socket mit **bind** durch einen Server-Prozeß an eine Adresse gebunden wurde, teilt der Server-Prozeß durch einen Aufruf der Funktion **listen** dem System mit, daß er bereit ist, mit anderen Prozessen über diesen Socket Verbindungen einzugehen. Bevor aber wirklich eine Verbindung für einen Socket, der mit **listen** vom Server-Prozeß abgehört wird, aufgebaut wird, muß der Server-Prozeß den Verbindungsversuch seitens des Clients mit dem Aufruf der Funktion **accept** akzeptieren. Ruft der Server-Prozeß **accept** vor einem Verbindungsversuch seitens des Clients auf, so blockiert normalerweise **accept** solange, bis der Client einen Verbindungswunsch äußert. Um eine solche Blockierung bei **accept** zu unterbinden, muß der Socket mit **fcntl** als nicht blockierend markiert werden. In diesem Fall kehrt **accept** sofort mit einer entsprechenden Fehlernummer zurück. Um festzustellen, ob ein Verbindungswunsch seitens des Clients ansteht, was man mit *pending* bezeichnet, kann die Funktion **select** verwendet werden. Die beiden Funktionen **listen** und **accept** sind in `<sys/socket.h>` bzw. `<linux/socket.h>` wie folgt deklariert:

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
gibt zurück: 0 (bei Erfolg); -1 bei Fehler

int accept(int sockfd, struct sockaddr *adr, int *adrllaenge);
gibt zurück: Filedeskriptor des akzeptierten Sockets (bei Erfolg); -1 bei Fehler
```

Beide Funktionen erwarten als ersten Parameter den Filedeskriptor des entsprechenden Sockets.

Der Parameter `backlog` legt die maximal erlaubte Anzahl von anstehenden (*pending*) Verbindungswünschen seitens des Clients fest. Wird dieses Maximum erreicht, so werden weitere Verbindungsversuche seitens des Clients abgelehnt, wobei dies dem Client mit dem Fehler `ECONNREFUSED` mitgeteilt wird. Da BSD die maximale Größe von `backlog` auf 5 festgelegt hat, sollten portable Programme diesen Wert nicht überschreiten.

Die Funktion **accept** macht aus einer anstehenden (*pending*) Verbindung eine wirkliche Verbindung, die auch einen neuen Filedeskriptor erhält. Dieser neue Filedeskriptor, der als Rückgabewert geliefert wird, erbt alle Attribute vom Socket, das zuvor mit **listen** abgehört wurde.

Die Parameter `adr` und `adrllaenge` geben die Adressen an, in die vom Systemkern die Adresse des Clients zu schreiben ist.

connect – Herstellen einer Verbindung zum Server (Client)

Auch ein Client könnte nach dem Erzeugen eines Sockets mit der Funktion **bind** diesem Socket eine Adresse zuweisen. Da diese lokale Adresse aber normalerweise für den Client nicht von Interesse ist, läßt er diesen Schritt oft aus und überläßt es dem Systemkern, irgendeine passende Adresse für den Socket zu finden.

In jedem Fall muß jedoch ein Client die Funktion **connect** aufrufen, um eine Verbindung zum Server herzustellen.

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

gibt zurück: 0 (bei Erfolg); -1 bei Fehler
```

Die Parameter beim **connect**-Aufruf spezifizieren den zu verbindenden Socket und die Zieladresse.

Hat ein Prozeß die Arbeit mit einem Socket beendet, sollte er diesen mit **close** schließen, um die damit verbundenen Ressourcen wieder freizugeben.

Nachdem hier die wichtigsten Funktionen für Sockets vorgestellt wurden, soll auf die einfachste Protokollfamilie eingegangen werden, die durch das Socket-API angeboten wird, nämlich die *Unix-Domain-Sockets*.

Unix-Domain-Sockets

Unix-Domain-Sockets sind keine Netzwerkprotokolle und können nur für Sockets auf dem lokalen Rechner verwendet werden. Trotzdem finden sie häufig Anwendung, da sie eine flexible Art der Interprozeßkommunikation (IPC) sind.

Die Adressen sind hierbei Dateien (Pfadnamen), die im Filesystem angelegt werden, wenn ein Socket an eine Datei gebunden wird.

Unix-Domain-Sockets bieten sowohl eine Stream- als auch eine Datagram-Schnittstelle an. Während die Datagram-Schnittstelle kaum benutzt wird, findet die Stream-Schnittstelle, die benannten Pipes ähnelt, doch häufiger Anwendung.

Die Unterschiede zwischen benannten Pipes und der Stream-Schnittstelle eines Unix-Domain-Sockets sind:

- *Benannte Pipes arbeiten verbindungslos*, was bedeutet, daß jeder Prozeß mit den entsprechenden Rechten eine in dieser Pipe stehende Nachricht lesen kann, ohne daß er vorher eine Verbindung zu dem Senderprozeß aufbauen muß.

- *Unix-Domain-Sockets sind verbindungsorientiert*, was bedeutet, daß immer zuerst eine Verbindung zwischen den beiden Prozessen, die miteinander kommunizieren möchten, aufzubauen ist. Nachrichten, die über diese private Verbindung ausgetauscht werden, können von keinem anderen Prozeß gelesen werden. Ein Server, der viele Verbindungen gleichzeitig verwalten kann, hat für jeden Kanal einen eigenen File-deskriptor.

Diese Unterschiede bringen es mit sich, daß Unix-Domain-Sockets besser für IPC geeignet sind als benannte Pipes und deswegen auch häufiger eingesetzt werden.

Bei Unix-Domain-Sockets sind die Adressen Dateinamen (Pfadnamen) im Filesystem. Existiert eine beim **bind**-Aufruf angegebene Datei noch nicht, wird sie als Socket-Datei mit den Zugriffsrechten 0666 neu angelegt. Sollte die Datei dagegen bereits existieren, beendet sich **bind** mit dem Fehlercode EADDRINUSE.

Unix-Domain-Adressen werden mit Hilfe der Struktur `sockaddr_un`, die in `<sys/un.h>` bzw. `<linux/un.h>` definiert ist, übergeben:

```
#define UNIX_PATH_MAX 108 /* Groesse ist versionsabhaengig */
struct sockaddr_un {
    unsigned short sun_family; /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

Die erste Komponente `sun_family` muß bei Unix-Domain-Sockets auf `AF_UNIX` gesetzt werden, und in der Komponente `sun_path` muß der Dateiname (Pfadname) eingetragen werden, der für die Verbindung benutzt werden soll.

Bei Parametern, die die Größe der Adresse bei den oben vorgestellten Funktionen festlegen, muß die Summe aus der Anzahl der Zeichen im Dateinamen (Pfadnamen) und der Größe der `sun_family`-Komponente angegeben werden.

Der in `sun_path` angegebene String muß zwar nicht unbedingt mit `\0` beendet sein, wird aber in den meisten Anwendungen doch mit `\0` terminiert.

Nachfolgend werden zur Demonstration von Unix-Domain-Socktes zwei einfache Programme vorgestellt, die über einen Socket miteinander kommunizieren.

Das Programm 19.10 (`sockserv.c`) ist dabei der Server, der eine Verbindung zu einem Unix-Domain-Socket (Datei `/tmp/socket`) annimmt und die dorthin geschriebenen Zeichen mit ihren Hexazahlen (entsprechend dem ASCII-Code) auf der Standardausgabe ausgibt. Diese Form eines Servers bezeichnet man mit *iterativer Server*, denn er kann zu einem bestimmten Zeitpunkt immer nur einen Client bedienen. Server, die abwechselnd mehrere Clients gleichzeitig bedienen können, werden *Concurrent-Server* genannt.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "eighdr.h"
```

```

int
main(void)
{
    int            sockfd, connfd, i, n;
    char           puffer[MAX_ZEICHEN];
    size_t         adrlaenge;
    struct sockaddr_un  adresse;

    if ( (sockfd = socket(PF_UNIX, SOCK_STREAM, 0)) == -1)
        fehler_meld(FATAL_SYS, "Server: socket-Aufruf schlug fehl");

    unlink("/tmp/socket"); /* Existierendes /tmp/server loeschen */

    adresse.sun_family = AF_UNIX; /* Unix-Domain-Socket */
    strcpy(adresse.sun_path, "/tmp/socket");

    adrlaenge = sizeof(adresse.sun_family) + strlen(adresse.sun_path);

    /* Casting, um sockaddr_un- in sockaddr-Zeiger umzuwandeln */
    if (bind(sockfd, (struct sockaddr *) &adresse, adrlaenge) == -1)
        fehler_meld(FATAL_SYS, "Server: bind-Aufruf schlug fehl");

    if (listen(sockfd, 5) == -1)
        fehler_meld(FATAL_SYS, "Server: listen-Aufruf schlug fehl");

    while ((connfd = accept(sockfd, (struct sockaddr *)&adresse,
                           &adrlaenge)) >= 0) {
        while ( (n = read(connfd, puffer, MAX_ZEICHEN)) > 0) {
            printf("Server: ");
            if (puffer[0] == 'q')
                break;
            for (i=0; i<n-1; i++)
                printf("%c=%02x ", puffer[i], puffer[i]);
            printf("\n");
            fflush(stdout);
        }
        close(connfd);
    }

    close(sockfd);
    return(0);
}

```

Programm 19.10 (sockserv.c): Server, der die aus dem Socket gelesenen Zeichen hexadezimal ausgibt

Das Programm 19.11 (sockclie.c) ist der Client, der Zeilen von der Standardeingabe liest und dann über den zuvor eingerichteten Socket an den Server schickt.

```

#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "eighdr.h"

```

```

int
main(void)
{
    int            sockfd;
    char           zeile[MAX_ZEICHEN];
    size_t         adrlaenge;
    struct sockaddr_un  adresse;

    if ( (sockfd = socket(PF_UNIX, SOCK_STREAM, 0)) == -1)
        fehler_meld(FATAL_SYS, "Client: socket-Aufruf schlug fehl");

    adresse.sun_family = AF_UNIX; /* Unix-Domain-Socket */
    strcpy(adresse.sun_path, "/tmp/socket");

    adrlaenge = sizeof(adresse.sun_family) + strlen(adresse.sun_path);

    /* Casting, um sockaddr_un- in sockaddr-Zeiger umzuwandeln */
    if (connect(sockfd, (struct sockaddr *)&adresse, adrlaenge) == -1)
        fehler_meld(FATAL_SYS, "Client: connect-Aufruf schlug fehl");

    printf("Client: ");
    while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
        if (write(sockfd, zeile, strlen(zeile)) != strlen(zeile))
            fehler_meld(FATAL_SYS, "Client: Fehler beim Schreiben "
                        "in den Socket");
        if (strlen(zeile) == 2 && zeile[0] == 'q')
            break;
        printf("Client: ");
    }

    close(sockfd);

    return(0);
}

```

Programm 19.11 (sockclie.c): Client, der aus stdin gelesene Zeilen über die Socket-Verbindung an den Server schickt

Nachdem man die beiden Programme kompiliert und gelinkt hat

```

cc -o sockserv sockserv.c fehler.c
cc -o sockclie sockclie.c fehler.c

```

kann man auf einer virtuellen Konsole das Programm `sockserv` und auf einer anderen das Programm `sockclie` starten. Jede beim Clientprogramm `sockclie` eingegebene Zeile wird dann auf der anderen virtuellen Konsole vom Serverprogramm `sockserv` wieder ausgegeben, wobei zu jedem einzelnen Zeichen noch dessen hexadezimaler ASCII-Code mitausgegeben wird.

Da Unix-Domain-Sockets einige Vorteile gegenüber Pipes aufweisen, wie z.B. die Vollduplexfähigkeit, werden sie oft für die Interprozesskommunikation (IPC) eingesetzt; dafür wird auch eine eigene Funktion **socketpair** angeboten:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socketpair(int d, int type, int protocol, int sv[2]);
```

gibt zurück: 0 (bei Erfolg); -1 bei Fehler

Die ersten drei Parameter entsprechen denen von der der Funktion **socket** und der letzte Parameter entspricht weitgehend dem Parameter bei der Funktion **pipe**. Hier trägt die Funktion **socketpair** die beiden Filedeskriptoren für den Socket ein, der aber nun anders als eine Pipe vollduplexfähig ist

In Kapitel 19.2.2 ist bei der Realisierung einer Stream Pipe unter BSD bzw. unter Linux eine Anwendung zur Funktion **socketpair** gegeben.

Da SVR4 auch *Unix Domain Sockets* unterstützt, ist das Programm 19.10 (bsd44_cs.c) auch unter SVR4 lauffähig.

In der Funktion **serv_initverbind** wird dabei zunächst mit einem **socket**-Aufruf ein *Unix Domain Socket* eingerichtet, bevor eine Variable vom Strukturtyp `sockaddr_un` entsprechend dem zwischen Client und Server vereinbarten Namen gesetzt wird. Mit einem **bind**-Aufruf wird dieser Name (`/tmp/opend`) dem *Socket* zugeordnet. Mit dem nachfolgenden **listen**-Aufruf wird der Kern darüber informiert, daß dieser Prozeß der Server ist und auf Verbindungsanforderungen durch Clients wartet. Das zweite Argument (5) bei **listen** ist die maximal mögliche Anzahl von ausstehenden Client-Anforderungen, die der Kern für diesen Filedeskriptor in einer Warteschlange halten kann. In vielen Implementierungen ist dieser maximale Wert auf 5 festgelegt.

In der Funktion **cli_verbind** im Programm 19.10 (bsd44_cs.c) wird zunächst mit einem **socket**-Aufruf die Client-Seite einer *Unix Domain Socket* eingerichtet, bevor eine Variable vom Strukturtyp `sockaddr_un` mit dem Client-spezifischen Namen (hier `/var/tmp/xxxxx`, wobei `xxxxx` für die Prozeß-ID steht) gesetzt wird. Mit einem **bind**-Aufruf wird dieser Name dem Client-Socket zugeordnet. Mit dem darauffolgenden **chmod**-Aufruf werden für diesen Socket-Namen die Zugriffsrechte auf 700 (`rwX-----`) festgelegt. Diese Rechte und die User-ID des Socket werden von der Funktion **serv_bereit** benutzt, um die Berechtigung eines Clients zu überprüfen. Danach wird die entsprechende Variable vom Strukturtyp `sockaddr_un` neu gesetzt, um mit einem **connect**-Aufruf eine Verbindung zu dem mit dem Server vereinbarten Pfadnamen herzustellen.

In der Funktion **serv_bereit** wird mit der **accept**-Funktion auf die Anforderung eines Clients (**cli_verbind**-Aufruf) gewartet. Die Funktion **accept** liefert bei ihrer Rückkehr immer einen neuen Deskriptor, der eine Client-Server-Verbindung darstellt. Zusätzlich wird der vom Client seinem Socket zugeordnete Pfadname (siehe **cli_verbind**) über das zweite Argument von **accept** (`sockaddr_un *`) geliefert. Mit einem **stat**-Aufruf werden die Zugriffsrechte für den Pfadnamen erfragt, bevor geprüft wird, ob für diesen Pfadnamen nur die Zugriffsrechte *user-read*, *user-write* und *user-execute* gesetzt sind. Zusätzlich wird überprüft, ob die mit diesem Pfadnamen verknüpften Zeiten (Zugriffs-, Kreierungs- und Modifikationszeiten) nicht älter als 30 Sekunden sind.

Nur wenn alle diese Überprüfungen erfolgreich sind, wird angenommen, daß der Client (effektive User-ID) der Besitzer des Sockets ist. Diese Form der Überprüfung ist natürlich nicht sehr einsichtig, aber zur Zeit die einzige Möglichkeit. Diese umständliche Form der Überprüfung könnte wesentlich vereinfacht werden, wenn der Kern die effektive User-ID bei **accept** liefern würde.

Abbildung 19.8 verdeutlicht die Konstellation, die nach einem **cli_verbind**-Aufruf vorliegt, wenn der zwischen Server und Clients vereinbarte Pfadname `/tmp/opensd` ist.

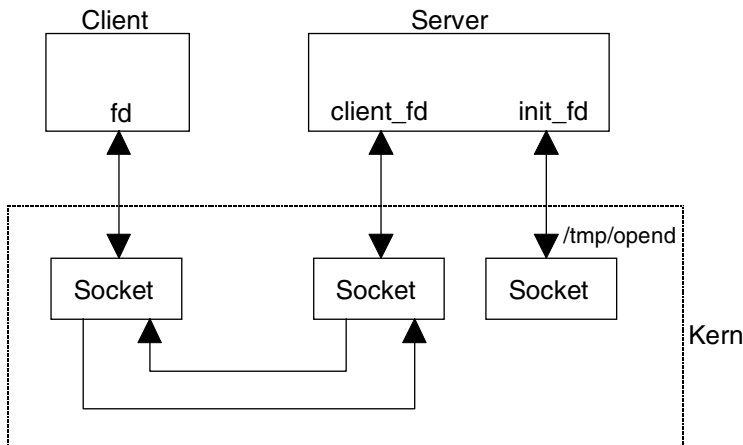


Abbildung 19.9: Client-Server-Verbindung über ein Unix Domain Socket

```

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <stddef.h>
#include <time.h>
#include "eighdr.h"

/*----- serv_initverbind -----*/
int serv_initverbind(const char *name)
{
    int fd, groesse;
    struct sockaddr_un unix_adr;

    /*--- Kreieren eines Unix domain stream socket ---*/
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0 )
        return(-1);

    unlink(name);

    /*--- Schreiben der socket-adress-Struktur ---*/
    memset(&unix_adr, 0, sizeof(unix_adr));
    unix_adr.sun_family = AF_UNIX;
  
```

```

strcpy(unix_adr.sun_path, name);
groesse = strlen(unix_adr.sun_path) + sizeof(unix_adr.sun_family);

/*--- Zuordnen des Namens zum Filedescriptor ---*/
if (bind(fd, (struct sockaddr *) &unix_adr, groesse) < 0)
    return(-2);

/*--- Kern mitteilen, dass man der Server ist ---*/
if (listen(fd, 5) < 0)
    return(-3);

return(fd);
}

/*----- serv_bereit -----*/
int serv_bereit(int initfd, uid_t *uidzgr)
{
    int                client_fd, groesse;
    time_t             ablauf_zeit;
    struct sockaddr_un  unix_adr;
    struct stat         statpuff;

    groesse = sizeof(unix_adr);
    if ( ( client_fd =
          accept(initfd, (struct sockaddr *) &unix_adr, &groesse)) < 0)
        return(-1);

    groesse -= sizeof(unix_adr.sun_family);
    unix_adr.sun_path[groesse] = '\0';

    if (stat(unix_adr.sun_path, &statpuff) < 0)
        return(-2);

    if ((statpuff.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statpuff.st_mode & S_IRWXU) != S_IRWXU)
        return(-3);    /* nicht rwx----- */

    /*--- Name vom Client darf nicht aelter als 30 Sek. sein ---*/
    ablauf_zeit = time(NULL) - 30;
    if (statpuff.st_atime < ablauf_zeit ||
        statpuff.st_ctime < ablauf_zeit ||
        statpuff.st_mtime < ablauf_zeit)
        return(-4);    /* i-node ist zu alt */

    if (uidzgr != NULL)
        *uidzgr = statpuff.st_uid;

    unlink(unix_adr.sun_path);

    return(client_fd);
}

/*----- cli_verbind -----*/

```



```

int cli_verbind(const char *name)
{
    int          fd, groesse;
    struct sockaddr_un  unix_adr;

    /*--- Kreieren eines Unix domain stream socket ---*/
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /*--- Schreiben der socket-adress-Struktur mit eigener Adresse ---*/
    memset(&unix_adr, 0, sizeof(unix_adr));
    unix_adr.sun_family = AF_UNIX;
    sprintf(unix_adr.sun_path, "/var/tmp/%05d", getpid());
    groesse = strlen(unix_adr.sun_path) + sizeof(unix_adr.sun_family);
    if (groesse != 16)
        fehler_meld(FATAL_SYS, "ungueltige Laenge");

    unlink(unix_adr.sun_path);

    /*--- Zuordnen des Namens zum Filedeskriptor ---*/
    if (bind(fd, (struct sockaddr *) &unix_adr, groesse) < 0)
        return(-2);

    if (chmod(unix_adr.sun_path, S_IRWXU) < 0)
        return(-3);

    /*--- Schreiben der socket-adress-Struktur mit Server-Adresse ---*/
    memset(&unix_adr, 0, sizeof(unix_adr));
    unix_adr.sun_family = AF_UNIX;
    strcpy(unix_adr.sun_path, name);
    groesse = strlen(unix_adr.sun_path) + sizeof(unix_adr.sun_family);

    if (connect(fd, (struct sockaddr *) &unix_adr, groesse) < 0)
        return(-4);

    return(fd);
}

```

Programm 19.12 (*bsd44_cs.c*): Realisierung von **serv_initverbind**, **serv_bereit** und **cli_verbind** mit Sockets

19.6 Client-Server-Realisierung mit nicht verwandten Prozessen

In Kapitel 19.4 wurde ein Server entwickelt, der für das Öffnen von Dateien zuständig ist. Der dort entwickelte Server wurde vom Client durch **fork** und **exec** gestartet. Es bestand also immer ein Verwandtschaftsverhältnis zwischen Client (Elternprozeß) und Server (Kindprozeß).

In diesem Kapitel wird nun ein Server entwickelt, der das gleiche leistet wie der von Kapitel 19.4 (Öffnen von Dateien), aber nicht als Kindprozeß, sondern als Dämonprozeß abläuft, so daß nicht verwandte Prozesse (Clients) Anforderungen an diesen Server schicken können.

Bei dieser Client-Server-Realisierung werden die im vorherigen Kapitel vorgestellten Funktionen **serv_initverbind**, **serv_bereit** und **cli_verbind** verwendet. Es wird dabei auch gezeigt, wie ein Server mehrere Clients mit den Funktionen **select** oder **poll** bedienen kann.

Der zwischen Client und Server vereinbarte Pfadname der Stream Pipe wird in der Headerdatei `cliser2.h` definiert.

```
#ifndef __CLISER2
#define __CLISER2

#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include "eighdr.h"

/* Zwischen Server und Client vereinbarter Name */
/* fuer die benannte Stream Pipe */
/* /tmp sollte hier durch einen anderen Pfad ersetzt werden. */
#define CS_NAME "/tmp/openssl"

#endif
```

Programm 19.13 Headerdatei **cliser2.h**: Gemeinsame Definitionen für Clients und Server

19.6.1 Client

Das Client-Programm 19.11 (`opencli2.c`) ist dem Programm 19.7 (`opencli.c`) im Kapitel 19.4 sehr ähnlich. Die `main`-Funktion hat sich nicht verändert, und es wird auch das gleiche Protokoll benutzt. Ein wesentlicher Unterschied zum Programm 19.7 (`opencli.c`) ist, daß das Programm 19.11 (`opencli2.c`) anstelle von **fork** und **exec** nun **cli_verbind** in der Funktion **server_open** verwendet.

```
#include "cliser2.h"
#include <sys/uio.h> /* struct iovec */

#define PUFF_GROESSE 8192

static int server_open(char *name, int openflag);

/*----- main -----*/
int
main(int argc, char *argv[])
{
    int    n, fd;
    long   zeichzahl;
```

```

char    puffer[PUFF_GROESSE], zeile[MAX_ZEICHEN];

while (fgets(zeile, MAX_ZEICHEN, stdin) != NULL) {
    zeile[strlen(zeile) - 1] = '\0'; /* \n mit \0 ueberschreiben */
    zeichzahl = 0;
    if ( (fd = server_open(zeile, O_RDONLY)) >= 0) {
        while ( (n = read(fd, puffer, PUFF_GROESSE)) > 0) {
            zeichzahl += n;
            if (write(STDOUT_FILENO, puffer, n) != n)
                fehler_meld(FATAL_SYS, "write-Fehler");
        }
        if (n < 0)
            fehler_meld(FATAL_SYS, "read-Fehler");
        fprintf(stderr, "---- %s: %ld Zeichen ---\n", zeile, zeichzahl);
        close(fd);
    }
}
exit(0);
}

/*----- server_open -----
*          sendet den Dateinamen und open-Flags an den entspr.
*          open-Server und empfaengt dann den Filesdeskriptor
*          fuer die von diesem Server geoeffnete Datei          */
static int server_open(char *name, int openflag)
{
    char        puffer[10];
    struct iovec iov[3];
    static int   csfd = -1;

    if (csfd < 0) { /* Verbindung zum Server herstellen */
        if ( (csfd = cli_verbind(CS_NAME)) < 0)
            fehler_meld(FATAL_SYS, "cli_verbind-Fehler");
    }
    sprintf(puffer, " %d", openflag);
    iov[0].iov_base = "open ";
    iov[0].iov_len  = strlen("open ");
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = puffer;
    iov[2].iov_len  = strlen(puffer)+1; /* +1 wegen abschl. \0 */
    if (writev(csfd, &iov[0], 3) !=
        iov[0].iov_len + iov[1].iov_len + iov[2].iov_len)
        fehler_meld(FATAL_SYS, "writev-Fehler");

    return( empfang_fd(csfd, write) );
}

```

Programm 19.14 (opencli2.c): Client, der zum Öffnen einer Datei einen Server benutzt

19.6.2 Server

Da hier anders als in Kapitel 19.4 eine Realisierung von nicht verwandten Clients und Server vorgenommen wird, muß der Server sich in einem Array den Zustand jeder Client-Verbindung merken. Zur Verwaltung dieser Arrays bietet das Programm 19.12 (openser2.c) die drei Funktionen `client_add`, `client_loesch` und `client_allokiere` an.

Da der Server als Dämonprozeß abläuft, werden Fehler nicht mit der Funktion `fehler_meld`, sondern mit der Funktion `log_meld` ausgegeben. `log_meld` schreibt die entsprechenden Fehlermeldungen nicht wie `fehler_meld` auf die Standardfehlerausgabe, sondern in eine Log-Datei.

Die `main`-Funktion ruft nach der Abarbeitung der Kommandozeile die Funktion `schleife` auf. Diese Funktion `schleife` ruft zunächst **`serv_initverbind`** auf, bevor sie in einer Endlosschleife die Client-Anforderungen entgegennimmt.

Die Entgegennahme von Client-Anforderungen erfolgt dabei mit der Funktion **`select`**. Nach einer Rückkehr von **`select`** gibt es grundsätzlich zwei Möglichkeiten:

1. Der Deskriptor `init_fd` ist für Lesen bereit, was bedeutet, daß ein neuer Client mit **`cli_verbind`** eine Verbindung zum Server angefordert hat. Um diese Anforderung zu bedienen, wird anschließend **`serv_bereit`** aufgerufen, und dann mit einem `client_add`-Aufruf der Deskriptor und die User-ID des Clients im `client[]`-Array festgehalten.
2. Eine bereits bestehende Client-Verbindung ist für Lesen bereit, was bedeutet, daß der entsprechende Client eine neue Anforderung geschickt oder sich eben beendet hat. Hat ein Client sich beendet, so liefert der darauffolgende **`read`**-Aufruf 0 (für Dateiende) als Rückgabewert, andernfalls liegt eine neue Client-Anforderung an, die mit einem Aufruf der Funktion `anforderung` abgehandelt wird.

Die Variable `allmenge` enthält immer alle momentan benutzten Deskriptoren. Wenn ein neuer Client eine Verbindung zum Server herstellt, so wird das entsprechende Bit in der Deskriptormenge gesetzt. Dieses Bit wird wieder gelöscht, wenn der Client sich beendet.

```
#include "cliser2.h"
#include <syslog.h>
#include <sys/time.h>

/*----- Konstanten -----*/
#define MAX_ARGC      100
#define REALLOC_ZAHL  10

/*----- Datentyp Client -----*/
typedef struct { /* Struktur fuer jeden verbundenen Client */
    int    fd; /* Filedeskriptor oder -1 */
    uid_t  uid;
} Client;

/*----- Globale Variablen -----*/
char    fehl_meldung[MAX_ZEICHEN];
Client  *client=NULL; /* Adresse des allokierten Arrays */
```

```

int      client_anzahl=0; /* Anzahl der Eintraege im Array client[] */
int      debug;          /* TRUE, wenn interaktiv (kein Daemon) */

/*----- Prototypen fuer lokale Funktionen -----*/
static int  daemonisierung(void);
static void client_allokiere(void);
static int  client_add(int fd, uid_t uid);
static void client_loesch(int fd);
static void schleife(void);
static void anforderung(char *puffer, int byte_gelesen,
                        int client_fd, uid_t uid);
static int  puffer_argv(char *puffer, int *argc, char *argv[]);

/*----- main -----*/
int
main(int argc, char *argv[])
{
    int  z;

    log_open("openser2", LOG_PID, LOG_USER);

    opterr = 0; /* getopt soll nicht auf stderr schreiben */
    while ( (z = getopt(argc, argv, "d")) != EOF) {
        if (z == 'd')
            debug = 1;
        else if (z == '?')
            fehler_meld(FATAL, "unerlaubte Option: -%c", optopt);
    }
    if (debug == 0)
        daemonisierung();

    schleife(); /* Realisierung dieser Funktion, die niemals zurueckkehrt,
                sowohl mit select als auch mit poll moeglich */
}

/*----- daemonisierung -----*/
static int  daemonisierung(void)
{
    pid_t    pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0); /* Elternprozess beendet sich */

    /*---- Ab hier wird nur vom Kindprozess ausgefuehrt */
    setsid(); /* Kind wird Session-Fuehrer */
    umask(0); /* Dateikreierungsmaske loeschen */
    return(0);
}

/*----- client_allokiere -----*/

```

```

static void client_allokiere(void)
{
    int i;

    if (client == NULL)
        client = malloc(REALLOC_ZAHL * sizeof(Client));
    else
        client = realloc(client, (client_anzahl+REALLOC_ZAHL) * sizeof(Client));
    if (client == NULL)
        fehler_meld(FATAL_SYS, "Speicherplatzmangel (bei client[]-Array)");

    for (i=client_anzahl; i<client_anzahl+REALLOC_ZAHL; i++)
        client[i].fd = -1;
    client_anzahl += REALLOC_ZAHL;
}

/*----- client_add -----*/
static int client_add(int fd, uid_t uid)
{
    int i;

    if (client == NULL) /* Beim ersten Aufruf muss immer allokiert werden */
        client_allokiere();

    do {
        for (i=0; i<client_anzahl; i++) {
            if (client[i].fd == -1) {
                client[i].fd = fd;
                client[i].uid = uid;
                return(i);
            }
        }
        client_allokiere(); /* notwendig, da kein freier Platz gefunden */
    } while (1);
}

/*----- client_loesch -----*/
static void client_loesch(int fd)
{
    int i;

    for (i=0; i<client_anzahl; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
        }
    }
    log_meld(FATAL, "kein Client-Eintrag zu Filedeskr. %d gefunden", fd);
}

/*----- schleife -----*/
static void schleife(void)
{

```

```

int      i, n, maxfd, maxi, initfd, client_fd, byte_gelesen;
char     puffer[MAX_ZEICHEN];
uid_t    uid;
fd_set   rmenge, allmenge;

FD_ZERO(&allmenge);
if ( (initfd = serv_initverbind(CS_NAME)) < 0)
    log_meld(FATAL_SYS, "serv_initverbind-Fehler");
FD_SET(initfd, &allmenge);
maxfd = initfd;
maxi = -1;

while (1) {
    rmenge = allmenge;
    if ( (n = select(maxfd+1, &rmenge, NULL, NULL, NULL)) < 0)
        log_meld(FATAL_SYS, "select-Fehler");

    if (FD_ISSET(initfd, &rmenge)) { /* Neue Clientanforderung zulassen */
        if ( (client_fd = serv_bereit(initfd, &uid)) < 0)
            log_meld(FATAL_SYS, "serv_bereit-Fehler: %d", client_fd);
        i = client_add(client_fd, uid);
        FD_SET(client_fd, &allmenge);
        if (client_fd > maxfd)
            maxfd = client_fd; /* groesster Filedeskr. fuer select */
        if (i > maxi)
            maxi = i; /* Neue Anzahl von Clients im Array client[] */
        log_meld(WARNUNG, "Neue Verbindung: uid %d, fd %d", uid, client_fd);
        continue;
    }

    for (i=0; i<=maxi; i++) { /* Durchlaufen des client[]-Arrays */
        if ( (client_fd = client[i].fd) >= 0 &&
            FD_ISSET(client_fd, &rmenge)) {
            if ( (byte_gelesen = read(client_fd, puffer, MAX_ZEICHEN)) < 0)
                log_meld(FATAL_SYS, "Lese-Fehler fuer Filedeskr.%d", client_fd);
            else if (byte_gelesen == 0) {
                log_meld(WARNUNG, "Verbindung beendet: uid %d, fd %d",
                    client[i].uid, client_fd);
                client_loesch(client_fd);
                FD_CLR(client_fd, &allmenge);
                close(client_fd);
            } else
                anforderung(puffer, byte_gelesen, client_fd, client[i].uid);
        }
    }
}

/*----- anforderung -----*/
static void anforderung(char *puffer, int byte_gelesen,
                        int client_fd, uid_t uid)
{
    char     *argv[MAX_ARGC];
    int      argc, neuofd;

```

```

if (puffer[byte_gelesen-1] != '\0') {
    sprintf( Fehl_meldung, "Anforderung von %d ohne abschl. \\0: %*.s\n",
            uid, byte_gelesen, byte_gelesen, puffer);
    send_fehl(client_fd, -1, Fehl_meldung);
    return;
}
log_meld(WARNUNG, "Anforderung: %s, von uid %d", puffer, uid);
if (puffer_argv(puffer, &argc, argv) < 0) {
    send_fehl(client_fd, -1, Fehl_meldung);
    log_meld(WARNUNG, Fehl_meldung);
    return;
}
if ( (neufd = open(argv[1], atoi(argv[2]))) < 0) {
    sprintf( Fehl_meldung, "kann %s nicht oeffnen: %s\n",
            argv[1], strerror(errno));
    send_fehl(client_fd, -1, Fehl_meldung);
    log_meld(WARNUNG, Fehl_meldung);
    return;
}
if (send_fd(client_fd, neufd) < 0)
    log_meld(FATAL_SYS, "send_fd-Fehler");
log_meld(WARNUNG, "Filesdeskr. %d fuer %s ueber Filesdeskr. %d geschickt",
        neufd, argv[1], client_fd);
close(neufd);
}

/*----- puffer_argv -----
 *      legt die im 'puffer' enthaltenen Argumente im Array argv ab.
 *      Die Anzahl der Argumente wird dabei in 'argc' abgelegt.      */
static int puffer_argv(char *puffer, int *argc, char *argv[])
{
    char    *zgr;

    if (strtok(puffer, " \\t\\n") == NULL)
        return(-1);

    argv[*argc=0] = puffer;
    while ( (zgr = strtok(NULL, " \\t\\n")) != NULL) {
        if (++*argc >= MAX_ARGC-1)
            return(-1);
        argv[*argc] = zgr;
    }
    argv[++*argc] = NULL;

    if (*argc != 3 || strcmp(argv[0], "open")) {
        strcpy(Fehl_meldung, "Falsches Protokoll (erwartet: open name flag)\\n");
        return(-1);
    }
    return(0);
}

```

Programm 19.15 (openser2.c): Server, der von Clients geschickte Dateinamen öffnet

Nachdem man die beiden Programme 19.7 (opencli.c) und 19.8 (openser.c) kompiliert und gelinkt hat

```
cc -o opencli2 opencli2.c readwrit.c svr4.c svr4_cs.c fehler.c (SVR4)
cc -o openser2 openser2.c readwrit.c svr4.c svr4_cs.c fehler.c (SVR4)
cc -o opencli2 opencli2.c readwrit.c bsd4_4.c bsd44_cs.c fehler.c -lsocket -lnsl
                                                    (BSD/Linux4)
cc -o openser2 openser2.c readwrit.c bsd4_4.c bsd44_cs.c fehler.c -lsocket -lnsl
                                                    (BSD/Linux5)
```

muß man openser2 im Hintergrund starten. Startet man dann opencli2 im Vordergrund, so kann man Dateinamen interaktiv eingeben. Zu diesen Dateien wird dann deren Inhalt und die Anzahl der Bytes in dieser Datei ausgegeben.

Im Programm 19.12 (openser3.c) wird die Funktion schleife nicht mit **select**, sondern mit der Funktion **poll** gelöst. Beim folgenden Listing sind nur die Unterschiede zum Programm 19.11 (fett gedruckt) angegeben. Auch werden in diesem Listing die völlig identischen Daten und Funktionen zu Programm 19.11 (openser2.c) nicht nochmals angegeben.

```
#include "cliser2.h"
#include <limits.h>
#include <syslog.h>
#include <poll.h>

.....
.....

/*----- main -----*/
int
main(int argc, char *argv[])
{
    int z;

    log_open("openser3", LOG_PID, LOG_USER);

    opterr = 0; /* getopt soll nicht auf stderr schreiben */
    while ( (z = getopt(argc, argv, "d")) != EOF) {
        if (z == 'd')
            debug = 1;
        else if (z == '?')
            fehler_meld(FATAL, "unerlaubte Option: -%c", optopt);
    }
    if (debug == 0)
        daemonisierung();

    schleife(); /* Realisierung dieser Funktion, die niemals zurueckkehrt,
                sowohl mit select als auch mit poll moeglich */
}
```

4. Unter Linux -lsocket weglassen.

5. Unter Linux -lsocket weglassen.

```

}

/*----- daemonisierung -----*/
static int daemonisierung(void)
{
    .....
}

/*----- client_allokiere -----*/
static void client_allokiere(void)
{
    .....
}

/*----- client_add -----*/
static int client_add(int fd, uid_t uid)
{
    .....
}

/*----- client_loesch -----*/
static void client_loesch(int fd)
{
    .....
}

/*----- schleife -----*/
static void schleife(void)
{
    int          i, n, maxi, initfd, client_fd, byte_gelesen;
    char         puffer[MAX_ZEICHEN];
    uid_t        uid;
    struct pollfd *pollfd;

    if ( (pollfd = malloc(OPEN_MAX * sizeof(struct pollfd))) == NULL)
        fehler_meld(FATAL_SYS, "Speicherplatz-Mangel");
    if ( (initfd = serv_initverbind(CS_NAME)) < 0)
        log_meld(FATAL_SYS, "serv_initverbind-Fehler");
    client_add(initfd, 0); /* [0] wird fuer initfd benutzt */
    pollfd[0].fd = initfd;
    pollfd[0].events = POLLIN;
    maxi = 0;

    while (1) {
        if ( (n = poll(pollfd, maxi+1, -1)) < 0)
            log_meld(FATAL_SYS, "poll-Fehler");

        if (pollfd[0].revents & POLLIN) { /*Neue Clientanforderung zulassen */
            if ( (client_fd = serv_bereit(initfd, &uid)) < 0)
                log_meld(FATAL_SYS, "serv_bereit-Fehler: %d", client_fd);
            i = client_add(client_fd, uid);
            pollfd[i].fd = client_fd;
            pollfd[i].events = POLLIN;
            if (i > maxi)

```

```

        maxi = i; /* Neue Anzahl von Clients im Array client[] */
        log_meld(WARNUNG, "Neue Verbindung: uid %d, fd %d", uid, client_fd);
    }
    for (i=1; i<=maxi; i++) { /* Durchlaufen des client[]-Arrays */
        if ( (client_fd = client[i].fd) >= 0) {
            if (pollfd[i].revents & POLLHUP) {
                log_meld(WARNUNG, "Verbindung beendet: uid %d, fd %d",
                        client[i].uid, client_fd);
                client_loesch(client_fd);
                pollfd[i].fd = -1;
                close(client_fd);
            } else if (pollfd[i].revents & POLLIN) {
                if ( (byte_gelesen = read(client_fd, puffer, MAX_ZEICHEN)) < 0)
                    log_meld(FATAL_SYS, "Lese-Fehler fuer Fd. %d", client_fd);
                else if (byte_gelesen == 0) {
                    log_meld(WARNUNG, "Verbindung beendet: uid %d, fd %d",
                            client[i].uid, client_fd);
                    client_loesch(client_fd);
                    pollfd[i].fd = -1;
                    close(client_fd);
                } else
                    anforderung(puffer, byte_gelesen, client_fd, client[i].uid);
            }
        }
    }
}

/*----- anforderung -----*/
static void anforderung(char *puffer, int byte_gelesen,
                        int client_fd, uid_t uid)
{
    .....
}

/*----- puffer_argv -----*/
static int puffer_argv(char *puffer, int *argc, char *argv[])
{
    .....
}

```

Programm 19.16 (**openser3.c**): Alternative Server-Realisierung zu *openser2.c* mit Funktion **poll**

Im Array-Element `client[0]` befindet sich dabei immer der `initfd`-Deskriptor. Die Ankunft einer neuen Client-Verbindungsanforderung wird durch `POLLIN` beim `initfd`-Deskriptor angezeigt. Zur Abhandlung einer solchen Verbindungsanforderung wird auch hier ***serv_bereit*** aufgerufen.

Für einen bereits existierenden Client müssen zwei verschiedene **poll**-Ereignisse behandelt werden:

1. Beendigung eines Clients (wird durch `POLLHUP` angezeigt)
2. Ankommen einer neuen Anforderung durch einen bereits existierenden Client (wird durch `POLLIN` angezeigt). Solche neu angekommenen Client-Anforderungen werden durch den Aufruf der Funktion `anforderung` abgehandelt.

19.7 Netzwerkprogrammierung mit TCP/IP

Netzwerkprogrammierung gewinnt mit der zunehmenden Vernetzung von Computern ständig an Bedeutung. Um verschiedene Rechner in einem Netzwerk miteinander kommunizieren zu lassen, werden hauptsächlich Sockets eingesetzt. Die Grundlagen der Socket-Programmierung wurden bereits in Kapitel 19.5.3 behandelt.

Das am häufigsten verwendete Protokoll für die Kommunikation in lokalen und weltweiten Netzen ist die TCP/IP-Protokollfamilie. Unter den meisten heutigen Unix-Systemen und auch unter Linux steht eine vollständige und stabile TCP/IP-Implementierung zur Verfügung, mit der es möglich ist, Linux-/Unix-Rechner sowohl als TCP/IP-Server als auch als Client einzusetzen.

Die derzeit gültige TCP/IP-Version ist *IPv4*, die hier auch beschrieben wird. Die Nachfolgeversion (*IPv6*) befindet sich in der Designphase; sie sollte abwärtskompatibel zu *IPv4* sein.

19.7.1 Byteanordnung bei TCP/IP

TCP/IP-Protokolle können auch in Netzen eingesetzt werden, die nicht aus gleichen Rechnern bestehen. Hieraus ergeben sich dann jedoch Architekturunterschiede. Einer der häufigsten Unterschiede ist die interne Anordnung der Bytes zur Speicherung von Zahlen. Nimmt man z.B. den Datentyp `int` von der Programmiersprache C, der üblicherweise unter Linux/Unix 32 Bit (4 Byte) umfaßt, so gibt es verschiedene Möglichkeiten für die Anordnung dieser 4 Byte im Speicher, wobei die beiden häufigsten die folgenden sind:

Big-Endian

Architekturen, die dieser Strategie folgen, speichern das höchstwertige Byte an der niedrigsten Adresse, das nächst höchstwertige Byte an der nächst höheren Adresse und so weiter.

Little-Endian

Architekturen, die dieser Strategie folgen, gehen genau umgekehrt zur Big-Endian-Strategie vor. Sie speichern das niederwertigste Byte an der niedrigsten Adresse, das nächst niederwertige Byte an der nächst höheren Adresse und so weiter.

Es existieren jedoch auch Rechner, die keiner dieser beiden Strategien folgen, sondern noch andere Anordnungsstrategien verwenden.

Unabhängig von der am lokalen Rechner verwendeten Anordnungsstrategie schreibt TCP/IP für die Übertragung von Protokollinformationen die *Big-Endian*-Anordnung vor. Für Anwendungsdaten schlägt es diese Strategie nur vor, überprüft dies aber nicht. Die Reihenfolge der Bytes bei der Übertragung von Zahlen bezeichnet man mit *network byte order*.

Um Zahlen von der lokal verwendeten Byte-Reihenfolge (*host byte order*) in die *network byte order* zu konvertieren bzw. umgekehrt, stehen die folgenden Funktionen zur Verfügung:

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
    gibt zurück: die network byte order zum long-Wert hostlong, der in host byte order übergeben wird

unsigned short int htons(unsigned short int hostshort);
    gibt zurück: die network byte order zum short-Wert hostshort, der in host byte order übergeben wird

unsigned long int ntohl(unsigned long int netlong);
    gibt zurück: die host byte order zum long-Wert netlong, der in network byte order übergeben wird

unsigned short int ntohs(unsigned short int netshort);
    gibt zurück: die host byte order zum short-Wert netshort, der in network byte order übergeben wird
```

Auch wenn die Prototypen zu diesen Funktionen für *unsigned*-Zahlen ausgelegt sind, können sie doch auch für vorzeichenbehaftete Zahlen verwendet werden.

Bei den Prototypen der obigen Funktionen steht der Datentyp *long* für 32-Bit-Werte, was bedeutet, daß hierfür unter Linux/Unix der Datentyp *int* (32 Bit) und nicht der Datentyp *long* (64 Bit) zu verwenden ist.

19.7.2 IP-Adressen und Port-Nummern

IPv4-Verbindungen setzen sich aus vier Teilen zusammen:

- ▶ *Local Host* (IP-Adresse des lokalen Rechners)
- ▶ *Local Port* (Port-Nummer am lokalen Rechner)
- ▶ *Remote Host* (IP-Adresse des entfernten Rechners)
- ▶ *Remote Port* (Port-Nummer am entfernten Rechner)

Vor dem Aufbau einer Verbindung muß jeder dieser vier Teile gesetzt werden. Eine IP-Adresse ist dabei eine 32 Bit lange Zahl, die im gesamten Netzwerk eindeutig ist, was bedeutet, daß keine IP-Adresse mehrmals an unterschiedliche Rechner vergeben sein darf.

IP-Adressen setzen sich aus vier Zahlen zusammen, die mit Punkt voneinander getrennt sind, wie etwa 192.168.1.2. Das höchstwertige Byte der Adresse ist die Zahl ganz links (192). Dieses Format für IP-Adressen wird auch *punktierte Dezimaldarstellung* (*dotted-decimal notation*) genannt.

Lokale Netze, die nicht ständig mit dem Internet verbunden sind, sollten IP-Nummern verwenden, die mit 192.168 beginnen, da diese Nummernkombinationen speziell für diesen Zweck reserviert sind und Nummern-Konflikte vermeiden.

Da üblicherweise auf einem Rechner mit einer IP-Nummer mehrere TCP/IP-Anwendungen laufen, reicht die IP-Nummer alleine nicht aus, um eine Verbindung zu einem Rechner eindeutig zu identifizieren. Hierfür werden nun die *Port-Nummern* benötigt. Bei den Port-Nummern handelt es sich um 16-Bit-Zahlen, die den Endpunkt einer Verbindung zu einem Rechner eindeutig festlegen.

Mit der IP-Adresse und der Port-Nummer kann nun der Endpunkt einer Verbindung in einem TCP/IP-Netzwerk, wozu z.B. auch das Internet zählt, eindeutig festgelegt werden. Ein TCP-Verbindung wird dann durch zwei Verbindungsendpunkte gebildet, die jeweils durch eine IP-Nummer und eine Port-Nummer eindeutig festgelegt sind.

Meist werden die Port-Nummern intern vom System in zwei Klassen aufgeteilt. Z.B. sind in Linux die Port-Nummern von 0 bis 1024 für Prozesse reserviert, die mit Superuser-Rechten laufen.

19.7.3 IP-Socket-Adressen

Bei Sockets werden die IP-Adressen in der Struktur `sockaddr_in` gespeichert:

```
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in {
    short int     sin_family; /* AF_INET */
    unsigned short int sin_port; /* Port-Nummer */
    struct in_addr sin_addr; /* IP-Adresse */
};
```

Der ersten Komponente `sin_family` muß dabei `AF_INET` zugewiesen werden, um die Adresse als IP-Adresse zu kennzeichnen.

Die zweite Komponente `sin_port` enthält die Port-Nummer in der *network byte order* und die dritte Komponente die IP-Nummer des Rechners für diese TCP-Adresse.

Werden in `sin_port` und `sin_addr` nur 0-Bytes hinterlegt, so ist man nicht an diesen Werten interessiert, was oft für Server-Prozesse der Fall ist, da diese Verbindungen zu jeder Adresse des lokalen Rechners annehmen. Eine Anwendung, die jedoch genau auf eine Adresse ausgelegt ist, muß sie in den beiden Komponenten `sin_port` und `sin_addr` genau spezifizieren.

19.7.4 Manipulieren, Konvertieren und Extrahieren von IP-Adressen

Um eine IP-Adresse von der punktierten Dezimaldarstellung in einen numerischen Wert umzuwandeln, stehen die Funktionen **inet_aton** und **inet_addr** zur Verfügung.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
int inet_aton(const char *cp, struct in_addr *inp);
```

gibt zurück: Wert verschieden 0 (bei Erfolg); 0 bei Fehler

inet_aton konvertiert die übergebene IP-Adresse *cp* von der punktierten Dezimaldarstellung in einen numerischen Wert, den sie im Speicherplatz hinterlegt, auf den *inp* zeigt.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
unsigned long int inet_addr(const char *cp);
```

gibt zurück: numerischen Wert (*network byte order*) zur IP-Adresse *cp* (bei Erfolg); -1 bei Fehler

Die heute veraltete Funktion **inet_addr** konvertiert – wie die Funktion **inet_aton** – die übergebene IP-Adresse *cp* von der punktierten Dezimaldarstellung in einen numerischen Wert (*network byte order*), den sie als Rückgabewert liefert.

Diese Funktion sollte heute nicht mehr verwendet werden, da sie zwei Probleme aufweist, die aus ihrem Rückgabotyp `long` resultieren:

- Es kann bei der Rückgabe nicht zwischen -1 (für Fehler) und der gültigen Adresse 255.255.255.255 unterschieden werden.
- Während andere Funktionen den Datentyp `struct in_addr` für numerische Werte von IP-Adressen verwenden, liefert diese Funktion einen `long`-Wert, was ein unschönes Casting für die anderen Funktionen erfordert, die mit diesem Wert weiterarbeiten sollen.

Um eine IP-Adresse von ihrem numerischen Wert in die punktierte Dezimaldarstellung umzuwandeln, steht die Funktion **inet_ntoa** zur Verfügung.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

gibt zurück: Punktierte Dezimaldarstellung zur numerischen IP-Adresse *in* (bei Erfolg)

inet_ntoa (*ntoa*=*network to ascii*) konvertiert die übergebene numerische IP-Adresse *in*, die in *network byte order* vorliegen muß, in die punktierte Dezimaldarstellung. Der zurückgegebene String wird in einem statisch allokierten Puffer abgelegt, der beim nächsten Aufruf von **inet_ntoa** wieder überschrieben wird.

Zum Extrahieren der Netzwerknummer aus einer in punktierte Dezimaldarstellung angegebenen IP-Adresse steht die Funktion **inet_network** zur Verfügung.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long int inet_network(const char *cp);
```

gibt zurück: numerische Netzwerknummer (in *host byte order*) zur IP-Adresse *cp* (bei Erfolg); -1 bei Fehler

inet_network extrahiert die Netzwerknummer aus der in punktierte Dezimaldarstellung übergebenen IP-Adresse *cp* und liefert deren numerischen Wert in *host byte order* als Rückgabe.

Zum Extrahieren der Netzwerknummer aus einer numerischen IP-Adresse steht die Funktion **inet_netof** zur Verfügung.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long int inet_netof(struct in_addr in);
```

gibt zurück: Netzwerknummer aus punktierte Dezimaldarstellung (in *host byte order*) zur numerischen IP-Adresse *in*

inet_netof extrahiert die Netzwerknummer (entsprechender Teil der punktierten Dezimaldarstellung) aus der übergebenen numerischen IP-Adresse *in* und liefert diesen Teil in *host byte order* als Rückgabewert.

Zum Extrahieren der Adresse des lokalen Rechners aus einer numerischen IP-Adresse steht die Funktion **inet_lnaof** zur Verfügung.


```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
unsigned long int inet_lnaof(struct in_addr in);
```

gibt zurück: Adreßteil des lokalen Rechners aus punktierter Dezimaldarstellung (in *host byte order*) zur IP-Adresse *in*

inet_lnaof ermittelt die Adresse des lokalen Rechners (entsprechender Teil der punktierten Dezimaldarstellung) aus der übergebenen numerischen IP-Adresse und liefert diesen in *host byte order* als Rückgabewert.

Um aus einer Netzwerknummer und einer lokalen Rechneradresse eine vollständige numerische IP-Adresse zu generieren, steht die Funktion **inet_makeaddr** zur Verfügung

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
struct in_addr inet_makeaddr(int net, int host);
```

gibt zurück: aus *net* und *host* zusammengesetzte numerische IP-Adresse

inet_makeaddr generiert aus einer Netzwerknummer (*net*) und einer lokalen Rechneradresse (*host*) eine vollständige numerische IP-Adresse, die als Rückgabewert geliefert wird. Beide Adressen müssen in *host byte order* übergeben werden.

Die in einigen der obigen Funktionen verwendete Struktur *in_addr* ist in *<netinet/in.h>* wie folgt definiert:

```
struct in_addr {
    unsigned long int s_addr;
}
```

Beispiel

Demonstrationsbeispiel zum Extrahieren und Konvertieren von IP-Adressen

Das folgende Programm 19.17 (*ipadr.c*) demonstriert die Anwendung der obigen Funktionen, indem es sie auf die als erstes Argument angegebene IP-Adresse anwendet.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "eighdr.h"
```

```

int
main(int argc, char *argv[])
{
    struct in_addr  gesamt_ip, netz_ip, lokal_ip;
    unsigned long   gesamt_num, netz_num, lokal_num;

    if (argc != 2)
        fehler_meld(FATAL, "usage ipadresse");

    printf("\nGesamte IP-Adresse: %s\n", argv[1]);
    inet_aton(argv[1], &gesamt_ip);
    printf("...%s = %ld\n", argv[1], gesamt_ip);
    printf("...%s = %ld\n", inet_ntoa(gesamt_ip), gesamt_ip);

    printf("\nNetzwerknummer der IP-Adresse: %lu\n",
           inet_netof(gesamt_ip));
    netz_ip.s_addr = netz_num = ntohl(inet_netof(gesamt_ip)<<8);
    printf("...%s = %lu\n", inet_ntoa(netz_ip), netz_num);
    netz_ip = inet_makeaddr(htonl(netz_num), 0);
    printf("...%s = %ld\n", inet_ntoa(netz_ip), netz_ip);

    printf("\nLokaler Teil der IP-Adresse:\n");
    lokal_ip.s_addr = lokal_num = ntohl(inet_lnaof(gesamt_ip));
    printf("...%s = %lu\n", inet_ntoa(lokal_ip), lokal_num);
    lokal_ip = inet_makeaddr(0, htonl(lokal_num));
    printf("...%s = %ld\n", inet_ntoa(lokal_ip), lokal_ip);

    printf("\nZusammengesetzte IP-Adresse:\n");
    gesamt_ip = inet_makeaddr(htonl(netz_num), htonl(lokal_num));
    printf("...%s = %ld\n\n", inet_ntoa(gesamt_ip), gesamt_ip);

    exit(0);
}

```

Programm 19.17 (ipadr.c): Demonstration der Manipulationsfunktionen für IP-Adressen

Nachdem man dieses Programm 19.17 (ipadr.c) kompiliert und gelinkt hat

```
cc -o ipadr ipadr.c fehler.c
```

kann man es starten, wie z.B.

```
$ ipadr 193.25.129.12
```

```

Gesamte IP-Adresse: 193.25.129.12
...193.25.129.12 = 209787329
...193.25.129.12 = 209787329

```

```

Netzwerknummer der IP-Adresse: 12654977
...193.25.129.0 = 8460737
...193.25.129.0 = 8460737

```

```

Lokaler Teil der IP-Adresse:
...0.0.0.12 = 201326592

```

```
...0.0.0.12 = 201326592
```

```
Zusammengesetzte IP-Adresse:  
...193.25.129.12 = 209787329
```

```
$
```

19.7.5 Das Domain-Name-System (DNS)

Einem Rechner ist in einem Netzwerk immer eine eindeutige Nummer zugeteilt. Da sich Menschen Nummern nicht so leicht merken können wie Namen, wird an die Nummern zusätzlich noch ein Name vergeben. Der aus Host- und Domainname zusammengesetzte Name identifiziert einen Rechner innerhalb eines Netzwerks, wie etwa `elefant.saugtier.network`. Der `Hostname` (`elefant`) bezeichnet den einzelnen Rechner und der `Domain-Name` (`saugtier.network`) das Netzwerk, in dem sich der Rechner befindet.

Hat man nur einen (nicht vernetzten) Rechner kann man für den Domain-Namen zwei beliebige Namen verwenden.

Wenn man für einen Internetanschluß einen weltweit gültigen Domain-Namen benötigt, bekommt man die erforderlichen Daten von einem sogenannten Internet-Provider, der über einen vollwertigen Internetanschluß verfügt, oder in Absprache mit dem NIC (*Network Information Center*; <http://www.nic.de>).

19.7.6 Name-Server

Der Name-Server ist ein Rechner, der für die Umsetzung zwischen Rechnernamen und IP-Nummern zuständig ist. Bei kleinen Netzen sind die lokalen IP-Nummern meist in Form einer Tabelle in einer bestimmten Datei (meist `/etc/hosts`) hinterlegt, wie z.B.

```
#
# hosts          This file describes a number of hostname-to-address
#                mappings for the TCP/IP subsystem.  It is mostly
#                used at boot time, when no name servers are running.
#                On small systems, this file can be used instead of a
#                "named" name server.  Just add the names, addresses
#                and any aliases to this file...
#
127.0.0.1        localhost
193.25.29.100    berlinw.winet.sta      berlinw, hauptstadt
193.25.29.12     herold.sta.erl.siemens.de herold
193.25.29.130    berlin2.linet.sta      berlin2
193.25.29.140    capital.linet.sta      capital
193.25.29.19     server1.winet.sta      server1
194.95.193.10    fen.baynet.de          fen
```

Bei größeren Netzen, wie z.B. dem Internet, werden diese Daten dagegen meist in eigenen Datenbanken gehalten. Gibt man z.B. den Namen eines Servers in Finnland an, sucht der Name-Server in seiner Datenbank dessen IP-Adresse. Findet er ihn dort nicht, kontaktiert einen anderen Name-Server, was natürlich einige Zeit dauern kann.

Für Informationen zu Rechnern im Netz steht die in `<netdb.h>` definierte Struktur `hostent` zur Verfügung:

```
struct hostent {
    char *h_name;           /* Offizielle Name des Rechners */
    char **h_aliases;       /* Aliasliste */
    int h_addrtype;         /* Host-Adresstyp (AF_INET bei IPv4) */
    int h_length;           /* Laenge der Adresse */
    char **h_addr_list;     /* List von Adresse vom Nameserver */
};
```

Die Komponente `h_aliases` ist ein Stringarray, das eventuell vorhandene Aliasnamen enthält, wobei das letzte Element in diesem Array immer ein NULL-Zeiger ist.

Die Komponente `h_addrtype` gibt den Adreßtyp an, was `AF_INET` bei *IPv4* ist. Anwendungen, die *IPv6* verwenden, verwenden hier einen anderen Adreßtyp.

Die Komponente `h_length` gibt dabei die Länge der numerischen Adresse an. Bei `AF_INET` muß hier also `sizeof(struct in_addr)` angegeben werden.

Die Komponente `h_addr_list` ist ein Array von Zeigern auf die Adressen für den entsprechenden Rechner, wobei das letzte Element in diesem Array immer ein NULL-Zeiger ist. Bei `AF_INET` zeigt jeder Zeiger in diesem Array (außer dem letzten) auf einen Speicherplatz mit dem Datentyp `struct in_addr`.

Zum Erfragen von Informationen zu einem Rechner stehen die beiden Funktionen ***gethostbyname*** und ***gethostbyaddr*** zur Verfügung.

```
#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(const char *name);

gibt zurück: Zeiger auf struct hostent des gefundenen Rechners (bei Erfolg); NULL bei Fehler
```

```
#include <netdb.h>
#include <sys/socket.h> /* für AF_INET */

extern int h_errno;

struct hostent *gethostbyaddr(const char *addr, int len, int type);

gibt zurück: Zeiger auf struct hostent des gefundenen Rechners (bei Erfolg); NULL bei Fehler
```

gethostbyname liefert zum Rechnernamen `name` und ***gethostbyaddr*** liefert zum Rechner mit IP-Adresse `addr` die entsprechende `struct hostent`-Information.

Beide geben also einen Zeiger auf die Struktur `hostent` zurück. Der zugehörige Speicherplatz wird statisch von den beiden Funktionen allokiert, was bedeutet, daß ein nachfolgender Aufruf der jeweiligen Funktion diesen Speicherplatz wieder überschreibt.

Die Funktion ***gethostbyaddr*** hat drei Parameter, die zusammen die entsprechende Adresse bilden. Der erste Parameter `addr` (`struct in_addr *addr` bei `AF_INET`) legt die IP-Adresse fest, der nächste Parameter `len` gibt die Länge dieser Adresse an und der letzte Parameter `type` spezifiziert den Typ der Adresse, was `AF_INET` bei *IPv4* ist.

Tritt bei der Suche nach einem Rechner ein Fehler auf, wird die entsprechende Fehlernummer von der betreffenden Funktion in die globale Variable `h_errno` geschrieben. Die zugehörige Fehlermeldung kann man sich – ähnlich zu ***perror*** – mit der Funktion ***h_error*** ausgeben lassen.

```
#include <netdb.h>

void herror(const char *s);
```

Hinweis

Viele Programme überprüfen ganz gezielt nur auf die Fehlernummer `NETDB_INTERNAL`, was auf einen fehlerhaften Aufruf einer Systemfunktion hinweist. In diesem Fall enthält `errno` dann den Grund des Fehlers.

Beispiel

Ausgeben der Informationen zu einem Rechnernamen bzw. einer IP-Adresse

Das folgende Programm 19.18 (`netzhost.c`) erwartet auf der Kommandozeile ein Argument, das entweder ein Rechnernamen, ein Aliasname oder eine IP-Adresse ist. Es gibt dann alle Informationen, die es zu dem entsprechenden Rechner ermitteln kann (aus Struktur `hostent`), auf der Standardausgabe aus.

```
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    struct hostent  *rechner;
    struct in_addr  ip_adr, **adr_zgr;
    char            **zgr;
    int             erst = 1;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s rechnername|ip-adresse", argv[0]);
```

```

if (inet_aton(argv[1], &ip_adr) != 0)
    rechner = gethostbyaddr((char *) &ip_adr, sizeof(ip_adr), AF_INET);
else
    rechner = gethostbyname(argv[1]);

if (rechner == NULL) {
    perror("Fehler beim Suchen des Rechners");
    exit(1);
}

printf(".....Offizieller Hostname: %s\n", rechner->h_name);

printf(".....Aliase:");
for (zgr = rechner->h_aliases; *zgr != NULL; zgr++) {
    printf("%s%s", (erst==1) ? " " : ", ", *zgr);
    erst = 0;
}
if (erst == 1)
    printf(" keine vorhanden");
printf("\n");

printf(".....IP-Adressen:");
adr_zgr = (struct in_addr **)rechner->h_addr_list;
erst = 1;
for ( ; *adr_zgr != NULL; adr_zgr++) {
    printf("%s%s", (erst==1) ? " " : ", ", inet_ntoa(**adr_zgr));
    erst = 0;
}
printf("\n\n");

exit(0);
}

```

Programm 19.18 (netzhost.c): Ausgeben der Informationen zu einem Rechnernamen bzw. einer IP-Adresse

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o netzhost netzhost.c fehler.c
```

kann man es starten, wie z.B.:

```

$ netzhost 193.25.29.100
.....Offizieller Hostname: berlinw.winet.sta
.....Aliase: berlinw, hauptstadt, head
.....IP-Adressen: 193.25.29.100

$ netzhost hauptstadt
.....Offizieller Hostname: berlinw.winet.sta
.....Aliase: berlinw, hauptstadt, head
.....IP-Adressen: 193.25.29.100

$ netzhost berlinw.winet.sta
.....Offizieller Hostname: berlinw.winet.sta
.....Aliase: berlinw, hauptstadt, head

```

```

.....IP-Adressen: 193.25.29.100

$ netzhos fen
.....Offizieller Hostname: fen.baynet.de
.....Aliase: fen
.....IP-Adressen: 194.95.193.10

$ netzhos hallo
Fehler beim Suchen des Rechners: Unknown host

$

```

19.7.7 Informationen zu Port-Nummern

Der Internetstandard schreibt einen gewissen Satz von Port-Nummern vor, die von der *Internet Assigned Numbers Authority (IANA)*; <http://www.iana.org> verwaltet werden.

Die den entsprechenden Diensten und Protokollen zugeteilten Port-Nummern sind in der Datei `/etc/services` angegeben.

Der Zugriff auf diese Datei erfolgt üblicherweise mit der Funktion ***getservbyname***.

Für Informationen zu den entsprechenden Diensten steht die in `<netdb.h>` definierte Struktur `servent` zur Verfügung:

```

struct servent {
    char *s_name;      /* Offizieller Servicename */
    char **s_aliases;  /* Aliasliste */
    int s_port;        /* Port-Nummer */
    char *s_proto;     /* zu verwendendes Protokoll */
}

```

Die Komponente `s_aliases` ist ein Stringarray, das eventuell vorhandene Aliasnamen enthält, wobei das letzte Element in diesem Array immer ein `NULL`-Zeiger ist.

Die Komponente `s_port` enthält die Port-Nummer (in *network byte order*) und die Komponente `s_proto` enthält den Namen des Protokolls für diesen Dienst.

Zum Erfragen von Informationen zu einzelnen Diensten stehen die beiden Funktionen ***getservbyname*** und ***getservbyport*** zur Verfügung.

```

#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
    gibt zurück: Zeiger auf struct servent des gefundenen Rechners (bei Erfolg); NULL bei Fehler

struct servent *getservbyport(int port, const char *proto);
    gibt zurück: Zeiger auf struct servent des gefundenen Rechners (bei Erfolg); NULL bei Fehler

```

getservbyname liefert zum Dienst `name`, der das Protokoll `proto` benutzt, die entsprechende `struct-servent`-Information. **getservbyport** liefert zum Dienst mit der Port-Nummer `port` (in *network byte order*), der das Protokoll `proto` benutzt, die entsprechende `struct-servent`-Information.

Beide Funktionen geben also einen Zeiger auf die Struktur `servent` zurück. Der zugehörige Speicherplatz wird statisch von den beiden Funktionen allokiert, was bedeutet, daß nachfolgende Aufrufe der jeweiligen Funktion diesen wieder überschreiben.

Tritt bei der Suche nach einem Dienst ein Fehler auf, wird die entsprechende Fehlernummer von der betreffenden Funktion in die globale Variable `h_errno` geschrieben. Die zugehörige Fehlermeldung kann man sich wieder mit der Funktion **h_error** ausgeben lassen.

Neben diesen beiden Funktionen werden noch drei weitere Funktionen angeboten, mit denen man die Datei `/etc/services` zeilenweise durchlaufen kann.

```
#include <netdb.h>

struct servent *getservent(void);

    gibt zurück: Zeiger auf struct servent des gefundenen Rechners (bei Erfolg); NULL bei Fehler oder Datei-
                                     ende (EOF)

void setservent(int stayopen);

void endservent(void);
```

Die Funktion **setservent** öffnet die Datei `/etc/services` und positioniert den Schreib-/Lesezeiger auf den ersten relevanten Eintrag in dieser Datei. Wird für den Parameter `stayopen` ein Wert verschieden von 0 (TRUE) angegeben, dann wird die Datei `/etc/services` bei Aufrufen der Funktionen **getservbyname** und **getservbyport** nicht geschlossen, was bei der Angabe von 0 sehr wohl der Fall ist.

Die Funktion **getservent** liest den aktuellen Eintrag (Zeile) aus der Datei `/etc/services` und gibt die zugehörige `servent`-Information zurück. Sie positioniert den Schreib-/Lesezeiger auf den nächsten Eintrag, so daß beim nächsten **getservent**-Aufruf dieser gelesen wird.

Die Funktion **endservent** schließt die Datei `/etc/services`.

Nachfolgend sind zwei Programmbeispiele zu den obigen Funktionen gegeben.

Beispiel

Ausgeben aller Dienste aus der Datei /etc/services

```
#include <netdb.h>
#include <netinet/in.h>
```



```

int
main(void)
{
    struct servent *service;
    char          **zgr;
    int           erst;

    setservent(1);

    while ( (service = getservent()) != NULL) {
        printf("Service: %-12s, ", service->s_name);
        printf("Port: %-5d, ", ntohs(service->s_port));
        printf("Protokoll: %s, ", service->s_proto);
        erst = 1;
        for (zgr = service->s_aliases; *zgr != NULL; zgr++) {
            printf("%s%s", (erst==1) ? "Aliase: " : ", ", *zgr);
            erst = 0;
        }
        printf("\n");
    }

    endservent();

    if (h_errno != 0) {
        perror("Fehler beim Suchen des Service");
        exit(1);
    }

    exit(0);
}

```

Programm 19.19 (**alleserv.c**): Ausgeben aller in */etc/services* vorhandenen Dienste

Beispiel

Ausgeben aller verfügbaren Informationen zu einem Dienst

Das folgende Programm 19.20 (**ein_serv.c**) sucht zu den auf der Kommandozeile angegebenen String den entsprechenden offiziellen Namen oder ein entsprechendes Alias und gibt die zugehörige Information aus. Der Benutzer kann zusätzlich als zweites Argument noch das Protokoll angeben. Gibt der Benutzer beim Aufruf kein Protokoll an, so nimmt dieses Programm das Protokoll »tcp« für die Suche.

```

#include <netdb.h>
#include <netinet/in.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    struct servent *service;
    char          **zgr, *default_proto = "tcp", *proto;
    int           erst = 1;

```

```

if (argc != 2 && argc != 3)
    fehler_meld(FATAL, "usage: %s service [protokoll]", argv[0]);

proto = (argc == 3) ? argv[2] : default_proto;
if ( (service = getservbyname(argv[1], proto)) == NULL) {
    perror("Fehler beim Suchen dieses Services");
    exit(1);
}

printf(".....Service : %s\n", service->s_name);
printf(".....Port      : %d\n", ntohs(service->s_port));
printf(".....Aliase      :");
for (zgr = service->s_aliases; *zgr != NULL; zgr++) {
    printf("%s%s", (erst==1) ? " " : ", ", *zgr);
    erst = 0;
}
if (erst == 1)
    printf(" keine vorhanden");
printf("\n");
printf(".....Protokoll: %s\n", service->s_proto);

exit(0);
}

```

Programm 19.20 (ein_serv.c): Ausgeben aller verfügbaren Informationen zu einem Dienst

Nachdem man das Programm 19.20 (ein_serv.c) kompiliert und gelinkt hat

```
cc -o ein_serv ein_serv.c fehler.c
```

kann man es aufrufen, wie z.B.:

```

$ ein_serv route udp
.....Service : route
.....Port      : 520
.....Aliase     : router, routed
.....Protokoll: udp

$ ein_serv smtp
.....Service : smtp
.....Port      : 25
.....Aliase     : mail
.....Protokoll: tcp

$ ein_serv name
.....Service : nameserver
.....Port      : 42
.....Aliase     : name
.....Protokoll: tcp

$

```

19.7.8 Beispielprogramme zur Netzwerkprogrammierung mit TCP/IP

Hier wird ein Beispiel zur Netzwerkprogrammierung mit TCP/IP gegeben. Dazu wird ein einfacher Server für TCP/IP-Sockets entwickelt, der auf einem beliebigen Rechner im Netzwerk ablaufen kann. Die Aufgabe dieses Servers ist es, auf eine Verbindungsanforderung auf Port 2233 seitens eines Clients von einem anderen Rechner im Netz zu warten. Ist die Verbindung hergestellt, liest der Server zunächst den vom Client geschickten Namen der Datei (als eine Zeile) aus dem Socket, legt eine neue leere Datei dieses Namens auf dem Rechner an, an dem er abläuft. Danach kopiert der Server alle Daten, die er aus dem Socket liest, in diese Datei. Wenn die Gegenseite (der Client) die Verbindung beendet, schließt auch der Server die Verbindung und die neu erzeugte Datei. Anschließend wartet er auf eine neue Verbindungsanforderung.

Mit den folgenden beiden Programmen `tcpip_server.c` und `tcpip_client.c` ist es also möglich, Dateien in einem Netzwerk von einem Rechner auf einen anderen Rechner zu kopieren.

Beispiel

Server zum Kopieren von Dateien in einem Netzwerk

Das Warten auf TCP-Verbindungen entspricht weitgehend dem Warten auf Unix-Domain-Verbindungen. Der einzige Unterschied sind die Protokoll- und Adreßfamilien, wie das folgende Listing des Server-Programms (`tcpip_server.c`) zum Kopieren von Dateien in einem Netzwerk zeigt.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <arpa/inet.h>
4  #include <netinet/in.h>
5  #include <sys/socket.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <unistd.h>
10 #include "eighdr.h"
11
12 #define PORT_NUMMER 2233
13
14 #ifndef MSG_DATA
15 #define MSG_DATA(cmsg) ((cmsg)->msg_data)
16 #endif
17
18 int
19 main(void)
20 {
21     int          sockfd, connfd, fd, i, j, n, ngesamt;
22     struct sockaddr_in  adresse;
23     size_t       adrlaenge = sizeof(struct sockaddr_in);
```

```
24 char                puffer[MAX_ZEICHEN];
25
26 if ( (sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
27     fehler_meld(FATAL_SYS, "Fehler beim socket-Aufruf");
28
29 i = 1;
30 setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
31
32 adresse.sin_family = AF_INET;
33 adresse.sin_port    = htons(PORT_NUMMER);
34 memset(&adresse.sin_addr, 0, sizeof(adresse.sin_addr));
35
36 if (bind(sockfd, (struct sockaddr *) &adresse, sizeof(adresse)))
37     fehler_meld(FATAL_SYS, "Fehler beim bind-Aufruf");
38
39 if (listen(sockfd, 5))
40     fehler_meld(FATAL_SYS, "Fehler beim listen-Aufruf");
41
42 while ( (connfd = accept(sockfd, (struct sockaddr *)&adresse,
43                         &adrlaenge)) >= 0) {
44     printf(".....Datenempfang ");
45
46     /*----- Lesen des geschickten Dateinamens -----*/
47     j = 0;
48     while ( (n = read(connfd, &puffer[j], 1)) > 0) {
49         if (puffer[j] == '\n') {
50             puffer[j] = 0;
51             break;
52         }
53         j++;
54     }
55     if (n < 0)
56         fehler_meld(FATAL_SYS, "Fehler beim read-Aufruf");
57
58     printf("fuer Datei '%s' ", puffer);
59
60     /*----- Oeffnen der entsprechenden Datei zum Schreiben -----*/
61     if ( (fd = open(puffer, O_WRONLY|O_CREAT|O_TRUNC, 0644)) < 0) {
62         fehler_meld(WARNUNG_SYS, "kann '%s' nicht oeffnen", puffer);
63         close(connfd);
64         continue;
65     }
66
67     /*----- Lesen der geschickten Daten -----*/
68     ngesamt = 0;
69     while ( (n = read(connfd, puffer, sizeof(puffer))) > 0) {
70         if (write(fd, puffer, n) != n)
71             fehler_meld(FATAL_SYS, "Fehler beim write-Aufruf");
72         ngesamt += n;
73     }
74     if (n < 0)
75         fehler_meld(FATAL_SYS, "Fehler beim read-Aufruf");
76 }
```

```

77     printf("..beendet (%d Bytes)\n", ngesamt);
78     close(fd);
79     close(connfd);
80 }
81
82 if (connfd < 0)
83     fehler_meld(FATAL_SYS, "Fehler beim accept-Aufruf");
84
85 close(sockfd);
86
87 exit(0);
88 }

```

Programm 19.21 (**tcpip_server.c**): Server zum Kopieren von Dateien in einem Netzwerk

Die IP-Adresse, an die der Socket gebunden wird, spezifiziert beim Server-Programm eine Port-Nummer und keine wirkliche IP-Adresse.

Einer genaueren Erläuterung bedürfen noch die beiden folgenden Zeilen aus dem Listing des Programms 19.21 (**tcpipc_server.c**).

```

29     i = 1;
30     setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));

```

Die TCP-Implementierung der meisten Unix-Systeme machen üblicherweise Einschränkungen bezüglich der Wiederbenutzung eines Verbindungspunktes (lokaler Port am lokalen Rechner). So gilt z.B. für TCP-Ports, daß diese erst nach zwei Minuten wieder verwendet werden können. Mit der Option **SO_REUSEADDR** beim Aufruf der Funktion **setsockopt** wird nun festgelegt, daß diese Einschränkung aufzuheben ist und der entsprechende TCP-Port innerhalb einer kurzen Zeit wieder benutzt werden kann.

Zum Setzen und Erfragen von Optionen für Sockets stehen die beiden Funktionen **setsockopt** und **getsockopt** zur Verfügung:

```

#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int socket, int level, int optname,
               const void *optval, int optlen);

int getsockopt(int socket, int level, int optname,
               void *optval, int *optlen);

```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

Der erste Parameter **socket** spezifiziert den Socket, dessen Optionen zu setzen sind.

Der zweite Parameter **level** legt den Typ der entsprechenden Option fest: **SOL_SOCKET** z.B. weist auf eine allgemeine Socket-Option hin. Weitere Informationen hierzu können mit **man setsockopt** bzw. **man getsockopt** erfragt werden.

Der dritte Parameter `option` gibt die zu setzende bzw. die zu erfragende Option an. Die Vielzahl der verfügbaren Optionen können wieder mit **man setsockopt** bzw. **man getsockopt** erfragt werden.

Der Parameter `optval` zeigt auf die zu setzende bzw. zu erfragende Option.

Im letzten Parameter `optlen` wird bei **setsockopt** die Länge des zu setzenden Optionswerts (`optval`) angegeben. Bei **getsockopt** wird hier eine Adresse angegeben, an die diese Funktion die Länge des Optionswerts schreibt, den sie an der Adresse `optval` hinterlegt hat.

Beim Aufruf

```
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
```

wird ein Zeiger auf die Variable `i` (`&i`) übergeben. Da `i` auf einen Wert verschieden von 0 gesetzt ist, bedeutet dies, daß die Option `SO_REUSEADDR` aktiviert wird.

Beispiel

Client zum Kopieren von Dateien in einem Netzwerk

Das folgende Programm 19.22 (`tcip_client.c`) ist das Clientprogramm zur Kommunikation mit dem obigen Serverprogramm (`tcip_server.c`).

Es erwartet als erstes Argument den Namen oder die IP-Adresse des Rechners, auf dem das Serverprogramm `tcip_server.c` gerade läuft. Als weitere Argumente sind die Namen der Dateien anzugeben, die auf diesen entfernten Rechner zu kopieren sind.

```

1  #include <stdio.h>
2  #include <netdb.h>
3  #include <arpa/inet.h>
4  #include <netinet/in.h>
5  #include <sys/socket.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <fcntl.h>
9  #include <unistd.h>
10 #include "eighdr.h"
11
12 #define PORT_NUMMER 2233
13
14 #ifndef CMSG_DATA
15 #define CMSG_DATA(cmsg) ((cmsg)->cmsg_data)
16 #endif
17
18 int
19 main(int argc, char *argv[])
20 {
21     int                sockfd, i, n, fd, name_len;
22     struct sockaddr_in  adresse;
23     struct in_addr      inadr;
```

```
24 struct hostent      *rechner;
25 char                puffer[MAX_ZEICHEN];
26
27 if (argc < 3)
28     fehler_meld(FATAL, "usage: %s rechner datei(en)", argv[0]);
29
30 if (inet_aton(argv[1], &inadr))
31     rechner = gethostbyaddr((char *) &inadr, sizeof(inadr), AF_INET);
32 else
33     rechner = gethostbyname(argv[1]);
34
35 if (rechner == NULL) {
36     perror("Fehler beim Suchen des Rechners");
37     exit(1);
38 }
39
40 adresse.sin_family = AF_INET;
41 adresse.sin_port   = htons(PORT_NUMMER);
42 memcpy(&adresse.sin_addr, rechner->h_addr_list[0],
43        sizeof(adresse.sin_addr));
44
45 for (i=2; i<argc; i++) {
46
47     if ( (sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
48         fehler_meld(FATAL_SYS, "Fehler beim socket-Aufruf");
49
50     if (connect(sockfd, (struct sockaddr *) &adresse, sizeof(adresse)))
51         fehler_meld(FATAL_SYS, "Fehler beim connect-Aufruf");
52
53     if ( (fd = open(argv[i], O_RDONLY)) < 0) {
54         fehler_meld(WARNUNG_SYS, "kann Datei '%s' nicht oeffnen", argv[i]);
55         continue;
56     }
57     strcpy(puffer, argv[i]);
58     strcat(puffer, "\n");
59     name_len = strlen(puffer);
60
61     if (write(sockfd, puffer, name_len) != name_len) {
62         fehler_meld(WARNUNG_SYS, "Fehler beim Schicken des "
63                    "Namens der Datei '%s'", argv[i]);
64         close(fd);
65         close(sockfd);
66         continue;
67     }
68
69     while ( (n = read(fd, puffer, sizeof(puffer))) > 0)
70         if (write(sockfd, puffer, n) != n)
71             fehler_meld(FATAL_SYS, "Fehler beim write-Aufruf");
72     if (n < 0)
73         fehler_meld(FATAL_SYS, "Fehler beim read-Aufruf");
74
75     close(sockfd);
76 }
```

```

77
78     exit(0);
79 }
```

Programm 19.22 (tcpip_client.c): Client zum Kopieren von Dateien in einem Netzwerk

Nachdem man auf dem Zielrechner das Server-Programm `tcpip_server.c` kompiliert und gelinkt hat

```
cc -o tcpip_server tcpip_server.c fehler.c
```

kann man es dort starten:

```

$ hostname -i
193.25.29.100
$ tcpip_server &
$
```

Nachdem man auf dem Rechner, von dem kopiert werden soll, das Client-Programm `tcpip_client.c` kompiliert und gelinkt hat

```
cc -o tcpip_client tcpip_client.c fehler.c
```

kann man es dort zum Kopieren von Dateien auf den entfernten Rechner, auf dem das Server-Programm läuft, verwenden.

```

$ hostname -i
193.25.29.12
$ ls *.c
alleserv.c      fehler.c      openser3.c    tcpip_client.c
bsd44_cs.c      ipadr.c       sockclie.c    tcpip_server.c
ein_serv.c      netzhost.c    sockserv.c
$ tcpip_client 193.25.29.100 *.c
$
```

Am Zielrechner, auf dem das Server-Programm abläuft, werden dann die Dateien empfangen und im Working-Directory abgelegt, was das Server-Programm durch folgende Ausgaben meldet:

```

.....Datenempfang fuer Datei 'bsd44_cs.c' ..beendet (3317 Bytes)
.....Datenempfang fuer Datei 'ein_serv.c' ..beendet (941 Bytes)
.....Datenempfang fuer Datei 'fehler.c' ..beendet (2783 Bytes)
.....Datenempfang fuer Datei 'ipadr.c' ..beendet (1277 Bytes)
.....Datenempfang fuer Datei 'netzhost.c' ..beendet (1224 Bytes)
.....Datenempfang fuer Datei 'openser3.c' ..beendet (7777 Bytes)
.....Datenempfang fuer Datei 'sockclie.c' ..beendet (1170 Bytes)
.....Datenempfang fuer Datei 'sockserv.c' ..beendet (1450 Bytes)
.....Datenempfang fuer Datei 'tcpip_client.c' ..beendet (2135 Bytes)
.....Datenempfang fuer Datei 'tcpip_server.c' ..beendet (2473 Bytes)
```

Als letztes sei noch das Kommando **socklist** erwähnt, mit dem man sich alle momentan eingerichteten Sockets anzeigen lassen kann.

19.8 Übung

19.8.1 Parallele Matrizenmultiplikation durch mehrere Kindprozesse

Erstellen Sie ein Programm `matmult2.c`, das eine Multiplikation von zwei Matrizen durchführt. Für dieses Programm soll folgendes eingehalten werden:

- ▶ Die Deklarationen aller Matrizen können modulglobal sein.
- ▶ Für jedes Element der Ergebnismatrix ist ein Kindprozeß zu kreieren, dem über eine Stream Pipe der Zeilen- und Spaltenindex des zu berechnenden Elements der Ergebnismatrix mitgeteilt wird. Nachdem der jeweilige Kindprozeß diese Indizes aus der Stream Pipe gelesen hat, muß er – unter Zugriff auf die modulglobalen Eingabematrizen – dieses Element berechnen und dem Elternprozeß über die gleiche Stream Pipe dieses Ergebniselement zukommen lassen.
- ▶ Der Elternprozeß gibt zunächst die beiden Eingabematrizen aus und wartet dann auf die Ankunft aller Ergebnisse (aus den Stream Pipes), bevor er die vollständige Ergebnismatrix ausgibt.

20 Terminal-E/A

*Das ist nicht das Ende.
Nicht einmal der Anfang vom Ende.
Aber es ist vielleicht das Ende vom Anfang.*
Churchill

Der Begriff *Terminalsteuerung* umfaßt alle Funktionen zur Steuerung und Programmierung der seriellen Schnittstellen (seriellen Ports) eines Rechners und des Terminaltreibers des Betriebssystems. An den seriellen Ports können neben Terminals auch Modems, Drucker usw. angeschlossen sein.

In diesem Kapitel werden alle POSIX.1-Terminalfunktionen und einige zusätzliche Funktionen vorgestellt, die von SVR4 und BSD-Unix angeboten werden.

Zudem stellt dieses Kapitel die Bibliotheken `curses` und `S-Lang` vor, mit denen Semigraphikprogrammierung unter Linux/Unix möglich ist. Des weiteren werden hier die Eigenschaften einer Linux-Konsole detaillierter vorgestellt, bevor am Ende dieses Kapitels noch auf die Programmierung von virtuellen Konsolen unter Linux eingegangen wird.

20.1 Charakteristika eines Terminals im Überblick

Bevor in den nächsten Abschnitten detailliert auf die Eigenschaften und Einstellungsmöglichkeiten eines Terminals eingegangen wird, wird ein kurzer Überblick über diese gegeben.

20.1.1 Terminalmodi

Für ein Terminal gibt es zwei unterschiedliche Modi.

1. Zeilenorientierter Modus (*Canonical Mode*)

In diesem Modus werden nur ganze Zeilen und nicht einzelne Zeichen verarbeitet. Bei jeder Leseanforderung liefert der Terminaltreiber immer eine ganze Zeile. Dies ist die Voreinstellung für ein Terminal.

2. Zeichenorientierter Modus (*Noncanonical Mode*)

In diesem Modus wird jedes einzelne eingegebene Zeichen direkt vom Terminaltreiber geliefert. Der Terminal wartet in diesem Modus nicht auf ein Zeilenendezeichen, um

dann die ganze Zeile zu liefern, sondern liefert jedes eingegebene Zeichen sofort. Dieser Modus wird z.B. bei einem bildschirmorientierten Programm wie dem Editor **vi** gebraucht, da viele **vi**-Kommandos nicht mit Return abgeschlossen, sondern direkt nach der entsprechenden Eingabe wirksam werden. Eine weitere Eigenschaft dieses Modus ist, daß die Sonderbedeutung von einigen Terminalsteuerzeichen ausgeschaltet ist. Im **vi** bedeutet z.B. *Strg-D* nicht EOF, sondern »halbe Bildschirmseite weiterblättern«.

20.1.2 Eingabe- und Ausgabepuffer eines Terminals

Zu jedem Terminal existiert ein Eingabe- und ein Ausgabepuffer, für die folgendes gilt:

1. Die Größe des Eingabepuffers ist durch die Konstante `MAX_INPUT` festgelegt. Das Verhalten eines Systems bei einem vollen Eingabepuffer ist implementierungsabhängig. Meistens wird dies durch ein akustisches Signal angezeigt. Neben `MAX_INPUT` gibt es die Konstante `MAX_CANON`, die die maximale Anzahl von Bytes festlegt, die der Eingabepuffer im zeilenorientierten Modus (*Canonical Mode*) aufnehmen kann.
2. Obwohl auch der Ausgabepuffer nur eine begrenzte Anzahl von Bytes aufnehmen kann, ist für die Größe dieses Puffers keine Konstante definiert. Dies ist auch nicht notwendig, da bei einem vollen Ausgabepuffer der Kern den dorthin schreibenden Prozeß solange suspendiert, bis dort wieder Platz ist.
3. Wenn das `echo`-Flag eingeschaltet ist, wird jedes eingegebene Zeichen nicht nur im Eingabepuffer, sondern auch gleichzeitig im Ausgabepuffer abgelegt.
4. Um den ganzen Eingabe- oder Ausgabepuffer zu leeren (Lesen oder Schreiben), steht die Funktion ***tcflush*** zur Verfügung.

20.1.3 Struktur `termios`

Alle Attribute eines Terminals, die man abfragen oder setzen kann, sind in der Struktur `termios`, die in der Headerdatei `<termios.h>` definiert ist, enthalten.

```
struct termios {
    tcflag_t c_iflag;    /* Eingabeflags */
    tcflag_t c_oflag;    /* Ausgabeflags */
    tcflag_t c_cflag;    /* Kontrollflags */
    tcflag_t c_lflag;    /* Lokale Flags */
    cc_t      c_line;    /* line discipline; wird nur in sehr
                        systemspezifischen Anwendungen benutzt */
    cc_t      c_cc[NCCS]; /* Steuerzeichen */
}
```

In den nächsten Abschnitten werden diese Komponenten ausführlich besprochen. Der Datentyp `tcflag_t` ist meist als **unsigned long** definiert. Das Array `c_cc` enthält alle Steuerzeichen, die erfragt oder geändert werden können. Der Datentyp `cc_t` ist meist als **unsigned char** definiert. `NCCS` legt die Anzahl der Elemente im `c_cc`-Array fest. Der Wert von `NCCS` liegt üblicherweise zwischen 11 und 18; POSIX.1 definiert 11 Steuerzeichen, aber die meisten Unix-Systeme bieten zusätzliche Steuerzeichen an.

20.1.4 Spezielle Eingabezeichen

POSIX.1 definiert 11 verschiedene spezielle Eingabezeichen. SVR4 kennt 6 weitere und BSD-Unix 7 weitere spezielle Eingabezeichen. Tabelle 20.1 gibt eine Kurzbeschreibung dieser speziellen Eingabezeichen.

Name	Beschreibung	c_cc - Index	Eingeschaltet durch Flag	Typ. Wert	POSIX.1	Erweiterung SVR4 BSD	
CR	Carriage Return (Wagenrücklauf)	-	ICANON	\r	x		
DISCARD	discard output (Ausgabe wegwerfen)	VDISCARD	IEXTEN	Strg-O		x	x
DSUSP	delayed syspen (Suspendieren nur beim Lesen vom Kontrollterminal; Signal SIGTSTP)	VDSUSP	ISIG	Strg-Y		x	x
EOF	end-of-file (Dateiende)	VEOF	ICANON	Strg-D	x		
EOL	end-of-line (Zeilenende)	VEOL	ICANON		x		
EOL2	alternate end-of-line (alternatives Zeilenende)	VEOL2	ICANON			x	x
ERASE	backspace one character (letztes Zeichen löschen)	VERASE	ICANON	Strg-H Strg-?	x		
INTR	interrupt signal SIGINT (Unterbrechungs-Signal)	VINTR	ISIG	Strg-C	x		
KILL	erase line (Zeile löschen)	VKILL	ICANON	Strg-U	x		
LNEXT	Literal Text (Ausschalten spezieller Zeichen)	VLNEXT	IEXTEN	Strg-V		x	x
NL	linefeed (Neue Zeile)	-	ICANON	\n	x		
QUIT	quit signal SIGQUIT (Abbruch mit Speicherabzug)	VQUIT	ISIG	Strg-\	x		
REPRINT	reprint all input (Eingabezeichen neu ausgeben)	VREPRINT	ICANON	Strg-R		x	x
START	resume output (Ausgabe fortsetzen)	VSTART	IXON/IXOFF	Strg-Q	x		

Tabelle 20.1: Spezielle Eingabezeichen

Name	Beschreibung	c_cc - Index	Eingeschaltet durch Flag	Typ. Wert	POSIX.1	Erweiterung SVR4 BSD	
STATUS	status request (Statusinformation anfordern)	VSTATUS	ICANON	Strg-T			x
STOP	stop output (Ausgabe anhalten)	VSTOP	IXON/IXOFF	Strg-S	x		
SUSP	suspend signal SIGTSTP (Suspendieren von Prozessen)	VSUSP	ISIG	Strg-Z		x	x
WERASE	backspace one word (letztes Wort löschen)	VWERASE	ICANON	Strg-W		x	x

Tabelle 20.1: Spezielle Eingabezeichen

Nur die beiden Zeichen `START` und `STOP` werden durch ein Flag in der Komponente `c_iflag` in der Struktur `termios` eingeschaltet. Alle anderen Zeichen in der Tabelle 20.1 werden durch ein Flag in der Komponente `c_lflag` in der Struktur `termios` eingeschaltet. Bis auf die beiden speziellen Eingabezeichen `CR` und `NL` können alle anderen speziellen Eingabezeichen geändert werden. Dazu muß der entsprechende Eintrag im Array `c_cc` der Struktur `termios` geändert werden, wie z.B.

```
struct termios    terminal;
.....
terminal.c_cc[VEOF] = 6; /* ASCII-Code 6 = Ctrl-F */
```

Meist ist in `<termios.h>` ein eigenes Makro `CTRL` definiert, mit dem man den Code zu den einzelnen Kontrollzeichen ermitteln kann. Da dieses Makro jedoch nicht auf allen Systemen angeboten wird, empfiehlt sich der folgende Codeausschnitt:

```
#ifndef CTRL
#  define CTRL(ch) ((ch)&0x1F)
#endif
```

Will man z.B. `VEOF` auf *Strg-F* festlegen, ist nur folgendes anzugeben:

```
terminal.c_cc[VEOF] = CTRL('F');
```

Im Kapitel 20.3 werden diese speziellen Eingabezeichen detailliert behandelt.

20.1.5 Terminalflags

In Tabelle 20.2 wird ein Überblick über die Terminalflags gegeben, die man erfragen oder setzen kann. In Kapitel 20.4 werden diese Terminalflags detailliert beschrieben.

Komponente Flag	POSI X.1	Erweiterung SVR4 BSD		Bedeutung
c_iflag				
BRKINT	x			Generieren von SIGINT bei BREAK
ICRNL	x			Umwandeln von CR in NL bei der Eingabe
IGNBRK	x			Ignorieren von BREAK
IGNCR	x			Ignorieren von CR
IGNPAR	x			Ignorieren von Bytes mit Paritätsfehlern
IMAXBEL		x	x	Akustisches Signal bei vollem Eingabepuffer
INLCR	x			Umwandeln von NL in CR bei der Eingabe
INPCK	x			Einschalten der Eingabe-Paritätsprüfung
ISTRIP	x			Abschneiden des 8. Bits bei Eingabezeichen
IUCLC		x		Umwandeln von Groß- in Kleinbuchstaben bei der Eingabe
IXANY		x	x	Zulassen beliebiger Zeichen, um angehaltene Ausgabe fortzusetzen
IXOFF	x			Einschalten des START/STOP-Eingabeprotokolls
IXON	x			Einschalten des START/STOP-Ausgabeprotokolls
PARMRK	x			Markieren von Paritätsfehlern
c_oflag				
BSDLY		x		Verzögerungsart für Backspace (BS0 oder BS1)
CRDLY		x		Verzögerungsart für CR (CR0, CR1, CR2 oder CR3)
FFDLY		x		Verzögerungsart für <i>form-feed</i> (FF0 oder FF1)
NLDLY		x		Verzögerungsart für NL (NL0 oder NL1)
OCRNL		x		Umwandeln von CR in NL bei der Ausgabe
OFDEL		x		Auffüllzeichen ist DEL, sonst NUL
OFILL		x		Auffüllzeichen anstelle einer zeitlichen Verzögerung
OLCUC		x		Umwandeln von Klein- in Großbuchstaben bei der Ausgabe
ONLCR		x		Umwandeln von NL in CR-NL bei der Ausgabe
ONLRET		x		Einstellen von NL auf CR-Funktion
ONOCR		x		Unterdrücken von CR in Spalte 0
ONOEOT			x	Ignorieren von EOT (<i>Strg-D</i>) bei Ausgabe
OPOST	x			Einschalten einer implementierungsdefinierten Ausgabeart
OXTABS			x	Umwandeln von Tabs in Leerzeichen

Tabelle 20.2: Terminal-Flags

Komponente Flag	POSI X.1	Erweiterung SVR4 BSD		Bedeutung
TABDLY		x		Verzögerungsart für horizontale Tabs (TAB0, TAB1, TAB2, TAB3 oder XTABS)
VTDLY		x		Verzögerungsart für vertikale Tabs (VT0 oder VT1)
c_cflag				
CCTS_OFLOW			x	Einschalten des CTS-Ausgabeprotokolls
CIGNORE			x	Ignorieren von Kontrollflags
CLOCAL	x			Ausschalten der Modemsteuerung
CREAD	x			Aktivieren des Empfängers
CRTS_IFLOW			x	Einschalten des RTS-Eingabeprotokolls
CRTSCTS				(Linux) Einschalten der Hardware-Flußkontrolle (RTS- und CTS-Leitungen)
CSIZE	x			Bitanzahl für ein Zeichen (CS5, CS6, CS7 oder CS8)
CSTOPB	x			Zwei Stop-Bits anstelle von einem senden
HUPCL	x			Verbindungsabbruch bei Beendigung des letzten Prozesses
MDMBUF			x	Ausgabeprotokoll entsprechend dem <i>Modem-Carrier</i> -Flag
PARENB	x			Einschalten von Paritätsprüfung- und erzeugung
PARODD	x			Ungerade Parität, sonst gerade
c_lflag				
ALTWERASE			x	Verwendung eines alternativen WERASE-Algorithmus
ECHO	x			Einschalten der <i>Echo</i> -Funktion
ECHOCTL		x	x	Darstellung von Steuerzeichen als Zeichen
ECHOE	x			Gelöschte Zeichen mit Leerzeichen überschreiben
ECHOK	x			Zeichen wirklich löschen oder zur Neueingabe in neue Zeile positionieren
ECHOKE		x	x	Zeichen beim Löschen einer Zeile entfernen
ECHONL	x			Ausgabe von NL, sogar wenn <i>Echo</i> -Funktion nicht eingeschaltet
ECHOPRT		x	x	Ausgabe von gelöschten Zeichen für <i>Hardcopy</i>
FLUSHO		x	x	Leeren von Ausgabepuffern
ICANON	x			Zeilenorientierter Eingabemodus (<i>kanonischer Eingabemodus</i>)
IEXTEN	x			Einschalten des erweiterten Zeichensatzes für die Eingabe
ISIG	x			Einschalten der Sonderbedeutung von Terminalsteuerzeichen
NOFLSH	x			Ausschalten des Leeren von Puffers bei INTR oder QUIT

Tabelle 20.2: Terminal-Flags

Komponente Flag	POSIX.1	Erweiterung		Bedeutung
		SVR4	BSD	
NOKERNINFO			x	Ausschalten der Kern-Ausgabe bei STATUS
PENDIN		x	x	Neuausgabe von noch nicht verarbeiteten Eingabezeichen
TOSTOP	x			Senden des Signals SIGTTOU bei der Ausgabe durch Hintergrundprozesse
XCASE		x		Umwandeln von eingegebenen Groß- in Kleinbuchstaben

Tabelle 20.2: Terminal-Flags

20.1.6 Das Kommando stty

Alle zuvor beschriebenen speziellen Eingabezeichen und Terminalflags können mit den beiden Funktionen **tcgetattr** und **tcsetattr** erfragt oder geändert werden (siehe auch Kapitel 20.2). Daneben ist es mit dem Kommando **stty** möglich, diese speziellen Eingabezeichen und Terminalflags von der Kommandozeile oder aus einem Shellskript heraus zu erfragen oder zu ändern. Um die momentanen Terminaleinstellungen zu erfragen, muß **stty** mit der Option **-a** aufgerufen werden.

```
$ stty -a
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel n10 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echopr
echoctl echoke
$
```

Bei dieser Ausgabe bedeutet ein vorangestellter Querstrich (-), daß das entsprechende Flag ausgeschaltet ist. Die erste Zeile zeigt dabei unter anderem Zeilen- und Spaltenzahl für das aktuelle Terminalfenster (siehe auch Kapitel 20.8). Da **stty** ein Benutzerkommando und keine Systemfunktion ist, ist es durch POSIX.2 und nicht durch POSIX.1 spezifiziert.

20.1.7 Terminal-E/A-Funktionen und der Modul »Terminal Line Discipline«

Die meisten Unix-Systeme implementieren den kanonischen Modus (zeilenorientierten Modus) in einem eigenen Modul, das *Terminal Line Discipline* genannt wird. Dieses Modul ist zwischen den E/A-Funktionen und dem Terminalgerätetreiber eingeordnet. Abbildung 20.1 verdeutlicht dies.

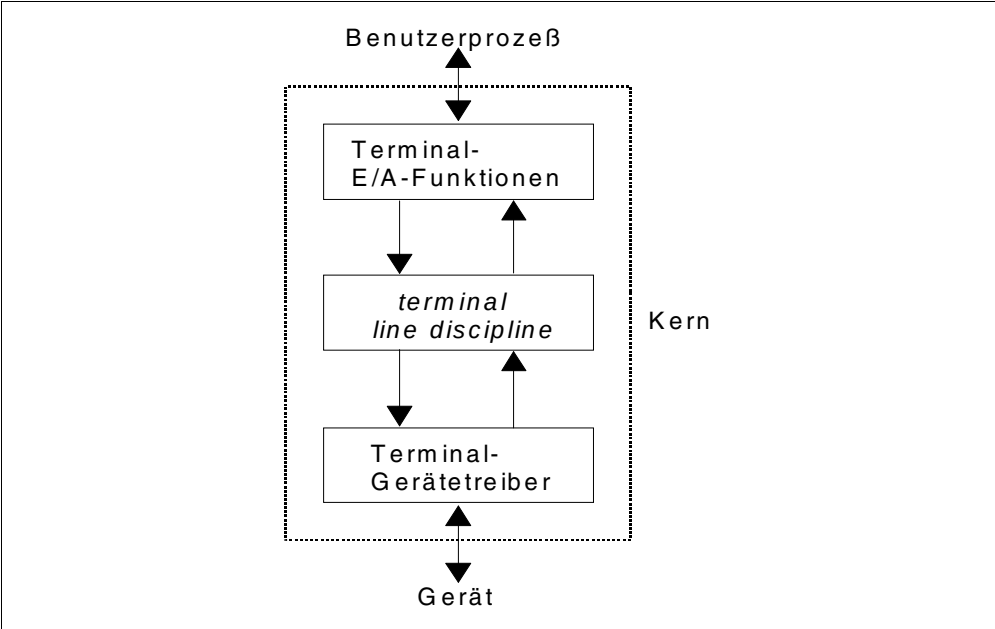


Abbildung 20.1: Das Modul Terminal Line Discipline

POSIX.1 bietet zum Erfragen und Ändern der Terminalcharakteristika die in Tabelle 20.3 zusammengefaßten Funktionen an.

Funktion	Beschreibung
<i>tcgetattr</i>	Erfragen der Attribute (Struktur <i>termios</i>)
<i>tcsetattr</i>	Setzen der Attribute (Struktur <i>termios</i>)
<i>cfgetispeed</i>	Erfragen der Eingabegeschwindigkeit
<i>cfgetospeed</i>	Erfragen der Ausgabegeschwindigkeit
<i>cfsetispeed</i>	Setzen der Eingabegeschwindigkeit
<i>cfsetospeed</i>	Setzen der Ausgabegeschwindigkeit
<i>tcdrain</i>	Warten bis gesamte Ausgabe übertragen ist
<i>tcflow</i>	Suspendieren der Übertragung
<i>tcflush</i>	Leeren der Ein- und /oder Ausgabepuffer
<i>tcsendbreak</i>	Schicken des <i>BREAK</i> -Zeichens
<i>tcgetpgrp</i>	Erfragen der Vordergrundprozeßgruppen-ID
<i>tcsetpgrp</i>	Setzen der Vordergrundprozeßgruppen-ID

Tabelle 20.3: Terminal-E/A-Funktionen von POSIX.1

Abbildung 20.2 verdeutlicht die Wirkungsweise der einzelnen Funktionen

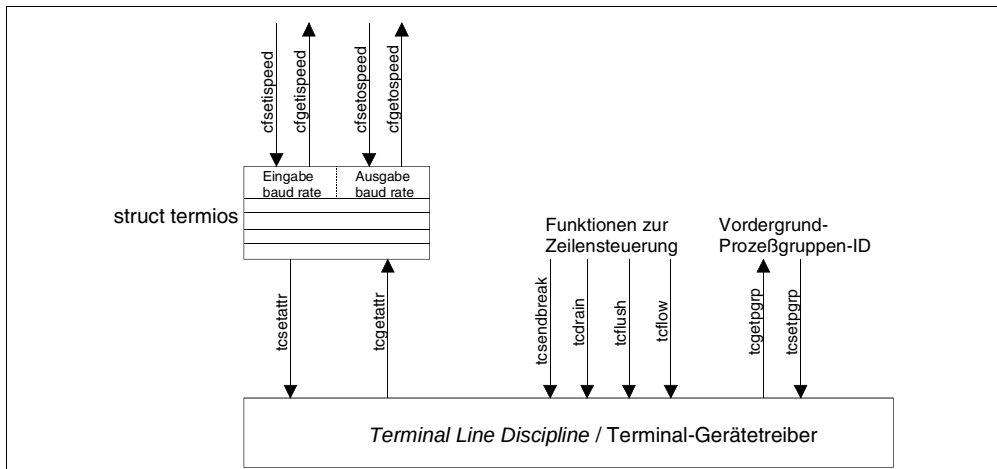


Abbildung 20.2: Terminal-E/A-Funktionen im Überblick

POSIX.1 legt nicht fest, in welcher Komponente der Struktur `termios` die Baudrate gespeichert ist. In vielen Systemen befindet sich die Baudrate in der Komponente `c_cflag`, in anderen Systemen wie z.B. BSD-Unix sind zwei eigene Komponenten in der Struktur `termios` für die Eingabe- und die Ausgabe-Baudrate vorgesehen.

20.2 Terminalattribute und Terminalidentifizierung

In diesem Kapitel werden zum einen Funktion vorgestellt, mit denen die Terminalattribute gesetzt oder abgefragt werden können. Zum anderen werden hier Funktionen vorgestellt, mit denen der Name eines Terminals erfragt werden kann oder aber festgestellt, ob ein Filedeskriptor auf ein Terminal eingestellt ist.

20.2.1 tcsetattr und tcgetattr – Setzen und Erfragen von Terminalattributen

Zum Setzen und Erfragen der in der Struktur `termios` gespeicherten Terminalattribute stehen die beiden Funktionen **`tcsetattr`** und **`tcgetattr`** zur Verfügung.

```
#include <termios.h>
```

```
int tcgetattr(int fd, struct termios *termzgr);
```

```
int tcsetattr(int fd, int option, const struct termios *termzgr);
```

beide geben zurück: 0 (bei Erfolg); -1 bei Fehler

`fd` muß ein Filedeskriptor für ein Terminal sein, andernfalls beenden sich beide Funktionen mit einem Fehler, wobei `errno` auf `ENOTTY` gesetzt wird.

Das Argument `option` bei der Funktion **`tcsetattr`** legt fest, wann die neuen Terminalattribute in Kraft treten sollen. Für `option` kann eine der folgenden Konstanten angegeben werden:

`TCSANOW`

Änderung sofort aktivieren.

`TCSADRAIN`

Änderung aktivieren, nachdem alle anstehenden Ausgaben übertragen wurden. Diese Option sollte benutzt werden, wenn Ausgabeparameter verändert werden.

`TCSAFLUSH`

Änderung erst aktivieren, nachdem alle anstehenden Ausgaben übertragen wurden. Zudem werden hier bei der Aktivierung alle noch nicht bearbeiteten Eingaben im Eingabepuffer weggeworfen.

`tcsetattr` liefert als Rückgabewert 0 für erfolgreich, wenn nur eine der geforderten Änderungen, aber eventuell nicht alle durchgeführt werden konnten. Um sicher zu sein, daß alle geforderten Änderungen durchgeführt wurden, muß man nach einem **`tcsetattr`**-Aufruf **`tcgetattr`** aufrufen und die aktuellen Terminalattribute mit den geforderten Attributen vergleichen, wie dies im folgenden Programm 20.1 (`tc.pruef.c`) gezeigt wird.

Beispiel

*Überprüfen, ob bei **`tcsetattr`** die geforderten Änderungen wirklich vorgenommen wurden*

```
#include <stdio.h>
#include <termios.h>
#include "eighdr.h"

int
main(void)
{
    struct termios  terminal, terminal_alt, terminal_neu;
    FILE           *fz;
    int            zeich;

    /*----- Oeffnen des Terminals */
    if ( (fz = fopen(ctermid(NULL), "w+")) == NULL)
        fehler_meld(FATAL_SYS, "kann Terminal nicht oeffnen");

    /*----- Aktuellen Attribute des Terminals erfragen */
    tcgetattr(fileno(fz), &terminal_alt);
    printf("vorher: ECHO=%d, ECHOE=%d\n",
           (terminal_alt.c_lflag & ECHO) && 1,
           (terminal_alt.c_lflag & ECHOE) && 1);

    /*----- Flags ECHO und ECHOE fuer Terminal ausschalten */
```

```

terminal_neu = terminal_alt;
terminal_neu.c_lflag &= ~(ECHO | ECHOE);
tcsetattr(fileno(fz), TCSAFLUSH, &terminal_neu);

/*----- Testen, ob Flags ECHO und ECHOE ausgeschaltet wurden */
tcgetattr(fileno(fz), &terminal);
if ( (terminal.c_lflag & ECHO) || (terminal.c_lflag & ECHOE) )
    fehler_meld(WARNUNG, "ECHO und ECHOE wurden nicht ausgeschaltet");
printf("nachher: ECHO=%d, ECHOE=%d\n",
       (terminal.c_lflag & ECHO) && 1,
       (terminal.c_lflag & ECHOE) && 1);

/*----- Terminal wieder in urspruenglichen Zustand bringen */
tcsetattr(fileno(fz), TCSAFLUSH, &terminal_alt);
tcgetattr(fileno(fz), &terminal);
if ( (terminal.c_lflag & ECHO) != ECHO ||
     (terminal.c_lflag & ECHOE) != ECHOE )
    fehler_meld(WARNUNG, "ECHO und ECHOE nicht wieder eingeschaltet");
printf("am Ende: ECHO=%d, ECHOE=%d\n",
       (terminal.c_lflag & ECHO) && 1,
       (terminal.c_lflag & ECHOE) && 1);

return(0);
}

```

Programm 20.1 (**tc_pruef.c**): Prüfen, ob **tcsetattr** die geforderten Änderungen vorgenommen hat

Nachdem man dieses Programm 20.1 (**tc_pruef.c**) kompiliert und gelinkt hat

```
cc -o tc_pruef tc_pruef.c fehler.c
```

ergibt sich z.B. folgender Ablauf:

```

$ tc_pruef
vorher: ECHO=1, ECHOE=1
nachher: ECHO=0, ECHOE=0
am Ende: ECHO=1, ECHOE=1
$

```

Um die Geräteeinstellungen eines Terminals zu erfragen, muß man die zugehörige Gerätedatei öffnen und **tcgetattr** mit den so erhaltenen Filedeskriptor aufrufen. Bei manchen tty-Geräten, die nur einmal geöffnet werden können, muß bei **open** das Flag **O_NONBLOCK** angegeben werden, so daß der **open**-Aufruf nicht blockiert wird. Um eventuell für spätere Lese- und Schreibzugriffe das Blockieren wieder einzuschalten, sollte man nach dem Öffnen der Gerätedatei mit **open** folgendes aufrufen:

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) & ~O_NONBLOCK);
```

20.2.2 ctermid – Erfragen des Kontrollterminalnamens

Zum Erfragen des Kontrollterminalnamens, der in den meisten Unix-Systemen **/dev/tty** ist, stellt POSIX.1 die Funktion **ctermid** zur Verfügung.

```
#include <stdio.h>
```

```
char *ctermid(char *zgr);
```

gibt zurück: Adresse, an der der Name des Kontrollterminals steht

Wird für `zgr` nicht ein `NULL`-Zeiger angegeben, so schreibt ***ctermid*** den Namen des Kontrollterminals an diese Adresse. Die Adresse `zgr` sollte dabei auf einen Speicherplatz zeigen, der groß genug ist, um mindestens `L_ctermid` Bytes aufzunehmen. Die Konstante `L_ctermid` ist in `<stdio.h>` definiert.

Wird für `zgr` ein `NULL`-Zeiger angegeben, so allokiert die Funktion ***ctermid*** den für den Namen benötigten Speicherplatz selbst, bevor sie den Kontrollterminalnamen dorthin schreibt.

In beiden Fällen liefert die Funktion ***ctermid*** die Adresse, an die sie den Kontrollterminalnamen geschrieben hat, als Rückgabewert.

Beispiel

Implementierung der Funktion *ctermid.c*

```
#include <stdio.h>
#include <string.h>

static char ctermid_name[L_ctermid];

char *ctermid(char *zgr)
{
    if (zgr == NULL)
        zgr = ctermid_name;
    return(strcpy(zgr, "/dev/tty"));
}
```

Programm 20.2 (***ctermid.c***) Mögliche Implementierung der Funktion ***ctermid***

20.2.3 isatty – Erfragen, ob ein Filedeskriptor auf Terminal eingestellt ist

Um festzustellen, ob ein Filedeskriptor mit einer Terminalgerätedatei verbunden ist, steht die Funktion ***isatty*** zur Verfügung.

```
#include <unistd.h>
```

```
int isatty(int fd);
```

gibt zurück: 1 (TRUE), wenn `fd` auf Terminalgerätedatei eingestellt; sonst 0 (FALSE)

Beispiel*Implementierung der Funktion isatty*

```
#include <termios.h>
#include "eighdr.h"

int isatty(int fd)
{
    struct termios terminal;

    return(tcgetattr(fd, &terminal) != -1);
}

#ifdef TEST
int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "Terminal" : "...kein Terminal...");
    printf("fd 1: %s\n", isatty(1) ? "Terminal" : "...kein Terminal...");
    printf("fd 2: %s\n", isatty(2) ? "Terminal" : "...kein Terminal...");
    exit(0);
}
#endif
```

Programm 20.3 (isatty.c): Mögliche Implementierung der Funktion isatty

Um diese Funktion **isatty** zu testen, muß man beim Kompilieren die Konstante `TEST` definieren.

```
cc -o isatty -DTEST isatty.c fehler.c
```

Nun können wir unsere Implementierung der Funktion **isatty** testen:

```
$ isatty
fd 0: Terminal
fd 1: Terminal
fd 2: Terminal
$ isatty </usr/include/stdio.h 2>/dev/null
fd 0: ...kein Terminal...
fd 1: Terminal
fd 2: ...kein Terminal...
$
```

20.2.4 ttyname – Erfragen von Terminalpfadnamen

Um den Pfadnamen eines Terminals zu erfragen, auf den ein offener Filedeskriptor eingestellt ist, steht die Funktion **ttyname** zur Verfügung.

```
#include <unistd.h>
```

```
char *ttyname(int fd);
```

gibt zurück: Adresse, an der der Terminalname steht; NULL bei Fehler

Beispiel

Implementierung der Funktion ttyname

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include "eighdr.h"

#define PRAEFIX          "/dev/"
#define PRAEFIX_LAENGE  strlen(PRAEFIX)

char *ttyname(int fd)
{
    struct stat    fdstat, devstat;
    DIR            *dir;
    struct dirent  *dir_eintrag;
    static char    pfadname[_POSIX_PATH_MAX + 1];
    char          *termpfad = NULL;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    strcpy(pfadname, PRAEFIX);
    if ( (dir = opendir(PRAEFIX)) == NULL)
        return(NULL);
    while ( (dir_eintrag = readdir(dir)) != NULL) {
        if (dir_eintrag->d_ino == fdstat.st_ino) {
            strncpy(pfadname + PRAEFIX_LAENGE, dir_eintrag->d_name,
                _POSIX_PATH_MAX - PRAEFIX_LAENGE);
            if (stat(pfadname, &devstat) >= 0    &&
                devstat.st_ino == fdstat.st_ino &&
                devstat.st_dev == fdstat.st_dev) {
                termpfad = pfadname;
                break;
            }
        }
    }
    closedir(dir);
}
```



```

    return(termpfad);
}

#ifdef TEST
int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? ttyname(0) : "...kein Terminal...");
    printf("fd 1: %s\n", isatty(1) ? ttyname(1) : "...kein Terminal...");
    printf("fd 2: %s\n", isatty(2) ? ttyname(2) : "...kein Terminal...");
    exit(0);
}
#endif

```

Programm 20.4 (ttyname.c): Mögliche Implementierung der Funktion **ttyname**

Um diese Implementierung der Funktion **ttyname** zu testen, muß man beim Kompilieren die Konstante **TEST** definieren.

```
cc -o ttyname -DTEST ttyname.c fehler.c
```

Nun können wir unsere Implementierung der Funktion **ttyname** testen.

```

$ ttyname
fd 0: /dev/tty
fd 1: /dev/tty
fd 2: /dev/tty
$ ttyname </dev/console >/dev/tty 2>/dev/null
fd 0: /dev/console
fd 1: /dev/tty
fd 2: ...kein Terminal...
$

```

Das folgende Programm 20.4 (ttyname2.c) folgt symbolischen Links im Gegensatz zu Programm 20.3 (ttyname.c).

```

#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <limits.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>
#include <unistd.h>
#include "eighdr.h"

#define PRAEFIX          "/dev/"
#define PRAEFIX_LAENGE  strlen(PRAEFIX)
#define MAX_NAME         100

char *ttyname(int fd)
{
    struct stat    fdstat, devstat;

```

```

DIR          *dir;
struct dirent *dir_eintrag;
static char  pfadname[_POSIX_PATH_MAX + 1];
int          laenge;
char         *termpfad = NULL;

if (isatty(fd) == 0)
    return(NULL);
if (fstat(fd, &fdstat) < 0)
    return(NULL);
if (S_ISCHR(fdstat.st_mode) == 0)
    return(NULL);

strcpy(pfadname, PRAEFIX);
if ( (dir = opendir(PRAEFIX)) == NULL)
    return(NULL);
while ( (dir_eintrag = readdir(dir)) != NULL) {
    strncpy(pfadname + PRAEFIX_LAENGE, dir_eintrag->d_name,
        _POSIX_PATH_MAX - PRAEFIX_LAENGE);
    if (stat(pfadname, &devstat) == 0) {
        if (devstat.st_mode & S_IFMT == S_IFLNK) {
            if ( (laenge = readlink(pfadname, pfadname, MAX_NAME)) < 0)
                fehler_meld(FATAL_SYS, "readlink-Fehler bei %s", pfadname);
            pfadname[laenge] = '\0';
            if (stat(pfadname, &devstat) < 0)
                continue;
        }
        if (devstat.st_ino == fdstat.st_ino &&
            devstat.st_dev == fdstat.st_dev) {
            termpfad = pfadname;
            break;
        }
    }
}
closedir(dir);
return(termpfad);
}

#ifdef TEST
int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? ttyname(0) : "...kein Terminal...");
    printf("fd 1: %s\n", isatty(1) ? ttyname(1) : "...kein Terminal...");
    printf("fd 2: %s\n", isatty(2) ? ttyname(2) : "...kein Terminal...");
    exit(0);
}
#endif

```

Programm 20.5 (ttyname2.c): Alternative Implementierung der Funktion **ttyname**

20.2.5 **getpass** – Verdecktes Einlesen eines Paßwortes

Um einen String verdeckt, also mit ausgeschalteter *ECHO*-Funktion einzulesen, steht die Funktion **getpass** zur Verfügung.

```
#include <stdlib.h>
```

```
char *getpass(const char *prompt);
```

gibt zurück: Adresse des eingegebenen Strings (bei Erfolg); NULL bei Fehler

Die Funktion **getpass** gibt den angegebenen `prompt` auf der Standardfehlerausgabe aus und liest dann vom Terminal (Geräte-datei `/dev/tty`) einen maximal 8 Zeichen langen String verdeckt ein. Bei der Eingabe ist dieser String mit Neue-Zeile-Zeichen oder EOF abzuschließen.

Wenn `/dev/tty` nicht geöffnet werden kann, liefert **getpass** einen NULL-Zeiger.

Beispiel

Implementierung der Funktion **getpass**

```
#include <signal.h>
#include <stdio.h>
#include <termios.h>
#include "eighdr.h"

#define MAX_PASSWORT 8 /* Maximal 8 Zeichen fuer ein Passwort */

char *getpass(const char *prompt)
{
    static char    puffer[MAX_PASSWORT + 1];
    char          *zgr;
    sigset_t       sig_maske, sig_alt;
    struct termios terminal, terminal_alt;
    FILE          *fz;
    int            zeich;

    if ( (fz = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fz, NULL);

    /* Blockieren der Signale SIGINT u. SIGTSTP */
    sigemptyset(&sig_maske);
    sigaddset(&sig_maske, SIGINT);
    sigaddset(&sig_maske, SIGTSTP);
    sigprocmask(SIG_BLOCK, &sig_maske, &sig_alt);

    tcgetattr(fileno(fz), &terminal_alt);
    terminal = terminal_alt;
    terminal.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fz), TCSAFLUSH, &terminal);
```

```

fputs(prompt, fz);

zgr = puffer;
while ( (zeich = getc(fz)) != EOF && zeich != '\n')
    if (zgr < &puffer[MAX_PASSWORT])
        *zgr++ = zeich;
*zgr = '\0';
putc('\n', fz); /* Echo fuer NL */

/* Terminal in alten Zustand zuruecksetzen */
tcsetattr(fileno(fz), TCSAFLUSH, &terminal_alt);

/* Alte Signalmaske wiederherstellen */
sigprocmask(SIG_SETMASK, &sig_alt, NULL);
fclose(fz);

return(puffer);
}

#ifdef TEST
int
main(void)
{
    char *zgr;

    if ( (zgr = getpass("Passwort: ")) == NULL)
        fehler_meld(FATAL_SYS, "getpass-Fehler");
    printf("    Dein eingegebenes Passwort ist ...%s...\n", zgr);

    /* ..... */
    /*--- Auswertung des Passworts ---*/
    /* ..... */

    /* Passwort aus Sicherheitsgruenden nun loeschen */
    while (*zgr)
        *zgr++ = 0;

    /*..... Weiterer Code .....*/
}
#endif

```

*Programm 20.6 (getpass.c): Mögliche Implementierung der Funktion **getpass***

20.3 Spezielle Eingabezeichen

Hier werden die speziellen Eingabezeichen aus Tabelle 20.1 detailliert beschrieben. Die meisten dieser speziellen Eingabezeichen werden nach ihrer Erkennung und Sonderbehandlung durch den Terminalgerätetreiber weggeworfen und nicht an den lesenden Prozeß zurückgeliefert. Ausnahmen sind lediglich die Neu-Zeile-Zeichen (NL, EOL, EOL2) und Carriage-Return (CR).

CR

Carriage-Return wird bei einer Eingabe im kanonischen (zeilenorientierten) Modus erkannt. Wenn die beiden Flags `ICANON` und `ICRNL` gesetzt sind und das Flag `IGNCR` nicht gesetzt ist, so wird ein CR-Zeichen in NL umgewandelt, so daß es wie ein NL-Zeichen wirkt. Dieses Zeichen CR oder eben das umgewandelte Zeichen NL wird an den lesenden Prozeß zurückgeliefert.

DISCARD

Dieses Eingabezeichen wird als spezielles interpretiert, wenn das Flag `IEXTEN` (Erweiterter Zeichensatz) für die Eingabe gesetzt ist. Es bewirkt dann, daß die nachfolgende Ausgabe so lange weggeworfen wird, bis ein erneutes DISCARD-Zeichen eingegeben wird oder diese DISCARD-Einstellung mit der Option `FLUSHO` (siehe Kapitel 20.4) wieder aufgehoben wird. DISCARD-Zeichen werden nicht an den lesenden Prozeß weitergeliefert.

DSUSP

Dieses Zeichen ist für die Jobkontrolle vorgesehen. Es wird als spezielles Eingabezeichen interpretiert, wenn die beiden Flags `IEXTEN` (erweiterter Zeichensatz) und `ISIG` gesetzt sind und Jobkontrolle unterstützt wird. Wie das spezielle Eingabezeichen `SUSP`, so generiert auch `DSUSP` das Signal `SIGTSTP`, das an alle Prozesse in der Vordergrundprozeßgruppe geschickt wird. Anders als `SUSP` wird `DSUSP` nur dann an die Prozeßgruppe geschickt, wenn ein Prozeß vom Kontrollterminal liest und nicht wenn `DSUSP` (*delayed-suspend*) eingegeben wird. `DSUSP`-Zeichen werden nicht an den lesenden Prozeß weitergeleitet.

EOF

Diese Eingabezeichen wird als spezielles interpretiert, wenn das Flag `ICANON` gesetzt ist. Wenn dieses Zeichen eingegeben wird, so werden alle noch zum Lesen anstehenden Zeichen sofort an den lesenden Prozeß weitergeleitet. Stehen keine Zeichen zum Lesen an, so wird 0 als Rückgabewert geliefert. Das EOF-Zeichen sollte immer am Anfang einer Zeile eingegeben werden. Im kanonischen (zeilenorientierten) Modus wird das EOF-Zeichen nicht an den lesenden Prozeß weitergeleitet.

EOL

Das EOL-Zeichen ist eine Alternative zum NL-Zeichen. EOL wird nur dann als spezielles Eingabezeichen erkannt, wenn das Flag `ICANON` gesetzt ist. Das Zeichen EOL, das normalerweise nicht benutzt wird, wird an den lesenden Prozeß weitergegeben.

EOL2

Das EOL2-Zeichen ist neben dem POSIX.1-Zeichen EOL in SVR4 und BSD-Unix eine Alternative zum NL-Zeichen. EOL2 wird nur dann als spezielles Eingabezeichen erkannt, wenn das Flag `ICANON` gesetzt ist. Das Zeichen EOL2, das normalerweise nicht benutzt wird, wird an den lesenden Prozeß weitergegeben.

ERASE

Das ERASE-Zeichen bewirkt das Löschen des letzten Zeichens in einer Zeile, wenn das Flag `ICANON` gesetzt ist. Am Anfang einer Zeile hat ERASE keine Auswirkung. Ein ERASE-Zeichen wird nicht an den lesenden Prozeß weitergeleitet.

INTR

Das INTR-Zeichen wird als spezielles Eingabezeichen interpretiert, wenn das Flag `ISIG` gesetzt ist. Das INTR-Zeichen generiert das Signal `SIGINT`, das an alle Prozesse in der Vordergrundprozeßgruppe geschickt wird. Ein INTR-Zeichen wird nicht an den lesenden Prozeß weitergeleitet.

KILL

Das KILL-Zeichen bewirkt das Löschen der ganzen aktuellen Zeile, wenn das Flag `ICANON` gesetzt ist. Ein KILL-Signal wird nicht an den lesenden Prozeß weitergeleitet.

LNEXT

Das LNEXT-Zeichen bewirkt das Ausschalten der Sonderbedeutung des nachfolgenden speziellen Eingabezeichens, wenn das Flag `IEXTEN` (erweiterter Zeichensatz) gesetzt ist. Während ein LNEXT-Zeichen nicht an den lesenden Prozeß weitergeleitet wird, wird aber das nachfolgende Zeichen an diesen Prozeß weitergeleitet.

NL

Das Neu-Zeile-Zeichen `NL` (*newline*) wird als spezielles Eingabezeichen interpretiert, wenn das Flag `ICANON` gesetzt ist. Das NL-Zeichen wird an den lesenden Prozeß weitergereicht.

QUIT

Das QUIT-Zeichen wird als spezielles Eingabezeichen erkannt, wenn das Flag `ISIG` gesetzt ist. Das QUIT-Zeichen generiert das Signal `SIGQUIT`, das an alle Prozesse in der Vordergrundprozeßgruppe geschickt wird. Ein QUIT-Signal wird nicht an den lesenden Prozeß weitergeleitet.

Im Unterschied zu `INTR` beendet das QUIT-Signal nicht nur den entsprechenden Prozeß, sondern bewirkt das Anlegen einer `core`-Datei.

REPRINT

Das REPRINT-Zeichen bewirkt, daß alle noch nicht gelesenen Eingabezeichen ausgegeben werden, wenn die beiden Flags `IEXTEN` und `ICANON` gesetzt sind. Ein REPRINT-Zeichen wird nicht an den lesenden Prozeß weitergereicht.

START

Das START-Zeichen wird als spezielles Eingabezeichen interpretiert, wenn das Flag `IXON` gesetzt ist, und es wird automatisch auf der Ausgabe generiert, wenn das Flag `IXOFF` gesetzt ist. Der Empfang eines START-Zeichens bei gesetztem `IXON`-Flag bewirkt das Fortsetzen einer zuvor mit dem STOP-Zeichen angehaltenen Ausgabe. In diesem Fall wird das START-Zeichen nicht an den lesenden Prozeß weitergeleitet.

STATUS

Das STATUS-Zeichen wird als spezielles Eingabezeichen erkannt, wenn die beiden Flags `IEXTEN` und `ICANON` gesetzt sind. Das STATUS-Zeichen generiert das Signal `SIGINFO`, das an alle Prozesse in der Vordergrundprozeßgruppe geschickt wird. Ist dabei das Flag `NOKERNINFO` nicht gesetzt, so wird zusätzlich Statusinformation über die Vordergrundprozeßgruppe am Terminal ausgegeben. Ein STATUS-Zeichen wird nicht an den lesenden Prozeß weitergereicht.

STOP

Das STOP-Zeichen wird als spezielles Eingabezeichen erkannt, wenn das Flag `IXON` gesetzt ist, und es wird automatisch auf der Ausgabe generiert, wenn das Flag `IXOFF` gesetzt ist. Der Empfang eines STOP-Zeichens bei gesetztem `IXON`-Flag bewirkt das Anhalten einer Ausgabe. In diesem Fall wird das STOP-Zeichen nicht an den lesenden Prozeß weitergeleitet. Die angehaltene Ausgabe wird bei der Eingabe des START-Zeichens fortgesetzt.

Wenn das `IXOFF`-Flag gesetzt ist, generiert der Terminaltreiber automatisch ein STOP-Zeichen, um ein Überlaufen des Eingabepuffers zu verhindern.

SUSP

Das SUSP-Zeichen ist für die Jobkontrolle vorgesehen. Das SUSP-Zeichen wird als spezielles Eingabezeichen erkannt, wenn das Flag `ISIG` gesetzt ist und Jobkontrolle unterstützt wird. Das SUSP-Zeichen generiert das Signal `SIGTSTP`, das an alle Prozesse in der Vordergrundprozeßgruppe geschickt wird. Ein SUSP-Zeichen wird nicht an den lesenden Prozeß weitergeleitet.

WERASE

Das WERASE-Zeichen bewirkt das Löschen des letzten Wortes, wenn die beiden Flags `IEXTEN` und `ICANON` gesetzt sind. Eventuell nach diesem Wort eingegebene Leer- und Tabulatorenzeichen werden dabei auch gelöscht. Der Wortanfang ist dabei durch ein voranstehendes Leer- und /oder Tabulatorzeichen festgelegt.

Durch Setzen des Flags `ALTWERASE` kann dies geändert werden. Ist `ALTWERASE` gesetzt, so wird als Wortanfang das erste nicht-alphanumerische Zeichen festgelegt. Ein WERASE-Zeichen wird nicht an den lesenden Prozeß weitergereicht.

Bis auf die beiden speziellen Eingabezeichen `CR` und `NL` können alle anderen speziellen Eingabezeichen geändert werden. Dazu muß der entsprechende Eintrag im Array `c_cc` (in Struktur `termios`) geändert werden. Als Array-Index muß dabei der Name des entsprechenden Zeichens mit vorangestelltem `V` (siehe auch Tabelle 20.1) verwendet werden, wie z.B. `c_cc[VSUSP]`.

POSIX.1 ermöglicht auch das Ausschalten der Sonderbedeutung dieser Eingabezeichen. Wenn `_POSIX_VDISABLE` definiert ist, so bewirkt die Zuweisung dieser Konstante an das entsprechende Element des Arrays `c_cc` das Ausschalten des zugehörigen speziellen Eingabezeichens.

Ob `_POSIX_VDISABLE` definiert ist, kann mit den Funktionen ***pathconf*** und ***fpathconf*** erfragt werden.

Beispiel

Ausschalten von WERASE und Ändern des EOF-Zeichens

Das folgende Programm 20.6 (`spezein.c`) schaltet die Sonderbedeutung von WERASE (Wort löschen) aus und setzt das EOF-Zeichen auf `Strg-F`.

```

#include <termios.h>
#include "eighdr.h"

int
main(void)
{
    struct termios    terminal;
    long              ausschalten;

    if (isatty(STDIN_FILENO) == 0)
        fehler_meld(FATAL, "stdin ist kein Terminal");

    if ( (ausschalten = fpathconf(STDIN_FILENO, _POSIX_VDISABLE)) < 0)
        fehler_meld(FATAL, "_POSIX_VDISABLE nicht definiert");

    if (tcgetattr(STDIN_FILENO, &terminal) < 0)
        fehler_meld(FATAL_SYS, "tcgetattr-Fehler");

    terminal.c_cc[VWERASE] = ausschalten; /* WERASE ausschalten */
    terminal.c_cc[VEOF]    = 6;          /* EOF auf Ctrl-F setzen */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &terminal) < 0)
        fehler_meld(FATAL_SYS, "tcsetattr-Fehler");

    exit(0);
}

```

Programm 20.7 (spezein.c): Ausschalten von WERASE und Ändern des EOF-Zeichens

Hinweis

`_POSIX_VDISABLE` ist in SVR4 und Linux als 0 und in BSD-Unix als 0337 (oktal) definiert.

Ein weiteres spezielles Eingabezeichen ist `BREAK`. `BREAK` ist eigentlich kein Zeichen, sondern ein Ereignis, das während einer asynchronen seriellen Datenübertragung eintritt. Das Eintreten dieses Ereignisses wird dem Gerätetreiber abhängig vom seriellen Interface auf verschiedene Arten mitgeteilt. Die meisten Terminals haben eine *BREAK*-Taste, die dieses `BREAK`-Ereignis auslöst. Bei asynchroner serieller Datenübertragung ist `BREAK` eine Folge von 0-Bits. Diese Bit-Folge ist immer länger als ein Byte und wird als ein einziges `BREAK` interpretiert. In Kapitel 20.8 wird das Senden eines `BREAK` behandelt.

20.4 Terminalflags

Hier werden die Terminalflags aus Tabelle 20.2 detailliert in einer alphabetischen Liste beschreiben. Diese Terminalflags bestehen aus einem oder mehreren Bits, die man setzen oder löschen kann. Neben diesem direkten Setzen/Löschen können die Flags auch mit Masken gesetzt oder gelöscht werden. Jede Maske hat dabei einen Namen und definiert mehrere Bits, die einzeln über eigene Namen angesprochen werden können. Um z.B. die Größe eines Zeichens festzulegen, muß man zunächst die entsprechenden Bits mit der Maske `CSIZE` auf 0 und dann einen der Werte `CS5`, `CS6`, `CS7` oder `CS8` als Zeichengröße setzen.

Programm 20.7 (flagmask.c) zeigt die Verwendung von Masken zum Erfragen oder Setzen von Flags.

```
#include <termios.h>
#include "eighdr.h"

int
main(void)
{
    struct termios    terminal;
    int               bitzahl;

    if (tcgetattr(STDIN_FILENO, &terminal) < 0)
        fehler_meld(FATAL_SYS, "tcgetattr-Fehler");

    bitzahl = terminal.c_cflag & CSIZE;
    if      (bitzahl == CS5)   printf("....5 Bits pro Byte....\n");
    else if (bitzahl == CS6)   printf("....6 Bits pro Byte....\n");
    else if (bitzahl == CS7)   printf("....7 Bits pro Byte....\n");
    else if (bitzahl == CS8)   printf("....8 Bits pro Byte....\n");
    else                       printf("....? Bits pro Byte....\n");

    terminal.c_cflag &= ~CSIZE; /* Bits auf 0 setzen          */
    terminal.c_cflag |= CS8;    /* 8 Bits pro Byte festlegen */

    if (tcsetattr(STDIN_FILENO, TCSANOW, &terminal) < 0)
        fehler_meld(FATAL_SYS, "tcsetattr-Fehler");

    exit(0);
}
```

Programm 20.8 (**flagmask.c**): Verwendung von Masken zum Setzen/Erfragen von Terminalflags

Die sechs von SVR4 angebotenen Verzögerungsflags BSDLY, CRDLY, FFDLY, NLDLY, TABDLY und VTDLY sind auch Masken. Bei allen bedeutet 0-Maske *keine Verzögerung*.

Wenn aber eine Verzögerung festgelegt wird, so bestimmen die beiden Flags OFILL und ODEL, ob der Treiber eine Verzögerung wirklich durchführt oder ob anstelle dessen Füllzeichen übertragen werden.

Nachfolgend werden die einzelnen Flags genauer beschrieben.

ALTWERASE

(c_lflag, BSD) Wenn dieses Flag gesetzt ist, so wird bei der Eingabe von WERASE ein alternativer Wort-Löschungsalgorithmus verwendet: Der Anfang des zu löschenden letzten Wortes ist nicht ein auf ein Leer- oder Tabzeichen folgendes Zeichen, sondern ein auf ein nicht-alphanumerisches Zeichen folgendes alphanumerisches Zeichen.

BRKINT

(`c_iflag`, POSIX.1) Wenn dieses Flag gesetzt und `IGNBRK` nicht gesetzt ist, werden beim Empfang eines `BREAK` die Ein- und Ausgabepuffer geleert und es wird das Signal `SIGINT` generiert, das an die Vordergrundprozeßgruppe geschickt wird, wenn das Terminal ein Kontrollterminal ist.

Wenn weder `IGNBRK` noch `BRKINT` gesetzt ist, wird `BREAK` als das Zeichen `\0` gelesen, außer es ist `PARMRK` gesetzt, in welchem Fall `BREAK` als 3-Byte-Sequenz `\337, \0, \0` gelesen wird.

BSDLY

(`c_oflag`, SVR4) Maske für die Verzögerungsart bei *Backspace*-Zeichen. Die Werte für diese Maske sind `BS0` oder `BS1`.

CCTS_OFLOW

(`c_cflag`, BSD) Wenn gesetzt, so wird das CTS-Ausgabeprotokoll eingeschaltet.

CIGNORE

(`c_cflag`, BSD) Wenn gesetzt, so werden die Kontrollflags ignoriert.

CLOCAL

(`c_cflag`, POSIX.1) Wenn gesetzt, so werden die Modem-Statuszeichen ignoriert, was gewöhnlich bedeutet, daß das Terminal nur lokal betrieben wird. Ist `CLOCAL` nicht gesetzt, so blockiert z.B. ein **open** auf die Terminalgerätedatei so lange, bis das Modem eine Antwort erhält.

CRDLY

(`c_oflag`, SVR4) Maske für die Verzögerungsart bei *Carriage-Return*. Die Werte für diese Maske sind `CR0`, `CR1`, `CR2` oder `CR3`.

CREAD

(`c_cflag`, POSIX.1) Wenn gesetzt, so können Zeichen empfangen werden.

CRTS_IFLOW

(`c_cflag`, BSD) Wenn gesetzt, so wird das RTS-Eingabeprotokoll eingeschaltet.

CRTSCTS

(`c_cflag`, Linux) Ist dieses Flag gesetzt, wird die Hardware-Flußkontrolle (RTS- und CTS-Leitungen) eingesetzt. Bei hohen Übertragungsraten (19200 bps und höher) muß die Hardware-Flußkontrolle verwendet werden, da die Software-Flußkontrolle (über `XON`- und `XOFF`-Zeichen) dann ineffektiv wird.

CSIZE

(`c_cflag`, POSIX.1) Maske für Bitanzahl pro Byte (sowohl für Übertragung als auch für den Empfang). Das Parity-Bit wird hierbei nicht mitgezählt. Die Werte für diese Maske sind `CS5`, `CS6`, `CS7` oder `CS8` für 5, 6, 7 oder 8 Bits pro Byte.

CSTOPB

(`c_cflag`, POSIX.1) Wenn gesetzt, so werden zwei Stop-Bits, ansonsten nur eins verwendet.

ECHO

(`c_lflag`, POSIX.1) Wenn gesetzt, so wird jedes eingegebene Zeichen auch auf dem Terminal ausgegeben. Diese *ECHO*-Funktion kann sowohl für den kanonischen als auch für den nicht-kanonischen Modus eingeschaltet werden.

Ist *ECHO* nicht gesetzt, so haben alle anderen mit *ECHO* beginnenden Flags (außer *ECHONL*) keine Auswirkung, selbst wenn sie gesetzt sind. Diese werden dann so interpretiert, als ob sie ausgeschaltet wären.

ECHOCTL

(`c_lflag`, SVR4 und BSD) Wenn dieses und das *ECHO*-Flag gesetzt sind, werden Steuerzeichen (ASCII-Code 0-37, außer Tab, Neuzeile-Zeichen, *START* und *STOP*) in der Form `^X` am Terminal angezeigt. *X* ist dabei das Zeichen, das sich aus der Addition von 64 auf den aktuellen ASCII-Wert ergibt. So würde z.B. für den ASCII-Wert 5 `^E` ($5+64=69=E$) ausgegeben. Für das ASCII-Zeichen *DELETE* (ASCII-Wert 127) wird `^?` am Terminal ausgegeben.

Wenn *ECHOCTL* nicht gesetzt ist, so werden ASCII-Steuerzeichen auch als solche bei der Ausgabe interpretiert. Dieses Flag kann in beiden Modi (kanonisch und nicht-kanonisch) gesetzt werden.

ECHOE

(`c_lflag`, POSIX.1) Wenn dieses Flag und *ICANON* gesetzt sind, so wird beim *ERASE*-Zeichen das letzte Zeichen in der aktuellen Zeile am Bildschirm gelöscht. Dieses Löschen erfolgt meist durch die Ausgabe der folgenden 3 Zeichen:

Backspace, Leerzeichen, Backspace

Bei einem *WERASE*-Zeichen erfolgt das Löschen des letzten Wortes meist durch eine Folge von solchen 3-Zeichen-Sequenzen.

ECHOK

(`c_lflag`, POSIX.1) Wenn dieses Flag und *ICANON* gesetzt sind, so wird beim *KILL*-Zeichen die ganze aktuelle Zeile gelöscht oder durch Ausgabe von *NL* eine neue Zeile für die Eingabe begonnen.

Wenn *ECHOKE* unterstützt wird, so gilt obiges nur, wenn *ECHOKE* nicht gesetzt ist.

ECHOKE

(`c_lflag`, SVR4 und BSD) Wenn dieses Flag und *ICANON* gesetzt sind, so wird beim *KILL*-Zeichen jedes Zeichen der aktuellen Zeile gelöscht. Wie diese Löschung erfolgt, wird mit den Flags *ECHOE* und *ECHOPRT* festgelegt.

ECHONL

(`c_lflag`, POSIX.1) Wenn dieses Flag und *ICANON* gesetzt sind, wird ein *NL*-Zeichen selbst dann ausgegeben, wenn das Flag *ECHO* nicht gesetzt ist.

ECHOPRT

(`c_lflag`, SVR4 und BSD) Wenn dieses Flag und die beiden Flags *ICANON* und *IECHO* gesetzt sind, dann werden beim *ERASE*-Zeichen (und *WERASE*-Zeichen) alle gelöschten Zeichen gedruckt. Dies ist z.B. auf einen Hardcopy-Terminal nützlich, wenn man genau mitverfolgen möchte, welche Zeichen gelöscht wurden.

FFDLY

(`c_oflag`, SVR4) Maske für die Verzögerungsart bei *FormFeed* (Seitenvorschub). Die Werte für diese Maske sind FF0 und FF1.

FLUSHO

(`c_lflag`, SVR4 und BSD) Wenn gesetzt, so werden Ausgabepuffer geleert. Dieses Flag wird bei einem DISCARD-Zeichen gesetzt und bei einem erneuten DISCARD-Zeichen wieder gelöscht.

HUPCL

(`c_cflag`, POSIX.1) Wenn gesetzt, so wird die Modemverbindung abgebrochen, sobald der letzte Prozeß die entsprechende Gerätedatei schließt.

ICANON

(`c_lflag`, POSIX.1) Wenn gesetzt, so ist kanonischer (zeilenorientierter) Modus eingeschaltet. Im kanonischen Modus sind die speziellen Eingabezeichen EOF, EOL, EOL2, ERASE, KILL, REPRINT, STATUS und WERASE eingeschaltet. Ist ICANON nicht gesetzt, so werden Leseanforderungen direkt vom Eingabepuffer bedient. Ein Lesezugriff kehrt erst dann zurück, wenn mindestens MIN Bytes empfangen oder aber die über TIME festgelegte Zeit verstrichen ist (siehe auch Kapitel 20.7).

ICRNL

(`c_iflag`, POSIX.1) Wenn gesetzt und Flag IGNCR nicht gesetzt ist, so wird ein empfangenes CR-Zeichen in NL umgewandelt.

IEXTEN

(`c_lflag`, POSIX.1) Wenn gesetzt, so ist der erweiterte implementierungsdefinierte Satz von Spezialzeichen eingeschaltet.

IGNBRK

(`c_iflag`, POSIX.1) Wenn gesetzt, so wird ein BREAK in der Eingabe ignoriert (siehe auch BRKINT).

IGNCR

(`c_iflag`, POSIX.1) Wenn gesetzt, so wird ein empfangenes CR-Zeichen ignoriert. Ist IGNCR nicht gesetzt, so kann ein empfangenes CR-Zeichen in NL umgewandelt werden, wenn das Flag ICRNL gesetzt ist.

IGNPAR

(`c_iflag`, POSIX.1) Wenn gesetzt, so werden Bytes mit Paritätsfehlern in der Eingabe ignoriert (siehe auch PARMRK und Hinweise weiter unten).

IMAXBEL

(`c_iflag`, SVR4 und BSD) Wenn gesetzt, so wird ein voller Eingabepuffer durch ein akustisches Signal angezeigt.

INLCR

(`c_iflag`, POSIX.1) Wenn gesetzt, so wird ein empfangenes NL-Zeichen in CR umgewandelt.

INPCK

(c_iflag, POSIX.1) Wenn gesetzt, so ist die Eingabe-Paritätsprüfung eingeschaltet. Wenn INPCK nicht gesetzt ist, so ist diese Paritätsprüfung für die Eingabe ausgeschaltet und PARMRK und IGNPAR haben bei Paritätsfehlern keine Auswirkung (siehe auch Hinweise weiter unten).

ISIG

(c_lflag, POSIX.1) Wenn gesetzt, so werden die Eingabezeichen mit den speziellen Zeichen INTR, QUIT, SUSP und DSUSP, die ein Signal generieren, verglichen. Bei Übereinstimmung wird das entsprechende Signal generiert.

ISTRIP

(c_iflag, POSIX.1) Wenn gesetzt, so wird bei den Eingabebytes das 8. Bit abgeschnitten. Wenn ISTRIP nicht gesetzt ist, bleiben alle 8 Bit erhalten.

IUCLC

(c_iflag, SVR4) Wenn die Flags IUCLC und IEXTEN gesetzt sind, so werden bei der Eingabe Groß- in Kleinbuchstaben umgewandelt.

IXANY

(c_iflag, SVR4 und BSD) Wenn gesetzt, so kann eine angehaltene Ausgabe mit jedem beliebigen Zeichen, also nicht nur mit START, fortgesetzt werden.

IXOFF

(c_iflag, POSIX.1) Wenn gesetzt, so ist das START/STOP-Eingabeprotokoll eingeschaltet. Wenn der Terminalgerätetreiber feststellt, daß der Eingabepuffer voll ist, gibt er das STOP-Zeichen aus. Das sendende Gerät sollte dieses Zeichen erkennen und seine Übertragung anhalten. Wenn dann später Zeichen aus dem Eingabepuffer bearbeitet wurden und somit wieder Platz im Eingabepuffer ist, schickt der Terminaltreiber das START-Zeichen, damit das Gerät mit dem Senden von Daten wieder fortfährt.

Dieses Flag ist nur für serielle Terminals relevant, da Netzwerk-Terminals und lokale Terminals direktere Formen der Flußkontrolle aufweisen. Daneben verwenden jedoch auch serielle Terminals oft eine Hardware-Flußkontrolle, die durch die Kontrollflags (c_cflag) gesteuert wird und somit die Software-Flußkontrolle (mit START und STOP) überflüssig macht.

IXON

(c_iflag, POSIX.1) Wenn gesetzt, ist das START/STOP-Ausgabeprotokoll eingeschaltet. Wenn der Terminalgerätetreiber ein STOP-Zeichen empfängt, so hält er seine Ausgabe an. Beim Empfang eines START-Zeichens wird diese Ausgabe wieder fortgesetzt. Wenn das Flag IXON nicht gesetzt ist, so werden START- und STOP-Zeichen als normale Zeichen behandelt.

MDMBUF

(c_cflag, BSD) Ausgabeprotokoll entsprechend dem Modem-Carrier-Flag.

NLDLY

(c_oflag, SVR4) Maske für die Verzögerungsart bei NL. Die Werte für diese Maske sind NLO oder NL1.

NOFLSH

(c_lflag, POSIX.1) Wenn gesetzt, so wird das voreingestellte Leeren von Puffern ausgeschaltet. Die Voreinstellung ist folgende: Wenn der Terminaltreiber die Signale SIGINT und SIGQUIT generiert, werden die Ein- und Ausgabepuffer geleert, und wenn er das Signal SIGSUSP generiert, wird der Eingabepuffer geleert.

NOKERNINFO

(c_lflag, BSD) Wenn gesetzt, wird beim STATUS-Zeichen keine Statusinformation ausgegeben. Unabhängig von diesem Flag wird jedoch beim STATUS-Zeichen das SIGINFO-Signal immer an die Vordergrundprozeßgruppe geschickt.

OCRNL

(c_oflag, SVR4) Wenn gesetzt, wird ein CR-Zeichen bei der Ausgabe in NL umgewandelt.

OFDEL

(c_oflag, SVR4) Wenn gesetzt, wird bei der Ausgabe als Auffüllzeichen das ASCII-Zeichen DEL benutzt. Wenn nicht gesetzt, wird als Auffüllzeichen bei der Ausgabe das ASCII-Zeichen NUL verwendet (siehe auch Flag OFILL).

OFILL

(c_oflag, SVR4) Wenn gesetzt, wird anstelle einer zeitlichen Verzögerung das entsprechende Auffüllzeichen (DEL oder NUL) übertragen (siehe auch bei den 6 Verzögerungsmasken BSDLY, CRDLY, FFDLY, NLDLY, TABDLY und VTDLY).

OLCUC

(c_oflag, SVR4) Wenn gesetzt, werden bei der Ausgabe Klein- in Großbuchstaben umgewandelt.

ONLCR

(c_oflag, SVR4 und BSD) Wenn gesetzt, wird ein NL-Zeichen bei der Ausgabe in CR-NL umgewandelt.

ONLRET

(c_oflag, SVR4) Wenn gesetzt, wird angenommen, daß NL bei der Ausgabe die gleiche Wirkungsweise wie ein CR hat.

ONOCR

(c_oflag, SVR4) Wenn gesetzt, wird die Ausgabe eines CR in der 0. Spalte einer Zeile unterdrückt.

ONOEOT

(c_oflag, BSD) Wenn gesetzt, werden EOT-Zeichen (*Strg-D*) nicht ausgegeben. Dies ist für Terminals wichtig, die *Strg-D* als HANGUP-Signal (Beendigung der Verbindung) interpretieren.

OPOST

(`c_oflag`, POSIX.1) Wenn gesetzt, wird eine implementierungsdefinierte Ausgabeart eingeschaltet.

OXTABS

(`c_oflag`, BSD) Wenn gesetzt, werden bei der Ausgabe Tabs durch die entsprechende Anzahl von Leerzeichen ersetzt. Dieses Flag hat die gleiche Wirkung, wie wenn die Verzögerungsart für horizontale Tabs (`TABDLY`) auf `XTABS` oder `TAB3` gesetzt wird.

PARENB

(`c_cflag`, POSIX.1) Wenn gesetzt, wird für auszugebende Zeichen die Paritätserzeugung und für empfangene Zeichen die Paritätsprüfung eingeschaltet. Wenn das Flag `PARODD` gesetzt ist, so wird mit ungerader Parität und ansonsten mit gerader Parität gearbeitet (siehe auch Hinweise weiter unten).

PARMRK

(`c_iflag`, POSIX.1) Wenn dieses Flag gesetzt und das Flag `IGNPAR` nicht gesetzt ist, so wird ein Byte mit einem Paritätsfehler vom Prozeß als eine 3-Byte-Sequenz `\337, \0, X` gelesen. `X` ist dabei das fehlerhafte Byte. Wenn `ISTRIP` nicht gesetzt ist, wird ein gültiges Byte `\337` an den Prozeß als eine 2-Byte-Sequenz `\337, \337` weitergeleitet. Ist aber `ISTRIP` gesetzt, wird das Zeichen `\337` mit abgeschnittenem höchstwertigem Bit, also als `\177` gesendet. Wenn weder `IGNPAR` noch `PARMRK` gesetzt ist, wird ein Byte mit einem Paritätsfehler als `\0` gelesen (siehe auch Hinweise weiter unten).

PARODD

(`c_cflag`, POSIX.1) Wenn gesetzt, wird für ein- und ausgehende Zeichen mit ungerader Parität, ansonsten mit gerader Parität gearbeitet (siehe auch `PARENB` und Hinweise weiter unten).

PENDIN

(`c_lflag`, SVR4 und BSD4.4) Wenn gesetzt, so werden bei der Eingabe des nächsten Zeichens noch nicht gelesene Zeichen vom System neu ausgegeben. Dies ist ähnlich zur Eingabe des speziellen Eingabezeichens `REPRINT`.

TABDLY

(`c_oflag`, SVR4) Maske für die Verzögerungsart bei horizontalen Tabs. Die Werte für diese Maske sind `TAB0`, `TAB1`, `TAB2`, `TAB3` oder `XTABS`. Der Wert `TAB3` ist gleich dem Wert `XTABS`. Beide bewirken, daß das System Tabs durch entsprechend viele Leerzeichen ersetzt. Die Tabulatorpositionen sind dabei fest auf 8 Leerzeichen eingestellt und können nicht geändert werden.

TOSTOP

(`c_lflag`, POSIX.1) Wenn gesetzt und Jobkontrolle unterstützt wird, so wird das Signal `SIGTTOU` zu der Prozeßgruppe geschickt, in der gerade ein Hintergrundprozeß versucht, auf sein Kontrollterminal zu schreiben. Die Voreinstellung des Signals `SIGTTOU` ist, daß es alle Prozesse in der Prozeßgruppe anhält. Das Signal `SIGTTOU` wird vom Terminaltreiber nicht generiert, wenn der Hintergrundprozeß, der auf das Kontrollterminal schreibt, dieses Signal entweder ignoriert oder blockiert.

VTDLY

(`c_oflag`, SVR4) Maske für die Verzögerungsart bei vertikalen Tabs. Die Werte für diese Maske sind VT0 oder VT1.

XCASE

(`c_lflag`, SVR4) Wenn dieses Flag und auch das Flag ICANON gesetzt sind, so wird angenommen, daß das Terminal nur Großschreibung kennt, und die ganze Eingabe wird in Kleinschreibung umgewandelt. Um einen Großbuchstaben einzugeben, muß dann diesem ein Backslash (\) vorangestellt werden. Dieses Flag ist heute veraltet, da es wahrscheinlich nur noch wenige Terminals gibt, die nur Großschreibung kennen.

Hinweise zur Parität

Man muß zwischen *Paritätserzeugung* und *-erkennung* und *Eingabe-Paritätsprüfung* unterscheiden.

1. Mit dem Setzen des Flags PARENB wird die Erzeugung und Erkennung von Paritätsbits eingeschaltet. In diesem Fall generiert der Gerätetreiber für das serielle Interface Paritätsbits für abgehende Zeichen und überprüft die Paritätsbits von eingehenden Zeichen.
2. Das Flag PARODD legt fest, ob mit gerader oder ungerader Parität zu arbeiten ist.
3. Wenn ein ankommendes Eingabezeichen ein falsches Paritätsbit hat, so wird geprüft, ob das Flag INPCK gesetzt ist. Ist dieses Flag gesetzt, wird noch das Flag IGNPAR überprüft. Ist dieses Flag gesetzt, wird das Byte mit dem Paritätsfehler ignoriert. Ist dagegen IGNPAR nicht gesetzt, so wird noch das Flag PARMRK überprüft, um festzustellen, welche Zeichen an den lesenden Prozeß weitergeleitet werden sollen.

20.5 Baudraten von Terminals

Der Begriff *Baudrate* steht für die Übertragung von *Bits pro Sekunde*. Obwohl die meisten Terminals für die Eingabe und für die Ausgabe die gleiche Baudrate benutzen, werden Funktionen angeboten, um diese einzeln zu verstellen, wenn die entsprechende Hardware dies zuläßt.

20.5.1 `cfgetispeed`, `cfgetospeed`, `cfsetispeed`, `cfsetospeed` – Erfragen und Setzen der Baudrate

Um die Baudrate für die Ein- oder Ausgabe eines Terminals zu erfragen oder zu ändern, stehen die Funktionen *`cfgetispeed`*, *`cfgetospeed`*, *`cfsetispeed`*, *`cfsetospeed`* zur Verfügung.


```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termzgr);
speed_t cfgetospeed(const struct termios *termzgr);

                                     beide geben zurück: momentan gesetzte Baudrate

int cfsetispeed(struct termios *termzgr, speed_t baudrate);
int cfsetospeed(struct termios *termzgr, speed_t baudrate);

                                     beide geben zurück: 0 (bei Erfolg); -1 bei Fehler
```

Der Rückgabewert der beiden **cfget...**-Funktionen und das **baudrate**-Argument der beiden **cfset...**-Funktionen ist eine der folgenden Konstanten: B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200 oder B38400. Von POSIX nicht vorgeschrieben, aber unter den meisten Systemen (wie z.B. auch unter Linux) vorhandene Konstanten sind: B57600, B115200, B230400 und B460800. Die Konstante B0 steht dabei für *Beendigung der Verbindung*.

Da die Baudraten für Ein- und Ausgabe in der **termios**-Struktur gespeichert sind, muß man zuerst mit **tcgetattr** die **termios**-Struktur für das betreffende Gerät erfragen, bevor man eine der beiden **cfget...**-Funktionen aufrufen kann.

Ebenso gilt, daß Baudraten, die mit den beiden **cfset...**-Funktionen eingestellt wurden, erst nach einem nachfolgenden **tcsetattr**-Aufruf für das entsprechende Gerät aktiviert werden.

Beispiel

Erfragen der eingestellten Baudraten für ein Gerät

Das Programm 20.8 (**baudget.c**) demonstriert, wie die Baudraten für ein Gerät, dessen Name auf der Kommandozeile angegeben ist, erfragt werden können.

```
#include <termios.h>
#include <fcntl.h>
#include "eighdr.h"

static unsigned long baudrate_wert(speed_t baud_konstante)
{
    if (baud_konstante == B0)          return(0);
    else if (baud_konstante == B50)    return(50);
    else if (baud_konstante == B75)    return(75);
    else if (baud_konstante == B110)   return(110);
    else if (baud_konstante == B134)   return(134);
    else if (baud_konstante == B150)   return(150);
    else if (baud_konstante == B200)   return(200);
    else if (baud_konstante == B300)   return(300);
    else if (baud_konstante == B600)   return(600);
    else if (baud_konstante == B1200)  return(1200);
```

```

    else if (baud_konstante == B1800)    return(1800);
    else if (baud_konstante == B2400)    return(2400);
    else if (baud_konstante == B4800)    return(4800);
    else if (baud_konstante == B9600)    return(9600);
    else if (baud_konstante == B19200)   return(19200);
    else if (baud_konstante == B38400)   return(38400);
    else if (baud_konstante == B57600)   return(57600);
    else if (baud_konstante == B115200)  return(115200);
    else if (baud_konstante == B230400)  return(230400);
    else if (baud_konstante == B460800)  return(460800);
    else return(-1);
}

int
main(int argc, char *argv[])
{
    int                fd;
    speed_t            ibaudrate, obaudrate;
    struct termios     terminal;

    if (argc != 2)
        fehler_meld(FATAL, "usage: %s geraetepfad", argv[0]);
    if ( (fd = open(argv[1], O_RDWR | O_NONBLOCK)) < 0)
        fehler_meld(FATAL_SYS, "kann %s nicht oeffnen", argv[1]);
    if (isatty(fd) == 0)
        fehler_meld(FATAL, "%s ist kein tty", argv[1]);

    if (tcgetattr(fd, &terminal) < 0)
        fehler_meld(FATAL_SYS, "tcgetattr-Fehler");
    if ( (ibaudrate = baudrate_wert(cfgetispeed(&terminal))) == -1)
        fehler_meld(FATAL, "ungueltige Eingabe-Baudrate");
    if ( (obaudrate = baudrate_wert(cfgetospeed(&terminal))) == -1)
        fehler_meld(FATAL, "ungueltige Ausgabe-Baudrate");

    printf("Eingabe-Baudrate: %lu\n", ibaudrate);
    printf("Ausgabe-Baudrate: %lu\n", obaudrate);

    exit(0);
}

```

Programm 20.9 (**baudget.c**) Erfragen der Baudraten eines Gerätes

20.6 Zeilensteuerung bei Terminals

20.6.1 **tcdrain, tcflow, tcflush und tcsendbreak** – Funktionen zur Zeilensteuerung eines Terminals

Für die Zeilensteuerung eines Terminals stehen die Funktionen **tcdrain**, **tcflow**, **tcflush** und **tcsendbreak** zur Verfügung.

```
#include <termios.h>

int tcdrain(int fd);
int tcflow(int fd, int aktion);
int tcflush(int fd, int puffer);
int tcsendbreak(int fd, int dauer);
```

Alle vier geben zurück: 0 (bei Erfolg); -1 bei Fehler

Alle vier Funktionen erwarten, daß der übergebene Filedeskriptor *fd* einem Terminal zugeordnet ist, andernfalls geben sie einen Fehler zurück, wobei sie *errno* auf *ENOTTY* setzen.

tcdrain

Die Funktion ***tcdrain*** wartet, bis die ganze Ausgabe übertragen ist.

tcflow

Die Funktion ***tcflow*** ermöglicht die Steuerung der Ein- und Ausgabe. Für das Argument *aktion* muß eine der folgenden vier Konstanten angegeben werden:

TCOOFF

Ausgabe wird suspendiert.

TCOON

Eine zuvor suspendierte Ausgabe wird fortgesetzt.

TCIOFF

Das System überträgt ein STOP-Zeichen, um das Senden von Daten durch das Terminalgerät anzuhalten.

TCION

Das System überträgt ein START-Zeichen, um das Senden von Daten durch das Terminalgerät fortzusetzen.

tcflush

Die Funktion ***tcflush*** leert den Eingabe- und /oder Ausgabepuffer. Die dabei noch in den jeweiligen Puffern befindlichen Daten werden ohne Bearbeitung weggeworfen. Für das Argument *puffer* muß eine der folgenden Konstanten angegeben werden.

TCIFLUSH

Alle Eingabepuffer leeren (die darin enthaltenen Daten wegwerfen).

TCOFLUSH

Alle Ausgabepuffer leeren (die darin enthaltenen Daten wegwerfen).

TCIOFLUSH

Alle Ein- und Ausgabepuffer leeren (die darin enthaltenen Daten wegwerfen).

tcsendbreak

Die Funktion ***tcsendbreak*** schickt für eine bestimmte Dauer eine zusammenhängende Folge von 0-Bytes. Wenn für das Argument *dauer* der Wert 0 angegeben ist, so werden für 0.25 bis 0.5 Sekunden 0-Bytes übertragen. Ist für *dauer* ein von 0 verschiedener Wert angegeben, so ist die Übertragungszeit implementierungsdefiniert (siehe entsprechende Manpage).

POSIX legt nicht die Einheit fest, die für *dauer* anzugeben ist. Der einzige portable Wert ist somit 0. Unter Linux gilt z.B. die folgende Konvention: Die Werte 0 oder 1 legen eine Viertel bis halbe Sekunde fest, der Wert 2 legt eine halbe bis ganze Sekunde fest usw.

20.7 Kanonischer und nicht-kanonischer Modus

Hier wird nochmals etwas detaillierter auf die beiden möglichen Terminalmodi eingegangen.

20.7.1 Kanonischer (zeilenorientierter) Modus

Im kanonischen Modus kehrt der Terminalgerätetreiber beim Lesen erst zurück, wenn eine ganze Zeile eingegeben wurde. Die Rückkehr von einer Leseanforderung erfolgt dabei, wenn eine der folgenden Bedingungen zutrifft:

- ▶ Eine Leseoperation kehrt zurück, wenn die geforderte Anzahl von Bytes gelesen wurde. Das heißt, daß mit einer Leseoperation nicht eine vollständige Zeile gelesen werden muß. Wenn nur ein Teil einer Zeile gelesen wird, so wird bei der nächsten Leseoperation das Lesen beim ersten noch nicht gelesenen Byte fortgesetzt.
- ▶ Eine Leseoperation kehrt zurück, wenn eines der Zeichen NL, EOL, EOL2 oder EOF gelesen wird. Ist ICRNL gesetzt und IGNCR nicht gesetzt, dann beendet auch ein CR eine Leseoperation. Bis auf EOF werden dabei alle anderen dieser Zeichen vom Terminalgerätetreiber an den lesenden Prozeß zurückgegeben.
- ▶ Eine Leseoperation kehrt auch zurück, wenn ein Signal abgefangen wird.

20.7.2 Nicht-kanonischer Modus

Um *nicht-kanonischen Modus* einzuschalten, muß das Flag ICANON (in *c_lflag*-Komponente der Struktur *termios*) ausgeschaltet werden. Im nicht-kanonischen Modus arbeitet ein Terminal nicht mehr zeilenorientiert, was heißt, daß die Sonderbedeutung der speziellen Eingabezeichen ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS und WERASE ausgeschaltet ist.

Im nicht-kanonischen Modus stellt sich nun die Frage, wann das System gelesene Daten an den Aufrufer zurückgeben soll, denn es gibt kein besonderes Zeichen mehr (wie das Neue-Zeilen-Zeichen im kanonischen Modus), das dem Terminaltreiber signalisiert, seine gelesenen Zeichen an den Aufrufer zurückzugeben.

Um dieses Problem zu lösen, teilt man dem System mit, daß es entweder nach einer bestimmten Anzahl von gelesenen Bytes oder nach einer bestimmten Zeitdauer zurückkehrt, je nachdem, was zuerst zutrifft.

Dazu werden im Array `c_cc` der `termios`-Struktur die zwei Variablen `MIN` und `TIME` angeboten. Die entsprechenden Indizes für das Array `c_cc` sind `VMIN` und `VTIME`.

`MIN` legt die minimale Anzahl der Bytes fest, die gelesen werden müssen, bevor eine Leseoperation zurückkehrt. `TIME` legt die Anzahl von Zehntelsekunden fest, die gewartet werden soll, bis eine Leseoperation zurückkehrt.

Dabei existieren vier Möglichkeiten für die Belegung von `MIN` und `TIME`:

1. `MIN > 0` und `TIME > 0`

Eine Leseoperation kehrt entweder nach `TIME`-Zehntelsekunden oder aber nach `MIN` gelesenen Zeichen zurück, je nachdem, was zuerst zutrifft, und gibt die gelesenen Bytes zurück. Es ist sichergestellt, daß immer mindestens ein Byte zurückgegeben wird, wenn die mit `TIME` eingeschaltete Zeitschaltuhr abgelaufen ist, denn die Zeitschaltuhr wird immer erst dann gestartet, wenn das erste Byte gelesen wurde. Dies kann zu einer Blockierung führen.

2. `MIN > 0` und `TIME == 0`

Eine Leseoperation kehrt zurück, wenn `MIN` Bytes gelesen wurden. Dies kann zu einer Blockierung führen.

3. `MIN == 0` und `TIME > 0`

In diesem Fall wird anders als im 1. Fall die Zeitschaltuhr schon zu Beginn der Leseoperation gestartet. Die Leseoperation kehrt hier zurück, wenn entweder ein Byte gelesen oder eben die mit `TIME` eingestellte Zeitschaltuhr abgelaufen ist. Dies bedeutet, daß entweder das gelesene oder kein Byte zurückgegeben wird.

4. `MIN == 0` und `TIME == 0`

Wenn Daten verfügbar sind, so liefert eine Leseoperation die geforderte Anzahl von Bytes. Sind keine Daten verfügbar, so kehrt die Leseoperation sofort zurück und liefert 0 als Rückgabewert.

Tabelle 20.4 faßt diese vier möglichen Kombinationen von `MIN` und `TIME` zusammen, und gibt für jeden möglichen Fall an, wie viele Bytes gelesen werden. *bytezahl* steht dabei für die bei einem **read**-Aufruf geforderte Anzahl von Bytes (3. Argument bei **read**), * steht für *Zeitschaltuhr abgelaufen* und + steht für *Zeitschaltuhr nicht abgelaufen*.

	<i>MIN > 0</i>	<i>MIN == 0</i>
TIME > 0	1. + [<i>MIN</i> , <i>bytezahl</i>] * [<i>1</i> , <i>MIN</i>] (Schaltuhr wird erst beim ersten gelesenen Byte gestartet) ---> Blockierung möglich	3. + [<i>1</i> , <i>bytezahl</i>] * 0 (Zeitschaltuhr wird zu Beginn der Leseoperation gestartet)
TIME == 0	2. [<i>MIN</i> , <i>bytezahl</i>], wenn verfügbar ---> Blockierung möglich	4. [<i>0</i> , <i>bytezahl</i>] (Leseoperation kehrt in jedem Fall ohne jegliches Warten sofort zurück)

Tabelle 20.4: Vier mögliche Fälle für nicht-kanonische Eingabe

Hinweis

MIN legt in allen vier Fällen nur das Minimum fest. Wenn ein Programm mehr als MIN Bytes anfordert, so kann die Leseoperation auch entsprechend mehr Bytes liefern. Dies gilt auch für den 3. und 4. Fall, wo MIN==0 ist.

Da POSIX.1 zuläßt, daß die Indizes VMIN und VTIME die gleichen Werte wie VEOF und VEOL haben, ist in Systemen wie z.B. SVR4, die dies aus Kompatibilitätsgründen realisieren, Vorsicht geboten. Beim Wechsel vom nicht-kanonischen in den kanonischen Modus muß nämlich VEOF und VEOL wiederhergestellt werden. Falls man dies unterläßt und setzt dann c_cc[VMIN] = 1, entspricht dies der Anweisung c_cc[VEOF] = 1, was dazu führt, daß das EOF-Zeichen auf **Strg-A** gesetzt wird. Der beste Weg, um dieses Problem zu umgehen, ist, die ganze termios-Struktur zu sichern, bevor man in den nicht-kanonischen Modus wechselt. Beim Zurückwechseln in den kanonischen Modus, kann man dann mit dieser Sicherungskopie den ursprünglichen termios-Inhalt wiederherstellen.

20.7.3 Umschalten zwischen cbreak- und raw-Terminalmodus

Das folgende Programm 20.9 (cbre_raw.c) enthält die zwei Funktionen tty_cbreak und tty_raw, um ein Terminal in cbreak- oder raw-Modus umzuschalten. Die Begriffe cbreak und raw stammen aus früheren Unix-Versionen. Im jeweiligen Modus hat ein Terminal die nachfolgend aufgezählten Einstellungen.

cbreak-Modus

- ▶ nicht-kanonischer Modus
- ▶ ECHO ausgeschaltet
- ▶ Leseoperationen liefern ein Byte (Fall 2: MIN=1 und TIME=0).

raw-Modus

- ▶ nicht-kanonischer Modus, wobei die Flags ISIG (Generierung von Signalen) und IEXTEN (erweiterter Eingabezeichensatz) ausgeschaltet sind. Zusätzlich ist noch BRKINT (Generierung von Signalen mit BREAK) ausgeschaltet.
- ▶ ECHO ausgeschaltet
- ▶ ICRNL, INPCK, ISTRIP und IXON ausgeschaltet
- ▶ CS8 und PARENB ausgeschaltet
- ▶ OPOST ausgeschaltet
- ▶ Leseoperationen liefern ein Byte (Fall 2: MIN=1, TIME=0).

Neben den beiden Funktionen `tty_cbreak` und `tty_raw` enthält das Programm 20.9 (`cbre_raw.c`) noch drei weitere Funktionen.

`tty_reset`

Zum Zurücksetzen des Terminals in seinen vorherigen Zustand.

`tty_atexit`

kann als Exit-Handler eingerichtet werden, um sicherzustellen, daß das Terminal bei einem **exit** wieder in seinen ursprünglichen Zustand zurückgesetzt wird.

`tty_termios`

ermöglicht das Erfragen der ursprünglichen Terminaleinstellungen.

```
#include <termios.h>
#include <unistd.h>

static struct termios alt_terminal;
static int alt_ttyfd = -1;
static enum { RESET, RAW, CBREAK } tty_modus = RESET;

/*----- tty_cbreak --- Terminal in cbreak-Modus umschalten -----*/
int tty_cbreak(int fd)
{
    struct termios terminal;

    if (tcgetattr(fd, &alt_terminal) < 0)
        return(-1);

    terminal = alt_terminal;

    /* ECHO und kanonischen Modus ausschalten */
    terminal.c_lflag &= ~(ECHO | ICANON);

    /* Fall 2: Immer nur 1 Byte; kein Timer */
    terminal.c_cc[VMIN] = 1;
    terminal.c_cc[VTIME] = 0;

    if (tcsetattr(fd, TCSAFLUSH, &terminal) < 0)
```

```

        return(-1);

tty_modus = CBREAK;
alt_ttyfd = fd;
return(0);
}

/*----- tty_raw --- Terminal in raw-Modus umschalten -----*/
int tty_raw(int fd)
{
    struct termios terminal;

    if (tcgetattr(fd, &alt_terminal) < 0)
        return(-1);

    terminal = alt_terminal;

    /* ECHO, kanonischen Modus, erweitert. Zeichensatz und
       Signalzeichen ausschalten */
    terminal.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    /* kein SIGINT bei BREAK, kein Umwandeln von CR nach NL,
       keine Eingabe-Paritaetspruefung, kein Abschneiden des 8.Bits,
       und kein START/STOP-Ausgabeprotokoll */
    terminal.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    /* alte Zeichengroesse und Paritaetspruefung ausschalten */
    terminal.c_cflag &= ~(CSIZE | PARENB);
    terminal.c_cflag |= CS8; /* 8 Bits pro Zeichen setzen */

    /* Spezielle implementierungsdefinierte Ausgabeart ausschalten */
    terminal.c_oflag &= ~(OPOST);

    /* Fall 2: Immer nur 1 Byte; kein Timer */
    terminal.c_cc[VMIN] = 1;
    terminal.c_cc[VTIME] = 0;

    if (tcsetattr(fd, TCSAFLUSH, &terminal) < 0)
        return(-1);

    tty_modus = RAW;
    alt_ttyfd = fd;
    return(0);
}

/*----- tty_reset --- Terminal in alten Modus zuruecksetzen -----*/
int tty_reset(int fd)
{
    if (tty_modus != CBREAK && tty_modus != RAW)
        return(0);
    if (tcsetattr(fd, TCSAFLUSH, &alt_terminal) < 0)
        return(-1);

    tty_modus = RAW;

```



```

    return(0);
}

/*----- tty_atexit --- mit atexit(tty_atexit) einzurichten -----*/
void tty_atexit(void)
{
    if (alt_ttyfd >= 0)
        tty_reset(alt_ttyfd);
}

/*----- tty_mode --- Urspruenglichen Terminalmodus erfragen -----*/
struct termios *tty_mode(void)
{
    return(&alt_terminal);
}

```

Programm 20.10 (cbre_raw.c): Umschalten zwischen cbreak- und raw-Terminalmodus

Um die Funktionen aus Programm 20.10 (cbre_raw.c) zu testen, wird das folgende Programm 20.11 (termodus.c) verwendet.

```

#include <signal.h>
#include "eighdr.h"

static void signal_faenger(int signr);

int
main(void)
{
    int i;
    char zeich;

    /*---- Einrichten der Signalhandler -----*/
    if (signal(SIGINT, signal_faenger) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhaendler (SIGINT) nicht einrichten");
    if (signal(SIGQUIT, signal_faenger) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhaendler (SIGQUIT) nicht einrichten");
    if (signal(SIGTERM, signal_faenger) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann Signalhaendler (SIGTERM) nicht einrichten");

    /*---- Terminal im raw-Modus -----*/
    printf("Terminal nun im raw-Modus\n"
           "=====\n\n"
           "Gib Zeichen ein (Ende mit Ctrl-D):\n");
    if (tty_raw(STDIN_FILENO) < 0)
        fehler_meld(FATAL_SYS, "tty_raw-Fehler");
    while ( (i = read(STDIN_FILENO, &zeich, 1)) == 1) {
        if ( (zeich &= 0xff) == 4)
            break;
        printf("%x\n", zeich);
    }
    if (i <= 0)
        fehler_meld(FATAL_SYS, "read-Fehler");
}

```

```

if (tty_reset(STDIN_FILENO) < 0)
    fehler_meld(FATAL_SYS, "tty_reset-Fehler");

/*---- Terminal im cbreak-Modus -----*/
if (tty_cbreak(STDIN_FILENO) < 0)
    fehler_meld(FATAL_SYS, "tty_cbreak-Fehler");
printf("\nTerminal nun im cbreak-Modus\n"
       "=====\n\n"
       "Gib Zeichen ein (Ende mit SIGINT (oft Ctrl-C)):\n");
while ( (i = read(STDIN_FILENO, &zeich, 1)) == 1) {
    zeich &= 0xff;
    printf("%x\n", zeich);
}
if (i <= 0)
    fehler_meld(FATAL_SYS, "read-Fehler");

tty_reset(STDIN_FILENO);
exit(0);
}

static void signal_faenger(int signr)
{
    printf(".... Signal empfangen ....\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

```

Programm 20.11 (termodus.c): Test der Funktion aus Programm 20.9 (cbre_raw.c)

Nachdem man diese beiden Programme 20.9 (cbre_raw.c) und 20.10 (termodus.c) kompiliert und gelinkt hat

```
cc -o termodus termodus.c cbre_raw.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ termodus
Terminal nun im raw-Modus
=====

Gib Zeichen ein (Ende mit Ctrl-D):
61          [Eingabe von a]
62          [Eingabe von b]
63          [Eingabe von c]
64          [Eingabe von d]
65          [Eingabe von e]
5           [Eingabe von Ctrl-E]
1f          [Eingabe von Ctrl-?]
[Eingabe von Ctrl-D]

```

```

Terminal nun im cbreak-Modus
=====

```

Gib Zeichen ein (Ende mit SIGINT (oft Ctrl-C)):

```
61      [Eingabe von a]
62      [Eingabe von b]
63      [Eingabe von c]
64      [Eingabe von d]
65      [Eingabe von e]
5       [Eingabe von Ctrl-E]
1f      [Eingabe von Ctrl-?]
[Eingabe von Ctrl-C]
.... Signal empfangen ....
$
```

20.8 Terminalfenstergrößen

In SVR4 und BSD-Unix unterhält der Kern für jedes Terminal und jedes Pseudoterminal eine Struktur `winsize`.

```
struct winsize {
    unsigned short ws_row;    /* Zeilenzahl des Fensters */
    unsigned short ws_col;    /* Spaltenzahl des Fensters */
    unsigned short ws_xpixel; /* horizontale Festgröße (in Pixel) */
    unsigned short ws_upixel; /* vertikale Fenstergröße (in Pixel) */
}
```

Für diese Struktur gelten die folgenden Regeln:

1. Die momentan gesetzten Werte dieser Struktur können mit der Angabe von `TIOCGWINSZ` in einem ***ioctl***-Aufruf erfragt werden.
2. Das Ändern der Werte dieser Struktur ist mit der Angabe von `TIOCGWINSZ` in einem ***ioctl***-Aufruf möglich. Wenn die dabei angegebenen Werte sich von den aktuell gesetzten Werten in der `winsize`-Struktur unterscheiden, wird der Vordergrundprozeßgruppe das Signal `SIGWINCH` geschickt. Die Voreinstellung für `SIGWINCH` ist das Ignorieren dieses Signals.
3. Das Speichern der aktuellen Werte in dieser `winsize`-Struktur und das Generieren des Signals `SIGWINCH`, wenn sich diese Werte ändern, sind die einzigen Aktionen des Kerns für diese Struktur. Die Interpretation der Werte in der Struktur `winsize` liegt vollständig in der Hand des jeweiligen Anwenderprogramms. So wird z.B. der Editor ***vi*** bei einer Änderung der Fenstergröße (Signal `SIGWINCH`) den Bildschirm neu aufbauen.

Beispiel

Erfragen und Ändern der Fenstergröße

Das Programm 20.11 (`window.c`) demonstriert das Erfragen und Ändern der Fenstergröße. Ein Kindprozeß ändert in diesem Programm im Abstand von 2 Sekunden die Fenstergröße. Der Elternprozeß fängt bei jeder Änderung durch das Kind das dabei generierte

Signal ab und gibt die aktuelle Fenstergröße aus. Das Programm 20.11 (window.c) muß mit einem Signal (z.B. *Strg-C* für SIGINT) abgebrochen werden.

```
#include <signal.h>
#include <termios.h>
#ifdef TIOCGWINSIZE
#include <sys/ioctl.h> /* fuer BSD notwendig */
#endif
#include "eighdr.h"

static void setze_wingroesse(int fd, int zeilzahl, int spaltzahl);
static void erfrage_wingroesse(int fd);
static void sigwinch_faenger(int signr);

int
main(void)
{
    pid_t pid;

    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sigwinch_faenger) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann sigwinch_faenger nicht einrichten");

    erfrage_wingroesse(STDIN_FILENO);

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");
    else if (pid > 0) /*--- Elternprozess -----*/
        while (1)
            pause();
    else { /*--- Kindprozess -----*/
        if (signal(SIGWINCH, SIG_IGN) == SIG_ERR)
            fehler_meld(FATAL_SYS, "kann SIGWINCH nicht ignorieren");
        sleep(2);
        setze_wingroesse(STDIN_FILENO, 25, 40);
        sleep(2);
        setze_wingroesse(STDIN_FILENO, 10, 120);
        sleep(2);
        setze_wingroesse(STDIN_FILENO, 5, 20);
        sleep(2);
        setze_wingroesse(STDIN_FILENO, 25, 80);
    }
}

static void setze_wingroesse(int fd, int zeilzahl, int spaltzahl)
{
    struct winsize groesse;

    groesse.ws_row = zeilzahl;
    groesse.ws_col = spaltzahl;
    if (ioctl(fd, TIOCSWINSZ, (char *) &groesse) < 0)
        fehler_meld(FATAL_SYS, "TIOCSWINSZ-Fehler");
}
```

```

}

static void erfrage_wingroesse(int fd)
{
    struct winsize  groesse;

    if (ioctl(fd, TIOCGWINSZ, (char *)&groesse) < 0)
        fehler_meld(FATAL_SYS, "TIOCGWINSZ-Fehler");
    printf("%d Zeilen, %d Spalten\n", groesse.ws_row, groesse.ws_col);
}

static void sigwinch_faenger(int signr)
{
    printf("          .....SIGWINCH empfangen\n");
    erfrage_wingroesse(STDIN_FILENO);
    if (signal(SIGWINCH, sigwinch_faenger) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann sigwinch_faenger nicht einrichten");
}

```

Programm 20.12 (*window.c*): Erfragen und Ändern der Fenstergröße

Nachdem man dieses Programm 20.11 (*window.c*) kompiliert und gelinkt hat

```
cc -o window window.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```

$ window
25 Zeilen, 80 Spalten
          .....SIGWINCH empfangen
25 Zeilen, 40 Spalten
          .....SIGWINCH empfangen
10 Zeilen, 120 Spalten
          .....SIGWINCH empfangen
25 Zeilen, 80 Spalten
          .....SIGWINCH empfangen
5 Zeilen, 20 Spalten
          .....SIGWINCH empfangen
Strg-C
$

```

20.9 termcap, terminfo und curses

Um typische Aktionen auf Terminals durchzuführen, wie z.B. Bildschirm löschen oder Cursor positionieren, werden die beiden Datenbanken *termcap* und *terminfo* und die *curses*-Bibliothek angeboten. Hier wird ein kurzer Einblick in diese Datenbank und Bibliothek gegeben. Eine vollständige Beschreibung würde den Rahmen dieses Buches sprengen. Zudem verlieren *termcap*, *terminfo* und *curses* zunehmend an Bedeutung, da man heute immer mehr Programme mit vollgraphischer Oberfläche anbietet.

20.9.1 termcap – Eine Datenbank von Terminaleigenschaften

termcap steht für *Terminal Capability* (Terminalfähigkeit). In der Datei */etc/termcap* sind die Eigenschaften von einer Vielzahl von Terminals hinterlegt, wie Zeilen- und Spaltenzahl oder ob ein Terminal Klein- und Großschreibung unterscheiden kann. Neben diesen Eigenschaften sind in */etc/termcap* noch die erforderlichen Terminalsteuersequenzen für typische Terminaloperationen (wie z.B. Bildschirm löschen, Cursor positionieren usw.) angegeben.

Der Vorteil dieser *termcap*-Methode ist, daß die speziellen Terminalsteuersequenzen nicht in den C-Programme »eingebrennt« werden müssen. Statt dessen verwenden die entsprechenden C-Programme fest vordefinierte Namen, wie z.B. **cl** (für *clear screen*). Da nun die zum »Bildschirm löschen« erforderlichen Steuerzeichen beim jeweiligen Terminal in der Textdatei */etc/termcap* angegeben sind und sich nicht im C-Code befinden, wird ein solches bildschirmorientiertes Programm wie z.B. **vi** auch für andere Terminals portierbar. Falls zu einem bestimmten Terminal noch kein Eintrag in */etc/termcap* existieren sollte, so kann man diesen nachträglich dort hinzufügen.

Diese *termcap*-Methode, die in BSD-Unix verwendet wird, hat den Nachteil, daß die Datei */etc/termcap* ständig anwächst, da immer mehr Terminals dort aufgenommen werden, was dazu führt, daß es immer länger dauert, bis ein bestimmter Eintrag gefunden wird.

20.9.2 terminfo – Eine andere Datenbank von Terminaleigenschaften

Der zuvor erwähnte Nachteil von *termcap* und die schlechte Namensgebung (Zweizeichennamen sind nicht selbsterklärend) führte dazu, daß man die *terminfo*-Methode mit der zugehörigen *curses*-Bibliothek, die in SVR4 verwendet wird, entwickelte.

Die Datenbank *terminfo* umfaßt mehrere Binärdateien, nämlich eine pro Terminal. Die Binärdateien befinden sich normalerweise in Subdirectories des Directories */usr/lib/terminfo*. Die Namen dieser Subdirectories in */usr/lib/terminfo* sind Buchstaben und Ziffern, unter denen sich die jeweiligen Beschreibungen der Terminals befinden, deren Name mit diesem Buchstaben oder dieser Ziffer beginnt.

```
$ ls -CF /usr/lib/terminfo
1/ 3/ 5/ 7/ 9/ X/ b/ d/ f/ h/ j/ l/ n/ p/ r/ t/ v/ x/
2/ 4/ 6/ 8/ P/ a/ c/ e/ g/ i/ k/ m/ o/ q/ s/ u/ w/ z/
$
```

Die beiden Datenbanken *termcap* und *terminfo* verwenden die gleichen Namen für die entsprechenden Terminals. Beispielsweise hat die Linux-Konsole sowohl in *termcap* als auch in *terminfo* den Namen *linux* (im Directory */usr/lib/terminfo/l/*). Welches Terminal von einem Programm zu benutzen ist, wird über die Environmentvariable *TERM* festgelegt.

20.9.3 curses – Eine Bibliothek für Semigraphik unter Unix

Die unter Unix standardmäßig angebotene Headerdatei `< curses.h >` bietet eine Vielzahl von Funktionen zur Bildschirmsteuerung an. Da eine Beschreibung all dieser Funktionen den hier gesteckten Rahmen sprengen würde, werden nur die grundlegenden Funktionen vorgestellt, um so einen einfacheren Einstieg in die Unix-Manuals zu ermöglichen.

Damit die betreffenden Funktionen aus `<urses.h>` dazugebunden werden, ist folgender Compiler-Aufruf in Unix notwendig:

```
cc -o prog prog.c -lcurses
```

20.9.4 curses-Modus ein- und ausschalten

initscr(void)

schaltet den Bildschirm in **curses-Modus**. **initscr** muß immer aufgerufen werden, bevor die nachfolgend vorgestellten **curses-Routinen** verwendet werden können.

endwin(void)

schaltet den Bildschirm aus dem `curses`-Modus wieder zurück in den normalen Textmodus. Dieses Zurückschalten sollte immer vor dem Verlassen eines Programms geschehen.

20.9.5 Bildschirm löschen, Cursor positionieren und Text ausgeben

Der Bildschirm entspricht bei der Bildschirmsteuerung in Unix einem x-y-Koordinatensystem, dessen Nullpunkt die linke obere Ecke ist ($y=0, x=0$). Der Cursor kann dabei nun unter Angabe eines (x,y)-Werts positioniert werden.

```

+-----> x (normalerweise von 0 bis COLS-1)
|
|
|
|
|
v
y (normalerweise von 0 bis LINES-1)

```

`LINES` und `COLS` sind dabei Variablen, die immer die maximal mögliche Anzahl von Zeilen und Spalten am jeweiligen Bildschirm enthalten.

Folgende Funktionen ermöglichen das Löschen des gesamten Bildschirms, Positionieren des Cursors und Ausgeben von Text an bestimmten Bildschirmpositionen:

clear(void)* und *erase(void)

löschen den gesamten Bildschirm und setzen den Cursor in die obere linke Ecke (0,0).

move(int y, int x)

positioniert den Cursor in der *y*. Zeile auf die *x*. Spalte. *x* muß dabei ein Wert aus dem Bereich 0 bis COLS-1 und *y* ein Wert aus dem Bereich 0 bis LINES-1 sein. Für die meisten Bildschirme gilt, daß LINES gleich 25 und COLS gleich 80 ist. Liegt einer der angegebenen Werte *x* oder *y* außerhalb des Fensters, dann hat ein **move**-Aufruf keinerlei Auswirkung.

addch(int zeich)

gibt das Zeichen *zeich* an der momentanen Cursorposition aus. **addch** benutzt dabei die momentan gesetzten Attribute (siehe weiter unten). Die Angabe von `\n` als *zeich* bewirkt, daß der Rest der Zeile gelöscht und der Cursor in die nächste Zeile bewegt wird.

addstr(char *string)

gibt den String *string* an der momentanen Cursorposition aus. **addstr** benutzt dabei **addch**, um jedes einzelne Zeichen des Strings *string* auszugeben.

printw(char *format, argument(e))

bewirkt eine formatierte Textausgabe auf dem Bildschirm. Die *format*-Angabe entspricht der bei **printf**. **printw** ruft genauso wie **addstr** für die Ausgabe jedes einzelnen Zeichens **addch** auf.

mvprintw(int y, int x, char *format, argument(e))

bewirkt eine formatierte Textausgabe in der *x*. Spalte der *y*. Zeile.

mvaddstr(int y, int x, char *string)

bewegt den Cursor auf die *x*. Spalte in der *y*. Zeile und gibt dort *string* aus.

mvaddch(int y, int x, int zeich)

bewegt den Cursor auf *x*. Spalte in *y*. Zeile und gibt dort Zeichen *zeich* aus.

refresh(void)

sollte nach jeder Veränderung des Bildschirms, z.B. mit **printw**, **addstr**, **clear**, **clrtoeol**... aufgerufen werden, um sie wirklich auf dem Bildschirm erscheinen zu lassen.

Beispiel*Demonstrationsbeispiel zur Cursorpositionierung*

Das nachfolgende Programm 20.12 (curpos.c) demonstriert die Wirkung einiger dieser Funktionen.

```
#include <curses.h>

int
main(void)
{
    /*----- curses-Modus einschalten -----*/
    initscr();

    /*----- In der 1.Spalte der 1.Zeile Text ausgeben -----*/
    move(0, 0);
    addstr("_<---- Position (0,0)"); refresh();

    /*----- In der 21.Spalte der 6.Zeile Text ausgeben -----*/
    move(5, 20);
    addstr("_<---- Position (5,20)"); refresh();

    /*----- In der 50.Spalte der 20.Zeile Cursorposition ausgeben --*/
    mvprintw(19, 49, "_<---- Position (19,49)"); refresh();

    /*----- In der 58.Spalte der 25.Zeile Text ausgeben -----*/
    move(24, 57);
    printw("Position (%d,%d) ---->", LINES-1, COLS-1); refresh();

    /*----- An der letzten Bildschirmstelle Position ausgeben -----*/
    move(LINES-1, COLS-1);
    addch('_'); refresh();

    /*----- Abschluss-Arbeiten (cleanup) -----*/
    getch();          /* Beliebige Taste einlesen */
    erase(); refresh(); /* Bildschirm loeschen */
    endwin();         /* curses-Modus wieder verlassen */
    exit(0);
}
```

Programm 20.13 (curpos.c): Demonstrationsbeispiel zur Cursorpositionierung

Das Programm 20.12 (curpos.c) liefert folgende Bildschirmausgabe.

```

_<---- Position (0,0)

        _<---- Position (5,20)

                                _<---- Position (19,49)

                                                Position (24,79) ---->_

```

Programmende mit Löschen des Bildschirms nach einem Tastendruck.

20.9.6 Attribute für Textausgaben festlegen

attrset(int attribut)

attrset schaltet das Attribut *attribut* für folgende Textausgaben ein. Alle anderen momentan gesetzten Attribute werden ausgeschaltet. Die wichtigsten Namen, die für den Parameter *attribut* angegeben werden können, sind:

A_STANDOUT

Text am jeweiligen Bildschirm hervorheben; z.B. durch fette oder inverse Ausgabe

A_UNDERLINE

Unterstreichen

A_REVERSE

Invers

A_BLINK

Blinken

A_DIM

»halbe Intensität«

A_BOLD
Fett

Mehrere Attribute können auch gleichzeitig gesetzt werden, dazu müssen die entsprechenden Namen mit | (bitweises OR) verknüpft werden. Um z.B. gleichzeitig Unterstreichen und Blinken einzuschalten, muß

```
attrset(A_UNDERLINE|A_BLINK)
```

aufgerufen werden. Mit dem Aufruf `attrset(0)` werden alle Attribute ausgeschaltet.

attron(int attribut)

attron schaltet zusätzlich zu den bereits gesetzten Attributen noch das Attribut `attribut` für folgende Textausgaben ein.

attroff(int attribut)

attroff schaltet nur das Attribut `attribut` aus, die restlichen Attribute bleiben davon unbetroffen.

standout(void)

entspricht dem Aufruf `attron(A_STANDOUT)`.

standend(void)

entspricht dem Aufruf `attrset(0)`.

Beispiel

Wirkung der unterschiedlichen Attribute

```
#include <curses.h>

int
main(void)
{
    initscr();
    clear();

    move(2, 30);
    attrset(A_STANDOUT);
    addstr("Hervorheben von Text an diesem Bildschirm");

    move(4, 30);
    attrset(A_UNDERLINE);
    addstr("Unterstreichen");

    move(6, 30);
    attrset(A_REVERSE);
    addstr("Inverse Darstellung");
```

```
move(8, 30);
attrset(A_BLINK);
addstr("Blinken");

move(10, 30);
attrset(A_DIM);
addstr("Halbe Intensitaet");

move(12, 30);
attrset(A_BOLD);
addstr("Fette Schrift");

move(14, 30);
attrset(A_UNDERLINE|A_REVERSE);
addstr("Invers und Unterstrichen");

getch();
endwin();
exit(0);
}
```

Programm 20.14 (**attr.c**): Wirkung der unterschiedlichen Attribute

Mögliche Ausgabe durch dieses Programm 20.13 (`attr.c`), wobei die inverse Darstellung und das Blinken bei dieser Ausgabe nicht erkennbar sind:

```
Hervorheben von Text an diesem Bildschirm

Unterstreichen

Inverse Darstellung

Blinken

Halbe Intensitaet

Fette Schrift

Invers und Unterstrichen
```

20.9.7 Einlesen von der Tastatur

echo(void)

bewirkt, daß nachfolgende Eingaben am Bildschirm angezeigt werden. Dies ist die Voreinstellung nach dem `initscr`-Aufruf.

noecho(void)

bewirkt, daß nachfolgende Eingaben am Bildschirm nicht angezeigt werden.

cbreak(void)

bewirkt, daß jedes einzelne Zeichen sofort eingelesen wird und nicht in einem Puffer zwischengespeichert wird, der erst bei Eingabe von Return abgearbeitet wird. Dies ist die Voreinstellung nach dem ***initscr***-Aufruf.

nocbreak(void)

bewirkt, daß jedes einzelne Zeichen zunächst in einem Puffer gespeichert wird, der erst bei Eingabe von Return, abgearbeitet wird.

getch(void)

liest ein Zeichen von der Tastatur. Abhängig davon, ob zuvor ***echo*** oder ***noecho*** aufgerufen wurde, wird dabei das eingegebene Zeichen am Bildschirm angezeigt oder nicht. Ob das Zeichen dem Programm sofort zur Verfügung gestellt wird oder erst nach der Eingabe von Return hängt davon ab, ob zuvor ***cbreak*** oder ***nocbreak*** aufgerufen wurde.

scanw(char *format, argument(e))

liest und formatiert Eingaben von der Tastatur. Die *format*-Angabe entspricht dabei der bei ***scanf***.

20.9.8 Funktions- und Positionierungstasten

Für alle Funktionstasten und sonstigen Steuertasten werden vordefinierte Namen angeboten. Um diese Namen verwenden zu können, muß zuvor `keypad(stdscr, 1)` aufgerufen werden. Einige der vielen in `<curses.h>` vordefinierten Namen sind:

<i>Name</i>	<i>steht für</i>
KEY_DOWN	Pfeil »nach unten«
KEY_UP	Pfeil »nach oben«
KEY_LEFT	Pfeil »nach links«
KEY_RIGHT	Pfeil »nach rechts«
KEY_F(1)	Funktionstaste <i>F1</i>
KEY_F(2)	Funktionstaste <i>F2</i>
.....	

Beispiel*Kombinationen von echo und noecho mit cbreak und nocbreak*

Das folgende Programm 20.14 (getch.c) zeigt alle Kombinationen von **echo** und **noecho** mit **cbreak** und **nocbreak**.

```
#include <stdio.h>
#include <curses.h>

int
main(void)
{
    char zeich;

    initscr();
    clear();
    refresh();
    /*----- nocbreak und noecho -----*/
    nocbreak(); /* Einzelne Zeichen erst nach RETURN einlesen */
    noecho();   /* Echo ausschalten */
    move(0, 0);
    printw("Gib ein 1. Zeichen ein (Abschluss mit CR): ");
    zeich=getch(); /* Eingeb. Zeichen nach RETURN einlesen und nicht zeigen */
    getch(); /* Dummy-getch, um RETURN zu ueberlesen */
    move(1, 0);
    printw("Dein eingegebenes Zeichen war '%c'", zeich);
    refresh();

    /*----- nocbreak und echo -----*/
    echo(); /* Echo einschalten */
    move(3, 0);
    printw("Gib ein 2. Zeichen ein (Abschluss mit CR): ");
    zeich=getch(); /* Eingeb. Zeichen nach RETURN einlesen und zeigen */
    getch(); /* Dummy-getch, um RETURN zu ueberlesen */
    move(4, 0);
    printw("Dein eingegebenes Zeichen war '%c'", zeich);
    refresh();

    /*----- cbreak und noecho -----*/
    cbreak(); /* Einzelne Zeichen sofort einlesen */
    noecho(); /* Echo ausschalten */
    move(6, 0);
    printw("Gib ein 3. Zeichen ein: ");
    zeich=getch(); /* Eingeb. Zeichen sofort einlesen und nicht zeigen */
    printw("\nDein eingegebenes Zeichen war '%c'\n", zeich);
    refresh();

    /*----- cbreak und echo -----*/
    echo(); /* Echo einschalten */
    move(9, 0);
    printw("Gib ein 4. Zeichen ein: ");
    zeich=getch(); /* Eingeb. Zeichen sofort einlesen und zeigen */
}
```

```

printw("\nDein eingegebenes Zeichen war '%c'\n", zeich);
refresh();

endwin();

exit(0);
}

```

Programm 20.15 (*getch.c*): Kombinationen von *echo* und *noecho* mit *cbreak* und *nocbreak*

Möglicher Ablauf des Programms 20.14 (*getch.c*):

Gib ein 1. Zeichen ein (Abschluss mit CR): a 	[Eingabe von a wird nicht angezeigt]
Dein eingegebenes Zeichen war 'a'	
Gib ein 2. Zeichen ein (Abschluss mit CR): b 	[Eingabe von b wird angezeigt]
Dein eingegebenes Zeichen war 'b'	
Gib ein 3. Zeichen ein: c	[Eingabe von c wird nicht angezeigt]
Dein eingegebenes Zeichen war 'c'	
Gib ein 4. Zeichen ein: d	[Eingabe von d wird angezeigt]
Dein eingegebenes Zeichen war 'd'	

Beispiel

Abfragen von Funktions- und Steuertasten

Das folgende Programm 20.15 (*taste.c*) zeigt, wie man Funktionstasten und sonstige Steuertasten in einem Programm einlesen und erkennen kann:

```

#include <curses.h>

#define ESC 27

int
main(void)
{
    int zeich,
        weiter=1;

    initscr();
    keypad(stdscr, 1); /* 1 schaltet automat. Steuerzeichen-Erkennung aus*/
    cbreak();
    noecho();
    while (weiter) {
        clear();
        printw("Funktions-Tasten und Pfeil-Tasten\n");
        printw("=====\\n\\n");
        printw("Druecke eine dieser Tasten\\n\\n");
        zeich=getch();
        switch(zeich) {
            case KEY_UP:    printw("Du hast 'Pfeil nach oben' gedrueckt"); break;

```

```

        case KEY_DOWN: printf("Du hast 'Pfeil nach unten' gedrueckt"); break;
        case KEY_LEFT: printf("Du hast 'Pfeil nach links' gedrueckt"); break;
        case KEY_RIGHT: printf("Du hast 'Pfeil nach rechts' gedrueckt"); break;
        case KEY_F(1): printf("Du hast F1 gedrueckt"); break;
        case KEY_F(2): printf("Du hast F2 gedrueckt"); break;
        case KEY_F(3): printf("Du hast F3 gedrueckt"); break;
        case KEY_F(4): printf("Du hast F4 gedrueckt"); break;
        case KEY_F(5): printf("Du hast F5 gedrueckt"); break;
        case KEY_F(6): printf("Du hast F6 gedrueckt"); break;
        case KEY_F(7): printf("Du hast F7 gedrueckt"); break;
        case KEY_F(8): printf("Du hast F8 gedrueckt"); break;
        case KEY_F(9): printf("Du hast F9 gedrueckt"); break;
        case ESC:      printf("ESC gedrueckt\n"); weiter=0; break;
        default:      printf("Taste mir nicht bekannt"); break;
    }
    refresh();
    if (weiter) {
        printf("\n\nWeiter mit beliebiger Taste.....");
        getch();
    }
}
endwin();

exit(0);
}

```

Programm 20.16 (**taste.c**): Abfragen von Funktions- und Steuertasten

20.9.9 Bildschirminhalte verschieben und Bildausschnitte kopieren

deleteln(void)

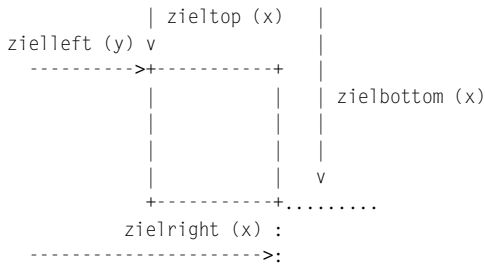
löscht Zeile der momentanen Cursorposition vom Bildschirm und rollt alle folgenden Zeilen des Fensters nach oben.

insertln(void)

fügt an Cursorposition eine Leerzeile ein. Alle folgenden Bildschirmzeilen werden nach unten gerollt; die letzte Zeile verschwindet dabei.

copywin(*stdscr, stdscr, int top, int left,*
int zieltop, int zielleft, int zielbottom, int zielright, 0)

kopiert einen rechteckigen Bildschirmausschnitt an eine andere Stelle. Die Koordinaten sind relativ zur oberen linken Ecke (0,0) des Bildschirms. Die ersten beiden Koordinaten definieren die linke obere Ecke des Quellrechtecks. Die nächsten vier Koordinaten definieren das Zielrechteck:

***clrtoeol***(void)

löscht alle Zeichen von der momentanen Cursorposition bis zum Zeilenende. Löschen bedeutet dabei genauso wie bei **clear**: Auffüllen mit Leerzeichen. **clrtoeol** verändert niemals die Cursorposition.

clrrobot(void)

löscht alle Zeichen von der momentanen Cursorposition bis zum Bildschirmende. Löschen bedeutet dabei genauso wie bei **clear**: Auffüllen mit Leerzeichen. **clrtoebot** verändert niemals die Cursorposition.

Beispiel

Schnee und Luftballone

Das nachfolgende Programm 20.16 (`bal flock.c`) simuliert unter Unix das Aufsteigen von Luftballons bzw. das Fallen von Schneeflocken, je nachdem was der Benutzer wünscht.

```
#include <stdio.h>
#include < curses.h>
#include <stdlib.h>
#include <time.h>

int
main(void)
{
    char wahl;
    int i, z=0;

    srand(time(NULL)+getpid()); /* Zufallszahlengenerator initialisieren */

    /*--- Einlesen, ob Luftballone oder Schneeflocken gewuenscht -----*/
    initscr();
    while (1) {
        clear();
        printw("1 : Luftballone steigen lassen\n");
        printw("2 : Schneeflocken fallen lassen\n\n");
        printw("Was wollen Sie tun ? ");
        wahl=getch();
        refresh();
    }
}
```

```

        if (wahl=='1' || wahl=='2')
            break;
    }

    /*----- Simulieren der Luftballone bzw. Schneeflocken -----*/
    clear();
    do {
        for (i=0 ; i<COLS ; i++) { /* Fuer jede Spalte */
            if (rand()%100<=3) { /* mit 4% Wahrscheinlichkeit */
                if (wahl=='1') /* Luftballon oder Schneeflocke */
                    mvaddch(LINES-1,i,'o');
                else
                    mvaddch(0,i,'*');
            }
        }
        move(0,0);
        if (wahl=='1')
            deleteln(); /* Bei Luftballonen: Bild nach oben ziehen */
        else
            insertln(); /* Bei Schneeflocken: Bild nach unten ziehen */
        refresh();
    } while (++z<=100);

    endwin();

    exit(0);
}

```

Programm 20.17 (**ballflock.c**): Schnee und Luftballone

Beispiel

Männlein im Walde

Das nachfolgende Programm 20.17 (**kuckkuck.c**) malt zunächst ein »Männchen« in der oberen linken Ecke des Bildschirms. Dieses Bild kopiert es dann an eine zufällige Stelle am Bildschirm und löscht den Rest des Bildschirms. Danach wird das neue Bild an eine zufällige Bildschirmstelle kopiert und der restliche Bildschirm gelöscht. Dieser Vorgang wird wiederholt, bis der Benutzer eine beliebige Taste drückt.

In diesem Programm werden die zufälligen Koordinaten so gewählt, daß sich beide Bilder (altes und neue) nie überlappen, da **copywin** zuerst das Zielrechteck löscht. Würden sich nun die beiden Bilder überlappen, würde ein Teil des alten Bild gelöscht, bevor dieses kopiert wird.

```

#include <stdio.h>
#include <curses.h>
#include <stdlib.h>
#include <time.h>

int
main(void)
{

```

```

int  altx=0, alty=0,
     x=0, y=0,
     i, j,
     z=0;

srand(time(NULL)); /* Zufallszahlengenerator initialisieren */

/*----- Maennchen in obere linke Ecke zeichnen -----*/
initscr();
move(altx,alty);
printw("  0  \n");
printw("--0--\n");
printw(" /  \ \n");
refresh();

/*----- Maennchen zufaellig am Bildschirm herumspringen lassen -----*/
do {
    while (abs(y-alty)<=3 && abs(x-altx)<=5) {
        x=rand()%(COLS-5); /* Neue Koordinaten zufaellig bestimmen */
        y=rand()%(LINES-3);
    }
    /* altes Bild dorthin kopieren */
    copywin(stdscr,stdscr,alty,altx,y,x,y+2,x+4,0);
    refresh();
    for (j=0 ; j<LINES ; j++) /* Altes Maennchen-Bild loeschen */
        if (j>=y && j<=y+2) {
            for (i=0 ; i<x ; i++) {
                mvaddch(j,i,' ');
                refresh();
            }
            move(j,x+5); refresh();
            clrtoeol(); refresh();
        } else if (j<y) {
            move(j,0); refresh();
            clrtoeol(); refresh();
        } else {
            move(j,0); refresh();
            clrtobot(); refresh();
            break;
        }
    altx=x; /* Koordinaten des aktuellen Bilds in altx und alty festhalten */
    alty=y;
} while (++z<=50);
endwin();

exit(0);
}

```

Programm 20.18 (kuckkuck.c): Männlein im Walde

Dies war nur ein kleiner Auszug aus der Vielzahl von Funktionen, die `<curses.h>` zur Verfügung stellt. Mehr Information hierzu kann in der Manpage `curses` nachgeschlagen werden.

20.10 S-Lang – Eine Alternative zu curses unter Linux

Die unter Linux angebotene Bibliothek S-Lang ist eine Alternative zu der im vorherigen Kapitel vorgestellten curses-Bibliothek für Semigraphik-Programmierung. Da S-Lang auch für DOS angeboten wird, wird S-Lang meist für Anwendungen verwendet, die sowohl unter Linux/Unix als auch unter DOS lauffähig sein sollen. Programme, die S-Lang benutzen, sollten

```
#include <slang.h>
```

angeben. Sollte sich die Headerdatei `<slang.h>` nicht im Directory `/usr/include` befinden, sondern in einem anderen Directory, wie z.B. `/usr/include/slang`, dann muß dieses Subdirectory mitangegeben werden, wie z.B.:

```
#include <slang/slang.h>
```

Beim Kompilieren eines Programms, das S-Lang benutzt, muß die Bibliothek `libslang.a` angegeben werden:

```
cc -o prog prog.c ... -lslang
```

20.10.1 S-Lang-Modus ein- und ausschalten

Das Einschalten des S-Lang-Modus geschieht mit folgender Funktion

```
int SLang_init_tty(int intr_zeich, int fluss_ctrl, int nachbehand);
```

Der erste Parameter `intr_zeich` legt das Unterbrechungszeichen fest, wobei -1 bedeutet, daß das aktuelle Unterbrechungszeichen des Terminals (üblicherweise Strg-C) zu verwenden ist.

Der zweite Parameter `fluss_ctrl` schaltet die Flußkontrolle ein (1) oder aus (0). Bei Terminals legt die Flußkontrolle fest, ob der Benutzer die Ausgabe anhalten (meist mit Strg-S) und anschließend wieder fortsetzen kann (meist mit Strg-Q). Eine solche Flußkontrolle ist zwar für zeilenorientierte Anwendungen empfehlenswert, aber Programme, die S-Lang verwenden, sind meist bildschirmorientiert, so daß hier eine Flußkontrolle nicht sehr sinnvoll ist.

Der dritte Parameter `nachbehand` schaltet den Nachbehandlungsmechanismus für Ausgaben ein (1) oder aus (0).

Typischerweise ruft man diese Funktion ***Slang_init_tty*** wie folgt auf:

```
Slang_init_tty (-1, 0, 1);
```

Bevor ein Programm, das den S-Lang-Modus eingeschaltet hat, sich beendet, muß es wieder in den normalen Textmodus zurückschalten. Dazu steht die folgende Funktion zur Verfügung:

```
void SLang_reset_tty(void);
```

Vorsicht ist geboten, wenn ein Programm sich nicht normal beendet oder es mit Strg-Z (Signal SIGTSTP) angehalten wird. Der Programmierer sollte für diese Fälle die Signale abfangen und bei den entsprechenden Signalhandlern in den Textmodus zurückschalten. Wenn es vorkommt, daß ein Programm, das in den S-Lang-Modus umgeschaltet hat, nicht korrekt in den Textmodus zurückschaltet, bevor es sich beendet oder angehalten wird, kann das Terminal eventuell nicht mehr richtig benutzbar sein. In diesem Fall ist **reset** oder **stty sane** auf der Kommandozeile einzugeben, um das Terminal wieder funktionsfähig zu machen.

S-Lang bietet zwei Gruppen von Ausgabefunktionen an:

- Funktionen zum direkten Zugriff auf das Terminal (SLtt-Funktionen)
- Funktionen zur Bildschirmverwaltung (SLsmg-Funktionen; *Screen Management*)

Die SLtt-Funktionen arbeiten direkt mit dem Terminal. Zu dieser Gruppe gehören unter anderem Funktionen, mit denen man Spezifika eines Terminals erfragen, den Cursor ein- oder ausschalten oder Vorder- und Hintergrundfarben für Ausgaben festlegen kann. Die meisten dieser Funktionen werden intern von S-Lang benutzt und sind für einen Programmierer nicht von Interesse.

Weit interessanter für einen Programmierer sind die SLsmg-Funktionen. Hierzu gehören z.B. Funktionen zur Ausgabe von Zeichen, zum Zeichnen von Linien oder zum Positionieren des Cursors. Diese Funktionen arbeiten nicht direkt mit dem Terminal, sondern in einem internen Pufferspeicher. Um das Terminal mit dem Inhalt dieses Pufferspeichers zu aktualisieren, muß der Programmierer die eigens dafür angebotene Funktion **SLsmg_refresh** aufrufen.

Bevor die nachfolgend vorgestellten Ausgabefunktionen benutzt werden können, muß S-Lang das aktuelle Terminal (in Environment-Variable *TERM* angegeben) in der Terminaldatenbank nachschlagen. Dazu steht der folgende Aufruf zur Verfügung:

```
void SLtt_get_terminfo(void);
```

Eine der wichtigsten Aufgaben dieser Funktion ist es, Informationen über die Bildschirmgröße des aktuellen Terminals aus der Terminal-Datenbank zu erfragen, damit sie diese Größe entsprechend einstellen kann. Die Anzahl der Zeilen und Spalten des Terminals wird in den globalen Variablen *SLtt_Screen_Rows* und *SLtt_Screen_Cols* hinterlegt. Die in der Terminal-Datenbank hinterlegten Größen berücksichtigen jedoch nicht größenveränderliche Terminals (wie z.B. **xterm** unter X-Windows). Bei dieser Art von Terminals sollte man mit **ioclt** und dem Flag *TIOCGWINSZ* (siehe Kapitel 20.8) die aktuelle Terminal-Größe ermitteln und die entsprechenden Werte den beiden globalen Variablen *SLtt_Screen_Rows* und *SLtt_Screen_Cols* zuweisen.

Beispiel

Ermitteln der aktuellen Größe bei einem größenveränderlichen Terminal

Das folgende Programm 20.19 (`sl_xterm.c`) demonstriert, wie man die aktuelle Größe eines größenveränderlichen Terminals ermitteln kann.

```
#include <slang.h>
#include <sys/ioctl.h>
#include <termios.h>
#include "eighdr.h"

int
main(void)
{
    struct winsize terminal;
    char        zeich;
    int          old_row=0, old_col=0, i=0;

    SLtt_Screen_Rows = SLtt_Screen_Cols = -1;

    while (1) {

        if (old_row != SLtt_Screen_Rows || old_col != SLtt_Screen_Cols) {
            SLang_init_tty(-1, 0, 1); /* Initialisieren von S-Lang          */
            SLtt_get_terminfo();      /* Initialisieren der Bildschirmverwaltung */
            SLsmg_init_smg();         /* Screen Manager einschalten          */
            old_row = SLtt_Screen_Rows;
            old_col = SLtt_Screen_Cols;
            i++;
        }

        if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &terminal) < 0)
            fehler_meld(FATAL_SYS, "kann Bildschirmgroesse nicht ermitteln");

        SLtt_Screen_Rows = terminal.ws_row;
        SLtt_Screen_Cols = terminal.ws_col;

        SLsmg_gotorc(0, 0); /* An Anfang der ersten Zeile          */
        SLsmg_erase_eol(); /* Inhalt der ersten Zeile loeschen */
        SLsmg_printf("Zeilen=%d, Spalten=%d", SLtt_Screen_Rows, SLtt_Screen_Cols);
        SLsmg_refresh();

        if (old_row != SLtt_Screen_Rows || old_col != SLtt_Screen_Cols) {
            SLsmg_gotorc(SLtt_Screen_Rows-1, 0); /* An Anfang der letzten Zeile */
            SLsmg_refresh();
            SLsmg_reset_smg(); /* Screen Manager wieder ausschalten */
            SLang_reset_tty(); /* In Textmodus zurueckschalten      */
            if (i > 2)
                break;
        }
    }
    exit(0);
}
```

Programm 20.19 (`sl_xterm.c`): Ermitteln der Zeilen- und Spaltenzahl eines variabel großen Terminalfensters

Nachdem man dieses Programm 20.19 (`sl_xterm.c`) kompiliert und gelinkt hat

```
cc -o sl_xterm sl_xterm.c fehler.c -lslang
```

kann man es in einem X-Terminal unter X-Windows starten. Es zeigt dann in der oberen linken Ecke des Terminals die maximale Zeilen- und Spaltenzahl dieses Terminals an. Ändert man die Größe, so wird immer links oben die neue maximale Zeilen- und Spaltenzahl angezeigt. Nach dem dritten Verändern der Terminalgröße beendet sich dieses Programm.

Programm 20.19 (`sl_xterm.c`) zeigt unter anderem auch folgendes: Um die `SLsmg`-Funktionen benutzen zu können, muß man diese Art der Bildschirmverwaltung zuerst initialisieren, was mit der folgenden Funktion möglich ist:

```
int SLsmg_init_smg(void);
```

Zudem zeigt dieses Programm, daß Programme, die `SLsmg`-Funktionen verwenden, vor ihrer Beendigung diesen `SLsmg`-Modus mit der folgenden Funktion wieder ausschalten müssen:

```
void SLsmg_reset_smg(void);
```

Der Aufruf dieser Funktion bewirkt, daß der von S-Lang intern verwendete Speicher freigegeben und das Terminal in seinen Ursprungszustand zurückgeschaltet wird. Vorher sollte man jedoch den Cursor an den Anfang der untersten Bildschirmzeile positionieren.

20.10.2 Bildschirm löschen, Cursor positionieren und Text ausgeben

Der Bildschirm entspricht bei der Bildschirmsteuerung in Linux/Unix einem x-y-Koordinatensystem, dessen Nullpunkt die linke obere Ecke ist ($y=0, x=0$). Der Cursor kann dabei nun unter Angabe eines (x,y)-Werts positioniert werden.

```
+-----> x
|
|
|
|
|
v
y
```

Folgende Funktionen ermöglichen das Löschen des gesamten Bildschirms, Positionieren des Cursors und Ausgeben von Text an bestimmten Bildschirmpositionen:

```
void SLtt_clr(void);
void SLsmg_clr(void);
```

Beide Funktionen löschen den ganzen Bildschirm.

void **SLsmg_gotorc**(int y, int x);

positioniert den Cursor in der y. Zeile auf die x. Spalte. Die linke obere Ecke des Bildschirms ist (0, 0) und die rechte untere Ecke des Bildschirms (SLtt_Screen_Rows-1, SLtt_Screen_Cols-1).

void **SLsmg_write_char**(char zeich);

gibt das Zeichen zeich an der aktuellen Cursorposition mit den momentan gesetzten Attributen (siehe weiter unten) aus und bewegt den Cursor weiter.

void **SLsmg_write_string**(char *string);

gibt die Zeichenkette string an der aktuellen Cursorposition mit den momentan gesetzten Attributen (siehe weiter unten) aus und bewegt den Cursor weiter.

void **SLsmg_write_nchars**(char *string, int anzahl);

gibt von der Zeichenkette string genau anzahl Zeichen an der aktuellen Cursorposition mit den momentan gesetzten Attributen (siehe weiter unten) aus und bewegt den Cursor weiter. Hierbei wird ein eventuelles \0-Byte nicht als Ende des Strings betrachtet, sondern mit ausgegeben.

void **SLsmg_write_nstring**(char *string, int anzahl);

gibt von der Zeichenkette string höchstens anzahl Zeichen an der aktuellen Cursorposition mit den momentan gesetzten Attributen (siehe weiter unten) aus und bewegt den Cursor weiter. Ist hier der String kürzer als anzahl Zeichen (wegen \0-Byte), werden für die fehlenden Zeichen Leerzeichen ausgegeben.

void **SLsmg_printf**(char *format, ...);

void **SLsmg_vprintf**(char *format, va_list args);

entsprechen den gleichnamigen Funktionen **printf** und **vprintf** aus der C-Standardbibliothek. Der entsprechend format aufbereitete String wird an der aktuellen Cursorposition mit den momentan gesetzten Attributen (siehe weiter unten) ausgegeben und der Cursor wird entsprechend weiterbewegt.

void **SLsmg_write_wrapped_string**(char *string, int y, int x,
int hoehe, int breite, int fuell);

Anders als die vorherigen Ausgabefunktionen schreibt diese Funktion nicht in den Bildschirmpuffer, was bei diesen Funktionen eine Aktualisierung des Bildschirminhalts mit **SLsmg_refresh** erfordert. Diese Funktion gibt den String in einem Rechteck umbrochen aus. Die linke obere Ecke dieses Rechtecks wird mit den Parametern y und x festgelegt, und die Höhe und Breite dieses Rechtecks wird mit den Parametern hoehe und breite spezifiziert. Ein \n im String erzwingt dabei den Anfang einer neuen Zeile. Wird für den Parameter fuell ein Wert verschieden von 0 angegeben, wird jede Zeile auf die volle Breite des Rechtecks mit Leerzeichen aufgefüllt.

void **SLsmg_refresh**(void);

muß aufgerufen werden, damit der physikalische Bildschirm mit dem Inhalt des Bildschirmpuffers aktualisiert wird, wenn dieser z.B. mit **SLsmg_printf** oder **SLsmg_write_char** geändert wurde.

Beispiel*Demonstrationsprogramm zur Cursorpositionierung in S-Lang*

Das nachfolgende Programm 20.20 (slcurpos.c) demonstriert die Wirkung einiger der eben vorgestellten Funktionen.

```
#include <slang.h>
#include <sys/ioctl.h>
#include <termios.h>
#include "eighdr.h"

int
main(void)
{
    /*----- S-Lang-Modus einschalten -----*/
    Slang_init_tty(-1, 0, 1); /* Initialisieren von S-Lang */
    SLtt_get_terminfo();      /* Initialisieren der Bildschirmverwaltung */
    SLsmg_init_smg();         /* Screen Manager einschalten */

    /*----- In der 1. Spalte der 1. Zeile Text ausgeben -----*/
    SLsmg_gotorc(0, 0);
    SLsmg_write_string("_<---- Position (0,0)"); SLsmg_refresh();

    /*----- In der 21. Spalte der 6. Zeile Text ausgeben -----*/
    SLsmg_gotorc(5, 20);
    SLsmg_write_string("_<---- Position (5,20)"); SLsmg_refresh();

    /*----- In der 50. Spalte der 20. Zeile Cursor-Position ausgeben ----*/
    SLsmg_write_wrapped_string("_<---- Position (19,49)", 19, 49, 3, 7, 1);
    SLsmg_refresh();

    /*----- In der 58. Spalte der 25. Zeile Text ausgeben -----*/
    SLsmg_gotorc(24, 57);
    SLsmg_printf("Position (%d,%d) ---->",
                SLtt_Screen_Rows-1, SLtt_Screen_Cols-1);
    SLsmg_refresh();

    /*----- An der letzten Bildschirmstelle Position ausgeben -----*/
    SLsmg_gotorc(SLtt_Screen_Rows-1, SLtt_Screen_Cols-1);
    SLsmg_write_char('_'); SLsmg_refresh();

    /*----- Abschluss-Arbeiten (Terminal in ursprgl. Zustand) -----*/
    SLang_getkey(); /* Beliebige Taste einlesen */
    SLsmg_cls();    /* Bildschirm loeschen; auch moegl: SLtt_cls(); */
    SLsmg_gotorc(SLtt_Screen_Rows-1, 0); /* An Anfang der letzten Zeile */
    SLsmg_refresh();
    SLsmg_reset_smg(); /* Screen Manager wieder ausschalten */
    SLang_reset_tty(); /* In Textmodus zurueckschalten */

    exit(0);
}
```

Programm 20.20 (slcurpos.c): Demonstrationsbeispiel zur Cursorpositionierung in S-Lang

Hat man das Programm 20.20 (slcurpos.c) kompiliert und gelinkt

```
cc -o slcurpos slcurpos.c -lslang
```

und man startet es, so liefert es folgende Bildschirmausgabe:

```
_<---- Position (0,0)

                                _<---- Position (5,20)

                                _<---- Position (19,49)

                                Position (24,79) ---->
```

Programmende mit Löschen des Bildschirms nach einem Tastendruck.

20.10.3 Vorder- und Hintergrundfarben für Textausgaben

Unter S-Lang steht eine Farbpalette mit mindestens 256 Einträgen zur Verfügung. Jeder Eintrag in dieser Palette definiert eine Vorder- und Hintergrundfarbe. Um einen neuen Paletteneintrag zu erstellen, stehen die beiden folgenden Funktionen zur Verfügung:

```
void SLtt_set_color(int index, char *name, char *vordergrund,
                   char *hintergrund);
```

oder

```
void SLtt_set_color_fgbg(int index, unsigned long vordergrund,
                        unsigned long hintergrund);
```

In beiden Funktionen gibt der erste Parameter den Index des neu zu definierenden Paletteneintrags an. Da der zweite Parameter `name` (bei **`SLtt_set_color`**) zur Zeit nicht benutzt wird, sollte hierfür `NULL` angegeben werden. Die letzten beiden Parameter legen bei beiden Funktionen die Vordergrund- und Hintergrundfarbe für diesen Paletteneintrag fest.

Bei der Funktion **`SLtt_set_color`** kann einer der in Tabelle 20.5 gezeigten Namen für die Parameter `vordergrund` (`fg`) bzw. `hintergrund` (`bg`) angegeben werden.

Vorder- oder Hintergrund (fg oder bg)	Vordergrund (nur für Parameter fg)
black	gray
red	brightred
green	brightgreen
brown	yellow
blue	brightblue
magenta	brightmagenta
lightgray	white

Tabelle 20.5: Mögliche Farbnamen für `vordergrund` bzw. `hintergrund` bei Funktion **`SLtt_set_color`**

Die Farben der linken Spalte in der Tabelle 20.5 können sowohl als Vordergrund- als auch als Hintergrundfarbe angegeben werden. Dagegen können die Farben der rechten Spalte in Tabelle 20.5 bei der Funktion **`SLtt_set_color`** nur als Vordergrundfarbe angegeben werden. Gibt man diese Farben aus der rechten Spalte trotzdem als Hintergrundfarbe an, so ist nicht festgelegt, in welcher Farbe die Ausgabe erscheint, wobei hieraus jedoch sehr oft eine blinkende Ausgabe resultiert.

Bei der Funktion **`SLtt_set_color_fgbg`** kann eine der folgenden in `<slang.h>` definierten Konstanten als Vordergrundfarbe angegeben werden. Als Hintergrundfarbe sind wieder nur die ersten acht Konstanten erlaubt, wenn man ein portables Programm garantieren möchte. Jedoch gilt auch hier, daß bei Angabe einer der hinteren acht Konstanten für die Hintergrundfarbe meist eine blinkende Ausgabe erscheint.

```
# define SLSMG_COLOR_BLACK      0x000000
# define SLSMG_COLOR_RED       0x000001
# define SLSMG_COLOR_GREEN     0x000002
# define SLSMG_COLOR_BROWN    0x000003
# define SLSMG_COLOR_BLUE     0x000004
# define SLSMG_COLOR_MAGENTA  0x000005
# define SLSMG_COLOR_CYAN     0x000006
# define SLSMG_COLOR_LGRAY    0x000007
# define SLSMG_COLOR_GRAY     0x000008
# define SLSMG_COLOR_BRIGHT_RED 0x000009
# define SLSMG_COLOR_BRIGHT_GREEN 0x00000A
# define SLSMG_COLOR_BRIGHT_BROWN 0x00000B
# define SLSMG_COLOR_BRIGHT_BLUE 0x00000C
# define SLSMG_COLOR_BRIGHT_CYAN 0x00000D
# define SLSMG_COLOR_BRIGHT_MAGENTA 0x00000E
# define SLSMG_COLOR_BRIGHT_WHITE 0x00000F
```

Da diese Konstanten lediglich die Werte 0 bis 15 repräsentieren, wird oft auch direkt mit den Zahlenwerten (bei zufälliger Farbauswahl) gearbeitet.

Die Aktivierung eines zuvor mit einer der Funktionen **SLtt_set_color** oder **SLtt_set_color_fgbg** erstellten Paletteneintrags erfolgt mit der folgenden Funktion:

```
void SLsmg_set_color(int index);
```

Nachfolgende Bildschirmausgaben erfolgen dann mit der Vordergrund- und Hintergrundfarbe, die für diesen Paletteneintrag mit dem Index `index` eingestellt wurden.

Beispiel

Demonstrationsprogramm zur farbigen Textausgabe

Das folgende Programm 20.21 (`sl_farbe.c`) gibt 20 Zeilen mit zufällig gewählter Vorder- und Hintergrundfarbe am Bildschirm aus, wobei es jedoch zuerst die Hintergrundfarbe des ganzen Bildschirms auf blau und die Vordergrundfarbe auf weiß einstellt. Bei der Ausgabe der nachfolgenden Zeilen, deren Vorder- und Hintergrundfarbe zufällig ist, wird jede zweite Zeile blinkend dargestellt.

```
#include <slang.h>
#include <stdlib.h>
#include <time.h>

int
main(void)
{
    int i, fg, bg;

    srand(time(NULL)+getpid()); /* Zufallszahlengenerator initialisieren */

    SLang_init_tty(-1, 0, 1); /* Initialisieren von S-Lang */
    SLtt_get_terminfo(); /* Initialisieren der Bildschirmverwaltung */
    SLsmg_init_smg(); /* Screen Manager einschalten */

    /* Bildschirmhintergrund auf blau; Vordergrundfarbe auf weiss */
    SLtt_set_color(1, NULL, "white", "blue");
    SLsmg_set_color(1);
    SLsmg_gotorc(0, 0);
    SLsmg_erase_eos(); /* Ganzen Bildschirm loeschen (mit Leerzeichen
                        ueberschreiben), so daß er blau wird */

    SLsmg_gotorc(1, 5);
    SLsmg_printf("Unterschiedliche Hinter- und "
                "Vordergrundfarben (jedes 2. blinkend)");
    SLsmg_refresh();

    for (i=3; i<=22; i++) {
        SLsmg_gotorc(i, 15);
        fg=rand()%16;
        if (i%2 == 0)
            do {} while ( (bg=rand()%8+8) == fg );
```

```

    else
        do {} while ( (bg=rand()%8) == fg || bg == 4); /* 4 == blau */
        SLtt_set_color_fgbg(i, fg, bg);
        SLsmg_set_color(i);
        SLsmg_printf("Vordergrund=%2d, Hintergrund=%2d", fg, bg);
        SLtt_set_color_fgbg(i+30, 15, 4);
        SLsmg_set_color(i+30);
        SLsmg_printf("    fg=%2d, bg=%2d", fg, bg);
        SLsmg_refresh();
    }

    /*----- Abschluss-Arbeiten (Terminal in ursprgl. Zustand) -----*/
    SLang_getkey(); /* Beliebige Taste einlesen */
    SLsmg_cls(); /* Bildschirm loeschen */
    SLsmg_gotorc(SLtt_Screen_Rows-1, 0); /* An Anfang der letzten Zeile */
    SLsmg_refresh();
    SLsmg_reset_smg(); /* Screen Manager wieder ausschalten */
    SLang_reset_tty(); /* In Textmodus zurueckschalten */

    exit(0);
}

```

Programm 20.21 (sl_farbe.c): Ausgeben von Zeilen mit zufällig gewählter Vorder- und Hintergrundfarbe

Ob die Farben wirklich auf einem Bildschirm dargestellt werden, hängt von vielen Faktoren ab. So gibt z.B. die globale Variable `SLtt_Use_Ansi_Colors`, die mit dem Aufruf von **`SLtt_get_terminfo`** gesetzt wird, an, ob Farben am jeweiligen Bildschirm verfügbar sind. Hat sie einen Wert von 1, so sind Farben verfügbar, während ein Wert 0 bedeutet, daß keine Farben verfügbar sind. Leider sind jedoch die *termcap*- und *terminfo*-Datenbanken nicht immer vollständig. Sollte für ein Farbterminal die Variable `SLtt_Use_Ansi_Colors` nicht richtig gesetzt werden, kann man dessen Farben trotzdem aktivieren, indem man die Environment-Variable `COLORTERM` auf 1 setzt.

20.10.4 Umschalten auf anderen Zeichensatz

Die meisten heutigen Terminals bieten mindestens zwei Zeichensätze an. Üblicherweise ist der erste Zeichensatz ISO-8859-1 (für die Ausgabe von Texten) und der zweite Zeichensatz für das Zeichnen von Linien. Zwischen diesen beiden Zeichensätzen kann man in S-Lang mit der folgenden Funktion hin- und herschalten:

```
void SLsmg_set_char_set(int zweiter_zeichensatz);
```

Wird für den Parameter `zweiter_zeichensatz` ein von 0 verschiedener Wert angegeben, wird der zweite Zeichensatz eingeschaltet, und bei der Angabe von 0 wird wieder der erste Zeichensatz eingeschaltet.

Für die häufig benutzten Linienzeichen des zweiten Zeichensatzes bietet S-Lang eine Reihe von symbolischen Konstanten (siehe Tabelle 20.6) an.

Konstante	Linienzeichen
SLSMG_HLINE_CHAR	-
SLSMG_VLINE_CHAR	
SLSMG_ULCORN_CHAR	┌
SLSMG_URCORN_CHAR	┐
SLSMG_LLCORN_CHAR	└
SLSMG_LRCORN_CHAR	┘
SLSMG_RTEE_CHAR	├
SLSMG_LTEE_CHAR	┤
SLSMG_UTEES_CHAR	┴
SLSMG_DTEE_CHAR	┬
SLSMG_PLUS_CHAR	⊕

Tabelle 20.6: Symbolische Konstanten für Linienzeichen im alternativen Zeichensatz

Beispiel

Ausgabe von blinkenden Dominosteinen

Das folgende Programm 20.22 (sldomino.c) gibt alle Dominosteine blinkend am Bildschirm aus.

```
#include <stdio.h>
#include <slang.h>

int
main(void)
{
    int i, j;

    /*---- S-Lang initialisieren -----*/
    SLang_init_tty(-1, 0, 1);
    SLtt_get_terminfo();
    SLsmg_init_smg();

    /*---- Hellgrauer Bildschirmhintergrund -----*/
    SLtt_set_color(1, NULL, "white", "lightgray");
    SLsmg_set_color(1);
    SLsmg_gotorc(0, 0);
    SLsmg_erase_eos();
    SLsmg_refresh();

    /*---- Ueberschrift weiss auf schwarzen Hintergrund ausgeben ---*/
    SLtt_set_color(1, NULL, "white", "black");
    SLsmg_set_color(1);
    SLsmg_gotorc(1, 22);
    SLsmg_printf(" D i e   D o m i n o s t e i n e   ");
```

```

SLsmg_refresh();

/*---- Alternativen Zeichensatz verwenden -----*/
SLsmg_set_char_set(1);

/*---- Dominosteine blau blinkend auf gelbem Hintergrund -----*/
SLtt_set_color(1, NULL, "blue", "yellow");
SLsmg_set_color(1);
for (i=0 ; i<=6 ; i++)
    for (j=i ; j<=6 ; j++) {
        SLsmg_gotorc(3+3*i, 11*j+2);
        SLsmg_printf("%c%c%c%c%c%c%c%c",
            SLSMG_ULCORN_CHAR,
            SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR,
            SLSMG_UTEER_CHAR,
            SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR,
            SLSMG_URCORN_CHAR);

        SLsmg_gotorc(4+3*i, 11*j+2);
        SLsmg_printf("%c %d %c %d %c",
            SLSMG_VLINE_CHAR, i, SLSMG_VLINE_CHAR, j, SLSMG_VLINE_CHAR);

        SLsmg_gotorc(5+3*i, 11*j+2);
        SLsmg_printf("%c%c%c%c%c%c%c%c",
            SLSMG_LLCORN_CHAR,
            SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR,
            SLSMG_DTEE_CHAR,
            SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR, SLSMG_HLINE_CHAR,
            SLSMG_LRCORN_CHAR);
        SLsmg_refresh();
    }

SLsmg_gotorc(25, 1);
SLsmg_printf("Programmende mit beliebiger Taste.....");
SLsmg_refresh();
SLang_getkey();

SLsmg_cls(); SLsmg_refresh();
SLsmg_reset_smg();
SLang_reset_tty();

exit(0);
}

```

Programm 20.22 (sldomino.c): Ausgabe aller Dominosteine, wobei diese blinken

S-Lang bietet zusätzlich Funktionen an, mit denen man horizontale oder vertikale Linien bzw. Rechtecke zeichnen kann:

```
void SLsmg_draw_hline(int laenge);
```

zeichnet eine horizontale Linie der angegebenen laenge.

```
void SLsmg_draw_vline(int laenge);
```

zeichnet eine vertikale Linie der angegebenen laenge.

```
void SLsmg_draw_box(int links, int oben, int hoehe, int breite);
```

zeichnet ein Rechteck, dessen linke obere Ecke mit den beiden ersten Parameter (links und oben) anzugeben ist. Die letzten beiden Parameter legen dann die hoehe und breite dieses Rechtecks fest.

Beispiel

Eingerahmtes Anzeigen des ersten und zweiten Zeichensatzes

Das folgende Programm 20.23 (sl_zsatz.c) zeigt die beiden Zeichensätze (ersten und zweiten) in Form von Tabellen an, die eingerahmt sind.

```
#include <slang.h>
#include <sys/ioctl.h>
#include <termios.h>
#include "eighdr.h"

/* zeigt in einer Tabelle die Zeichen des jeweiligen Zeichensatzes */
static void
zeige_zeichensatz(int spalte, int zweit_zeichsatz, char *ueberschrift)
{
    int i, j, n=0;

    SLsmg_gotorc(1, spalte + 2);
    SLsmg_write_string(ueberschrift);

    SLsmg_gotorc(4, spalte + 4);
    SLsmg_write_string("0 1 2 3 4 5 6 7 8 9 A B C D E F");

    SLsmg_set_char_set(zweit_zeichsatz);

    for (i = 0; i < 16; i++) {
        SLsmg_gotorc(6 + i, 2 + spalte);
        SLsmg_write_char(i < 10 ? i + '0' : (i - 10) + 'A');
        for (j = 0; j < 16; j++) {
            SLsmg_gotorc(6 + i, spalte + 4 + (j * 2));
            SLsmg_write_char(n++);
        }
    }
    SLsmg_refresh();
    SLsmg_set_char_set(0);
}

int
main(void)
{
    struct winsize terminal;
    int i;

    SLang_init_tty(-1, 0, 1); /* Initialisieren von S-Lang */
}
```



```

SLtt_get_terminfo();      /* Initialisieren der Bildschirmverwaltung */
SLsmg_init_smg();         /* Screen Manager einschalten          */

if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &terminal) < 0)
    fehler_meld(FATAL_SYS, "kann Bildschirmgroesse nicht ermitteln");

SLtt_Screen_Rows = terminal.ws_row;
SLtt_Screen_Cols = terminal.ws_col;

/* Ausgeben der beiden Zeichensätze */
zeige_zeichensatz(0, 0, "Erster Zeichensatz");
zeige_zeichensatz(40, 1, "Zweiter Zeichensatz");

/* Linien und Rechtecke zeichnen */
SLsmg_set_char_set(1);
SLsmg_gotorc( 0, 0); SLsmg_draw_hline(SLtt_Screen_Cols);
SLsmg_gotorc( 2, 0); SLsmg_draw_hline(SLtt_Screen_Cols);
SLsmg_gotorc(24, 0); SLsmg_draw_hline(SLtt_Screen_Cols);
SLsmg_gotorc(0, 39); SLsmg_draw_vline(SLtt_Screen_Rows);
SLsmg_draw_box(5, 3, 18, 36);
SLsmg_draw_box(5, 43, 18, 36);
SLsmg_refresh();
SLsmg_set_char_set(0);

SLang_getkey(); /* Programmende bei Tastendruck */

SLsmg_gotorc(SLtt_Screen_Rows-1, 0); /* An Anfang der letzten Zeile */
SLsmg_refresh();
SLsmg_reset_smg(); /* Screen Manager wieder ausschalten */
SLang_reset_tty(); /* In Textmodus zurueckschalten */

exit(0);
}

```

Programm 20.23 (**sl_zsatz.c**): Eingerahmtes Anzeigen des ersten und zweiten Zeichensatzes

20.10.5 Einlesen von der Tastatur

Zum Lesen eines Zeichens steht die folgende Funktion zur Verfügung:

```
unsigned int SLang_getkey(void);
```

Diese Funktion kehrt erst zurück, wenn ein Zeichen zur Verfügung steht. Es ist zu beachten, daß Steuerzeichen, wie z.B. die Funktionstasten F1, F2 usw., sich meist aus mehreren Zeichen zusammensetzen.

Beispiel

Ermitteln der Tastencodes für die einzelnen Tasten

Das folgende Programm 20.24 (**sl_tcode.c**) gibt zu jeder gedrückten Taste die zugehörigen Tastencodes aus. Beim Drücken der Taste 'q' beendet sich dieses Programm.

```
#include <stdio.h>
#include <ctype.h>
#include <slang.h>

int main(void)
{
    char zeich = 0;

    SLang_init_tty(-1, 0, 1);

    while (zeich != 'q') {
        zeich = SLang_getkey();
        printf("....%c = 0x%x\n", isprint(zeich) ? zeich : ' ', zeich);
        if (!SLang_input_pending(0))
            printf("\n");
    }

    SLang_reset_tty();

    exit(0);
}
```

Programm 20.24 (sl_tcode.c): Ausgeben der Tastencodes für die gedrückten Tasten

Hat man das Programm 20.24 (sl_tcode.c) kompiliert und gelinkt

```
cc -o sl_tcode sl_tcode.c -lslang
```

und man startet es, so ergibt sich z.B. der folgende Ablauf:

```
$ sl_tcode
....a = 0x61      [Taste 'a' gedrückt]

.... = 0x1b      [Taste 'F1' gedrückt]
....[ = 0x5b
....[ = 0x5b
....A = 0x41

....X = 0x58      [Taste 'X' gedrückt]

.... = 0x1b      [Taste 'Pfeil rechts' gedrückt]
....[ = 0x5b
....C = 0x43

.... = 0x1b      [Taste 'Bild hoch' gedrückt]
....[ = 0x5b
....5 = 0x35
....~ = 0x7e

....q = 0x71      [Taste 'q' gedrückt]

$
```

Im obigen Programm 20.24 (`sl_tcode.c`) wurde bereits von der Funktion ***SLang_input_pending*** Gebrauch gemacht. Diese Funktion ist in `<slang.h>` wie folgt definiert:

```
int SLang_input_pending(int timeout);
```

Sie wird immer dann verwendet, wenn man prüfen möchte, ob Zeichen in der Eingabe bereitstehen, ohne daß man den Prozeß blockiert. Diese Funktion gibt einen von 0 verschiedenen Wert zurück, wenn Zeichen in der Eingabe vorhanden sind, und sonst den Wert 0. Über den Parameter `timeout` kann man festlegen, wie viele Zehntel Sekunden diese Funktion maximal blockieren soll, wenn keine Zeichen in der Eingabe vorhanden sind. Gibt man für `timeout` den Wert 0 an, so prüft diese Funktion lediglich, ob Zeichen in der Eingabe vorhanden sind und kehrt in jedem Fall mit dem entsprechenden Rückgabewert sofort (ohne zu Blockieren) zurück.

Beispiel

Reaktionstest

Das folgende Programm 20.25 (`sl_reakt.c`) führt einen Reaktionstest für den Benutzer durch. Dazu benutzt es die beiden Funktionen ***SLang_getkey*** und ***SLang_input_pending***. Es gibt dem Benutzer ein Startzeichen, nach dem er so schnell wie möglich eine beliebige Taste drücken muß. Die dazu gebrauchte Zeit mißt das Programm und gibt sie aus. Es fängt auch den Fall ab, daß ein Benutzer schummeln möchte und schon vorher eine Taste gedrückt hat.

```
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <stdlib.h>
#include <slang.h>

static void delay(long mikrosek);

int
main(void)
{
    double    i, zeit, min=1000000000;
    clock_t   start, ende;

    srand(time(NULL)+getpid());

    do {
        SLang_init_tty(-1, 0, 1);
        SLtt_get_terminfo();
        SLsmg_init_smg();
        SLsmg_cls();
        SLsmg_refresh();

        SLsmg_gotorc(0, 20); SLsmg_printf("Reaktionstest");
        SLsmg_gotorc(1, 20); SLsmg_printf("=====");
        SLsmg_gotorc(3, 10); SLsmg_printf("Ich gebe dir gleich ein Zeichen");
```

```

SLsmg_gotorc(5, 10);
SLsmg_printf("Dann musst du so schnell wie moeglich "
             "eine Taste druecken....\n");
SLsmg_refresh();

delay(rand()%100000+1000000);
if (SLang_input_pending(0)) {
    SLsmg_gotorc(10, 10);
    SLsmg_printf("Du hast versucht zu schummeln. Das gilt nicht!");
    while (SLang_input_pending(0))
        SLang_getkey();
    SLsmg_refresh();
} else {
    SLsmg_gotorc(10, 10); SLsmg_printf("Los.....");
    SLsmg_refresh();
    start = clock(); /* Stopp-Uhr beginnt zu ticken */
    while (!SLang_input_pending(0))
        ;
    ende = clock(); /* Stopp-Uhr wird angehalten */
    while (SLang_input_pending(0)) /* Ueberlesen der eingegeb. Zeichen */
        SLang_getkey();

    zeit = (ende-start)/(double)CLOCKS_PER_SEC;
    if (zeit<min)
        min = zeit;
    SLsmg_gotorc(15, 10);
    SLsmg_printf("Du hast %.4lf Sek. gebraucht. "
                "(Bisheriger Rekord: %.4lf Sek.)", zeit, min);
}
SLsmg_gotorc(20, 0);
SLsmg_printf(".....Willst du es noch einmal probieren (j/n) ? ");
SLsmg_refresh();
} while (tolower(SLang_getkey())=='j');

SLsmg_refresh();
SLsmg_reset_smg();
SLang_reset_tty();

exit(0);
}

static void delay(long mikrosek)
{
    struct timeval timeout;

    timeout.tv_sec  = mikrosek / 1000000L;
    timeout.tv_usec = mikrosek % 1000000L;
    select(0, NULL, NULL, NULL, &timeout);
}

```

Programm 20.25 (sl_reakt.c): Ein Reaktionstest mit den Funktionen *SLang_getkey* und *SLang_input_pending*

Dies war nur eine Einführung in S-Lang. Für detailliertere Informationen muß man entweder die mitgelieferte Dokumentation lesen oder aber – falls diese nicht vorhanden ist – in der Headerdatei `<slang.h>` nachschlagen.

20.11 Die Linux-Konsole

Unter Linux ist die Konsole normalerweise ein serielles Terminal, das man mit der Ausgabe spezieller Kontrollzeichen direkt steuern kann. Normalerweise verwendet man dazu die zuvor vorgestellten Bibliotheken *curses* oder *S-Lang*, die dem Programmierer den Zugriff auf das Terminal auf einer höheren Ebene (*High-Level*) erlauben. Solche High-Level-Bibliotheken bieten für die einzelnen Terminal-Zugriffe (wie z.B. die Positionierung des Cursors oder Löschen einer Zeile) eigene Funktionen an, die der Programmierer aufrufen kann, ohne daß er sich um das spezielle Terminal kümmern muß. Diese Funktionen setzen dann die für das jeweilige Terminal erforderlichen Kontrollzeichen ab. Der Programmierer muß also bei den High-Level-Bibliotheken nichts über die Besonderheiten des jeweiligen Terminals wissen. Ein großer Vorteil dieser High-Level-Bibliotheken ist, daß man portable Programme schreiben kann, die nicht nur auf einen Terminal-Typ ausgelegt sind.

Daneben kann ein Programmierer jedoch auch auf der Low-Level-Schnittstelle programmieren, begrenzt dann aber die Verwendung seiner Programme auf diesen Terminal-Typ. Trotzdem beschreibt dieses Kapitel die Low-Level-Schnittstelle der Linux-Konsole, die auf dem am weitest verbreiteten Typ von seriellen Terminals, der DEC VT100 Familie, basiert.

Die Gründe für die Beschreibung der Low-Level-Eigenschaften der Linux-Konsole sind die folgenden:

1. Die Linux-Konsole ermöglicht es Linux-spezifische Eigenschaften und Fähigkeiten zu nutzen, die in keiner High-Level-Bibliothek zur Verfügung stehen.
2. Verwendet man eine Programmiersprache, die keine einfache Benutzung der High-Level-Bibliotheken ermöglicht, die hauptsächlich für C ausgelegt sind, empfiehlt sich eine Low-Level-Programmierung des Terminals.
3. Dem Leser soll ein Einblick in die Funktionsweise der Linux-Konsole gegeben werden, da eine solche Kenntnis auch dem besseren Verständnis von anderen Terminal-orientierten Programmen und der High-Level-Bibliotheken, wie *curses* und *S-Lang*, dient.

Programme, die direkt mit der Linux-Konsole arbeiten, sollten zu Beginn prüfen, ob sie gerade wirklich auf einer Linux-Konsole ablaufen. Dazu empfiehlt sich der folgende Codeausschnitt, der prüft, ob die Environment-Variable `TERM` den Namen `linux` enthält:

```
if ( !strcmp(getenv("TERM"), "linux") ) {  
    fprintf(stderr, "Programm nur auf Linux-Konsole ablauffähig\n");  
    exit(1);  
}
```

20.11.1 Erster Überblick über die Fähigkeiten einer Linux-Konsole

Die Linux-Konsole unterscheidet grundsätzlich drei Arten von Zeichen:

- ▶ *Normale Zeichen*, die unverändert ausgegeben werden.
- ▶ *Kontrollzeichen*, die eine bestimmte Aktion am Terminal auslösen, wie z.B. einen Klingelton oder einen Zeilenvorschub.
- ▶ *Escape-Sequenzen*, die in einen anderen Modus umschalten. Die Linux-Konsole bleibt dann in diesem Modus, bis dieser wieder ausgeschaltet wird oder mit einer erneuten Escape-Sequenz in einen anderen Modus umgeschaltet wird.

Beispiel

Einfaches Demonstrationsprogramm zur Programmierung der Linux-Konsole

Das folgende Programm 20.26 (`lk_erst.c`) soll nicht nur die verschiedenen Arten von Zeichen verdeutlichen, sondern bereits einen ersten Einblick in die Programmierung der Linux-Konsole geben.

```

1  #include <stdio.h>
2
3  int
4  main(void)
5  {
6      int i;
7
8      printf("\033[H"); /* Escape-Sequenz: Cursor in linke obere Ecke */
9      printf("\033[J"); /* Escape-Sequenz: ganzen Bildschirm loeschen */
10     fflush(stdout);
11     getchar();
12
13     /* Ungefähr in Bildschirmmitte den Text "Die Bildschirmmitte"
14        ausgeben, wobei "Bildschirmmitte" fett gedruckt wird.
15        Am Anfang der naechsten Zeile wird dann der Text
16        "Zeilenanfang" ausgegeben. */
17     printf("\033[12;35HDie \033[1mBildschirmmitte\033[0m\nZeilenanfang");
18     fflush(stdout);
19     getchar();
20
21     /* In der Mitte der Zeilen 13 bis Zeilen 19 den Text "Zeile i"
22        (i steht fuer die Zeilennummer) farbig ausgeben:
23        "Zeile 13" (rot=31)
24        "Zeile 14" (gruen=32)
25        "Zeile 15" (braun=33)
26        "Zeile 16" (blau=34)
27        "Zeile 17" (violett=35)
28        "Zeile 18" (tuerkis=36)
29        "Zeile 19" (hellgrau=37) */
30     for (i=1; i<=7; i++)
31         printf("\033[%d;40H\033[%dmZeile %d", 12+i, 30+i, 12+i);
32     fflush(stdout);

```

```
33     getchar();
34
35     printf("\033[25;1H"); /* An Anfang der letzten Zeile positionieren */
36     fflush(stdout);
37     exit(0);
38 }
```

Programm 20.26 (**lk_erst.c**): Erstes Demonstrationsprogramm zur Linux-Konsole

In den Zeilen 8 und 9 werden Escape-Sequenzen auf der Linux-Konsole ausgegeben: `\033[H` positioniert den Cursor in der linken oberen Ecke des Bildschirms (1,1) und `\033[J` löscht den Bildschirm von der Cursorposition bis zum Ende des Bildschirms, was in diesem Fall das Löschen des ganzen Bildschirms bewirkt.

Grundsätzlich gilt, daß eine Escape-Sequenz mit `\033` (oktaler ASCII-Code für *ESC*, was dezimal dem Wert 27 entspricht) beginnt.

Das nachfolgende Zeichen `[` leitet dann den *Modus Control Sequence Introducer* (CSI) ein. Im CSI-Modus können dann Dezimalzahlen als Parameter, die durch Semikolon zu trennen sind, angegeben werden. Fehlen diese Parameter, so wird dafür der Wert 0 oder 1 angenommen, je nachdem, was für den jeweiligen CSI-Modus sinnvoll ist. Bei der Sequenz `\033[H` bewirkt das Fehlen der Parameter, daß der Cursor auf (1,1) positioniert wird, als ob man die Escape-Sequenz `\033[1;1H` angegeben hätte.

Das letzte Zeichen einer Escape-Sequenz legt dann die auszuführende Aktion fest: `H` bedeutet Positionieren des Cursors und `J` bedeutet Löschen eines Teils des Bildschirms.

In der Zeile 17 wird zunächst die Escape-Sequenz `\033[12;35H` ausgegeben, wodurch der Cursor in die 12. Zeile und 35. Spalte positioniert wird. An dieser Position wird dann zunächst der Text »Die« ausgegeben, bevor dann mit der Escape-Sequenz `\033[1m` in Fettschrift umgeschaltet wird, so daß der nachfolgende Text »Bildschirmmitte« in Fettschrift ausgegeben wird. Mit der Escape-Sequenz `\033[0m` wird dann wieder in Normalschrift umgeschaltet. Beim folgenden »`\n`« handelt es sich um ein Kontrollzeichen, das den Cursor an den Anfang der nächsten Zeile positioniert, wo dann der Text »Zeilenanfang« ausgegeben wird. Sollte sich bei »`\n`« der Cursor schon in der letzten Bildschirmzeile befinden, wird der gesamte Bildschirm um eine Zeile nach oben geschoben (»gescrollt«), so daß die oberste Bildschirmzeile verschwindet und dafür unten eine neue leere Zeile angezeigt wird.

In den Zeilen 30 und 31 wird dann in der 40. Spalte der jeweiligen Zeilen der Text »Zeile 13«, »Zeile 14«, ..., »Zeile 19« farbig ausgegeben. Die Positionierung des Cursors erfolgt dabei mit der Escape-Sequenz `\033[%d;40H` und das Setzen der Farbe geschieht mit der Escape-Sequenz `\033[%dm`, wobei für `%d` ein Wert von 31 bis 37 eingesetzt wird. Die Vordergrundfarben haben hierbei die Nummern 30 (schwarz), 31 (rot), 32 (grün), 33 (braun), 34 (blau), 35 (violett), 36 (türkis) und 37 (hellgrau).

Vor dem Verlassen des Programms wird in der Zeile 35 mit der Escape-Sequenz `\033[25;1H` noch der Cursor an den Anfang der letzten Zeile positioniert.

20.11.2 Kontrollzeichen

Die einem Kontrollzeichen zugrundeliegende Operation wird sofort ausgeführt und anschließend wird im gerade aktiven Modus fortgefahren. Sowohl in den *terminfo*- und *termcap*-Dateien als auch in der entsprechenden Dokumentation werden Kontrollzeichen mit `^X` angegeben. Auch die vorliegende Beschreibung hält sich an dieser Konvention.

Um den numerischen Wert eines Kontrollzeichens zu ermitteln, steht auf vielen Systemen das Makro `CTRL` zur Verfügung, das in `<termios.h>` definiert ist. Da dieses Makro aber nicht auf alle Systemen angeboten wird, empfiehlt sich bei der Verwendung dieses Makros der folgende Codeausschnitt:

```
#ifndef CTRL
# define CTRL(z) ((z) & 0x1F)
#endif
```

Tabelle 20.7 zeigt die Kontrollzeichen der Linux-Konsole.

Kontroll- zeichen	ASCII- Name	Numerische r Wert	Angabe bei printf (oktaler Code)	Auswirkung auf Linux-Konsole
^G	BEL	7	\007	Klingelton
^H	BS	8	\010	bewegt Cursor eine Position nach links, ohne das Zeichen dort zu löschen; hat keine Auswirkung, wenn der Cursor sich zuvor in der 1. Spalte befand.
^I	HT	9	\011	(Horizontal Tab) bewegt Cursor an die nächste Tabulatorposition.
^J	LF	10	\012	(Line Feed) bewegt Cursor in die nächste Zeile an gleiche Spaltenposition; schiebt Bildschirminhalt nach oben, wenn der Cursor sich zuvor in der letzten Zeile befand.
^K	VT	11	\013	(Vertical Tab) entspricht einem LF (Line Feed).
^L	FF	12	\014	(Form Feed) entspricht einem LF (Line Feed).
^M	CR	13	\015	(Carriage Return) bewegt Cursor an den Anfang der aktuellen Zeile.
^N	SO	14	\016	(Shift Out) schaltet den G1-Zeichensatz ein; Kontrollzeichen verlieren hierbei ihre Sonderbedeutung.

Tabelle 20.7: Kontrollzeichen der Linux-Konsole

Kontroll- zeichen	ASCII- Name	Numerische r Wert	Angabe bei printf (oktaler Code)	Auswirkung auf Linux-Konsole
^O	SI	15	\017	(Shift In) schaltet den normalen G0- Zeichensatz ein; Kontrollzeichen besitzen hierbei ihre Sonderbedeu- tung.
^X	CAN	24	\030	beendet die aktuelle Escape-Sequenz.
^Z	SUB	26	\032	beendet die aktuelle Escape-Sequenz.
^[	ESC	27	\033	startet eine Escape-Sequenz.
^?	DEL	127	\177	wird ignoriert.
ALT-^[		155	\233	leitet wie ESC [eine CSI (Komman- dosequenz) ein; kann als Abkürzung verwendet werden, um eine CSI- Sequenz einzuleiten. Dieses Kontroll- zeichen setzt aber eine saubere 8-Bit- Kommunikation voraus. Da dies aber nicht immer garantiert ist, ist von sei- ner Verwendung abzuraten.

Tabelle 20.7: Kontrollzeichen der Linux-Konsole

Die Auswirkung der in Tabelle 20.7 angegebenen Kontrollzeichen hängt auch von den aktuellen Terminaleinstellungen ab. So impliziert z.B. oft ein LF (^J) auch ein CR (^M). Ein weiteres Beispiel ist DEL (^?), das so konfiguriert werden kann, daß es statt sich selbst ein BS (^H) bewirkt.

Mehr Informationen zu ASCII-Zeichen lassen sich mit **man ascii** erfragen. Den neueren 8-Bit-Zeichensatz ISO Latin 1 (*ISO 8859 Latin Alpahbet number 1*), der den ASCII-Standard ablösen wird, kann man mit **man iso_8859_1** nachschlagen.

20.11.3 Escape-Sequenzen

Escape-Sequenzen werden durch ^[eingeleitet. In Programmen oder Shellskripts verwendet man hierfür den Oktalcode \033. Es gibt, wie Tabelle 20.8 zeigt, verschiedene Arten von Escape-Sequenzen.

Escape-Sequenz	Bedeutung
^[x	Einfache Escape-Sequenzen; für x ist dabei das entsprechende Zeichen anzu- geben. Diese Art von Escape-Sequenzen führen eine einfache Operation auf der Konsole durch und verlassen anschließend den Escape-Modus.
^[[	startet eine CSI-Sequenz.
^[[	startet eine Kommandosequenz zum Setzen der Palette.

Tabelle 20.8: Verschiedene Arten von Escape-Sequenzen

Escape-Sequenz	Bedeutung
^[%	startet eine Zeichensatzauswahl: ^[%@ wählt den voreingestellten Zeichensatz (ISO 646 / ISO 8859-1) und ^[%G wählt den UTF-8 (8-wide-character-Unicode) aus.
^[(	wählt Font-Mapping für G0-Zeichensatz: ^[(B (ISO 8859-1 Mapping; Voreinstellung), ^[(O (VT 100 Graphik-Mapping), ^[(U (Null-Mapping) oder ^[(K (benutzerdefiniertes Mapping; kann mit Kommando mapscrn geladen werden).
^[)	wählt Font-Mapping für G1-Zeichensatz mit einem der Zeichen B, O, U oder K aus; siehe auch vorher bei ^[(.
^[#8	DEC-spezifische Testsequenz; füllt den Bildschirm mit Es.

Tabelle 20.8: Verschiedene Arten von Escape-Sequenzen

Einfache Escape-Sequenzen

Tabelle 20.9 zeigt die einfachen Escape-Sequenzen.

Escape-Sequenz	Bedeutung
^[M	bewegt den Cursor in der aktuellen Spalte eine Zeile nach oben; befand sich der Cursor in der obersten Zeile, wird der gesamte Bildschirminhalt um eine Zeile nach unten verschoben, so daß die letzte Bildschirmzeile verschwindet.
^[D	bewegt den Cursor in der aktuellen Spalte eine Zeile nach unten (<i>Line Feed</i>); befand sich der Cursor in der untersten Zeile, wird der gesamte Bildschirminhalt um eine Zeile nach oben verschoben, so daß die erste Bildschirmzeile verschwindet.
^[E	führt ein <i>Carriage Return</i> (CR) und <i>Line Feed</i> (LF) aus.
^[H	setzt an der aktuellen Cursorposition (Spalte) einen Tabulatorstop.
^[7	speichert die aktuelle Cursorposition mit den zugehörigen Attributen und dem aktuellen Zeichensatz; Ein erneutes Speichern überschreibt die vorherigen Werte.
^[8	restauriert eine zuvor gespeicherte Cursorposition mit ihren Attributen.
^[>	schaltet den Nummernblock in den numerischen Modus (ist die Voreinstellung).
^[=	schaltet den Nummernblock in den Anwendungsmodus (Cursortasten werden eingeschaltet).
^[c	setzt alle Terminal-Einstellungen in die Zustände zurück, die vor den Veränderungen durch Kontrollzeichen und Escape-Sequenzen vorlagen.
^[Z	gibt die Terminal-Kennung aus; bei Ausgabe von ^[[?6c emuliert die Konsole vollständig den Terminal-Typ DEC VT102.

Tabelle 20.9: Einfache Escape-Sequenzen

Hier ist noch darauf hinzuweisen, daß innerhalb von Escape-Sequenzen auch Kontrollzeichen erlaubt sind. Beispielsweise würde `^[[^GZ` zuerst die Terminal-Glocke erklingen lassen und dann die Terminal-Kennung ausgeben, und `^[[^XD` würde nur ein `D` ausgeben, da das Kontrollzeichen `^X` die Escape-Sequenz beendet (siehe auch Tabelle 20.7).

CSI-Sequenzen

CSI-Sequenzen setzen sich normalerweise aus drei Teilen zusammen:

- 1. `^[[` leitet eine CSI-Sequenz ein.
- 2. Es können bis zu 16 Parameter (Dezimalzahlen) angegeben werden, die durch Semikolons zu trennen sind. Für die einzelnen Parameter werden dabei nachfolgend die Bezeichnungen `par1`, `par2`, ..., `par16` verwendet. Fehlt die Angabe von Parametern, wird dafür automatisch einer der Werte 0 oder 1 angenommen, je nachdem, was im konkreten Fall sinnvoll ist.
- 3. Ein Kommandozeichen, das festlegt, wie die davor angegebenen Parameter auszuwerten sind, beendet eine CSI-Sequenz. Tabelle 20.10 zeigt die möglichen CSI-Kommandozeichen.

Zeichen	Bedeutung
h	aktiviert ANSI-Modi (siehe auch Tabelle 20.12)
l	deaktiviert ANSI-Modi (siehe auch Tabelle 20.12)
?h	aktiviert DEC-spezifische Modi (siehe auch Tabelle 20.13)
?l	deaktiviert DEC-spezifische Modi (siehe auch Tabelle 20.13)
@	fügt an der aktuellen Cursorposition <code>par1</code> Leerzeichen in der aktuellen Zeile ein.
A	bewegt den Cursor um <code>par1</code> Zeilen nach oben.
B	bewegt den Cursor um <code>par1</code> Zeilen nach unten.
C	bewegt den Cursor um <code>par1</code> Spalten nach rechts.
D	bewegt den Cursor um <code>par1</code> Spalten nach links.
E	bewegt den Cursor <code>par1</code> Zeilen nach unten an den Zeilenanfang (Voreinstellung: <code>par1=1</code>).
F	bewegt den Cursor <code>par1</code> Zeilen nach oben an den Zeilenanfang (Voreinstellung: <code>par1=1</code>).
G	bewegt den Cursor in die Spalte <code>par1</code> in der aktuellen Zeile.
H	bewegt den Cursor in die Zeile <code>par1</code> und Spalte <code>par2</code> (Voreinstellung ist: <code>par1=1</code> ; <code>par2=1</code>).
J	<code>par1=0</code> : löscht von Cursorposition bis zum Ende des Bildschirms. <code>par1=1</code> : löscht vom Bildschirmanfang (linke obere Ecke) bis zur Cursorposition. <code>par1=2</code> : löscht den gesamten Bildschirminhalt. (Voreinstellung: <code>par1=0</code>).

Tabelle 20.10: CSI-Kommandozeichen

Zeichen	Bedeutung
K	par1=0: löscht von Cursorposition bis zum Zeilenende. par1=1: löscht vom Anfang der Zeile bis zur Cursorposition. par1=2: löscht die gesamte Zeile. (Voreinstellung: par1=0).
L	fügt über der aktuellen Zeile par1 Leerzeilen ein.
M	löscht par1 Zeilen einschließlich der aktuellen Zeile.
P	löscht an der aktuellen Cursorposition par1 Zeichen und zieht den Rest der Zeile entsprechend nach.
X	löscht von der aktuellen Cursorposition par1 Zeichen in der aktuellen Zeile.
a	bewegt den Cursor um par1 Spalten nach rechts.
c	(entspricht ^[Z], bewirkt, daß das Terminal mit ^[[?6c antwortet.
d	bewegt den Cursor in die Zeile par1 bei Beibehaltung der aktuellen Spalte.
e	bewegt den Cursor um par1 Zeilen nach unten.
f	bewegt den Cursor in die Zeile par1 und Spalte par2 (Voreinstellung ist: par1=1; par2=1).
g	par1=0: löscht Tabulator an aktueller Cursorposition (Voreinstellung). par1=3: löscht alle Tabulatoren.
m	legt die Darstellung (Attribute) von Zeichen für die nachfolgenden Ausgaben fest; siehe auch Tabelle 20.11.
n	par1=5: Statusabfrage: Antwortet Terminal mit Ausgabe von ^[[On, ist dies richtig. par2=6: Abfrage der Cursorposition: Das Terminal antwortet mit der Ausgabe von ^[[x;yR, wobei x die Zeile und y die Spalte der aktuellen Cursorposition am Bildschirm ist.
q	par1=0: schaltet alle LED-Tasten (<i>Scroll-Lock</i> , <i>Num-Lock</i> , <i>Caps-Lock</i>) aus. par1=1: schaltet Taste <i>Scroll-Lock</i> (Rollen) ein. par1=2: schaltet Taste <i>Num-Lock</i> (Num) ein. par1=3: schaltet Taste <i>Caps-Lock</i> (Shift-Taste) ein.
r	legt einen Teilbereich des Bildschirms als Scroll-Bereich fest. par1 gibt dabei die erste und par2 die letzte Zeile an; Voreinstellung: par1=1; par2=letzte Bildschirmzeile.
s?	(entspricht ^[7], speichert die aktuelle Cursorposition mit den zugehörigen Attributen und dem aktuellen Zeichensatz; ein erneutes Speichern überschreibt die vorherigen Werte.
u?	(entspricht ^[8], restauriert eine zuvor gespeicherte Cursorposition mit ihren Attributen.
,	bewegt den Cursor in die Spalte par1 in der aktuellen Zeile.
]	setterm-Sequenzen; siehe auch Tabelle 20.14.

Tabelle 20.10: CSI-Kommandozeichen

Das Kommandozeichen `m` erlaubt das Setzen von Attributen für Zeichenausgaben. Tabelle 20.11 gibt die einzelnen Codes, die als `par1` anzugeben sind, um die entsprechenden Attribute ein- bzw. auszuschalten.

par1	Auswirkung
0	setzt alle Attribute wieder auf ihre Voreinstellung zurück.
1	schaltet starke Intensität (Fettschrift) ein.
2	schaltet schwache Intensität (<i>half-bright</i>) ein; wird an Farbbildschirmen mittels Farbe simuliert, wobei die dabei zu verwendende Farbe mit <code>^[]</code> festgelegt werden kann (siehe Tabelle 20.14).
4	schaltet Unterstreichung ein; wird an Farbbildschirmen mittels Farbe simuliert, wobei die dabei zu verwendende Farbe mit <code>^[]</code> festgelegt werden kann (siehe Tabelle 20.14).
5	schaltet Blinken ein.
7	schaltet inverse Darstellung ein.
10	wählt den Standardzeichensatz (ISO Latin 1) aus, wobei in diesem Zeichensatz Kontrollzeichen nicht angezeigt werden und das 8. Bit vor der Ausgabe gelöscht wird.
11	wählt das Null-Mapping aus, wobei in diesem Zeichensatz Kontrollzeichen graphisch dargestellt werden und das 8. Bit vor der Ausgabe gelöscht wird.
12	wählt das Null-Mapping aus, wobei in diesem Zeichensatz Kontrollzeichen graphisch dargestellt werden und das 8. Bit vor der Ausgabe nicht gelöscht wird.
21	schaltet normale Intensität ein.
22	schaltet normale Intensität ein.
24	schaltet Unterstreichung aus.
25	schaltet Blinken aus.
27	schaltet inverse Darstellung aus.
30-31	schaltet entsprechende Vordergrundfarbe ein (30=Schwarz; 31=Rot; 32=Grün; 33=Braun; 34=Blau; 35=Violett; 36=Türkis; 37=Weiß).
38	schaltet Unterstreichung ein; benutzt die voreingestellte Vordergrundfarbe.
39	Schaltet Unterstreichung aus; benutzt die voreingestellte Vordergrundfarbe.
40	Schaltet entsprechende Hintergrundfarbe ein (40=Schwarz; 41=Rot; 42=Grün; 43=Braun; 44=Blau; 45=Violett; 46=Türkis; 47=Weiß).
49	Setzt die voreingestellte Hintergrundfarbe.

Tabelle 20.11: Codes zum Festlegen von Attributen beim Kommandozeichen `m`

Neben den Attributen existieren noch zwei Arten von Modi:

- ▶ *ANSI-Modi*, die in Tabelle 20.12 gezeigt sind, werden durch das Kommandozeichen `h` ein- und durch das Kommandozeichen `l` ausgeschaltet.
- ▶ *DEC-spezifische Modi*, die in Tabelle 20.13 gezeigt sind, werden durch die Sequenz `?h` ein- und durch die Sequenz `?l` ausgeschaltet.

parl	Auswirkung
3	Kontrollzeichen werden angezeigt; Voreinstellung: Kontrollzeichen nicht anzeigen.
4	Einfügemodus wird eingeschaltet; Voreinstellung: kein Einfügemodus.
20	Nach jedem Zeilenvorschub (LF=Line Feed) wird automatisch ein Carriage-Return (CR) ausgegeben; Voreinstellung: keine automatische Ausgabe von CR bei LF.

Tabelle 20.12: ANSI-Modi (werden durch `h` ein- und durch `l` ausgeschaltet)

parl	Auswirkung, wenn gesetzt; Voreinstellung
1	Beim Drücken von Cursortasten wird das Präfix <code>^[0</code> statt des Präfixes <code>^[[</code> geschickt; ausgeschaltet.
3	schaltet den Bildschirm vom 80-Spaltenmodus in den 132-Spaltenmodus um (noch nicht implementiert); ausgeschaltet.
5	schaltet den ganzen Bildschirm in inverse Darstellung; ausgeschaltet.
6	aktiviert eine eingerichtete Scroll-Region, so daß Cursorpositionierungen relativ zur linken oberen Ecke der Scroll-Region stattfinden; deaktiviert.
7	schaltet den autowrap-Modus ein, in dem nach einer Ausgabe in der letzten Spalte automatisch am Anfang der nächsten Zeile fortgefahren wird. Ist der autowrap-Modus ausgeschaltet, werden Zeichen am rechten Rand der aktuellen Zeilen überschrieben; eingeschaltet.
8	schaltet automatische Tastenwiederholung ein; eingeschaltet.
9	setzt Mouse-Reporting-Modus auf 1. Beim Ausschalten wird dieser Modus auf 0 zurückgesetzt; ausgeschaltet.
25	macht Mauszeiger sichtbar; eingeschaltet.
1000	setzt Mouse-Reporting-Modus auf 2. Beim Ausschalten wird dieser Modus auf 0 zurückgesetzt; ausgeschaltet.

Tabelle 20.13: DEC-spezifische-Modi (werden durch `?h` ein- und durch `?l` ausgeschaltet)

Tabelle 20.14 zeigt die `setterm`-Sequenzen, die bei dem Kommandozeichen `]` erlaubt sind.

par1	Auswirkung
1	par2 gibt die Farbe an, die als Unterstreichungsfarbe (beim Unterstreichungsattribut) zu verwenden ist.
2	par2 gibt die Farbe an, die bei schwacher Intensität (<i>half-bright</i> -Attribut) zu verwenden ist.
8	legt die aktuelle gesetzte Hintergrund- und Vordergrundfarbe als Default-Attribute fest.
9	par2 legt die Minuten fest, nach denen der Bildschirmschoner aktiv werden soll; par2=0 schaltet den Bildschirmschoner aus.
10	par2 legt die Frequenz (in Hertz) der Terminal-Glocke fest; par2=0 setzt wieder die voreingestellte Frequenz.
11	par2 legt die Dauer, die die Terminal-Glocke klingen soll, in Millisekunden fest; par2=0 setzt wieder auf die voreingestellte Dauer zurück.
12	par2 legt die Konsole fest, die in den Vordergrund zu bringen ist; siehe auch nächstes Kapitel, bei dem virtuelle Konsolen behandelt werden.
13	bringt den Bildschirm wieder in den normalen Zustand, wenn der Bildschirmschoner aktiv ist.
14	par2 legt die Minuten fest, nach denen das VESA Power-Down aktiv werden soll; par2=0 schaltet VESA Power-Down aus.

Tabelle 20.14: setterm-Sequenzen beim Kommandozeichen]

Farben auf der Linux-Konsole

Einige Sequenzen haben Parameter, die Farben festlegen. Die Nummern der einzelnen Farben sind in Tabelle 20.15 angegeben.

Nummer	Farbe
0	Schwarz
1	Rot
2	Grün
3	Braun
4	Blau
5	Violett
6	Türkis
7	Hellgrau
8	Dunkelgrau
9	Hellrot
10	Hellgrün

Tabelle 20.15: Farbnummern auf der Linux-Konsole

Nummer	Farbe
11	Gelb
12	Hellblau
13	helles Violett
14	helles Türkis
15	Weiß

Tabelle 20.15: Farbnummern auf der Linux-Konsole

Während für Hintergrundfarben nur die Farbnummern 0 bis 7 möglich sind, können für Vordergrundfarben alle Nummern zwischen 0 und 15 verwendet werden.

In Wirklichkeit beschreiben die Farbnummern keine Farben, sondern sind nur Indizes in einer Tabelle (Farbpalette), in der die zugehörigen Farbe als RGB-Wert definiert ist. Man kann diese RGB-Werte auch ändern. Dazu steht die folgende Escape-Sequenz zur Verfügung:

`^[P nrrggbb`

`n` legt den Index des zu ändernden Paletteneintrags fest.

`rr` legt den Wert der zu verwendenden Rotkomponente fest.

`gg` legt den Wert der zu verwendenden Grünkomponente fest.

`bb` legt den Wert der zu verwendenden Blaukomponente fest.

Für `r`, `g` und `b` kann eine hexadezimale Ziffer angegeben werden. Das Zurücksetzen der Palette auf ihre voreingestellten Werte ist mit der Escape-Sequenz `^[R` möglich.

Beispiel

Zufälliges Ändern des Paletteneintrags 0

Das folgende Programm 20.27 (`lk_palet.c`) verändert den Paletteneintrag mit dem Index 0, der auf die Farbe Schwarz voreingestellt ist. Dazu legt es zunächst den Paletteneintrag 0 als Hintergrundfarbe fest. Danach löscht es den ganzen Bildschirm, so daß dieser schwarz erscheint, da die Voreinstellung für den Paletteneintrag 0 die Farbe Schwarz ist. Nach jedem Drücken der Return-Taste verändert das Programm den Paletteneintrag 0 mit zufälligen Werten, so daß der Bildschirm in einer anderen Farbe dargestellt wird. Bei Eingabe von EOF (`Strg-D`) beendet sich das Programm, wobei es jedoch zuvor noch die voreingestellte Palette wiederherstellt.

```
#include <stdio.h>

int
main(void)
{
    int i, x;
```



```
char palette[1000];

srand(getpid());
printf("\033[40m"); /* Schwarz als Hintergrundfarbe festlegen */
printf("\033[H\033[J"); /* Ganzen Bildschirm loeschen */
fflush(stdout);

while (getchar() != EOF) {
    /* Paletteneintrag für schwarz (0) zufaellig aendern */
    sprintf(palette, "\033[P0%x%x%x",
            rand()%0x100, rand()%0x100, rand()%0x100);
    printf("%s", palette);
    fflush(stdout);
}

printf("\033[R"); /* Voreinstellung fuer Palette wiederherstellen */
fflush(stdout);
exit(0);
}
```

Programm 20.27 (**lk_palette.c**): Zufälliges Ändern des Paletteneintrags 0

20.11.4 Repräsentation von Steuertasten

Neben den normalen Zeichen auf einer Tastatur gibt es auch Steuertasten, wie z.B. die Taste »Cursor hoch« oder die Taste »F2«. Die diesen Tasten zugeordneten Tastaturcodes zeigt die Tabelle 20.16.

Steuertaste	Code
F1	^[[A
F2	^[[B
F3	^[[C
F4	^[[D
F5	^[[E
F6	^[[17~
F7	^[[18~
F8	^[[19~
F9	^[[20~
F10	^[[21~
F11, Shift-F1, Shift-F11	^[[23~
F12, Shift-F2, Shift-F12	^[[24~
Shift-F3	^[[25~
Shift-F4	^[[26~

Tabelle 20.16: Codes der Steuertasten

Steuertaste	Code
Shift-F5	^[28~
Shift-F6	^[29~
Shift-F7	^[31~
Shift-F8	^[32~
Shift-F9	^[33~
Shift-F10	^[34~
Cursor hoch	^[A
Cursor tief	^[B
Cursor rechts	^[C
Cursor links	^[D
Pos1 (Home)	^[1~
Einfg (Ins)	^[2~
Entf (Del)	^[3~
Ende (End)	^[4~
Bild hoch (PgUp)	^[5~
Bild tief (PgDn)	^[6~

Tabelle 20.16: Codes der Steuertasten

Wurde die Escape-Sequenz `^[?1h` (siehe auch Tabelle 20.13) zuvor eingegeben, so wird bei den Cursortasten nicht das Präfix `^[`, sondern das Präfix `^[O` verwendet.

20.11.5 Direkter Bildschirmzugriff

Es ist auch möglich, direkt auf den Bildschirminhalt zuzugreifen. Dazu bietet Linux zwei zeichenorientierte Gerätedateien an:

<code>/dev/vcs*</code>	enthält nur den Text der entsprechenden Linux-Konsole (ohne Attribute).
<code>/dev/vcsa*</code>	enthält neben dem Text der entsprechenden Linux-Konsole auch die zugehörigen Attribute (Farbe usw.).

Für beide Gerätedateien gilt, daß man Superuser sein muß, um auf sie zuzugreifen.

Die virtuelle Konsole `/dev/vcs*`

Durch das Lesen der Gerätedatei `/dev/vcs0` kann man den Inhalt der aktuellen virtuellen Konsole (*virtual console screen*) erfragen. Wird der Inhalt der aktuellen virtuellen Konsole gerade nach unten gerollt, so enthält `/dev/vcs0` immer noch den Inhalt des Bildschirms, der vor dem Rollen sichtbar war. Es ist auch möglich, den Inhalt anderer virtueller Kon-

solen zu lesen. Dazu muß die entsprechende Nummer an `/dev/vcs` angehängt werden, wie z.B. `/dev/vcs2` für die zweite oder `/dev/vcs5` für die fünfte virtuelle Konsole.

Die Gerätedatei `/dev/vcs*` gibt keine Hinweise über die Größe der virtuellen Konsole. Es wird nur EOF für das Bildschirmende geliefert. Tritt z.B. ein EOF auf, nachdem man 2000 Bytes gelesen hat, kann man daraus nicht schließen, ob der Bildschirm 80 Spalten und 25 Zeilen oder aber 40 Spalten und 50 Zeilen hat.

Die virtuelle Konsole `/dev/vcsa*`

Durch das Lesen der Gerätedatei `/dev/vcsa0` kann man nicht nur den Inhalt der aktuellen virtuellen Konsole (*virtual console screen with attributes*) erfragen, sondern auch deren Attribute (Vordergrundfarbe, Hintergrundfarbe, Blinken, Fettschrift, Bildschirmgröße, aktuelle Cursorposition). Es ist auch möglich, den Inhalt und die Attribute anderer virtueller Konsolen zu lesen. Dazu muß die entsprechende Nummer an `/dev/vcsa` angehängt werden, wie z.B. `/dev/vcsa2` für die zweite oder `/dev/vcsa5` für die fünfte virtuelle Konsole.

Die ersten vier Bytes von `/dev/vcsa*` geben die folgenden Informationen:

Byte 0	Anzahl der Zeilen des Bildschirms
Byte 1	Anzahl der Spalten des Bildschirms
Byte 2	Aktuelle Spalte des Cursors
Byte 3	Aktuelle Zeile des Cursors

Der Rest dieser Datei enthält abwechselnd ein Text- und Attributbyte der entsprechenden Konsole.

Um sich diese ersten vier Bytes für eine Konsole `/dev/vcsa*` von der Kommandozeile aus anzeigen zu lassen, empfiehlt sich der nachfolgend gezeigte **od**-Aufruf.

```
$ od -N4 -tdC /dev/vcsa1 [erste virtuelle Konsole]
0000000  25  80  16  24
0000004
$ od -N4 -tdC /dev/vcsa2 [zweite virtuelle Konsole]
0000000  25  80   0  21
0000004
$
```

Zum Setzen der Cursorposition auf eine virtuelle Konsole muß man das dritte und vierte Byte der zugehörigen Datei `/dev/vcsa*` entsprechend ändern. Um die ersten beiden Bytes, die sowieso nicht geändert werden können, zu überspringen, muß man bei der Verwendung eines **echo**-Kommandos hierfür irgendeine Platzhalter, wie z.B. Leerzeichen einsetzen. Um z.B. den Cursor der dritten virtuellen Konsole in der 41. Spalte und der 11. Zeile zu positionieren, könnte das folgende Kommando eingegeben werden:

```
echo -n -e " \050\012" >/dev/vcsa3
```

Die Option **-n** verhindert die Ausgabe eines Neue-Zeile-Zeichens und die Option **-e** schaltet die Interpretation von Escapezeichen ein, so daß sowohl `\050` als Oktalzahl (dezimal 40) als auch `\012` als Oktalzahl (dezimal 10) ausgewertet wird. Es ist zu beachten, daß hier die Spalten- und Zeilenzählung bei 0 (und nicht bei 1) beginnt.

Nach diesen vier Bytes beginnt der Inhalt der entsprechenden virtuellen Konsole, wobei für eine Position immer zwei Bytes vorgesehen sind: Textbyte und zugehöriges Attributbyte. Die einzelnen Bits eines Attributbytes enthalten dabei die in Tabelle 20.17 angegebenen Informationen:

Bit	Attribut
7	Blinken (0=ausgeschaltet; 1=eingeschaltet)
6-4	Hintergrundfarbe (siehe auch Tabelle 20.15)
3	Fettschrift (0=ausgeschaltet; 1=eingeschaltet)
2-0	Vordergrundfarbe (siehe auch Tabelle 20.15)

Tabelle 20.17: Aufbau eines Attributbytes

Weitere Informationen zur Linux-Konsole können mit **man console_codes** nachgeschlagen werden.

20.11.6 Realisierung der Borland-Semigraphik auf einer Linux-Konsole

Als Demonstrationsbeispiel, das viele der in den vorherigen Kapiteln vorgestellten Konstrukte verdeutlichen soll, wird hier eine einfache Realisierung der Borland-Semigraphik gegeben, die sich unter DOS bei C- und PASCAL-Programmierer großer Beliebtheit erfreut.

Zunächst wird hier die Borland-Semigraphik von DOS kurz beschrieben, bevor die Datei `conio.h` vorgestellt wird, die eine einfache Realisierung dieser Semigraphik darstellt. Danach werden noch einige Demonstrationsprogramme gegeben, die diese Emulation aus `conio.h` verwenden, um Semigraphik unter Linux zu programmieren.

Kurze Beschreibung der Borland-Semigraphik (Headerdatei `conio.h`)

In der Borland-Semigraphik entspricht der Bildschirm einem x,y-Koordinatensystem, dessen Nullpunkt die linke obere Ecke ist (x=1,y=1). Der Cursor kann unter Angabe eines (x,y)-Werts (x = Spalte, y = Zeile) positioniert werden:

```
(1,1)
+-----> x (normalerweise von 1 bis 80)
|
|
|
|
| y (normalerweise von 1 bis 25)
v
```

Farben

Tabelle 20.18 zeigt die Farbeinstellungen, die in der Borland-Semigraphik möglich sind.

Name	Deutsche Bezeichnung		
BLACK	Schwarz	Hintergrundfarben	Vordergrundfarben
BLUE	Blau		
GREEN	Grün		
CYAN	Türkis		
RED	Rot		
MAGENTA	Violett		
BROWN	Braun		
LIGHTGRAY	hellgrau		
DARKGRAY	Dunkelgrau		
LIGHTBLUE	Hellblau		
LIGHTGREEN	Hellgrün		
LIGHTCYAN	helles Türkis		
LIGHTRED	Hellrot		
LIGHTMAGENTA	helles Violett		
YELLOW	Gelb		
WHITE	Weiß		

Tabelle 20.18: Farbeinstellungen in der Borland-Semigraphik

Semigraphikfunktionen

- cgets(char *str)**
liest eine Zeichenkette (String) von der Tastatur ein. Vom Benutzer eingegebene Zeichen werden auf dem Bildschirm mit den momentan gesetzten Farbattributen angezeigt.
- clreol()**
Rest einer Zeile ab Cursorposition löschen.

clrscr()

Bildschirm löschen; Cursor wird auf (1,1) positioniert.

cprintf(char *format, argument(e))

Text mit den aktuell gesetzten Farb- und Bildschirmattributen formatiert ausgeben; die format-Angabe entspricht der bei **printf**. Für einen Zeilenvorschub an den Anfang der nächsten Zeile ist \n\r oder \r\n (und nicht nur \n wie bei **printf**) anzugeben.

cputs(char *string)

string mit den aktuellen Farb- und Bildschirmattributen ausgeben.

cscanf(char *format, argument(e))

wie **scanf**, nur erfolgt die Eingabe mit den momentan gesetzten Vorder- und Hintergrundfarben.

delline()

Zeile löschen und unteren Fensterinhalt nachziehen.

getch()

Ungepuffertes Einlesen eines Zeichens (ohne Anzeige).

getche()

Ungepuffertes Einlesen eines Zeichens (mit Anzeige); siehe auch **getch**.

getpass(char *prompt)

gibt den String prompt aus und liest dann verdeckt ein Paßwort (maximal acht Zeichen), das es zurückliefert.

gettext(int left, int top, int right, int bottom, void *puffer)

kopiert einen rechteckigen Bildausschnitt in die Puffervariable puffer. Die Koordinatenangaben sind dabei absolut, und nicht fensterbezogen. Das Bildschirmrechteck wird von **gettext** sequentiell von links nach rechts und von oben nach unten im Speicher abgelegt. Für die Darstellung eines Zeichens werden zwei Bytes benötigt. War **gettext** erfolgreich, so liefert es 1 als Rückgabewert, andernfalls 0.

gotoxy(int x, int y)

Cursor in x-te Spalte, y-te Zeile positionieren.

highvideo()

legt hohe Intensität für folgende Textausgaben fest.

insline()

Zeile einfügen und unteren Fensterinhalt nach unten schieben.

kbhit()

prüft, ob eine Taste gedrückt wurde; wenn ja, liefert **kbhit** 1, sonst 0. Das eingegebene Zeichen kann mit **getch** bzw. **getche** nachträglich erfragt werden.

lowvideo()

legt niedrige Intensität für folgende Textausgaben fest.

movetext(int left, int top, int right, int bottom,
int zielleft, int zieltop)
Bildausschnitt (links oben (left, top), rechts unten (right, bottom)) an andere Stelle (links oben (zielleft und zieltop)) kopieren. Erfolgreiches **movetext** liefert 1 zurück, sonst 0.

normvideo()
legt normale Intensität für folgende Textausgaben fest.

putch(int zeich)
Zeichen zeich mit aktuellen Farb- und Bildschirmattributen ausgeben.

puttext(int left, int top, int right, int bottom, void *puffer)
kopiert einen (zuvor mit **gettext**) in puffer gespeicherten Bildausschnitt auf den Bildschirm. Koordinaten sind dabei absolut, nicht fensterbezogen. Erfolgreiches **gettext** liefert 1, sonst 0.

textattr(int attribut)
legt Vorder- und Hintergrundfarbe für folgende Textausgaben fest und faßt die Funktionen **textcolor** und **textbackground** zusammen. *attribut* hat die folgende Struktur:

+-----+ -----+	vvvv-Bits legen die Vordergrund- (0 bis 15) und
B h h h v v v v	hhh-Bits die Hintergrundfarbe (0 bis 7) fest.
+-----+ -----+	Ist das <i>B</i> -Bit 1, wird noch das Blinken eingeschaltet.

textbackground(int farbe)
legt die Hintergrundfarbe für Texte fest. Für *farbe* ist BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN oder LIGHTGRAY bzw. ein entsprechender Wert anzugeben.

textcolor(int farbe)
legt die Farbe für die folgenden Textausgaben fest. Für *farbe* ist einer der folgenden Namen (BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE) bzw. ihr Wert anzugeben. Die Addition der Konstanten BLINK zu *farbe* bewirkt, daß die folgenden Textausgaben blinken.

wherex(), **wherey**()
liefert die aktuelle Spalten- bzw. Zeilenposition des Cursors.

_setcursortype(int cur_typ)
legt die Cursorform fest (nur in Borland-C). Für *cur_typ* ist _NOCURSOR (unsichtbar), _SOLIDCURSOR (ausgefüllt) oder _NORMALCURSOR (Unterstrich) anzugeben.

Realisierung der Borland-Semigraphik

Die folgende Headerdatei (conio.h) enthält eine einfache Realisierung der Borland-Graphik auf einer Linux-Konsole.

```
#ifndef __CONIO_H
#define __CONIO_H
```

```

/* conio.h - Einfache Nachbildung der wichtigsten
   Semigraphik-Funktionen von Borland-C unter DOS
   (aus conio.h und dos.h) */

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <time.h>
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdlib.h>

/*===== Hilfsroutinen =====*/
/*=====*/

static struct termios alt_terminal;
static int alt_ttyfd = -1;
static enum { RESET, CBREAK } tty_modus = RESET;

/*----- tty_cbreak --- Terminal in cbreak-Modus umschalten -----*/
int tty_cbreak(int fd, int echo)
{
    struct termios terminal;

    if (tcgetattr(fd, &alt_terminal) < 0)
        return(-1);
    terminal = alt_terminal;
    terminal.c_lflag &= (echo==0) ? (~(ECHO | ICANON)) : (~ICANON);
    terminal.c_cc[VMIN] = 1; /* Fall 2: Immer nur 1 Byte; kein Timer */
    terminal.c_cc[VTIME] = 0;

    if (tcsetattr(fd, TCSANOW, &terminal) < 0)
        return(-1);
    tty_modus = CBREAK;
    alt_ttyfd = fd;
    return(0);
}

/*----- tty_reset --- Terminal in alten Modus zuruecksetzen -----*/
int tty_reset(int fd)
{
    if (tty_modus != CBREAK)
        return(-1);
    if (tcsetattr(fd, TCSANOW, &alt_terminal) < 0)
        return(-1);
    return(0);
}

/*----- wherexy --- Aktuelle Cursorkoordinaten ermitteln -----*/
void wherexy(int *x, int *y)

```



```

{
    char *term_name = ttyname(0), vcsa_name[100] = "/dev/vcsaX";
    FILE *vcsa;

    vcsa_name[strlen(vcsa_name)-1] = term_name[strlen(term_name)-1];
    if ( (vcsa = fopen(vcsa_name, "r")) == NULL) {
        fprintf(stderr, "kann '%s' nicht oeffnen\n", vcsa_name);
        return;
    }
    fgetc(vcsa); fgetc(vcsa);
    *x = fgetc(vcsa);
    *y = fgetc(vcsa);
    fclose(vcsa);
}

/*=====*/
/*===== conio-Teil =====*/
/*=====*/
#define BLACK      0
#define RED        1
#define GREEN      2
#define BROWN     3
#define BLUE       4
#define MAGENTA    5
#define CYAN       6
#define LIGHTGRAY  7
#define DARKGRAY   8
#define LIGHTRED   9
#define LIGHTGREEN 10
#define YELLOW     11
#define LIGHTBLUE  12
#define LIGHTMAGENTA 13
#define LIGHTCYAN  14
#define WHITE      15

#define BLINK      0x80

#define cscanf     scanf
#define cgets      gets

void clr_eol(void) { printf("\033[80X"); fflush(stdout); }
void gotoxy(int x, int y) { printf("\033[%d;%dH", y, x); fflush(stdout); }
void clrscr(void) { printf("\033[H\033[J"); fflush(stdout); }
void delline(void) { printf("\033[1M"); fflush(stdout); }
void insline(void) { printf("\033[1L"); fflush(stdout); }
void normvideo(void) { printf("\033[0m"); fflush(stdout); }
void highvideo(void) { printf("\033[1m"); fflush(stdout); }
void lowvideo(void) { printf("\033[2m"); fflush(stdout); }
int wherex(void) { int x, y; wherexy(&x,&y); return(x+1); }
int wherey(void) { int x, y; wherexy(&x,&y); return(y+1); }
int putch(int zeich) { printf("%c", zeich); fflush(stdout); }
int cputs(char *string) { printf("%s", string); fflush(stdout); }

```

```

void textcolor(int farbe) { printf("\033[2m\033[2;%d]", farbe & 0x7f);
                           printf("\033[%dm", (farbe & 0x80) ? 5 : 25);
                           fflush(stdout); }

void textbackground(int farbe) { printf("\033[%dm", 40+farbe%8);
                                fflush(stdout); }

void textattr(int attr)      { textcolor( attr & 0x0f );
                              textbackground( (attr >> 4) & 0x0f ); }

void sound(unsigned frequenz) { printf("\033[10;%d]", frequenz);
                              printf("\007");      fflush(stdout); }

void nosound(void)           { printf("\033[10;0]");  fflush(stdout); }

/*----- cprintf -----*/
void cprintf(const char *format, ...)
{
    char puffer[5000];
    va_list az;

    va_start(az, format);

    vsprintf(puffer, format, az);
    fprintf(stdout, "%s", puffer);
    fflush(stdout);

    va_end(az);
}

/*----- getch -----*/
int getch(void)
{
    int zeich;

    if (tty_cbreak(STDIN_FILENO, 0) < 0) {
        fprintf(stderr, "kann nicht in cbreak-Modus umschalten\n");
        return(EOF);
    }
    if (read(STDIN_FILENO, &zeich, 1) == 1)
        zeich &= 0xff;
    tty_reset(STDIN_FILENO);
    return(zeich);
}

/*----- getche -----*/
int getche(void)
{
    int zeich;

    if (tty_cbreak(STDIN_FILENO, 1) < 0) {
        fprintf(stderr, "kann nicht in cbreak-Modus umschalten\n");
        return(EOF);
    }

```

```

    }
    if (read(STDIN_FILENO, &zeich, 1) == 1)
        zeich &= 0xff;
    tty_reset(STDIN_FILENO);
    return(zeich);
}

/*----- kbhit -----*/
int kbhit(void)
{
    fd_set      lese_menge;
    struct timeval timeout;
    struct termios terminal;
    int         taste;

    if (tcgetattr(0, &alt_terminal) < 0)
        return(-1);
    terminal = alt_terminal;
    terminal.c_lflag &= ~ICANON; /* kanonischen Modus ausschalten */
    terminal.c_cc[VMIN] = 1; /* Immer nur 1 Byte; kein Timer */
    terminal.c_cc[VTIME] = 0;

    if (tcsetattr(0, TCSANOW, &terminal) < 0)
        return(-1);
    tty_modus = CBREAK;
    alt_ttyfd = 0;

    FD_ZERO(&lese_menge);
    FD_SET(0, &lese_menge);
    timeout.tv_sec = 0;
    timeout.tv_usec = 100;
    taste = select(1, &lese_menge, NULL, NULL, &timeout);

    tty_reset(STDIN_FILENO);
    return(taste);
}

/*----- delay -----*/
void delay(long millisek)
{
    int mikrosek = millisek*1000;
    struct timeval timeout;

    timeout.tv_sec = mikrosek / 1000000L;
    timeout.tv_usec = mikrosek % 1000000L;
    select(0, NULL, NULL, NULL, &timeout);
}

/*----- gettext -----*/
int gettext(int left, int top, int right, int bottom, char *puffer)
{
    char *term_name = ttyname(0), vcsa_name[100] = "/dev/vcsaX";
    FILE *vcsa;

```

```

int i, j, z=0,
    offset = 4 + ((top-1)*80+left-1)*2,
    pro_zeile = (right-left+1)*2,
    zeil_zahl = (bottom-top+1);

vcsa_name[strlen(vcsa_name)-1] = term_name[strlen(term_name)-1];

if ( (vcsa = fopen(vcsa_name, "r")) == NULL) {
    fprintf(stderr, "kann '%s' nicht oeffnen\n", vcsa_name);
    return;
}
fseek(vcsa, offset, SEEK_SET);
for (i=1; i<=zeil_zahl; i++) {
    for (j=1; j<=pro_zeile; j++)
        puffer[z++] = fgetc(vcsa);
    fseek(vcsa, 80*2L-pro_zeile, SEEK_CUR);
}
fclose(vcsa);
}

/*----- puttext -----*/
int puttext(int left, int top, int right, int bottom, char *puffer)
{
    char *term_name = ttyname(0), vcsa_name[100] = "/dev/vcsaX";
    FILE *vcsa;
    int i, j, z=0,
        offset = 4 + ((top-1)*80+left-1)*2,
        pro_zeile = (right-left+1)*2,
        zeil_zahl = (bottom-top+1);

    vcsa_name[strlen(vcsa_name)-1] = term_name[strlen(term_name)-1];
    if ( (vcsa = fopen(vcsa_name, "w")) == NULL) {
        fprintf(stderr, "kann '%s' nicht oeffnen\n", vcsa_name);
        return;
    }
    fseek(vcsa, offset, SEEK_SET);
    for (i=1; i<=zeil_zahl; i++) {
        for (j=1; j<=pro_zeile; j++)
            fputc(puffer[z++], vcsa);
        fseek(vcsa, 80*2L-pro_zeile, SEEK_CUR);
    }
    fclose(vcsa);
}

/*----- movetext -----*/
int movetext(int left, int top, int right, int bottom,
             int zielleft, int zieltop)
{
    char *term_name = ttyname(0), vcsa_name[100] = "/dev/vcsaX", puffer[5000];
    FILE *vcsa;
    int i, j, z=0,
        offset = 4 + ((top-1)*80+left)*2,
        pro_zeile = (right-left+1)*2,

```

```

        zeil_zahl = (bottom-top+1),
        ziel_offset = 4 + ((zieltop-1)*80+zielleft)*2;

vcsa_name[strlen(vcsa_name)-1] = term_name[strlen(term_name)-1];
if ( (vcsa = fopen(vcsa_name, "w+")) == NULL) {
    fprintf(stderr, "kann '%s' nicht oeffnen\n", vcsa_name);
    return;
}

fseek(vcsa, offset, SEEK_SET);
for (i=1; i<=zeil_zahl; i++) {
    for (j=1; j<=pro_zeile; j++)
        puffer[z++] = fgetc(vcsa);
    fseek(vcsa, 80*2L-pro_zeile-2, SEEK_CUR);
}

fseek(vcsa, ziel_offset, SEEK_SET);
z = 0;
for (i=1; i<=zeil_zahl; i++) {
    for (j=1; j<=pro_zeile; j++)
        fputc(puffer[z++], vcsa);
    fseek(vcsa, 80*2L-pro_zeile-2, SEEK_CUR);
}
fclose(vcsa);
}

/*----- _setcursortype -----*/
#define _NOCURSOR    0
#define _SOLIDCURSOR 1
#define _NORMALCURSOR 2

void _setcursortype(int cursor)
{
    if (cursor == _NOCURSOR) {
        printf("\033[?25l");
        printf("\033[?1000l");
    } else
        printf("\033[?25h");
    fflush(stdout);
}

/*----- getpass -----*/
#define MAX_PASSWORT 8 /* Maximal 8 Zeichen fuer ein Passwort */
char *getpass(const char *prompt)
{
    static char    puffer[MAX_PASSWORT + 1];
    char          *zgr;
    sigset_t       sig_maske, sig_alt;
    struct termios terminal, terminal_alt;
    FILE          *fz;
    int            zeich;

    if ( (fz = fopen(ctermid(NULL), "r+")) == NULL)

```

```

        return(NULL);
    setbuf(fz, NULL);

    /* Blockieren der Signale SIGINT u. SIGTSTP */
    sigemptyset(&sig_maske);
    sigaddset(&sig_maske, SIGINT);
    sigaddset(&sig_maske, SIGTSTP);
    sigprocmask(SIG_BLOCK, &sig_maske, &sig_alt);

    tcgetattr(fileno(fz), &terminal_alt);
    terminal = terminal_alt;
    terminal.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fz), TCSAFLUSH, &terminal);

    fputs(prompt, fz);

    zgr = puffer;
    while ( (zeich = getc(fz)) != EOF && zeich != '\n')
        if (zgr < &puffer[MAX_PASSWORT])
            *zgr++ = zeich;
    *zgr = '\0';
    putc('\n', fz); /* Echo fuer NL */

    /* Terminal in alten Zustand zuruecksetzen */
    tcsetattr(fileno(fz), TCSAFLUSH, &terminal_alt);

    /* Alte Signalmaske wieder herstellen */
    sigprocmask(SIG_SETMASK, &sig_alt, NULL);
    fclose(fz);

    return(puffer);
}

#endif /* __CONIO_H */

```

Programm 20.28 (conio.h): Emulation der Borland-Semigraphik auf einer Linux-Konsole

Es ist anzumerken, daß diese Borland-Emulation nur vom Superuser verwendet werden kann, da sie direkt auf die virtuelle Konsole `/dev/vcsa*` zugreift.

Um diese Realisierung in einem Programm verwenden zu können, gibt es zwei Möglichkeiten:

1. Man kopiert `conio.h` nach `/usr/include`. Dann kann man im Programm `#include <conio.h>` angeben.
2. Man kopiert `conio.h` in das Working-Directory. Dann muß man im Programm `#include "conio.h"` angeben.

Beispielprogramme zur Verwendung der Borland-Semigraphik-Emulation

Hier werden einige Demonstrationsprogramme vorgestellt, die diese einfache Emulation aus `conio.h` verwenden, um Semigraphik unter Linux zu programmieren.

Beispiel 1: Demonstrationsprogramm zur Cursorpositionierung mit conio.h

Das nachfolgende Programm 20.29 (**lkurpos.c**) demonstriert die Wirkung einiger Funktionen aus `conio.h`.

```
#include "conio.h" /* evtl.: #include <conio.h> */

int
main(void)
{
    int x, y;

    /*----- Bildschirm loeschen -----*/
    clrscr();

    /*----- In der 1.Spalte der 1.Zeile Text ausgeben -----*/
    gotoxy(1, 1);
    cputs("_<---- Position (1,1)");

    /*----- In der 20.Spalte der 5.Zeile Text ausgeben -----*/
    gotoxy(20, 5);
    cputs("_<---- Position (20,5)");

    /*----- Cursor relativ 10 Spalten nach links und 5 Zeilen nach unten ---*/
    /*----- ziehen und dann Position ausgeben -----*/
    gotoxy(20, 5);
    x=wherex()-10;
    y=wherey()+5;
    gotoxy(x,y);
    printf("_<---- Position (%d,%d)", wherex(), wherey());

    /*----- In der 50.Spalte der 20.Zeile Cursor-Position ausgeben -----*/
    gotoxy(50,20);
    printf("_<---- Position (%d,%d)", wherex(), wherey());

    /*----- In der 56.Spalte der 25.Zeile Text ausgeben -----*/
    gotoxy(56,25);
    printf("Position (78,25) ---->");

    /*----- In der 78.Spalte der 25.Zeile Unterstrich ausgeben -----*/
    gotoxy(78,25);
    putchar('_');

    /*----- Nach Tastendruck Bildschirm loeschen und Programm beenden -----*/
    getch();
    clrscr();

    exit(0);
}
```

Programm 20.29 (lkurpos.c): Demonstrationsbeispiel zur Cursorpositionierung mit conio.h

Hat man das Programm 20.29 (`lkurpos.c`) kompiliert und gelinkt

```
cc -o lkurpos lkurpos.c
```

und man startet es, so liefert es folgende Bildschirmausgabe:

```

_<---- Position (1,1)

                _<---- Position (20,5)

            _<---- Position (10,10)

                                _<---- Position (50,20)

                                                Position (78,25) ---->_

```

Programmende mit Löschen des Bildschirms nach einem Tastendruck.

Beispiel 2: Schnee und Luftballone mit conio.h

In Kapitel 20.9.9 wurde das Programm 20.16 (`balflock.c`) vorgestellt, das das Aufsteigen von Luftballons bzw. das Fallen von Schneeflocken simuliert, je nachdem was der Benutzer wünscht. Diese Aufgabe soll hier nun unter Verwendung von `conio.h` gelöst werden, wobei die aufsteigenden Luftballone aber zufällige Farben besitzen sollen.

Das nachfolgende Programm 20.30 (`lkbalflo.c`) löst diese Aufgabe.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "conio.h" /* evtl.: #include <conio.h> */

#define MAXX 80
#define MAXY 25

int

```



```

main(void)
{
    char wahl;
    int i;

    srand(time(NULL)+getpid()); /* Zufallszahlengenerator initialisieren */

    /*----- Einlesen, ob Luftballone oder Schneeflocken gewuenscht -----*/
    while (1) {
        clrscr();
        printf("1 : Luftballone steigen lassen\n");
        printf("2 : Schneeflocken fallen lassen\n\n");
        printf("Was wollen Sie tun ? "); fflush(stdout);
        wahl = getche();
        if (wahl == '1' || wahl == '2')
            break;
    }

    /*----- Simulieren der Luftballone bzw. Schneeflocken -----*/
    clrscr();
    do {
        for (i=1 ; i<=MAXX ; i++) { /* Fuer jede Spalte */
            if (rand()%100<=3) { /* mit 4% Wahrscheinlichkeit */
                if (wahl == '1') { /* Luftballon oder Schneeflocke */
                    gotoxy(i,MAXY);
                    textcolor(rand()%15+1); /* Luftballon-Farbe zufaellig */
                    /* zwischen 1 und 15 waehlen */
                    putchar('o');
                } else {
                    gotoxy(i,1);
                    putchar('*');
                }
            }
        }
        gotoxy(1,1);
        if (wahl == '1')
            delline(); /* Bei Luftballonen: Bild nach oben ziehen */
        else
            insline(); /* Bei Schneeflocken: Bild nach unten ziehen */
        delay(100); /* 100 Millisekunden pausieren */
    } while (!kbhit());

    textcolor(WHITE);
    clrscr();
    system("reset");

    exit(0);
}

```

Programm 20.30 (**lkbalflo.c**): Schnee und Luftballone

Beispiel 3: Männlein im Walde

In Kapitel 20.9.9 wurde das Programm 20.17 (`kuckkuck.c`) vorgestellt, das ständig ein Männchen an einer anderen Stelle des Bildschirms zeigt. Diese Aufgabenstellung wird hier nun mit zwei Programmen unter der Verwendung von `conio.h` gelöst:

Das Programm 20.31 (`lk_kuck.c`) verwendet dazu die Funktion ***movetext*** und das Programm 20.32 (`lk_kuck2.c`) benutzt hierfür die beiden Funktionen ***gettext*** und ***puttext***.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "conio.h"    /* evtl. auch: #include <conio.h> */

#define MAXX  80
#define MAXY  25

int
main(void)
{
    int  altx=1, alty=1, x, y, i, j;

    srand(time(NULL)); /* Zufallszahlengenerator initialisieren */

    /*----- Maennchen in obere linke Ecke zeichnen -----*/
    clrscr();
    gotoxy(altx,alty);
    cprintf("  0  \n\r");
    cprintf("--U--\n\r");
    cprintf(" /  \ \ \n\r");

    /*----- Maennchen zufaellig am Bildschirm herumspringen lassen -----*/
    do {
        x=rand()%(MAXX-5)+1; /* Neue Koordinaten zufaellig bestimmen */
        y=rand()%(MAXY-3)+1;
        movetext(altx,alty,altx+4,alty+2,x,y); /* altes Bild dorthin kopieren */

        for (j=1 ; j<=MAXY ; j++)    /* Altes Maennchen-Bild loeschen */
            if ( !(j>=y && j<=y+2) ) {
                gotoxy(1,j);
                clreol();
            } else {
                gotoxy(1,j);
                for (i=1 ; i<x ; i++)
                    putchar(' ');
                gotoxy(x+5,j);
                clreol();
            }
        altx=x; /* Koordinaten des aktuellen Bilds in altx,alty festhalten */
        alty=y;
        delay(500); /* Halbe Sekunde pausieren */
    } while (!kbhit());
}
```

```

    clrscr();

    exit(0);
}

```

Programm 20.31 (lk_kuck.c): Männlein im Walde mit movetext

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "conio.h" /* evtl. auch: #include <conio.h> */

#define MAXX 80
#define MAXY 25

int
main(void)
{
    int x=1, y=1;
    char mann[100];

    srand(time(NULL)); /* Zufallszahlengenerator initialisieren */

    /*----- Maennchen in obere linke Ecke zeichnen -----*/
    clrscr();
    gotoxy(x,y);
    cprintf(" 0 \n\r");
    cprintf("--0--\n\r");
    cprintf(" / \ \ \n\r");
    gettext(x, y, x+4, y+2, mann); /* Maennchen-Bild speichern */

    /*----- Maennchen zufaellig am Bildschirm herumspringen lassen -----*/
    do {
        clrscr(); /* Bildschirm loeschen */
        x=rand()%(MAXX-5)+1; /* Neue Koordinaten zufaellig bestimmen */
        y=rand()%(MAXY-3)+1;
        puttext(x, y, x+4, y+2, mann); /* Maennchen an neue Position malen */
        delay(500); /* Halbe Sekunde pausieren */
    } while (!kbhit());
    clrscr();

    exit(0);
}

```

Programm 20.32 (lk_kuck2.c): Männlein im Walde mit gettext und puttext

Beispiel 4: Ausgabe aller Vorder- und Hintergrundfarben

Das folgende Programm 20.33 (lk_farbe.c) gibt zunächst alle Vordergrundfarben aus, wobei es alle Ausgaben, die eine ungerade Farbnummer haben, blinken lässt. Danach zeigt es alle Hintergrundfarben, wobei es als Vordergrundfarbe eine um 1 höhere Farbnummer verwendet. Diese Ausgabe zeigt es zweimal: einmal durch die Verwendung der

beiden Funktionen **textbackground** und **textcolor** und das andere mal durch Verwendung der Funktion **textattr**.

```
#include "conio.h"    /* evtl. auch: #include <conio.h> */

int
main(void)
{
    int i, farbe;

    /*--- Ganzen Bildschirmhintergrund auf Hellgrau einstellen ---*/
    textbackground(LIGHTGRAY);
    clrscr();

    /*--- Ausgabe aller Vordergrundfarben; jede 2. dabei blinkend ---*/
    for (i=0 ; i<=15 ; i++) {
        farbe=i;
        if (farbe%2==1)
            farbe += BLINK;
        textcolor(farbe);
        printf("Vordergrund-Farbe %d\n\r", i);
    }
    gotoxy(1,25); printf("Weiter mit beliebiger Taste.....");
    getch();

    /*--- Ausgabe aller Hintergrundfarben -----*/
    clrscr();
    gotoxy(1,5);
    for (i=0 ; i<=7 ; i++) {
        textcolor(i+1);
        textbackground(i);
        printf("Hintergrund-Farbe %d; Vordergrund-Farbe %d\n\r\n\r", i, i+1);
    }
    gotoxy(1,25); printf("Weiter mit beliebiger Taste.....");
    getch();

    /*--- Gleiche Ausgabe wie zuvor unter Verwendung von textattr -----*/
    clrscr();
    gotoxy(1,5);
    for (i=0 ; i<=7 ; i++) {
        textattr((i<<4) + i+1);
        printf("Hintergrund-Farbe %d; Vordergrund-Farbe %d\n\r\n\r", i, i+1);
    }
    gotoxy(1,25); printf("Ende mit beliebiger Taste.....");
    getch();

    /*--- Bildschirm wieder auf die normalen Werte zurueckstellen -----*/
    system("reset");

    exit(0);
}
```

Programm 20.33 (lk_farbe.c): Ausgabe aller Vorder- und Hintergrundfarben

Beispiel 5: Ermitteln der Tastencodes bei einer Linux-Konsole

Das folgende Programm 20.34 (`lk_tcode.c`) gibt zu jeder gedrückten Taste die zugehörigen Tastencodes aus. Beim Drücken der Taste 'q' beendet sich dieses Programm.

```
#include <stdio.h>
#include "conio.h" /* evtl. auch: #include <conio.h> */

#define ESC 27

int
main(void)
{
    char zeich;

    clrscr();
    printf("Tastencodes\n");
    printf("=====\n\n");
    printf("Bei jedem Tastendruck gibt dieses Programm den zugehoerigen\n");
    printf("Tastencode aus\n\n");

    while ((zeich=getch()) != 'q') { /* Taste q bewirkt Programmabbruch */
        if (zeich == ESC)
            printf("^[");
        else
            printf("%2c", zeich);
        printf(" = 0x%x (%3d; \\%03o)\n", zeich, zeich, zeich);
    }
    exit(0);
}
```

Programm 20.34 (`lk_tcode.c`): Ausgeben der Tastencodes bei einer Linux-Konsole

20.12 Die Programmierung von virtuellen Konsolen unter Linux

Unter Linux ist es möglich, mehrere Terminal-Sitzungen an einem Bildschirm gleichzeitig zu betreiben. Man spricht dann von virtuellen Konsolen. Das Umschalten zwischen diesen virtuellen Konsolen erfolgt unter Linux üblicherweise mit den Tastenkombinationen Alt-F1 bis Alt-F6 bzw. mit Strg-Alt-F1 bis Strg-Alt-F6 unter X-Windows.

Hat man X-Windows gestartet, so kann man dorthin mit der Tastenkombination Alt-F7 bzw. Strg-Alt-F7 umschalten. Jede virtuelle Konsole kann dabei ihre eigenen Tastatur- und Terminal-Einstellungen haben.

Die von Linux zur Programmierung von virtuellen Konsolen angebotene Schnittstelle baut auf einer Schnittstelle auf, die auch von einigen anderen Unix-Versionen verwendet wird. Linux hat diese Schnittstelle jedoch noch erweitert.

20.12.1 Wichtige Headerdateien, Funktionen und Strukturen

Zur Programmierung von virtuellen Konsolen benötigt man eine ganze Reihe von Headerdateien:

```
#include <fcntl.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <sys/vt.h>
#include <sys/kd.h>
#include <sys/param.h>
```

Die meisten Aktionen, die virtuelle Konsolen betreffen, werden mit der in `<sys/ioctl.h>` deklarierten Funktion ***ioctl*** durchgeführt.

```
#include <sys/ioctl.h>
```

```
int ioctl(int fd, int operation, ...)
```

gibt zurück: -1 (bei Fehler); 0 bei Erfolg

Die entsprechenden durchzuführenden operationen für virtuelle Konsolen sind in `<sys/vt.h>` bzw. `<sys/kd.h>` definiert.

In `<sys/vt.h>` befinden sich Konstanten, die mit dem Präfix **VT** beginnen und Aktionen auf den virtuellen Bildschirm einer virtuellen Konsole ermöglichen.

In `<sys/kd.h>` befinden sich Konstanten, die mit dem Präfix **KD** beginnen und Aktionen ermöglichen, die den Zeichensatz und die Tastatur einer virtuellen Konsole betreffen.

Hier wird vor allen Dingen auf die Operationen in `<sys/vt.h>` eingegangen.

Zwei wichtige Strukturen sowie die zugehörigen Konstanten aus `<sys/vt.h>`, die bei der Funktion ***ioctl*** im Zusammenhang mit virtuellen Terminals angegeben werden können, sind:

```
#define VT_AUTO 0x00 /* Systemkern schaltet automatisch zwischen
                        den virtuellen Konsolen hin und her,
                        wenn die entsprechende Tastenkombination
                        gedrueckt wird oder ein Programm eine
                        entsprechende Anforderung an den Kern
                        schickt. */

#define VT_PROCESS 0x01 /* Kern fragt vor dem Umschalten auf eine
                          andere virtuelle Konsole nach, ob dieses
                          Umschalten auch wirklich stattfinden soll */

struct vt_mode {
    char mode; /* ist mit einer der beiden Konstanten
                VT_AUTO oder VT_PROCESS gesetzt. */
    char waitv; /* wenn gesetzt, wird das Schreiben auf eine
                 virtuelle Konsole, die z.Z. nicht aktiv ist,
```

```

        solange blockiert, bis diese aktiviert wird. */
short relsig; /* Signal, das Systemkern sendet, wenn er vom
               Prozeß die Freigabe der virtuellen Konsole
               fordert. */
short acqsig; /* Signal, das Systemkern sendet, wenn er den
               Prozeß warnen will, daß er sich diese
               virtuelle Konsole angeeignet hat. */
short frsig; /* ungenutzt; ist aber aus Kompatibilitaets-
               grunden zu SVR4 auf 0 zu setzen. */
};

struct vt_stat {
    unsigned short v_active; /* enthaelt die Nummer der gerade
                              aktiven virtuellen Konsole */
    unsigned short v_signal; /* Zu schickendes Signal
                              (noch nicht implementiert) */
    unsigned short v_state; /* Bitmaske, die anzeigt, welche der
                              ersten 16 virtuellen Konsolen
                              momentan offen sind. Anmerkung:
                              Linux unterstuetzt bis zu
                              63 virtuelle Konsolen */
};

```

20.12.2 Öffnen einer neuen virtuellen Konsole

Bevor man mit **ioctl** Aktionen auf virtuellen Konsolen durchführen kann, muß man zunächst die Gerätedatei `/dev/tty` für das Terminal öffnen, da **ioctl** einen Filedeskriptor erwartet:

```

if ( (fd = open("/dev/tty", O_RDWR)) < 0 )
    fehler_meld(FATAL_SYS, "kann '/dev/tty' nicht oeffnen");

```

Um nun eine ungenutzte virtuelle Konsole zu finden, die noch nicht von anderen Prozessen benutzt wird, muß man beim **ioctl**-Aufruf als operation die Konstante `VT_OPENQRY` angeben:

```

int vt_nr;
....
if (ioctl(fd, VT_OPENQRY, &vt_nr) < 0 || vt_nr == -1) {
    fehler_meld(..., "kann keine freie virtuelle Konsole finden");
    ....
}

```

Wenn weniger als 63 virtuelle Konsolen gerade benutzt werden, dann allokiert der Systemkern dynamisch eine neue virtuelle Konsole.

20.12.3 Erfragen von Informationen zu virtuellen Konsolen

Um zu ermitteln, ob das aktuelle Terminal eine virtuelle Konsole ist, muß man das aktuelle Terminal öffnen und anschließend beim **ioctl**-Aufruf als operation die Konstante `VT_GETMODE` angeben:

```

struct vt_mode  vtmodus;

if ( (fd = open("/dev/tty", O_RDWR)) < 0)
    fehler_meld(FATAL_SYS, "kann '/dev/tty' nicht oeffnen");
....
if (ioctl(fd, VT_GETMODE, &vtmodus) < 0) {
    fehler_meld(..., "Aktuelles Terminal ist keine virtuelle Konsole");
    ....
}

```

Um die Nummer der aktuellen virtuellen Konsole zu ermitteln, muß man beim **ioctl**-Aufruf für operation die Konstante **VT_GETSTATE** angeben:

```

struct vt_stat  vtstatus;
....
if (ioctl(fd, VT_GETSTATE, &vtstatus) < 0) {
    fehler_meld(..., "kann Status für virtuelle Konsolen nicht ermitteln");
    ....
}
aktuelle_virtuelle_konsole = vtstatus.v_active;

```

20.12.4 Einfache Programmierung von virtuellen Konsolen

Um auf eine andere virtuelle Konsole umzuschalten, muß bei **ioctl** als operation die Konstante **VT_ACTIVATE** angegeben werden. Das Aktivieren einer virtuellen Konsole kann unter Umständen einige Zeit dauern, vor allem dann, wenn diese sich im Graphikmodus befindet. Soll ein Prozeß warten, bis die entsprechende virtuelle Konsole wirklich aktiviert ist, muß als operation bei einem erneuten **ioctl**-Aufruf die Konstante **VT_WAITACTIVE** angegeben werden.

```

int vt_nr;
....
ioctl(vt_fd, VT_ACTIVATE, vt_nr);
ioctl(vt_fd, VT_WAITACTIVE, vt_nr);

```

Virtuelle Konsolen werden zwar automatisch allokiert, wenn sie geöffnet werden, jedoch werden geschlossene Konsolen nicht wieder automatisch aus dem Speicher entfernt. Um den für eine virtuelle Konsole reservierten Speicherbereich wieder freizugeben, muß man **ioctl** mit der Operation **VT_DISALLOCATE** aufrufen.

```

int vt_nr;
....
ioctl(vt_fd, VT_DISALLOCATE, vt_nr);

```

Beispiel

Starten einer Shell auf einer neuen virtuellen Konsole

Das folgende Programm 20.35 (vk_erst.c) sucht eine ungenutzte virtuelle Konsole und startet darauf eine Shell. Erst wenn der Benutzer diese Shell beendet, wird wieder auf die ursprüngliche virtuelle Konsole zurückgeschaltet, und die andere virtuelle Konsole wird wieder freigegeben.


```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/vt.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

/*----- main -----*/
int
main(void)
{
    int          vt_nr, vt_fd;
    struct vt_stat vtstat;
    char         terminal_name[100];
    pid_t        kind;

    if ( (vt_fd = open("/dev/tty", O_RDWR, 0)) < 0)
        fehler_meld(FATAL_SYS, "kann /dev/tty nicht oeffnen");

    if (ioctl(vt_fd, VT_GETSTATE, &vtstat) < 0)
        fehler_meld(FATAL_SYS, "tty ist keine virtuelle Konsole");

    if (ioctl(vt_fd, VT_OPENQRY, &vt_nr) < 0 || vt_nr == -1)
        fehler_meld(FATAL_SYS, "kann keine freie virtuelle Konsole finden");

    sprintf(terminal_name, "/dev/tty%d", vt_nr);
    if (access(terminal_name, (W_OK|R_OK)) < 0)
        fehler_meld(FATAL_SYS, "Unzureichende Zugriffsrechte auf tty");

    if ( (kind = fork()) == 0) {
        ioctl(vt_fd, VT_ACTIVATE, vt_nr);
        ioctl(vt_fd, VT_WAITACTIVE, vt_nr);
        setsid();
        close(0); close(1); close(2);
        close(vt_fd);
        vt_fd = open(terminal_name, O_RDWR, 0); dup(vt_fd); dup(vt_fd);
        printf("-----\n");
        printf("        .....Virtuelle Konsole (tty%d).....\n", vt_nr);
        printf("-----\n\n");
        execlp("/bin/bash", "bash", NULL);
    }
    wait(NULL);

    ioctl(vt_fd, VT_ACTIVATE, vtstat.v_active);

```

```

    ioctl(vt_fd, VT_WAITACTIVE, vtstat.v_active);
    ioctl(vt_fd, VT_DISALLOCATE, vt_nr);

    exit(0);
}

```

Programm 20.35 (vk_erst.c): Starten einer Shell auf einer anderen virtuellen Konsole

Das folgende Programm 20.36 (vk_zweit.c) zeigt einige weitere nützliche Konstanten, die bei **ioctl** im Zusammenhang mit virtuellen Konsolen benutzt werden können:

VT_RELDISP	gibt eine virtuelle Konsole frei.
VT_ACKAOK	übernimmt eine virtuelle Konsole, so daß diese aktiv wird.
VT_SETMODE	setzt den Modus für die gerade aktive virtuelle Konsole.
KIOCSOUND	schaltet den Terminal-Ton mit der als dritten Parameter bei ioctl angegebener Frequenzzahl ein; Wert 0 schaltet den Ton wieder aus.

Dieses Programm 20.36 (vk_zweit.c) sucht eine ungenutzte virtuelle Konsole, und setzt mit einem **setterm**-Aufruf deren Hintergrundfarbe auf Türkis und deren Vordergrundfarbe auf Schwarz, so daß der Benutzer immer sofort weiß, auf welcher virtuellen Konsole er sich befindet. Zudem schaltet es für diese virtuelle Konsole den Terminal-Ton ein. Das Zurückschalten von der neu eingerichteten virtuellen Konsole zur ursprünglichen Konsole erfolgt mit der Eingabe von 1. Mit der Eingabe von 2 kann wieder von der ursprünglichen auf die neu eingerichtete virtuelle Konsole umgeschaltet werden. Um eine virtuelle Konsole zu beenden, muß man q eingeben, wobei jedoch immer zuerst die neu eingerichtete Konsole zu beenden ist, bevor das Programm auf der ursprünglichen Konsole beendet werden kann. Alle anderen Eingaben werden als einzelne Zeichen an der jeweiligen Konsole wieder ausgegeben.

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/vt.h>
#include <sys/kd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "eighdr.h"

void vk_signale_einrichten(void);
void vk_freigabe(int signr);
void vk_uebernehmen(int signr);
void vk_umschalten_init(int fd);
void vk_umschalten(pid_t akt_pid, pid_t neu_pid, int neu_nr);

```

```

int vt_fd, vt_nr, hertz = 0;

/*----- main -----*/
int
main(void)
{
    int          zeich, eig_nr, and_nr;
    struct vt_stat vtstat;
    char         terminal_name[100];
    pid_t        pid;

    vk_signale_einrichten();

    if ( (vt_fd = open("/dev/tty", O_RDWR, 0)) < 0)
        fehler_meld(FATAL_SYS, "kann /dev/tty nicht oeffnen");

    if (ioctl(vt_fd, VT_GETSTATE, &vtstat) < 0)
        fehler_meld(FATAL_SYS, "tty ist keine virtuelle Konsole");

    if (ioctl(vt_fd, VT_OPENQRY, &and_nr) < 0 || and_nr == -1)
        fehler_meld(FATAL_SYS, "kann keine freie virtuelle Konsole finden");

    sprintf(terminal_name, "/dev/tty%d", and_nr);
    if (access(terminal_name, (W_OK|R_OK)) < 0)
        fehler_meld(FATAL_SYS, "Unzureichende Zugriffsrechte auf tty");

    vt_nr = eig_nr = vtstat.v_active;
    vk_umschalten_init(vt_fd);

    if ( (pid = fork()) == 0 ) {
        ioctl(vt_fd, VT_ACTIVATE, and_nr);
        ioctl(vt_fd, VT_WAITACTIVE, and_nr);
        and_nr = eig_nr;
        if (ioctl(vt_fd, VT_GETSTATE, &vtstat) < 0)
            fehler_meld(FATAL_SYS, "tty ist keine virtuelle Konsole");
        vt_nr = eig_nr = vtstat.v_active;
        setsid();
        close(0); close(1); close(2);
        close(vt_fd);
        vt_fd = open(terminal_name, O_RDWR, 0); dup(vt_fd); dup(vt_fd);
        hertz = 2000;
        ioctl(vt_fd, KIOCSOUND, hertz);
        system("setterm -foreground black -background cyan -store");
        system("setterm -clear");
        printf("-----\n");
        printf("          Virtuelle Konsole (tty%d) aktiviert\n", eig_nr);
        printf("-----\n\n");
        while ( (zeich = getchar()) != 'q' && getppid() != 1)
            if (zeich == 'l') {
                fflush(NULL);
                vk_umschalten(getpid(), getppid(), and_nr);
            } else if (zeich != '\n')
                printf("...%c\n", zeich);
    }
}

```

```

        vk_umschalten(getpid(), getppid(), and_nr);
        exit(0);
    } else if (pid > 0) {
        while ( (zeich = getchar()) != 'q' || waitpid(pid, NULL, WNOHANG) == 0)
            if (zeich == '2' && waitpid(pid, NULL, WNOHANG) == 0) {
                fflush(NULL);
                vk_umschalten(getpid(), pid, and_nr);
            } else if (zeich == 'q' && waitpid(pid, NULL, WNOHANG) == 0) {
                printf(".....Bitte zuerst andere Konsole mit 'q' beenden\n");
                printf(".....Umschalten auf andere Konsole mit '2' moeglich\n");
            } else if (zeich != '\n')
                printf("...%c\n", zeich);

        if (ioctl(vt_fd, VT_DISALLOCATE, and_nr) < 0)
            fehler_meld(WARNUNG_SYS, "kann virtuelle Konsole nicht freigeben");
    }
    exit(0);
}

/*----- vk_signale_einrichten -----*/
/* Hier werden die beiden Signale SIGUSR1 und SIGUSR2 zur Freigabe *
 * bzw. Uebernahme von virtuellen Konsolen verwendet. Es koennen auch *
 * andere Signale verwendet werden, wobei jedoch immer zu beachten *
 * ist, dass diese beiden Signale nicht noch von anderen Funktionen *
 * oder fuer andere Zwecke benutzt werden. */
void
vk_signale_einrichten(void)
{
    struct sigaction sigact;

    /* Keine anderen Signale maskieren, wenn diese Signalhandler
       aufgerufen werden */
    sigemptyset(&sigact.sa_mask);

    /* Evtl. sigaddset-Aufrufe hinzufuegen, wenn diese waehrend der
       Umschaltung von virtuellen Konsolen maskiert werden sollen. */
    sigact.sa_flags = 0;
    sigact.sa_handler = vk_freigabe;
    sigaction(SIGUSR1, &sigact, NULL);
    sigact.sa_handler = vk_uebernehmen;
    sigaction(SIGUSR2, &sigact, NULL);
}

/*----- vk_freigabe -----*/
void
vk_freigabe(int signr)
{
    ioctl(vt_fd, VT_RELDISP, 1);
}

/*----- vk_uebernehmen -----*/
void
vk_uebernehmen(int signr)

```

```
{
    ioctl(vt_fd, VT_RELDISP, VT_ACKACQ);
    ioctl(vt_fd, KIOCSOUND, hertz);
    printf("-----\n");
    printf(".....Virtuelle Konsole (tty%d) aktiviert.....\n", vt_nr);
    printf("-----\n\n");
}

/*----- vk_umschalten_init -----*/
void
vk_umschalten_init(int fd)
{
    struct vt_mode    vtmodus;

    vtmodus.mode      = VT_PROCESS;
    vtmodus.waitv     = 1;
    vtmodus.relsig     = SIGUSR1;
    vtmodus.acqsig     = SIGUSR2;
    vtmodus.frsig      = 0;
    ioctl(fd, VT_SETMODE, &vtmodus);
}

/*----- vk_umschalten -----*/
void
vk_umschalten(pid_t akt_pid, pid_t neu_pid, int neu_nr)
{
    kill(akt_pid, SIGUSR1);
    kill(neu_pid, SIGUSR2);
    ioctl(vt_fd, VT_ACTIVATE, neu_nr);
    ioctl(vt_fd, VT_WAITACTIVE, neu_nr);
}
```

Programm 20.36 (**vk_zweit.c**): Hin- und Herschalten zwischen zwei unterschiedlich konfigurierten virtuellen Konsolen

Als letztes werden noch zwei **ioctl**-Konstanten vorgestellt, mit denen man die Umschaltung von virtuellen Konsolen vollständig aus- bzw. wieder einschalten kann.

```
ioctl(vt_fd, VT_LOCKSWITCH, 0);    /* Einschalten der Umschaltung */
ioctl(vt_fd, VT_UNLOCKSWITCH, 0); /* Ausschalten der Umschaltung */
```

Dies war lediglich eine Einführung in die Programmierung von virtuellen Konsolen. Nützliche Programme im Zusammenhang mit virtuellen Konsolen sind unter anderem:

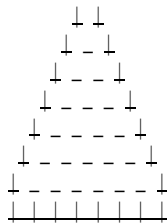
chvt n	wechselt zur n-ten virtuellen Konsolen; verhält sich weitgehend wie die Tastenkombinationen Alt-Fn. Dieses Kommando kann allerdings im Unterschied zu den Tastenkombinationen aus einem Programm heraus aufgerufen werden.

deallocvt [n]	gibt den Speicherplatz von allen ungenutzten virtuellen Konsolen wieder frei. Ist n angegeben, so wird nur der Speicherplatz der n. ten virtuellen Konsole freigegeben, wenn diese zur Zeit nicht genutzt wird.
splitvt ...	teilt den Bildschirm in zwei Fenster, in denen gleichzeitig zwei Shells ablaufen. Näheres hierzu kann mit man splitvt erfragt werden.

20.13 Übung

20.13.1 Das Galton-Brett

Das Galton-Brett ist ein schräg aufgestelltes Brett, auf dem Hindernisse (z.B. Nägel) nach folgenden Prinzip angeordnet sind:



Läßt man nun eine Kugel oben an diesem Brett los, so fällt sie an jedem Hindernis mit der gleichen Wahrscheinlichkeit in das Loch links oder rechts. Erstellen Sie ein Programm `galton.c`, das dieses zufällige Durchlaufen der Kugel am Bildschirm simuliert. Die Größe des Brettes soll dabei variabel und vom Benutzer wählbar sein.

Erstellen Sie zusätzlich ein Programm `sgalton.c`, das diese Aufgabenstellung mit S-Lang löst, wobei jedoch neue Kugeln immer eine andere zufällige Farbe besitzen sollen.

20.13.2 Simulation eines Wettrennens

Erstellen Sie ein Programm `rennen.c`, das einen 60 Meterlauf am Bildschirm simuliert, an dem bis zu 8 Läufer teilnehmen können.

Die Zahl der teilnehmenden Läufer (maximal 8) soll der Benutzer wählen können.

Die Reihenfolge, in der die Läufer im Ziel einlaufen, soll unten am Bildschirm angezeigt werden.

Das Programm `rennen.c` soll keine Arrays verwenden, um die aktuellen Läuferpositionen zu speichern. Diese Informationen soll sich das Programm vom Bildschirm erfragen.

Erstellen Sie zusätzlich ein Programm `lkrennen.c`, das diese Aufgabenstellung unter Verwendung von `conio.h` (Emulation der Borland-Graphik) löst, wobei jedoch die einzelnen Läufer unterschiedliche Farben besitzen sollen. Zusätzlich soll bei diesem Programm das Rennen durch einen Tastendruck angehalten werden können und erst beim nächsten Tastendruck wieder fortgesetzt werden. Wie das Programm `rennen.c`, so soll auch das

Programm `lkrennen.c` keine Arrays verwenden, um die aktuellen Positionen der einzelnen Läufer zu speichern.

Möglicher Ablauf dieses Programms `rennen.c` bzw. `lkrennen.c`:

1. Bild:

```
60-Meter Lauf
=====

Dieses Programm simuliert einen 60 Meter Lauf, an dem bis zu
8 Laeufer teilnehmen koennen.

Wieviele Laeufer sollen teilnehmen: 8
```

2. Bild:

```
1 *|                                     |
2 *|                                     |
3 *|                                     |
4 *|                                     |
5 *|                                     |
6 *|                                     |
7 *|                                     |
8 *|                                     |
```

Start mit einer beliebiger Taste.....

3. Bild (Momentaufnahme während des Rennens):

```
1 ***** |
2 ***** |
3 ***** |
4 ***** |
5 ***** |
6 ***** |
7 ***** |
8 ***** |
           Zieleinlauf:
```

4. Bild (nachdem letzter Läufer das Ziel erreicht hat):

```
1 *****
2 *****
3 *****
4 *****
5 *****
6 *****
7 *****
8 *****
           Zieleinlauf:
           1.Platz: Laeufer 8
           2.Platz: Laeufer 7
           3.Platz: Laeufer 6
           4.Platz: Laeufer 3
           5.Platz: Laeufer 4
           6.Platz: Laeufer 1
           7.Platz: Laeufer 2
           8.Platz: Laeufer 5
```


20.13.3 Realisierung von conio.h mit curses

Der Nachteil der in Kapitel 20.11.6 beschriebenen Headerdatei `conio.h` zur Emulation der Borland-Semigraphik unter Linux ist, daß man Superuser sein muß, um damit arbeiten zu können. Erstellen Sie unter Verwendung der `curses`-Routinen eine Headerdatei `ncconio.h`, die es auch normalen Benutzern erlaubt, diese Borland-Semigraphik-Emulation unter Linux/Unix zu benutzen. Schlagen Sie bei Bedarf direkt in `<curses.h>` bzw. `<ncurses.h>` nach oder eben in der entsprechenden Manpage zu `curses` bzw. `ncurses`.

20.13.4 Spiel 21 gegen den Computer

Erstellen Sie unter Verwendung von `conio.h` ein Programm `spiel21.c`, bei dem der Benutzer das *Spiel 21* gegen den Computer spielt. Beim *Spiel 21* liegen zu Beginn 21 Streichhölzer auf dem Tisch. Nun müssen abwechselnd zwischen 1 und 4 Streichhölzer vom Tisch genommen werden. Derjenige, der das letzte Streichholz vom Tisch nimmt, hat verloren. Die noch auf dem Tisch liegenden Streichhölzer sollen dabei immer farbig angezeigt werden.

20.13.5 Chaos-Musik

Erstellen Sie ein Programm `musik.c`, das unter Verwendung von `conio.h` Chaos-Musik erzeugt, indem es wiederholt folgende Funktion berechnet:

$$x_{t+1} = 4 * x_t * (1 - x_t) \quad ; \text{für } t=0, 1, 2, 3, \dots, n \text{ (n ist einzugeben)}$$

Der Startwert x_0 soll dabei eingegeben werden; er muß größer als 0, aber kleiner als 1 sein.

Für jeden Zeitpunkt t wird dann die Tonfrequenz $x*2000$ verwendet. Diese Frequenz soll dabei nicht nur akustisch, sondern auch farblich horizontal am Bildschirm angezeigt werden.

Für die farbliche Ausgabe sollte zu Beginn die maximale Frequenz von 2000 durch 7 (maximale Anzahl von Hintergrundfarben) geteilt werden, was dann einer Farbfrequenz-Einheit entspricht.

Für jede errechnete Tonfrequenz sollte dann eine horizontale Leiste ausgegeben werden, die alle abgedeckten Frequenzstufen farblich unterschiedlich nebeneinander ausgibt, bevor der entsprechende Ton dazu ausgegeben wird. Würde z.B. die momentan errechnete Frequenz 1652 sein, dann könnte das, wie es unten im 2. Bild gezeigt ist, am Bildschirm angezeigt werden (leider ist hier keine Farbausgabe möglich). Nach dieser Ausgabe soll der gesamte Bildschirm gelöscht werden, um dann den nächsten Ton am Bildschirm zu veranschaulichen.

Ein Ton sollte immer 200 Millisekunden lang erklingen.

Für die Erstellung dieses C-Programms werden einige Funktionen aus der Headerdatei `conio.h` benötigt:

Die noch vorhandenen Leben und die bisher zurückgelegten Meter sollten immer links unten am Bildschirm angezeigt werden. Fährt ein Benutzer mehrmals, dann sollte ihm immer nach der Beendigung eines Durchgangs zusätzlich noch die bisher weiteste zurückgelegte Strecke angezeigt werden.

Das Ende eines Durchgangs (Auftreffen auf ein Hindernis oder keine Leben mehr) sollte durch eine simulierte »Explosion« angezeigt werden, bevor der Benutzer gefragt wird, ob er nochmals fahren möchte.

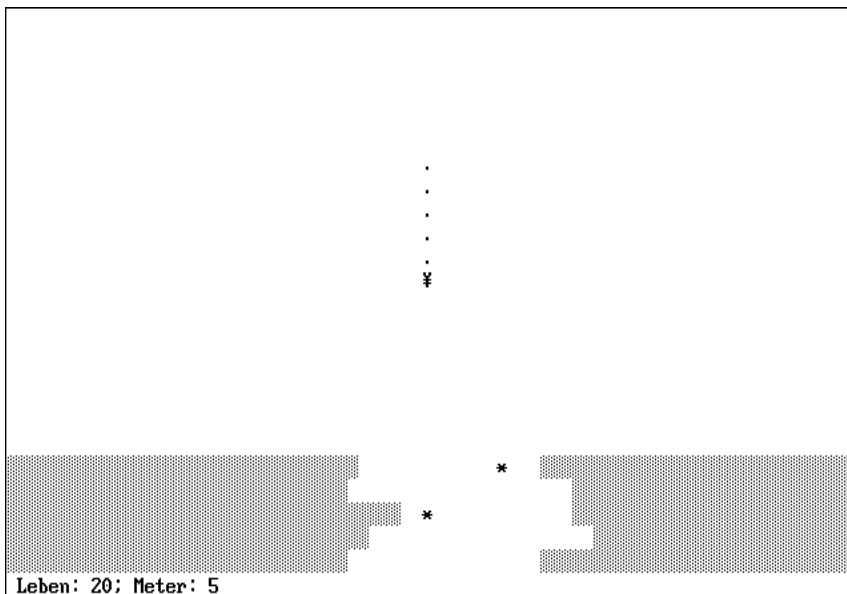
Möglicher Ablauf dieses Programms `autofahr.c`:

Es existieren 3 Schwierigkeitsgrade:

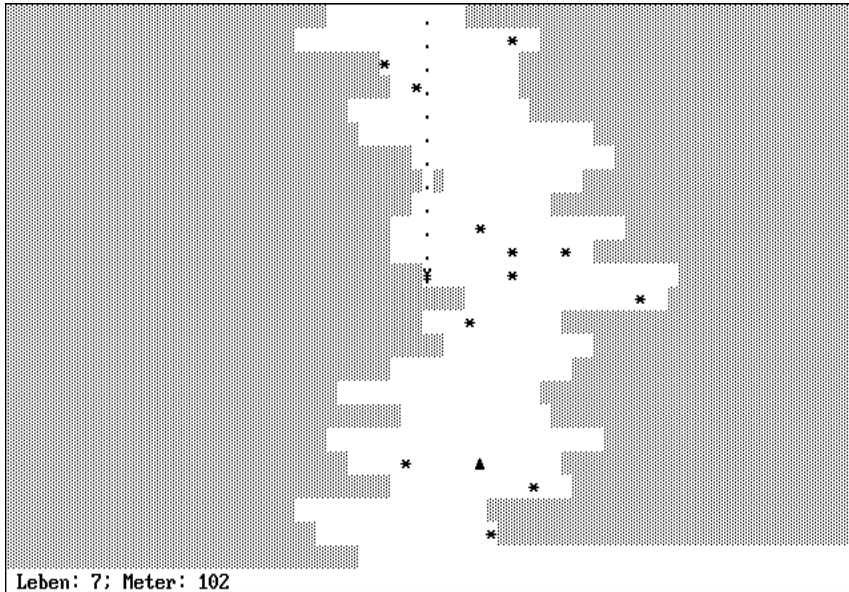
- 1 : Anfaenger
- 2 : Fortgeschrittener
- 3 : Profi

Was wählst Du ? 1

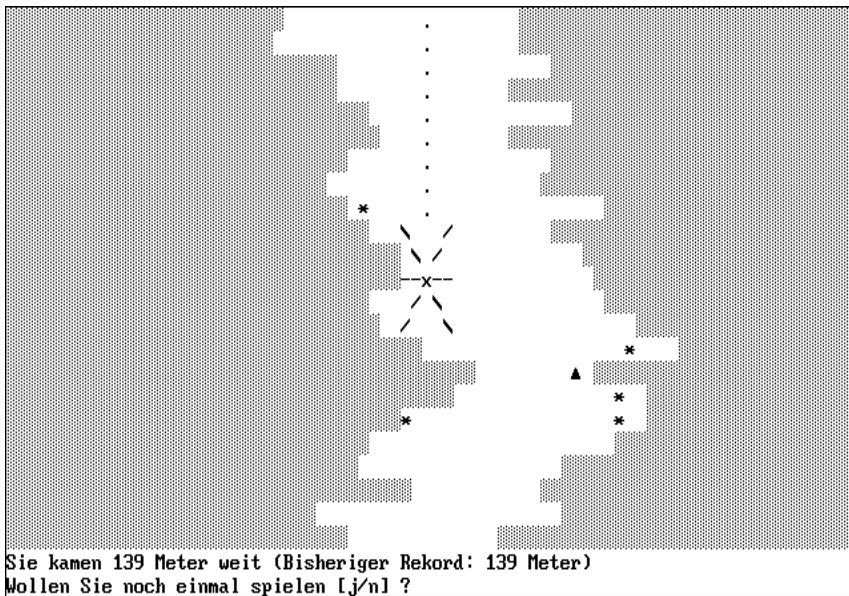
Ziemlich am Anfang der Autofahrt:



Mitten in der Autofahrt:



Nach Auftreffen auf ein Hindernis:



Hier Programmende nach Eingabe von **n**. Bei Eingabe von **j** würde eine neue Autofahrt gestartet.

20.13.7 Buchstaben-Memory

Erstellen Sie ein C-Programm `buchmemo.c`, das unter Verwendung von `conio.h` dem Benutzer für eine gewisse Zeit eine Zeichenkette (bestehend aus Kleinbuchstaben) in der Mitte des Bildschirms zeigt. Nach Ablauf einer Zeitspanne, welche abhängig von der Länge der Zeichenkette sein sollte, wird diese Zeichenkette wieder verdeckt.

Der Benutzer muß dann versuchen, sich der nun verdeckten Zeichenkette zu erinnern, und seine gemerkten Zeichen eingeben. Bei jeder Zeicheneingabe wird der direkt darüberliegende Buchstabe aufgedeckt, so daß der Benutzer immer sofort sehen kann, ob sein eingegebener Buchstabe richtig war oder nicht. Falsche Buchstaben sollten dabei beim Aufdecken invers dargestellt werden.

Wie lange die vom Programm zufällig ermittelte Zeichenkette sein soll, muß der Benutzer am Anfang des Programms eingeben.

Hinweis

1. Verdecken eines Zeichens läßt sich hier leicht dadurch erreichen, indem man den entsprechenden Buchstaben mit der Vordergrundfarbe `LIGHTGRAY` und der Hintergrundfarbe `WHITE` ausgibt.
2. Welches Zeichen zur Zeit an der Bildschirmposition `x,y` steht, kann mit **`gettext(x, y, x, y, puffer)`** erfragt werden. Nach diesem Aufruf steht das Zeichen von der Bildschirmposition `x,y` in `puffer[0]`; `puffer` ist in diesem Fall wie folgt zu deklarieren:

```
char puffer[2];
```

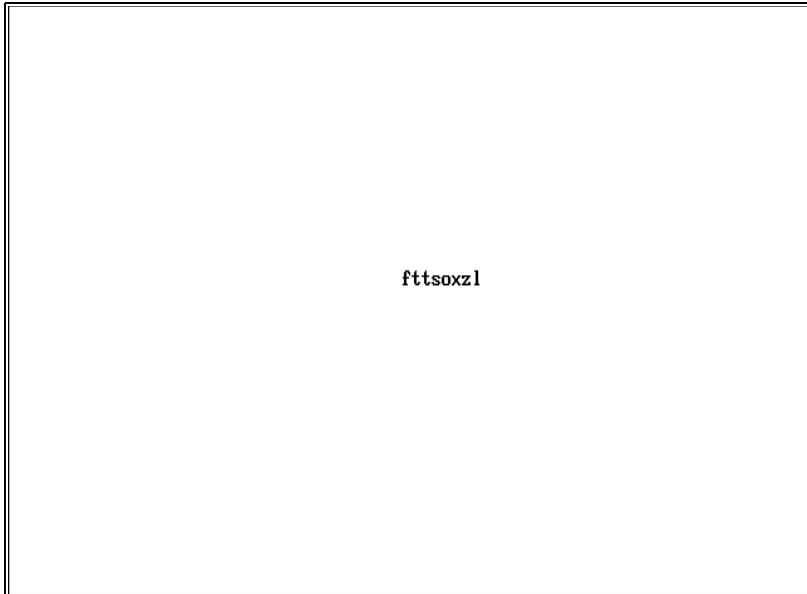
Möglicher Ablauf des Programms `buchmemo.c`:

```
Buchstaben-Memory
=====
```

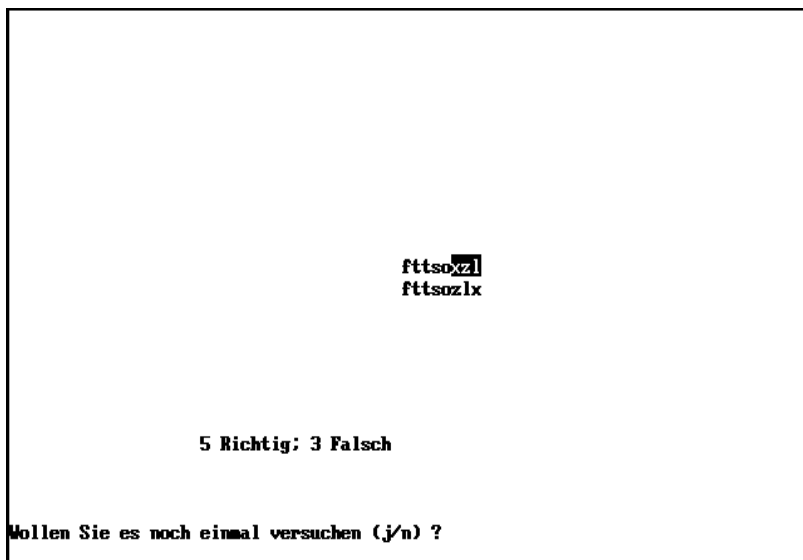
```
Dieses Programm zeigt Ihnen fuer eine gewisse Zeit eine
Zeichenkette. Nach Ablauf dieser Zeit verdeckt es diese
Zeichenkette wieder und Sie sollten sich dann erinnern,
welche Zeichenkette dies war und sie eingeben.
```

```
Wie viele Zeichen soll die Zeichenkette haben (1 bis 30): 8
```

2. Bild (Anzeigen der zufällig ermittelten Zeichenkette für eine gewisse Zeitspanne):



3. Bild (Nach dem Verdecken, was hier nicht gezeigt wurde, wird die Zeichenkette **fttsozlx** eingegeben):



Programmende durch Eingabe von **n**.

20.13.8 Erraten von AND, NAND, OR, NOR und XOR-Gatter

Diese Aufgabenstellung stammt aus einem Informatik-Wettbewerb:

Fritz findet auf dem Flohmarkt eine Kiste mit kleinen grauen Kästchen. Die Kästchen haben jeweils zwei gelbe Buchsen, beschriftet mit »Eingang1« und »Eingang2« und eine grüne Buchse »Ausgang«. Neben der Kiste liegen Schildchen, die sich von den grauen Kästchen gelöst haben. Die Schildchen tragen Aufschriften: AND-Gatter, NAND-Gatter, NOR-Gatter, OR-Gatter und XOR-Gatter. Fritz ist ein gewitzter Elektroniker: Er belegt die Eingänge jedes Kästchens mit verschiedenen Kombinationen von 0 Volt und 5 Volt Spannung und mißt die Ausgangsspannung. Schon nach kurzer Zeit hat er alle Schildchen wieder den richtigen Kästchen zugeordnet.

Erstellen Sie ein C-Programm `gatter.c`, das unter Verwendung von `conio.h` zufällig ein Gatter auswählt. Danach soll der Benutzer die Eingänge belegen, und ihm wird die Spannung am Ausgang angezeigt. Er soll dann raten, um welches Gatter es sich handelt. Der Benutzer kann dabei maximal viermal belegen und maximal dreimal raten.

Möglicher Ablauf dieses Programms `gatter.c`:

```
AND, NAND, OR, NOR oder XOR
=====
```

```
Eingang 1:  ---|  NAND  OR  |
              |  NOR   AND  | --- Ausgang:
Eingang 2:  ---|  ____XOR____  |
```

```
AND  0
NAND 1
OR   2
NOR  3
XOR  4
```

Willst Du (b)elegen oder raten (0,1,2,3,4)? b

Nächstes Bild (hier werden die Eingänge mit 0 und 5 belegt, bevor wieder **b** eingegeben wird):

```
AND, NAND, OR, NOR oder XOR
=====
```

1.Belegung: 0 ? 5 = 5

```
Eingang 1:  0---|  NAND  OR  |
              |  NOR   AND  | --- Ausgang: 5 Volt
Eingang 2:  5---|  ____XOR____  |
```

AND 0
NAND 1
OR 2
NOR 3
XOR 4

Willst Du (b)elegen oder raten (0,1,2,3,4)? b

Nächstes Bild (hier werden die Eingänge mit 5 und 5 belegt, bevor **4** (für XOR) eingegeben wird):

AND, NAND, OR, NOR oder XOR
=====

1.Belegung: 0 ? 5 = 5
2.Belegung: 5 ? 5 = 0

Eingang 1: 5---|

NAND	OR
NOR	AND

 |
Eingang 2: 5---|

XOR

 | --- Ausgang: 0 Volt

AND 0
NAND 1
OR 2
NOR 3
XOR 4

Willst Du (b)elegen oder raten (0,1,2,3,4)? 4

Nächstes Bild (neuer Rateversuch mit Eingabe von 1):

AND, NAND, OR, NOR oder XOR
=====

1.Belegung: 0 ? 5 = 5
2.Belegung: 5 ? 5 = 0

Eingang 1: 5---|

NAND	OR
NOR	AND

 |
Eingang 2: 5---|

XOR

 | --- Ausgang: 0 Volt

1.Rateversuch: XOR

AND 0
NAND 1
OR 2
NOR 3
XOR 4

Leider hast Du falsch geraten!

Willst Du (b)elegen oder raten (0,1,2,3,4)? 1

AND, NAND, OR, NOR oder XOR

=====

1.Belegung: 0 ? 5 = 5

2.Belegung: 5 ? 5 = 0

Eingang 1: 5---| NAND OR |
 | NOR AND |--- Ausgang: 0 Volt
Eingang 2: 5---|___XOR___|

1.Rateversuch: XOR

2.Rateversuch: NAND

AND 0

NAND 1

OR 2

NOR 3

XOR 4

Super - das war richtig!!

Das Gatter war ein NAND

Willst Du ein neues Gatter raten (j/n) ? n

21 Weitere nützliche Funktionen und Techniken

Sicher ist, daß nichts sicher ist. Selbst das nicht.

Ringelnetz

Hier werden weitere Funktionen vorgestellt, die sehr wertvolle Dienste bei der Systemprogrammierung leisten können. Zunächst werden Funktionen zur Dateinamenexpansion vorgestellt, bevor in einem weiteren Kapitel Funktionen beschrieben werden, die zum Arbeiten mit regulären Ausdrücken innerhalb von Programmen benötigt werden. Das folgende Kapitel stellt dann Funktionen und Techniken vor, mit denen man Optionen auf der Kommandozeile abarbeiten kann.

21.1 Expandierung von Dateinamen

Bei Programmen, die viel mit Dateien arbeiten, wird sehr oft die Dateinamenexpansion benötigt, wie z.B. **ls *.txt** zum Auflisten aller Dateien, die mit **.txt** enden. Hier werden zwei übliche Techniken vorgestellt, um Dateinamenexpansion innerhalb eines Programms durchzuführen.

21.1.1 Dateinamenexpansion mit der Funktion **popen**

Eine schon früher unter Unix übliche Methode zum Expandieren von Dateinamen innerhalb eines Programms, ist der Aufruf der Funktion **popen** mit dem Kommando, das die entsprechenden Expansionszeichen (*****, **?**, **[]** usw.) der Shell enthält. Da **popen** eine Shell als Kindprozeß startet, übernimmt diese Shell die Dateinamenexpansion. Über den von **popen** gelieferten **FILE**-Zeiger kann dann die Ausgabe des aufgerufenen Kommandos in das Programm eingelesen werden.

Beispiel

Demonstrationsprogramm zur Dateinamenexpansion mit popen

Das Programm 21.1 (**popenexp.c**) demonstriert die Anwendung dieser Technik, indem es die Zeilen aller im Working-Directory enthaltenen C-Dateien (Endung **.c**) zählt und ausgibt.

```
#include <sys/wait.h>
#include "eighdr.h"

int
main(void)
{
```

```

char  dateiname[MAX_ZEICHEN],
      zeile[MAX_ZEICHEN];
FILE  *cprogs, *datei;
int   z, total=0;

if ( (cprogs = popen("ls *.c", "r")) == NULL)
    fehler_meld(FATAL_SYS, "Fehler bei popen");

while (fgets(dateiname, sizeof(dateiname), cprogs) != NULL) {
    dateiname[strlen(dateiname)-1] = 0;
    if ( (datei = fopen(dateiname, "r")) == NULL)
        fehler_meld(WARNUNG_SYS, "kann Datei '%s' nicht zum Lesen oeffnen",
                     dateiname);
    else {
        z = 0;
        while (fgets(zeile, sizeof(zeile), datei))
            z++;
        total += z;
        fprintf(stdout, "%30s : %5d\n", dateiname, z);
        fclose(datei);
    }
}
fprintf(stdout, "-----\n"
          "%30s : %5d\n", "Gesamt", total);

if (!WIFEXITED(pclose(cprogs)))
    fehler_meld(FATAL_SYS, "Fehler bei pclose");

return(0);
}

```

Programm 21.1 (popenexp.c): Demonstrationsprogramm zur Dateinamenexpandierung mit **popen**

Nachdem man das Programm 21.1 (popenexp.c) kompiliert und gelinkt hat

```
cc -o popenexp popenexp.c fehler.c
```

ergibt sich beim Start z.B. der folgende Ablauf:

```

$ popenexp
          fehler.c :   103
        globdem2.c :    16
        globdemo.c :    46
        popenexp.c :    37
-----
          Gesamt :   202
$

```

21.1.2 Dateinamenexpandierung mit der Funktion **glob**

Muß man viele Dateinamen expandieren, ist der Aufruf von **popen** nicht sehr effizient, da in diesem Fall jedesmal ein Kindprozeß gestartet wird. Mit der Funktion **glob** kann man Dateinamen expandieren, ohne daß dazu der Start eines Kindprozesses notwendig ist. Allerdings ist der Aufruf von **glob** etwas komplexer als der von **popen**. Zudem kann es Unix-Varianten geben, die die nachfolgend beschriebene Funktion **glob**, die zwar von POSIX.2 vorgeschrieben ist, nicht zur Verfügung stellen.

```
#include <glob.h>

int glob(const char *pattern, int flags,
         int errfunc(const char *epath, int errno),
         glob_t *pglob);

                                gibt zurück: 0 (bei Erfolg);
                                GLOB_NOSPACE (bei Speicherplatzmangel)
                                GLOB_ABEND  (bei einem Lesefehler)
                                GLOB_NOMATCH (bei keiner Übereinstimmung)

void globfree(glob_t *pglob);
```

Der erste Parameter *pattern* ist eine Mustervorgabe, nach der die Dateinamenexpandierung durchzuführen ist. Als Expandierungszeichen sind dabei die Metazeichen der jeweiligen Shell zugelassen: *****, **?**, **[]** usw.

Das Ergebnis der durchgeführten Dateinamenexpandierung wird in die Variable *pglob*, deren Adresse als letztes Argument beim Aufruf anzugeben ist, geschrieben. Der Datentyp dieser Variablen ist eine Struktur, die wie folgt in *<glob.h>* definiert ist:

```
typedef struct
{
    int  gl_pathc;    /* Anzahl der Pfadnamen in gl_pathv */
    char **gl_pathv; /* Liste von passenden Pfadnamen */
    int  gl_offs;     /* Anzahl der freizulassenden Einträge
                       in gl_pathv (für Flag GLOB_DOOFFS) */
    int  gl_flags;    /* Flags für die Dateinamenexpandierung */
} glob_t;
```

Der Parameter *flags* bei der Funktion **glob** legt das Verhalten dieser Funktion fest. Hierfür kann der Wert 0 bzw. eine oder mehrere der folgenden Konstanten angegeben werden. Sind mehrere Konstanten angegeben, sind diese mit **|** (bitweises OR) zu verknüpfen.

GLOB_ERR

legt fest, daß bei einem Lesefehler (z.B. weil ein Directory keine Leserechte besitzt) nach einem eventuellen Aufruf der Funktion **errfunc** die Funktion **glob** zu beenden ist und zum Aufrufer zurückzukehren ist.

GLOB_MARK

legt fest, daß an alle gefundenen Directory-Namen ein Slash / anzuhängen ist.

GLOB_NOSORT

legt fest, daß die gefundenen Namen nicht zu sortieren sind, was normalerweise der Fall ist.

GLOB_DOOFFS

legt fest, daß die ersten Elemente von `gl_pathv` frei bleiben sollen. Wie viele Elemente frei bleiben sollen, ist über die Komponente `gl_offs` der übergebenen Strukturvariablen `pglob` festzulegen. Diese Konstante ermöglicht es, weitere eigene Argumente am Anfang der Liste `gl_pathv` einzusetzen, wenn man dieses Stringarray direkt an einen **execv**-Aufruf in einem Programm übergeben möchte.

GLOB_NOCHECK

legt fest, daß **glob** bei keiner Übereinstimmung wenigstens das übergebene `pattern` zurückliefert. Normalerweise wird in diesem Fall nichts zurückgegeben, außer das Muster enthält überhaupt keine Expandierungs-Metazeichen.

GLOB_APPEND

legt fest, daß gefundene Dateinamen an eine bereits existierende Liste von Dateinamen, die durch vorausgehende **glob**-Aufruf gefunden wurde, angehängt wird. Dies ermöglicht die Auswertung von mehreren Mustern.

GLOB_NOESCAPE

legt fest, daß das Quoting von Expandierungs-Metazeichen mit Backslash \ ausgeschaltet ist und der Backslash seine Sonderbedeutung für das Ausschalten von Metazeichen verliert und als normales Zeichen behandelt wird.

GLOB_PERIOD

legt fest, daß auch Punkte am Anfang von Dateinamen durch das angegebene `pattern` abgedeckt werden, was normalerweise nicht der Fall ist.

Wenn ein Fehler bei der Ausführung der Funktion **glob** auftritt, wie z.B. bei fehlenden Zugriffsrechten auf die entsprechenden Directories, wird die vom Benutzer als drittes Argument beim **glob**-Aufruf angegebene Funktion **errfunc** aufgerufen.

Dieser Funktion **errfunc**, die die folgende Prototypdeklaration besitzt:

```
int errfunc(const char *epath, int eerrno);
```

wird der Pfadname (`epath`), bei dem der Fehler auftrat, sowie die Fehlernummer (`eerrno`), was der Wert der globalen Variablen `errno` ist, übergeben. Der Wert von `errno` wurde durch den fehlgeschlagenen Aufruf einer der Funktionen **opendir**, **readdir** oder **stat** gesetzt.

Falls die Funktion **errfunc** einen Wert verschieden von 0 zurückgibt oder falls die Konstante `GLOB_ERR` im Argument `flags` gesetzt ist, beendet sich **glob** nach dem Aufruf von **errfunc**, ansonsten wird mit der Dateinamenexpandierung fortgefahren.

Das Ergebnis eines **glob**-Aufrufs wird in der übergebenen Strukturvariablen `pglob` hinterlegt. Der Datentyp dieser Variablen ist die Struktur `glob_t`, die unter anderem die beiden folgenden Komponenten enthält:

`gl_pathc`

enthält die Anzahl der Pfadnamen, die zum übergebenen Muster passen.

`gl_pathv`

Stringarray, das die Pfadnamen enthält, die zum übergebenen Muster passen. Das Ende dieser Liste wird durch einen `NULL`-Zeiger angezeigt.

Wird **glob** mehrmals aufgerufen, sollte das Flag `GLOB_APPEND` nach dem ersten Aufruf gesetzt werden, damit nachfolgende Aufrufe ihre Ergebnisse an das Ende von `gl_pathv` anhängen.

glob allokiert dynamisch die Datenstrukturen, in denen es seine Resultate speichert.

Wird die zurückgegebene Struktur `glob_t` nicht mehr benötigt, sollte der von ihr belegte Speicher über die Funktion **globfree** wieder freigegeben werden.

Beispiel

Demonstrationsprogramm zur Funktion **glob**

Das Programm 21.2 (`globdemo.c`) demonstriert die Verwendung der Funktion **glob**. Es liest die auszuwertenden `pattern` als Argumente auf der Kommandozeile ein, ruft dann für jedes angegebene Muster die Funktion **glob** zur Dateinamenexpandierung auf und gibt am Ende das Ergebnis der Dateinamenexpandierung aus.

```
#include <errno.h>
#include <glob.h>
#include "eighdr.h"

int fehlroutine(const char *pathname, int fehler)
{
    fehler_meld(WARNUNG_SYS, "Fehler beim Zugriff auf %s", pathname);

    return 0; /* durch Rueckgabe 0 wird angezeigt, daß glob
               seine Ausführung fortsetzen soll */
}

int
main(int argc, char *argv[])
{
    glob_t  ergeb;
    int     i, r, flags;

    if (argc < 2)
        fehler_meld(FATAL, "usage: %s 'pattern1' 'pattern2' ....", argv[0]);

    flags = 0; /* flags zunächst auf 0, später auf GLOB_APPEND setzen */
```

```

/* alle Kommandozeilenargumente durchlaufen */
for (i=1; i < argc; i++) {
    r = glob(argv[i], flags, fehlroutine, &ergeb);

    if (r == GLOB_NOSPACE) /* GLOB_ABEND wegen fehlroutine nicht moegl. */
        fehler_meld(FATAL_SYS, "Speicherplatzmangel");

    flags |= GLOB_APPEND;
}

if (ergeb.gl_pathc == 0) {
    fehler_meld(WARNUNG, "keine Übereinstimmungen gefunden");
    r = 1;
} else {
    for (i=0; i < ergeb.gl_pathc; i++)
        fprintf(stdout, " %s\n", ergeb.gl_pathv[i]);
    r = 0;
}

globfree(&ergeb);

return r;
}

```

Programm 21.2 (globdemo.c): Demonstrationsprogramm zur Funktion **glob**

Nachdem man dieses Programm 21.2 (globdemo.c) kompiliert und gelinkt hat

```
cc -o globdemo globdemo.c fehler.c
```

ergeben sich beim Start z.B. die folgende Abläufe:

```

$ globdemo '*.c' '/usr/include/std*.h'
fehler.c
globdemo.c
popenexp.c
/usr/include/stdarg.h
/usr/include/stdio.h
/usr/include/stdlib.h

$ globdemo '??' '*.txt'
keine Übereinstimmungen gefunden

$

```

Beispiel

Weiteres Demonstrationsbeispiel zu **glob**

Das Programm 21.3 (globdem2.c) simuliert den Kommandoaufruf

```
ls -l *.c /usr/include/??h
```



```
#include <errno.h>
#include <glob.h>
#include "eighdr.h"

int
main(void)
{
    glob_t globpuf;

    globpuf.gl_offs = 2;
    glob("*.c", GLOB_DOOFFS|GLOB_NOSORT, NULL, &globpuf);
    glob("/usr/include/??h", GLOB_APPEND, NULL, &globpuf);
    globpuf.gl_pathv[0] = "ls";
    globpuf.gl_pathv[1] = "-l";
    execvp("ls", globpuf.gl_pathv);
}
```

Programm 21.3 (**globdem2.c**): Simulation des Kommandoaufrufs **ls -l *.c /usr/include/??h**

Nachdem man dieses Programm 21.3 (**globdem2.c**) kompiliert und gelinkt hat

```
cc -o globdem2 globdem2.c
```

ergibt sich beim Start z.B. der folgende Ablauf:

```
$ globdem2
-rw-r--r--  1 root    root          292 Feb 18  1995 /usr/include/ar.h
-rw-r--r--  1 root    root          8213 Mar  4  1998 /usr/include/db.h
-rw-r--r--  1 root    root       116946 Oct  5  1996 /usr/include/rx.h
-rw-r--r--  1 root    root        51143 Nov  6 16:12 /usr/include/tk.h
-rw-r--r--  1 hh      users        2783 Dec 14 12:03 fehler.c
-rw-r--r--  1 hh      users         344 Dec 14 19:31 globdem2.c
-rw-r--r--  1 hh      users        1175 Dec 14 18:57 globdemo.c
-rw-r--r--  1 hh      users         989 Dec 14 12:25 popenexp.c
$
```

21.2 String-Vergleiche mit regulären Ausdrücken

Neben den ANSI-C-Funktionen **strcmp** und **strncmp** zum Vergleichen von Strings werden unter Unix/Linux meist noch mächtigere Funktionen angeboten, die String-Vergleiche mit Hilfe von regulären Ausdrücken ermöglichen. Dieses Kapitel stellt solche Funktionen vor.

21.2.1 String-Vergleiche mit den Metazeichen der Dateinamenexpandierung

Im vorherigen Kapitel wurden die beiden Funktionen **popen** und **pglob** vorgestellt, mit denen man das Expandieren von Dateinamen innerhalb eines Programms durchführen kann. Für Vergleiche von Strings existiert eine ähnliche Funktion **fnmatch**, die ebenso wie

diese Funktionen die entsprechenden Expandierungszeichen (*, ?, [] usw.) der Shell anbietet, jedoch diese Expandierungszeichen nicht auf Dateinamen, sondern auf normale Strings anwenden läßt.

#include <fnmatch>

int **fnmatch**(const char **pattern*, const char **string*, int *flags*);

gibt zurück:0 (wenn *pattern* den String *strings* abdeckt);

FNM_NOMATCH, wenn keine Abdeckung vorliegt;

anderen Wert bei Fehler

In *pattern* können neben normalen ASCII-Zeichen auch die Expandierungszeichen (*, ?, [] usw.) der Shell angegeben werden. Der übergebende *string* wird darauf hin untersucht, ob er durch das in *pattern* angegebene Muster abgedeckt wird.

Der Parameter *flags* legt fest, wie die Expandierung durchzuführen ist. Hierfür kann eine oder mehrere der folgenden Konstanten (mit bitweisem OR | verknüpft) angegeben werden:

FNM_NOESCAPE	<i>Backslash</i> (\) wird als normales Zeichen interpretiert, so daß es seine Sonderbe- deutung (Ausschalten eines folgenden Expandierungszeichens) verliert.
FNM_PATHNAME	Der <i>Slash</i> (/) wird in <i>string</i> nicht durch ein in <i>pattern</i> angegebenes Expan- dierungszeichen abgedeckt.
FNM_PERIOD	Ein führender Punkt in <i>pattern</i> deckt nur dann in <i>string</i> einen Punkt ab, wenn dies dort das erste Zeichen ist oder aber wenn FNM_PATHNAME gesetzt ist und der Punkt in <i>string</i> direkt nach einem <i>Slash</i> (/) steht.

Meist wird jedoch für *flags* nur der Wert 0 angegeben, da diese Konstanten hauptsäch-
lich auf die Dateinamenexpandierung ausgelegt sind.

Beispiel

Suchen von Wörtern nach einem vorgegebenem Muster

Das Programm 21.4 (fnmatch.c) demonstriert die Anwendung der Funktion *fnmatch*. Es sucht in den auf der Kommandozeile angegebenen Dateien nach Wörtern, die durch das erste Argument abgedeckt werden. Dieses erste *pattern*-Argument darf die Expandie-
rungszeichen (*, ?, [] usw.) der Shell enthalten.

1 #include <stdio.h>

2 #include <fnmatch.h>

3 #include "eighdr.h"

4

5 int

6 main(int argc, char *argv[])

7 {

```

8   FILE   *dz;
9   char   zeile[MAX_ZEICHEN];
10  char   *wort,
11         *trennzeich = "@<>|_-^.:,;#'+*~'?=()[]/%%$\\\"!  \n";
12  int     i, zeilnr, erst;
13
14  if (argc < 3)
15      fehler_meld(FATAL, "usage: %s pattern datei(en)", argv[0]);
16
17  for (i=2; i<argc; i++) {
18      zeilnr = 1;
19      if ( (dz = fopen(argv[i], "r")) == NULL) {
20          fehler_meld(WARNUNG_SYS, "kann '%s' nicht oeffnen", argv[i]);
21          continue;
22      }
23
24      if (argc > 3)
25          printf(".....%s\n", argv[i]);
26
27      while (fgets(zeile, MAX_ZEICHEN, dz) != NULL) {
28          erst = 1;
29          wort = strtok(zeile, trennzeich);
30          while (wort != NULL) {
31              if (fnmatch(argv[1], wort, FNM_NOESCAPE) == 0)
32                  if (erst == 1) {
33                      printf("%d: %s", zeilnr, wort);
34                      erst = 0;
35                  } else
36                      printf(", %s", wort);
37              wort = strtok(NULL, trennzeich);
38          }
39          if (erst == 0)
40              printf("\n");
41          zeilnr++;
42      }
43
44      fclose(dz);
45  }
46  exit(0);
47  }

```

Programm 21.4 (fnmatch.c): Suchen von Wörtern nach einem vorgegebenem Muster

Nachdem man dieses Programm 21.4 (fnmatch.c) kompiliert und gelinkt hat

```
cc -o fnmatch fnmatch.c fehler.c
```

kann man es starten, wie z.B.:

```

$ fnmatch "[pws][m-z]*" fnmatch.c
1: stdio
10: wort
25: printf
29: wort, strtok

```

```

30: wort
31: wort
33: printf, wort
36: printf, wort
37: wort, strtok
40: printf
$

```

21.2.2 String-Vergleiche mit regulären Ausdrücken

Man unterscheidet unter Unix/Linux zwei Arten von regulären Ausdrücken:

- grundlegende reguläre Ausdrücke (*basic regular expressions*), die weitgehend den beim Kommando **grep** erlaubten regulären Ausdrücken entsprechen
- erweiterte reguläre Ausdrücke (*extended regular expressions*), die weitgehend den beim Kommando **egrep** erlaubten regulären Ausdrücken entsprechen

Nähere Informationen dazu lassen sich mit **man grep** erfragen. Detaillierter auf reguläre Ausdrücke wird in den drei ersten Bänden dieser Reihe »Linux-Unix-Grundlagen«, »Linux-Unix-Profitools« und »Linux-Unix-Shells« eingegangen.

Um mit regulären Ausdrücken in Programmen arbeiten zu können, werden entsprechend dem POSIX-Standard vier Funktionen zur Verfügung gestellt.

```

#include <regex.h>

int regcomp(regex_t *preg, const char *regex, int cflags);
                                     gibt zurück: 0 (bei Erfolg); Fehlernummer bei Fehler

int regexexec(const regex_t *preg, const char *string,
               size_t nmatch, regmatch_t pmatch[], int eflags);
                                     gibt zurück: 0 (bei Erfolg); REG_NOMATCH bei Fehler

size_t regerror(int errcode, const regex_t *preg,
                 char *errbuf, size_t errbuf_size);
                                     gibt zurück: Anzahl der nach errbuf geschriebenen Zeichen

void regfree(regex_t *preg);

```

Die einzelnen Funktionen werden nachfolgend erläutert.

Die Funktion **regcomp**

Die Funktion **regcomp** wird verwendet, um einen regulären Ausdruck in eine Form (regex_t) zu transformieren (kompilieren), wie sie für die Aufrufe der Funktionen **regexexec** und **regerror** benötigt wird. Dazu muß dieser Funktion die Adresse (preg) einer Variable vom Datentyp regex_t übergeben werden, damit sie den im zweiten Argument regex angegebenen regulären Ausdruck transformieren und dorthin speichern kann.

Das letzte Argument `cflags` legt die Art der Transformation fest. Für `cflags` kann eine oder auch mehrere (mit bitweisem OR | verknüpft) der folgenden Konstanten angegeben werden:

`REG_EXTENDED`

Anstelle der grundlegenden regulären Ausdrücke werden die erweiterten regulären Ausdrücke verwendet.

`REG_ICASE`

Es wird nicht zwischen Groß- und Kleinschreibung unterschieden.

`REG_NOSUB`

Eine Abdeckung von Teilstrings ist nicht gefordert. Dies bewirkt, daß die beiden Parameter `nmatch` und `pmatch` bei der Funktion ***regex*** ignoriert werden.

`REG_NEWLINE`

Wenn `REG_NEWLINE` nicht gesetzt ist, dann wird das Neue-Zeile-Zeichen wie jedes andere Zeichen auch behandelt. `^` und `$` erkennen dann nur den Anfang und das Ende eines Strings, in dem eventuell dazwischen Neue-Zeile-Zeichen eingebettet sein können. Setzt man `REG_NEWLINE`, dann beziehen sich die regulären Ausdrücke immer nur auf eine Zeile, wie dies auch bei Programmen wie ***grep*** oder ***sed*** der Fall ist.

Beispiel

Finden aller Leerzeilen in Dateien

Das folgende Programm 21.5 (`leerzeil.c`) findet in allen auf der Kommandozeile angegebenen Dateien die Leerzeilen und gibt deren Zeilennummern aus. Als Leerzeilen werden dabei leere Zeilen oder Zeilen, die nur Leer- und Tabzeichen enthalten, interpretiert.

```
1  #include <regex.h>
2  #include "eighdr.h"
3
4  int
5  main(int argc, char *argv[])
6  {
7      FILE      *dz;
8      char      zeile[MAX_ZEICHEN];
9      int       i, zeilnr;
10     regex_t    preg;
11
12     if (argc < 2)
13         fehler_meld(FATAL, "usage: %s datei(en)", argv[0]);
14
15     for (i=1; i<argc; i++) {
16         zeilnr = 1;
17         if ( (dz = fopen(argv[i], "r")) == NULL) {
18             fehler_meld(WARNUNG_SYS, "kann '%s' nicht oeffnen", argv[i]);
19             continue;
20         }
21
22         if (argc > 2)
```

```

23     printf(".....%s\n", argv[i]);
24
25     while (fgets(zeile, MAX_ZEICHEN, dz) != NULL) {
26         zeile[strlen(zeile)-1] = 0; /* Neuezeilezeichen entfernen */
27         if (regcomp(&preg, "^[ \\t]*$", REG_EXTENDED|REG_NEWLINE) == 0)
28             if (regexexec(&preg, zeile, 0, NULL, 0) == 0)
29                 printf("%d\n", zeilnr);
30         zeilnr++;
31     }
32
33     fclose(dz);
34 }
35 exit(0);
36 }

```

Programm 21.5 (leerzeil.c): Finden aller Leerzeilen in Dateien

Nachdem man dieses Programm 21.5 (leerzeil.c) kompiliert und gelinkt hat

```
cc -o leerzeil leerzeil.c fehler.c
```

kann man es starten, wie z.B.:

```

$ leerzeil leerzeil.c fnmatch.c
.....leerzeil.c
3
11
14
21
24
32
.....fnmatch.c
4
13
16
23
26
43
$

```

Die Funktion *regexexec*

Die Funktion **regexexec**, die wie folgt in `<regex.h>` deklariert ist

```
int regexexec(const regex_t *preg, const char *string,
              size_t nmatch, regmatch_t pmatch[], int eflags);
```

wird verwendet, um zu prüfen, ob der zuvor mit **regcomp** transformierte reguläre Ausdruck (preg) einen Teilstring in String (string) abdeckt.

Die beiden Parameter `nmatch` und `pmatch` liefern Informationen über Stellen, an denen sich abgedeckte Teilstrings in `string` befinden. Dies gilt jedoch nur, wenn bei der Transformation des regulären Ausdrucks mit **regcomp** nicht das Flag `REG_NOSUB` gesetzt war. Das übergebene Array zu `pmatch` muß dabei mindestens eine Dimension von `nmatch` haben. Die einzelnen Elemente dieses Arrays werden dann von **regex** entsprechend auf die Offsets der einzelnen abgedeckten Teilstrings in `string` gesetzt.

Die Struktur `regmatch_t` ist in `<regex.h>` wie folgt definiert.

```
typedef struct {
    regoff_t rm_so;
    regoff_t rm_eo;
} regmatch_t;
```

Die Komponente `rm_so` gibt das Startoffset und `rm_eo` gibt das Endoffset eines abgedeckten Teilstrings an. Ungenutzte Einträge im Array `pmatch` werden durch den Wert -1 in den Komponenten `rm_so` und `rm_eo` angezeigt. Wie viele Gruppen von Strings maximal abgedeckt werden können, kann durch Zugriff auf die Komponente `re_nsub` in der `regex_t`-Variable `preg` in Erfahrung gebracht werden, wie dies in Programm 21.6 (`regex.c`) gezeigt ist.

Für `eflags` kann eine oder auch mehrere (mit bitweisem OR | verknüpft) der folgenden Konstanten angegeben werden:

`REG_NOTBOL`

schaltet die Sonderbedeutung des Metazeichens `^` (Anfang einer Zeile) aus.

`REG_NOTEOL`

schaltet die Sonderbedeutung des Metazeichens `$` (Ende einer Zeile) aus.

Die Funktion **regfree**

Um nicht mehr benötigte kompilierte reguläre Ausdrücke wieder freizugeben, steht die Funktion **regfree** zur Verfügung.

```
void regfree(regex_t *preg);
```

POSIX.2 legt nicht fest, ob eine `regex_t`-Variable `preg` mehrmals an **regcomp** übergeben werden kann, ohne daß man diese zuvor freigibt. Programmierer sollten sich deshalb nicht darauf verlassen, sondern jedesmal zuvor **regfree** aufrufen.

Die Funktion **regerror**

Immer wenn die Funktionen **regcomp** oder **regex** einen Wert ungleich 0 zurückgeben, kann man sich mit **regerror** die dazugehörige Fehlermeldung ausgeben lassen.

```
size_t regerror(int errcode, const regex_t *preg,
                char *errbuf, size_t errbuf_size);
```

Übergibt man für `errbuf` den `NULL`-Zeiger und/oder für `errbuf_size` den Wert 0, so liefert **regerror** die Anzahl der Zeichen, die die zu `errcode` gehörende Fehlermeldung umfaßt. So kann man im voraus die Länge der Fehlermeldung erfragen, um einen Puffer der entsprechenden Größe zu allokalieren; siehe dazu auch das Programm 21.6 (`regex.c`).

Beispiel

Demonstrationsprogramm zu den hier vorgestellten Funktionen

Das folgende Programm 21.6 (`regex.c`) ist ein Demonstrationsbeispiel zu den hier vorgestellten Funktionen.

```
#include <regex.h>
#include "eighdr.h"

#define MAX_TEILSTRING 100

/*----- substr -----*/
void
print_substr(char *string, int start, int ende)
{
    int i;

    for (i=start; i<ende; i++)
        printf("%c", string[i]);
}

/*----- reg_fehler -----*/
void
reg_fehler(char *paeafix, const regex_t *preg, int fehlernr)
{
    char *puffer;
    size_t groesse = regerror(fehlernr, preg, NULL, 0);

    if ( (puffer = malloc(groesse)) == NULL)
        fehler_meld(FATAL_SYS, "Speicherplatzmangel");
    regerror(fehlernr, preg, puffer, groesse);
    fprintf(stderr, "%s%s\n", paeafix, puffer);
    free(puffer);
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    FILE *dz;
    char zeile[MAX_ZEICHEN];
    int i, j, zeilnr, fehlernr;
    regex_t preg;
    regmatch_t pmatch[MAX_TEILSTRING];

    if (argc < 2)
```



```

fehler_meld(FATAL, "usage: %s datei(en)", argv[0]);

for (i=1; i<argc; i++) {
    zeilnr = 1;
    if ( (dz = fopen(argv[i], "r")) == NULL) {
        fehler_meld(WARNUNG_SYS, "kann '%s' nicht oeffnen", argv[i]);
        continue;
    }

    if (argc > 2)
        printf(".....%s\n", argv[i]);

    if ( (fehlernr = regcomp(&preg,
        "([a-zA-Z][a-zA-Z_0-9]*[().*[]]|"
        "^(.*);.*$)|(^[:;]+$)",
        REG_EXTENDED)) != 0)
        reg_fehler("regcomp: ", &preg, fehlernr);
    else {
        while (fgets(zeile, MAX_ZEICHEN, dz) != NULL) {
            char puffer[80];
            zeile[strlen(zeile)-1] = 0;
            sprintf(puffer, "....."
                "Zeile %d.....\n", zeilnr);
            if ( (fehlernr = regexec(&preg, zeile,
                preg.re_nsub, pmatch, 0)) != 0)
                reg_fehler(puffer, &preg, fehlernr);
            else {
                printf("%s", puffer);
                for (j = 0; j < preg.re_nsub; j++) {
                    if (pmatch[j].rm_so != -1) {
                        print_substr(zeile,
                            pmatch[j].rm_so, pmatch[j].rm_eo);
                        printf(" (%d)\n", j);
                    }
                }
                zeilnr++;
            }
        }
        regfree(&preg);
        fclose(dz);
    }
    exit(0);
}

```

Programm 21.6 (regexec.c): Demonstrationsprogramm zu den hier vorgestellten Funktionen

Nachdem man dieses Programm 21.6 (regexec.c) kompiliert und gelinkt hat

```
cc -o regexec regexec.c fehler.c
```

kann man es starten. Hier soll dazu die folgende Datei regtest.txt verwendet werden.

```

1 #include <regex.h>
2 #include "eighdr.h"
3
4 #define MAX_TEILSTRING 100
5
6 /*----- substr -----*/
7 void
8 print_substr(char *string, int start, int ende)
9 {
10     int i;
11
12     for (i=start; i<ende; i++)
13         printf("%c", string[i]);
14 }

```

Nachfolgend ein Ablaufbeispiel zum Programm 21.6 (regexec.c):

```

$ regexec regtest.txt
.....Zeile 1.....
#include <regex.h> (0)
.....Zeile 2.....
#include "eighdr.h" (0)
.....Zeile 4.....
#define MAX_TEILSTRING 100 (0)
.....Zeile 6.....
/*----- substr -----*/ (0)
.....Zeile 7.....
void (0)
.....Zeile 8.....
print_substr(char *string, int start, int ende) (0)
print_substr(char *string, int start, int ende) (1)
.....Zeile 9.....
{ (0)
.....Zeile 10.....
    int i; (0)
    int i; (2)
    int i (3)
.....Zeile 12.....
    for (i=start; i<ende; i++) (0)
    for (i=start; i<ende; i++) (2)
    for (i=start; i<ende (3)
.....Zeile 13.....
        printf("%c", string[i]); (0)
        printf("%c", string[i]); (2)
        printf("%c", string[i]) (3)
.....Zeile 14.....
} (0)

$

```

21.3 Abarbeiten von Optionen auf der Kommandozeile

Neben normalen Argumenten auf einer Kommandozeile, wie z.B. Dateinamen oder Strings, sind dort auch Optionen erlaubt. Während früher nur Einzeichen-Optionen (*short options*) mit einem vorangestellten Querstrich, wie z.B. **-a** oder **-l**, üblich waren, sind heute auch Strings als Optionen (*long options*) mit zwei vorangestellten Minuszeichen, wie z.B. **--all** oder **--format=long**, erlaubt.

Auf jede Option kann dabei eventuell auch ein Argument folgen, wobei üblicherweise Argumente von kurzen Optionen durch ein Leerzeichen und von langen Optionen entweder durch ein Leerzeichen oder ein Gleichheitszeichen (=) getrennt werden.

Zur Abarbeitung von Kommandozeilenoptionen gibt es viele Techniken, von denen die gebräuchlichsten in diesem Kapitel vorgestellt werden.

21.3.1 Die traditionelle Technik

Diese ursprüngliche Technik zur Abarbeitung von kurzen Optionen, nämlich die eigene Auswertung der Argumente im Stringarray `argv`, wird auch heute noch am häufigsten angewendet.

Diese Methode, die zunächst nur auf kurze Optionen ausgelegt ist, erlaubt es, daß Optionen einzeln oder gruppiert (wie z.B. **-w -l** oder **-w -lc** oder **-lwc**) angegeben werden können oder daß sie sogar zwischen anderen Argumenten stehen dürfen, was jedoch nicht empfehlenswert ist.

Das nachfolgende Programmbeispiel verdeutlicht diese traditionelle Technik, wobei es zusätzlich jedoch auch lange Optionen zuläßt.

Beispiel

Traditionelle Abarbeitung von Optionen (einfaches wc-Programm)

Das folgende Programm 21.7 (`wc2.c`) ist eine einfache Realisierung des Kommandos **wc**, das die Zeilen (**-l**, **--lines**), Wörter (**-w**, **--words**) und Zeichen (**-c**, **--chars**, **--bytes**) in den auf der Kommandozeile angegebenen Dateien zählt. Sind keine Optionen angegeben, so entspricht dies der Angabe von **-lwc**. Sind keine Dateinamen auf der Kommandozeile angegeben, so liest dieses Programm den auszuwertenden Text von der Standardeingabe.

```
#include <ctype.h>
#include <string.h>
#include "eighdr.h"
```

```
#define MAX_NAMEN 500
```

```
/*----- Auswerten einer Datei ueber stdin -----*/
void
```

```

auswert(long int *zeilen, long int *woerter, long int *zeichen)
{
    int    zeich, im_wort=0;

    *zeilen = *woerter = *zeichen = 0;
    while ((zeich=getchar()) != EOF) {
        (*zeichen)++;
        if (zeich=='\n')
            (*zeilen)++;
        if (!isspace(zeich)) {
            if (!im_wort) {
                (*woerter)++;
                im_wort=1;
            }
        } else
            im_wort=0;
    }
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    long int    lines=0, words=0, chars=0,
               zeil_zahl, wort_zahl, zeich_zahl,
               gesamtzeilen=0, gesamtwoerter=0, gesamtzeichen=0,
               i, j=0;
    char        *zgr, *dateiname[MAX_NAMEN];

    dateiname[0] = ""; /* Voreinst. ist stdin, wenn keine Dateien angegeben */
    for (i=1 ; i<argc ; i++) {
        zgr = argv[i];
        if (*zgr == '-' && *(zgr+1) && *(zgr+1) != '-') {
            while (*++zgr) {
                switch (tolower(*zgr)) {
                    case 'l': lines = 1; break;
                    case 'w': words = 1; break;
                    case 'c': chars = 1; break;
                    default : fehler_meld(FATAL, "usage: %s [optionen] [datei(en)]\n"
                                           "      -l, --lines\n"
                                           "      -w, --words\n"
                                           "      -c, --chars, --bytes\n", argv[0]);
                }
            }
        } else if (*zgr == '-' && *(zgr+1) == '-' && *(zgr+2)) {
            if (!strcmp(&zgr[2], "lines")) lines = 1;
            else if (!strcmp(&zgr[2], "words")) words = 1;
            else if (!strcmp(&zgr[2], "chars")) chars = 1;
            else if (!strcmp(&zgr[2], "bytes")) chars = 1;
            else fehler_meld(FATAL, "usage: %s [optionen] [datei(en)]\n"
                           "      -l, --lines\n"
                           "      -w, --words\n"
                           "      -c, --chars, --bytes\n", argv[0]);
        } else
    }
}

```

```

        dateiname[j++] = zgr;
    }
    if (lines==0 && words==0 && chars==0)
        lines = words = chars = 1;

    i=0;
    do {
        if (j>0 && freopen(dateiname[i], "r", stdin) != stdin)
            fehler_meld(FATAL_SYS, "Fehler bei freopen von '%s' mit stdin",
                        dateiname[i]);

        auswert(&zeil_zahl, &wort_zahl, &zeich_zahl);
        gesamtzeilen += zeil_zahl;
        gesamtwoerter += wort_zahl;
        gesamtzeichen += zeich_zahl;
        if (i==0) {
            if (lines) printf("%10s", "Zeilen");
            if (words) printf("%10s", "Woerter");
            if (chars) printf("%12s", "Zeichen");
            printf("      Dateiname\n");
            printf("-----\n");
        }
        if (lines) printf("%10ld", zeil_zahl);
        if (words) printf("%10ld", wort_zahl);
        if (chars) printf("%12ld", zeich_zahl);
        printf("      %s\n", dateiname[i]);
    } while (++i < j);

    if (j>1) {
        printf("-----\n");
        if (lines) printf("%10ld", gesamtzeilen);
        if (words) printf("%10ld", gesamtwoerter);
        if (chars) printf("%12ld", gesamtzeichen);
        printf("      %s\n", "Gesamt");
    }

    exit(0);
}

```

Programm 21.7 (wc2.c): Eigenes Abarbeiten der argv-Optionen (einfaches wc-Programm)

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o wc2 wc2.c fehler.c
```

kann man es starten, wie es die nachfolgenden Ablaufbeispiele verdeutlichen.

```
$ wc2 -l fehler.c -c eighdr.h
```

Zeilen	Zeichen	Dateiname
103	2783	fehler.c
74	3358	eighdr.h
177	6141	Gesamt

```
$ wc2 fehler.c -c eighdr.h -lw
  Zeilen   Woerter   Zeichen   Dateiname
-----
    103      281     2783   fehler.c
     74      324     3358   eighdr.h
-----
    177      605     6141   Gesamt

$ wc2 fehler.c eighdr.h
  Zeilen   Woerter   Zeichen   Dateiname
-----
    103      281     2783   fehler.c
     74      324     3358   eighdr.h
-----
    177      605     6141   Gesamt

$ wc2 fehler.c --bytes eighdr.h -lw
  Zeilen   Woerter   Zeichen   Dateiname
-----
    103      281     2783   fehler.c
     74      324     3358   eighdr.h
-----
    177      605     6141   Gesamt

$ wc2 --lines fehler.c --words eighdr.h
  Zeilen   Woerter   Dateiname
-----
    103      281   fehler.c
     74      324   eighdr.h
-----
    177      605   Gesamt

$
```

21.3.2 Die getopt-Funktionen

Zur Auswertung der Kommandozeilenoptionen werden auch eigene Funktionen angeboten.

Zur Auswertung von kurzen Optionen steht die Funktion **getopt** zur Verfügung:

```
#include <unistd.h>

extern char *optarg;          /* globale Variablen, die in */
extern int optind, opterr, optopt; /* <unistd.h> deklariert sind. */

int getopt(int argc, char * const argv[], const char *optstring);

      gibt zurück: Optionszeichen (bei Erfolg);
                  ? bei einem unbekannten Optionszeichen;
                  : bei einem fehlenden Parameter zu einer Option;
                  EOF am Ende der Optionsliste
```

Die Funktion **getopt** arbeitet die Kommandozeilenargumente, die wie bei der **main**-Funktion über die Parameter `argc` und `argv` festgelegt sind, schrittweise ab.

Startet ein String in `argv` mit `'-'` und ist kein String der Form `»-«` oder `»--«`, so wird dieser String von **getopt** als Option(en) interpretiert. Die nach `'-'` folgenden Zeichen sind dann für **getopt** die entsprechenden Optionszeichen. Die Funktion **getopt** liefert bei jedem Aufruf das nächste anstehende Optionszeichen, wobei es die globale Variable `optind` und eine interne statische Variable `nextchar` so setzt, daß sie bei ihrem nächsten Aufruf ihre Abarbeitung an der entsprechenden Stelle fortsetzen kann.

Sind alle Optionen abgearbeitet, liefert die Funktion **getopt** EOF, wobei sie zuvor jedoch in der globalen Variablen `optind` den Index des Strings (Arguments) in `argv` einträgt, der keine Option mehr ist.

Falls die Funktion **getopt** ein Optionszeichen liest, das nicht durch den Benutzer in `optstring` als Option festgelegt wurde, so gibt sie eine Fehlermeldung auf der Standardausgabe aus, schreibt das unerlaubte Zeichen in die globale Variable `optopt` und liefert das Zeichen `'?'` als Rückgabewert.

Um die Ausgabe der Fehlermeldung durch **getopt** zu unterbinden, muß vor dem Aufruf von **getopt** die globale Variable `opterr` auf 0 gesetzt werden.

Die für ein Programm erlaubten Optionen muß der Programmierer in der Variablen `optstring` angeben. Falls eine Option ein weiteres Argument erwartet, so muß nach dieser Option ein Doppelpunkt (:) angegeben werden. Findet die Funktion **getopt** bei ihrer Abarbeitung der Kommandozeile eine solche Option, die ein Argument erwartet, setzt sie den globalen Zeiger `optarg` auf das nachfolgende Zeichen im aktuellen `argv`-String bzw., wenn das Ende des aktuellen `argv`-Strings erreicht ist, auf den nachfolgenden `argv`-String.

Werden nach einem Optionszeichen in `optstring` zwei Doppelpunkte (::) angegeben, so bedeutet dies, daß bei dieser Option ein optionales Argument angegeben werden kann. Beim Lesen einer so spezifizierten Option setzt die Funktion **getopt** den globalen Zeiger `optarg` auf den Anfang des restlichen `argv`-Strings bzw. auf NULL, wenn das Ende des aktuellen `argv`-Strings erreicht ist.

Die Funktion **getopt** vertauscht eventuell die Strings in `argv`, so daß die Argumente, die keine Optionen sind, am Ende der Stringliste `argv` stehen.

Daneben gibt es jedoch noch zwei andere Modi:

- ▶ Wenn das erste Zeichen in `optstring` ein `'+'` ist oder aber die Environment-Variable `POSIXLY_CORRECT` gesetzt ist, beendet die Funktion **getopt** die Abarbeitung der Optionen, sobald sie auf ein Argument trifft, das nicht mit `'-'` beginnt, also keine Option mehr ist.
- ▶ Wenn das erste Zeichen in `optstring` ein `'-'` ist, dann wird jedes Argument, das nicht mit `'-'` beginnt, so behandelt, als ob es ein Argument einer Option mit dem ASCII-Code 1 ist. Dieser Modus wird von Programmen benutzt, bei denen die Reihenfolge

der Optionen und der normalen Argumente nicht vorgeschrieben ist, also diese auch gemischt auf der Kommandozeile angegeben werden können.

Unabhängig vom verwendeten Modus zeigt das spezielle Argument '--' an, daß die Optionenangabe beendet ist.

Beispiel

Abarbeitung von Optionen mit getopt (einfaches wc-Programm)

Das folgende Programm 21.8 (wc3.c) ist wieder eine einfache Realisierung des Kommandos wc, das die Zeilen (**-l**), Wörter (**-w**) und Zeichen (**-c**) in den auf der Kommandozeile angegebenen Dateien zählt. Sind keine Optionen angegeben, so entspricht dies der Angabe von **-lwc**. Sind keine Dateinamen auf der Kommandozeile angegeben, so liest auch dieses Programm den auszuwertenden Text von der Standardeingabe.

```
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include "eighdr.h"

#define MAX_NAMEN 500

/*----- Auswerten einer Datei ueber stdin -----*/
void
auswert(long int *zeilen, long int *woerter, long int *zeichen)
{
    ..... /* siehe vorheriges Programm 21.7 (wc2.c) */
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    long int    lines=0, words=0, chars=0,
               zeil_zahl, wort_zahl, zeich_zahl,
               gesamtzeilen=0, gesamtwoerter=0, gesamtzeichen=0,
               i, j=0;
    char        option, *dateiname[MAX_NAMEN];

    opterr = 0; /* Automatische Fehlerausgabe ausschalten */
    dateiname[0] = ""; /* Voreinst. ist stdin, wenn keine Dateien angegeben */
    while ( (option = getopt(argc, argv, "-lwc")) != EOF) {
        switch (tolower(option)) {
            case 'l': lines = 1; break;
            case 'w': words = 1; break;
            case 'c': chars = 1; break;

            case 1 : if ( (dateiname[j] = malloc(strlen(optarg)+1)) == NULL)
                        fehler_meld(FATAL_SYS, "Speicherplatzmangel");
                    strcpy(dateiname[j++], optarg);
                    break;
        }
    }
}
```



```

        case '?': fehler_meld(FATAL, "Unerlaubte Option: '-%c'\n"
                                "usage: %s [-lwc] [datei(en)]\n",
                                optopt, argv[0]);
                break;
    }
}
if (lines==0 && words==0 && chars==0)
    lines = words = chars = 1;

..... /* siehe vorheriges Programm 21.7 (wc2.c) */

exit(0);
}

```

Programm 21.8 (wc3.c): Abarbeiten der Optionen mit der Funktion *getopt* (einfaches wc-Programm)

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o wc3 wc3.c fehler.c
```

kann man es starten, wie es die nachfolgenden Ablaufbeispiele verdeutlichen.

```

$ wc3 -l fehler.c -c eighdr.h
  Zeilen   Zeichen   Dateiname
-----
    103     2783   fehler.c
     74     3358   eighdr.h
-----
    177     6141   Gesamt

$ wc3 fehler.c -c eighdr.h -lw
  Zeilen  Woerter   Zeichen   Dateiname
-----
    103     281     2783   fehler.c
     74     324     3358   eighdr.h
-----
    177     605     6141   Gesamt

$ wc3 fehler.c eighdr.h
  Zeilen  Woerter   Zeichen   Dateiname
-----
    103     281     2783   fehler.c
     74     324     3358   eighdr.h
-----
    177     605     6141   Gesamt

$

```

Im Programm 21.8 (wc3.c) startet das `optstring`-Argument mit dem Zeichen `-`, so daß normale Argumente und Optionen in einer beliebigen Reihenfolge auf der Kommandozeile angegeben werden können.

Beispiel

Abarbeitung von Optionen vor den normalen Argumenten mit getopt (einfaches wc-Programm)

Das folgende Programm 21.9 (wc4.c) ist wieder wie das vorherige Programm 21.8 (wc3.c) eine einfache Realisierung des Kommandos **wc**, nur daß dieses Programm fordert, daß die Optionen vor den Dateinamen angegeben sind. Diese Forderung wird durch die Angabe von '+' als erstes Zeichen im optstring-Argument realisiert.

```
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include "eighdr.h"

#define MAX_NAMEN 500

/*----- Auswerten einer Datei ueber stdin -----*/
void
auswert(long int *zeilen, long int *woerter, long int *zeichen)
{
    ..... /* siehe vorheriges Programm 21.7 (wc2.c) */
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    long int    lines=0, words=0, chars=0,
               zeil_zahl, wort_zahl, zeich_zahl,
               gesamtzeilen=0, gesamtwoerter=0, gesamtzeichen=0,
               i, j=0;
    char        option, *dateiname[MAX_NAMEN];

    opterr = 0; /* Automatische Fehlerausgabe ausschalten */
    dateiname[0] = ""; /* Voreinst. ist stdin, wenn keine Dateien angegeben */
    while ( (option = getopt(argc, argv, "+lwc")) != EOF ) {
        switch (tolower(option)) {
            case 'l': lines = 1; break;
            case 'w': words = 1; break;
            case 'c': chars = 1; break;

            case '?': fehler_meld(FATAL, "Unerlaubte Option: '-%c'\n"
                                "usage: %s [-lwc] [datei(en)]\n",
                                optopt, argv[0]);
                    break;
        }
    }

    for (i=optind; i<argc; i++)
        dateiname[j++] = argv[i];
}
```

```

if (lines==0 && words==0 && chars==0)
    lines = words = chars = 1;

..... /* siehe vorheriges Programm 21.7 (wc2.c) */

exit(0);
}

```

Programm 21.9 (**wc4.c**): Abarbeiten von Optionen vor den normalen Argumenten mit Funktion **getopt** (einfaches wc-Programm)

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o wc4 wc4.c fehler.c
```

kann man es starten, wie die nachfolgenden Ablaufbeispiele verdeutlichen.

```

$ wc4 -l -w fehler.c eighdr.h
  Zeilen   Woerter   Dateiname
-----
    103      281    fehler.c
     74      324    eighdr.h
-----
    177      605    Gesamt

$ wc4 -l fehler.c -w eighdr.h
  Zeilen   Dateiname
-----
    103    fehler.c
Fehler bei freopen von '-w' mit stdin: No such file or directory

$

```

Zur gleichzeitigen Auswertung von langen und kurzen Optionen stehen die beiden Funktionen **getopt_long** und **getopt_long_only** zur Verfügung.

```
#include <getopt.h>
```

```
int getopt_long(int argc, char * const argv[],
                const char *optstring,
                const struct option *longopts, int *longindex);
```

```
int getopt_long_only(int argc, char * const argv[],
                     const char *optstring,
                     const struct option *longopts, int *longindex);
```

beide geben zurück: Optionszeichen, wenn eine kurze Option gefunden wurde;
 ? bei einer unbekannten Option oder einer mehrdeutigen Optionsangabe;
 : bei einem fehlenden Parameter zu einer Option;
 EOF am Ende der Optionsliste;
 val für eine lange Option, wenn flag auf NULL gesetzt ist;
 0 für eine lange Option, wenn flag nicht auf NULL gesetzt ist.

Die beiden Funktionen **getopt_long** und **getopt_long_only** verhalten sich weitgehend wie die Funktion **getopt**, außer daß sie eben auch lange Optionen akzeptieren.

Die Namen von langen Optionen müssen dabei nicht vollständig ausgeschrieben sein, sondern können auch abgekürzt werden, solange diese Abkürzung eindeutig ist.

Parameter zu langen Optionen können auf zwei verschiedene Arten angegeben werden: `--option=param` oder `--option param`.

Die Funktion **getopt_long_only** unterscheidet sich von der Funktion **getopt_long** darin, daß nicht nur '--', sondern auch '-' am Anfang einer langen Option angegeben werden kann. Falls jedoch zu einer Optionsangabe, die mit '-' beginnt, keine entsprechende lange Option existiert, so wird sie auch bei **getopt_long_only** als kurze Option interpretiert.

Das Argument zum vierten Parameter `longopts` muß bei beiden Funktionen ein Zeiger auf das erste Element eines Arrays von Strukturen des Datentyps `struct option` sein. Dieser Datentyp ist in `<getopt.h>` wie folgt definiert:

```
struct option {
    const char *name;
    int        has_arg;
    int        *flag;
    int        val;
};
```

Die Bedeutung der einzelnen Komponenten ist dabei folgende:

`name`

enthält den Namen der jeweiligen langen Option.

`has_arg`

zeigt an, ob die Option kein, ein oder aber ein optionales Argument hat. Dazu bietet `<getopt.h>` die folgenden Konstanten an:

```
#define no_argument      0 /* Option hat kein Argument      */
#define required_argument 1 /* Option erfordert ein Argument */
#define optional_argument 2 /* Option hat ein optionales Argument */
```

`flag` und `val`

`flag` legt fest, wie Ergebnisse für eine lange Option zurückzugeben sind: Ist `flag` auf `NULL` gesetzt, so wird der in `val` angegebene Wert als Rückgabe geliefert. So kann z.B. in `val` das Optionszeichen einer kurzen Option angegeben werden, die äquivalent zu dieser entsprechenden langen Option ist. Ist `flag` nicht auf `NULL` gesetzt, so wird der Wert 0 zurückgegeben, wobei in diesem Fall der Speicherplatz, auf den `flag` zeigt, entweder den Wert `val` enthält, wenn die Option gefunden wurde, oder aber noch den ursprünglichen Wert enthält, wenn die Option nicht gefunden wurde.

Das letzte Element des Struktur-Arrays, dessen Adresse als Argument für `longopts` übergeben wird, muß eine Struktur sein, bei der alle Komponenten auf 0 bzw. `NULL` gesetzt ist.

Wenn das letzte Argument `longindex` nicht `NULL` ist, so wird dort der Index des Elements im Array `longopts` hinterlegt, zu dem eine Option gefunden wurde.

Beispiel*Demonstrationsprogramm zur Funktion `getopt_long`*

Das folgende Programm 21.10 (`getopt_l.c`) demonstriert die Anwendung der Funktion **`getopt_long`**.

```
#include <stdio.h>
#include <getopt.h>

int
main(int argc, char *argv[])
{
    int
        option,
        zif_optind=0,
        akt_optind,
        option_index = 0;
    struct option long_options[] = {
        { "dump",    0, NULL,  0 },
        { "output",  1, NULL,  'o' },
        { "input",   1, NULL,  'i' },
        { "list",    2, NULL,  'l' },
        { "verbose", 0, NULL,  0 },
        { "help",    0, NULL,  0 },
        { NULL,      0, NULL,  0 }
    };

    printf("Folgende Optionen wurden angegeben:\n");

    while (1) {
        akt_optind = optind;
        option = getopt_long(argc, argv, "i:o:l::h012",
                             long_options, &option_index);

        if (option == EOF)
            break;

        switch (option) {
            case 0:
                printf("  --%s", long_options[option_index].name);
                if (optarg)
                    printf(" mit Parameter %s", optarg);
                printf("\n");
                break;

            case '0':
            case '1':
            case '2':
                if (zif_optind != 0 && zif_optind != akt_optind)
                    fprintf(stderr, "....Ziffernangabe nur in "
```

```

        "einer Option erlaubt\n");
    zif_optind = akt_optind;
    printf("  -%c\n", option);
    break;

case 'i':
case 'o':
    printf("  -%c mit Parameter '%s'\n", option, optarg);
    break;

case 'l':
    printf("  -%c", option);
    if (optarg)
        printf(" mit Parameter %s", optarg);
    printf("\n");
    break;

case 'h':
    printf("  -h\n");
    break;

case '?':
    break;

default:
    fprintf(stderr, " ....unerlaubte Option '-%c'\n", option);
    break;
}
}

if (optind < argc) {
    printf("Normale Argumente (keine Optionen):\n ");
    while (optind < argc)
        printf("%s ", argv[optind++]);
    printf("\n");
}

exit (0);
}

```

Programm 21.10 (*getopt_l.c*): Demonstrationsprogramm zur Funktion *getopt_long*

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o getopt_l getopt_l.c
```

kann man es starten, wie das nachfolgende Ablaufbeispiel verdeutlicht.

```
$ getopt_l -l --help=xxx -h -l2 string1 -output --input=indat --list=hallo dateil
Folgende Optionen wurden angegeben:
```

```
-l
getopt_l: option '--help' doesn't allow an argument
-h
-l
```

```
-2
-o mit Parameter 'output'
-i mit Parameter 'input'
-l mit Parameter 'hallo'
Normale Argumente (keine Optionen):
string1 datei1

$
```

Beispiel*Abarbeitung von Optionen mit getopt_long (einfaches wc-Programm)*

Das folgende Programm 21.11 (wc5.c) ist wieder eine einfache Realisierung des Kommandos **wc**, das die Zeilen (**-l**, **--lines**), Wörter (**-w**, **--words**) und Zeichen (**-c**, **--chars**, **--bytes**) in den auf der Kommandozeile angegebenen Dateien zählt. Sind keine Optionen angegeben, so entspricht dies der Angabe von **-lwc**. Sind keine Dateinamen auf der Kommandozeile angegeben, so liest auch dieses Programm den auszuwertenden Text von der Standardeingabe. Gegenüber den vorherigen **wc**-Versionen wurde dieses Programm noch um die beiden langen Optionen **--help** (Ausgabe von Help-Informationen) und **--version** (Ausgabe der Versionsnummer) erweitert.

```
#include <ctype.h>
#include <string.h>
#include <getopt.h>
#include "eighdr.h"

#define VERSION "wc (Version: 0.99)"
#define MAX_NAMEN 500

/*----- Ausgeben von usage-Information -----*/
void
usage(char *progname)
{
    fprintf(stderr,
        "Usage: %s [option(en)] [datei(en)]\n"
        "Gibt Anzahl der Zeilen, Woerter und Bytes fuer jede Datei,\n"
        "und die Gesamtzahl für alle Dateien aus, wenn mehr als eine\n"
        "Datei angegeben ist.\n"
        "Ist keine Datei angegeben, so wird von stdin gelesen.\n"
        " -c, --bytes, --chars  Ausgeben der Byte-Anzahl\n"
        " -l, --lines           Ausgeben der Zeilen-Anzahl\n"
        " -w, --words           Ausgeben der Wort-Anzahl\n"
        "      --help           Ausgeben dieser Help-Info mit exit\n"
        "      --version        Ausgeben der Versionsnummer mit exit\n",
        progname);
}

/*----- Auswerten einer Datei ueber stdin -----*/
void
auswert(long int *zeilen, long int *woerter, long int *zeichen)
{
```

```

..... /* siehe vorheriges Programm 21.7 (wc2.c) */
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    long int    lines=0, words=0, chars=0,
               zeil_zahl, wort_zahl, zeich_zahl,
               gesamtzeilen=0, gesamtwoerter=0, gesamtzeichen=0,
               i, j=0;
    int         option, option_index = 0,
               fehler = 0, help=0, version = 0;
    struct option long_options[] = {
        { "bytes",    0, NULL, 'c' },
        { "chars",    0, NULL, 'c' },
        { "words",    0, NULL, 'w' },
        { "lines",    0, NULL, 'l' },
        { "help",     0, NULL, 0 },
        { "version",  0, NULL, 0 },
        { NULL,       0, NULL, 0 }
    };
    char *dateiname[MAX_NAMEN];

    dateiname[0] = ""; /* Voreinst. ist stdin, wenn keine Dateien angegeben */

    opterr = 0;
    while (1) {
        option = getopt_long(argc, argv, "-lwc", long_options, &option_index);

        if (option == EOF)
            break;

        switch (option) {
            case 'l': lines = 1; break;
            case 'w': words = 1; break;
            case 'c': chars = 1; break;

            case 0 : if (!strcmp(long_options[option_index].name, "help"))
                       help = 1;
                     else if (!strcmp(long_options[option_index].name, "version"))
                       version = 1;
                     break;

            case 1 : if ( (dateiname[j] = malloc(strlen(optarg)+1)) == NULL)
                       fehler_meld(FATAL_SYS, "Speicherplatzmangel");
                       strcpy(dateiname[j++], optarg);
                       break;

            case '?' : fehler = 1;
                       if (argv[optind-1][1] != '-')
                           fehler_meld(WARNUNG, "...unerlaubte Option '-%c'",
                                       optopt);

```



```

        else
            fehler_meld(WARNUNG, "...unerlaubte Option '%s'",
                        argv[optind-1]);
        break;
    }
}

if (fehler) {
    usage(argv[0]);
    exit(1);
}
if (help) {
    usage(argv[0]);
    exit(0);
}
if (version) {
    fprintf(stderr, "%s\n", VERSION);
    exit(0);
}

if (lines==0 && words==0 && chars==0)
    lines = words = chars = 1;

..... /* siehe vorheriges Programm 21.7 (wc2.c) */

exit(0);
}

```

Programm 21.11 (**wc5.c**): Abarbeiten der Optionen mit der Funktion **getopt_long** (einfaches wc-Programm)

21.3.3 Das GNU-Softwarepaket **popt**

Das Softwarepaket **popt** (kann von der WWW-Seite <http://metalab.unc.edu/pub/Linux/distributions/redhat/code/popt> heruntergeladen werden) sollte auf jedem System verwendet werden können, das sich an den POSIX-Standard hält.

Das Softwarepaket **popt** kann unter der *GNU General Public License* (GPL) oder der *GNU Library General Public License* (LGPL) weitergegeben werden.

Gegenüber den in Kapitel 21.3.2 vorgestellten **getopt**-Funktionen weist das **popt**-Softwarepaket einige Vorteile auf:

- ▶ Es bietet das sogenannte Option-Aliasing an, mit dem der Benutzer neue Optionen hinzufügen kann, die Kombinationen von den bereits existierenden Optionen sind.
- ▶ Da **popt** keine globale Variablen verwendet, kann eine **argv**-Kommandozeile mehrmals auf verschiedene Art mit **popt** untersucht werden.
- ▶ Es ermöglicht eine Klassifizierung von Argumenten, die nach Optionen angegeben werden können, da es die Angabe von deren Datentypen erlaubt.

Die Struktur `poptOption`

Über ein Array, dessen Elemente als Datentyp die Struktur `poptOption` haben, werden die Optionen für eine Kommandozeile spezifiziert. Jeder Eintrag in diesem Array spezifiziert eine Option, die auf der Kommandozeile angegeben werden kann. Die Struktur `poptOption` ist in `<popt.h>` wie folgt definiert:

```
struct popOption {
    const char *longName; /* may be NULL */
    char      shortName; /* may be '\0' */
    int       argInfo;
    void      *arg;       /* depends on argInfo */
    int       val;        /* 0 means don't return, just update flag */
    char      *descrip;   /* description for autohelp -- may be NULL */
    char      *argDescrip; /* argument description for autohelp */
};
```

`longName` legt dabei den langen Namen und `shortName` den kurzen Namen (ein Zeichen) für ein und dieselbe Option fest.

Die Komponente `argInfo` legt fest, welcher Typ von Argument nach dieser Option erwartet wird. Dazu sind in `<popt.h>` eigene Konstanten definiert, von denen die wichtigsten die folgenden sind:

```
#define POPT_ARG_NONE    0 /* kein Argument:  int *arg */
#define POPT_ARG_STRING  1 /* String-Argument: char **arg */
#define POPT_ARG_INT     2 /* int-Argument:   int *arg */
#define POPT_ARG_LONG    3 /* long-Argument:  long *arg */
```

Wird die entsprechende Konstante noch mit bitweisem OR (`|`) mit der Konstante `POPT_ARGFLAG_ONEDASH` verknüpft, so können lange Optionsnamen nicht nur mit zwei Querstrichen (`--langoption`), sondern auch mit einem Querstrich (`-langoption`) auf der Kommandozeile angegeben werden.

Die Komponente `arg` legt eine Adresse fest, an die das entsprechende Optionsargument zu hinterlegen ist. Bei numerischen Optionsargumenten (`argInfo` ist `POPT_ARG_INT` oder `POPT_ARG_LONG`) werden diese entsprechend konvertiert, wobei eine Fehlermeldung zurückgegeben wird, wenn die Konvertierung nicht erfolgreich ist. Erwartet eine Option kein Argument (`argInfo=POPT_ARG_NONE`), wird an den Speicherplatz, auf den `arg` zeigt, der Wert 1 geschrieben, wenn die betreffende Option in der Kommandozeile gefunden wird. Wird `arg` mit `NULL` initialisiert, wird das Optionsargument ignoriert.

Die Komponente `val` legt den Wert fest, der zurückzugeben ist, wenn diese Option gefunden wird. Wird für `val` der Wert 0 angegeben, kehrt die entsprechende `popt`-Funktion nicht zurück, sondern setzt ihre Untersuchung der Kommandozeile mit der nächsten Option fort.

In den beiden Komponenten `descrip` und `argDescrip` kann Text angegeben werden, der bei `--help`, `--usage` oder `-?` automatisch zu dieser Option (`descrip`) bzw. zum Optionsargument (`argDescrip`) auf der Standardfehlerausgabe (`stderr`) auszugeben ist. Wenn eine

solche automatische Hilfsinformation gewünscht ist, sollte im `poptOption`-Array als ein Eintrag das in `<popt.h>` definierte Makro `POPT_AUTOHELP` angegeben werden.

```
#define POPT_AUTOHELP { NULL, '\0', POPT_ARG_INCLUDE_TABLE, \
                        poptHelpOptions, \
                        0, "Help options", NULL },
```

Dieses Makro fügt ein weiteres Options-Array ein. Die Einträge dieses Arrays sind für die entsprechenden automatischen Hilfsinformationen zuständig. Wird beim Aufruf des Programms dann `--help`, `--usage` oder `-?` angegeben, dann wird die entsprechende Information zu den einzelnen Optionen auf `stderr` ausgegeben und das Programm mit dem exit-Status 0 beendet.

In der `argInfo`-Komponente können noch zwei weitere Konstanten angegeben werden, die für spezielle Anwendungen vorgesehen sind:

```
#define POPT_ARG_INCLUDE_TABLE 4 /* arg points to table */
#define POPT_ARG_CALLBACK      5 /* table-wide callback...
                                must be set first in
                                table; arg points to
                                callback, descrip points to
                                callback data to pass */
```

Wird eine dieser beiden Konstanten in `argInfo` angegeben, so legt dieser Eintrag keine Option fest, was durch die Angabe von `NULL` für `longName` und `\0` für `shortName` angezeigt werden muß.

Mit `POPT_ARG_INCLUDE_TABLE` kann ein weiteres Options-Array, das an anderer Stelle definiert ist, in das aktuelle Options-Array übernommen werden. So ist eine Schachtelung von verschiedenen Options-Arrays möglich. Dies ermöglicht es z.B. zu allen in einem Programmpaket bereitgestellten Kommandos einen Standardsatz von Kommandozeilenoptionen zur Verfügung zu stellen. In diesem Fall muß die Komponente `arg` ein Zeiger auf das entsprechende Options-Array sein.

Mit `POPT_ARG_CALLBACK` ist es möglich, eine Funktion (*callback*) anzugeben, die aufzurufen ist, wenn die entsprechende Option gefunden wird. Dies ermöglicht es Programmen, die Options-Arrays von anderen Stellen einfügen, die entsprechenden, dazu bereitgestellten Funktionen aufzurufen, so daß sie sich selbst nicht um die Abarbeitung dieser Optionen kümmern müssen. Eine *callback*-Funktion, deren Adresse in der Komponente `arg` anzugeben ist, sollte den folgenden Prototyp haben:

```
typedef void (*poptCallbackType)(poptContext con,
                                enum poptCallbackReason reason,
                                const struct poptOption *opt,
                                const char *arg, void *data);
```

Der erste Parameter `con` legt den Kontext (siehe weiter unten) fest. Der Parameter `opt` zeigt auf die Option, die den Callback auslöste, und `arg` enthält das zugehörige Optionsargument, was `NULL` ist, wenn zu dieser Option kein Argument vorgesehen ist. Das Options-Array, dessen Elemente die möglichen Optionen für die Kommandozeile festlegen,

muß sein Ende dadurch anzeigen, daß im letzten Element alle Komponenten der `poptOption`-Struktur auf 0 bzw. `NULL` gesetzt sind. Im Parameter `data` schließlich wird der in der Komponente `descrip` angegebene String übergeben, der bei der entsprechenden Option angegeben ist, die den Callback definierte. Über diese Komponente `descrip` können somit an *Callback*-Funktionen beliebige Informationen übergeben werden.

Der *popt*-Kontext

popt ermöglicht es, daß mehrere Kommandozeilen abgearbeitet werden oder aber eine Kommandozeile auf unterschiedliche Weise interpretiert werden kann. Um dies zu ermöglichen, arbeitet *popt* mit sogenannten Kontexten. In einem Kontext werden alle Informationen zu einem bestimmten Satz von Optionen gespeichert. Dazu verwendet *popt* eine interne Struktur `poptContext`.

Ein Kontext kann mit der Funktion ***poptGetContext*** erzeugt werden:

```
#include <popt.h>

poptContext poptGetContext(char *name, int argc, char **argv,
                           const struct poptOption *options,
                           int flags);
```

gibt zurück: Struktur `poptContext`

Der erste Parameter `name` wird nur für den Alias-Mechanismus benutzt, der weiter unten erläutert wird. Hier ist entweder der Name der entsprechenden Anwendung, deren Optionen bearbeitet werden sollen, oder aber eben `NULL` anzugeben, wenn kein Option-Aliasing erwünscht ist.

Die nächsten beiden Argumente `argv` und `argc` legen die Kommandozeilenargumente fest, die zu bearbeiten sind.

Für den Parameter `options` ist das entsprechende Options-Array anzugeben.

Der letzte Parameter `flags` wird zur Zeit nicht genutzt, und es sollte hierfür aus Kompatibilitätsgründen zu zukünftigen *popt*-Versionen der Wert 0 angegeben werden.

Ein `poptContext` enthält neben anderen Informationen auch Information darüber, welche Optionen bereits gesetzt wurden und welche noch nicht.

Soll die Bearbeitung einer Kommandozeile von Beginn an wieder gestartet werden, muß man den Kontext mit der Funktion ***poptResetContext*** wieder zurücksetzen.

Ist eine Kommandozeile vollständig abgearbeitet, sollte man den Kontext mit der Funktion ***poptFreeContext*** wieder freigeben. Diese beiden Funktionen sind in `<popt.h>` wie folgt deklariert:

```
#include <popt.h>

void poptResetContext(poptContext con);

void poptFreeContext(poptContext con);
```

Abarbeiten der Kommandozeile

Nachdem ein Kontext `poptContext` einmal erzeugt ist, kann mit der Abarbeitung der Kommandozeile begonnen werden. Dazu steht die Funktion ***poptGetNextOpt*** zur Verfügung:

```
#include <popt.h>

int poptGetNextOpt(poptContext con);
```

gibt zurück:Komponente `val` (bei Erfolg);
-1 beim letzten Kommandozeilenargument;
POPT_ERROR_... bei Fehler

Diese Funktion bearbeitet das nächste anstehende Argument in der Kommandozeile. Findet sie dazu einen entsprechenden Eintrag im Options-Array, trägt sie in der entsprechenden `arg`-Komponente dieses Eintrags das Optionsargument ein, wenn diese nicht mit `NULL` gesetzt ist. Ist die `val`-Komponente für diesen Eintrag nicht auf 0 gesetzt, gibt sie den Wert der Komponente zurück. Ist aber die `val`-Komponente auf 0 gesetzt, kehrt ***poptGetNextOpt*** nicht zurück, sondern setzt sofort die Bearbeitung der Kommandozeile mit dem nächsten Argument fort.

Die Funktion ***poptGetNextOpt*** gibt -1 zurück, wenn das letzte Kommandozeilenargument untersucht wurde.

Durch die Rückgabe von anderen negativen Werten, die durch die `POPT_ERROR_...`-Konstanten in `<popt.h>` definiert sind, zeigt ***poptGetNextOpt*** das Auftreten eines Fehlers (siehe weiter unten) an.

Wenn alle Kommandozeilenoptionen über die `arg`-Zeiger abgehandelt werden und alle `val`-Komponenten des Options-Arrays den Wert 0 haben, reduziert sich das Abarbeiten einer Kommandozeile auf die folgende Codezeile:

```
kdo_zeile = poptGetNextOpt(context);
```

Da aber viele Anwendungen eine speziellere Bearbeitung der Kommandozeile erfordern, benötigt man meist die folgende Vorgehensweise:

```
while ( (option = poptGetNextOpt(context)) > 0 ) {
    switch (option) {
        ....
    }
```

```
    /* Bearbeitung der einzelnen Argumente */  
    .....  
}  
}
```

Um Optionsargumente zu erhalten, gibt es zwei mögliche Vorgehensweisen.

- ▶ Man läßt das Optionsargument durch die Funktion **poptGetNextOpt** in die *arg*-Komponente des entsprechenden Eintrags im Options-Array eintragen.
- ▶ Man verwendet die Funktion **poptGetOptArg**.

```
#include <popt.h>
```

```
char *poptGetOptArg(poptContext con);
```

gibt zurück: Optionsargument, das nach der letzten mit **poptGetNextOpt** gelesenen Option angegeben ist, oder NULL, wenn kein Argument angegeben wurde.

Kommandozeilenargumente, die keine Optionen sind, wie z.B. Dateinamen oder Strings, beginnen üblicherweise nicht mit einem Querstrich (-). Trifft *popt* auf ein solches Argument, nimmt es dieses in seine interne Liste von übriggebliebenen Argumenten (*leftover arguments*) auf.

Mit den folgenden drei Funktionen **poptGetArg**, **poptPeekArg** und **poptGetArgs** kann auf diese Liste zugegriffen werden.

```
#include <popt.h>
```

```
char *poptGetArg(poptContext con);
```

gibt zurück: nächstes übriggebliebene Argument und markiert es als bearbeitet

```
char *poptPeekArg(poptContext con);
```

gibt zurück: nächstes übriggebliebene Argument, markiert es aber nicht als bearbeitet

```
char **poptGetArgs(poptContext con);
```

gibt zurück: alle übriggebliebenen Argumente als *argv*-Array, wobei das Ende dieses Arrays mit NULL angezeigt wird.

Automatische Hilfsinformationen

popt kann, wie zuvor schon erwähnt wurde, automatisch Hilfsinformation erzeugen, die die verfügbaren Optionen eines Programms beschreibt.

Es gibt zwei Arten von Hilfsinformationen:

- usage
zeigt die Aufrufmöglichkeiten eines Programms mit all seinen Optionen, beschreibt aber die einzelnen Optionen nicht genauer.
- help und -?
gibt eine kurze Beschreibung zu jeder verfügbaren Optionen aus.

Ist die Erzeugung einer automatischen Hilfsinformation erwünscht, müssen die entsprechenden Texte in den Komponenten `descrip` und `argDescrip` in den einzelnen Einträgen des `poptOption`-Arrays angegeben werden.

Zudem muß – wie bereits beschrieben – das Makro `POPT_AUTOHELP` im `poptOption`-Array bei dessen Initialisierung angegeben werden.

Zur Ausgabe der automatisch generierten Hilfsinformation stehen die beiden Funktionen ***poptPrintHelp*** und ***poptPrintUsage*** zur Verfügung.

```
#include <popt.h>

void poptPrintHelp(poptContext con, FILE *f, int flags);

void poptPrintUsage(poptContext con, FILE *f, int flags);
```

Die *popt*-Fehlerbehandlung

Alle *popt*-Funktionen, bei denen Fehler auftreten können, geben im Fehlerfall negative Fehlernummern zurück. Die entsprechenden Konstanten zu den Fehlernummern, die in `<popt.h>` definiert sind, sind in Tabelle 21.1 gezeigt.

Konstante	Beschreibung
POPT_ERROR_NOARG	Zu einer Option, die ein Argument erwartet, fehlt dieses; kann nur von der Funktion <i>poptGetNextOpt</i> zurückgegeben werden.
POPT_ERROR_BADOPT	Auf der Kommandozeile wurde eine Option angegeben, die nicht im Options-Array angegeben ist; kann nur von der Funktion <i>poptGetNextOpt</i> zurückgegeben werden.
POPT_ERROR_OPTSTOODEEP	Ein Satz von Options-Aliase ist zu tief geschachtelt. Momentan erlaubt <i>popt</i> nur eine Tiefe von 10 Ebenen, um Rekursionen zu vermeiden; kann nur von der Funktion <i>poptGetNextOpt</i> zurückgegeben werden.
POPT_ERROR_BADQUOTE	Es fehlt ein schließendes oder ein führendes Anführungszeichen; kann nur von den Funktionen <i>poptArgvString</i> , <i>poptReadConfigFile</i> und <i>poptReadDefaultConfig</i> zurückgegeben werden.

Tabelle 21.1: *popt*-Fehlerkonstanten

Konstante	Beschreibung
POPT_ERROR_BADNUMBER	Konvertierung eines Strings in einen numerischen Wert schlug fehl, da der String nicht numerische Zeichen enthielt; kann nur von der Funktion poptGetNextOpt zurückgegeben werden, wenn diese ein Optionsargument vom Typ POPT_ARG_INT oder POPT_ARG_LONG bearbeitet.
POPT_ERROR_OVERFLOW	Konvertierung eines Strings in einen numerischen Wert schlug fehl, da die betreffende Zahl zu groß oder zu klein ist; kann nur von der Funktion poptGetNextOpt zurückgegeben werden, wenn diese ein Optionsargument vom Typ POPT_ARG_INT oder POPT_ARG_LONG bearbeitet.
POPT_ERROR_ERRNO	Der Aufruf einer Systemfunktion schlug fehl, wobei <code>errno</code> den entsprechenden Fehlercode enthält; kann nur von den Funktionen poptReadConfigFile und poptReadDefaultConfig zurückgegeben werden.

Tabelle 21.1: popt-Fehlerkonstanten

Um sich die entsprechenden Fehlermeldungen aufbereiten zu lassen, stehen die beiden Funktionen **poptStrerror** und **poptBadOption** zur Verfügung.

```
#include <popt.h>

const char *poptStrerror(const int error);
                                gibt zurück: Fehlermeldung, die zur Fehlernummer error gehört.

char *poptBadOption(poptContext con, int flags);
                                gibt zurück: Option, bei der in der Funktion poptGetNextOpt ein Fehler auftrat.
```

Übergibt man bei **poptBadOption** für den Parameter `flags` die Konstante `POPT_BADOPTION_NOALIAS`, so wird die »äußerste« Option zurückgegeben.

Ansonsten sollte man für `flags` den Wert 0 angeben, was bewirkt, daß die zurückgegebene Option dann auch durch ein Alias spezifiziert worden sein kann.

Tritt während der Abarbeitung der Argumente einer Kommandozeile ein Fehler auf, kann z.B. mit folgendem Aufruf die entsprechende zugehörige Fehlermeldung ausgegeben werden:

```
fprintf(stderr, "%s: %s\n",
        poptBadOption(optCon, POPT_BADOPTION_NOALIAS),
        poptStrerror(fehlerrnr));
```


Options-Aliase

Ein großer Vorteil von *popt* gegenüber den **getopt**-Funktionen ist, daß der Aufrufer eines mit *popt* entworfenen Programms selbst Aliase an einzelne Optionen oder Optionsgruppen vergeben kann.

Aliase, also andere Namen, kann er dabei an eine einzelne Option oder an eine ganze Gruppe von Optionen vergeben. Dazu muß er in der Datei */etc/popt* oder in der Datei *.popt* in seinem Home-Directory Zeilen des folgenden Formats angeben:

```
progname alias options-alias option(en)
```

progname ist dabei der Programmname, der bei **poptGetContext** für den ersten Parameter *name* angegeben wurde.

Das Schlüsselwort **alias** legt fest, daß durch diese Zeile ein Alias definiert wird.

options-alias spezifiziert dabei das Options-Alias, was eine kurze oder lange Option sein kann.

Der Rest der Zeile (*option(en)*) legt die Optionen fest, die bei Angabe von *options-alias* auf der Kommandozeile hierfür einzusetzen sind.

Definierte Options-Aliase müssen aktiviert werden, bevor sie von **poptGetNextArg** entsprechend aufgelöst werden können. Dazu stehen drei Funktionen zur Verfügung, die in *<popt.h>* wie folgt deklariert sind:

```
int poptReadDefaultConfig(poptContext con, int useEnv);
```

liest die in */etc/popt* und in *.popt* (im Home-Directory) definierten Aliase. Der Parameter *useEnv* ist für zukünftige Erweiterungen definiert, für den momentan NULL anzugeben ist.

```
int poptReadConfigFile(poptContext con, char *fn);
```

öffnet die Datei *fn* und liest die darin definierten Aliase. Hiermit ist es möglich, programmspezifische Aliase zu definieren, die nicht global für alle Programme verwendet werden, die mit **popt** ihre Optionen abarbeiten.

```
int poptAddAlias(poptContext con, struct poptAlias alias, int flags);
```

fügt ein neues Alias zum Kontext hinzu. Diese Funktion ermöglicht es einem Programm lokal Aliase zu definieren, die nicht aus einer Konfigurationsdatei gelesen werden. Das entsprechende Alias wird dabei durch den Parameter *alias* spezifiziert, dessen Datentyp die Struktur *poptAlias* ist:

```
struct poptAlias {
    char *longName; /* kann NULL sein */
    char shortName; /* kann '\0' sein */
    int argc;
    char **argv;    /* Freigabe mit free muss moeglich sein */
};
```

Die Komponenten `longName` und `shortName` legen den langen und den kurzen Namen des entsprechenden Options-Alias fest.

Die Komponenten `argc` und `argv` spezifizieren die Optionen, die für das Options-Alias einzusetzen sind.

Der Parameter `flags` bei der Funktion ***poptAddAlias*** ist für zukünftige Erweiterungen definiert, für ihn ist momentan `NULL` anzugeben.

Argument-Strings

Normalerweise wird *popt* verwendet, um Argumente zu bearbeiten, die in einem `argv`-Array vorliegen. Es können jedoch auch Anwendungsfälle auftreten, bei denen Strings zu analysieren sind, die eine vollständige Kommandozeile enthalten, die noch nicht in `argv`-Form vorliegt. Für solche Anwendungsfälle existiert die Funktion ***poptParseArgvString***, die einen String in ein Array von einzelnen Argumenten zerlegt.

```
#include <popt.h>

int poptParseArgvString(char *string, int *argcZgr, char ***argvZgr);
```

Die Funktion ***poptParseArgvString*** geht bei der Zerlegung des Strings `string` ähnlich vor wie die Shell. Sie hinterlegt die einzelnen Argumente in das Stringarray, auf das `argvZgr` zeigt, und die Anzahl der erzeugten Argumente schreibt sie in den Speicherplatz, auf den `argcZgr` zeigt. Das Stringarray, auf das `argvZgr` zeigt, wird dabei von der Funktion ***poptParseArgvString*** dynamisch allokiert, dessen spätere Freigabe mit ***free*** in der Verantwortung des Aufrufers dieser Funktion liegt.

Das durch ***poptParseArgvString*** erzeugte Stringarray kann dann an die Funktion ***poptGetContext*** übergeben werden.

Abarbeiten zusätzlicher Argumente

Manche Anwendungen bieten von sich aus so etwas ähnliches wie Options-Aliase an. Um solche zusätzliche Argumente in einen Kontext einzufügen, steht die Funktion ***poptStuffArgs*** zur Verfügung.

```
#include <popt.h>

int poptStuffArgs(poptContext con, char **argv);
```

Der Aufrufer dieser Funktion muß dafür sorgen, daß das Array `argv` am Ende einen `NULL`-Zeiger enthält.

Nach einem Aufruf von ***poptStuffArgs*** werden beim nächsten Aufruf von ***poptGetNextOpt*** diese zusätzlichen Argumente abgearbeitet. Die Bearbeitung der normalen Argumente wird erst wieder fortgesetzt, wenn alle zusätzlichen Argumente abgearbeitet sind.

Demonstrationsprogramm zu den *popt*-Funktionen

Das folgende Programm 21.12 (`popt1.c`) demonstriert einige der eben vorgestellten Funktionen.

```
#include <stdio.h>
#include <stdlib.h>
#include <popt.h>

void option_callback(poptContext con, enum poptCallbackReason reason,
                    const struct poptOption * opt,
                    char * arg, void * data) {
    fprintf(stdout, "callback: %c %s %s ",
            opt->val, (char *) data, arg);
}

int
main(int argc, char *argv[])
{
    int          rc, arg1=0, arg3=0,
                inc=0, help=0, usage=0, kurzopt=0;
    char         *arg2 = "(nicht gesetzt)";
    poptContext context;
    char         **rest;

    struct poptOption callbackArgs[] = {
        { NULL, '\0', POPT_ARG_CALLBACK,
          option_callback, 0, "irgendwelche Daten" },
        { "cb", 'c', POPT_ARG_STRING, NULL, 'c',
          "Testen von Argument-Callbacks" },
        { NULL, '\0', 0, NULL, 0 }
    };

    struct poptOption moreCallbackArgs[] = {
        { NULL, '\0', POPT_ARG_CALLBACK | POPT_CBFLAG_INC_DATA,
          option_callback, 0, NULL },
        { "cb2", 'c', POPT_ARG_STRING, NULL,
          'c', "Testen von Argument-Callbacks" },
        { NULL, '\0', 0, NULL, 0 }
    };

    struct poptOption moreArgs[] = {
        { "inc", 'i', 0, &inc, 0, "Eingefuegtes Argument" },
        { NULL, '\0', 0, NULL, 0 }
    };

    struct poptOption options[] = {
        { "arg1", '\0', 0, &arg1, 0,
```

```

        "Beschreibung zum ersten Argument, "
        "welche hier absichtlich etwas laenger ist, "
        "um einen Zeilenumbruch zu erreichen", NULL },
    { "arg2", '2', POPT_ARG_STRING, &arg2, 0,
      "Zweites Argument", "string" },
    { "arg3", '3', POPT_ARG_INT, &arg3, 0,
      "Drittes Argument", "anzahl" },
    { "kurz", '\0', POPT_ARGFLAG_ONEDASH, &kurzopt, 0,
      "Als Praefix auch ein - erlaubt", NULL },
    { "hidden", '\0', POPT_ARG_STRING | POPT_ARGFLAG_DOC_HIDDEN,
      NULL, 0,
      "Sollte nicht gezeigt werden", NULL },
    { NULL, '\0', POPT_ARG_INCLUDE_TABLE, &moreArgs, 0,
      "Mehr Argumente" },
    { NULL, '\0', POPT_ARG_INCLUDE_TABLE, &callbackArgs, 0,
      "Callback-Argumente" },
    { NULL, '\0', POPT_ARG_INCLUDE_TABLE, &moreCallbackArgs, 0,
      "Mehr Callback-Argumente" },
    POPT_AUTOHELP
    { NULL, '\0', 0, NULL, 0 }
};

context = poptGetContext("popt1", argc, argv, options, 0);
poptReadConfigFile(context, ".popt1rc");

if ((rc = poptGetNextOpt(context)) < -1) {
    fprintf(stderr, "popt1: ungueltiges Argument %s: %s\n",
             poptBadOption(context, POPT_BADOPTION_NOALIAS),
             poptStrerror(rc));
    return 2;
}

if (help) {
    poptPrintHelp(context, stdout, 0);
    return(0);
} if (usage) {
    poptPrintUsage(context, stdout, 0);
    return(0);
}

fprintf(stdout, "arg1: %d\n", arg1);
fprintf(stdout, "arg2: %s\n", arg2);
if (arg3)    fprintf(stdout, "arg3: %d\n", arg3);
if (inc)     fprintf(stdout, "inc: %d\n", inc);
if (kurzopt) fprintf(stdout, "kurz: %d\n", kurzopt);

rest = poptGetArgs(context);
if (rest) {
    fprintf(stdout, "Rest: \"%s\"", *rest++);
    while (*rest)
        fprintf(stdout, ", \"%s\"", *rest++);
    fprintf(stdout, "\n");
}

```

```

    fprintf(stdout, "\n");

    exit(0);
}

```

Programm 21.12 (*popt1.c*): Demonstrationsprogramm zu den *popt*-Funktionen

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o pop1 pop1.c -lpopt
```

kann man es starten, wie die folgenden Ablaufbeispiele verdeutlichen.

```

$ pop1
arg1: 0
arg2: (nicht gesetzt)

$ pop1 --help

Usage: pop1 [OPTION...]
  --arg1          Beschreibung zum ersten Argument, welche hier
                  absichtlich etwas laenger ist, um einen Zeilenumbruch zu
                  erreichen
  -2, --arg2=string Zweites Argument
  -3, --arg3=anzahl Drittes Argument
  --kurz          Als Praefix auch ein - erlaubt

Mehr Argumente
  -i, --inc       Eingefuegtes Argument

Callback-Argumente
  -c, --cb=ARG    Testen von Argument-Callbacks

Mehr Callback-Argumente
  -c, --cb2=ARG   Testen von Argument-Callbacks

Help options
  -?, --help      Show this help message
  --usage         Display brief usage message

$ pop1 --usage

Usage: pop1 [-i?] [--arg1] [-2 string] [-3 anzahl] [--kurz] [-c ARG] [-c ARG]
          [--usage]

$ pop1 -i --arg1 -3 543 --kurz
arg1: 1
arg2: (nicht gesetzt)
arg3: 543
inc: 1
kurz: 1

$ pop1 -i hallo wie gehts denn
arg1: 0

```

```
arg2: (nicht gesetzt)
inc: 1
Rest: "hallo", "wie", "gehts", "denn"

$
```

Für die folgenden Ablaufbeispiele wird angenommen, daß die Konfigurationsdatei `.popt1rc` (im Working Directory) den folgenden Inhalt hat:

```
popt1 alias --zwei --arg2
popt1 alias --two --arg1 --arg2 alias
popt1 alias --normalarg --
popt1 alias -T --arg2
popt1 alias -O --arg1

popt1 exec --echo-args echo
popt1 alias -e --echo-args
popt1 exec -a /bin/echo
```

Nun können beim Aufruf von **popt1** auch Alias-Optionen angegeben werden.

```
$ popt1 --two --zwei abc
arg1: 1
arg2: abc

$ popt1 -a -T hallo -O
./popt1 ; --arg2 hallo --arg1

$ popt1 --two --normalarg eins zwei drei
arg1: 1
arg2: alias
Rest: "eins", "zwei", "drei"

$
```

Realisierung des wc-Programms mit popt

Das folgende Programm 21.13 (`wc6.c`) ist eine Realisierung des früher vorgestellten **wc**-Programms (`wc5.c`) unter Verwendung von **popt**.

```
#include <ctype.h>
#include <string.h>
#include <popt.h>
#include "eighdr.h"

#define VERSION "wc (popt-Version: 0.99)"
#define MAX_NAMEN 500

/*----- Ausgeben von usage-Information -----*/
void
usage(char *programe)
{
    fprintf(stderr,
```

```

    "Usage: %s [option(en)] [datei(en)]\n"
    "Gibt Anzahl der Zeilen, Woerter und Bytes fuer jede Datei,\n"
    "und die Gesamtzahl für alle Dateien aus, wenn mehr als eine\n"
    "Datei angegeben ist.\n"
    "Ist keine Datei angegeben, so wird von stdin gelesen.\n"
    "  -c, --bytes, --chars    Ausgeben der Byte-Anzahl\n"
    "  -l, --lines             Ausgeben der Zeilen-Anzahl\n"
    "  -w, --words             Ausgeben der Wort-Anzahl\n"
    "  -?, --help              Ausgeben dieser Help-Info mit exit\n"
    "      --version           Ausgeben der Versionsnummer mit exit\n",
    progname);
}

/*----- Auswerten einer Datei ueber stdin -----*/
void
auswert(long int *zeilen, long int *woerter, long int *zeichen)
{
    int    zeich, im_wort=0;

    *zeilen = *woerter = *zeichen = 0;
    while ((zeich=getchar()) != EOF) {
        (*zeichen)++;
        if (zeich=='\n')
            (*zeilen)++;
        if (!isspace(zeich)) {
            if (!im_wort) {
                (*woerter)++;
                im_wort=1;
            }
        } else
            im_wort=0;
    }
}

/*----- main -----*/
int
main(int argc, char *argv[])
{
    long int    lines=0, words=0, chars=0,
               zeil_zahl, wort_zahl, zeich_zahl,
               gesamtzeilen=0, gesamtwoerter=0, gesamtzeichen=0,
               i, j=0, rc;

    int        fehler = 0, help =0, version = 0;
    char        *dateiname[MAX_NAMEN];
    poptContext context;
    char        **rest;

    struct poptOption optionen[] = {
        { "bytes", 'c', POPT_ARG_NONE, &chars, 0,
          "Ausgeben der Byte-Anzahl", NULL },
        { "chars", 'c', POPT_ARG_NONE, &chars, 0,
          "Ausgeben der Byte-Anzahl", NULL },
        { "lines", 'l', POPT_ARG_NONE, &lines, 0,

```

```

        "Ausgeben der Zeilen-Anzahl", NULL },
    { "words", 'w', POPT_ARG_NONE, &words, 0,
      "Ausgeben der Wort-Anzahl", NULL },
    { "version", 'v', POPT_ARG_NONE, &version, 0,
      "Ausgeben der Versionsnummer mit exit", NULL },
    { "help", '?', POPT_ARG_NONE, &help, 0,
      "Ausgeben dieser Help-Info mit exit", NULL },
    { NULL, '\0', 0, NULL, 0 }
};

dateiname[0] = ""; /* Voreinst. ist stdin, wenn keine Dateien angegeben */
context = poptGetContext(NULL, argc, argv, optionen, 0);

if ((rc = poptGetNextOpt(context)) < -1) {
    fehler_meld(WARNUNG, "....unerlaubte Option %s: %s\n",
               poptBadOption(context, POPT_BADOPTION_NOALIAS),
               poptStrerror(rc));
    fehler = 1;
}

rest = poptGetArgs(context);
while (*rest) {
    if ( (dateiname[j] = malloc(strlen(*rest)+1)) == NULL)
        fehler_meld(FATAL_SYS, "Speicherplatzmangel");
    strcpy(dateiname[j++], *rest++);
}

if (fehler || help) {
    usage(argv[0]);
    exit(fehler ? 1 : 0);
}

if (version) {
    fprintf(stderr, "%s\n", VERSION);
    exit(0);
}

if (lines==0 && words==0 && chars==0)
    lines = words = chars = 1;

i=0;
do {
    if (j>0 && freopen(dateiname[i], "r", stdin) != stdin)
        fehler_meld(FATAL_SYS, "Fehler bei freopen von '%s' mit stdin",
                    dateiname[i]);

    auswert(&zeil_zahl, &wort_zahl, &zeich_zahl);
    gesamtzeilen += zeil_zahl;
    gesamtwoerter += wort_zahl;
    gesamtzeichen += zeich_zahl;
    if (i==0) {
        if (lines) printf("%10s", "Zeilen");
        if (words) printf("%10s", "Woerter");
        if (chars) printf("%12s", "Zeichen");
    }
} while (*rest++);

```



```
        printf("      Dateiname\n");
        printf("-----\n");
    }
    if (lines) printf("%10ld", zeil_zahl);
    if (words) printf("%10ld", wort_zahl);
    if (chars) printf("%12ld", zeich_zahl);
    printf("      %s\n", dateiname[i]);
} while (++i < j);

if (j>1) {
    printf("-----\n");
    if (lines) printf("%10ld", gesamtzeilen);
    if (words) printf("%10ld", gesamtwoerter);
    if (chars) printf("%12ld", gesamtzeichen);
    printf("      %s\n", "Gesamt");
}

exit(0);
}
```

Programm 21.13 (wc6.c): Realisierung des wc-Programms mit *popt*

22 Wichtige Entwicklungswerkzeuge

*Es ist nicht genug, zu wissen,
man muß auch anwenden;
es nicht nicht genug, zu wollen,
man muß auch tun.*

Goethe

In diesem Kapitel wird ein kurzer Einblick in wichtige Entwicklungswerkzeuge gegeben, die bei der Systemprogrammierung unter Linux/Unix sehr hilfreich sein können.

22.1 gcc – Der GNU-C-Compiler

Die meisten Kommandozeilenangaben des GNU-Compilers **gcc** entsprechen denen anderer C-Compiler unter Unix, jedoch gibt es auch einige **gcc**-spezifische Angaben und Eigenheiten. Hier werden die wichtigsten Kommandozeilenangaben und Eigenschaften des **gcc**-Compilers kurz vorgestellt. Für weitergehende Informationen ist **man gcc** aufzurufen.

22.1.1 Aufrufsyntax

`gcc [option(en)] datei(en)` bzw. auch

`cc [option(en)] datei(en)1` bzw. auch

`g++ [option(en)] datei(en)2` bzw. auch

`c++ [option(en)] datei(en)3`

22.1.2 Klassifikation der Dateitypen durch Suffixe

gcc ist das Kommando zum Aufruf des GNU-C-Compilers. Es erzeugt ausführbare Programme, indem es die angegebenen *datei(en)* kompiliert bzw. assembliert, bevor es den Linker **ld** aufruft, um die entsprechenden Objektdateien zu einem ausführbaren Programm zusammenbinden zu lassen. Die Voreinstellung ist, daß **gcc** das erzeugte Programm in einer Datei mit dem Namen `a.out` ablegt.

-
1. Diese zweite Aufrufform ist auch oft möglich, da meist ein symbolischer Link `/usr/bin/cc -> /usr/bin/gcc` existiert.
 2. Bei **g++** handelt es sich um den GNU-C++-Compiler.
 3. Diese zweite Aufrufform ist auch oft möglich, da meist ein symbolischer Link `/usr/bin/c++ -> /usr/bin/g++` existiert.

Als *datei(en)* akzeptiert **gcc** eine ganze Reihe von Dateitypen, die **gcc** dabei über das Suffix klassifiziert. Die wichtigsten Suffixe sind in Tabelle 22.1 zusammengefaßt:

Suffix	Dateityp
.c	C-Quellprogramm Ein C-Quellprogramm wird zunächst in eine Objektdatei übersetzt, wobei für den Namen der Objektdatei das Suffix <i>.c</i> durch <i>.o</i> ersetzt wird. Falls nur ein C-Quellprogramm beim cc -Aufruf angegeben ist, wird die <i>.o</i> -Datei sofort gelinkt und dann gelöscht.
.h	C-Headerdatei
.C	C++-Quellprogramm
.cc	C++-Quellprogramm
.cxx	C++-Quellprogramm
.m	Objective-C-Quellprogramm
.s	Assembler-Quellprogramm Ein Assembler-Quellprogramm wird zunächst assembliert und daraus dann eine Objektdatei erstellt, wobei für den Namen der Objektdatei das Suffix <i>.s</i> durch <i>.o</i> ersetzt wird. Falls nur ein Assembler-Quellprogramm beim gcc -Aufruf angegeben ist, wird die <i>.o</i> -Datei sofort gelinkt und dann gelöscht.
.S	Assembler-Quellprogramm Anders als bei der Endung <i>.s</i> wird ein solches Assemblerprogramm auch durch den Präprozessor »geschickt«.
.i	Vom Präprozessor vorverarbeitetes C-Quellprogramm Ein solches vorverarbeitetes C-Quellprogramm wird zunächst in eine Objektdatei übersetzt, wobei für den Namen der Objektdatei das Suffix <i>.i</i> durch <i>.o</i> ersetzt wird. Falls nur eine <i>.i</i> -Datei beim cc -Aufruf angegeben ist, wird die <i>.o</i> -Datei sofort gelinkt und dann gelöscht.
.ii	Vom Präprozessor vorverarbeitetes C++-Quellprogramm
andere Suffixe	Dateien, deren Namen mit einem anderen Suffix enden (wie z.B. Objektdateien mit Suffix <i>.o</i> oder Bibliotheken mit Suffix <i>.a</i>), werden von gcc solange ignoriert, bis alle auf der Kommandozeile angegebenen Quellprogramme kompiliert oder assembliert sind. Erst dann übergibt gcc alle gerade generierten Objektdateien zusammen mit diesen explizit auf der Kommandozeile erwähnten Objekt- und Bibliotheksdateien an den Linker ld , damit dieser sie alle zu einem ausführbaren Programm zusammenbindet.

Tabelle 22.1: Die wichtigsten Suffixe für den **gcc**- bzw **g++**-Compiler

gcc legt normalerweise seine übersetzten Dateien im Working-Directory ab. Deshalb ist es wichtig, daß das Working-Directory nicht schreibgeschützt ist.

22.1.3 Wichtige Optionen

Tabelle 22.2 zeigt einige Optionen, die beim Arbeiten mit **gcc** häufig benötigt werden.

Option	Bedeutung
-ansi	schaltet den ANSI-C-Standard ein, so daß nur ANSI-C-Konstrukte in den zu kompilierenden Quellprogrammen verwendet werden können.
-c	(<i>compile only</i>) die angegebenen Quellprogramme nur kompilieren und nicht linken. In diesem Fall werden die erzeugten Objektdateien (Suffix .o) nicht gelöscht.
-C	(<i>Comment</i>) veranlaßt den Präprozessor, alle Kommentarzeilen an den Compiler weiterzuleiten. Ausnahme sind dabei Kommentare, die in Zeilen mit Präprozessoranweisungen stehen; wird oft im Zusammenhang mit der Option -E benutzt.
-Dname[=wert]	(<i>Define</i>) definiert den Namen <i>name</i> für den Präprozessor, als ob dieser Name mit #define in jedem Quellprogramm definiert wäre. Falls nur -Dname angegeben ist, entspricht dies der Angabe -Dname=1 . Wird für <i>wert</i> ein String angegeben, muß die Interpretation der Anführungszeichen durch die Shell ausgeschaltet werden, wie z.B. ' -D"sprache=german"' oder -D\"sprache=german\". Sollte der String Leerzeichen enthalten, empfiehlt sich die erste Angebeform.
-E	die angegebenen Quellprogramme werden nur durch den Präprozessor geschickt und das Ergebnis wird auf der Standardausgabe ausgegeben.
-g, -ggdb	(<i>debug</i>) fügt Debug-Information zum generierten Programm bzw. zu den Objektdateien hinzu. -g veranlaßt gcc , nur Standard-Debug-Informationen hinzuzufügen, während -ggdb dagegen bewirkt, daß gcc spezielle Debug-Informationen hinzufügt, die nur der Debugger gdb versteht. gcc kann im übrigen – anders als andere Compiler – auch für optimierten Code Debug-Informationen generieren.
-Idirectory	(<i>Include-directory</i>) fügt das angegebene <i>directory</i> zur Liste der Directories hinzu, in denen nach #include -Dateien zu suchen ist. Die voreingestellte Suche für in spitzen Klammern (< . . >) angegebene #include -Dateien ist das Directory /usr/include und für in Anführungszeichen (" . . ") angegebene #include -Dateien das Working-Directory.
-lname	(<i>library</i>) verwendet zum Linken die Bibliothek libname.so bzw. libname.a . Wenn nicht anders vorgegeben, verwendet gcc zum Linken dynamische Bibliotheken (libname.so) statt statischer Bibliotheken (libname.a). Der Linker sucht nach Funktionen (<i>unresolved references</i>) in allen angegebenen Bibliotheken in der Reihenfolge, in der diese angegeben sind, bis jeweils der erste passende Eintrag gefunden wurde.

Tabelle 22.2: Wichtige **gcc**-Optionen

Option	Bedeutung
-Ldirectory	(Library) fügt das angegebene <i>directory</i> zur Liste der Directories hinzu, in denen nach Bibliotheksdateien zu suchen ist. Wenn nicht anders angegeben, zieht gcc dynamische Bibliotheken (<i>shared libraries</i>) der Verwendung von statischen Bibliotheken (<i>static libraries</i>) vor. Das voreingestellte Directory für die Suche nach Bibliotheken ist <i>/usr/lib</i> .
-o name	(output) Normalerweise erzeugt gcc eine Ausgabedatei mit dem Namen <i>a.out</i> . Wird ein anderer Name <i>name</i> für die von gcc erzeugte Datei gewünscht, ist dies mit dieser Option möglich. Diese Option ist auch sehr nützlich, wenn die Ausgabedatei(en) in ein anderes Directory abzulegen sind.
-O,-On	(Optimize) schaltet den Optimierer ein. Über die Angabe einer Ziffer <i>n</i> kann man diese Optimierungsstufe festlegen. -O ohne Angabe einer Ziffer entspricht der niedrigsten Optimierungsstufe (-O1). -O0 schaltet die Optimierung aus. Zur Zeit ist -O3 die höchste Optimierungsstufe.
-p	(profiling) fügt in den Objektdateien zusätzlichen Profilingcode hinzu. Der Profilingcode zählt mit, wie oft die einzelnen Funktionen aufgerufen werden und schreibt diese Information in die Datei <i>gmon.out</i> . Mit Hilfe des Kommandos gprof kann daraus dann nach dem Programmlauf eine lesbare Protokolldatei generiert werden, die angibt, wie oft die einzelnen Funktionen aufgerufen wurden.
-pendantic	weist gcc an, alle Warnungen und Fehlermeldungen auszugeben, die vom ANSI-C-Standard gefordert werden.
-static	zum Linken werden nur statische Bibliotheken verwendet.
-S	Die angegebenen C-Dateien werden übersetzt, jedoch nicht assembliert oder gelinkt. Die dabei erzeugten Assemblerprogramme werden in Dateien mit dem Suffix <i>.s</i> abgelegt.
-Uname	(Undefine) Definition des Namens <i>name</i> für den Präprozessor aufheben, so als ob die Definition für <i>name</i> mit #undef in jedem Quellprogramm aufgehoben worden wäre. Falls derselbe Name sowohl in einer -D als auch einer -U Option erwähnt ist, so hat -U eine höhere Priorität.
-Wall	aktiviert alle im allgemeinen sinnvollen Warnungen, über die gcc verfügt. Mit dieser Option erreicht man einen ähnlich sicheren Code, wie wenn man den Syntaxprüfer lint auf seine Quellprogramme anwenden würde. gcc erlaubt es jedoch, einzelne Warnungen an- oder auszuschalten. Um sich alle diese Warnungstypen (-Wtyp) auflisten zu lassen, muß man man gcc aufrufen.

Tabelle 22.2: Wichtige **gcc**-Optionen

22.1.4 C-Erweiterungen im gcc

gcc bietet einige Konstrukte an, die nicht von ANSI C vorgeschrieben sind. Nachfolgend sind einige wichtige solche Konstrukte beschrieben. Weitere Informationen dazu können mit dem Aufruf **man gcc** erfragt werden.

Der Datentyp long long

Der Datentyp **long long** steht für eine Speichereinheit, die mindestens so viele Bytes wie **long** umfaßt. Auf 32-Bit-Plattformen (wie z.B. bei den Intel-X86-Prozessoren) ist **long** 32 Bit und **long long** 64 Bit groß. Auf 64-Bit-Plattformen (wie z.B. dem Alpha-Prozessor) sind sowohl **long** als auch **long long** 64 Bit groß; dasselbe gilt auf diesen Plattformen für Zeiger. In der nächsten Revision von ANSI C wird dieser Datentyp **long long** sehr wahrscheinlich im ANSI-C-Standard aufgenommen werden.

inline-Funktionen

Von dieser Art von Funktionen wird insbesondere in den Linux-Kernprogrammen Gebrauch gemacht. Die Funktionen laufen so schnell wie Makros ab, da das Stackmanagement entfällt, andererseits bieten **inline**-Funktionen die Vorteile von Funktionen (Typüberprüfung der Argumente, Auswertung der Argumente vor dem Funktionsaufruf usw.) an. Programme, die **inline**-Funktionen verwenden, müssen wenigstens mit der minimalen Optimierung (**-O** bzw. **-O0**) kompiliert werden.

Zusätzliche alternative Schlüsselwörter

gcc bietet eine Reihe von zusätzlichen Schlüsselwörtern an, die nicht von ANSI C vorgeschrieben sind. Solche zusätzlichen Schlüsselwörter werden von **gcc** in zwei Varianten angeboten: einmal das Schlüsselwort selbst (wie z.B. **attribute**) und zum anderen das Schlüsselwort mit zwei vorangestellten und zwei angefügten Unterstrichen (wie z.B. **__attribute__**). Wird **gcc** mit der Option **-ansi** aufgerufen, kann er die zusätzlichen normalen Schlüsselwörter nicht erkennen. Deshalb wurde zu jedem zusätzlichen Schlüsselwort alternativ in den Headerdateien ein entsprechender Datentyp mit zwei vorangestellten und zwei angefügten Unterstrichen angeboten.

Das zusätzliche Schlüsselwort **attribut** ermöglicht es, **gcc** mehr Informationen über eine Funktion, Variable oder einen Datentyp zu geben, als dies mit den Standardkonstrukten von ANSI C möglich ist. Nachfolgend sind einige mögliche Attribute angegeben:

aligned

legt fest, wie eine Variable oder Datenstruktur im Speicher anzuordnen ist.

packed

legt fest, daß bei der Ausrichtung der Daten keine Lücken verwendet werden sollen.

noreturn

legt fest, daß eine Funktion nie zum Aufrufer zurückkehrt, was es **gcc** ermöglicht, besseren Code zu generieren.

Attribute für Funktionen müssen der Funktionsdeklaration hinzugefügt werden, wie z.B.:

```
void function(int, float) __attribute__((__noreturn__));
```

Das Schlüsselwort **__attribut__** ist nach den Funktionsparametern, gefolgt von dem zu setzenden Attribut, das sich in doppelten Klammernpaaren befindet, anzugeben.

Sollen mehrere Attribute gesetzt werden, müssen diese mit Komma getrennt werden, wie z.B.:

```
extern void ext2_panic (struct super_block *, const char *,
                      const char *, ...)
    __attribute__((noreturn, format(printf, 3, 4)));
```

Diese Deklaration legt fest, daß **ext2_panic** nicht zur aufrufenden Funktion zurückkehrt und daß die übergebenen Argumente (ab dem dritten) wie bei der Funktion **printf** zu behandeln sind: Das dritte Argument legt den Formatierungs-String fest und das vierte Argument ist der erste zu ersetzende Parameter im Formatierungs-String.

An späterer Stelle in diesem Kapitel werden weitere Attribute (z.B. beim Erzeugen von dynamischen Bibliotheken) vorgestellt. Alle möglichen Attribute können mit dem Aufruf **man gcc** erfragt werden.

22.2 ld – Der Linux/Unix-Linker

ld ist der Linux/Unix-Linker, der mehrere Objektdateien zu einem ausführbaren Programm zusammenbindet. Objektdateien erkennt **ld** am Suffix **.o**. Archivbibliotheken, in denen der Linker **ld** nach *unresolved references* suchen soll, erkennt er am Suffix **.a**. Falls eine der angegebenen *datei(en)* weder das **.o**- noch das **.a**-Suffix hat, so nimmt **ld** an, daß es sich um eine Archivbibliothek oder um eine Textdatei, die Link-Editordirektiven enthält, handelt.

Die Voreinstellung ist, daß **ld** das erzeugte Programm in einer Datei mit dem Namen **a.out** ablegt, wenn keine Fehler aufgetreten sind, ansonsten bricht **ld** mit einer Fehlermeldung ab.

Explizit auf der Kommandozeile angegebene Bibliotheken werden nur nach *unresolved references* durchsucht, die aus zuvor angegebenen Objektdateien resultieren. Allgemein gilt, daß man alle Objektdateien vor den Bibliotheken auf der Kommandozeile angeben sollte.

22.2.1 Aufrufsyntax

```
ld [option(en)] datei(en)
```

22.2.2 Einige wichtige Optionen

Da **ld** auf den einzelnen Systemen auch die unterschiedlichsten Optionen anbietet, werden hier (in Tabelle 22.3) nur die wichtigsten Optionen vorgestellt, die auch auf den meisten Systemen gültig sind. Um spezielle für ein System angebotenen Optionen zu erfahren, wird man fast immer **man ld** aufrufen oder aber auf die mitgelieferte Dokumentation zurückgreifen müssen.

Option	Bedeutung
-e startsymbol	(entry) Die Adresse des Symbols <i>startsymbol</i> soll die Startadresse für das erzeugte ausführbare Programm sein.
-lname	(library) verwendet zum Linken die Bibliothek libname.so bzw. libname.a . Wenn nicht anders vorgegeben, verwendet gcc zum Linken dynamische Bibliotheken (libname.so) statt statischer Bibliotheken (libname.a). Der Linker sucht nach Funktionen (<i>unresolved references</i>) in allen angegebenen Bibliotheken in der Reihenfolge, in der diese angegeben sind, bis jeweils der erste passende Eintrag gefunden wurde.
-Ldirectory	(Library) fügt das angegebene <i>directory</i> zur Liste der Directories hinzu, in denen nach Bibliotheksdateien zu suchen ist. Wenn nicht anders angegeben, zieht gcc dynamische Bibliotheken (<i>shared libraries</i>) der Verwendung von statischen Bibliotheken (<i>static libraries</i>) vor. Das voreingestellte Directory für die Suche nach Bibliotheken ist <i>/usr/lib</i> .
-o name	(output) Normalerweise erzeugt ld eine Ausgabedatei mit dem Namen <i>a.out</i> . Wird ein anderer Name <i>name</i> für die von ld erzeugte Datei gewünscht, ist dies mit dieser Option möglich. Diese Option ist auch sehr nützlich, wenn die Ausgabedatei(en) in ein anderes Directory abzulegen sind.
-s	(strip) entfernt Zeilennummereinträge und Symboltabelleninformation bei der Generierung des ausführbaren Programms.
-u symbol-name	(undefine) bewirkt, daß <i>symbolname</i> als undefiniertes Symbol in der Symboltabelle eingetragen wird. Dies ist beim ausschließlichen Laden einer Bibliothek nützlich, da die Symboltabelle anfänglich leer ist und mindestens eine <i>unresolved reference</i> benötigt wird, um ld zu zwingen, Funktionen aus einer Bibliothek in das Programm zu übernehmen. Diese Option muß unbedingt vor dem entsprechenden Bibliotheksnamen auf der Kommandozeile angegeben sein.
-V	(Version) bewirkt die Ausgabe der Versionsnummer von ld .

Tabelle 22.3: Wichtige **ld**-Optionen

Hinweis

Da **ld** automatisch von **cc** bzw. **gcc** aufgerufen, nachdem **cc** bzw. **gcc** alle C- und Assemblerprogramme assembliert bzw. kompiliert hat, wird meist **cc** bzw. **gcc** zur Erzeugung eines ausführbaren Programms verwendet.

22.3 gdb – Der GNU-Debugger

gdb ist der übliche Debugger unter Unix. Hier wird der **GNU-gdb** von der *Free Software Foundation* beschrieben, dessen Bedienung weitgehend der entspricht, wie sie auch für die unter anderen Unix-Systemen angebotenen Debugger des gleichen Namens (*gdb*) gilt. Der hier beschriebene **gdb** ist ein kommandozeilenorientierter Debugger, zu dem inzwischen mehrere graphische Debugger angeboten werden, wie z.B.:

xxgdb

ist eine graphische Oberfläche zum GNU-Debugger **gdb** und ermöglicht ein leichtes Debuggen von C- bzw. C++-Programmen, indem man im eingeblendeten Quellcode mit der Maus Breakpoints setzen kann, sich den Inhalt von Variablen und des Stacks anzeigen lassen kann usw.

ddd

ist wie **xxgdb** eine graphische Oberfläche zum GNU-Debugger **gdb** und ermöglicht ebenso ein leichtes Debuggen von C- bzw. C++-Programmen, indem man im eingeblendeten Quellcode mit der Maus Breakpoints setzen kann, sich den Inhalt von Variablen und des Stacks anzeigen lassen kann usw.

kdbg

ist eine vielversprechende beim KDE mitgelieferte graphische Oberfläche zum GNU-Debugger **gdb** und ermöglicht ebenso ein leichtes interaktives Debuggen von C- bzw. C++-Programmen.

Hier wird der kommandozeilenorientierte **gdb** kurz beschrieben, da die Kenntnis der grundlegenden **gdb**-Kommandos auch das Debuggen mit einem der eben erwähnten grafischen Oberflächen zum **gdb** erleichtert. Detailliertere Informationen zum **gdb** können mit **info gdb** erfragt werden.

gdb kann benutzt werden, um Fehler in Programmen zu finden. Er kann zum Debuggen von Programmen verwendet werden, die in C, C++ oder Modula-2 geschrieben wurden. In Zukunft wird wohl auch das Debuggen von Fortran-Programmen möglich sein.

22.3.1 Allgemeines

Um sich einen ersten Überblick über die von **gdb** angebotenen Kommandos zu verschaffen, empfiehlt es sich **gdb** zu starten und dann das **gdb**-Kommando **help** aufzurufen:

```
$ gdb
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type »show copying« to see the conditions.
There is absolutely no warranty for GDB; type »show warranty« for details.
GDB 4.16.patched (i486-unknown-linux --target i486-linux),
Copyright 1996 Free Software Foundation, Inc.
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
```

```
obscure -- Obscure features
internals -- Maintenance commands
```

Type »help« followed by a class name for a list of commands in that class.
 Type »help« followed by command name for full documentation.
 Command name abbreviations are allowed if unambiguous.

```
(gdb) help breakpoints
```

Making program stop at certain points.

List of commands:

```
awatch -- Set a watchpoint for an expression
rwatch -- Set a read watchpoint for an expression
watch -- Set a watchpoint for an expression
catch -- Set breakpoints to catch exceptions that are raised
break -- Set breakpoint at specified line or function
clear -- Clear breakpoint at specified line or function
delete -- Delete some breakpoints or auto-display expressions
disable -- Disable some breakpoints
enable -- Enable some breakpoints
thbreak -- Set a temporary hardware assisted breakpoint
hbreak -- Set a hardware assisted breakpoint
tbreak -- Set a temporary breakpoint
condition -- Specify breakpoint number N to break only if COND is true
commands -- Set commands to be executed when a breakpoint is hit
ignore -- Set ignore-count of breakpoint number N to COUNT
```

Type "help" followed by command name for full documentation.
 Command name abbreviations are allowed if unambiguous.

```
(gdb) quit
```

Dieses Ablaufbeispiel zeigt unter anderem, daß man sich detailliertere Informationen zu den einzelnen gdb-Kommando anzeigen lassen kann, indem man

```
help gdb-kommandoname
```

eingibt.

Das Verlassen des **gdb** erfolgt mit dem gdb-Kommando **quit**.

gdb kann – wie im vorherigen Ablaufbeispiel gezeigt – ohne Angabe von Argumenten oder Optionen aufgerufen werden. Üblicherweise ruft man den **gdb** jedoch mit einem Argument, dem Namen des zu debuggenden Programms, auf.

```
gdb progname
```

Zusätzlich zum Namen des zu debuggenden Programms kann auch eine *core-Datei* angegeben werden, die bei einem vorherigen Start des Programms vom System generiert wurde.

```
gdb progname core-datei
```

Der **gdb** bietet auch die Möglichkeit an, ein gerade ablaufendes Programm zu debuggen. Dazu muß der **gdb** sich mit diesem laufenden Prozeß, dessen Prozeß-ID *pid* ist, verbinden, was durch folgende Aufrufform erreicht wird:

```
gdb progname pid
```

Im **gdb** ist es nicht notwendig, die entsprechenden gdb-Kommandos vollständig auszusprechen, sondern diese können auch abgekürzt eingegeben werden. So kann z.B. statt dem Kommando **run** nur **r**, statt dem Kommando **help** nur **h** oder für das Kommando **quit** nur **q** eingegeben werden.

Um das letzte Kommando zu wiederholen, muß bloß die Return-Taste gedrückt werden, was das schrittweise Debuggen eines Programmes erheblich erleichtert.

Einige gdb-Kommandos können auch mit Formatangaben aufgerufen werden, um das Ausgabeformat von Werten festzulegen. Solche Formatangaben müssen mit / beginnend unmittelbar nach dem entsprechenden gdb-Kommando angegeben werden. Formatangaben bestehen aus vier Komponenten: /zfg, die im einzelnen folgende Bedeutung haben:

- ▶ Für das optionale **z** ist ein Wiederholungszähler (Voreinstellung 1) anzugeben.
- ▶ Für **f** ist ein Formatbuchstabe anzugeben: **o** (oktal), **x** (hexadezimal), **d** (dezimal), **u** (unsigned), **t** (binär), **f** (float), **a** (Adresse), **i** (instruction; Befehl), **c** (char) oder **s** (string).
- ▶ Für das optionale **g** ist eine Größe anzugeben: **b** (byte), **h** (halfword, 2 bytes), **w** (word, 4 bytes), **g** (giant, 8 bytes). Die Voreinstellung für die Größe ist ein zur Formatangabe passender Wert.

Hat man einmal für ein gdb-Kommando eine Formatangabe festgelegt, muß man diese bei einem erneuten Aufruf des Kommandos nicht wieder eingeben, da **gdb** dann immer die zuletzt definierte Formatangabe wiederverwendet.

Nachfolgend werden die am häufigsten benutzten gdb-Kommandos kurz vorgestellt:

attach, at

gdb soll sich mit einem gerade ablaufenden Prozeß verbinden. Dazu ist beim Aufruf von **attach** die PID des entsprechenden Prozesses anzugeben. Dieses Kommando hält den Prozeß an, an den sich **gdb** anhängen soll. Ein Loslösen von einem solchen Prozeß ist mit dem gdb-Kommando **detach** möglich.

backtrace, bt

zeigt den aktuellen Stack-Inhalt an.

break, b

setzt einen Breakpoint. Als Argument kann dabei ein Funktionsname, eine Zeilennummer der gerade aktiven Datei (Datei, deren Code gerade ausgeführt wird), ein Dateiname gefolgt von einer Zeilennummer (*dateiname:zeilennr*) oder sogar eine beliebige Adresse (**Adresse*) angegeben werden. **gdb** vergibt an jeden Breakpoint eine Nummer, welche er dem Benutzer auch mitteilt.

clear, cl

löscht einen Breakpoint. Als Argumente sind die gleichen Argumente wie bei **break** erlaubt.

condition, cond

legt für einen Breakpoint, dessen Nummer hier als erstes Argument anzugeben ist, eine Bedingung fest, die mit den weiteren Argumenten festgelegt wird, wie z.B.

condition 2 zgr == NULL

Die Ausführung des Programms wird dann an diesem Breakpoint nur noch angehalten, wenn die angegebene Bedingung zu diesem Zeitpunkt erfüllt ist.

continue, c

setzt die Ausführung eines angehaltenen Programms fort.

delete, d

löscht einen Breakpoint. Die Nummer des zu löschenden Breakpoints muß als Argument angegeben werden. Ist keine Nummer angegeben, werden alle Breakpoints gelöscht.

display, disp

zeigt den Wert eines Ausdrucks, der durch die angegebenen Argumente festgelegt wird, jedesmal an, wenn die Ausführung des Programms angehalten wird. Über eine zusätzliche Formatangabe kann man dabei noch festlegen, wie dieser Wert auszugeben ist. Jedem mit **display** anzuzeigenden Ausdruck wird von gdb eine Nummer zugeteilt, die er dem Benutzer mitteilt. Um das automatische Anzeigen eines Werts für einen Ausdruck wieder auszuschalten, muß **undisplay** mit dieser Nummer aufgerufen werden. Wird **undisplay** ohne jegliche Argumente aufgerufen, werden alle automatischen Anzeigen, die mit **display** eingerichtet wurden, ausgeschaltet.

help, h

gibt Hilfsinformationen aus. Ohne Argumente wird eine kurze Zusammenfassung der verfügbaren Hilfe angezeigt (siehe oben). Wird als Argument bei **help** ein gdb-Kommando angegeben, wird Hilfsinformation zu diesem speziellen gdb-Kommando ausgegeben.

list, l

zeigt die ersten 10 Zeilen um die aktuelle Zeile der Datei, deren Code gerade ausgeführt wird, an. Aufeinanderfolgende Aufrufe von **list** zeigen immer die nächsten 10 folgenden Zeilen an. Wird eine Zahl als Argument angegeben, dann werden 10 Zeilen um diese Zeilennummer herum angezeigt. Ein Rückwärtsblättern ist mit der Angabe einer negativen Zahl als Argument möglich. Mit der Angabe eines Dateinamens gefolgt von einer Zeilennummer (*dateiname:zeilennr*) werden 10 Zeilen aus der Datei *dateiname* um diese Zeilennummer herum angezeigt. Wird als Argument ein Funktionsname angegeben, werden die ersten 10 Zeilen dieser Funktion aufgelistet. Bei einem Argument der Form **Adresse* werden die Zeilen angezeigt, die den Code zu dieser Adresse umgeben.

next, n

setzt ein angehaltenes Programm fort, indem es den Code bis zur nächsten Zeile des Quellprogramms ausführt. Handelt es sich bei der aktuellen Zeile des Quellprogramms um eine Funktion, so wird diese vollständig ausgeführt. Soll diese Funktion schrittweise durchlaufen werden, muß das gdb-Kommando **step** verwendet werden.

nexti

setzt ein angehaltenes Programm fort, indem es den Code bis zum nächsten Assemblerbefehl ausführt. Funktionsaufrufe werden dabei vollständig ausgeführt. Soll eine Funktion schrittweise durchlaufen werden, muß das gdb-Kommando **stepi** verwendet werden.

print, p

gibt den Wert des Ausdrucks, der über die Argumente festgelegt wird, in der entsprechend festgelegten Form aus. Möchte man sich z.B. die Adresse in einem `int`-Zeiger (`int *zgr`) ausgeben lassen, muß man **print zgr** angeben. Möchte man aber den Wert sehen, auf den dieser Zeiger zeigt, so muß man **print *zgr** eingeben. Bei der Ausgabe von Strukturvariablen mit **print** werden die einzelnen Komponenten dieser Struktur angezeigt. Über eine zusätzliche Formatangabe kann man bei **print** noch festlegen, wie die entsprechenden Werte auszugeben sind.

run, r

startet das aktuelle Programm von Beginn an. Die Argumente für **run** sind die Argumente, die man beim Aufruf des Programms auf der Kommandozeile angeben würde. Dabei können die Shell-Metazeichen für Dateinamenexpansion (`*`, `[]` usw.) genauso angegeben werden wie Zeichen zur Ein-/Ausgabeumlenkung (`<`, `>`, `>>` usw.). Pipes dagegen sind hier nicht erlaubt. Wird **run** ohne Argumente aufgerufen, benutzt es die Argumente des letzten **run**-Aufrufs oder die Argumente, die mit dem letzten **set args**-Kommando festgelegt wurden.

set

weist Variablen Werte zu, wie z.B.:

```
set x = 5
```

```
set sum = 0
```

Um die Kommandozeilenargumente für das Programm, das man gerade mit gdb analysiert, nachträglich festzulegen oder aber neu zu setzen, steht das Kommando **set args ...** zur Verfügung. Das **set**-Kommando verfügt über eine Vielzahl von weiteren Subkommandos, die man sich mit **help set** anzeigen lassen kann.

step, s

setzt ein angehaltenes Programm fort, indem es den Code bis zur nächsten Zeile des Quellprogramms ausführt. Handelt es sich bei der aktuellen Zeile des Quellprogramms um eine Funktion, wird nur die erste Zeile dieser Funktion ausgeführt. Soll diese Funktion vollständig durchlaufen werden, muß das gdb-Kommando **next** verwendet werden.

stepi

setzt ein angehaltenes Programm fort, indem es den Code bis zum nächsten Assemblerbefehl ausführt. Handelt es sich bei der aktuellen Zeile des Quellprogramms um eine Funktion, so wird nur die erste Assembleranweisung dieser Funktion ausgeführt. Soll diese Funktion vollständig durchlaufen werden, muß das gdb-Kommando **nexti** verwendet werden.

quit, q

beendet den gdb.

whatis, wha

zeigt den Datentyp des als Argument übergebenen Ausdrucks an.

where, whe

zeigt den aktuellen Stack-Inhalt an.

x

zeigt den Inhalt von Speicher an. **x** verhält sich weitgehend wie **print**, kann jedoch nur den Inhalt von Adressen, die als Argumente anzugeben sind, in einer beliebigen Form anzeigen. Die Form, in der ein Speicherinhalt anzuzeigen ist, kann über eine zusätzliche Formatangabe festgelegt werden.

22.4 strace – Mitprotokollieren aller Systemaufrufe

Bei der Fehlersuche in Programmen kann das Kommando **strace**, das jeden Aufruf einer Systemfunktion mitprotokolliert, wertvolle Dienste leisten. Möchte man sich z.B. alle Aufrufe von Systemfunktionen beim Ablauf des Kommandos **date** anzeigen lassen, muß man nur **strace date** aufrufen.

```
$ strace date
execve("/bin/date", ["date"], [/* 51 vars */]) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40007000
mprotect(0x40000000, 20673, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
mprotect(0x8048000, 31120, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
stat("/etc/ld.so.cache", {st_mode=S_IFREG|0644, st_size=9353, ...}) = 0
open("/etc/ld.so.cache", O_RDONLY) = 3
mmap(0, 9353, PROT_READ, MAP_SHARED, 3, 0) = 0x40008000
close(3) = 0
stat("/etc/ld.so.preload", 0xbffff76c) = -1 ENOENT (No such file or directory)
open("/lib/libc.so.5", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\3"... , 4096) = 4096
mmap(0, 761856, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x4000b000
mmap(0x4000b000, 530945, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3, 0) = 0x4000b000
mmap(0x4008d000, 21648, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x81000) = 0x4008d000
mmap(0x40093000, 204536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40093000
close(3) = 0
```



```
$ strace
usage: strace [-dffhiqrsttvVxx] [-a column] [-e expr] ... [-o file]
             [-p pid] ... [-s strsize] [-u username] [command [arg ...]]
    or: strace -c [-e expr] ... [-O overhead] [-S sortby] [command [arg ...]]
-c -- count time, calls, and errors for each syscall and report summary
-f -- follow forks, -ff -- with output into separate files
-F -- attempt to follow vforks, -h -- print help message
-i -- print instruction pointer at time of syscall
-q -- suppress messages about attaching, detaching, etc.
-r -- print relative timestamp, -t -- absolute timestamp, -tt -- with usecs
-T -- print time spent in each syscall, -V -- print version
-v -- verbose mode: print unabbreviated argv, stat, termio[s], etc. args
-x -- print non-ascii strings in hex, -xx -- print all strings in hex
-a column -- alignment COLUMN for printing syscall results (default 40)
-e expr -- a qualifying expression: option=[!]all or option=[!]val1[,val2]...
    options: trace, abbrev, verbose, raw, signal, read, or write
-o file -- send trace output to FILE instead of stderr
-O overhead -- set overhead for tracing syscalls to OVERHEAD usecs
-p pid -- trace process with process id PID, may be repeated
-s strsize -- limit length of print strings to STRSIZE chars (default 32)
-S sortby -- sort syscall counts by: time, calls, name, nothing (default time)
-u username -- run command as username handling setuid and/or setgid
$
```

22.4.2 Beschreibung

strace verfolgt den Ablauf des Kommandos *command* mit, und gibt alle Systemaufrufe und Signale auf die Standardfehlerausgabe oder, wenn die Option **-o file** angegeben ist, in die Datei *file* aus.

Jede Zeile der Ausgabe enthält einen Systemaufruf, seine Argumente in Klammern und den Rückgabewert, wie z.B.:

```
open("brief.txt", O_RDONLY) = 3
```

Bei einem Fehler (meist der Rückgabewert -1) wird die Fehlernummer (als symbolischer Name) und die zugehörige Fehlermeldung mitausgegeben, wie z.B.:

```
open("brief2.txt", O_RDONLY) = -1 ENOENT (No such file or directory)
```

Des weiteren gilt folgendes:

- ▶ Signale werden mit ihren Signalnamen ausgegeben.
- ▶ Argumente werden, wenn möglich, in lesbarer Form ausgegeben.
- ▶ Bei Zeigern auf Strukturen werden nicht die in den Zeigern enthaltenen Adressen, sondern die einzelnen Komponenten der Strukturen (in geschweiften Klammern) ausgegeben, auf die diese Zeiger zeigen.
- ▶ Bei Zeigern auf Zeichenketten werden nicht die in den Zeigern enthaltenen Adressen, sondern die Zeichenketten (in Anführungszeichen) ausgegeben, auf die diese Zeiger zeigen.

- Nicht druckbare Zeichen werden, wie in C üblich, als Escape-Sequenzen ausgegeben.
- Während für Strukturen geschweifte Klammern verwendet werden, zeigen eckige Klammer Arrays an.

22.4.3 Optionen

Tabelle 22.4 gibt eine kurze Beschreibung zu den **strace**-Optionen.

Option	Bedeutung
-c	strace erstellt eine Zeitstatistik für jeden Systemaufruf und gibt diese am Ende aus.
-d	strace gibt eigene Debug-Informationen aus.
-f	Kreiert ein mit strace überwachter Prozeß mit fork Kindprozesse, werden die Systemaufrufe dieser Kindprozesse ebenfalls protokolliert.
-ff	Wenn die Option -o file angegeben ist, werden die Systemaufrufe jedes Kindprozesses in der Datei <i>file.pid</i> protokolliert, wobei <i>pid</i> die PID des jeweiligen Kindprozesses ist.
-h	strace gibt Help-Information aus.
-i	Am Anfang jeder Zeile wird der Befehlszähler (<i>Instruction-Pointer</i>) zum Zeitpunkt des Systemaufrufs ausgegeben.
-q	strace unterdrückt die Meldungen über das Anhalten und Freigeben von Prozessen. Dies geschieht automatisch, wenn die Ausgabe in eine Datei umgelenkt wird. Diese Option ist nur sinnvoll in Verbindung mit den Optionen -f oder -p pid .
-r	Zu jedem Systemaufruf wird der Zeitabstand zum vorherigen Systemaufruf in Sekunden und Mikrosekunden ausgegeben.
-t	Am Anfang jeder Zeile wird die aktuelle Uhrzeit im Format <i>hh:mm:ss</i> ausgegeben.
-tt	wie -t , nur daß noch die Mikrosekunden mitausgegeben werden.
-ttt	Am Anfang jeder Zeile wird die aktuelle Uhrzeit in Sekunden und Mikrosekunden (seit Beginn der Epoche) ausgegeben.
-T	Am Ende jeder Zeile wird die von diesem Systemaufruf benötigte Zeit ausgegeben.
-v	Alle komplexen Daten werden vollständig ausgegeben. Hierzu gehören z.B. Strukturen und Stringarrays. Normalerweise werden hierzu nur die ersten Komponenten oder Zeichen ausgegeben.
-V	strace gibt seine Versionsnummer aus.
-x	Nichtdruckbare Zeichen in Strings werden als hexadezimale Zahlen ausgegeben.

Tabelle 22.4: **strace**-Optionen

Option	Bedeutung
-xx	Alle Zeichen in Strings werden als hexadezimale Zahlen ausgegeben.
-a column	Die Rückgabewerte der Systemaufrufe werden in die Spalte <i>column</i> geschrieben. Voreinstellung ist -a 40 .
-e expr	Hier kann ein Ausdruck <i>expr</i> angegeben werden, der die Protokollierung der Systemaufrufe genauer festlegt. Der Ausdruck hat folgendes Format: [<i>typ</i>]=[<i>!</i>]<i>wert1</i>[,<i>wert2</i>]... Für <i>typ</i> kann trace, abbrev, verbose, raw, signal, faults, read oder write angegeben werden. Der Wert ist abhängig davon entweder ein Name oder eine Zahl. Der voreingestellte <i>typ</i> ist trace. Wird das Ausrufezeichen angegeben, so negiert es den Wert*. Die Angabe -e open, welche identisch zur Angabe -e trace=open ist, bewirkt, daß nur die <i>open</i>-Aufrufe von strace protokolliert werden. Bei der Angabe -e trace=!open dagegen werden alle Systemaufrufe außer <i>open</i> protokolliert. Als Spezialfälle kann für die Werte auch all oder none angegeben werden.
-e abbrev=wert	beeinflußt die Ausgabe der einzelnen Komponenten von großen Strukturen. Voreinstellung ist abbrev=all . Die Option -v entspricht abbrev=none .
-e faults	Falsche Speicherzugriffe werden mitausgegeben. Diese Option wird nur von System V angeboten.
-e raw=liste	gibt die Argumente der in <i>liste</i> angegebenen Systemaufrufe nicht symbolisch, sondern als hexadezimale Zahlen aus.
-e read=liste	gibt bei allen Leseoperationen, die auf die in <i>liste</i> angegebenen Filedeskriptoren stattfinden, die gelesenen Daten als Hexa- und ASCII-Dump aus. Um sich z.B. alle von den Filedeskriptoren 3 und 5 gelesenen Daten anzeigen zu lassen, muß -e read=3,5 angegeben werden.
-e signal=liste	Es werden nur die in <i>liste</i> angegebenen Signale ausgegeben, wenn sie auftreten. Die Voreinstellung ist -e signal=all . Möchte man sich z.B. alle auftretenden Signale außer SIGUSR1 anzeigen lassen, muß man -e signal=!SIGUSR1 angeben.
-e trace=liste	Es werden nur die Systemaufrufe protokolliert, die in <i>liste</i> angegeben sind. Voreinstellung ist -e trace=all .
-e trace=file	Es werden alle Systemaufrufe protokolliert, die zum Filesystem gehören. Dies ist z.B. unter Linux eine Abkürzung zu folgender Angabe -e trace=access,acct,chdir,chmod,chown,chroot,creat,execve,link,lstat,mkdir,mknod,mount,open,readlink,rename,rmdir,stat,statfs,swapon,symlink,truncate,umount,unlink,uselib,utime .
-e trace=ipc	Es werden alle Systemaufrufe protokolliert, die zur Interprozeßkommunikation (IPC von System V) gehören. Dies ist z.B. unter Linux eine Abkürzung zu -e trace=msgctl,msgget,msgrcv,msgsnd,semctl,semget,semop,shmat,shmctl,shmdt,shmget .

Tabelle 22.4: **strace**-Optionen

Option	Bedeutung
-e trace=network	Es werden alle Systemaufrufe protokolliert, die zur Netzwerkkommunikation gehören. Dies ist z.B. unter Linux eine Abkürzung zu -e trace=accept,bind,connect,getpeername,getsockname,getsockopt,listen,recv,recvfrom,recvmsg,send,sendmsg,sendto,setsockopt,shutdown,socket,socketpair .
-e trace=process	Es werden alle Systemaufrufe protokolliert, die zur Prozeßsteuerung gehören. Dies ist z.B. unter Linux eine Abkürzung zu -e trace=_exit,fork,waitpid,execve,wait4,clone .
-e trace=signal	Es werden alle Systemaufrufe protokolliert, die zur Signalbehandlung gehören. Dies ist z.B. unter Linux eine Abkürzung zu -e trace=pause,kill,signal,sigaction,siggetmask,sigsetmask,sigsuspend,sigpending,sigreturn,sigprocmask .
-e verbose=liste	Für alle Systemfunktionen, die in <i>liste</i> angegeben sind, wird bei Argumenten, die Zeiger auf Strukturen sind, der Inhalt der Strukturen und nicht nur der Zeigerwert (als hexadezimale Zahl) ausgegeben. Die Voreinstellung ist -e verbose=all .
-e write=liste	gibt bei allen Schreiboperationen, die auf die in <i>liste</i> angegebenen Filedeskriptoren stattfinden, die geschriebenen Daten als Hexa- und ASCII-Dump aus. Um sich z.B. alle auf die Filedeskriptoren 3 und 5 geschriebenen Daten anzeigen zu lassen, muß -e write=3,5 angegeben werden.
-o file	strace schreibt seine Ausgabe nicht auf die Standardfehlerausgabe, sondern in die Datei <i>file</i> . Ist die Option -ff angegeben, wird die Ausgabe in die Datei <i>file.pid</i> geschrieben, wobei für <i>pid</i> die PID des Prozesses eingesetzt wird.
-O overhead	Durch das Protokollieren der Systemaufrufe entsteht ein Overhead, der eine mit -c erstellte Statistik verfälscht. So kann der heuristisch vom Programm selbst ermittelte Wert korrigiert werden. Die Genauigkeit kann ein Aufrufer selbst überprüfen. Dazu muß er nur die Systemzeit, die das zu überwachende Programm verbraucht, mit dem Kommando time und der Option -c ermitteln und beide Werte vergleichen. Für <i>overhead</i> sind Mikrosekunden anzugeben.
-p pid	schaltet eine Überwachung für den gerade ablaufenden Prozeß mit der PID <i>pid</i> ein.
-s strsize	legt fest, daß für Strings <i>strsize</i> Zeichen auszugeben sind. Dateinamen zählen nicht zu solchen Strings, da diese immer vollständig ausgegeben werden. Die Voreinstellung ist: -s 32 .
-S sortby	sortiert die Statistik, die bei Angabe der Option -c erstellt wird, nach der Spalte <i>sortby</i> . Für <i>sortby</i> kann time , calls , name oder nothing (für unsortiert) angegeben werden. Die Voreinstellung ist: -S time .

Tabelle 22.4: **strace**-Optionen

- a. Das Ausrufezeichen ! hat in manchen Shells eine Sonderbedeutung. In diesen Shells muß sie durch das Voranstellen von \ ausgeschaltet werden.

22.5 Tools zum Auffinden von Speicherüberschreibungen und -lücken

Einer der schwerwiegendsten Fehler in C-Programmen ist das Schreiben in fremden Speicher (*buffer overruns*). Das Auffinden solcher Fehler ist meist sehr mühsam und zeitraubend. Ein weiterer häufig auftretender Fehler in C-Programmen sind sogenannte Speicherlücken (*memory leaks*), die dadurch entstehen, daß Speicher, der dynamisch allokiert wurde und nicht mehr benötigt wird, nicht wieder mit *free* freigegeben wird.

In diesem Kapitel werden einige Tools vorgestellt, die das Auffinden von Speicherüberschreibungen (*buffer overruns*) und Speicherlücken (*memory leaks*) wesentlich erleichtern.

Bei allen hier vorgestellten Tools wird das folgende Programm 22.1 (*schlimm.c*) verwendet, das viele Speicherüberschreibungen der unterschiedlichsten Art sowie eine Speicherlücke bei seinem Ablauf erzeugt.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  char g_array[5];
5
6  int
7  main(void)
8  {
9      char l_array[5];
10     char *dynam;
11
12     /*----- Kleine Speicherüberschreibung (hinten) ----*/
13     dynam = malloc(5);
14     strcpy(dynam, "12345");
15     printf("1: %s\n", dynam);
16     free(dynam);
17
18     /*----- Groessere Speicherüberschreibung (hinten) ----*/
19     dynam = malloc(5);
20     strcpy(dynam, "12345678");
21     printf("2: %s\n", dynam);
22
23     /*----- Speicherüberschreibung (vorne) ----*/
24     *(dynam - 1) = '\0';
25     printf("3: %s\n", dynam); /* Speicherluecke: kein free fuer dynam */
26
27     /*----- Speicherüberschreibung (lokales Array; vorne; hinten) -*/
28     strcpy(l_array, "12345");
29     printf("4: %s\n", l_array);
30     l_array[-1] = '\0';
31     printf("5: %s\n", l_array);
32
33     /*----- Speicherüberschreibung (globales Array; vorne; hinten) -*/
34     strcpy(g_array, "12345");
35     printf("6: %s\n", g_array);
36     g_array[-1] = '\0';
```

```
37     printf("7: %s\n", g_array);
38
39     exit(0);
40 }
```

Programm 22.1 (**schlimm.c**): Fehlerhaftes Programm, das Speicherüberschreibungen und eine Speicherlücke erzeugt

Nachdem man dieses Programm 22.1 (`schlimm.c`) kompiliert und gelinkt hat

```
cc-o schlimm schlimm.c
```

kann man es starten und ablaufen lassen.

```
$ schlimm
1: 12345
2: 12345678
3: 12345678
4: 12345
5: 12345
6: 12345
7: 12345
$
```

Obwohl das Programm 22.1 (`schlimm.c`) gespickt ist von Speicherüberschreibungen, läuft es überraschenderweise fehlerfrei ab. Es wäre nun ein Trugschluß, wenn man diese Probleme nicht ernst nehmen würde, da Speicherüberschreibungen oft dazu führen, daß Programme sich meist erst später nicht mehr richtig verhalten. Dies führt dann dazu, daß man den Fehler, der aus einer früheren Speicherüberschreibung resultierte, an der falschen Programmstelle sucht.

Nachfolgend werden nun Tools vorgestellt, mit denen man Speicherüberschreibungen und Speicherlücken lokalisieren kann. Es empfiehlt sich, diese Tools auch für scheinbar richtige Programme zu verwenden, um eventuell versteckte Speicherüberschreibungen, die zunächst keine Auswirkung haben – wie beim obigen Programm 22.1 (`schlimm.c`) – ausfindig zu machen.

22.5.1 efence – Electric Fence (Elektrischer Zaun)

Dieses einfach zu verwendende Tool hilft beim Auffinden von Speicherüberschreibungen. Das Tool *Electric Fence*, das bei vielen Linux-Distributionen mitgeliefert wird oder aber unter <ftp://sunsite.unc.edu/pub/Linux/devel/lang/c> zu finden ist, ist eine Bibliothek (`libefence.a`), die die normale **malloc**-Funktion der C-Bibliothek durch eine eigene ersetzt. Diese **malloc**-Funktion allokiert nicht nur wie die normale **malloc**-Funktion aus der C-Bibliothek den von einem Programm angeforderten Speicherplatz, sondern sie allokiert zusätzlich unmittelbar hinter diesem Speicherplatz einen Speicherbereich, auf den auf keinen Fall zugegriffen werden darf. Versucht ein Prozeß also hinter dem von ihm allokierten Speicherbereich lesend oder schreibend zuzugreifen, schickt der Systemkern automatisch das Signal `SIGSEGV` (*Segmentation fault* oder *Segmentation violation*) und bricht den entsprechenden Prozeß ab. Detailliertere Informationen zu Electric Fence können mit **man libefence** erfragt werden.

Hier soll nur auf die sehr einfache Benutzung von Electric Fence näher eingegangen werden. Um Electric Fence zu benutzen, muß man bei der Generierung eines Programms lediglich mit **-lefence** die Bibliothek `libefence.a` dazulinken. Das daraus resultierende Verhalten eines Programms soll nachfolgend für das Programm 22.1 (`schlimm.c`) gezeigt werden.

Zunächst kompilieren und linken wir das Programm 22.1 (`schlimm.c`)

```
cc -o schlimm schlimm.c -lefence
```

und starten dann das Programm `schlimm`:

```
$ schlimm
Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
1: 12345
Segmentation fault

$
```

Electric Fence deckt auf, daß das scheinbar fehlerfreie Programm 22.1 (`schlimm.c`) doch nicht ganz ohne Makel ist. An welcher Stelle im Programm nun genau das Problem liegt, kann man unter Verwendung des Debuggers **gdb** in Erfahrung bringen. Dazu sollte man das Programm beim Kompilieren und Linken mit Debug-Informationen versehen: Option **-g** oder Option **-ggdb** für gdb-spezielle Debug-Informationen.

```
cc -o schlimm schlimm.c -ggdb -lefence
```

Nun kann man das Programm debuggen.

```
$ gdb schlimm
GNU gdb 4.17.0.4 with Linux/x86 hardware watchpoint and FPU support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) run
Starting program: /home/hh/sysprog/kap22/schlimm

Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.
1: 12345

Program received signal SIGSEGV, Segmentation fault.
strcpy (dest=0x400c4ff8 "12345678", src=0x804a76d "12345678")
    at ../sysdeps/generic/strcpy.c:35
../sysdeps/generic/strcpy.c:35: No such file or directory.
(gdb) where
#0  strcpy (dest=0x400c4ff8 "12345678", src=0x804a76d "12345678")
    at ../sysdeps/generic/strcpy.c:35
#1  0x8048a00 in main () at schlimm.c:20
(gdb) quit
$
```

gdb teilt uns also nach dem Programmabsturz und dem folgenden Aufruf des **gdb**-Kommandos **where** mit, daß das Problem im Programm `schlimm.c` in der Zeile 20 liegt, die den zweiten Aufruf der Funktion **strcpy** enthält. Electric Fence konnte also nur das zweite Problem, bei dem eine größere Speicherüberschreibung stattfand, erkennen. Die erste leichte Speicherüberschreibung, die in Zeile 14 stattfand, entging Electric Fence.

Der Grund dafür liegt in der Speicherausrichtung (*memory alignment*), mit der die heute üblichen CPUs arbeiten. Diese Speicherausrichtung bewirkt, daß nicht einzelne Bytes, sondern immer ein Vielfaches der im jeweiligen Prozessor verwendeten Wortbreite bei **malloc** allokiert wird: 4 Byte bei 32-Bit-Prozessoren und 8 Byte bei 64-Bit-Prozessoren. Die von Electric Fence bereitgestellte **malloc**-Funktion hält sich standardgemäß an diese Konvention und liefert nur Adressen zurück die ein Vielfaches von `sizeof(int)` sind.

Für das Programm 22.1 (`schlimm.c`) bedeutet dies, daß beim ersten Aufruf von **malloc** (in Zeile 13) nicht nur die geforderten 5 Byte, sondern eben 8 Byte allokiert wurden. Dies bewirkte, daß die leichte Speicherüberschreibung in Zeile 14 nicht zu einem unerlaubten Zugriff führte, weshalb sie auch nicht erkannt werden konnte.

Um auch solche leichten Speicherüberschreibungen abfangen zu können, bietet Electric Fence eine eigene Environment-Variable `EF_ALIGNMENT`, mit der man die Speicherausrichtung der **malloc**-Funktion von Electric Fence festlegen kann. Setzt man diese Variable z.B. auf den Wert 3, so allokiert **malloc** nur noch Speicherbereiche, deren Anfangsadresse und deren Größe durch 3 teilbar ist, was normalerweise nicht sehr sinnvoll ist.

Um die Speicherausrichtung der **malloc**-Funktion von Electric Fence vollständig auszuschalten, muß man also nur die Environment-Variable `EF_ALIGNMENT` auf 1 setzen. Die dadurch bedingte Verlangsamung eines Programms sollte während der Testphase, in der das Beseitigen von Fehlern im Vordergrund steht, keine allzu große Rolle spielen.

Das Programm 22.1 (`schlimm.c`) soll nun nochmals dem Debugger **gdb** vorgelegt werden, wobei jedoch zuvor die Environment-Variable `EF_ALIGNMENT` auf 1 gesetzt wird.

```
$ export EF_ALIGNMENT=1
$ gdb schlimm
GNU gdb 4.17.0.4 with Linux/x86 hardware watchpoint and FPU support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) run
Starting program: /home/hh/sysprog/kap22/schlimm
```

Electric Fence 2.0.5 Copyright (C) 1987-1995 Bruce Perens.

```
Program received signal SIGSEGV, Segmentation fault.
strcpy (dest=0x400c4ffb "12345", src=0x804a760 "12345")
    at ../sysdeps/generic/strcpy.c:35
../sysdeps/generic/strcpy.c:35: No such file or directory.
```



```
(gdb) where
#0  strcpy (dest=0x400c4ffb "12345", src=0x804a760 "12345")
    at ../sysdeps/generic/strcpy.c:35
#1  0x80489c3 in main () at schlimm.c:14
(gdb) quit
$
```

Wie zu sehen ist, wurde nun auch die leichte Speicherüberschreibung in Zeile 14 erkannt.

Nachfolgend werden noch zwei weitere Environment-Variablen vorgestellt, mit denen man das Verhalten von Electric Fence beeinflussen kann.

EF_PROTECT_BELOW

Electric Fence kann nicht nur Speicherüberschreibungen erkennen, die am Ende eines allokierten Speicherbereichs auftreten, sondern auch solche, die vor einem allokierten Speicherbereich stattfinden. Dazu muß nur die Environment-Variable `EF_PROTECT_BELOW` auf 1 gesetzt werden. Die **malloc**-Funktion von Electric Fence allokiert dann zusätzlich vor dem angeforderten Speicherbereich noch ein kleines Stück Speicher, auf den keine lesenden oder schreibenden Zugriffe seitens des Prozesses erlaubt sind. In diesem Fall kann Electric Fence jedoch keine Speicherüberschreibungen am Ende des allokierten Speicherbereichs erkennen. Der Grund dafür wird hier nicht näher erläutert. Interessierte Leser seien auf die Manpage zu Electric Fence (**man libefence**) verwiesen.

EF_PROTECT_FREE

Wird diese Environment-Variable auf 1 gesetzt, gibt die von Electric Fence zur Verfügung gestellte **free**-Funktion den freizugebenden Speicherbereich nicht wirklich frei, sondern versieht diesen mit einem Zugriffsschutz, so daß nachfolgende Lese- oder Schreibzugriffe auf diesen Speicherbereich vom Systemkern als unzulässige Zugriffe erkannt werden und zur Beendigung des entsprechenden Prozesses führen. So kann man testen, ob ein Programm einen einmal freigegebenen Speicherbereich unerlaubterweise später nochmals benutzt, was auch ein häufiger Fehler in C-Programmen ist.

Electric Fence ist jedoch kein Allheilmittel, da es lediglich Speicherüberschreibungen auf dem Heap, also bei dem Speicher, der mit **malloc** allokiert wurde, erkennen kann. Speicherüberschreibungen in statisch allokierten Puffern (Arrays) werden von Electric Fence ebenso wenig erkannt wie Speicherlücken (*memory leaks*).

Auf die eventuelle Verlangsamung von Programmen, die Electric Fence benutzen, wurde bereits hingewiesen.

Ein weiterer Nachteil von Electric Fence ist sein sehr hoher Speicherverbrauch, da er für jeden **malloc**-Aufruf mindestens eine Page (Speicherseite) reservieren muß, um Speicherbereiche mit unterschiedlichen Zugriffsrechten (für erlaubte und verbotene Zugriffe) einrichten zu können. Dies kann vor allen Dingen bei Programmen mit sehr vielen **malloc**-Aufrufen, die jeweils nur kleine Speicherbereiche anfordern, zu einem Speicherbedarf führen, der um das Hundert- oder sogar Tausendfache größer ist als der des gleichen Programmes, das mit der von der C-Bibliothek bereitgestellten **malloc**-Funktion arbeitet⁴.

22.5.2 checkergcc – C-Compiler zum Auffinden von Speicherüberschreibungen und -lücken

Das Programm **checkergcc**, das unter `ftp://sunsite.unc.edu/pub/Linux/devel/lang/c` zu finden ist, ist eine Alternative zum GNU-C-Compiler **gcc**. Es kompiliert und linkt nicht nur das entsprechende Programm, wie es **gcc** tut, sondern es fügt noch zusätzlichen Code zu einem Programm hinzu, der beim Auffinden von Speicherüberschreibungen und -lücken mithilft. Nachdem man z.B. das Programm 22.1 (`schlimm.c`) mit **checkergcc** kompiliert und gelinkt hat

```
checkergcc -o schlimm schlimm.c
```

kann man es starten und erhält eine äußerst umfangreiche Liste der darin enthaltenen Fehler. Nachfolgend ist nur ein Teil dieser Ausgabe gezeigt.

```
$ schlimm
.....
From Checker (pid:06043): (byh) block bounds violation in the heap.
When Writing 1 byte(s) at address 0x08078ff9, inside the heap (sbrk).
0 bytes after a block (start: 0x8078ff4, length: 5, mdesc: 0x0).
The block was allocated from:
    pc=0x08051ac0 in malloc() at ./1-malloc/malloc.c:251
    pc=0x080481dc in main() at schlimm.c:13
    pc=0x0804810c in _start() at :0
Stack frames are:
    pc=0x080649a0 in strcpy() at strcpy.c:35
    pc=0x08048211 in main() at schlimm.c:14
    pc=0x0804810c in _start() at :0
.....
.....
$
```

Diese Ausgabe informiert darüber, an welcher Stelle (Zeile 13) der Speicher allokiert wurde, der in Zeile 14 mit **strcpy** überschritten wurde. In der weiteren hier nicht gezeigten Ausgabe werden noch weitere Überschreibungen gefunden.

Über die Environment-Variable **CHECKEROPTS** kann die Form der Überprüfung eines mit **checkergcc** kompilierten Programms gesteuert werden. Um sich die möglichen Angaben in **CHECKEROPTS** anzeigen zu lassen, muß **CHECKEROPTS** vor dem Aufruf des entsprechenden Programms auf `--help` gesetzt werden.

```
$ export CHECKEROPTS=--help
$ schlimm
This program has been compiled with 'checkergcc' or 'checkerg++'.
Checker is a memory access detector.
Checker is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
```

-
4. Im Falle von Speicherengpässen beim Arbeiten mit Electric Fence wird man nicht umhinkönnen, den Swap-Bereich zu vergrößern.

General Public License for more details.
 For more information, set CHECKEROPTS to '--help'
 Checker reads options from the environment variable CHECKEROPTS
 Options are:

```

-s --silent          Do not print the welcome message.
-q --quiet           Same as --silent.
-a --abort           Abort on startup.
-h --help            Print this message.
-n --nosymtab        Do not use symbol table.
-o=file --output=file Redirect Checker's output to 'file'.
-i=file --image=file Set the image file (Checker finds it for you)
-p --profile         Display profile information.
-d=xx --disable=xxx  Disable an address or a range of addresses.
-S --stop            Stop just before main.
-D=end --detector=end Do leak detection at the end of the program.
-m=a --malloc0=a     Set the behavior of malloc(0).
-v --verbose         Verbose.
-u=end --inuse=end    Do inuse at the end.
-Wsignal=sig         Emit a warning when 'sig' is received.
-Wno-signal=sig       Do not emit a warning when 'sig' is received.
--aged-queue=n       Set the size of the aged block queue.
--leak-size-threshold=n Minimum size of a leak to be displayed.
--bytes-per-state=n  Number of bytes handled by a bitmap state.
--no-signals         Disable the signal manager.
--Wsbkr --Wno-sbrk   Emit a warning if sbrk() is called.
-A --Wmemalign       Emit a warning if the alignment arg isn't a power of 2.
-t --trace           Trace calls to malloc, free...
-w --weak-check-copy Only copy the bitmap for memcpy, memmove, bcopy

```

\$

Eine hilfreiche Angabe für CHECKEROPTS ist dabei --detector=end. Hiermit legt man fest, daß bei der Ausführung des mit **checkergcc** kompilierten Programms ein Detektor für Speicherlücken zu starten ist.

```

$ export CHECKEROPTS=--detector=end
$ schlimm
.....
.....
From Checker (pid:06120): (gar) garbage detector results.
There is 1 leak and 0 potential leak(s).
Leaks consume 5 bytes (0 KB) / 131562 KB.
( 0.00% of memory is leaked.)
Found 1 block(s) of size 5.
Block at ptr=0x807a35c
    pc=0x08051a00 in malloc_1() at ./l-malloc/malloc.c:211
    pc=0x0804826d in main() at schlimm.c:19
    pc=0x0804810c in _start() at :0
$

```

Es wurde also genau die Stelle (Zeile 19) gefunden, an der Speicher allokiert wird, der später nicht mehr freigegeben wird.

22.5.3 mpr und mcheck – Auffinden von Speicherlücken und -überschreibungen

Zum Auffinden von Speicherlücken wird die Bibliothek **mpr** (`libmpr.a`) angeboten, die unter `ftp://sunsite.unc.edu/pub/Linux/devel/lang/c` zu finden ist. Wird diese Bibliothek beim Linken eines Programms (`-lmpr`) angegeben, dann werden beim Ablauf des entsprechenden Programms alle **malloc**- und **free**-Aufrufe protokolliert. Wenn das Programm beendet wird, wird automatisch geprüft, ob zu allen **malloc**-Aufrufen auch entsprechende **free**-Aufrufe stattgefunden haben.

Um auch Speicherüberschreibungen entdecken zu können, enthält **mpr** eine eigene Version der **malloc**-Funktion, die Unterstützung für das Debuggen von Speicherallokierungen bietet. Diese Unterstützung wird durch den Aufruf der Funktion **mcheck** aktiviert.

Diese eigene **malloc**-Version von **mpr** arbeitet ähnlich wie Electric Fence, nur daß hier die Überprüfung von Speicherüberschreibungen nicht der Hardware überlassen wird, sondern statt dessen bestimmte Bytefolgen vor und hinter dem allokierten Speicherbereich zusätzlich abgelegt werden. Die von **mpr** bereitgestellte **free**-Funktion prüft dann diese Bytefolgen und kann so feststellen, ob sie manipuliert wurden, was auf eine Speicherüberschreibung hinweist. In einem solchen Fall ruft **free** die Funktion **abort** auf, um das Programm zu beenden.

Startet man ein Programm, das mit `-lmpr` gelinkt wurde, werden alle Speicherbereiche angezeigt, die überschritten wurden. Dies gilt jedoch nur für die Speicherbereiche, die auch mit **free** wieder freigegeben wurden. Anders als Electric Fence kann **mpr** jedoch nur melden, daß eine Speicherüberschreibung stattgefunden hat, aber nicht an welcher Stelle.

Nachdem man das Programm 22.1 (`schlimm.c`) wie folgt kompiliert und gelinkt hat.

```
cc -o schlimm schlimm.c -ggdb -lmpr
```

kann man es unter **gdb** ablaufen lassen.

```
$ gdb schlimm
.....
(gdb) run
Starting program: schlimm
1: 12345
mcheck: memory clobbered past end of allocated block
```

```
Program received signal SIGABRT, Aborted.
0x8055811 in kill ()
(gdb) where
#0  0x8055811 in kill ()
#1  0x8055002 in raise (sig=6) at raise.c:27
#2  0x804fbb7 in abort () at abort.c:61
#3  0x804a2b8 in mabort ()
#4  0x804a002 in checkhdr ()
#5  0x804a038 in freehook ()
#6  0x8049915 in free ()
#7  0x804817e in main () at schlimm.c:16
```

```
#8 0x80480ee in __crt_dummy__ ()
(gdb) quit
$
```

Diese Ausgabe informiert also darüber, daß das Problem in Zeile 16 des Programms `schlimm.c` liegt. Der Fehler wurde also beim ersten Aufruf der Funktion **free** entdeckt, was darauf hindeutet, daß es ein Problem mit dem Speicherbereich gibt, auf den `dynam` zeigt.

Wie Electric Fence und **checkergcc** ist auch **mpr** nicht in der Lage, Speicherüberschreibungen in lokalen oder globalen Variablen zu finden, sondern nur bei Speicherbereichen, die dynamisch mit **malloc** auf dem Heap allokiert werden.

Auch wenn **mpr** zum Auffinden von Speicherüberschreibungen eingesetzt werden kann, liegt seine eigentliche Stärke im Auffinden von Speicherlücken. Dazu muß man die beiden Environment-Variablen `MPRPC` und `MPRFI` entsprechend setzen.

`MPRPC` wird von **mpr** benötigt, um die Folge von Funktionsaufrufen richtig abzuarbeiten, wenn in die Log-Datei (Protokolldatei) geschrieben wird. Das Setzen von `MPRPC` erfolgt mit:

```
MPRPC=`mprpc progname`
```

`MPRFI` legt fest, durch welches Programm die Log-Datei aufzubereiten ist. Für kleine Programme setzt man `MPRFI` meist mit `cat >mpr.log`, während man bei größeren Programmen die Log-Datei mit `gzip >mpr.log` komprimieren läßt.

Um beim Programm `schlimm.c` einen Programmabbruch – bedingt durch die Speicherüberschreibungen – zu vermeiden, wird `schlimm.c` hier nach `halbschlimm.c` kopiert und dort werden die folgenden Änderungen vorgenommen.

```
$ diff schlimm.c halbschlimm.c
13c13
<   dynam = malloc(5);
---
>   dynam = malloc(6);
19c19
<   dynam = malloc(5);
---
>   dynam = malloc(9);
$
```

Nun können wir uns zum Programm `halbschlimm.c` eine Log-Datei erstellen lassen.

```
$ cc -o halbschlimm halbschlimm.c -lmpr -ggdb
$ MPRPC='mprpc halbschlimm' MPRFI="cat >mpr.log" ./halbschlimm
1: 12345
2: 12345678
3: 12345678
4: 12345
5: 12345
6: 12345
```

```
7: 12345
$ ls -l mpr.log
-rw-r--r--  1 hh      topgroup      130 Feb  2 18:32 mpr.log
$
```

Ist die Log-Datei einmal erzeugt, gibt es mehrere mpr-Werkzeuge zum Analysieren dieser Log-Datei. Nachfolgend werden zwei wichtige mpr-Tools kurz vorgestellt.

mpr [*option(en)*] *progname* <*logfile*>

Hierdurch werden die Adressen aus der Log-Datei in Funktionsnamen und Positionen im Quellprogramm konvertiert. Die Voreinstellung ist, daß **mpr** den Funktionsnamen und die Zeilennummern anzeigt, in denen **malloc**-Aufrufe stattfanden. Bei Angabe der Option **-f** werden außerdem noch die Dateinamen und bei der Option **-l** die Zeilennummern in der jeweiligen Datei mit angezeigt. Die Ausgabe erfolgt wieder in dem Format, das für Log-Dateien benutzt wird, und kann somit als Eingabe für alle anderen mpr-Tools verwendet werden.

mprlk <*logfile*>

Dieses Tool überprüft unter zuhelfenahme der Log-Datei, ob Speicherbereiche existieren, die niemals freigegeben wurden, und schreibt eine neue Log-Datei auf die Standardausgabe, die nur Allokierungen enthält, zu denen keine Freigabe erfolgte.

Zum Auffinden der Speicherlücke im Programm `halbschlimm.c` ist dann folgender Aufruf notwendig.

```
$ mprlk <mpr.log | mpr -f -l halbschlimm
m:main(halbschlimm.c,19):9:134631432
$
```

22.6 ar – Erstellen und Verwalten von statischen Bibliotheken

Das Kommando **ar** ermöglicht es, mehrere Dateien in einer sogenannten statischen Archivbibliothek (*archiv_datei*) unterzubringen. Ebenso können mit **ar** neue Dateien in einer bereits erstellten Archivbibliothek aufgenommen bzw. aus ihr extrahiert oder entfernt werden.

22.6.1 Aufrufsyntax

```
ar [-V] [-s]schlüssel [posname] archiv_datei [datei(en)]
```

Eine statische Archivbibliothek enthält am Anfang eine sogenannte Symboltabelle, die Informationen über die in der Bibliothek enthaltenen Dateien bereitstellt, um einen möglichst effizienten Zugriff auf die jeweiligen Dateien durch die entsprechenden Tools zu ermöglichen, wie z.B. dem Linker **ld**, dem man wohl eine Bibliothek von Objektdateien auf der Kommandozeile übergibt. Eine Symboltabelle wird nur dann von **ar** erstellt, wenn sich wenigstens eine Datei in der Bibliothek befindet.

Die Angaben auf der Kommandozeile bedeuten im einzelnen:

<i>ar -V</i>	Dieser Aufruf bewirkt die Ausgabe der Versionsnummer von ar auf die Standardfehlerausgabe.
<i>schlüssel</i>	legt die in einem Archiv durchzuführende Operation fest.
<i>posname</i>	muß der Name einer Datei aus dem Archiv sein. Hiermit kann eine Position innerhalb eines Archivs festgelegt werden.
<i>archiv_datei</i>	ist der Name des entsprechenden Archivs.
<i>datei(en)</i>	legt die zu bearbeitenden Dateien fest.

22.6.2 Schlüsselangabe

Ein *schlüssel* setzt sich aus zwei Teilen zusammen:

<i>funktion</i>	legt die auszuführende Aktion fest. <i>funktion</i> muß immer angegeben sein, wobei davor ein – (Querstrich) stehen kann oder auch nicht.
<i>zusatz</i>	läßt Zusatzangaben zu der auszuführenden Aktion zu.

funktion

Für *funktion* muß genau einer der Buchstaben aus Tabelle 22.5 angegeben werden, wobei dem jeweiligen Buchstaben optional ein Querstrich (-) voranstehen darf.

<i>Buchstabe</i>	<i>Bedeutung</i>
d	(d ele ~~t~~ e) löscht die angegebenen <i>datei(en)</i> aus dem Archiv <i>archiv_datei</i> . Ist als <i>zusatz v</i> angegeben, werden die Namen aller gelöschten Dateien angezeigt.
r	(r eplac ~~e~~) ersetzt im Archiv die angegebenen <i>datei(en)</i> . Wenn nach r der <i>zusatz u</i> angegeben ist, werden nur die Dateien im Archiv ersetzt, die seit ihrer letzten Archivierung verändert wurden. Ist nach r einer der <i>zusätze a</i> oder b oder i angegeben, so muß der <i>posname</i> angegeben sein; in diesem Fall werden neue Dateien nach (a) bzw. vor (b,i) <i>posname</i> eingefügt. In allen anderen Fällen werden neue Dateien am Ende des Archivs aufgenommen.
q	(q uick app ~~e~~ nd) hängt die angegebenen <i>datei(en)</i> am Ende des Archivs an. Hierbei wird nicht geprüft, ob von den angegebenen <i>datei(en)</i> bereits welche im Archiv vorhanden sind. Die <i>Zusätze a, b</i> oder i haben hier keine Auswirkung. Bei q wird die Symboltabelle nicht aktualisiert. Zu ihrer Aktualisierung müßte dann nachträglich ar r archiv_datei bzw. ranlib archiv_datei aufgerufen werden.
t	(t able) gibt ein Inhaltsverzeichnis für das Archiv <i>archiv_datei</i> aus. Sind keine <i>datei(en)</i> angegeben, so wird ein Inhaltsverzeichnis für das gesamte Archiv ausgegeben. Sind <i>datei(en)</i> angegeben, so werden nur diese, falls im Archiv vorhanden, aufgelistet.

Tabelle 22.5: Mögliche *funktion*-Angaben

Buchstabe	Bedeutung
p	(print) gibt die angegebenen <i>datei(en)</i> aus dem Archiv <i>archiv_datei</i> auf der Standardausgabe aus. Sind keine <i>datei(en)</i> angegeben, werden alle im Archiv enthaltenen Dateien auf der Standardausgabe ausgegeben.
m	(move) verlagert die angegebenen <i>datei(en)</i> an das Ende des Archivs <i>archiv_datei</i> . Ist nach m einer der <i>zusätze</i> a oder b oder i angegeben, so muß der <i>posname</i> angegeben sein; in diesem Fall werden die Dateien nicht am Archivende, sondern nach (a) bzw. vor (b,i) <i>posname</i> eingefügt.
x	(extract) extrahiert die angegebenen <i>datei(en)</i> aus dem Archiv. Sind keine <i>datei(en)</i> angegeben, so werden alle Dateien aus dem Archiv extrahiert. Extrahieren bedeutet hier, daß die entsprechenden Dateien aus dem Archiv in das Working Directory kopiert werden. Der Inhalt des Archivs wird bei dieser Option niemals verändert.

Tabelle 22.5: Mögliche *funktion*-Angaben**zusatz**

Anders als bei *funktion*, wo nur eine Angabe erlaubt ist, können bei der *zusatz*-Angabe mehrere der in Tabelle 22.6 gezeigten Buchstaben gleichzeitig angegeben werden.

Buchstabe	Bedeutung
v	(verbose) Normalerweise gibt ar keine speziellen Meldungen aus. Diese <i>zusatz</i> -Angabe bewirkt, daß beim Erzeugen eines neuen Archivs für jede betroffene Datei eine kurze Information ausgegeben wird. Wird v bei der <i>funktion</i> t angegeben, so wird eine umfangreichere Information zu den entsprechenden Dateien ausgegeben. Wird v bei der <i>funktion</i> x angegeben, so wird für jede extrahierte Datei deren Name gemeldet.
c	(create) unterdrückt die Meldung, die normalerweise beim Anlegen eines Archivs ausgegeben wird.
l	(local) veranlaßt ar , temporäre Dateien nicht in <i>/tmp</i> , sondern im Working-Directory abzulegen. Diese Option ist veraltet, da das neue ar keine temporären Dateien mehr anlegt.
s	(symbol table) bewirkt, daß die Symboltabelle für ein Archiv neu erstellt wird, selbst wenn ar nicht mit einem Kommando aufgerufen wird, das den Inhalt des Archivs ändert. Diese Option ist zur Wiederherstellung der Symboltabelle nützlich, wenn diese zuvor mit strip entfernt wurde. Der Aufruf von ar r archiv_datei ist äquivalent zum Aufruf ranlib archiv_datei
u	(update) Wenn u mit der <i>funktion</i> r verwendet wird, so werden nur die Dateien ersetzt, die seit ihrer letzten Archivierung modifiziert wurden.
a	(after) wenn a zusammen mit einer der <i>funktionen</i> r oder m angegeben wird, so werden die <i>datei(en)</i> nach der mit <i>posname</i> spezifizierten Datei im Archiv eingefügt.

Tabelle 22.6: Mögliche *zusatz*-Angaben

Buchstabe	Bedeutung
b	(b efore) wenn b zusammen mit einer der <i>funktionen</i> r oder m angegeben wird, so werden die <i>datei(en)</i> vor der mit <i>posname</i> spezifizierten Datei im Archiv eingefügt.
i	(i nsert) wenn i zusammen mit einer der <i>funktionen</i> r oder m angegeben wird, so werden die <i>datei(en)</i> vor der mit <i>posname</i> spezifizierten Datei im Archiv eingefügt.
o	(o riginal) Beim Extrahieren von Dateien werden deren ursprünglichen Zeitmarken übernommen. Normalerweise erhalten extrahierte Dateien als Zeitmarken den Zeitpunkt des Extrahierens.

Tabelle 22.6: Mögliche *zusatz*-Angaben

Hinweis

Wenn bei *datei(en)* dieselbe Datei zweimal angegeben ist, kann sie auch zweimal im Archiv aufgenommen werden.

Archivdateien sollten immer das Suffix *.a* haben.

Das Kommando **ar** bewirkt keine nennenswerte Speicherplatzeinsparungen, da die entsprechenden Dateien nicht komprimiert werden.

Manche UNIX-Systeme fordern, daß die Symboltabelle einer Archivdatei eventuell zuerst mit **ranlib** *archiv_datei* bzw. **ar s** *archiv_datei* aktualisiert werden muß, bevor sie von **ld** bearbeitet werden kann.

Zur Erstellung und Pflege von Archiven können auch die beiden Kommandos **tar** und **cpio** verwendet werden. Es ist aber wichtig zu wissen, daß alle drei Kommandos verschiedene Archivformate benutzen, und somit ein einmal erstelltes Archiv auch nur wieder mit dem gleichen Kommando bearbeitet werden kann.

22.6.3 Typische Anwendungen

- Das Kommando **ar** wird verwendet, um eine Archivbibliothek von kompilierten C-Funktionen anzulegen, die dem Linker **ld** zum Einbinden der benötigten Funktionen vorgelegt wird. **ld** wird zwar automatisch von **cc** bzw. **gcc** aufgerufen, kann jedoch auch direkt aufgerufen werden.
- **ar** kann auch verwendet werden, um miteinander verwandte Textdateien (wie z.B. C-Quellprogramme oder Briefe) in einem Archiv unterzubringen. Dies führt zu einer erheblichen Reduzierung der Dateien in einem Directory und dient so der Übersichtlichkeit.
- **ar** wird häufig auch verwendet, wenn eine große Zahl von Dateien kopiert werden muß. In diesem Fall werden alle zu kopierenden Dateien zunächst in einem Archiv abgelegt, bevor das gesamte Archiv kopiert wird.

Beispiel

```
$ ar -rcsv libgraphik.a linie.o kreis.o bogen.o rahmen.o
ar: creating libgraphik.a
q - linie.o
q - kreis.o
q - bogen.o
q - rahmen.o
$
```

Mit dem obigen Kommando wird eine Archivbibliothek *libgraphik.a* erstellt, die vier Objektdateien enthält. Der Zusatz **v** veranlaßt die Ausgabe aller Namen der Dateien, die im Archiv aufgenommen werden.

```
ar -q libgraphik.a punkt.o
```

Die Objektdatei *punkt.o* wird am Archivende eingefügt, ohne daß geprüft wird, ob diese Datei bereits im Archiv vorhanden ist oder nicht.

```
ar d libgraphik.a rahmen.o
```

Die Datei *rahmen.o* wird aus dem Archiv *libgraphik.a* entfernt.

```
ar -r libgraphik.a kreis.o
```

Die Datei *kreis.o* im Archiv wird durch ein neues *kreis.o* ersetzt.

```
$ ar -t libgraphik.a
linie.o
kreis.o
bogen.o
punkt.o
$
```

Es wird der Inhalt der Archivbibliothek *libgraphik.a* aufgelistet.

```
ar -x libgraphik.a linie.o
```

Die Datei *linie.o* wird aus der Archivbibliothek *libgraphik.a* in das Working-Directory kopiert. Der Inhalt der Archivbibliothek bleibt bei diesem Aufruf unverändert.

```
ar -t /usr/lib/libc.a | sort | more
```

Mit diesem Aufruf kann man sich alle Objektdateien aus der C-Standardbibliothek auflisten lassen. Um sich alle Funktionen aus einer Bibliothek, wie z.B. der C-Standardbibliothek, auflisten zu lassen, muß das Kommando **nm** (**n**ame **m**apper) verwendet werden, wie z.B.:

```
$ nm /usr/lib/libc.a | more
.....
.....
assert.o:
    U _IO_stderr_
```

```

00000000 T __assert_fail
00000004 C __assert_program_name
          U abort
          U fflush
          U fprintf
          U strchr

```

```

setenv.o:
          U __environ
          U __errno_location
          U free
          U malloc
          U memcpy
00000000 T setenv
000001e4 T unsetenv

```

```

ftime.o:
          U __gettimeofday
00000000 T ftime
.....
.....
$

```

Das Kürzel `T` bedeutet dabei, daß die Funktion hier definiert ist, während `U` anzeigt, daß diese Funktion lediglich aufgerufen wird.

22.7 Dynamische Bibliotheken

Dynamische Bibliotheken weisen einige Vorteile gegenüber statischen Bibliotheken auf:

- ▶ Der ausführbare Code einer dynamischen Bibliothek wird vom System nur einmal in den Speicher geladen, so daß alle Prozesse, die diese dynamische Bibliothek benutzen, den gleichen Code benutzen. Deswegen sollte man Code, der von mehreren Programmen benutzt werden kann, in eine dynamische Bibliothek packen. Dies spart viel Speicher beim gleichzeitigen Ablauf dieser Programme.
- ▶ Diese Einsparung an Speicher bringt natürlich auch Geschwindigkeitsvorteile mit sich, da dadurch weniger Paging (Ein- und Auslagern von Speicherseiten) stattfindet.
- ▶ Da der Code einer dynamischen Bibliothek beim Linken nicht in das entsprechende Programm eingefügt wird, sind die aus dem Linken resultierenden Programme (ausführbaren Dateien) kleiner als solche, die mit statischen Bibliotheken gelinkt werden. Dies spart zum einen Speicherplatz auf der Festplatte, zum anderen führt dies aber auch zu schnelleren Programmen, da das Laden von kleineren Programmen in den Arbeitsspeicher natürlich auch weniger Zeit beansprucht.
- ▶ Werden Fehler in einer dynamischen Bibliothek behoben oder eben nur erforderliche Änderungen (wie z.B. Code-Optimierungen) an ihr vorgenommen, so erfordert dies keine neue Generierung der Programme, die diese Bibliothek benutzen.

Bei statischen Bibliotheken dagegen müßte man alle Programme, die diese Bibliothek benutzen, neu kompilieren und linken.

Allerdings haben dynamische Bibliotheken auch einige kleine Nachteile gegenüber statischen Bibliotheken, die hier nicht verschwiegen werden sollen:

- ▶ Ein mit dynamischen Bibliotheken gelinktes Programm ist für sich allein nicht ablauffähig, da es auch immer die zugehörigen dynamischen Bibliotheken benötigt. Das bedeutet, daß bei Auslieferung dieses Programms an einen Kunden oder an einen anderen Benutzer niemals vergessen werden darf, die zugehörigen dynamischen Bibliotheken mitzuliefern, da sonst dieses Programm nicht ablauffähig sein wird; es sei denn, es handelt sich um dynamische Bibliotheken, die allgemein vom jeweiligen System angeboten werden.
- ▶ Da die von einem Programm benutzten dynamischen Bibliotheken beim Programmstart erst gesucht und geladen werden müssen, kann dies beim ersten Laden einer dynamischen Bibliothek zu einem zusätzlichen Zeitaufwand führen, den statische Bibliotheken nicht haben. Dieser Nachteil gilt jedoch nur für das erstmalige Laden einer dynamischen Bibliothek. Da jedoch meist die entsprechenden dynamischen Bibliotheken schon für einen anderen Prozeß in den Hauptspeicher geladen wurden, trifft dieser Nachteil beim Start eines Programms für einen Großteil von Programmen, die mit dynamischen Bibliotheken arbeiten, nicht mehr zu.

Mit dem heute auf nahezu allen Unix/Linux-Systemen verfügbaren Binärformat **ELF** (*Executable and Linking Format*) ist das Erzeugen und Benutzen von dynamischen Bibliotheken weitgehend standardisiert worden.

22.7.1 Entwerfen von dynamischen Bibliotheken

Ein wichtiger Grundsatz beim Entwurf von dynamischen Bibliotheken ist, daß diese immer abwärtskompatibel sein sollten. Dies bedeutet, daß ein Programm, das mit einer älteren Version dieser dynamischen Bibliothek gelinkt wurde, auch mit der neuen Version weiterhin lauffähig sein sollte. Dieser Grundsatz muß nur dann nicht eingehalten werden, wenn eine völlig neue Version (*major version*) zu einer dynamischen Bibliothek entwickelt wird. Man spricht dann von einem Versionsprung.

Jede dynamische Bibliothek hat einen speziellen Namen, den sogenannten *soname*, der den eigentlichen Namen der Bibliothek sowie die Versionsnummer beinhaltet. Solange dynamische Bibliotheken abwärtskompatibel sind, also ihre Schnittstellen sich nicht ändern, sollte sich nur eine der Nummern (*minor number*) hinter der Hauptversionsnummer (*major number*) ändern.

Als Beispiel möge die C-Bibliothek von Linux dienen, die Abwärtskompatibilität für alle Unterversionen mit der gleichen Hauptversionsnummer garantiert. Da z.B. alle C-Bibliotheken mit der Hauptversionsnummer 5 abwärtskompatibel sind, benutzen sie alle den gleichen *soname* `libc.so.5`, der lediglich ein symbolischer Link auf die eigentliche, aktuelle dynamische Bibliothek ist:

```
$ ls -l /lib/libc.so.5
lrwxrwxrwx 1 root root 14 Jun 2 1998 /lib/libc.so.5 -> libc.so.5.4.44
$
```

Beim Namen `/lib/libc.so.5.m.r` gibt `m` die Unterversionsnummer und `r` die Releasenummer an. Programme, die mit der dynamischen C-Bibliothek gelinkt werden, linkt man nun üblicherweise nicht direkt mit der aktuellen C-Bibliothek `/lib/libc.so.5.m.r`, sondern eben mit `/usr/lib/libc.so`, was immer ein symbolischer Link auf die gerade aktuelle Version der C-Bibliothek (hier Version 5) ist. Dies vereinfacht die Aktualisierung von dynamischen Bibliotheken ganz erheblich. Um z.B. die Version 5.m.r gegen die Version 5.x.y auszutauschen, muß nur die Datei `libc.so.5.x.y` in das Directory `/lib` kopiert werden und das Programm **ldconfig** aufgerufen werden.

Das Programm **ldconfig** sucht alle dynamischen Bibliotheken, die in bestimmten Directories liegen und erzeugt dann bei Bedarf einen symbolischen Link (Name mit der Hauptversionsnummer, wie z.B. `libc.so.5`) auf die jeweilige Version. **ldconfig** untersucht standardmäßig immer die beiden Directories `/lib` und `/usr/lib`. Daneben untersucht es noch die Dateien der Directories, die auf der Kommandozeile beim **ldconfig**-Aufruf angegeben werden und die Directories, deren Namen sich in der Datei `/etc/ld.so.conf` befinden.

Linkt man nun ein C-Programm, so wird der Linker immer nach einer Datei mit den Namen `/usr/lib/libc.so` suchen, die ein symbolischer Link auf die gerade aktuelle dynamische C-Bibliothek ist:

```
$ ls -l /usr/lib/libc.so
lrwxrwxrwx 1 root root 19 Jun 2 1998 /usr/lib/libc.so -> /lib/libc.so.5.4.44
$
```

Um sich alle durch **ldconfig** eingerichteten symbolischen Links anzeigen zu lassen, muß nur **ldconfig -p** aufgerufen werden:

```
$ ldconfig -p
589 libs found in cache '/etc/ld.so.cache' (version 1.7.0)
libzvt.so.0 (libc5) => /usr/i486-linux-libc5/lib/libzvt.so.0
libzvt.so (libc5) => /usr/i486-linux-libc5/lib/libzvt.so
libz.so.1 (libc6) => /usr/X11R6/lib/libz.so.1
libz.so.1 (libc6) => /usr/X386/lib/libz.so.1
libz.so.1 (libc5) => /usr/i486-linux-libc5/lib/libz.so.1
libz.so (libc6) => /usr/X11R6/lib/libz.so
libz.so (libc6) => /usr/X386/lib/libz.so
libz.so (libc5) => /usr/i486-linux-libc5/lib/libz.so
libxv3.so.3 (libc4) => /usr/i486-linuxaout/lib/libxv3.so.3
libxview.so.3 (libc5) => /usr/i486-linux-libc5/lib/libxview.so.3
libxview.so.3 (libc6) => /usr/openwin/lib/libxview.so.3
libxview.so (libc5) => /usr/i486-linux-libc5/lib/libxview.so
libxview.so (libc6) => /usr/openwin/lib/libxview.so
.....
libICE.so (libc5) => /usr/i486-linux-libc5/lib/libICE.so
libGLU.so (libc5) => /usr/i486-linux-libc5/lib/libGLU.so
```

```

libGLU.so (libc6) => /usr/lib/libGLU.so
libGL.so (libc5) => /usr/i486-linux-libc5/lib/libGL.so
libGL.so (libc6) => /usr/lib/libGL.so
libFnl.so.0 (libc5) => /usr/i486-linux-libc5/lib/libFnl.so.0
libFnl.so (libc5) => /usr/i486-linux-libc5/lib/libFnl.so
libEZ.so.1.3 (libc5) => /usr/i486-linux-libc5/lib/libEZ.so.1.3
libEZ.so.1 (libc5) => /usr/i486-linux-libc5/lib/libEZ.so.1
libEZ.so (libc5) => /usr/i486-linux-libc5/lib/libEZ.so
ld-linux.so.2 (ELF) => /lib/ld-linux.so.2
ld-linux.so.1 (libc5) => /usr/i486-linux-libc5/lib/ld-linux.so.1
ld-linux.so.1 (ELF) => /lib/ld-linux.so.1
$

```

Benutzer, die eigene dynamische Bibliotheken entwerfen wollen, sollten wissen, was zu beachten ist, damit eine neue dynamische Bibliothek abwärtskompatibel bleibt. Es gibt drei Arten von Änderungen an einer dynamischen Bibliothek, die diese inkompatibel zu vorherigen Versionen werden läßt:

1. Das Ändern oder Entfernen von Funktionsschnittstellen, was üblicherweise die von außen aufrufbaren Funktionen sind.
2. Das Ändern eines Funktionscodes in der Form, daß diese Funktion sich nicht mehr so verhält, wie es in der ursprünglichen Spezifikation festgelegt ist.
3. Das Ändern von Datenstrukturen, die nach außen sichtbar sind. Hierzu zählt jedoch nicht das Anfügen zusätzlicher Komponenten am Ende von Strukturen, die innerhalb der Bibliothek allokiert werden.

Dagegen ziehen die folgenden Modifikationen an einer dynamischen Bibliothek keine Inkompatibilität nach sich:

- Hinzufügen neuer Funktionen mit anderen Namen, um die Funktionalität einer existierenden dynamischen Bibliothek zu erweitern.
- Hinzufügen weiterer Komponenten am Ende von Strukturen, die innerhalb der Bibliothek allokiert werden. Dies gilt jedoch nicht für Datenstrukturen, die nicht innerhalb der Bibliothek allokiert werden, da dann Programme, die mit früheren Versionen gelinkt wurden, nicht genügend Speicherplatz allokiert haben. Ebenso sollten keine Datenstrukturen erweitert werden, die in Arrays verwendet werden.

22.7.2 Generieren von dynamischen Bibliotheken

Beim Erzeugen von dynamischen Bibliotheken muß man sich an die folgenden Regeln halten:

- Beim Kompilieren des Quellcodes mit **gcc** muß die Option **-fPIC** (*Position-Independent-Code*) angegeben werden, um positionsunabhängigen Code zu erzeugen, der an jede beliebige Adresse gelinkt und geladen werden kann.
- Zum Linken sollte **cc** bzw. **gcc** verwendet werden. Ein direktes Linken mit dem Linker **ld** ist nicht empfehlenswert, da der jeweilige C-Compiler automatisch den Linker **ld**

mit den erforderlichen Optionen aufruft. Ein typischer Aufruf zum Linken einer dynamischen Bibliothek mit **gcc** ist:

```
gcc -shared -Wl,-soname,soname -o bibname objektdatei(en) bibliothek(en)
```

- **-Wl** leitet dabei die Optionen an **ld** weiter, wobei die Kommas durch Leerzeichen ersetzt werden. Für *soname* ist der Bibliotheksname (mit Hauptversionsnummer) und für *bibname* der vollständige Bibliotheksname mit allen zugehörigen Versionsnummern anzugeben. Für *objektdatei(en)* ist eine Liste der Objektdateien anzugeben, die in diese dynamische Bibliothek aufzunehmen sind, und für *bibliothek(en)* ist eventuell eine Liste der Bibliotheken anzugeben, aus denen Funktionen in den Objektdateien aufgerufen werden. So empfiehlt es sich fast immer, die C-Bibliothek hier anzugeben: **-lc**. Um z.B. die dynamische Bibliothek `libtoll.so.1.2.5` mit dem *soname* `libtoll.so.1` aus den Objektdateien `toll.o` und `syntab.o` zu erzeugen, könnte der folgende Aufruf verwendet werden:

```
gcc -shared -Wl,-soname,libtoll.so.1 -o libtoll.so.1.2.5 toll.o syntab.o -lc
```

- Beim **gcc** sollte niemals die Option **-fomit-frame-pointer** angegeben werden.

22.7.3 Installieren von dynamischen Bibliotheken

Die Installation von dynamischen Bibliotheken erfolgt üblicherweise mit dem Programm **ldconfig**. Um eine dynamische Bibliothek korrekt zu installieren, empfiehlt sich die folgende Vorgehensweise:

1. Kopieren der dynamischen Bibliothek in das Directory, in dem sie aufbewahrt werden soll.
2. Erzeugen eines symbolischen Links in `/usr/lib` mit dem Namen *bibname*, der auf die dynamische Bibliothek verweist. Dies ist nur erforderlich, wenn man möchte, daß der Linker diese Bibliothek automatisch findet, so daß man nicht immer beim Linken die Option **-Lpfadname** angeben muß.
3. Eventuelles Eintragen des Directorys, in dem sich der symbolische Link bzw. die dynamische Bibliothek befindet, in die Datei `/etc/ld.so.conf`. Dieser Eintrag ist jedoch nicht notwendig, wenn die dynamische Bibliothek bzw. der symbolische Link sich in einem der Directories `/lib` oder `/usr/lib` befindet, oder der entsprechende Directoryname schon in `/etc/ld.so.conf` eingetragen ist.
4. Aufrufen des Programms **ldconfig**, das einen weiteren symbolischen Link mit dem *soname* in dem Directory erzeugt, in dem die dynamische Bibliothek installiert wurde. **ldconfig** trägt die Bibliothek danach in den dynamischen Lade-Cache (Datei `/etc/ld.so.cache`) ein, so daß der dynamische Lader die Bibliothek findet, wenn Programme gestartet werden, die mit ihr gelinkt wurden, ohne daß ein zeitaufwendiges Durchsuchen von vielen Directories erforderlich ist. Löscht man z.B. die Datei `/etc/ld.so.cache`, wird dies fast immer dazu führen, daß das System merklich langsamer wird. In diesem Fall sollte man mit einem Aufruf von **ldconfig** eine neue Datei `/etc/ld.so.cache` erzeugen.

22.7.4 Beispiel für das Erzeugen, Installieren und Benutzen einer dynamischen Bibliothek

In den vorherigen Kapiteln dieses Buches wurde immer das C-Programm `fehler.c` statisch dazugelinkt, um eine einheitliche und einfache Ausgabe von Fehlermeldungen zu erreichen. Hier soll nun dieses C-Programm `fehler.c` in eine dynamische Bibliothek umgewandelt werden, so daß es alle Programme, die es ab jetzt benutzen möchten, nicht mehr statisch dazubinden müssen, sondern es als dynamische Bibliothek benutzen können. Die dazu erforderlichen Schritte sind nachfolgend gezeigt:

1. Kompilieren des C-Programms `fehler.c`, um daraus eine Objektdatei zu erzeugen:

```
gcc -fPIC -Wall -g -c fehler.c
```

2. Generieren der dynamischen Bibliothek, indem man die Objektdatei `fehler.o` mit entsprechenden Optionen linkt:

```
gcc -g -shared -WL,-soname,libfehler.so.1 -o libfehler.so.1.0 fehler.o -lc
```

3. Kopieren der Datei `libfehler.so.1.0` nach `/usr/local/lib`, was üblicherweise nur dem Superuser erlaubt ist:

```
cp libfehler.so.1.0 /usr/local/lib
```

4. Erzeugen eines symbolischen Links in `/usr/lib`, was üblicherweise nur dem Superuser erlaubt ist:

```
cd /usr/lib
ln -sf ../local/lib/libfehler.so.1.0 libfehler.so.1
```

5. Erzeugen eines symbolischen Links für den Linker, der benutzt werden soll, wenn die dynamische Bibliothek beim Linken mit `-l` angegeben wird. Hier soll der Name `fehler` angegeben werden können, also `-lfehler`:

```
cd /usr/lib
ln -sf libfehler.so.1 libfehler.so
```

6. Aufrufen des Programms `ldconfig`:

```
ldconfig
```

Nun soll noch ein Programm 22.2 (`fehlinfo.c`) erstellt werden, das die eben erzeugte und installierte dynamische Bibliothek benutzt.

```
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include "eighdr.h"
```

```
int
main(void)
{
```



```

pid_t  pid;

srand(time(NULL)+getpid());

fehler_meld(WARNUNG, "Warnung (Kennung '%s')", "WARNUNG");
errno = rand()%50+1;
fehler_meld(WARNUNG_SYS, "Warnung mit Systemmeldung "
            "(Kennung '%s')", "WARNUNG_SYS");

if ( (pid=fork()) < 0)
    perror("fork-Fehler");

else if (pid == 0)
    fehler_meld(FATAL, "Fataler Fehler "
                "(Kennung '%s')", "FATAL");

else if (pid > 0) {
    errno = rand()%50+1;
    fehler_meld(FATAL_SYS, "Fataler Fehler mit Systemmeldung "
                "(Kennung '%s')", "FATAL_SYS");
}

exit(0);
}

```

Programm 22.2 (**fehlinfoc**): Ausgeben der Aufrufmöglichkeiten der Funktion **fehler_meld**

Um die dynamische Bibliothek `libfehler.so` zum Programm `fehlinfoc` dazu zu linkern, empfiehlt sich die nachfolgend gezeigte Vorgehensweise:

```

$ gcc -Wall -g -c fehlinfoc.c           [Kompilieren: fehlinfoc.c -> fehlinfoc.o]
$ gcc -g -o fehlinfoc fehlinfoc.o -lfehler [Linken mit der dynamischen Bibliothek]
$

```

Nun können wir das erzeugte Programm `fehlinfoc` starten.

```

$ fehlinfoc
Warnung (Kennung 'WARNUNG')
Warnung mit Systemmeldung (Kennung 'WARNUNG_SYS'): File too large
Fataler Fehler mit Systemmeldung (Kennung 'FATAL_SYS'): Text file busy
Fataler Fehler (Kennung 'FATAL')
$

```

Um die von einem Programm benötigten dynamischen Bibliotheken zu erfahren, muß man nur das Kommando **ldd** mit dem entsprechenden Programmnamen aufrufen, wie z.B.:

```

$ ldd /usr/bin/clear fehlinfoc
/usr/bin/clear:
    libncurses.so.4 => /lib/libncurses.so.4 (0x40009000)
    libc.so.6 => /lib/libc.so.6 (0x4004a000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00000000)
fehlinfoc:

```

```
libfehler.so.1 => /usr/local/lib/libfehler.so.1 (0x40009000)
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00000000)
```

```
$
```

22.7.5 Möglichkeiten zur Benutzung von dynamischen Bibliotheken

Existiert sowohl eine dynamische wie auch eine statische Bibliothek zu einem Namen, wie z.B.:

```
$ ls /usr/lib/libc.*
/usr/lib/libc.a  /usr/lib/libc.so
$
```

so bindet der Linker automatisch die dynamische Bibliothek dazu, wenn er keine anderen Anweisungen erhält. Neben diesem einfachen Dazubinden von dynamischen Bibliotheken beim Linken, gibt es noch drei weitere Möglichkeiten, dynamische Bibliotheken zu benutzen. Diese Möglichkeiten werden nachfolgend vorgestellt.

Benutzen von nicht installierten Bibliotheken

Startet man ein Programm, das dynamische Bibliotheken benutzt, versucht der dynamische Lader in dem Cache für Bibliotheken (`/etc/ld.so.cache`), der durch den Aufruf von **ldconfig** unter Zuhilfenahme der Datei `/etc/ld.so.conf` (enthält die Directories für dynamische Bibliotheken) erzeugt wurde, die vom Programm benutzten Bibliotheken zu finden. Ist jedoch die Environment-Variable `LD_LIBRARY_PATH` gesetzt, werden zuerst die darin enthaltenen Directories, die wie bei `PATH` mit Doppelpunkt voneinander zu trennen sind, durchsucht, bevor der Cache zum Auffinden der entsprechenden dynamischen Bibliothek herangezogen wird. So ist es möglich, daß man mit anderen Versionen von dynamischen Bibliotheken arbeitet, als die, welche installiert sind. Dies mag z.B. notwendig sein, wenn man ältere Programmversionen hat, die nicht mit einer neu installierten dynamischen Bibliothek ablauffähig sind, dafür aber mit einer älteren Version dieser dynamischen Bibliothek. In diesem Fall kopiert man die ältere Version in ein bestimmtes Directory und setzt vor dem Programmstart die Environment-Variable `LD_LIBRARY_PATH` entsprechend. Eleganter ist hierbei noch, das entsprechende Programm nicht direkt zu starten, sondern sich ein Shellskript zu erstellen, das in etwa das folgende Aussehen hat:

```
#!/bib/sh
export LD_LIBRARY_PATH=alt_bibl_dir:$LD_LIBRARY_PATH
exec alt_programm $*
```

Für `alt_bibl_dir` ist das Directory anzugeben, in dem sich die ältere Version der entsprechenden dynamischen Bibliothek befindet, und für `alt_programm` ist der Name des zu startenden Programms anzugeben.

Vorladen von dynamischen Bibliotheken

Manchmal möchte man nicht eine ganze dynamische Bibliothek, sondern nur einige Funktionen ersetzen. Da der dynamische Lader nach Funktionen sucht, indem er bei der ersten geladenen Bibliothek beginnt und dann in den anderen Bibliotheken in der Reihenfolge fortfährt, in der diese geladen wurden, reicht es zunächst aus, nur eine neue Bibliothek zu laden, die nur die neuen Funktionen enthält, die zu ersetzen sind.

Ein Beispiel hierzu ist die Bibliothek `zlibc`, die Funktionen, welche von der C-Bibliothek zur Dateibearbeitung angeboten werden, durch eigene Funktionen ersetzt, die mit komprimierten Dateien arbeiten können. Wird eine Datei geöffnet, sucht `zlibc` sowohl nach der angegebenen Datei als auch nach einer mit **gzip** gepackten Version dieser Datei. Findet es die angegebene ungepackte Datei, verhält sich die entsprechende Funktion genauso wie die Version dieser Funktion in der C-Bibliothek. Existiert die angegebene Datei aber nicht, dafür aber eine gepackte Version dieser Datei, entpackt sie diese, ohne daß das aufrufende Programm sich darum kümmern muß.

Um eine Bibliothek vorzuladen, gibt es zwei Möglichkeiten:

1. Setzen der Environment-Variable `LD_PRELOAD`.

```
LD_PRELOAD=/lib/vorlad.o exec /bin/progname $*
```

2. Eintragen der vorzuladenden Objektdatei in die Datei `/etc/ld.so.preload`. Für die Bibliothek `zlibc` könnte die folgende Zeile in die Datei `/etc/ld.so.preload` eingetragen werden.

```
/lib/uncompress.o
```

Dynamisches Laden zur Laufzeit (*shared objects*)

Größere Softwarepakete werden unter Unix/Linux üblicherweise in Module zerlegt, die getrennt voneinander entwickelt werden. Manchmal sind diese Module eigenständige Programme, die mit anderen Modulen des Softwarepakets über Pipes oder andere Formen der Interprozeßkommunikation (IPC) kommunizieren. Eine andere Möglichkeit der Kommunikation ist die Implementierung von sogenannten *shared objects* (geteilten Objekten). Solche *shared objects* können entweder Objektdateien oder dynamische Bibliotheken sein. Da der Linker nichts von den *shared objects* wissen muß, ist es noch nicht einmal erforderlich, daß diese zum Zeitpunkt des Linkens existieren müssen. Ein weiterer Unterschied von *shared objects* zu dynamischen Bibliotheken ist, daß sie anders installiert werden wie die meisten dynamischen Bibliotheken.

Daneben müssen die von *shared objects* verwendeten Symbolnamen nicht eindeutig und einmalig sein, was sie meist auch nicht sind, da verschiedene *shared objects*, die für die gleiche Schnittstelle entwickelt wurden, normalerweise auch Eintrittspunkte mit den gleichen Namen verwenden, was bei dynamischen Bibliotheken absolut unmöglich ist.

Die häufigste Anwendung von *shared objects* sind sogenannte generische Schnittstellen. Generische Schnittstellen sind im Prinzip nichts anderes als Funktionszeiger, denen erst

zur Laufzeit die Adresse der entsprechenden Funktion zugewiesen wird. So ist es möglich, daß Programme beliebig erweiterbar sind, ohne daß sie erneut kompiliert oder gelinkt werden müssen.

Ein Beispiel für die Verwendung von generischen Schnittstellen könnte ein Programm sein, das Simulationen für Industrieprozesse nach verschiedenen Verfahren durchführen kann. Dieses Programm verwendet intern ein eigenes Format, um die berechneten Werte graphisch am Bildschirm darzustellen. Wird nun eine generische Schnittstelle geschaffen, die die Durchführung der Simulation in zur Laufzeit geladene *shared objects* (unterschiedliche Verfahren) verlagert, kann jederzeit ein neues Simulationsverfahren hinzugefügt werden, ohne daß dieses Programm neu kompiliert und gelinkt werden muß. Generische Schnittstellen setzen allerdings immer eine gute Dokumentation ihrer Funktionsweise voraus, damit auch andere Programmierer, die die Interna des jeweiligen aufrufenden Hauptprogramms nicht kennen, sie benutzen und so den Funktionsumfang des Hauptprogramms erweitern können.

Dynamisches Laden erfordert die folgenden Aktivitäten:

Öffnen einer Bibliothek,

Suchen einer beliebigen Anzahl von Symbolen in dieser Bibliothek,

Auftretende Fehler behandeln und

Schließen der Bibliothek.

Die hierzu notwendigen Funktionen ***dlopen***, ***dlsym***, ***dlderror*** und ***dlclose*** sind in der Headerdatei `<dlfcn.h>` deklariert:

```
include <dlfcn.h>

void *dlopen(const char *filename, int flag);
                                gibt zurück: Zeiger für weitere Zugriffe auf die Bibliothek (bei Erfolg); NULL bei Fehler

void *dlsym(void *handle, char *symbol);
                                gibt zurück: Adresse, an die die Funktion symbol geladen wurde (bei Erfolg);
                                NULL, wenn das symbol nicht in der Bibliothek gefunden wurde

const char *dlderror(void);
                                gibt zurück: NULL, wenn in der Zwischenzeit kein Fehler aufgetreten ist;
                                Adresse eines Strings, der die Fehlermeldung enthält,
                                wenn bei einer vorherigen dl..-Operation ein Fehler aufgetreten ist.

int dlclose(void *handle);
```

Nachfolgend werden diese Funktionen im einzelnen beschrieben:

dlopen

dlopen lädt die dynamische Bibliothek, deren Name über den Parameter `filename` angegeben ist, und gibt einen Zeiger zurück, mit dem nun Zugriffe (mit den Funktionen

dlsym und **dlclose**) auf diese Bibliothek möglich sind. Wird für *filename* ein absoluter Pfad (beginnt mit /) angegeben, muß **dlopen** die Bibliothek nicht suchen. Dies ist der übliche Weg, **dlopen** aufzurufen. Ist der für *filename* angegebene Pfad kein absoluter Pfadname, sucht **dlopen** die entsprechende Bibliothek an den folgenden Stellen in der angegebenen Reihenfolge:

- ▶ in den Directories, die in der Environment-Variable `LD_ELF_LIBRARY_PATH` (durch Semikolons getrennt) angegeben sind, oder wenn `LD_ELF_LIBRARY_PATH` nicht existiert, in `LD_LIBRARY_PATH`
- ▶ die Bibliotheken, die in der Datei `/etc/ld.so.cache` aufgeführt sind; diese Datei wird mit dem Aufruf des Programms **ldconfig** erzeugt (siehe auch vorher)
- ▶ im Directory `/usr/lib`
- ▶ im Directory `/lib`

Gibt man für *filename* den `NULL`-Zeiger an, öffnet **dlopen** die Datei des aktuell ausgeführten Programms, was nur in sehr wenigen Fällen sinnvoll ist.

Undefinierte externe Referenzen (Bezüge) in der dynamischen Bibliothek werden aufgelöst, indem andere zuvor mit `RTLD_GLOBAL` geöffnete Bibliotheken und die Bibliotheken durchsucht werden, die in der Abhängigkeitsliste dieser Bibliothek enthalten sind.

Für *flag* kann eine der folgenden Konstanten angegeben werden:

`RTLD_LAZY`

Undefinierte Symbole in der dynamischen Bibliothek werden erst dann aufgelöst, wenn der Code dieser dynamischen Bibliothek ausgeführt wird.

`RTLD_NOW`

Alle undefinierten Symbole in der dynamischen Bibliothek werden aufgelöst, bevor die Funktion **dlopen** zurückkehrt. Wenn das nicht möglich, liefert **dlopen** den Rückgabewert `NULL`. Dieses Flag wird meist während der Entwicklung und Fehlersuche gesetzt, denn so wird man sofort über unaufgelöste Referenzen in *shared objects* informiert, und man muß nicht über einen unerklärlichen Programmabsturz beim weiteren Ablauf rätseln.

Mit bitweisem OR (`|`) kann noch die folgende Konstante mit einer der beiden vorherigen Konstanten verknüpft werden.

`RTLD_GLOBAL`

In diesem Fall werden die hier definierten externen Symbole den Bibliotheken, die nachfolgend geladen werden, zur Verfügung gestellt.

Enthält eine dynamische Bibliothek eine Funktion namens **_init**, wird diese ausgeführt, bevor **dlopen** zurückkehrt.

Wird die gleiche Bibliothek mehrmals geöffnet, dann wird immer der gleiche Zeiger (*handle*) zurückgegeben.

dlsym

dlsym sucht in der Bibliothek `handle`, was der Rückgabewert der zuvor mit **dlopen** erfolgreich geöffneten Bibliothek sein muß, nach dem Symbol mit dem Namen `symbol`. **dlsym** liefert die Adresse, an die dieses Symbol geladen wurde, oder, falls dieses nicht gefunden werden konnte, den `NULL`-Zeiger. Da es aber Symbole geben kann, die die Adresse `NULL` haben, läßt dieser Rückgabewert nicht unbedingt auf einen Fehler schließen. Deswegen ist es in diesem Fall empfehlenswert, die nachfolgend beschriebene Funktion **dlerror** heranzuziehen, um eine Fehlerüberprüfung durchzuführen.

dlerror

gibt `NULL` zurück, wenn kein Fehler seit dem Öffnen der dynamischen Bibliothek oder seit dem letzten Aufruf von **dlerror** aufgetreten ist, oder aber die Adresse der entsprechenden Fehlermeldung. Da jeder Aufruf von **dlerror** dazu führt, daß eine eventuell vorhandene Fehlermeldung nach diesem Aufruf nicht mehr zur Verfügung steht, sollte man diese Fehlermeldung in einer eigenen Variablen speichern, wenn man sie für spätere Zwecke wieder benötigt.

dlclose

Jedesmal, wenn **dlopen** eine Bibliothek öffnet, wird ein interner Referenzzähler erhöht. Dieser Referenzzähler wird bei jedem Aufruf der Funktion **dlclose** um 1 erniedrigt. Erst wenn dieser Referenzzähler bedingt durch einen **dlclose**-Aufruf 0 wird, wird auch die Bibliothek geschlossen und der für sie allokierte Speicherplatz freigegeben.

Enthält die dynamische Bibliothek eine Funktion namens **_fini**, wird diese ausgeführt, bevor **dlclose** zurückkehrt. Durch den Referenzzähler ist es möglich, beliebig oft die entsprechende Bibliothek zu öffnen und zu schließen, ohne sich darum kümmern zu müssen, ob die zugehörigen *shared objects* bereits vom aufrufenden Code geladen wurden.

Wenn diese Funktionen in einem Programm verwendet werden, muß man beim Linken dieses Programms die Bibliothek `libdl.so` mit der Option **-ldl** dazulinken.

Beispiel

Laden der mathematischen Funktion `sin` zur Ausgabe des Sinus

Das folgende Programm 22.3 (`sinus.c`) demonstriert das dynamische Laden eines *shared object*, indem es die Funktion **sin** aus der mathematischen Bibliothek lädt, um sie im Programm verwenden zu können.

```
#include <dlfcn.h>
#include "eighdr.h"

int
main(int argc, char *argv[])
{
    void      *handle;
    double    i, (*sinus)(double);
    const char *fehlmeld;
```

```

if (argc != 2)
    fehler_meld(FATAL, "usage: %s biblname", argv[0]);

if ( (handle = dlopen(argv[1], RTLD_LAZY)) == NULL)
    fehler_meld(FATAL, "kann Bibliothek '%s' nicht oeffnen: %s",
                argv[1], dlerror());

sinus = dlsym(handle, "sin");
if ( (fehler_meld = dlerror()) != NULL)
    fehler_meld(FATAL, "Fehler in Bibliothek '%s': %s",
                argv[1], fehler_meld);

for (i=0.0; i<=0.5; i+=0.1)
    printf("%lf\n", (*sinus)(i));

dlclose(handle);

exit(0);
}

```

Programm 22.3 (sinus.c): Laden der mathematischen Funktion **sin** zur Ausgabe des Sinus

Nachdem man dieses Programm 22.3 (sinus.c) kompiliert und gelinkt hat.

```
cc -o sinus sinus.c fehler.c -ldl
```

kann man es starten, indem man ihm den Pfadnamen der mathematischen Bibliothek als Argument auf der Kommandozeile übergibt.

```

$ sinus /usr/lib/libm.so
0.000000
0.099833
0.198669
0.295520
0.389418
0.479426
$

```

Nun soll aber eine eigene Funktion **sin** erstellt werden, die nicht den Sinus berechnet, sondern nur das Doppelte des übergebenen Werts zurückliefert, wie dies im nachfolgenden Programm 22.4 (sin.c) umgesetzt ist.

```

#include <stdio.h>

double sin(double wert)
{
    return(2*wert);
}

```

Programm 22.4 (sin.c): Funktion **sin**, die das Doppelte des übergebenen Werts zurückgibt

Aus diesem Programm wird nun mit den folgenden Aufrufen eine dynamische Bibliothek erstellt.

```
$ gcc -fPIC -Wall -g -c sin.c
$ gcc -g -shared -WL,-soname,sin.so.1 -o sin.so.1.0 sin.o -lc
$ ln -sf sin.so.1.0 sin.so.1
$
```

Startet man das Programm **sinus** erneut, dieses Mal übergibt man ihm aber die eigene dynamische Bibliothek `sin.so.1` im Working-Directory, verwendet es nun das *shared object* **sin** aus der eigenen Bibliothek.

```
$ LD_LIBRARY_PATH=. sinus sin.so.1
0.000000
0.200000
0.400000
0.600000
0.800000
1.000000
$
```

Beim Arbeiten mit *shared objects* ist also ein einfaches Austauschen von Funktionen möglich, ohne daß die Originalprogramme erneut kompiliert oder gelinkt werden müssen.

22.8 make – Ein Werkzeug zur automatischen Programmgenerierung

Eines der wichtigsten Tools bei der Softwareentwicklung unter Linux/Unix ist der Programmgenerator **make**. Bei der Softwareentwicklung spielt sich immer wieder folgendes Szenario ab: Ein Softwareprojekt besteht aus einer bestimmten Anzahl von Modulen, die zunächst für sich getrennt kompiliert werden müssen, bevor die daraus resultierenden Objektdateien mit dem Linker zu einem ablauffähigen Programm zusammengebunden werden können. Wenn nun die Schnittstelle (Headerdatei) eines Moduls geändert wird, dann müssen alle von diesen Schnittstellen abhängigen Module neu kompiliert werden, bevor wieder gelinkt werden kann. Da die Abhängigkeiten der einzelnen Module untereinander in einem großen Softwareprojekt äußerst komplex sein können, ist es meist nicht offensichtlich, für welche Module bei Änderungen von Schnittstellen eine erneute Kompilierung durchgeführt werden muß.

Mit dem Tool **make** kann dieses Problem gelöst werden. **make** muß dazu eine Datei vorgelegt werden, in der die Abhängigkeiten der Module untereinander beschrieben sind. **make** sorgt dann dafür, daß alle von den Änderungen betroffenen Module automatisch kompiliert werden, bevor das ablauffähige Programm mit dem Linker zusammengebunden wird.

Hier wird eine kurze Einführung in **make** gegeben⁵.

5. Im Buch *Linux-Unix Profitools* dieser Buchreihe befindet sich eine detailliertere Beschreibung des Tools **make**.

22.8.1 Das Makefile

Nehmen wir z.B. für ein Softwareprojekt den in Abbildung 22.1 gezeigten Abhängigkeitsbaum (*dependency tree*) für die einzelnen Module an.

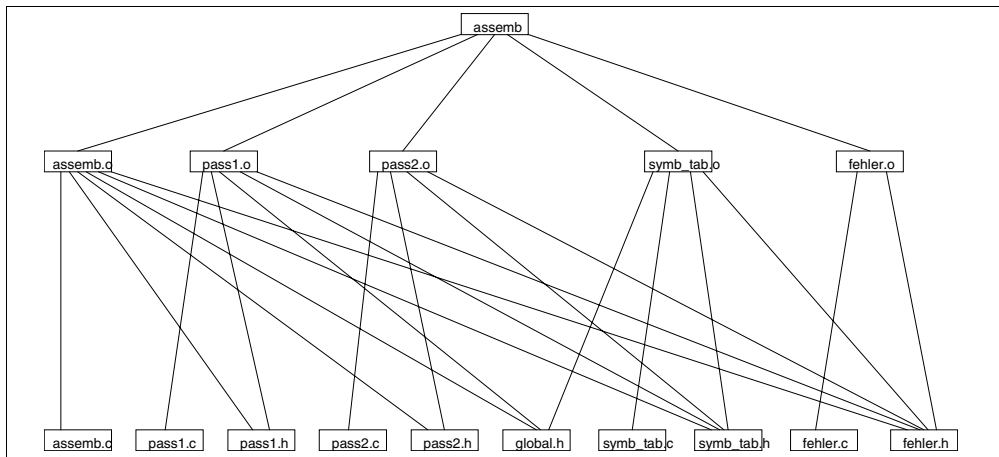


Abbildung 22.1: Abhängigkeitsbaum für die einzelnen Module in einem Softwareprojekt

Solche Abhängigkeiten werden beim Arbeiten mit **make** in einer Beschreibungsdatei, dem sogenannten *Makefile*, angegeben. Zu dem Abhängigkeitsbaum in Abbildung 22.1 könnte z.B. folgendes *makefile*⁶ angegeben werden:

```

$ n1 -ba makefile 7
1  #---- Makefile fuer das Assemblerprogramm ----
2  #-----
3
4  #.....Linker-Teil.....
5  assemb : assemb.o pass1.o pass2.o symb_tab.o fehler.o
6      cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
7
8  #.....Kompilierungs-Teil.....
9  assemb.o : assemb.c global.h pass1.h pass2.h symb_tab.h fehler.h
10     cc -c assemb.c      # Option -c bedeutet: nur Kompilieren
11
12  pass1.o : pass1.c pass1.h global.h symb_tab.h fehler.h
13     cc -c pass1.c
14
15  pass2.o : pass2.c pass2.h symb_tab.h fehler.h
16     cc -c pass2.c
17
18  symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h

```

6. Es ist sowohl des Namens *Makefile* als auch *makefile* für die make-Beschreibungsdatei erlaubt.

7. Statt dem Namen *makefile* könnte auch der Name *Makefile* verwendet werden.

```

19      cc -c symb_tab.c
20
21 fehler.o : fehler.c fehler.h
22      cc -c fehler.c
$

```

Anhand dieses Makefiles lassen sich bereits einige grundlegende Regeln aufstellen:

Leerzeilen werden von make ignoriert

Zwecks besserer Lesbarkeit können beliebig viele Leerzeilen in einem Makefile angegeben sein. **make** überliest solche Leerzeilen einfach.

Kommentare werden mit # eingeleitet

Alle Zeichen ab # bis zum Zeilenende werden von **make** als Kommentar interpretiert und folglich ignoriert. Ein Kommentar kann in einem Makefile als eine eigene Zeile angegeben werden, er kann aber auch am Ende einer für **make** relevanten Zeile stehen.

Ein Eintrag besteht aus einer Abhängigkeitsbeschreibung mit Kommandos

Einträge in einem Makefile setzen sich aus zwei Komponenten zusammen:

Abhängigkeitsbeschreibung (dependency line) und den
dazugehörigen Kommandozeilen

Zwischen diesen beiden Komponenten darf keine Leerzeile angegeben werden. Im obigen Makefile sind sechs Einträge angegeben. So handelt es sich z.B. bei den Zeilen 18 und 19 im obigen Makefile um einen Eintrag:

```

symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h
            cc -c symb_tab.c

```

Es sei hier angemerkt, daß neben solchen Abhängigkeitseinträgen noch andere Angaben erlaubt sind, wie z.B. Makrodefinitionen; dazu aber später mehr.

Regeln für eine Abhängigkeitsbeschreibung

Eine Abhängigkeitsbeschreibung muß immer vollständig in einer Zeile angegeben werden, wobei folgende Syntax einzuhalten ist:

ziel : objekt1 objekt2

Eine solche Zeile beschreibt, von welchen *objekten* das *ziel* (target) abhängig ist. Vor dem *ziel* darf nie ein Tabulatorzeichen angegeben sein, und es muß mit Doppelpunkt von den *objekten* getrennt sein. Die einzelnen *objekte* müssen mit Leer- oder Tabulatorzeichen voneinander getrennt angegeben werden. Als Beispiel möge die 15. Zeile aus obigen Makefile dienen:

```

pass2.o : pass2.c pass2.h symb_tab.h fehler.h

```

Diese Zeile besagt, daß die Objektdatei *pass2.o* von den Dateien *pass2.c*, *pass2.h*, *symb_tab.h* und *fehler.h* abhängt. Solche Zeilen beschreiben also die Abhängigkeiten entsprechend dem Abhängigkeitsbaum.

Für die Abhängigkeitsbeschreibung gilt weiterhin folgendes:

- ▶ Es sind auch Abhängigkeitsbeschreibungen erlaubt, bei denen nur das *ziel* (mit Doppelpunkt) ohne *objekte* angegeben ist. Fehlende Abhängigkeiten in einer Abhängigkeitsbeschreibung bedeuten, daß die zugehörigen Kommandozeilen bei Anforderung immer ausgeführt werden.
- ▶ In einer Abhängigkeitsbeschreibung darf auch mehr als ein *ziel* angegeben werden.
- ▶ Ein gleiches *ziel* kann mehrmals angegeben werden
So können die unterschiedlichen Arten von Abhängigkeiten hervorgehoben werden. Beispielsweise kann im obigen *makefile* der Eintrag

```
symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h
cc -c symb_tab.c
```

- ▶ wie folgt aufgetrennt werden:

```
symb_tab.o : symb_tab.c      # Implementations-Abhängigkeit
cc -c symb_tab.c
.....
.....
.....
symb_tab.o : symb_tab.h global.h fehler.h # Schnittstellen-Abhängigkeit
```

Der zur Generierung von *symb_tab.o* erforderliche Compileraufruf (**cc -c symb_tab.c**) ist nur bei der ersten Abhängigkeitsbeschreibung angegeben. Nichtsdestoweniger wird die Kompilierung von *symb_tab.c* nicht nur bei Änderung von *symb_tab.c*, sondern auch bei Änderungen in den Headerdateien *symb_tab.h*, *global.h* und *fehler.h* durchgeführt.

Allgemein gilt: Wenn ein gleiches *ziel* mehrmals verwendet wird, dann dürfen Kommandozeilen nur bei einer Abhängigkeitsbeschreibung angegeben sein.

- ▶ Will man für die verschiedenen *objekte*, von denen ein *ziel* abhängig ist, unterschiedliche Kommandos ausführen lassen, so muß der doppelte Doppelpunkt :: in den jeweiligen Abhängigkeitsbeschreibungen verwendet werden.

Regeln für Kommandozeilen

Die direkt nach einer Abhängigkeitsbeschreibung angegebenen Kommandozeilen müssen immer mit mindestens einem Tabulatorzeichen eingerückt sein. Da **make** jede Zeile, die mit einem Tabulatorzeichen beginnt, als Kommandozeile interpretiert, ist es äußerst wichtig, daß Kommandozeilen immer mit Tabulatorzeichen eingerückt sind. Andere Zeilen dagegen sollten nie mit einem Tabulatorzeichen beginnen, denn **make** meldet in solchen Fällen immer einen Fehler, selbst wenn es sich um Leerzeilen handelt oder um

Zeilen, in denen nur ein Kommentar angegeben ist. Da die falsche oder fehlende Angabe von Tabulatorzeichen ein häufiger Fehler ist und die dann von **make** gelieferten Fehlermeldungen nicht sehr aussagekräftig sind, sollte man sein Makefile in solchen Fällen in einer Form auflisten, welche die Tabulatorzeichen erkennen läßt. Dazu empfiehlt sich der folgende Aufruf:

```
cat -vt -e makefile
```

Tabulatorzeichen werden dann mit `^I` (Option **-vt**) und das Zeilenende wird mit `$` (Option **-e**) angezeigt.

Diese Sonderregelung für Tabulatorzeichen gilt nur am Zeilenbeginn, an allen anderen Stellen können beliebig Tabulatorzeichen angegeben werden.

Für die Kommandozeilen gilt weiterhin folgendes:

- Zu einer Abhängigkeitsbeschreibung können auch mehr als eine Kommandozeile angegeben werden. In diesem Fall sind die Kommandozeilen direkt untereinander anzugeben und immer mit Tabulatorzeichen einzurücken, wie z.B.:

```
symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h
    echo "symb_tab.o wird generiert"
    cc -c symb_tab.c
```

Falls *symb_tab.o* neu erzeugt werden muß, wird zuerst die Meldung *symb_tab.o wird generiert* ausgegeben, bevor der Compiler zur Übersetzung von *symb_tab.c* aufgerufen wird.

- Für jede Kommandozeile wird eine eigene Subshell gestartet.
- Mehrere Kommandos in einer Zeile sind mit Semikolon zu trennen.
- Mehrere Kommandos können mit Semikolon und Fortsetzungszeichen `\` zu einer Zeile zusammengefaßt werden. Shell-Kommandos, die zur Ablaufsteuerung eines Shell-Skripts verwendet werden (**if**, **for**, **while**, ..), erstrecken sich meist über mehrere Zeilen. Werden solche Kommandos in Makefiles verwendet, dann müssen Semikolons und das Zeilen-Fortsetzungszeichen `\` verwendet werden, um sie von der Shell als eine Kommandozeile interpretieren zu lassen.
- Auf Shell-Variablen kann in einer Kommandozeile zugegriffen werden, indem dem Namen der betreffenden Shell-Variablen ein **\$\$** (doppeltes **\$**) vorangestellt wird.
- **@** am Anfang einer Kommandozeile schaltet die automatische Ausgabe dieser Kommandozeile vor seiner Ausführung aus. Dies gilt nicht für die Option **-n**.
- **-** (Querstrich) am Anfang einer Kommandozeile schaltet den automatischen **make**-Abbruch bei Auftreten eines Fehlers in dieser Kommandozeile ab.
- Über die Variable **SHELL** kann die Shell festgelegt werden, die **make** zur Ausführung der Kommandozeilen verwenden soll. Soll z.B. die C-Shell benutzt werden, so könnte **SHELL = /bin/csh** im Makefile angegeben werden. Voreingestellt ist meist die Bourne-Shell.

Abhängigkeitsbeschreibung und Kommandozeilen in einer Zeile

Eine Abhängigkeitsbeschreibung und die dazugehörigen Kommandozeilen können auch in einer Zeile angegeben werden, wenn sie mit Semikolon voneinander getrennt sind:

ziel : *objekt1 objekt2 ...* ; *kdozeile1; kdozeile2*

So kann man z.B. den folgenden Eintrag aus obigen *makefile*

```
fehler.o : fehler.c fehler.h
        cc -c fehler.c
```

auch wie folgt angeben:

```
fehler.o : fehler.c fehler.h ; cc -c fehler.c
```

Dies ist im übrigen die einzige Ausnahme, bei der eine Kommandozeile nicht mit einem Tabulatorzeichen beginnen muß.

**Das Zeilenfortsetzungszeichen **

Abhängigkeitsbeschreibungen müssen, wie bereits erwähnt, in einer Zeile angegeben werden. Da in größeren Projekten ein *ziel* von sehr vielen *objekten* abhängen kann, erhält man oft sehr lange Beschreibungszeilen. Aus Gründen der besseren Lesbarkeit ist es deshalb erlaubt, eine solche Beschreibung über mehrere Zeilen zu erstrecken. Dazu muß am Ende jeder Zeile (außer der letzten) das Fortsetzungszeichen `\` angegeben werden. **make** fügt dann solche Zeilen zu einer Zeile zusammen. So kann z.B. der Eintrag

```
symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h
        cc -c symb_tab.c
```

auch wie folgt angegeben werden:

```
symb_tab.o : symb_tab.c \
              symb_tab.h \
              global.h \
              fehler.h
        cc -c symb_tab.c
```

Dabei ist zu beachten, daß das Fortsetzungszeichen `\` wirklich das letzte Zeichen der Zeile ist und keine Leer-, Tabulator- oder sonstige Zeichen mehr folgen.

Fortsetzungszeichen am Ende eines Kommentars werden ignoriert.

Abhängigkeitsüberprüfung anhand der Zeitmarken

Die zu einer Änderungsbeschreibung angegebenen Kommandozeilen werden von **make** immer dann ausgeführt, wenn eines der in der Abhängigkeitsbeschreibung angegebenen *objekte* eine neuere Zeitmarke (*time stamp*) besitzt als *ziel* oder wenn das *ziel* noch nicht existiert. Eine Zeitmarke für eine Datei enthält immer das Datum und die Zeit der letzten Änderung an dieser Datei. Die aktuellen Zeitmarken für Dateien können immer mit **ls -l** aufgelistet werden.

Anhand dieser vom Betriebssystem eingetragenen Zeitmarken ist es für **make** ein leichtes zu prüfen, ob eines der *objekte* in einer Abhängigkeitsbeschreibung jünger ist als das *ziel*. Bevor **make** aber den Vergleich der Zeitmarken in einer bestimmten Abhängigkeitsbeschreibung durchführt, prüft es noch, ob eines der dort erwähnten *objekte* eventuell in einer anderen Abhängigkeitsspezifikation als *ziel* angegeben ist. Trifft dies zu, so wird erst diese Änderungsbeschreibung bearbeitet. Auf den Abhängigkeitsbaum bezogen bedeutet dies, daß **make** die Zeitmarken der einzelnen Knoten in diesem Baum von unten nach oben überprüft. Erst wenn eine Ebene vollständig aktualisiert ist, wird die nächste Ebene bearbeitet. Man spricht oft auch von direkten und indirekten Abhängigkeiten. So besteht z.B. zwischen *assemb* und *pass2.o* oder zwischen *pass2.o* und *pass2.c* eine direkte Abhängigkeit. Eine indirekte Abhängigkeit besteht hier z.B. zwischen *assemb* und *pass2.c*. Bedient man sich dieser Definition, dann kann man sagen, daß **make** zuerst immer alle indirekten Abhängigkeiten abarbeitet, bevor es die direkten Abhängigkeiten bearbeitet.

Bevor **make** also den ersten Eintrag im obigen *makefile* bearbeitet, überprüft es zuerst, ob eine der Objektdateien *assemb.o*, *pass1.o*, *pass2.o*, *fehler.o*, *symb_tab.o* aufgrund von Schnittstellenänderungen oder Änderungen in den Implementationen neu kompiliert werden muß. Nehmen wir z.B. an, daß *pass2.c* geändert wurde, so wird **make** zuerst die Kompilierung von *pass2.c* veranlassen:

```
cc -c pass2.c
```

bevor es die einzelnen Module zu einem ablauffähigen Programm *assemb* linken läßt:

```
cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
```

Zusammenfassend kann gesagt werden, daß **make** erst dann, wenn alle Module auf der rechten Seite einer Abhängigkeitsbeschreibung aktualisiert sind, die dazu angegebenen Kommandozeilen ausführt.

22.8.2 Einfache Aufrufformen von make

Im folgenden werden mögliche Aufrufformen von **make** mit einigen wichtigen Optionen vorgestellt.

make-Aufruf ohne Angabe von Argumenten

Um unser Assemblerprogramm mit obigen *makefile* generieren zu lassen, muß **make** ohne jegliche Argumente aufgerufen werden:

```
$ make
  cc -c assemb.c      # Option -c bedeutet: nur Kompilieren
  cc -c pass1.c
  cc -c pass2.c
  cc -c symb_tab.c
  cc -c fehler.c
  cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
$
```

Wie zu sehen ist, gibt **make** jedes Kommando aus, bevor es dieses zur Ausführung bringt. Wird **make** ohne jegliche Argumente aufgerufen, so bestimmt der erste Eintrag, was zu erzeugen ist. Da in unserem Fall

```
assemb : assemb.o pass1.o pass2.o symb_tab.o fehler.o
        cc -o assemb assemb.o pass1.o pass2.o symb_tab.o fehler.o
```

als erstes angegeben ist, wird das Assemblerprogramm *assemb* erzeugt, wobei zuvor alle notwendigen Kompilierungen der einzelnen Module durchgeführt werden.

Gibt man dagegen z.B. die Zeilen 12 und 13

```
pass1.o : pass1.c pass1.h global.h symb_tab.h fehler.h
        cc -c pass1.c
```

als ersten Eintrag im obigen *makefile* an, dann würde der Aufruf von **make** (ohne jegliche Argumente) lediglich die Objektdatei *pass1.o* erzeugen:

```
$ make
        cc -c pass1.c
$
```

make-Aufruf mit Angabe von Zielen

Unabhängig von der Reihenfolge der Einträge kann man durch die Angabe von *zielen* beim **make**-Aufruf erreichen, daß ausschließlich diese *ziele* erzeugt werden. Dazu muß man

make *ziel1 ziel2*

aufrufen. Soll z.B. nur die Objektdatei *symb_tab.o* generiert werden, so lautet der Aufruf wie folgt:

```
$ make symb_tab.o
        cc -c symb_tab.c
$
```

Sollen z.B. nur die Objektdateien *fehler.o* und *pass2.o* generiert werden, ist folgender Aufruf notwendig:

```
$ make fehler.o pass2.o
        cc -c fehler.c
        cc -c pass2.c
$
```

Um das Assemblerprogramm *assemb* (unabhängig von der Reihenfolge der Einträge) vollständig generieren zu lassen, wird der folgende Aufruf verwendet:

make assemb

Angenommen unser Assemblerprogramm soll in zwei Versionen angeboten werden: *assemb* und *assemb2*. Bei der zweiten Version *assemb2* soll es sich um eine erweiterte Version handeln, die mehr Kommandos kennt und deshalb anstelle des Moduls *symb_tab.c* das Modul *symb_ta2.c* verwendet. Beide können über dasselbe *makefile* generiert werden:

```

$ n1 -ba makefile
1  #---- Makefile fuer das Assemblerprogramm ----
2  #-----
3
4  #.....Linker-Teil.....
5  assemb : assemb.o pass1.o pass2.o symb_tab.o fehler.o
6      echo "assemb wird nun gelinkt....."
7      cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
8
9  assemb2 : assemb.o pass1.o pass2.o symb_ta2.o fehler.o
10     echo "assemb2 wird nun gelinkt....."
11     cc -o assemb2 assemb.o pass1.o pass2.o fehler.o symb_ta2.o
12
13  #.....Kompilierungs-Teil.....
14  assemb.o : assemb.c global.h pass1.h pass2.h symb_tab.h fehler.h
15      cc -c assemb.c          # Option -c bedeutet: nur Kompilieren
16
17  pass1.o : pass1.c pass1.h global.h symb_tab.h fehler.h
18      cc -c pass1.c
19
20  pass2.o : pass2.c pass2.h symb_tab.h fehler.h
21      cc -c pass2.c
22
23  symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h
24      cc -c symb_tab.c
25
26  symb_ta2.o : symb_ta2.c symb_tab.h global.h fehler.h
27      cc -c symb_ta2.c
28
29  fehler.o : fehler.c fehler.h
30      cc -c fehler.c
31
32  #.....Cleanup.....
33  cleanup :
34      echo "Folgende Dateien werden nun geloescht:"
35      echo "      " *.o
36      /bin/rm -f *.o
$

```

Die gegenüber dem ursprünglichen Makefile neu hinzugekommenen Zeilen sind im obigen Listing fett gedruckt.

Möchten wir nun die erste Version des Assemblers *assemb* erzeugen, so brauchen wir nur

make assemb

aufzurufen. Möchten wir dagegen die zweite Version des Assemblers *assemb2* generieren, so muß

make assemb2

aufgerufen werden. Es kann also ein und dasselbe Makefile für die Generierung unterschiedlicher Versionen oder eventuell sogar verschiedener Programme benutzt werden.

Abhängigkeitsangaben ohne Abhängigkeiten

Es sind auch Abhängigkeitsbeschreibungen erlaubt, bei denen nur das *ziel* (mit Doppelpunkt) ohne *objekte* angegeben ist. Im vorangegangenen *makefile* wurde beim Ziel *cleanup* hiervon Gebrauch gemacht:

```
cleanup :
    echo "Folgende Dateien werden nun gelöscht:"
    echo "      " *.o
    /bin/rm -f *.o
```

Um nun alle Objektdateien des Working-Directory zu löschen, muß z.B. nur

make cleanup

aufgerufen werden. Fehlende Abhängigkeiten in einer Abhängigkeitsbeschreibung bewirken nämlich, daß die zugehörigen Kommandozeilen bei Anforderung immer ausgeführt werden. Bei obigem Aufruf muß darauf geachtet werden, daß keine Datei mit dem Namen *cleanup* im Working Directory existiert, denn in diesem Fall werden nicht, wie wir im nächsten Kapitel sehen, die *cleanup*-Kommandozeilen ausgeführt, sondern **make** meldet, daß die Datei *cleanup* bereits auf dem neuesten Stand (*up to date*) ist.

Die Option -s

Wird beim Aufruf von **make** die Option **-s** (*silent*) angegeben, gibt **make** die Kommandos nicht nochmals explizit vor ihrer Ausführung aus:

```
$ make -s cleanup
Folgende Dateien werden nun gelöscht:
    assemb.o fehler.o pass1.o pass2.o symb_ta2.o symb_tab.o
$
```

Die Option -n

Wird **make** mit der Option **-n** (*no execute*) aufgerufen, zeigt es an, welche Kommandozeilen es ausführen würde, führt diese aber nicht aus:

```
$ make -n          [Nur anzeigen, was zu generieren ist]
    cc -c assemb.c      # Option -c bedeutet: nur Kompilieren
    cc -c pass1.c
    cc -c pass2.c
    cc -c symb_tab.c
    cc -c fehler.c
    echo "assemb wird nun gelinkt....."
    cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
$ make -s assemb8
assemb.c
```

8. Abhängig vom Compiler werden die gerade kompilierten Dateien angezeigt oder nicht. Hier wird zum besseren Nachvollziehen der stattfindenden Aktionen angenommen, daß die gerade kompilierten Dateien angezeigt werden, was z.B. für den **gcc** von Linux nicht gilt.

```

pass1.c
pass2.c
symb_tab.c
fehler.c
assemb wird nun gelinkt.....
$ make -n assemb2
    cc -c symb_ta2.c
    echo "assemb2 wird nun gelinkt....."
    cc -o assemb2 assemb.o pass1.o pass2.o fehler.o symb_ta2.o
$ make assemb2
    cc -c symb_ta2.c
    echo "assemb2 wird nun gelinkt....."
assemb2 wird nun gelinkt.....
    cc -o assemb2 assemb.o pass1.o pass2.o fehler.o symb_ta2.o
$

```

Simulation des Arbeitens mit make

Wir wollen nun Änderungen an Dateien simulieren, wie sie während der Softwareentwicklung in der Praxis ständig vorkommen. Dazu ändern wir nicht den Inhalt einer entsprechenden Datei, sondern lediglich deren Zeitmarke mit dem Kommando **touch**. Das Kommando **touch** trägt immer die aktuelle Zeit als neue Zeitmarke für eine Datei ein und simuliert so eine Änderung an einer Datei:

```

$ touch global.h
$ make -n assemb
    cc -c assemb.c          # Option -c bedeutet: nur Kompilieren
    cc -c pass1.c
    cc -c symb_tab.c
    echo "assemb wird nun gelinkt....."
    cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
$ make -s assemb
assemb.c
pass1.c
symb_tab.c
assemb wird nun gelinkt.....
$ touch symb_ta2.c
$ make -n assemb2
    cc -c symb_ta2.c
    echo "assemb2 wird nun gelinkt....."
    cc -o assemb2 assemb.o pass1.o pass2.o fehler.o symb_ta2.o
$ make -s assemb2
symb_ta2.c
assemb2 wird nun gelinkt.....
$ touch fehler.c
$ make -n assemb
    cc -c fehler.c
    echo "assemb wird nun gelinkt....."
    cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
$ make -s assemb
fehler.c
assemb wird nun gelinkt.....
$ touch fehler.h

```

```

$ make -n assemb
    cc -c assemb.c          # Option -c bedeutet: nur Kompilieren
    cc -c pass1.c
    cc -c pass2.c
    cc -c symb_tab.c
    cc -c fehler.c
    echo "assemb wird nun gelinkt....."
    cc -o assemb assemb.o pass1.o pass2.o fehler.o symb_tab.o
$ make -s assemb
assemb.c
pass1.c
pass2.c
symb_tab.c
fehler.c
assemb wird nun gelinkt.....
$ make -n assemb2
    cc -c symb_ta2.c
    echo "assemb2 wird nun gelinkt....."
    cc -o assemb2 assemb.o pass1.o pass2.o fehler.o symb_ta2.o
$ make -s assemb2
symb_ta2.c
assemb2 wird nun gelinkt.....
$

```

Weitere wichtige Optionen

Tabelle 22.7 zeigt einige weitere wichtige **make**-Optionen.

Option	Bedeutung
-e	(e nvironment) Priorität von Shell-Variablen über die von Makrodefinitionen in Makefiles stellen.
-f mfile	(f ile) Soll make ein Makefile benutzen, das nicht einen der beiden Namen <i>Makefile</i> oder <i>makefile</i> hat, so kann man über diese Option den Namen des gewünschten Makefiles angeben. Mehrfache Angabe von -f mfile ist dabei auch erlaubt. make arbeitet dann die einzelnen <i>mfiles</i> nacheinander ab. Wird für <i>mfile</i> ein Querstrich – angegeben, so liest make die Spezifikationen für ein Makefile von der Standard-eingabe.
-i	(i gnore errors) Alle eventuell auftretenden Fehler ignorieren; kann auch mit <i>IGNORE</i> : im Makefile festgelegt werden.
-k	Generierung des aktuellen Ziels beim Auftreten eines Fehlers zwar abbrechen, aber mit der Generierung des nächsten Ziels, das von dem momentan behandelten Ziel unabhängig ist, fortfahren.
-p	(p rint) Alle für diesen make -Lauf gültigen Makrodefinitionen, Abhängigkeitsbeschreibungen mit zugehörigen Kommandozeilen und Suffixregeln ausgeben.
-q	(q uestion) Anzeigen über exit-Status, ob Ziele auf dem neuesten Stand sind (exit-Status 0) oder erst generiert werden müssten (exit-Status ungleich 0).

Tabelle 22.7: Weitere wichtige **make**-Optionen

Option	Bedeutung
-r	(remove suffix rules) Alle vordefinierten Suffixregeln ausschalten.
-t	(touch) Ohne Generierung die Zeitmarken der Ziele mit touch auf aktuelle Zeit setzen.

Tabelle 22.7: Weitere wichtige **make**-Optionen

22.8.3 Makros

Das Makefile des letzten Kapitels enthielt einige Wiederholungen. Da in größeren Softwareprojekten die einzelnen *ziele* von sehr vielen *objekten* abhängen können und dort oft ein Makefile die Generierung für mehrere Versionen des gleichen Produkts enthält, kann es zu häufigen Wiederholungen in Makefiles kommen. Dies bedeutet nicht nur unnütze Tipparbeit, sondern hat auch den Nachteil, daß derartig aufgeblähte Makefiles nicht gut lesbar sind. Durch die Verwendung von Makros werden nicht nur diese Nachteile vermieden, sondern auch flexiblere Makefiles erstellt, die eine leichtere Anpassung an neue Gegebenheiten zulassen. Man denke dabei nur an die Debug-Option **-g** beim Kompilieren und Linken. Soll z.B. während der Entwicklung eines Programms kurzfristig eine Debug-Information für ein Programm erzeugt werden, so müssen alle entsprechenden Kommandozeilen im Makefile geändert werden. Bei Benutzung eines Makros dagegen ist nur die Änderung dieses Makros im Makefile notwendig, um das Makefile für die Generierung von Debug-Information auszustatten.

Unter Verwendung von Makros können wir unser *makefile* aus dem letzten Kapitel wie folgt schreiben:

```
$ n1 -ba makefile
1  #--- Makefile fuer das Assemblerprogramm -----
2  #-----
3
4  #.....Makrodefinitionen.....
5  CC = cc
6  CFLAGS = -c
7  LD = cc    # ld ist der eigentliche UNIX-Linker (ld=Abk fuer loader)
8  LDFLAGS = -o
9  DEBUG = # jetzt leer; fuer Debugging auf -g setzen
10
11  EXT = o
12  BASISOBSJS = assemb.${EXT} pass1.${EXT} pass2.${EXT} fehler.${EXT}
13  OBJJS1      = $(BASISOBSJS) symb_tab.${EXT}
14  OBJJS2      = $(BASISOBSJS) symb_ta2.${EXT}
15  ZIEL1 = assemb
16  ZIEL2 = assemb2
17
18  CLEANAKTION = \
19      echo "Folgende Dateien werden nun geloescht:"; \
20      echo "      " *.o; /bin/rm -f *.o
21
22
```

```

23  #.....Linker-Teil.....
24  ${ZIEL1} : ${OBS1}
25      echo "${ZIEL1} wird nun gelinkt....."
26      ${LD} ${DEBUG} ${LDFLAGS} ${ZIEL1} ${OBS1}
27
28  ${ZIEL2} : ${OBS2}
29      echo "${ZIEL2} wird nun gelinkt....."
30      ${LD} ${DEBUG} ${LDFLAGS} ${ZIEL2} ${OBS2}
31
32  #.....Kompilierungs-Teil.....
33  assemb.o : assemb.c global.h pass1.h pass2.h symb_tab.h fehler.h
34      ${CC} ${DEBUG} ${CFLAGS} assemb.c
35
36  pass1.o : pass1.c pass1.h global.h symb_tab.h fehler.h
37      ${CC} ${DEBUG} ${CFLAGS} pass1.c
38
39  pass2.o : pass2.c pass2.h symb_tab.h fehler.h
40      ${CC} ${DEBUG} ${CFLAGS} pass2.c
41
42  symb_tab.o : symb_tab.c symb_tab.h global.h fehler.h
43      ${CC} ${DEBUG} ${CFLAGS} symb_tab.c
44
45  symb_ta2.o : symb_ta2.c symb_tab.h global.h fehler.h
46      ${CC} ${DEBUG} ${CFLAGS} symb_ta2.c
47
48  fehler.o : fehler.c fehler.h
49      ${CC} ${DEBUG} ${CFLAGS} fehler.c
50
51  #.....Cleanup.....
52  cleanup :
53      ${CLEANAKTION}
$

```

Zunächst wollen wir dieses *makefile* testen:

```
$ make -s cleanup
```

Folgende Dateien werden nun gelöscht:

```
    assemb.o fehler.o pass1.o pass2.o symb_ta2.o symb_tab.o
```

```
$ make fehler.o
```

```
    cc -c fehler.c
```

```
fehler.c
```

```
$ make -s
```

```
assemb.c
```

```
pass1.c
```

```
pass2.c
```

```
symb_tab.c
```

```
assemb wird nun gelinkt.....
```

```
$ make -s assemb2
```

```
symb_ta2.c
```

```
assemb2 wird nun gelinkt.....
```

```
$
```

Dieses *makefile* scheint das gleiche zu leisten wie das *makefile* aus dem vorangegangenen Kapitel.

Anhand dieses Makefiles wollen wir nun die für Makros geltenden Regeln erarbeiten.

Definition von Makros mit *makroname* = *string*

Eine Makrodefinition ist eine Zeile, die ein Gleichheitszeichen = enthält⁹:

makroname = *string*

Mit dieser Definition wird dem *makronamen* der nach dem = angegebene *string* zugeordnet.

Die Definition eines Makros erstreckt sich vom Zeilenanfang bis zum Zeilenende bzw. bis zum Start eines Kommentars (#).

Links und rechts vom = müssen keine Leer- oder Tabulatorzeichen angegeben werden; sind doch welche angegeben, so werden sie von **make** ignoriert.

Zum *string* gehören alle Zeichen vom ersten relevanten Zeichen bis zum Zeilenende bzw. bis zum Start eines Kommentars. Relevant bedeutet hier: Zeichen, die keine Leer- oder Tabulatorzeichen sind, denn **make** ignoriert alle führenden Leer- und Tabulatorzeichen in einem *string*.

Damit **make** eine Makrodefinition von einer Kommandozeile unterscheiden kann, darf eine Zeile, die eine Makrodefinition enthält, niemals mit einem Tabulatorzeichen beginnen.

Wird am Ende einer Zeile, die eine Makrodefinition enthält, ein Fortsetzungszeichen \ angegeben, so setzt **make** beim Zusammenfügen hierfür genau ein Leerzeichen ein und entfernt in der Folgezeile alle am Anfang stehenden Leer- und Tabulatorzeichen.

Obwohl Makrodefinitionen überall in einem Makefile angegeben werden dürfen, ist es dennoch empfehlenswert, alle Makrodefinitionen am Anfang eines Makefiles anzugeben. Dies erleichtert das Auffinden und Ändern von Makros.

Makronamen sind Folgen von Buchstaben, Ziffern und Unterstrichen

Bei der Vergabe von Makronamen sind Buchstaben¹⁰, Ziffern und Unterstriche (_) erlaubt. So sind z.B. die folgenden Makrodefinitionen zulässig:

```
BIBOPT = -lcurses
objekte = main.o eingabe.o bild.o
323 = dreihundert und dreiundzwanzig
12_drei_gsuffa = Lasst uns Einen heben
LIBDIR = /usr/lib
```

9. Dieses Gleichheitszeichen darf natürlich nicht in einem Kommentar stehen.

10. keine Umlaute oder ß

make ist case-sensitiv, d.h. es unterscheidet Klein- und Großbuchstaben. So sind z.B. *Option* und *option* zwei verschiedene Makronamen.

Obwohl Kleinbuchstaben in Makrodefinitionen erlaubt sind, ist es Konvention, für Makronamen nur Großbuchstaben zu verwenden.

Obwohl neben Buchstaben, Ziffern und Unterstrichen noch andere Zeichen für Makronamen erlaubt sind, ist von deren Benutzung abzuraten, da hieraus oft vermeidbare Fehler resultieren. Werden z.B. Shell-Metazeichen wie », > oder ; benutzt, führt dies fast immer zu einer falschen Interpretation durch **make**.

Zugriff auf Makros mit `$(makroname)` oder `${makroname}`

Auf den Wert (*string*) eines Makronamens kann zugegriffen werden, indem der Makroname mit runden oder geschweiften Klammern umgeben und dieser Klammerung dann ein **\$** vorangestellt wird:

`$(makroname)` oder
`${makroname}`

Dafür wird von **make** der zugehörige *string* aus der Makrodefinition eingesetzt.

Bei Makronamen, die nur aus einem Zeichen bestehen, ist die Angabe von runden bzw. geschweiften Klammern beim Zugriff nicht erforderlich. Wenn z.B. folgende Makrodefinition existiert:

C = `/usr/bin/cc`

so kann auf den String des Makros *C* mit **`$C`**, **`$(C)`** oder **`${C}`** zugegriffen werden.

Zugriff auf andere Makros ist bei der Makrodefinition erlaubt

Bei einer Makrodefinition darf auch auf andere Makros zugegriffen werden. Diese Makros müssen dabei nicht unbedingt vorher, sondern können auch später definiert werden. Wenn z.B. in einem Makefile die folgenden Makrodefinitionen (in der angegebenen Reihenfolge) vorliegen:

```
BASISOBJ = assem.$(EXT) pass1.$(EXT) pass2.$(EXT) fehler.$(EXT)
EXT = o
```

dann wird ein Zugriff mit **`$(BASISOBJ)`** von **make** zu folgendem String expandiert:

```
assemb.o pass1.o pass2.o fehler.o
```

Diese eben erwähnte Konvention ist jedoch gefährlich, wie nachfolgend gezeigt wird:

```
OBJ1 = assemb.o
OBJ1 = $(OBJ1) pass1.o
OBJ1 = $(OBJ1) pass2.o
OBJ1 = $(OBJ1) fehler.o
```

Man erwartet nun, daß folgendes gilt:

```
OBJS1 = assemb.o pass1.o pass2.o fehler.o
```

Tatsächlich gilt aber folgendes:

```
OBJS1 = $(OBJS1) fehler.o
```

da **make** Makros erst dann auflöst, wenn sie benötigt werden. Dieses verspätete Auflösen von Makros mag unsinnig erscheinen, hat aber seinen Sinn, wenn man allgemeine Suffixregeln erstellt, die implizite Abhängigkeiten erzeugen.

Aus diesem Grund wird man in Makefiles oft Angaben wie die folgenden sehen, wenn zu lange Makrodefinitionen vermieden werden sollen:

```
OBJ_1 = assemb.o
OBJ_2 = $(OBJS1) pass1.o
OBJ_3 = $(OBJS1) pass2.o
OBJ_4 = $(OBJS1) fehler.o
OBJS1 = $(OBJ_1) $(OBJ_2) $(OBJ_3) $(OBJ_4)
```

Das GNU-**make** von Linux bietet für solche Angaben eine eigene Zuweisungsform an:

```
OBJS1 := assemb.o
OBJS1 := $(OBJS1) pass1.o
OBJS1 := $(OBJS1) pass2.o
OBJS1 := $(OBJS1) fehler.o
```

Der Operator `:=` veranlaßt das GNU-**make** dazu, bereits bei der Zuweisung die entsprechenden Makros aufzulösen. Daneben bietet das GNU-**make** noch eine elegantere Lösung zu diesem an:

```
OBJS1 := assemb.o
OBJS1 += pass1.o
OBJS1 += pass2.o
OBJS1 += fehler.o
```

Wird in einer Abhängigkeitsbeschreibung auf ein Makro zugegriffen, bevor es definiert ist, so wird dort der Leerstring und nicht der *string* aus der späteren Makrodefinition eingesetzt.

Wird dagegen in einer Kommandozeile auf ein Makro zugegriffen, das erst später definiert ist, so wird bereits dort der erst später definierte *string* eingesetzt.

String-Substitution bei einem Makrozugriff

String-Substitution bedeutet, daß bei einem Makrozugriff die Suffixe von Wörtern aus dem Makro-String durch eine neue Zeichenkette ersetzt werden können. Dazu muß folgende Konstruktion angegeben werden:

```
${makroname:altsuffix=newsuffix}
```


Der String *altsuffix* wird dabei überall dort durch *neusuffix* ersetzt, wo *altsuffix* ein Leer-, Tabulator- oder Neue-Zeile-Zeichen folgt.

Bei der String-Substitution darf die Angabe von *neusuffix* auch weggelassen werden. Es wird dann hierfür der Leer-String angenommen. *altsuffix* muß dagegen immer angegeben sein.

Typische vordefinierte Makros

```
AR = ar
ARFLAGS = rv
AS = as
ASFLAGS =
CC = cc
CFLAGS = -O
F77 = f77
F77FLAGS =
GET = get
GFLAGS =
LD = ld
LDFLAGS =
LEX = lex
LFLAGS =
MAKE = make
MAKEFLAGS = b
YACC = yacc
YFLAGS =
$ = $
```

Interne Makros

\$@ Name des aktuellen Ziels

Für das Makro **\$@** setzt **make** immer das Ziel aus der aktuellen Abhängigkeitsbeschreibung ein. Eine Ausnahme bilden dabei Bibliotheksangaben, wo für **\$@** der Bibliotheksname eingesetzt wird. **\$@** kann auch in Suffixregeln benutzt werden.

\$\$@ Name des aktuellen Ziels in einer Abhängigkeitsbeschreibung

Für das Makro **\$\$@** setzt **make** genau wie bei **\$@** immer das momentane Ziel der aktuellen Abhängigkeitsbeschreibung ein. Die Verwendung von **\$\$@** ist allerdings nur auf der rechten Seite von Abhängigkeitsbeschreibungen und nicht in Kommandozeilen erlaubt. In Suffixregeln darf **\$\$@** benutzt werden.

\$* Name des aktuellen Ziels ohne Suffix

Für das Makro **\$*** setzt **make** immer das momentane Ziel aus der aktuellen Abhängigkeitsbeschreibung ein. Anders als bei **\$@** wird hierbei jedoch ein eventuell vorhandenes Suffix (wie z.B. *.o*, *.c*, *.a*, usw.) entfernt. **\$*** darf nicht in Abhängigkeitsbeschreibungen, sondern nur in den zugehörigen Kommandozeilen oder in Suffixregeln verwendet werden.

\$? Namen von neueren *objekten*

Für das Makro **\$?** setzt **make** aus der aktuellen Abhängigkeitsbeschreibung immer die *objekte* der rechten Seite ein, die neuer als das momentane Ziel sind. **\$?** darf nicht in einer Abhängigkeitsbeschreibung, sondern nur in den zugehörigen Kommandozeilen benutzt werden. In Suffixregeln darf **\$?** nicht benutzt werden.

\$< Name eines neueren *objekts* entsprechend den Suffixregeln

Das interne Makro **\$<** darf nur in Suffixregeln oder beim speziellen Ziel **.DEFAULT** benutzt werden. Dieses Makro **\$<** enthält ähnlich dem Makro **\$?** immer die Namen von neueren Objekten zu einem veralteten Ziel.

\$% Name einer Objektdatei aus einer Bibliothek

Um Objektdateien aus Bibliotheken zu benennen, muß folgende Syntax verwendet werden:

bibliotheksname(objektdatei)

Während das Makro **\$@** in diesem Fall den *bibliotheksname* liefert, liefert das Makro **\$%** den Namen der entsprechenden *objektdatei* aus der Bibliothek. **\$%** kann sowohl in normalen Abhängigkeitsangaben als auch in Suffixregeln verwendet werden.

Die Modifikatoren D und F für interne Makros

Bei allen internen Makros außer **\$?**¹¹ können noch zusätzlich die beiden sogenannten Modifikatoren **D** und **F** angegeben werden. Ihre Angabe bewirkt, daß ähnlich den Kommandos **dirname** und **basename** von einem Pfadnamen entweder nur der Directorypfad (**D**) oder der Dateiname (**F**) genommen wird. Erlaubte und sinnvolle Anwendungen dieser Modifikatoren wären somit:

- ▶ für den Zugriff auf den Basisnamen: **\${@F}**, **\$\${@F}**, **\${*F}**, **\${<F}**
- ▶ für den Zugriff auf den Directorypfad: **\${@D}**, **\$\${@D}**, **\${*D}**, **\${<D}**

Makrodefinitionen auf der Kommandozeile

Makrodefinitionen können **make** auch über die Kommandozeile mitgeteilt werden. Dazu ist die entsprechende Makrodefinition als ein Argument beim **make**-Aufruf anzugeben.

Makrodefinitionen über Shell-Variablen

Auch über Shell-Variablen kann eine Makrodefinition einem Makefile mitgeteilt werden. Man muß dabei nur beachten, daß ein Zugriff auf den Inhalt einer solchen Shell-Variablen nicht wie in der Shell mit *\$variable*, sondern mit **\${variable}** oder **\$(variable)**¹² erfolgen muß.

11. Manche **make**-Versionen lassen die Verwendung der Modifikatoren **D** und **F** jedoch auch für das Makro **\$?** zu.

12. Besteht der Name der *variable* nur aus einem Zeichen, so ist auch der Zugriff mit **\$\$variable** gestattet.

Um sicherzustellen, daß eine Shell-Variable in der für **make** gestarteten Subshell verfügbar ist, gibt es zwei Möglichkeiten:

- ▶ **Exportieren von Shell-Variablen**
Um den Wert einer Shell-Variablen einer Subshell zur Verfügung zu stellen, muß man in der Bourne- und Korn-Shell diese zuvor mit dem Kommando **export** exportieren. In der C-Shell müßte man der Shell-Variablen mit **setenv** den entsprechenden *string* zuweisen.
- ▶ **Zuweisungen an Shell-Variablen direkt vor **make****
Bei einem Aufruf eines Kommandos ist es erlaubt, unmittelbar vor dem Kommandonamen Zuweisungen an Shell-Variablen vorzunehmen. Solche Zuweisungen an Shell-Variablen gelten dann nur für die Dauer der Subshell, die durch diesen Aufruf gestartet wird. Diese Form der Übergabe von Shell-Variablen an Makefiles ist jedoch nur in der Bourne- und Korn-Shell erlaubt.

Prioritäten für Makrodefinitionen

Die Priorität der einzelnen Makrodefinitionen untereinander (von der niedrigsten bis zur höchsten) ist:

1. vordefinierte Makros
2. über Shell-Variablen definierte Makros
3. selbstdefinierte Makros
4. auf der Kommandozeile als Argumente angegebene Makrodefinitionen

Wird **make** dagegen mit der Option **-e** aufgerufen, so gelten folgende Prioritäten (von der niedrigsten bis zur höchsten):

1. vordefinierte Makros
2. **selbstdefinierte Makros**
3. **über Shell-Variablen definierte Makros**
4. auf der Kommandozeile als Argumente angegebene Makrodefinitionen

22.8.4 Suffixregeln

Bestimmte Dateinamen erfordern immer die gleichen Generierungsschritte. So ist z.B. für C-Programmdateien (Suffix *.c*) immer ein Aufruf des C-Compilers notwendig, um daraus eine Objektdatei (Suffix *.o*) generieren zu lassen. Für solche fest vorgegebenen Generierungsschritte sind von **make** sogenannte Suffixregeln vordefiniert, wie z.B.:

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

Falls nun für eine Objektdatetei keine explizite Generierungsregel im Makefile vorgegeben ist, verwendet **make** seine vordefinierten Suffixregeln und kann so selbst die erforderlichen Generierungsschritte ermitteln.

Eine Suffixregel kann auch vom Benutzer definiert werden und ist grundsätzlich wie folgt aufgebaut:

```
.von.nach:
    kommandozeilen
```

Eine solche Regel legt fest, welche Suffixabhängigkeiten gelten, nämlich daß Dateien mit dem Suffix *.nach* immer aus Dateien mit dem Suffix *.von* generiert werden. Die dazu notwendigen Generierungsschritte werden dabei über die *kommandozeilen* festgelegt.

Bei den meisten **make**-Versionen können auch Suffixregeln definiert werden, bei denen nur ein Suffix (anstelle von zwei) angegeben ist. Anstelle von

```
.von.nach:
```

wird dann nur

```
.von:
```

angegeben. *.nach* wird also nicht angegeben. Ein solches nicht angegebenes Suffix bezeichnet man oft auch als Null-Suffix.

So ist z.B. für C-Programme von den meisten **make**-Versionen die folgende Suffixregel vordefiniert:

```
.c:
    $(CC) $(CFLAGS) $< $(LDFLAGS) -o $@
```

Solche Suffixregeln sind besonders dann von Vorteil, wenn sich Programme aus nur einem Modul zusammensetzen, da sie festlegen, wie z.B. *prog* aus *prog.c* zu generieren ist. Ruft man z.B. **make prog** auf, dann würde **make** auch ohne Vorhandensein eines Makefiles folgendes aufrufen:

cc -O prog.c -o prog

Kompilieren und Linken wird also, da die Option **-c** nicht vorhanden ist, mit einem **cc**-Aufruf durchgeführt.

SCCS-Dateien lassen sich immer daran erkennen, daß sie mit dem Präfix *s.* beginnen. Suffixe, die sich auf SCCS-Dateien beziehen, werden in **make** immer durch ein Anhängen des Zeichens *~* (Tilde-Zeichen) an das Suffix gekennzeichnet.

Vordefinierte Suffixregeln

Um sich alle für einen **make**-Lauf vordefinierten Suffixregeln ausgeben zu lassen, könnte z.B. folgendes aufgerufen werden:

```
make -pf - 2>/dev/null </dev/null
```

Um alle vordefinierten Suffixregeln auszuschalten, muß beim **make**-Aufruf die Option **-r** angegeben werden. Sind nur einzelne Suffixregeln auszuschalten, so kann dies mit folgender Angabe im Makefile erfolgen:

```
.von.nach: ;
```

Das spezielle Ziel .SUFFIXES

Alle für **make** relevanten Suffixe müssen nach dem speziellen Ziel **.SUFFIXES** angegeben sein. Die Default-Einstellung für **.SUFFIXES** ist z.B. beim GNU-make unter Linux:

```
SUFFIXES := .out .a .ln .o .c .cc .C .p .f .F .r .y .l .s .S .mod .sym .def .h .info .dvi .tex  
.texinfo .texi .txinfo .w .ch .web .sh .elc .el
```

Die Reihenfolge der Suffixangaben legt dabei ihre Priorität für **make** fest. Möchte der Benutzer eigene Suffixe mit zugehörigen Suffixregeln definieren, so muß er diese neuen Suffixe in seinem Makefile nach **.SUFFIXES:** angeben. Diese werden dann zu den bereits definierten Suffixen hinzugefügt. Um alle von **make** vordefinierten Suffixregeln auszuschalten, müßte der Benutzer zunächst nur

```
.SUFFIXES:
```

ohne irgendwelche Suffixe angeben. Danach könnte er dann mit einer neuen Angabe

```
.SUFFIXES: .....
```

die Suffixe festlegen, die für diesen **make**-Lauf relevant sein sollen, und für die er eigene Suffixregeln im Makefile definiert hat.

22.8.5 Spezielle Zielangaben

.DEFAULT:

Verwendet man in einem Makefile Dateien, für deren Generierung weder explizit Kommandos angegeben sind noch irgendwelche Suffixregeln existieren, so führt **make** beim Fehlen einer solchen Datei immer die nach dem speziellen Ziel **.DEFAULT:** angegebenen Kommandos zur Generierung der betreffenden Datei aus.

.IGNORE:

Normalerweise bricht **make** beim Auftreten eines Fehlers sofort die ganze Generierung ab. Soll **make** aber grundsätzlich alle auftretenden Fehler ignorieren, so muß man folgende Zeile in einem Makefile angeben:

```
.IGNORE:
```

Das gleiche erreicht man auch, wenn man beim **make**-Aufruf die Option **-i** angibt.

.PRECIOUS:

Wenn die Generierung eines Programms mit der *intr*- oder *quit*-Taste abgebrochen wird, dann entfernt **make** immer zuerst das aktuelle Ziel, bevor es die Generierung abbricht. Wenn nun bestimmte Ziele beim Auftreten eines Fehlers nicht zu löschen sind, so muß man im Makefile die entsprechenden Zielnamen mit

```
.PRECIOUS: ziel1 ziel2 ....
```

angeben. Eine solche Angabe kann an einer beliebigen Stelle im Makefile stehen und bewirkt, daß **make** diese Ziele *ziel1* *ziel2* und alle davon abhängigen Ziele nicht entfernt, bevor es den **make**-Lauf abbricht.

.SILENT:

Das vollständige Ausschalten der automatischen Ausgabe von Kommandos vor ihrer Ausführung erreicht man auch durch die folgende Angabe in einem Makefile:

```
.SILENT:
```

In diesem Fall ist dann für dieses Makefile die automatische Ausgabe durch **make** immer ausgeschaltet. Das gleiche erreicht man auch, wenn man beim **make**-Aufruf die Option **-s** angibt.

.SUFFIXES:

siehe Seite 1121.

A

Headerdatei eighdr.h und Modul fehler.c

A.1 Headerdatei eighdr.h

Die meisten Programme (Beispiele und Übungen) in diesem Buch verwenden die Headerdatei eighdr.h. Diese Headerdatei macht wichtige System-Headerdateien, die fast immer benötigt werden (<sys/types.h>, <stdio.h>, <stdlib.h>, <string.h> und <unistd.h>), zum Bestandteil (**#include**) des jeweiligen Programms, so daß in den betreffenden Programmen auf diese #include's verzichtet werden kann, was die Programme kürzer macht und dem Programmierer Schreibarbeit erspart. Daneben enthält die Headerdatei eighdr.h noch nützliche Konstanten-, Makro- und Datentypdefinitionen. Auch enthält sie Prototypdeklarationen von einigen wichtigen Funktionen, die im Rahmen der Arbeit an diesem Buch entwickelt wurden.

```
#ifndef __EIGHDR
#define __EIGHDR

/*-- Headerdatei, die alle wichtigen System-Headerdateien included und ----*/
/*--           wichtige Konstanten und Makros definiert           ----*/
/*--           (sollte nach allen System-Headerdateien included werden) ----*/

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_ZEICHEN    4096    /*--- Maximale Pufferlaenge */

#define WARNUNG        0    /*--- Kennungen fuer unterschiedl. Fehlerarten */
#define WARNUNG_SYS    1
#define FATAL          2
#define FATAL_SYS      3
#define DUMP           4

extern int  debug; /* Aufrufer von log_meld oder log_open muss debug setzen:
                    0, wenn interaktiv; 1, wenn Daemon-Prozess */

/*----- Nuetzliche Makros -----*/
#define min(x,y)      ((x) < (y) ? (x) : (y))
#define max(x,y)      ((x) > (y) ? (x) : (y))

/*----- Eigene Typdefinitionen -----*/
typedef enum { FALSE=0, TRUE=1 } bool;
typedef void  sigfunk(int); /* Datentyp fuer Signalhandler */
```

```

/*----- Zentrale Fehlerroutinen -----*/
extern void fehler_meld(int kennung, const char *fmt,...);
extern void log_meld(int kennung, const char *fmt,...);

/*----- log_open -----
        initialisiert syslog() bei einem Daemon-Prozess */
extern void log_open(const char *kennung, int option, int facility);

/*----- Synchronisationroutinen -----*/
extern void INIT_SYNCH(void); /* Synchronisation initialisieren */
extern void HALLO_PAPA(pid_t pid); /* Kind signal. Elternpr., dass fertig */
extern void WARTE_AUF_PAPA(void); /* Kind wartet auf Signal vom Elternpr. */
extern void HALLO_KIND(pid_t pid); /* Elternpr. signal. Kind, dass fertig */
extern void WARTE_AUF_KIND(void); /* Elternpr. wartet auf Signal vom Kind */

/*----- Funktionen aus sperre.c -----*/
extern int sperre_einaus(int fd, int kdo, int sperr_typ,
                        off_t offset, int wie, off_t laenge);
extern pid_t sperre_testen(int fd, int sperr_typ,
                          off_t offset, int wie, off_t laenge);

/*----- Einrichten einer Sperre -----*/
#define lese_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLK, F_RDLCK, offset, wie, laenge)
#define lesewarte_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLKW, F_RDLCK, offset, wie, laenge)
#define schreib_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLK, F_WRLCK, offset, wie, laenge)
#define schreibwarte_sperre(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLKW, F_WRLCK, offset, wie, laenge)

/*----- Aufheben einer Sperre -----*/
#define sperre_aufheben(fd,offset,wie,laenge) \
    sperre_einaus(fd, F_SETLK, F_UNLCK, offset, wie, laenge)

/*----- Testen einer Sperre -----*/
#define lesesperre_vorhanden(fd,offset,wie,laenge) \
    sperre_testen(fd, F_RDLCK, offset, wie, laenge)
#define schreibsperre_vorhanden(fd,offset,wie,laenge) \
    sperre_testen(fd, F_WRLCK, offset, wie, laenge)

#endif

```

Programm A.1 Headerdatei **eighdr.h**: Eigene Headerdatei, die in den meisten Programmen verwendet wird

A.2 Zentrales Fehlermeldungsmodul fehler.c

Das Programm `fehler.c` wird von den meisten Programmen in diesem Buch zur Ausgabe von Fehlermeldungen benutzt. Es bietet dazu die drei globalen und von jedermann benutzbaren Routinen `fehler_meld`, `log_meld` und `log_open` an.

Während `fehler_meld` auf die Standardfehlerausgabe schreibt, verwendet `log_meld` die Funktion `syslog` zur Ausgabe der entsprechenden Fehlermeldung. `log_meld` wird von Dämonprozessen verwendet. Zum Initialisieren von `syslog` muß zunächst `log_open` aufgerufen werden.

Die Parameter der beiden Funktionen `fehler_meld` und `log_meld` sind identisch. Das erste Argument legt dabei fest, wie der entsprechende Fehler zu behandeln ist. Es sind die folgenden in `eighdr.h` definierten Konstanten als erstes Argument erlaubt:

```
WARNUNG
WARNUNG_SYS
FATAL
FATAL_SYS
DUMP
```

Es wurde dabei die folgende Regelung bei der Vergabe der Konstantennamen gewählt:

- ▶ Die Endung `SYS` bedeutet, daß zusätzlich zur eigenen Meldung noch die zum entsprechenden Fehler gehörende Systemfehlermeldung auszugeben ist.
- ▶ Nur bei den `WARNUNG`-Konstanten bewirkt die Fehlerroutine nicht die Beendigung des gesamten Programms.
- ▶ Bei Angabe der `FATAL`- und `DUMP`-Konstanten bewirkt die Fehlerroutine einen Programmabbruch. Nur bei der `DUMP`-Konstante wird mit ***abort*** das Programm beendet und ein *core dump* (Speicherabzug) erzeugt. Bei `FATAL` und `FATAL_SYS` wird das Programm mittels `exit(1)` beendet.

Die weiteren Argumente zu `fehler_meld` entsprechen denen bei einem ***printf***-Aufruf.

```
#include <errno.h>
#include <stdarg.h>
#include <syslog.h>
#include "eighdr.h"

int debug; /* Aufrufer von log_meld oder log_open muss debug setzen:
            0, wenn interaktiv; 1, wenn Daemon-Prozess */
/*---- Lokale Routinen zur Abarbeitung der Argumentliste -----*/
static void fehl_meldung(int sys_meld, const char *fmt, va_list az)
{
    int fehler_nr = errno;
    char puffer[MAX_ZEICHEN];

    vsprintf(puffer, fmt, az);
    if (sys_meld)
        sprintf(puffer+strlen(puffer), ": %s ", strerror(fehler_nr));
    fflush(stdout); /* fuer Fall, dass stdout und stderr gleich sind */
    fprintf(stderr, "%s\n", puffer);
    fflush(NULL); /* alle Ausgabepuffer flushen */
    return;
}

static void log_meldung(int sys_meld, int prio, const char *fmt, va_list az)
```

```

{
    int fehler_nr = errno;
    char puffer[MAX_ZEICHEN];

    vsprintf(puffer, fmt, az);
    if (sys_meld)
        sprintf(puffer+strlen(puffer), ": %s ", strerror(fehler_nr));
    if (debug) {
        fflush(stdout); /* fuer Fall, dass stdout und stderr gleich sind */
        fprintf(stderr, "%s\n", puffer);
        fflush(NULL); /* alle Ausgabepuffer flushen */
    } else {
        strcat(puffer, "\n");
        syslog(prio, puffer);
    }
    return;
}

/*---- Global aufrufbare Fehlerrouinen -----*/
void fehler_meld(int kennung, const char *fmt, ...)
{
    va_list az;

    va_start(az, fmt);
    switch (kennung) {
        case WARNUNG:
        case FATAL:
            fehl_meldung(0, fmt, az);
            break;
        case WARNUNG_SYS:
        case FATAL_SYS:
        case DUMP:
            fehl_meldung(1, fmt, az);
            break;
        default:
            fehl_meldung(1, "Falscher Aufruf von fehler_meld...", az);
            exit(3);
    }
    va_end(az);

    if (kennung==WARNUNG || kennung==WARNUNG_SYS)
        return;
    else if (kennung==DUMP)
        abort(); /* core dump */
    exit(1);
}

void log_meld(int kennung, const char *fmt, ...)
{
    va_list az;

    va_start(az, fmt);
    switch (kennung) {

```

```
    case WARNUNG:
    case FATAL:
        log_meldung(0, LOG_ERR, fmt, az);
        break;
    case WARNUNG_SYS:
    case FATAL_SYS:
        log_meldung(1, LOG_ERR, fmt, az);
        break;
    default:
        log_meldung(1, LOG_ERR, "Falscher Aufruf von fehler_meld...", az);
        exit(3);
}
va_end(az);

if (kennung==WARNUNG || kennung==WARNUNG_SYS)
    return;
exit(2);
}
/*---- log_open -----*/
        initialisiert syslog() bei einem Daemon-Prozess                */
void log_open(const char *kennung, int option, int facility)
{
    if (debug==0)
        openlog(kennung, option, facility);
}
```

*Programm A.2 Programm **fehler.c**: Zentrales Fehlermeldungsmodul*

B Ausgewählte Lösungen zu den Übungen

Hier finden Sie einige ausgewählte Lösungen zu den Übungen. Alle Programmlistings, die Lösungen zu den einzelnen Übungen sind, können ebenso wie alle Beispielprogramme online von der WWW-Adresse <http://www.addison-wesley.de/service/herold/syprog.tgz> heruntergeladen werden.

B.1 Ausgewählte Lösungen zu Kapitel 4 (Elementare E/A-Funktionen)

B.1.1 Duplizieren und mehrmaliges Öffnen derselben Datei

Jeder **open**-Aufruf liefert einen neuen Dateitabelleneintrag. Da in diesem Fall beide **open**-Aufrufe die gleiche Datei (`datei1`) öffnen, zeigen beide Dateitabelleneinträge auf den gleichen Eintrag in der v-node-Tabelle. Jeder **dup**-Aufruf dupliziert den entsprechenden Filedeskriptor in der Prozeßtable, so daß sich nach diesen Aufrufen die in Abbildung B4.1 gezeigte Konstellation ergibt.

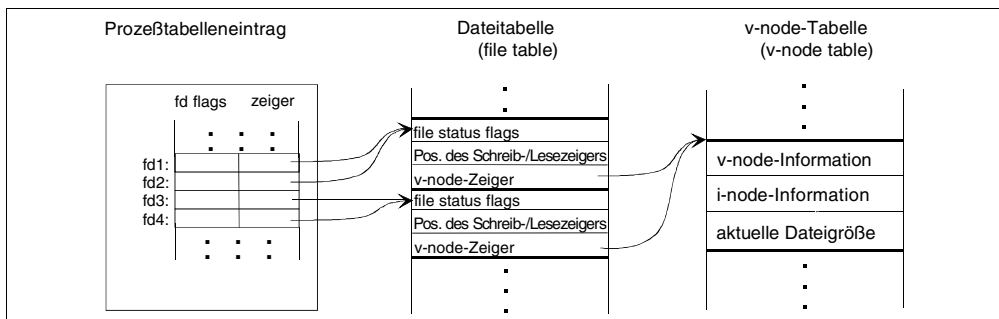


Abbildung B.1: Konstellation nach Duplizieren und mehrmaligem Öffnen derselben Datei

Ein `fcntl` mit `F_SETFD` setzt nur die entsprechenden **fdflags** des jeweils angegebenen Filedeskriptors `fd1`, `fd2`, `fd3` oder `fd4`. Dagegen würde z.B. ein `fcntl` mit `F_SETFL` die file status flags im entsprechenden Dateitabelleneintrag setzen, was bedeutet, daß dies hier immer Auswirkung auf zwei Filedeskriptoren hat. Abbildung B4.1 verdeutlicht dies. So würde z.B. ein `F_SETFL` auf `fd1` zugleich auch Auswirkung auf `fd2` haben; umgekehrt gilt dies auch. Dasselbe trifft auch auf die beiden Filedeskriptoren `fd3` und `fd4` zu.

B.1.2 Nachvollziehen einer Notation in der Korn-Shell

kdo *>aus 2>&1*

Hier wird zuerst die Standardausgabe in die Datei *aus* umgelenkt, dann wird der Filedeskriptor für die Standardausgabe (1) mit *dup2* dupliziert und auf die Standardfehlerausgabe (2) gelegt. Dies führt dazu, daß bei diesem Aufruf sowohl die Standardausgabe als auch die Standardfehlerausgabe in die Datei *aus* umgelenkt werden.

kdo *2>&1 >aus*

Hier wird zuerst der Filedeskriptor 1 dupliziert, so daß sowohl die Standardausgabe (1) als auch die Standardfehlerausgabe (2) auf das Terminal eingestellt sind. Erst dann wird die Standardausgabe (1) in die Datei *aus* umgelenkt. Dies führt dazu, daß bei diesem Aufruf die Standardausgabe auf die Datei *aus* und die Standardfehlerausgabe auf das Terminal eingestellt sind.

B.2 Ausgewählte Lösungen zu Kapitel 5 (Dateien, Directories und ihre Attribute)

B.2.1 Makro S_ISLNK für SVR4

Um sich ein eigenes Makro *S_ISLNK* zu definieren, wäre z.B. die folgende Angabe denkbar:

```
#if !defined(S_ISLNK) && defined(S_IFLNK)
#define S_ISLNK(modus)    (((modus) & S_IFMT) == S_IFLNK)
#endif
```

B.2.2 Ändern der Zugriffsrechte existierender Dateien mit *creat* oder *open*

Wenn man versucht, eine bereits existierende Datei mit *open* oder *creat* neu anzulegen, so bleiben deren alten Zugriffsrechte erhalten und werden nicht durch die Angaben beim *open*- bzw. *creat*-Aufruf geändert. Der nachfolgende Ablauf verdeutlicht dies.

```
$ rm um1 um2      [Löschen der Dateien um1 und um2]
$ who >um1        [Dateien um1 und um2 neu anlegen]
$ who >um2
$ chmod a-r um1 um2 [Alle Leserechte fuer um1 und um2 entziehen]
$ ls -l um1 um2    [Anzeigen der aktuellen Zugriffsrechte]
--w----- 1 hh    bin          62 Jun 23 10:42 um1
--w----- 1 hh    bin          62 Jun 23 10:42 um2
$ umaskdem        [Aufrufen von Programm 5.4 (umaskdem.c)]
$ ls -l um1 um2   [Zugriffsrechte haben sich nicht geändert]
--w----- 1 hh    bin          0 Jun 23 10:44 um1
--w----- 1 hh    bin          0 Jun 23 10:44 um2
$
```

B.2.3 unlink und Zeit der letzten i-node-Änderung

unlink erniedrigt den Link-Zähler der entsprechenden Datei um 1. Dieses Erniedrigen hat die Auswirkung, daß die Zeit der letzten i-node-Änderung aktualisiert, also verändert wird. Wenn allerdings der Link-Zähler bereits 1 ist, so wird durch ein **unlink** die letzte Referenz auf diese Datei entfernt, was zur Folge hat, daß der ganze i-node entfernt wird und somit eine Aktualisierung der Zeit der letzten i-node-Änderung nicht mehr sinnvoll ist.

B.2.4 Maximale Tiefe eines Directory-Baums

Der Unixkern kennt zwar kein Limit für die Tiefe eines Directory-Baums, aber viele Kommandos schlagen fehl, wenn sie mit Pfadnamen umgehen müssen, die länger als `PATH_MAX` sind. Das folgende Programm `treetief.c` erzeugt einen Directory-Baum, der 50 Ebenen tief ist. Als Directory-Namen wählt es dabei immer einen sehr langen Namen »Allmaecht.....«. Nachdem es diesen Directorybaum erfolgreich angelegt hat, erfragt es mit `getcwd` den Pfadnamen der tiefsten Ebene. Es benötigt dazu mehrere `getcwd`-Aufrufe, da es sich langsam (in 100er Schritten) an die Länge dieses Pfadnamens, für den es ja Speicherplatz zur Verfügung stellen muß, herantastet.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include "eighdr.h"

#define HOMEDIR "/home/hh"
#define DIRNAME "AllmaechtIstDasEinLangerDirectoryName"

int
main(void)
{
    int    i, groesse = PATH_MAX;
    char   *pfad;

    if (chdir(HOMEDIR) < 0)
        fehler_meld(FATAL_SYS, "chdir-Fehler");

    /*-- Kreieren eines Directorybaums mit 50 Subdirectories,
       wobei jedes den sehr langen Namen "AllmaechtIst....." hat */
    for (i=0; i<50; i++) {
        if (mkdir(DIRNAME, S_IRWXU | S_IRGRP|S_IXGRP | S_IROTH|S_IXOTH) < 0)
            fehler_meld(FATAL_SYS, "mkdir-Fehler bei i=%d", i);
        if (chdir(DIRNAME) < 0)
            fehler_meld(FATAL_SYS, "chdir-Fehler bei i=%d", i);
    }
    if (creat("Blattdatei", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        fehler_meld(FATAL_SYS, "creat-Fehler");

    /*-- Ausgabe des eben erzeugten sehr langen Pfadnamen ----*/
    if ( (pfad = malloc(groesse)) == NULL)
```

```

    fehler_meld(FATAL_SYS, "Speicherplatzmangel");
while (1) {
    if (getcwd(pfad, groesse) != NULL)
        break;
    else {
        fehler_meld(WARNUNG_SYS, "getcwd-Fehler, groesse=%d", groesse);
        groesse += 100;
        if ( (pfad = realloc(pfad, groesse)) == NULL)
            fehler_meld(FATAL_SYS, "Speicherplatzmangel");
    }
}
printf("Laenge des folgenden Pfadnamens: %d\n", strlen(pfad));
printf("%s\n", pfad);

exit(0);
}

```

Programm B.1 **treetief.c**: Kreieren eines Directory-Baums mit 50 Ebenen

Für dieses Programm *treetief.c* könnte sich z.B. der folgende Ablauf ergeben.

```

$ treetief
getcwd-Fehler, groesse=1024: Math result not representable
getcwd-Fehler, groesse=1124: Math result not representable
getcwd-Fehler, groesse=1224: Math result not representable
getcwd-Fehler, groesse=1324: Math result not representable
getcwd-Fehler, groesse=1424: Math result not representable
getcwd-Fehler, groesse=1524: Math result not representable
getcwd-Fehler, groesse=1624: Math result not representable
getcwd-Fehler, groesse=1724: Math result not representable
getcwd-Fehler, groesse=1824: Math result not representable
Laenge des folgenden Pfadnamens: 1908
/home/hh/AllmaechtIstDasEinLangerDirectoryname/AllmaechtIstDasEinLangerDirectoryname/
..... [vollstaendiger Pfadname (mit 1908 Bytes)]
$

```

B.2.5 Root-Directory eines Prozesses

Die **chroot**-Funktion wird von FTP (*File Transfer Program*) benutzt, um fremde Benutzer ohne eigenen Loginnamen am lokalen System (*anonymous FTP*) in ein eigenes Directory unterzubringen, das für sie mit **chroot** als deren Root-Directory eingerichtet wird. So wird verhindert, daß solche fremden Benutzer in ein anderes Directory wechseln oder auf andere Dateien im Directory-Baum zugreifen können.

chroot kann auch verwendet werden, um eine Kopie des Filesystems an einer anderen Stelle herzustellen. Diese Kopie kann dann beliebig modifiziert werden (z.B. Installation und Test eines neuen Softwareprodukts), ohne daß dies Einfluß auf das originale Filesystem hat.

B.3 Ausgewählte Lösungen zu Kapitel 7 (Datums- und Zeitfunktionen)

B.3.1 Letztes Jahr bei 32 Bit für time_t

Während des Jahres 2038.

B.3.2 Maximale Prozeßlaufzeit bei 32 Bit für clock_t

Ungefähr nach 248 Tagen.

B.4 Ausgewählte Lösungen zu Kapitel 8 (Nicht-lokale Sprünge)

B.4.1 Mehrfaches Aufrufen von setjmp

Bei longjmp wird immer zu dem Programmzustand zurückgekehrt, der in der angegebenen jmp_buf-Variable mit setjmp hinterlegt wurde. Das Programm 8.4 (zweijmp.c) würde z.B. folgendes ausgeben:

```
$ zweijmp
.....Rueckkehr von b ----> a.....
.....Rueckkehr von c ----> main.....
.....Rueckkehr von b ----> a.....
$
```

B.4.2 Rückkehr zu einer nicht mehr im Stack vorhandenen Funktion

Eine Rückkehr zu einer nicht mehr im Stack vorhandenen Funktion muß zwangsläufig zu einem fehlerhaften Programmverlauf führen. Das Programm 8.5 (overjmp.c) würde z.B. folgendes ausgeben:

```
$ overjmp
.....Rueckkehr von d ----> c.....
Segmentation fault      [Anormale Programmbeendigung]
$
```

B.5 Ausgewählte Lösungen zu Kapitel 9 (Der Unix-Prozeß)

B.5.1 Ändern des Environment eines Elternprozesses nicht möglich

Ein aktueller Prozess verändert immer nur seine Environment-Liste, die sich am Anfang seines Speichers befindet (siehe auch Abbildung 9.3). Da er niemals Zugriff auf den Speicherbereich des Elternprozesses hat, kann er dort auch keine Änderungen in der Environment-Liste vornehmen. Ein Vererben von Environment-Variablen an Kindprozesse ist dagegen möglich, denn der Kern muß beim Start von Kindprozessen nur die zum Export markierten Variablen in das Environment des Kindprozesses kopieren.

B.5.2 Zugriff auf Adresse 0 des Datensegments meist nicht möglich

Die Klassifizierung von 0 als unerlaubte Adresse ermöglicht es, die Zeigerkonstante `NULL` nachzubilden, die oft mit `0`, `0L` oder `(void *)0` definiert ist. Diese Vereinbarung bewirkt, daß jeder (von C her) unerlaubte Zugriff über einen Zeiger, der mit `NULL` gesetzt ist, zu einem automatischen Abbruch des entsprechenden Prozesses führt.

B.5.3 Gefahren bei der Verwendung von lokalen Variablen

a) Eine elegante Allokierungsroutine, oder nicht ?

Die Funktion `allokier` ist zwar elegant, aber falsch. Sie allokiert mit

```
char array[groesse];
```

auf dem Stack lokalen Speicher von `groesse` Bytes und gibt dann die Anfangsadresse dieses Speichers an den Aufrufer zurück. Da jedoch nach dem Verlassen einer Funktion der lokal auf dem Stack allokierte Speicherplatz nicht mehr zur Verfügung steht, ist die dem Aufrufer zurückgegebene Adresse nicht mehr gültig. Also ist mit solchen Konstruktionen größte Vorsicht geboten.

Nach einer Rückkehr aus einer Funktion wäre dagegen die Adresse eines mit `malloc`, `calloc` oder `realloc` (auf dem Heap) allokierten Speicherplatzes auch weiterhin gültig.

b) Rückgabe eines Zeigers auf eine lokale Variable

Dieser Code ist inkorrekt, weil er in der Zeigervariablen `zgr` die Adresse der lokalen Variablen `ergeb` speichert und auch noch zurückgibt. Die Variable `ergeb` existiert aber nur für die Dauer des inneren Blocks. Nach dem Verlassen dieses Blocks ist diese Variable `ergeb` nicht mehr vorhanden und somit ist auch die zuvor an `zgr` zugewiesene Adresse nicht mehr gültig.

c) Schreiben in eine Struktur über einen Zeiger

Dieser Programmausschnitt zeigt einen häufigen C-Fehler. Man deklariert nur einen Zeiger auf eine Struktur

```
struct adresse *zgr;
```

ohne den dazugehörigen Speicherplatz zu allokieren. Später schreibt man dann über den Zeiger in diese Struktur:

```
strcpy(zgr->name, "Hans Mayer");  
zgr->alter = 10;
```

Da der Zeiger-Variablen `zgr` aber nirgends ein definierter Wert (Adresse) zugewiesen wurde, findet hier ein Überschreiben von fremdem Speicherplatz statt. Richtig wäre z.B.

```
struct adresse adr;  
struct adresse *zgr = &adr;
```

Nun hat `zgr` eine definierte Adresse und man kann mit `zgr` in die Struktur (hier Variable `adr`) schreiben.

B.6 Ausgewählte Lösungen zu Kapitel 10 (Die Prozeßsteuerung)

B.6.1 Kreieren eines Zombies

```
#include "eighdr.h"  
  
int  
main(void)  
{  
    pid_t pid;  
  
    if ( (pid = fork()) < 0 )  
        fehler_meld(FATAL_SYS, "fork-Fehler");  
  
    else if (pid == 0) /*---- Kind ----*/  
        exit(0);  
  
    /*----- Elternprozess -----*/  
    sleep(5);  
    system("ps");  
    exit(0);  
}
```

Programm B.2 **pszombie.c**: Kreieren und Anzeigen eines Zombieprozesses

Nachdem man dieses Programm `pszombie.c` kompiliert und gelinkt hat

```
cc -o pszombie pszombie.c fehler.c
```

ergibt sich z.B. der folgende Ablauf:

```
$ pszombie
PID TTY STAT   TIME COMMAND
 58 v02 S      0:02 -tcsh
117 v02 S      0:00 pszombie
118 v02 Z      0:00 (pszombie) <zombie>    [Dies ist der Zombie-Prozeß]
119 v02 R      0:00 ps
$
```

B.6.2 Vorsicht bei Aufruf von `vfork` in einer anderen Funktion als `main`

Das Programm `vforkfal.c` führt auf Systemen, in denen **`vfork`** nicht mit dem früher vorgestellten COW-Verfahren arbeitet, zu Problemen (meist Programmabsturz mit Anlegung einer `core`-Datei).

Das Problem liegt dabei darin, daß bei **`vfork`** der Kindprozeß zuerst gestartet wird. Dieser Kindprozeß verläßt zunächst die Funktion `a` und ruft sofort die Funktion `b` auf. In der Funktion `b` schreibt dieser Kindprozeß 100 Nullen auf den Stack, bevor er sich mit **`_exit`** beendet.

Wenn nun der Elternprozeß zur Ausführung kommt, ist der von beiden Prozessen benutzte Stack bereits vom Kindprozeß (durch die Rückkehr aus Funktion `a` und dem Aufruf von Funktion `b` mit anschließendem Schreiben) verändert. Da die Rückkehrinformation meist auch im Stack untergebracht ist, ist diese Rückkehrinformation des Elternprozesses (von Funktion `a` zurück nach `main`) durch den Kindprozeß nun zerstört und der Elternprozeß greift sehr wahrscheinlich auf ungültige Adressen zu, was zwangsläufig zum Programmabsturz führt.

B.6.3 Erfragen der eigenen saved Set-User-ID durch einen Prozeß

Es existiert keine eigene Funktion zum Erfragen der eigenen saved Set-User-ID. Statt dessen müßte der betreffende Prozeß zum Zeitpunkt seines Starts seine effektive User-ID selbst in einer eigenen Variablen sichern, um sie dann später zur Verfügung zu haben.

B.7 Ausgewählte Lösungen zu Kapitel 11 (Attribute eines Prozesses)

B.7.1 Kreieren einer neuen Session durch einen Kindprozeß

```
#include "eighdr.h"

int
main(void)
{
    pid_t    pid, vorder_grp;

    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid == 0) { /*----- Kindprozess -----*/
        if (setsid() == -1)
            fehler_meld(FATAL_SYS, "setsid-Fehler");
        printf(" Kindprozess: PID=%d, PPID=%d, GRP-Fuehrer=%d, ",
               getpid(), getppid(), getpgrp());
        if ( (vorder_grp = tcgetpgrp(STDIN_FILENO)) == -1)
            fehler_meld(FATAL_SYS, "tcgetpgrp-Fehler");
        printf("Vorder-GRP: %d\n", vorder_grp);
        exit(0);

    } else { /*----- Elternprozess -----*/
        sleep(3); /* Sicherstellen, dass Kind bereits Session kreiert */
        printf("Elternprozess: PID=%d, PPID=%d, GRP-Fuehrer=%d, ",
               getpid(), getppid(), getpgrp());
        if ( (vorder_grp = tcgetpgrp(STDIN_FILENO)) == -1)
            fehler_meld(FATAL_SYS, "tcgetpgrp-Fehler");
        printf("Vorder-GRP: %d\n", vorder_grp);
        exit(0);
    }
}
```

*Programm B.3 **kindsess.c**: Kreieren einer Session durch Kindprozeß*

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o kindsess kindsess.c fehler.c
```

könnte sich z.B. der folgende Ablauf ergeben:

```
$ kindsess
 Kindprozess: PID=325, PPID=324, GRP-Fuehrer=325, tcgetpgrp-Fehler: Not a typewriter
Elternprozess: PID=324, PPID=58, GRP-Fuehrer=324, Vorder-GRP: 324
$
```

B.7.2 Kontrollterminal für eine verwaiste Prozeßgruppe

```
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include "eighdr.h"
static void print_ids(char *name);
int
main(void)
{
    int    zeich;
    pid_t  pid;

    print_ids("Elternprozess");
    if ( (pid = fork()) < 0)
        fehler_meld(FATAL_SYS, "fork-Fehler");

    else if (pid > 0) { /*--- Elternprozess ----*/
        sleep(5); /* Sicherstellen, dass Kind sich selbst angehalten hat */
        exit(0); /* Elternprozess beendet sich */

    } else { /*--- Kindprozess -----*/
        print_ids("Kindprozess");
        sleep(10);
        print_ids("Kind");
        if (read(0, &zeich, 1) != 1)
            fehler_meld(FATAL_SYS, "Lesefehler vom Kontrollterminal");
        exit(0);
    }
}

static void print_ids(char *name)
{
    printf("%s: pid=%d, ppid = %d, pgrp = %d\n",
           name, getpid(), getppid(), getpgrp());
    fflush(stdout);
}
```

Programm B.4 waisgrp.c: Kindprozeß wird Mitglied einer verwaisten Prozeßgruppe

Nachdem man dieses Programm kompiliert und gelinkt hat

```
cc -o waisgrp waisgrp.c fehler.c
```

könnte sich z.B. der folgende Ablauf ergeben.

```
$ waisgrp
Elternprozess: pid=726, ppid = 58, pgrp = 726
Kindprozess: pid=727, ppid = 726, pgrp = 726
Kindprozess: pid=727, ppid = 1, pgrp = 726
Lesefehler vom Kontrollterminal: I/O error
$
```

Wie man sieht, hat der Kindprozeß kein Kontrollterminal mehr.

B.8 Ausgewählte Lösungen zu Kapitel 13 (Signale)

B.8.1 Implementierung der Funktion raise

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

int raise(int signr)
{
    return( kill(getpid(), signr) );
}
```

Programm B.5 **raise.c**: Mögliche Implementierung der Funktion raise

B.8.2 Nicht-lokaler Sprung unmittelbar nach alarm

Die Gefahr bei diesem Codeausschnitt liegt zwischen dem **alarm**-Aufruf und dem **setjmp**-Aufruf. Wenn der Prozeß zwischen diesen beiden Aufrufen (durch ein Signal) vom Kern blockiert wird, so wird die Zeitschaltuhr ausgeschaltet und der entsprechende Signalhandler aufgerufen. Im Signalhandler wird nun **longjmp** aufgerufen. Da aber zuvor noch kein **setjmp** aufgerufen wurde, ist die Variable `progzust` noch nicht initialisiert und der **longjmp**-Aufruf wird sehr wahrscheinlich zum Programmabsturz führen.

B.8.3 Umständliche Beendigung bei der abort-Implementierung

Mit **_exit** würde der Beendigungsstatus des Prozesses nicht anzeigen, daß der Prozeß durch das Signal SIGABRT beendet wurde.

B.8.4 Aufruf einer nicht-reentrant Funktion im Signalhandler

Nachfolgend ist eine mögliche Implementierung des Programms `nonreent.c` gegeben.

```
#include <pwd.h>
#include <signal.h>
#include "eighdr.h"

static void alm_sighandler(int signr);

int
main(void)
{
    struct passwd *zgr;

    if (signal(SIGALRM, alm_sighandler) == SIG_ERR)
        fehler_meld(FATAL_SYS, "kann alm_sighandler nicht installieren");
    alarm(1);
}
```

```

while (1) {
    if ( (zgr = getpwnam("hh")) == NULL)
        fehler_meld(FATAL_SYS, "getpwnam-Fehler");
    if (strcmp(zgr->pw_name, "hh") != 0)
        printf("Rueckgabewert falsch! pw_name = %s\n", zgr->pw_name);
    }
}

static void alarm_sighandler(int signr)
{
    struct passwd *rootzgr;

    printf(".... In Signalhandler .....\n");
    if ( (rootzgr = getpwnam("root")) == NULL)
        fehler_meld(FATAL_SYS, "Fehler bei getpwnam(root)");
    alarm(1);
}

```

*Programm B.6 **nonreent.c**: Aufruf einer nicht-reentrant Funktion im Signalhandler*

Nachdem man dieses Programm B.13.2 (`nonreent.c`) kompiliert und gelinkt hat.

```
cc -o nonreent nonreent.c signal.c fehler.c
```

könnte sich z.B. der folgende Ablauf ergeben.

```

$ nonreent
.... In Signalhandler ....
Rueckgabewert falsch! pw_name = root
.... In Signalhandler ....
.... In Signalhandler ....
.... In Signalhandler ....
Rueckgabewert falsch! pw_name = root
.... In Signalhandler ....
Segmentation fault (core dumped)
$

```

Das Ablaufgeschehen des Programms `nonreent.c` hängt vom Zufall ab. Normalerweise wird dieses Programm bei der Rückkehr aus dem Signalhandler durch das Signal SIGSEGV beendet. Der Grund dafür ist, daß die `main`-Funktion bei einem **getpwnam**-Aufruf durch das Signal SIGALRM unterbrochen wurde. Da der dadurch aufgerufene Signalhandler nun seinerseits **getpwnam** aufruft, führt dies dazu, daß gewisse interne Zeiger nun verändert werden und damit bei der Fortsetzung dieser Funktion (in `main`) nach der Rückkehr aus dem Signalhandler keine gültigen Adressen mehr für **getpwnam** vorliegen.

B.9 Ausgewählte Lösungen zu Kapitel 14 (STREAMS in System V)

B.9.1 Anzahl der verschiedenen Arten von Informationen bei `getmsg`

Bis zu fünf verschiedene Arten von Information kann `getmsg` zurückliefern: die Daten, die Länge der Daten, die Kontrollinformation, die Länge der Kontrollinformation und die Flags.

B.10 Ausgewählte Lösungen zu Kapitel 15 (Fortgeschrittene Ein- und Ausgabe)

B.10.1 Gegenüberstellung der Signalmengen- und Deskriptormengenfunktionen

Die folgende Tabelle gruppiert die Funktionen, die ähnliches leisten.

FD_ZERO	sigemptyset
FD_SET	sigaddset
FD_CLR	sigdelset
FD_ISSET	sigismember
-	sigfillset

Der Unterschied zwischen diesen beiden Funktionsgruppen ist die Reihenfolge ihrer Argumente. Bei den Signalmengenfunktionen wird die Adresse der Signalmenge immer als erstes Argument und die Signalnummer als zweites Argument angegeben. Bei den Deskriptormengenfunktionen dagegen ist die Nummer das erste Argument und die Adresse der Menge das nächste Argument

B.10.2 Ändern der Limits für Deskriptormengen

In SVR4 und BSD-Unix definiert die Konstante `FD_SETSIZE` die maximale Anzahl von File-deskriptoren für den Datentyp `fd_set`. Um diese z.B. auf 3000 festzulegen, könnte der folgende Code angegeben werden.

```
#define FD_SETSIZE 3000
#include <sys/types.h>
```

B.11 Ausgewählte Lösungen zu Kapitel 16 (Dämonprozesse)

B.11.1 Schließen der Filedesktoren 0, 1 und 2 durch einen Dämonprozeß

Der Ablauf von Programm *daemclo* ist von der jeweiligen Implementierung abhängig. Das Schließen der drei ersten Filedesktoren bewirkt, daß das vor dem Aufruf von *daemonisierung* zugeordnete Kontrollterminal geschlossen wird, so daß *getlogin* kein Kontrollterminal hat und somit nicht in der Datei *utmp* seinen Logineintrag nachschlagen kann.

Unter 4.4BSD wird allerdings der Loginname in der Prozeßtafel gehalten und bei einem *fork* an den Kindprozeß vererbt. Dies bedeutet, daß dort ein Prozeß zu jeder Zeit seinen Loginnamen erfragen kann, außer der Elternprozeß (wie *init*) hatte kein Kontrollterminal.

B.12 Ausgewählte Lösungen zu Kapitel 17 (Pipes und FIFOs)

B.12.1 Starten eines Koprozesses ohne Signalhandler

Nachdem der Elternprozeß (*romkomm*) sich beendet hat, muß man sich dessen Beendigungsstatus ausgeben lassen, z.B. mit

```
echo $?      (in Bourne- oder Korn-Shell)
```

Die dabei ausgegebene Nummer ist 128 plus die Signalnummer für das Signal *SIGPIPE*.

B.12.2 Lesen und Schreiben in einer Pipe mit Standard-E/A-Funktionen

Zuerst müßte die folgende Deklaration in der *main*-Funktion hinzugefügt werden.

```
FILE *pipe_lesedz,  
      *pipe_schreibdz;
```

Als nächstes müßte dann vor der *while*-Schleife mittels *fdopen* den beiden Pipe-File-desktoren ein Dateizeiger (*FILE **) zugeteilt werden. Danach müßte für diese Dateizeiger noch Zeilenpufferung eingestellt werden. Der entsprechende Code ist nachfolgend angegeben.

```
if ( (pipe_lesedz = fdopen(pipe2[0], "r")) == NULL)  
    fehler_meld(FATAL_SYS, "fdopen-Fehler");  
if ( (pipe_schreibdz = fdopen(pipe1[1], "w")) == NULL)
```

```
fehler_meld(FATAL_SYS, "fdopen-Fehler");
if (setvpuf(pipe_lesedz, NULL, _IOLBF, 0) < 0)
    fehler_meld(FATAL_SYS, "setvbuf-Fehler");
if (setvpuf(pipe_schreibdz, NULL, _IOLBF, 0) < 0)
    fehler_meld(FATAL_SYS, "setvbuf-Fehler");
```

Die **write**- und **read**-Anweisungen in der **while**-Schleife müßten dann noch durch folgenden Code ersetzt werden:

```
if (fputs(zeile, pipe_schreibdz) == EOF)
    fehler_meld(FATAL_SYS, "Fehler beim Schreiben in Pipe1");
if (fgets(zeile, MAX_ZEICHEN, pipe_lesedz) == NULL) {
    fehler_meld(WARNUNG, "Kind hat Pipe geschlossen");
    break;
}
```

Die folgende if-Anweisung im Programm 17.11 (romkomm.c) ist damit überflüssig geworden und müßte dann noch entfernt werden.

```
if (n == 0) {
    fehler_meld(WARNUNG, "Kind hat Pipe geschlossen");
    break;
}
```

B.12.3 Kein Schließen der Schreibseite einer Pipe

Wenn die Schreibseite einer Pipe niemals geschlossen wird, so erhält der Leser aus der Pipe niemals ein EOF. Im Programm 17.4 (primfak.c) würde dies dazu führen, daß more weiterhin versucht, aus der Pipe zu lesen und somit für immer blockieren würde. Dieses Programm würde sich also nicht selbst beenden, sondern müßte mit einem Signal (wie z.B. SIGINT) beendet werden.

B.12.4 Gleichzeitiges Schreiben der Standardausgabe und -fehlerausgabe in Pipe

Man müßte die Standardfehlerausgabe dieses Programms in die Standardausgabe umlenken. Dies läßt sich durch die Angabe `2>&1` im `kdozeile`-Argument beim **popen**-Aufruf erreichen.

B.13 Ausgewählte Lösungen zu Kapitel 18 (Message-Queues, Semaphore und Shared Memory)

B.13.1 Unerlaubtes Lesen von Messages durch fremde Prozesse

Wenn ein fremder Prozeß eine Message aus einer Message-Queue liest, die nicht für ihn gedacht ist, so geht diese für den Server (Client-Anforderung) bzw. für den Client (Server-Antwort) gedachte Message verloren.

Um aus einer nicht für ihn eingerichteten Message-Queue zu lesen, muß ein fremder Prozeß nur deren Kennung kennen und für die Message-Queue muß Leserecht für others gewährt sein.

Literaturverzeichnis

Maurice J. Bach: The Design of the UNIX Operating System. Prentice Hall International, INC., London, 1986.

Dieses Buch beschreibt den Aufbau und die Funktionsweise von Unix System V. Es gilt als Standardwerk für die Unix-Systemimplementierung.

Michael Beck u. a.: Linux-Kernel-Programmierung. Addison-Wesley, Bonn, 4. Auflage, 1995.

Dieses Buch wendet sich an alle, die mehr über die Interna von Linux wissen möchten. Wer die Funktionsweise und die Implementierung kennenlernen oder selbst mit dem Systemkern experimentieren will, sollte dieses Buch lesen.

Helmut Herold: C-Kompaktreferenz, Addison-Wesley, Bonn, 1. Auflage, 1999.

Dieses Buch ist eine Kurzfassung zur Programmiersprache C, die für das schnelle Nachschlagen von C-Funktionen, C-Konstrukten und allgemeinen Algorithmen konzipiert wurde. Es beschreibt kurz und prägnant die einzelnen C-Konstrukte und die standardisierten Headerdateien. Zudem stellt es wesentliche Programmiertechniken, wichtige Algorithmen und nützliche Programme vor, die beim tagtäglichen Programmieren sehr hilfreich sein können.

Helmut Herold: Linux-Unix Grundlagen, Addison-Wesley, Bonn, 4. Auflage, 1999.

Dieses Buch ist eine Einführung in das Betriebssystem Unix und geht insbesondere auf das immer beliebtere und frei verfügbare System Linux ein. Es macht den Leser anhand von leicht nachvollziehbaren Beispielen mit den grundlegenden Linux/Unix-Kommandos und -Konzepten vertraut. Der Anhang gibt eine umfangreiche und alphabetisch geordnete Beschreibung aller grundlegenden Linux/Unix-Kommandos und eignet sich zum Nachschlagen.

Helmut Herold: Linux-Unix Shells, Addison-Wesley, Bonn, 4. Auflage, 1999.

Dieses Buch behandelt die fünf heute am weitest verbreiteten Unix-Shells: Bourne-Shell, Korn-Shell, C-Shell, bash und tcsh. Es beschreibt die einzelnen Shells und ihre Konstrukte ausführlich und leicht nachvollziehbar anhand von über 200 Shell-Programm-Beispielen, die online über den Verlag zu beziehen sind. Die meisten Unix-Systeme bieten standardgemäß mehrere Shells, manche Systeme wie Linux bieten standardgemäß sogar alle fünf Shells an.

Helmut Herold: Linux-Unix Profitools, Addison-Wesley, Bonn, 3. Auflage, 1999.

Dieses Buch behandelt die mächtigen Linux-Unix-Werkzeuge awk, sed, lex, yacc und make. awk eignet sich hervorragend dazu, die tagtäglich anfallenden Analysen und Manipulationen von Daten leicht und elegant durchführen zu lassen. sed ist ein nicht interaktiver Editor, der seine Editieranweisungen entweder aus einer Datei oder von der Kommandozeile liest. lex und yacc sind Tools, die ursprünglich zum Schreiben von Compilern und Interpretern entwickelt wurden, inzwischen aber in vielen anderen Bereichen der Softwareentwicklung

gewinnbringend eingesetzt werden. Beide Tools werden in diesem Buch äußerst ausführlich anhand leicht nachvollziehbarer Beispiele beschrieben, wobei in diesem Buch unter anderem ein nahezu vollständiges Frontend eines C-Compilers gegeben wird, indem ein Profiler für C-Programme realisiert wird. make schließlich ist das Tool schlechthin zur automatischen Programmgenerierung unter Linux/Unix. In diesem Buch wird make anhand praktischer Programmbeispiele detailliert vorgestellt.

Helmut Herold: Linux-Unix Kurzreferenz, Addison-Wesley, Bonn, 2. Auflage, 1999.

Dieses Buch ist eine Kurzreferenz zu allen Bänden dieser Buchreihe. Es enthält neben der Beschreibung anderer wichtiger Linux/Unix-Kommandos und -Tools (wie z.B. Shells, make, awk, sed, lex, yacc) auch eine Kurzfassung zu allen typischen Aufrufformen der hier behandelten Funktionen. Die Systemaufrufe werden hierbei ebenso wie alle ANSI-C-Funktionen nicht nur kurz vorgestellt, sondern oft wird noch ein kleiner Codeausschnitt angegeben, der zeigt, wie diese Funktionen zu verwenden sind. Dieses Buch soll neben den Manpages dem Programmierer nützliche und schnelle Informationen beim täglichen Programmieren seines Linux/Unix-Systems geben.

Fridolin Hofmann. Betriebssysteme: Grundkonzepte und Modellvorstellungen. Teubner, Stuttgart, 2. Auflage, 1991.

Dieses Buch gibt eine umfassende Beschreibung der Grundkonzepte und Modellvorstellungen von Betriebssystemen.

S.J. Leffler, M.K. McKusick, M.J. Karels und J.S. Quaterman: The Design and Implementation of the 4.3BSD Unix Operating System. Addison-Wesley Publishing, Reading, 1989.

Anders als Bach beschreibt dieses Buch nicht die Implementierung von System V, sondern die von BSD Unix. Es gilt ebenfalls als Standardwerk für die Entwicklung von eigenen Unix-Systemen.

W. Richard Stevens: Advanced Programming in the UNIX Environment. Addison-Wesley Publishing, Reading, 1992.

Dies ist das Standardwerk zum Programmieren unter Unix. Es beschreibt die gesamte Breite der Systemaufrufe von BSD4.3 über SVR4 bis zum POSIX-Standard.

W. Richard Stevens: Programmieren von UNIX-Netzen. Coedition Verlage Carl Hanser und Prentice-Hall, München und London, 1992.

Dies ist das Standardwerk zur Programmierung von Unix-Netzen.

W. Richard Stevens: TCP/IP Illustrated: The Protocols, Volume 1. Addison-Wesley Publishing, Reading, 1994.

Dieses Buch ist das Standardwerk für jeden, der sich mit TCP/IP vertraut machen möchte.

Andrew S. Tanenbaum: Modern Operating Systems. Prentice Hall International, INC., London, 1986.

Dieses Buch beschreibt grundlegende Prinzipien der Arbeitsweise von klassischen und verteilten Betriebssystemen.

Andrew S. Tanenbaum: Betriebssysteme - Entwurf und Realisierung - Teil 1 Lehrbuch. Coedition Verlage Carl Hanser und Prentice-Hall, Berlin und London, 1990.

Tanenbaum beschreibt hier den Aufbau und die Funktion seines Minix Systems. Minix (Mini Unix) wurde von Tanenbaum für Ausbildungszwecke geschrieben. Es verdeutlicht sehr anschaulich die Konzepte einer Unix-Implementierung, ist aber wegen seiner Beschränkungen nur wenig praxistauglich. Die Entwicklung von Linux begann übrigens unter Minix.

Kevin Washburn und Jim Evans: TCP/IP. Addison-Wesley, Bonn, 1994.

In diesem Buch werden die TCP/IP-Protokolle und ihre Anwendung sehr ausführlich beschrieben.

Stichwortverzeichnis

Operator 107
Operator 108
#define 106
#elif 111
#else 112
#endif 112
#error 112
#if 111
#ifdef 111
#ifndef 111
#include 109
#line 112
#pragma 112
#undef 112

!

/bin/bash Programm 11
/bin/csh Programm 10
/bin/ksh Programm 10
/bin/sh Programm 10
/bin/tcsh Programm 11
/dev/conslog STREAM 708
/dev/fd Directory 259
/dev/log STREAM 708
/dev/stderr Directory 260
/dev/stdin Directory 260
/dev/stdout Directory 260
/dev/tty 558
/etc/conf/cf.d/mtune Datei 756
/etc/group Datei 374
/etc/hosts Datei 377
/etc/ld.so.cache Datei 1091
/etc/ld.so.conf Datei 1089
/etc/ld.so.preload Datei 1095
/etc/motd 552
/etc/networks Datei 377
/etc/passwd Datei 10, 369
/etc/protocols Datei 377
/etc/services Datei 377
/etc/shadow Datei 10, 373
/etc/syslog.conf Datei 710
/etc/termcap Datei 922
/etc/ttys 549
<asm/segment.h> Headerdatei 446
<asm/uaccess.h> Headerdatei 446
<assert.h> Headerdatei 37, 124, 125
<cpio.h> Headerdatei 37
<ctype.h> Headerdatei 37, 124, 127
<curses.h> Headerdatei 923
<dirent.h> Headerdatei 37, 317
<dlfcn.h> Headerdatei 1096
<errno.h> Headerdatei 26, 37, 124, 128, 214
<fcntl.h> Headerdatei 37, 222
<float.h> Headerdatei 37, 124, 128
<ftw.h> Headerdatei 37
<getopt.h> Headerdatei 1032
<grp.h> Headerdatei 37, 374
<langinfo.h> Headerdatei 38
<limits.h> Headerdatei 38, 124, 130, 222
<linux/fs.h> Headerdatei 329
<linux/locks.h> Headerdatei 334
<linux/socket.h> Headerdatei 836
<linux/un.h> Headerdatei 839
<linux/vfs.h> Headerdatei 338
<locale.h> Headerdatei 38, 124, 131
<math.h> Headerdatei 38, 124, 136
<netbd.h> Headerdatei 378
<nl_types.h> Headerdatei 38
<popt.h> Headerdatei 1038
<pwd.h> Headerdatei 38, 369
<regex.h> Headerdatei 38
<search.h> Headerdatei 38
<setjmp.h> Headerdatei 38, 124, 403
<shadow.h> Headerdatei 374
<signal.h> Headerdatei 38, 125, 600
<slang.h> Headerdatei 936
<stdarg.h> Headerdatei 38, 121, 125, 193, 194
<stddef.h> Headerdatei 38, 125, 141
<stdio.h> Headerdatei 38, 125, 167

<stdlib.h> Headerdatei 38, 125, 142
 <string.h> Headerdatei 38, 125, 152
 <stropts.h> Headerdatei 663, 682
 <sys/acct.h> Headerdatei 541
 <sys/ioctl.h> Headerdatei 986
 <sys/ipc.h> Headerdatei 38, 755
 <sys/kd.h> Headerdatei 986
 <sys/msg.h> Headerdatei 38
 <sys/param.h> Headerdatei 686
 <sys/sem.h> Headerdatei 38
 <sys/shm.h> Headerdatei 38
 <sys/socket.h> Headerdatei 816, 836
 <sys/stat.h> Headerdatei 38, 266
 <sys/times.h> Headerdatei 38
 <sys/types.h> Headerdatei 38, 51, 225
 <sys/uio.h> Headerdatei 696
 <sys/un.h> Headerdatei 839
 <sys/utsname.h> Headerdatei 39
 <sys/vfs.h> Headerdatei 338
 <sys/vt.h> Headerdatei 986
 <sys/wait.h> Headerdatei 39, 505
 <tar.h> Headerdatei 38
 <termios.h> Headerdatei 38, 880
 <time.h> Headerdatei 38, 125, 385
 <ulimit.h> Headerdatei 38
 <unistd.h> Headerdatei 38, 222
 <utime.h> Headerdatei 38
 <utmp.h> Headerdatei 380
 <varargs.h> Headerdatei 193, 194
 __add_wait_queue Funktion 70
 __copy_from_user Funktion 447
 __copy_to_user Funktion 448
 __DATE__ Makro 113
 __FILE__ Makro 113
 __get_free_page Funktion 474
 __get_free_pages Funktion 471
 __get_user Funktion 446
 __iget Funktion 340
 __LINE__ Makro 113
 __pgprot Makro 454
 __put_user Funktion 447
 __remove_wait_queue Funktion 70
 __sleep_on Funktion 71
 __STDC__ Makro 113
 __TIME__ Makro 113
 __exit Funktion 424
 _IOBF Konstante 200
 _IOLBF Konstante 200
 _IONBF Konstante 201
 _namei Funktion 341
 _PC_CHOWN_RESTRICTED Konstante 44

_PC_LINK_MAX Konstante 44
 _PC_MAX_CANON Konstante 44
 _PC_MAX_INPUT Konstante 44
 _PC_NAME_MAX Konstante 44
 _PC_NO_TRUNC Konstante 44
 _PC_PATH_MAX Konstante 44
 _PC_PIPE_BUF Konstante 44
 _PC_VDISABLE Konstante 44
 _POSIX_ARG_MAX Konstante 40
 _POSIX_CHILD_MAX Konstante 40
 _POSIX_CHOWN_RESTRICTED
 Konstante 42, 44, 282
 _POSIX_JOB_CONTROL Konstante 42, 44,
 555
 _POSIX_LINK_MAX Konstante 40
 _POSIX_MAX_CANON Konstante 40
 _POSIX_MAX_INPUT Konstante 40
 _POSIX_NAME_MAX Konstante 40
 _POSIX_NGROUPS_MAX Konstante 40
 _POSIX_NO_TRUNC Konstante 42, 44, 225
 _POSIX_OPEN_MAX Konstante 40
 _POSIX_PATH_MAX Konstante 40
 _POSIX_PIPE_BUF Konstante 40
 _POSIX_SAVED_IDS Konstante 42, 44, 271
 _POSIX_SSIZE_MAX Konstante 40
 _POSIX_STREAM_MAX Konstante 40
 _POSIX_TZNAME_MAX Konstante 40
 _POSIX_VDISABLE Konstante 42, 44
 _POSIX_VERSION Konstante 42, 44
 _SC_ARG_MAX Konstante 43
 _SC_CHILD_MAX Konstante 43
 _SC_CLK_TCK Konstante 43
 _SC_JOB_CONTROL Konstante 44
 _SC_NGROUPS_MAX Konstante 43
 _SC_OPEN_MAX Konstante 43
 _SC_PASS_MAX Konstante 43
 _SC_SAVED_IDS Konstante 44, 271
 _SC_STREAM_MAX Konstante 43
 _SC_TZNAME_MAX Konstante 44
 _SC_VERSION Konstante 44
 _SC_XOPEN_VERSION Konstante 44

A

Abhängigkeitsbeschreibung (make) 1102
 abort Funktion 143, 648
 abs Funktion 145
 absoluter Pfadname 14
 accept Funktion 837
 access Funktion 276, 299

access time 307
access_ok Funktion 446
acct Funktion 541
acct Struktur 541
accton Kommando 541
acos Funktion 136
add_wait_queue Funktion 70
addch Funktion 924
addstr Funktion 924
Adreßraum
 Linear 448
 Virtuell 457
advisory locking 579
ändern der Dateieinstellungen 248
aktualisieren
 Bildschirm 924
alarm Funktion 630
alloca Funktion 439
ANSI C 101
ANSI-C-Bibliothek 124
ar Kommando 1082
ARG_MAX Konstante 41, 43
Argument 102
Array
 dynamisch 437
Arten
 Datei- 11, 265
asctime Funktion 391
asin Funktion 136
assert Funktion 126
Asynchrone E/A 673, 681
atan Funktion 136
atan2 Funktion 136
atexit Funktion 143, 424
atof Funktion 145
atoi Funktion 145
atol Funktion 145
atomare Operation 243
Attribute festlegen 926
Attribute von Dateien 264
attroff Funktion 927
attron Funktion 927
attrset Funktion 926
Aufrufsyntax
 make 1106
autofahr.c 998
automatic Variable 412

B

bash Programm 11
Baudrate 908
bedingte Kompilierung 111
Beenden eines Programms 423, 424
Beendigung von Prozessen 502
benannte
 Pipe 744
 Stream Pipe 828
Benutzerinformationen 369
Benutzerklasse 268
Bibliothek
 dynamisch 1087
 statisch 1082
Bibliotheksfunktionen 33
Big-Endian 856
Bildschirm aktualisieren 924
Bildschirm löschen 923
Bildschirmausschnitt kopieren 932
bind Funktion 836
Blockierung 567
bmap Funktion 345
Boot-Block 287
Booten 72
Bootmanager 289
Borland-Semigraphik 968
Bottom-Half Routinen 80
Bourne-Again-Shell 11
Bourne-Shell 10
BSD 36
BSD-Unix 7
bsearch Funktion 148
bss segment 432
buchmemo.c 1001
Buchstaben-Memory 1001
BUFSIZ Konstante 202

C

Cache 327
caddr_t Datentyp 52, 684
calloc Funktion 142, 433
cat Kommando 232
cbreak Funktion 929
cc_t Datentyp 880
ceil Funktion 136
cfgetispeed Funktion 909
cfgetospeed Funktion 909
cfsetispeed Funktion 909

cfsetospeed Funktion 909
 CHAR_BIT Konstante 130
 CHAR_MAX Konstante 130
 CHAR_MIN Konstante 130
 chattr Kommando 364
 chdir Funktion 299, 314
 check_media_change Funktion 349
 checkgcc Programm 1078
 child process 486
 CHILD_MAX Konstante 41, 43, 490
 chmod Funktion 273, 299
 chmod Kommando 268
 chown Funktion 281, 299
 chroot Funktion 367
 chvt Kommando 993
 cleanup (make) 1109
 clear Funktion 923
 clearenv Funktion 431
 clearerr Funktion 173
 cli_verbind Funktion 830
 CLK_TCK Konstante 42
 clock Funktion 398
 clock_t Datentyp 32, 52, 385
 CLOCKS_PER_SEC Konstante 32, 386, 398
 CLOCKS_PER_SEC Makro 538
 clone Funktion 501
 close Funktion 228
 closedir Funktion 317
 closelog Funktion 711
 close-on-exec-Bit 249
 clrtobot Funktion 933
 clrtoeol Funktion 933
 CMSG_DATA Makro 820
 cmsghdr Struktur 820
 COLS Variable 923
 conio.h Headerdatei 971
 connect Funktion 838
 connld Steuermodul 830
 const Schlüsselwort 117
 copy_from_user Funktion 447
 copy_from_user_ret Funktion 447
 copy_to_user Funktion 448
 copy_to_user_ret Funktion 448
 copy-on-write 488
 copywin Funktion 932
 cos Funktion 136
 cosh Funktion 136
 COW-Verfahren 488
 cpio Kommando 311
 cpu_idle Funktion 75
 CPU-Zeit 32

CPU-Zeit erfragen 398
 creat Funktion 226, 299
 create Funktion 342
 creation time 307
 cron Dämon 704
 crontab Kommando 704
 crypt Kommando 373
 csh Programm 10
 C-Shell 10
 CSI-Sequenzen 959
 ctermid Funktion 890
 ctime Funktion 391
 CTRL Makro 956
 curses 921
 curses-Modus ausschalten 923
 curses-Modus einschalten 923
 Cursor positionieren 924
 Cursor-Steuertasten 929

D

Dämon 703
 cron 704
 inetd 552, 704
 lpd 704
 lpsched 749
 sendmail 703
 syslogd 703
 telnetd 553
 update 704
 Dämonprozesse 575
 data segment 432
 Datagram-Protokolle 834
 Datei
 /etc/conf/cf.d/mtune 756
 /etc/group 374
 /etc/hosts 377
 /etc/networks 377
 /etc/passwd 10, 369
 /etc/protocols 377
 /etc/services 377
 /etc/shadow 10, 373
 /etc/syslog.conf 710
 /etc/termcap 922
 Abschneiden 305
 access time 307
 Änderungszeit 307
 Arten 31, 229
 Attribute 264
 creation time 307

- Eigentümer 272
- Eigentümer ändern 281
- einfache 11, 265
- Einstellungen ändern 248
- Einstellungen erfragen 248
- EOF-Flag 173
- Fehler-Flag 173
- Geräte- 265, 12
- Größe 303
- Kreieren 226
- Kreierungsmaske 278
- Lesen (blockweise, binär) 194
- Lesen (bytewise) 229
- Lesen (ein Zeichen) 173, 175, 177
- Lesen (einer Zeile) 179
- Lesen (formatiert) 180
- Löcher 306
- Löschen 212
- löschen 296
- Log- 703
- mit Stream verknüpfen 170
- modification time 307
- Name 13
- öffnen 168, 222, 226
- positionieren 204, 206, 207, 234
- Pufferung 201
- schließen 172, 228
- schreiben (blockweise, binär) 194
- schreiben (bytewise) 231
- schreiben (ein Zeichen) 173, 175
- schreiben (einer Zeile) 179
- schreiben (formatiert mit
 Argumentzeiger) 193
- schreiben (formatiert) 185
- sperrern 567, 568
- Temporäre 207
- umbenennen 213
- utmp 380
- wtmp 380
- Zeit 307
- Zeit der i-node-Änderung 307
- Zugriffszeit 307
- Dateiarten 11, 265
- Dateigröße 11
- Dateinamenexpansion 1007, 1013
- Dateistruktur 11
- Dateisystem 13, 283, 329
- Dateitabelle 240
- Dateitabellen (Kern) 240
- Datentyp
 - caddr_t 52, 684
 - cc_t 880
 - clock_t 32, 52, 385
 - dev_t 52, 325
 - div_t 142
 - fd_set 52, 674
 - FILE 253
 - fpos_t 52
 - gid_t 52
 - ino_t 52
 - jmp_buf 405
 - key_t 754
 - ldiv_t 142
 - mode_t 52, 225
 - nlink_t 52
 - off_t 52, 235
 - pid_t 52
 - ptrdiff_t 52, 141
 - rlim_t 52
 - sig_atomic_t 52
 - sigatomic_t 641
 - sigjmp_buf 641
 - sigset_t 52, 618
 - size_t 52, 141, 230, 385
 - ssize_t 41, 52, 230
 - tcflag_t 880
 - time_t 32, 52, 385
 - uid_t 52
 - va_list 121
 - void 116
 - wchar_t 52, 105, 141
- Datentypen 114
- Datumsangaben 385
- DBL_DIG Konstante 129
- DBL_MANT_DIG Konstante 128
- ddd 1062
- deallocvt Kommando 994
- Debugger 1061
- delay() 998
- delayed write 327
- deleteIn Funktion 932
- dev_t Datentyp 52, 325
- Device Number 325
- difftime Funktion 396
- DIR Struktur 318
- dir_namei Funktion 341
- Directory 12, 13, 265, 311
 - anlegen 313
 - Hierarchie durchlaufen 318
 - Home- 14
 - lesen 317
 - löschen 212, 314

- Parent- 14
- Root- 13
- umbenennen 213
- wechseln 314
- Working- 13, 315
- Zugriffsrechte 312
- Directorycache 351
- dirent Struktur 317
- div Funktion 147
- div_t Datentyp 142
- dlclose Funktion 1096
- dllerror Funktion 1096
- dlopen Funktion 1096
- dlsym Funktion 1096
- DNS 863
- do_it_prof Funktion 83
- do_it_virt Funktion 83
- do_mmap Funktion 461
- do_no_page Funktion 476
- do_page_fault Funktion 474
- do_process_times Funktion 82
- do_swap_page Funktion 477
- do_timer Funktion 80
- do_wp_page Funktion 476
- Domain Name System 863
- dos.h 997
- down Funktion 72
- du Kommando 304
- dup Funktion 245
- dup2 Funktion 245
- Duplizieren eines Filedeskriptors 245, 248, 259
- dynamische
 - Arrays 437
 - Bibliotheken 1087
- dynamischer
 - Speicher 433
 - Speicher (Stack) 439

E

- E/A-Funktionen 18, 20, 167, 221
- E/A-Multiplexing 671
- echo Funktion 928
- EDOM Konstante 128, 137
- efence Bibliothek 1074
- effektive GID 269, 281
- effektive UID 269, 281
- Eigentümer einer Datei 272, 281
- eighdr.h Headerdatei 1123

- Eingabezeichen 896
- Electric Fence 1074
- Elementare E/A-Funktionen 20, 221
- ELF Binärformat 1088
- Ellipsen-Prototypen 120
- ELOOP Fehler 301
- Elternprozeß 483, 486
- empfang_fd Funktion 813
- ENAMETOOLONG Konstante 225
- endgrent Funktion 375
- endpwent Funktion 371
- endservent Funktion 868
- endwin Funktion 923
- enum 115
- environ Variable 427
- Environment 427, 479
- EOF-Flag 173
- EPIPE Fehler 235
- ERANGE Konstante 128, 137
- erase Funktion 923
- errno Variable 26, 128, 214
- Erweiterte Partition 288
- Escapesequenzen 957
- exec Funktion 299
- execl Funktion 521
- execle Funktion 521
- execlp Funktion 521
- execv Funktion 521
- execve Funktion 521
- execvp Funktion 521
- exit Funktion 143, 423
- EXIT_FAILURE Konstante 142
- EXIT_SUCCESS Konstante 142
- Exit-Handler 424
- Exit-Status 421
- exp Funktion 136
- Expandierung
 - Dateinamen 1007, 1013
- export Kommando 477
- ext2_new_inode Funktion 342
- ext2_read_super Funktion 329
- ext2-Filesystem 354

F

- F_DUPFD Konstante 248
- F_FREESP Konstante 305
- F_GETFD Konstante 248, 249
- F_GETFL Konstante 248, 249
- F_GETLK Konstante 248, 569, 570

- F_GETOWN Konstante 248, 249
- F_OK Konstante 277
- F_SETFD Konstante 248, 249
- F_SETFL Konstante 248, 249
- F_SETLK Konstante 248, 569, 570
- F_SETLKW Konstante 248, 569, 570
- F_SETOWN Konstante 248, 249
- fabs Funktion 136
- fasync Funktion 349
- fattach Funktion 831
- fchdir Funktion 314
- fchmod Funktion 273
- fchown Funktion 281
- fclose Funktion 172
- fcntl Funktion 248, 568
- FD_CLOEXEC Konstante 249
- FD_CLR Makro 674
- FD_ISSET Makro 674
- fd_set Datentyp 52, 674
- FD_SET Makro 674
- FD_ZERO Makro 674
- fdetach Funktion 831
- fdopen Funktion 254
- fehler.c 1124
- fehler_meld (eigene Fehlerroutine) 16, 1124
- Fehler-Flag 173
- Fehlermeldung 26, 214
- Fehlerroutine 16, 1124
- Fenstergröße 919
- feof Funktion 173
- ferror Funktion 173
- fflush Funktion 203
- fg Kommando 561
- fgetc Funktion 175
- fgetpos Funktion 206
- fgets Funktion 179
- FIFO 12, 265, 744
- File
 - Operationen (Linux intern) 346
- FILE Datentyp 167, 253
- file sharing 241
- file Struktur (Linux intern) 346
- file transfer walk 318
- file_operations Struktur 346
- file_system_type Struktur 329
- Filedeskriptor 17, 221, 253
 - duplizieren 245, 248, 259
- fileno Funktion 254
- Filesystem 283, 329
- filesystems Zeiger 329
- Filterprogramm 734
- finger Kommando 370
- flock Funktion 571
- flock Struktur 569
- floor Funktion 136
- FLT_DIG Konstante 128
- FLT_MANT_DIG Konstante 128
- FLT_MAX_EXP Konstante 129
- FLT_MIN_EXP Konstante 129
- FLT_RADIX Konstante 128
- FLT_ROUNDS Konstante 129
- FM_NAMESZ Konstante 665
- fmod Funktion 136
- fnmatch Funktion 1013
- follow_link Funktion 341, 345
- fopen Funktion 168
- fork Funktion 486
- Fortsetzungszeichen (make) 1105
- fpathconf Funktion 43
- fpos_t Datentyp 52, 204
- fprintf Funktion 185
- fputc Funktion 175
- fputs Funktion 179
- fread Funktion 194
- free Funktion 142, 438
- free_area Tabelle 473
- free_area_struct Struktur 472
- free_page Funktion 474
- free_pages Funktion 474
- Freigabe von Speicher 438
- freopen Funktion 170
- frexp Funktion 137
- fscanf Funktion 180
- fseek Funktion 204
- fsetpos Funktion 206
- fstat Funktion 264
- fstatfs Funktion 338
- fsync Funktion 328, 348
- ftell Funktion 204
- ftruncate Funktion 305
- ftw Funktion 301, 318
- Funktion
 - __add_wait_queue 70
 - __copy_from_user 447
 - __copy_to_user 448
 - __get_free_page 474
 - __get_free_pages 471
 - __get_user 446
 - __iget 340
 - __pgprot 454
 - __put_user 447
 - __remove_wait_queue 70

- __sleep_on 71
- _exit 424
- _namei 341
- abort 143, 648
- abs 145
- accept 837
- access 276, 299
- access_ok 446
- acct 541
- acos 136
- add_wait_queue 70
- addch 924
- addstr 924
- alarm 630
- alloca 439
- asctime 391
- asin 136
- assert 126
- atan 136
- atan2 136
- atexit 143, 424
- atof 145
- atoi 145
- atol 145
- attroff 927
- attron 927
- attrset 926
- bind 836
- bmap 345
- bsearch 148
- calloc 142, 433
- cbreak 929
- ceil 136
- cfgetispeed 909
- cfgetospeed 909
- cfsetispeed 909
- cfsetospeed 909
- chdir 299, 314
- check_media_change 349
- chmod 273, 299
- chown 281, 299
- chroot 367
- clear 923
- clearenv 431
- clearerr 173
- cli_verbind 830
- clock 398
- clone 501
- close 228
- closedir 317
- closelog 711
- clrtobot 933
- clrtoeol 933
- connect 838
- copy_from_user 447
- copy_from_user_ret 447
- copy_to_user 448
- copy_to_user_ret 448
- copywin 932
- cos 136
- cosh 136
- cpu_idle 75
- creat 226, 299
- create 342
- ctermid 890
- ctime 391
- delay 998
- deleteln 932
- difftime 396
- dir_namei 341
- div 147
- dlclose 1096
- dllerror 1096
- dlopen 1096
- dlsym 1096
- do_it_prof 83
- do_it_virt 83
- do_mmap 461
- do_no_page 476
- do_page_fault 474
- do_process_times 82
- do_swap_page 477
- do_timer 80
- do_wp_page 476
- down 72
- dup 245
- dup2 245
- echo 928
- Ellipsen-Prototypen 120
- empfang_fd 813
- endgrent 375
- endpwent 371
- endservent 868
- endwin() 923
- erase 923
- exec 299
- execl 521
- execle 521
- execlp 521
- execv 521
- execve 521
- execvp 521

- exit 143, 423
- exp 136
- ext2_new_inode 342
- ext2_read_super 329
- fabs 136
- fasync 349
- fattach 831
- fchdir 314
- fchmod 273
- fchown 281
- fclose 172
- fcntl 248, 568
- fdetach 831
- fdopen 254
- feof 173
- ferror 173
- fflush 203
- fgetc 175
- fgetpos 206
- fgets 179
- fileno 254
- flock 571
- floor 136
- fmod 136
- fnmatch 1013
- follow_link 341, 345
- fopen 168
- fork 486
- fpathconf 43
- fprintf 185
- fputc 175
- fputs 179
- fread 194
- free 142, 438
- free_page 474
- free_pages 474
- freopen 170
- frexp 137
- fscanf 180
- fseek 204
- fsetpos 206
- fstat 264
- fstatfs 338
- fsync 328, 348
- ftell 204
- ftruncate 305
- ftw 301, 318
- fwrite 194
- get_ds 448
- get_empty_inode 341
- get_free_page 474
- get_fs 448
- get_user 446
- get_user_ret 446
- getc 175
- getch 929
- getchar 173
- getcwd 315
- getegid 484
- getenv 143, 430, 479
- geteuid 484
- getgid 29, 484
- getgrent 375
- getgrgid 374
- getgrnam 374
- getgroups 376
- gethostbyaddr 378, 864
- gethostbyname 378, 864
- gethostname 379
- getitimer 634
- getlogin 541
- getmsg 661
- getnetbyaddr 378
- getnetbyname 378
- getopt 1026
- getopt_long 1031
- getopt_long_only 1031
- getpagesize 686
- getpass 895
- getpgid 555
- getpgrp 554
- getpid 483
- getpmsg 661
- getppid 483
- getprotobyname 378
- getprotobynumber 378
- getpwent 371
- getpwnam 371
- getpwuid 371
- getrlimit 439
- getrusage 443, 538
- gets 179
- getservbyname 378, 867
- getservbyport 378, 867
- getservent 868
- getsockopt 873
- gettimeofday 387
- getuid 29, 484
- glob 1009
- globfree 1009
- gmtime 389
- goodness 90

- h_error 865
- HALLO_KIND 517, 645, 729
- HALLO_PAPA 517, 645, 729
- htonl 857
- htons 857
- iget 340
- inet_addr 859
- inet_aton 859
- inet_lnaof 860
- inet_makeaddr 861
- inet_netof 860
- inet_network 860
- inet_ntoa 859
- init 75
- INIT_SYNCH 517, 645, 729
- initgroups 376
- initscr 923
- insertln 932
- interruptible_sleep_on 71
- ioctl 348, 663, 986
- iput 341
- isalnum 127
- isalpha 127
- isascii 128
- isastream 664
- isatty 890
- iscntrl 127
- isdigit 127
- isgraph 127
- islower 127
- isprint 127
- ispunct 127
- isspace 127
- isupper 127
- isxdigit 127
- kfree 462
- kmalloc 462
- labs 145
- lchown 281, 299
- ldexp 137
- ldiv 147
- link 295, 299, 343
- listen 837
- lnamei 341
- localeconv 134
- localtime 389
- lock_super 334
- lockf 571
- log 137
- log10 137
- longjmp 404
- lookup 343
- lseek 234, 347
- lstat 264, 299
- main 420
- malloc 142, 433
- mblen 152
- mbstowcs 152
- mbtowc 152
- mcheck 1080
- memchr 153
- memcmp 153
- memcpy 153
- memcpy_fromfs 447
- memcpy_tofs 447
- memmove 153
- memset 154
- mk_pte 454
- mkdir 299, 313, 343
- mkfifo 299, 744
- mknod 299, 344
- mktime 389
- mlock 691
- mlockall 691
- mmap 348, 441, 683
- modf 137
- mount 330, 331
- mount_root 330
- move 924
- move_last_runqueue 87
- msgctl 762
- msgget 758
- msgrcv 761
- msgsnd 759
- msync 689
- munlock 691
- munlockall 691
- munmap 689
- mvaddch 924
- mvaddstr 924
- mvprintw 924
- namei 341
- nanosleep 636
- nftw 301, 318
- nocbreak 929
- noecho() 928
- nosound 998
- notify_change 336
- ntohl 857
- ntohs 857
- open 222, 299, 349
- open_namei 337

- opendir 299, 317
- openlog 711
- pathconf 43, 299
- pause 634
- pclose 731
- permission 346
- perror 27, 214
- pgd_alloc 451
- pgd_bad 451
- pgd_clear 451
- pgd_free 451
- pgd_none 451
- pgd_offset 451
- pgd_present 451
- pgd_val 450
- pgprot_val 454
- pipe 718
- pmd_alloc 451
- pmd_alloc_kernel 452
- pmd_bad 452
- pmd_clear 452
- pmd_free 452
- pmd_free_kernel 452
- pmd_none 452
- pmd_offset 452
- pmd_page 452
- pmd_present 452
- pmd_val 451
- poll 678
- popen 731, 1007
- poptAddAlias 1045
- poptBadOption 1044
- poptFreeContext 1040
- poptGetArg 1042
- poptGetArgs 1042
- poptGetContext 1040
- poptGetNextOpt 1041
- poptGetOptArg 1042
- poptParseArgvString 1046
- poptPeekArg 1042
- poptPrintHelp 1043
- poptPrintUsage 1043
- poptReadConfigFile 1045
- poptReadDefaultConfig 1045
- poptResetContext 1040
- poptStrerror 1044
- poptStuffArgs 1046
- pow 137
- printf 185
- printw 924
- Prototyping 119
- psignal 614
- pte_alloc 454
- pte_alloc_kernel 454
- pte_clear 454
- pte_dirty 455
- pte_exec 455
- pte_exprotect 455
- pte_free 455
- pte_free_kernel 455
- pte_mkclean 455
- pte_mkdirty 455
- pte_mkexec 455
- pte_mkold 455
- pte_mkread 456
- pte_mkwrite 456
- pte_mkyoung 456
- pte_modify 456
- pte_none 456
- pte_offset 456
- pte_page 456
- pte_present 456
- pte_rdp protect 456
- pte_read 457
- pte_val 452
- pte_write 457
- pte_wrprotect 457
- pte_young 457
- put_inode 337
- put_super 337
- put_user 447
- put_user_ret 447
- putc 175
- putchar 173
- putenv 430, 479
- putmsg 659
- putpmsg 659
- puts 179
- qsort 150
- rand 143
- read 229, 347
- read_inode 335
- read_super 333
- readdir 317, 347
- readlink 299, 302, 344
- readv 695
- realloc 142, 433
- reentrant- 627
- refresh 924
- regcomp 1016
- regerror 1019
- regex 1018

- regfree 1019
- register_filesystem 329
- release 348
- remount_fs 338
- remove 212, 299
- remove_wait_queue 70
- rename 213, 299, 344
- revalidate 349
- rewind 207
- rewinddir 317
- rmdir 299, 314, 344
- run_old_timers 84
- run_timer_list 84
- sbrk 435
- scanf 180
- scanw 929
- sched_scheduler 86
- schedule 87
- select 347, 635, 673
- semctl 774
- semget 773
- semop 776
- send_fd 813
- send_fehl 813
- serv_bereit 829
- serv_initverbind 829
- set_fs 448
- SET_PAGE_DIR 451
- set_pte 457
- setbuf 201
- setegid 535
- setenv 430, 479
- seteuid 535
- setfsgid 65, 536
- setfsuid 65, 536
- setgid 532
- setgrent 375
- setgroups 376
- sethostname 380
- setitimer 634
- setjmp 404
- setlocale 132
- setpgid 555
- setpwent 371
- setregid 535
- setreuid 535
- setrlimit 439
- setscheduler 86
- setservent 868
- setsid 556
- setsockopt 873
- setuid 532
- setup 330
- setup_arch 75
- setvbuf 201
- shm_swap 473
- shmat 784
- shmdt 786
- shmget 782
- shrink_mmap 473
- sigaction 619
- sigaddset 618
- sigdelset 618
- sigemptyset 618
- sigfillset 618
- sigismember 618
- siglongjmp 639
- signal 30, 600
- sigpending 625
- sigprocmask 623
- sigsetjmp 639
- sigsuspend 642
- sin 137
- sinh 137
- sleep 635
- sleep_on 71
- smap 346
- socket 835
- socketpair 841
- sound 998
- sprintf 192
- sqrt 137
- srand 143
- sscanf 192
- standend 927
- standout 927
- start_kernel 73
- stat 264, 299
- statfs 338
- stime 387
- strcat 154
- strchr 154
- strcmp 155
- strcoll 155
- strcpy 155
- strcspn 155
- stream_pipe 807
- strerror 27, 155, 215
- strftime 393
- strlen 155
- strncat 155
- strncmp 156

- strncpy 156
 - strpbrk 156
 - strptime 394
 - strrchr 157
 - strspn 158
 - strstr 158
 - strtod 146
 - strtok 158
 - strtol 146
 - strtoul 146
 - strxfrm 160
 - swap_out 473
 - swapoff 468
 - swapon 465
 - symlink 301, 343
 - sync 328
 - sys_chmod 337
 - sys_chown 337
 - sys_fchmod 337
 - sys_fchown 337
 - sys_fstatfs 338
 - sys_ftruncate 337
 - sys_mount 331
 - sys_setup 330
 - sys_statfs 338
 - sys_truncate 337
 - sys_umount 332
 - sys_utime 337
 - sys_write 337
 - sysconf 43, 441
 - syslog 709, 711
 - system 143, 527
 - tan 137
 - tcdrain 911
 - tcflow 911
 - tcflush 911
 - tcgetattr 887
 - tcgetpgrp 558
 - tcsendbreak 911
 - tcsetattr 887
 - tcsetpgrp 558
 - tempnam 209
 - time 387
 - timer_bh 81
 - times 537
 - timespec 636
 - timeval 636
 - tmpfile 209
 - tmpnam 208
 - tolower 127
 - toupper 127
 - truncate 299, 305, 345
 - try_to_free_page 473
 - ttyname 892
 - umask 278
 - umount 332
 - uname 378
 - ungetc 177
 - unlink 296, 299, 343
 - unlock_super 334
 - unsetenv 430, 479
 - up 72
 - update_one_process 82
 - update_process_times 82
 - update_times 81
 - update_wall_times 82
 - usleep 635
 - utime 308
 - utimes 309
 - vfork 498
 - vfprintf 193
 - vfree 463
 - vmalloc 463
 - vprintf 193
 - vsprintf 194
 - wait 504
 - wait3 515
 - wait4 515
 - waitpid 504
 - wake_up 72
 - wake_up_interruptible 72
 - wake_up_process 72
 - WAKE_AUF_KIND 517, 645, 729
 - WAKE_AUF_PAPA 517, 645, 729
 - wcstombs 152
 - wctomb 152
 - write 231, 347
 - write_inode 337
 - write_super 338
 - writev 695
 - xchg 81
 - Funktionstasten 929
 - fwrite Funktion 194
- G**
- Ganzzahltypen 114
 - gatter.c 1003
 - gcc Compiler 1055
 - gdb Debugger 1061
 - Geräte 325

Gerätedatei 265, 325
 Gerätenummer 325
 Gerätedatei 12
 get_ds Funktion 448
 get_empty_inode Funktion 341
 get_free_page Funktion 474
 get_fs Funktion 448
 get_user Funktion 446
 get_user_ret Funktion 446
 getc Makro/Funktion 175
 getch Funktion 929
 getchar Makro/Funktion 173
 getcwd Funktion 315
 getegid Funktion 484
 getenv Funktion 143, 430, 479
 geteuid Funktion 484
 getgid Funktion 29, 484
 getgrent Funktion 375
 getgrgid Funktion 374
 getgrnam Funktion 374
 getgroups Funktion 376
 gethostbyaddr Funktion 378, 864
 gethostbyname Funktion 378, 864
 gethostname Funktion 379
 getitimer Funktion 634
 getlogin Funktion 541
 getmsg Funktion 661
 getnetbyaddr Funktion 378
 getnetbyname Funktion 378
 getopt Funktion 1026
 getopt_long Funktion 1031
 getopt_long_only Funktion 1031
 getpagesize Funktion 686
 getpass Funktion 895
 getpgid Funktion 555
 getpgrp Funktion 554
 getpid Funktion 483
 getpmsg Funktion 661
 getppid Funktion 483
 getprotobyname Funktion 378
 getprotobynumber Funktion 378
 getpwent Funktion 371
 getpwnam Funktion 371
 getpwuid Funktion 371
 getrlimit Funktion 439
 getrusage Funktion 443, 538
 gets Funktion 179
 getservbyname Funktion 378, 867
 getservbyport Funktion 378, 867
 getservent Funktion 868
 getsockopt Funktion 873

gettimeofday Funktion 387
 getty Programm 549
 gettytab Datei 550
 getuid Funktion 29, 484
 GID 269, 281
 gid_t Datentyp 52
 Gleitpunkttypen 114
 glob Funktion 1009
 globfree Funktion 1009
 gmtime Funktion 389
 goodness Funktion 90
 group Struktur 374
 Group-ID 29, 269, 281
 Grunddatentypen 114
 Gruppendatei 374

H

h_error Funktion 865
 HALLO_KIND Funktion 517, 645, 729
 HALLO_PAPA Funktion 517, 645, 729
 Hard-Link 292, 295
 Hardware-Interrupts 78
 Headerdatei 110
 <asm/segment.h> 446
 <asm/uaccess.h> 446
 <assert.h> 37, 124, 125
 <cpio.h> 37
 <ctype.h> 37, 124, 127
 <curses.h> 923
 <dirent.h> 37, 317
 <dlfcn.h> 1096
 <errno.h> 26, 37, 124, 128, 214
 <fcntl.h> 37, 222
 <float.h> 37, 124, 128
 <ftw.h> 37
 <getopt.h> 1032
 <grp.h> 37, 374
 <langinfo.h> 38
 <limits.h> 38, 124, 130, 222
 <linux/fs.h> 329
 <linux/locks.h> 334
 <linux/socket.h> 836
 <linux/un.h> 839
 <linux/vfs.h> 338
 <locale.h> 38, 124, 131
 <math.h> 38, 124, 136
 <netbd.h> 378
 <nl_types.h> 38
 <popt.h> 1038

<pwd.h> 38, 369
 <regex.h> 38
 <search.h> 38
 <setjmp.h> 38, 124, 403
 <shadow.h> 374
 <signal.h> 38, 125, 600
 <slang.h> 936
 <stdarg.h> 38, 121, 125, 193, 194
 <stddef.h> 38, 125, 141
 <stdio.h> 38, 125, 167
 <stdlib.h> 38, 125, 142
 <string.h> 38, 125, 152
 <stropts.h> 663, 682
 <sys/acct.h> 541
 <sys/ioctl.h> 986
 <sys/ipc.h> 38, 755
 <sys/kd.h> 986
 <sys/msg.h> 38
 <sys/param.h> 686
 <sys/sem.h> 38
 <sys/shm.h> 38
 <sys/socket.h> 816, 836
 <sys/stat.h> 38, 266
 <sys/times.h> 38
 <sys/types.h> 38, 51, 225
 <sys/uio.h> 696
 <sys/un.h> 839
 <sys/utsname.h> 39
 <sys/vfs.h> 338
 <sys/vt.h> 986
 <sys/wait.h> 39, 505
 <tar.h> 38
 <termios.h> 38, 880
 <time.h> 38, 125, 385
 <ulimit.h> 38
 <unistd.h> 38, 222
 <utime.h> 38
 <utmp.h> 380
 <varargs.h> 193, 194
 conio.h 971
 dos.h 997
 eighdr.h 1123
 Headerdateien 109, 124
 Heap 433, 435
 Home-Directory 14
 hostent Struktur 378, 864
 hostname Kommando 380
 htonl Funktion 857
 htons Funktion 857
 HUGE_VAL Konstante 137

I

ID

Prozeßgruppe 554
 IEEE 35
 iget Funktion 340
 Implementierung 35, 102
 implementierungsdefiniertes Verhalten 103
 in_addr Struktur 861
 inet_addr Funktion 859
 inet_aton Funktion 859
 inet_lnaof Funktion 860
 inet_makeaddr Funktion 861
 inet_netof Funktion 860
 inet_network Funktion 860
 inet_ntoa Funktion 859
 inetd Dämon 552, 704
 inetd Prozeß 552
 INFTIM Konstante 680
 init Funktion 75
 INIT_SYNCH Funktion 517, 645, 729
 initgroups Funktion 376
 init-Prozeß 485, 549
 initscr Funktion 923
 ino_t Datentyp 52
 i-node 282, 289
 Operationen (Linux intern) 342
 inode Struktur 339
 inode_operations Struktur 342
 insertln Funktion 932
 INT_MAX Konstante 131
 INT_MIN Konstante 131
 Interprozeßkommunikation 717, 753, 805
 interruptible_sleep_on Funktion 71
 Interrupts 78
 Intervalltimer 633
 ioctl Funktion 348, 663, 986
 iovec Struktur 696
 IPC 717, 753, 805
 IPC_INFO Konstante 775, 784
 IPC_INFO Konstante 763
 IPC_NOWAIT Konstante 760
 ipc_perm Struktur 755
 IPC_PRIVATE Konstante 754
 ipcrm Kommando 755, 770
 ipcs Kommando 755
 iput Funktion 341
 isalnum Funktion 127
 isalpha Funktion 127
 isascii Funktion 128
 isastream Funktion 664

isatty Funktion 890
iscntrl Funktion 127
isdigit Funktion 127
isgraph Funktion 127
islower Funktion 127
isprint Funktion 127
ispunct Funktion 127
isspace Funktion 127
isupper Funktion 127
isxdigit Funktion 127
ITIMER_PROF Konstante 633
ITIMER_REAL Konstante 633
ITIMER_VIRTUAL Konstante 633
itimerval Struktur 633

J

jiffies Variable 80
jmp_buf Datentyp 405
Jobkontrolle 559

K

Kalenderzeit erfragen 387
Kalenderzeit setzen 387
Kalenderzeit umwandeln 389, 391, 393
Kalenderzeiten Differenz 396
kdbg 1062
Keine Pufferung 201
Kern
 Dateitabellen 240
 Datenstrukturen 240
key_t Datentyp 754
kfree Funktion 462
kill Funktion 628
kill Kommando 604, 613
Kindprozeß 486
kmalloc Funktion 462
Kommando
 accton 541
 cat 232
 chattr 364
 chmod 268
 crontab 704
 crypt 373
 export 477
 fg 561
 finger 370
 hostname 380

ipcrm 755, 770
ipcs 755
kill 604, 613
last 380
login 380
lp 749
lsattr 364
mkfifo 745
newgrp 376
ps 503, 562
setenv 477
size 433
stty 561, 885
time 32
ulimit 441
uname 379
who 380

Kommandozeile

 Optionen 1023

Kommandozeile (in Makefile) 1103

Kommentar (make) 1102

Kompilierung

 bedingte 111

Konkatenation 107

Konsole

 Linux- 953

 Virtuell 985

Konstante

 _IOFBF 200

 _IOLBF 200

 _IONBF 201

 _PC_CHOWN_RESTRICTED 44

 _PC_LINK_MAX 44

 _PC_MAX_CANON 44

 _PC_MAX_INPUT 44

 _PC_NAME_MAX 44

 _PC_NO_TRUNC 44

 _PC_PATH_MAX 44

 _PC_PIPE_BUF 44

 _PC_VDISABLE 44

 _POSIX_ARG_MAX 40

 _POSIX_CHILD_MAX 40

 _POSIX_CHOWN_RESTRICTED 42,
44, 282

 _POSIX_JOB_CONTROL 42, 44, 555

 _POSIX_LINK_MAX 40

 _POSIX_MAX_CANON 40

 _POSIX_MAX_INPUT 40

 _POSIX_NAME_MAX 40

 _POSIX_NGROUPS_MAX 40

 _POSIX_NO_TRUNC 42, 44, 225

_POSIX_OPEN_MAX	40	LOG_CONS	711
_POSIX_PATH_MAX	40	LOG_CRIT	712
_POSIX_PIPE_BUF	40	LOG_CRON	711
_POSIX_SAVED_IDS	42, 44, 271	LOG_DAEMON	711
_POSIX_SSIZE_MAX	40	LOG_DEBUG	713
_POSIX_STREAM_MAX	40	LOG_EMERG	712
_POSIX_TZNAME_MAX	40	LOG_ERR	713
_POSIX_VDISABLE	42, 44	LOG_INFO	713
_POSIX_VERSION	42, 44	LOG_KERN	712
_SC_ARG_MAX	43	LOG_LOCAL0	712
_SC_CHILD_MAX	43	LOG_LOCAL1	712
_SC_CLK_TCK	43	LOG_LOCAL2	712
_SC_JOB_CONTROL	44	LOG_LOCAL3	712
_SC_NGROUPS_MAX	43	LOG_LOCAL4	712
_SC_OPEN_MAX	43	LOG_LOCAL5	712
_SC_PASS_MAX	43	LOG_LOCAL6	712
_SC_SAVED_IDS	44, 271	LOG_LOCAL7	712
_SC_STREAM_MAX	43	LOG_LPR	712
_SC_TZNAME_MAX	44	LOG_MAIL	712
_SC_VERSION	44	LOG_NDELAY	711
_SC_XOPEN_VERSION	44	LOG_NEWS	712
ARG_MAX	41, 43	LOG_NOTICE	713
CHILD_MAX	41, 43, 490	LOG_PERROR	711
CLK_TCK	32, 42	LOG_PID	711
CLOCKS_PER_SEC	32, 386, 398	LOG_SYSLOG	712
ENAMETOOLONG	225	LOG_USER	712
F_DUPFD	248	LOG_UUCP	712
F_FREESP	305	LOG_WARN	713
F_GETFD	248, 249	MAX_CANON	42, 44, 880
F_GETFL	248, 249	MAX_INPUT	42, 44, 880
F_GETLK	248, 569, 570	MAXHOSTNAMELEN	379
F_GETOWN	248, 249	MSGMAX	758
F_OK	277	MSGMNB	758
F_SETFD	248, 249	MSGMNI	758
F_SETFL	248, 249	MSGTQL	758
F_SETLK	248, 569, 570	NAME_MAX	42, 44, 225, 317
F_SETLKW	248, 569, 570	NBPG	686
F_SETOWN	248, 249	NCCS	880
FD_CLOEXEC	249	NGROUPS_MAX	41, 43, 376
FMNAMESZ	665	NULL	386
INFTIM	680	O_ACCMODE	250
IPC_INFO	775, 784	O_APPEND	223, 232, 249
IPC_NOWAIT	760	O_ASYNC	249
IPC_PRIVATE	754	O_CREAT	223
ITIMER_PROF	633	O_EXCL	223
ITIMER_REAL	633	O_NDELAY	223
ITIMER_VIRTUAL	633	O_NOCTTY	223
L_tmpnam	208	O_NONBLOCK	223, 249
LINK_MAX	42, 44, 294	O_RDONLY	223, 249
LOG_ALERT	712	O_RDWR	223, 249
LOG_AUTH	711	O_SYNC	224, 232, 249

O_TRUNC 223
O_WRONLY 223, 249
OPEN_MAX 41, 43, 222, 228
P_tmpdir 210
PASS_MAX 43
PATH_MAX 42, 44
PIPE_BUF 42, 44, 721
POSIX_SOURCE 51
R_OK 277
RLIM_INFINITY 440
RLIMIT_CORE 440
RLIMIT_CPU 440
RLIMIT_DATA 440
RLIMIT_FSIZE 440
RLIMIT_MEMLOCK 440, 690
RLIMIT_NOFILE 441
RLIMIT_NPROC 441
RLIMIT_OFILE 441
RLIMIT_RSS 441
RLIMIT_STACK 441
RLIMIT_VMEM 441
RMSGD 668
RMSGN 668
RNORM 668
RUSAGE_BOTH 444
RUSAGE_CHILDREN 444
RUSAGE_SELF 444
S_BANDURG 682
S_ERROR 682
S_HANGUP 682
S_HIPRI 682
S_IFMT 267
S_INPUT 682
S_IRGRP 224, 268, 274, 279, 312
S_IROTH 224, 268, 274, 279, 312
S_IRUSR 224, 268, 274, 279, 312
S_IRWXG 224, 274, 279, 313
S_IRWXO 224, 274, 279, 313
S_IRWXU 224, 274, 279, 313
S_ISGID 224, 270, 274, 312
S_ISUID 224, 270, 274, 312
S_ISVTX 224, 273, 274, 312
S_IWGRP 224, 268, 274, 279, 312
S_IWOTH 224, 268, 274, 279, 312
S_IWUSR 224, 268, 274, 279, 312
S_IXGRP 224, 268, 274, 279, 312
S_IXOTH 224, 268, 274, 279, 312
S_IXUSR 224, 268, 274, 279, 312
S_MSG 682
S_OUTPUT 682
S_RDBAND 682
S_RDNORM 682
S_WRBAND 682
S_WRNORM 682
SEEK_CUR 205, 234
SEEK_END 205, 234
SEEK_SET 205, 234
SEMMNI 773
SEMMNS 773
SEMMSL 773
SEMOPN 773
SEMVMX 773
SHMLBA 785
SHMMAX 781
SHMMIN 781
SHMMNI 781
SHMSEG 781
SIG_ERR 601
SIGXCPU 440
SIGXFSZ 440
SNDPIPE 667
SNDZERO 667
SOCK_DGRAM 835
SOCK_STREAM 835
SSIZE_MAX 41
STDERR_FILENO 222
stderr 168
stdin 168
STDIN_FILENO 222
stdout 168
STDOUT_FILENO 222
STREAM_MAX 41, 43
TMP_MAX 208
TZNAME_MAX 41, 44
UIO_MAX 697
UIO_MAXIOV 697
VT_ACKAOK 990
VT_ACTIVATE 988
VT_DISALLOCATE 988
VT_GETMODE 987
VT_GETSTATE 988
VT_KIOCSOUND 990
VT_LOCKSWITCH 993
VT_OPENQRY 987
VT_RELDISP 990
VT_SETMODE 990
VT_UNLOCKSWITCH 993
VT_WAITACTIVE 988
W_OK 277
WCONTINUED 509
WNOHANG 508
WNOWAIT 509

- WUNTRACED 508
- X_OK 277
- XOPEN_VERSION 44
- Kontrollterminal 557
- Kontrollzeichen 956
- Kopieren
 - Bildschirmausschnitt 932
- Koprozeß 737
- Korn-Shell 10
- Kostante
 - IPC_INFO 763
- kreieren eines Prozesses 486
- Kreierungsmaske 278
- ksh Programm 10

- L**

- L_tmpnam Konstante 208
- labs Funktion 145
- last Kommando 380
- Lautsprecher
 - ausschalten 998
 - einschalten 998
- LC_ALL Makro 132
- LC_COLLATE Makro 132
- LC_CTYPE Makro 133
- LC_MONETARY Makro 133
- LC_NUMERIC Makro 133
- LC_TIME Makro 133
- lchown Funktion 281, 299
- lconv Struktur 134
- ld Linker 1060
- LDBL_DIG Konstante 129
- LDBL_MANT_DIG Konstante 128
- ldconfig Kommando 1089
- ldd Kommando 1093
- ldexp Funktion 137
- ldiv Funktion 147
- ldiv_t Datentyp 142
- Lesen von Directory 317
- LILO 289
- Limit
 - Ressourcen 439
- Limits 39
- Linearer Adreßraum 448
- LINES Variable 923
- Link 292, 295, 297
 - Hard- 292
 - Soft- 297
 - symbolischer 12, 266, 297
- link Funktion 295, 299, 343
- LINK_MAX Konstante 42, 44, 294
- Linux 7, 37
- Linux-Konsole 953
- listen Funktion 837
- Little-Endian 856
- ln Kommando 292, 297
- lnamei Funktion 341
- localeconv Funktion 134
- localtime Funktion 389
- lock_super Funktion 334
- lockf Funktion 571
- Löcher in Dateien 306
- löschen
 - Bildschirm 923
- log Funktion 137
- log Gerätetreiber (SVR4) 708
- LOG_ALERT Konstante 712
- LOG_AUTH Konstante 711
- LOG_CONS Konstante 711
- LOG_CRIT Konstante 712
- LOG_CRON Konstante 711
- LOG_DAEMON Konstante 711
- LOG_DEBUG Konstante 713
- LOG_EMERG Konstante 712
- LOG_ERR Konstante 713
- LOG_INFO Konstante 713
- LOG_KERN Konstante 712
- LOG_LOCAL0 Konstante 712
- LOG_LOCAL1 Konstante 712
- LOG_LOCAL2 Konstante 712
- LOG_LOCAL3 Konstante 712
- LOG_LOCAL4 Konstante 712
- LOG_LOCAL5 Konstante 712
- LOG_LOCAL6 Konstante 712
- LOG_LOCAL7 Konstante 712
- LOG_LPR Konstante 712
- LOG_MAIL Konstante 712
- log_meld (eigene Fehlerroutine) 1124
- LOG_NDELAY Konstante 711
- LOG_NEWS Konstante 712
- LOG_NOTICE Konstante 713
- LOG_PERROR Konstante 711
- LOG_PID Konstante 711
- LOG_SYSLOG Konstante 712
- LOG_USER Konstante 712
- LOG_UUCP Konstante 712
- LOG_WARN Konstante 713
- log10 Funktion 137

Log-Datei 703
 Login
 Netzwerk 552
 Terminal 549
 login Kommando 380
 login Programm 550
 Loginname 9
 Login-Prozeß 549
 logisches Laufwerk 288
 lokal-spezifisches Verhalten 103
 long long Datentyp 1059
 LONG_MAX Konstante 131
 LONG_MIN Konstante 131
 longjmp Funktion 404
 lookup Funktion 343
 lost_ticks Variable 80
 lost_ticks_system Variable 80
 lp Kommando 749
 lpd Dämon 704
 lpsched Dämonprozeß 749
 lsattr Kommando 364
 lseek Funktion 234, 347
 lstat Funktion 264, 299

M

main Funktion 420
 major Makro 325
 Major Number 325
 make
 Abhängigkeitsbeschreibung 1102
 Aufrufsyntax 1106
 cleanup 1109
 dependency line 1102
 Fortsetzungszeichen 1105
 Kommandozeile in Makefile 1103
 Kommentar 1102
 Makro 1112
 Optionen 1109
 Struktur eines Makefiles 1101
 Time stamps 1105
 Zeitmarken 1105
 make Kommando 1100
 Makefile 1101
 Makro 106
 CLOCKS_PER_SEC 538
 CMMSG_DATA 820
 CTRL 956
 FD_CLR 674
 FD_ISSET 674

FD_SET 674
 FD_ZERO 674
 getc 175
 getchar 173
 putc 175
 putchar 173
 setjmp 404
 WCOREDUMP 505
 WIFEXITED 505
 WIFEXITSTATUS 505
 WIFSIGNALED 505
 WIFSTOPPED 505
 WSTOPSIG 505
 WTERMSIG 505
 Makro (make) 1112
 Makrodefinition
 rekursiv 109
 malloc Funktion 142, 433
 mandatory locking 579
 Master Boot Record 288
 MAX_CANON Konstante 42, 44, 880
 MAX_INPUT Konstante 42, 44, 880
 MAXHOSTNAMELEN Konstante 379
 MB_CUR_MAX Konstante 142
 MB_LEN_MAX Konstante 130
 mblen Funktion 152
 MBR 288
 mbstowcs Funktion 152
 mbtowc Funktion 152
 mcheck Funktion 1080
 mem_map_t Datentyp 469
 memchr Funktion 153
 memcmp Funktion 153
 memcpy Funktion 153
 memcpy_fromfs Funktion 447
 memcpy_toofs Funktion 447
 memmove Funktion 153
 Memory Mapped I/O 683
 memset Funktion 154
 Message Queues 753, 756
 Messages 657
 MIN Variable 913
 minor Makro 325
 minor Number 325
 mk_pte Funktion 454
 mkdir Funktion 299, 313, 343
 mke2fs 291
 mke2fs Kommando 291
 mkfifo Funktion 299, 744
 mkfifo Kommando 745
 mknod Funktion 299, 344

mktime Funktion 389
mlock Funktion 691
mlockall Funktion 691
mmap Funktion 348, 441, 683
mode_t Datentyp 52, 225
modf Funktion 137
modification time 307
mount Funktion 330, 331
mount Kommando 272
mount_root Funktion 330
move Funktion 924
move_last_runqueue Funktion 87
mpr Bibliothek 1080
msgctl Funktion 762
msgget Funktion 758
msghdr Struktur 816, 819
msgid_ds Struktur 757
msginfo Struktur 763
MSGMAX Konstante 758
MSGMNB Konstante 758
MSGMNI Konstante 758
msgrcv Funktion 761
msgsnd Funktion 759
MSGTQL Konstante 758
msync Funktion 689
Multiplexing 671
munlock Funktion 691
munlockall Funktion 691
munmap Funktion 689
musik1.c 997
mv Kommando 295
mvaddch Funktion 924
mvaddstr Funktion 924
mvprintw Funktion 924

N

Nachrichtenwarteschlangen 753, 756
NAME_MAX Konstante 42, 44, 225, 317
Named Pipes 12, 265
namei Funktion 341
nanosleep Funktion 636
NBPG Konstante 686
NCCS Konstante 880
NDEBUG Makro 125
netent Struktur 378
Netzwerkinformation 377
Netzwerk-Logins 552
Netzwerkprogrammierung 856
newgrp Kommando 376

nftw Funktion 301, 318
NGROUPS_MAX Konstante 41, 43, 376
nichtdruckbare Zeichen 105
nicht-lokaler Sprung 403
nlink_t Datentyp 52
nm Kommando 1086
nocbreak Funktion 929
noecho Funktion 928
nosound() 998
notify_change Funktion 336
NR_TASKS Konstante 69
ntohl Funktion 857
ntohs Funktion 857
NULL Konstante 386
NULL Makro 141
Null-Signal 629

O

O_ACCMODE Konstante 250
O_APPEND Konstante 223, 232, 249
O_ASYNC Konstante 249
O_CREAT Konstante 223
O_EXCL Konstante 223
O_NDELAY Konstante 223
O_NOCTTY Konstante 223
O_NONBLOCK Konstante 223, 249
O_RDONLY Konstante 223, 249
O_RDWR Konstante 223, 249
O_SYNC Konstante 224, 232, 249
O_TRUNC Konstante 223
O_WRONLY Konstante 223, 249
Objekt 102
off_t Datentyp 52, 235
offsetof Makro 141
open Funktion 222, 299, 349
OPEN_MAX Konstante 41, 43, 222, 228
open_namei Funktion 337
opendir Funktion 299, 317
openlog Funktion 711
Operationen
 File (Linux intern) 346
 i-node (Linux intern) 342
 Superblock (Linux intern) 334
Operator
 # 107
Operator ## 108
option Struktur 1032
Optionen 1023
 make 1109

P

- P_tmpdir Konstante 210
- Page Faults 474
- Page Middle Directory 449, 451
- page Struktur 469
- page_hash_table Array 470
- pagedaemon 486
- Pagedirectory 449, 450
- Page-Größe 686
- Pages 445
- Pagetabelle 449, 452
- Paging 463
- paging 486
- Parameter 102
- parent process 483, 486
- Parent-Directory 14
- Partition 286
- Partitionstabelle 288
- PASS_MAX Konstante 43
- passwd Kommando 269
- passwd Struktur 369
- Paßwortdatei 10, 369
- PATH_MAX Konstante 42, 44, 225
- pathconf Funktion 43, 299
- pause Funktion 634
- pclose Funktion 731
- Peripheriegeräte 325
- permission Funktion 346
- perror Funktion 27, 214
- Pfadname 14
 - absolut 14
 - relativ 14
- pgd_alloc Funktion 451
- pgd_bad Funktion 451
- pgd_clear Funktion 451
- pgd_free Funktion 451
- pgd_none Funktion 451
- pgd_offset Funktion 451
- pgd_present Funktion 451
- pgd_val Makro 450
- pgprot_t Struktur 454
- pgprot_val Makro 454
- PID 22, 483
- pid_t Datentyp 52
- Pipe 718
 - benannt 744
 - Named 12, 265
- pipe Funktion 718
- PIPE_BUF Konstante 42, 44, 721
- pmd_alloc Funktion 451
- pmd_alloc_kernel Funktion 452
- pmd_bad Funktion 452
- pmd_clear Funktion 452
- pmd_free Funktion 452
- pmd_free_kernel Funktion 452
- pmd_none Funktion 452
- pmd_offset Funktion 452
- pmd_page Funktion 452
- pmd_present Funktion 452
- pmd_val Makro 451
- poll Funktion 678
- pollfd Struktur 678
- Polling 516, 672
- popen Funktion 731, 1007
- P-Operation 779
- popt Softwarepaket 1037
- poptAddAlias Funktion 1045
- poptAlias Struktur 1045
- poptBadOption Funktion 1044
- poptContext Struktur 1040
- poptFreeContext Funktion 1040
- poptGetArg Funktion 1042
- poptGetArgs Funktion 1042
- poptGetContext Funktion 1040
- poptGetNextOpt Funktion 1041
- poptGetOptArg Funktion 1042
- poptOption Struktur 1038
- poptParseArgvString Funktion 1046
- poptPeekArg Funktion 1042
- poptPrintHelp Funktion 1043
- poptPrintUsage Funktion 1043
- poptReadConfigFile Funktion 1045
- poptReadDefaultConfig Funktion 1045
- poptResetContext Funktion 1040
- poptSterror Funktion 1044
- poptStuffArgs Funktion 1046
- positionieren
 - Cursor 924
- POSIX 35
- POSIX_SOURCE Konstante 51
- pow Funktion 137
- PPID 483
- Präprozessor 106
 - #elif 111
 - #else 112
 - #endif 112
 - #error 112
 - #if 111
 - #ifdef 111
 - #ifndef 111
 - #line 112

- #pragma 112
- #undef 112
- __DATE__ 113
- __FILE__ 113
- __LINE__ 113
- __STDC__ 113
- __TIME__ 113
- Preallokation 362
- Primären Partitionen 288
- Primitive Systemdatentypen 51, 119
- printf Funktion 185
- printw Funktion 924
- Programm
 - /bin/bash 11
 - /bin/csh 10
 - /bin/ksh 10
 - /bin/sh 10
 - /bin/tcsh 11
 - autofahr.c 998
 - bash 11
 - buchmemo.c 1001
 - csh 10
 - Filter 734
 - gatter.c 1003
 - getty 549
 - ksh 10
 - login 550
 - musik1.c 997
 - sh 10
 - tcsh 11
 - TELNET 553
 - ttymon 552
- Programm beenden 423, 424
- protoent Struktur 378
- Prototypen 119
- Prozeß 21, 60, 419
 - Beendigung 502
 - Buchführung 541
 - child 486
 - Dämon- 575, 703
 - Eltern- 483, 486
 - Ende 421, 423, 424
 - Environment 427
 - Exit-Status 421
 - Gruppe 554
 - Hierarchie 485
 - ID 483
 - inetd 552
 - Informationen 537
 - init 549
 - init- 485
 - Kennung 483
 - Kind- 486
 - Ko- 737
 - kreieren 486
 - Login 549
 - pagedaemon 486
 - parent 483, 486
 - Scheduler- 485
 - Speicher-Layout 431
 - Startup 419
 - suspendieren 642
 - telnetd 553
 - verwaist 503
 - Zombie 503
- Prozeßgruppe 554
 - verwaist 565
- Prozeßgruppenführer 554
- Prozeßgruppen-ID 554
- Prozeßhierarchie 485
- Prozeß-ID 22
- Prozeßsteuerung 483
- Prozeßtabelle 69, 240
- ps Kommando 503, 562
- Pseudoterminal 553
- psignal Funktion 614
- pt_val Makro 452
- pte_alloc Funktion 454
- pte_alloc_kernel Funktion 454
- pte_clear Funktion 454
- pte_dirty Funktion 455
- pte_exec Funktion 455
- pte_exprotect Funktion 455
- pte_free Funktion 455
- pte_free_kernel Funktion 455
- pte_mkclean Funktion 455
- pte_mkdirty Funktion 455
- pte_mkexec Funktion 455
- pte_mkold Funktion 455
- pte_mkread Funktion 456
- pte_mkwrite Funktion 456
- pte_mkyoungt Funktion 456
- pte_modify Funktion 456
- pte_none Funktion 456
- pte_offset Funktion 456
- pte_page Funktion 456
- pte_present Funktion 456
- pte_rdp protect Funktion 456
- pte_read Funktion 457
- pte_write Funktion 457
- pte_wrprotect Funktion 457
- pte_young Funktion 457

ptrdiff_t Datentyp 52, 141
Puffer
 leeren 203
Puffercache 327
Pufferung
 keine 201
 Voll- 200
 voreingestellt 201
 Zeilen- 200
Pufferung (Standard-E/A) 200
put_inode Funktion 337
put_super Funktion 337
put_user Funktion 447
put_user_ret Funktion 447
putc Makro/Funktion 175
putchar Makro/Funktion 173
putenv Funktion 430, 479
putmsg Funktion 659
putpmsg Funktion 659
puts Funktion 179

Q

qsort Funktion 150

R

R_OK Konstante 277
race condition 515
raise Funktion 628
rand Funktion 143
RAND_MAX Konstante 142
ranlib Kommando 1083
read Funktion 229, 347
read_inode Funktion 335
read_super Funktion 333
readdir Funktion 317, 347
readlink Funktion 299, 302, 344
readv Funktion 695
reale GID 269, 281
reale UID 269, 281
realloc Funktion 142
record locking 568
Reentrant-Funktionen 627
refresh Funktion 924
regcomp Funktion 1016
regerror Funktion 1019
regexec Funktion 1018
regfree Funktion 1019

register Schlüsselwort 412
register_filesystem Funktion 329
regmatch_t Struktur 1019
regular file 11, 265
rekursive Makrodefinitionen 109
relativer Pfadname 14
release Funktion 348
remount_fs Funktion 338
remove Funktion 212, 299
remove_wait_queue Funktion 70
rename Funktion 213, 299, 344
Ressourcenlimit 439
revalidate Funktion 349
rewind Funktion 207
rewinddir Funktion 317
RLIM_INFINITY Konstante 440
rlim_t Datentyp 52
rlimit Struktur 440
RLIMIT_CORE Konstante 440
RLIMIT_CPU Konstante 440
RLIMIT_DATA Konstante 440
RLIMIT_FSIZE Konstante 440
RLIMIT_MEMLOCK Konstante 440, 690
RLIMIT_NOFILE Konstante 441
RLIMIT_NPROC Konstante 441
RLIMIT_OFILE Konstante 441
RLIMIT_RSS Konstante 441
RLIMIT_STACK Konstante 441
RLIMIT_VMEM Konstante 441
rmdir Funktion 299, 314, 344
RMSGD Konstante 668
RMSGN Konstante 668
RNORM Konstante 668
Root-Directory 13
run_old_timers Funktion 84
run_timer_list Funktion 84
rusage Struktur 444, 515
RUSAGE_BOTH Konstante 444
RUSAGE_CHILDREN Konstante 444
RUSAGE_SELF Konstante 444

S

S_BANDURG Konstante 682
S_ERROR Konstante 682
S_HANGUP Konstante 682
S_HIPRI Konstante 682
S_IFMT Konstante 267
S_INPUT Konstante 682
S_IRGRP Konstante 224, 268, 274, 279, 312

- S_IROTH Konstante 224, 268, 274, 279, 312
S_IRUSR Konstante 224, 268, 274, 279, 312
S_IRWXG Konstante 224, 274, 279, 313
S_IRWXO Konstante 224, 274, 279, 313
S_IRWXU Konstante 224, 274, 279, 313
S_ISBLK Makro 266
S_ISCHR Makro 266
S_ISDIR Makro 266
S_ISFIFO Makro 266
S_ISGID Konstante 224, 270, 274, 312
S_ISLNK Makro 266
S_ISREG Makro 266
S_ISSOCK Makro 266
S_ISUID Konstante 224, 270, 274, 312
S_ISVTX Konstante 224, 273, 274, 312
S_IWGRP Konstante 224, 268, 274, 279, 312
S_IWOTH Konstante 224, 268, 274, 279, 312
S_IWUSR Konstante 224, 268, 274, 279, 312
S_IXGRP Konstante 224, 268, 274, 279, 312
S_IXOTH Konstante 224, 268, 274, 279, 312
S_IXUSR Konstante 224, 268, 274, 279, 312
S_MSG Konstante 682
S_OUTPUT Konstante 682
S_RDBAND Konstante 682
S_RDNORM Konstante 682
S_WRBAND Konstante 682
S_WRNORM Konstante 682
Saved Set-Group-ID Bit 270
Saved Set-User-ID 270
saved Set-User-ID-Bit 533
Saved-Text Bit 272
sbrk Funktion 435
scanf Funktion 180
scanw Funktion 929
SCHAR_MAX Konstante 130
SCHAR_MIN Konstante 130
SCHED_FIFO Konstante 85
SCHED_OTHER Konstante 85
sched_param Struktur 85
SCHED_RR Konstante 85
sched_scheduler Funktion 86
schedule Funktion 87
Scheduler 85
Scheduler-Prozeß 485
Schedulingalgorithmus 85
SEEK_CUR Konstante 205, 234
SEEK_END Konstante 205, 234
SEEK_SET Konstante 205, 234
select Funktion 347, 635, 673
sem Struktur 772
sem_queue Struktur 772
sem_undo Struktur 772
Semaphore 753, 770
semaphore Struktur 72
sembuf Struktur 777
semctl Funktion 774
semget Funktion 773
semid_ds Struktur 771
Semigraphik
 Borland-C 968
 Turbo-C 968
seminfo Struktur 775
SEMMNI Konstante 773
SEMMNS Konstante 773
SEMMSL Konstante 773
semop Funktion 776
SEMOPN Konstante 773
semun Struktur 774
SEMVMX Konstante 773
send_fd Funktion 813
send_fehl Funktion 813
senden von Signalen 628
sendmail Dämon 703
Sequencing 834
serv_bereit Funktion 829
serv_initverbind Funktion 829
servent Struktur 378, 867
Session 556
set_fs Funktion 448
SET_PAGE_DIR Funktion 451
set_pte Funktion 457
setbuf Funktion 201
setegid Funktion 535
setenv Funktion 430, 479
setenv Kommando 477
seteuid Funktion 535
setfsuid Funktion 65, 536
setfsuid Funktion 65, 536
setgid Funktion 532
setgrent Funktion 375
Set-Group-ID Bit 269, 281
setgroups Funktion 376
sethostname Funktion 380
setitimer Funktion 634
setjmp Funktion / Makro 404
setlocale Funktion 132
setpgid Funktion 555
setpwent Funktion 371
setregid Funktion 535
setreuid Funktion 535
setrlimit Funktion 439
setscheduler Funktion 86

- setservent Funktion 868
- setsid Funktion 556
- setsockopt Funktion 873
- setuid Funktion 532
- setup Funktion 330
- setup_arch Funktion 75
- Set-User-ID Bit 269, 281
- setvbuf Funktion 201
- sh Programm 10
- shared Memory 753, 780
- shared objects 1095
- Shell 10
- shm_swap Funktion 473
- shmat Funktion 784
- shmdt Funktion 786
- shmget Funktion 782
- shmid_ds Struktur 780
- shminfo Struktur 784
- SHMMAX Konstante 781
- SHMMIN Konstante 781
- SHMMNI Konstante 781
- SHMSEG Konstante 781
- shrink_mmap Funktion 473
- SHRT_MAX Konstante 131
- SHRT_MIN Konstante 130
- SIG Signal 610
- sig_atomic_t Datentyp 52
- SIG_DFL Signal 601
- SIG_ERR Konstante 601
- SIG_IGN Signal 600
- SIGABRT Signal 610, 648
- sigaction Funktion 619
- sigaction Struktur 619
- sigaddset Funktion 618
- SIGALRM Signal 610, 630
- sigatomic_t Datentyp 641
- SIGBUS Signal 610, 687
- SIGCHLD Signal 504, 610
- SIGCLD Signal 610
- SIGCONT Signal 610
- sigdelset Funktion 618
- sigemptyset Funktion 618
- SIGEMT Signal 611
- sigfillset Funktion 618
- SIGFPE Signal 611
- SIGHUP Signal 611
- SIGILL Signal 611
- SIGINFO Signal 611
- siginfo Struktur 650
- SIGINT Signal 611
- SIGIO Signal 611, 673, 681, 683
- SIGIOT Signal 612
- sigismember Funktion 618
- sigjmp_buf Datentyp 641
- SIGKILL Signal 612
- siglongjmp Funktion 639
- Signal 29
 - Null- 629
 - SIG 610
 - SIG_DFL 601
 - SIG_IGN 600
 - SIGABRT 610, 648
 - SIGALRM 610, 630
 - SIGBUS 610, 687
 - SIGCHLD 504, 610
 - SIGCLD 610
 - SIGCONT 610
 - SIGEMT 611
 - SIGFPE 611
 - SIGHUP 611
 - SIGILL 611
 - SIGINFO 611
 - SIGINT 611
 - SIGIO 611, 673, 681, 683
 - SIGIOT 612
 - SIGKILL 612
 - SIGPIPE 612
 - SIGPOLL 612, 673, 681
 - SIGPROF 612
 - SIGPWR 612
 - SIGQUIT 612
 - SIGSEGV 612, 687
 - SIGSTOP 612
 - SIGSYS 613
 - SIGTERM 613
 - SIGTRAP 613
 - SIGTSTP 613
 - SIGTTIN 613
 - SIGTTOU 613
 - SIGURG 613, 683
 - SIGUSR1 614
 - SIGUSR2 614
 - SIGVTALRM 614
 - SIGWINCH 614, 919
 - SIGXCPU 614
 - SIGXFSZ 614
- signal Funktion 30, 600
- Signale 599
 - senden 628
- Signal-Handler 30
- Signalkonzept 599, 600
- Signalmaske 622

- Signalmengen 618
- Signalnamen 607
- Signalnummer 607
- sigpending Funktion 625
- SIGPIPE Signal 612
- SIGPOLL Signal 612, 673, 681
- sigprocmask Funktion 623
- SIGPROF Signal 612
- SIGPWR Signal 612
- SIGQUIT Signal 612
- SIGSEGV Signal 612, 687
- sigset_t Datentyp 52, 618
- sigsetjmp Funktion 639
- SIGSTOP Signal 612
- sigsuspend Funktion 642
- SIGSYS Signal 613
- SIGTERM Signal 613
- SIGTRAP Signal 613
- SIGTSTP Signal 613
- SIGTTIN Signal 613
- SIGTTOU Signal 613
- SIGURG Signal 613, 683
- SIGUSR1 Signal 614
- SIGUSR2 Signal 614
- SIGVTALRM Signal 614
- SIGWINCH Signal 614, 919
- SIGXCPU Konstante 440
- SIGXCPU Signal 614
- SIGXFSZ Konstante 440
- SIGXFSZ Signal 614
- sin Funktion 137
- sinh Funktion 137
- size Kommando 433
- size_t Datentyp 52, 141, 230, 385
- S-Lang 936
- sleep Funktion 635
- sleep_on Funktion 71
- smap Funktion 346
- SNDPIPE Konstante 667
- SNDZERO Konstante 667
- SOCK_DGRAM Konstante 835
- SOCK_STREAM Konstante 835
- sockaddr Struktur 836
- sockaddr_in Struktur 858
- sockaddr_un Struktur 839
- Socket 12, 833
- socket Funktion 835
- socketpair Funktion 841
- Sockets 266, 856
- socklist Kommando 876
- Soft-Link 297
- SOLARIS 7
- sound() 998
- special file 12, 265
- Speicher
 - Allokierung 433
 - Allokierung (Stack) 439
 - dynamisch 433
 - dynamisch (Stack) 439
 - Freigabe 438
- sperrern von Dateien 567, 568
- spezielle Eingabezeichen 896
- splitvt Kommando 994
- sprintf Funktion 192
- Sprung
 - nicht-lokal 403
- sqrt Funktion 137
- srand Funktion 143
- sscanf Funktion 192
- SSIZE_MAX Konstante 41
- ssize_t Datentyp 41, 52, 230
- st_blksize 202
- Stack 432
- Standardausgabe 18, 168, 221
- Standard-E/A-Funktionen 18, 167
- Standardeingabe 18, 168, 221
- Standardfehlerausgabe 18, 168, 221
- Standard-Headerdateien 124
- Standardisierung 35
- standend Funktion 927
- standout Funktion 927
- start_kernel Funktion 73
- Startup-Routine 419
- stat Funktion 264, 299
- stat Struktur 202, 263, 264
- statfs Funktion 338
- statfs Struktur 338
- static Schlüsselwort 412
- Statische Bibliotheken 1082
- stderr Konstante 168
- STDERR_FILENO Konstante 18, 222
- stdin Konstante 168
- STDIN_FILENO Konstante 18, 222
- stdout Konstante 168
- STDOUT_FILENO Konstante 18, 222
- Steuertasten 965
- Sticky Bit 272
- stime Funktion 387
- str_list Struktur 665
- str_mlist Struktur 665
- strace Kommando 1067
- strace Logger 709

- strbuf Struktur 658
- strcat Funktion 154
- strchr Funktion 154
- strcmp Funktion 155
- strcoll Funktion 155
- strcpy Funktion 155
- strcspn Funktion 155
- STREAM
 - /dev/conslog 708
 - /dev/log 708
- Stream
 - siehe Datei 168
- Stream Pipes 805, 807
- STREAM_MAX Konstante 41, 43
- stream_pipe Funktion 807
- STREAM-Messages 657
- Stream-Protokolle 834
- STREAMS 655
- strerr Logger 709
- strerror Funktion 27, 155, 215
- strftime Funktion 393
- String
 - Lesen (formatiert) 192
 - Schreiben (formatiert mit Argumentzeiger) 194
 - Schreiben (formatiert) 192
- strlen Funktion 155
- strncat Funktion 155
- strncmp Funktion 156
- strncpy Funktion 156
- strpbrk Funktion 156
- strptime Funktion 394
- strrchr Funktion 157
- strrecvfd Struktur 814
- strspn Funktion 158
- strstr Funktion 158
- strtod Funktion 146
- strtok Funktion 158
- strtoul Funktion 146
- Struktur
 - acct 541
 - cmsghdr 820
 - Datei- 11
 - flock 569
 - group 374
 - hostent 378, 864
 - in_addr 861
 - iovec 696
 - ipc_perm 755
 - itimerval 633
 - msghdr 816, 819
 - msgid_ds 757
 - netent 378
 - option 1032
 - passwd 369
 - pollfd 678
 - poptAlias 1045
 - poptContext 1040
 - poptOption 1038
 - protoent 378
 - regmatch_t 1019
 - rlimit 440
 - rusage 515
 - sem 772
 - sembuf 777
 - semid_ds 771
 - semun 774
 - servent 378, 867
 - shmid_ds 780
 - sigaction 619
 - siginfo 650
 - sockaddr_in 858
 - stat 202
 - str_list 665
 - str_mlist 665
 - strbuf 658
 - strrecvfd 814
 - termios 880
 - timeval 388, 633, 675
 - timezone 388
 - tm 385
 - tms 537
 - utmp 380
 - utsname 379
 - winsize 919
- strxfrm Funktion 160
- stty Kommando 561, 885
- super_block Struktur 332
- super_blocks Array 332
- super_operations Struktur 334
- Superblock 287
 - Operationen (Linux intern) 334
- SVID 36
- SVR4 36
- swap_out Funktion 473
- swapoff Funktion 468
- swapon Funktion 465
- swapper 485
- symbolische Links 12, 266, 297
- symbolische Verweise 12
- symlink Funktion 301, 343

sync Funktion 328
 Synchronisation 515
 sys_chmod Funktion 337
 sys_chown Funktion 337
 sys_fchmod Funktion 337
 sys_fchown Funktion 337
 sys_fstatfs Funktion 338
 sys_ftruncate Funktion 337
 sys_mount Funktion 331
 sys_setup Funktion 330
 sys_siglist Variable 614
 sys_statfs Funktion 338
 sys_truncate Funktion 337
 sys_umount Funktion 332
 sys_utime Funktion 337
 sys_write Funktion 337
 sysconf Funktion 43, 441
 syslog Funktion 709, 711
 syslogd Dämon 703
 syslogd Logger 709
 System booten 72
 system Funktion 143, 527
 System V Interface Definition 36
 System V Release 4 7, 36
 Systemaufrufe 33
 Systemdatentypen 51, 119
 Systeminformationen 369

T

Tabelle

Datei- 240
 Prozeß- 240
 v-node- 240
 tan Funktion 137
 tar Kommando 311
 Task 21, 60, 63
 task_struct Struktur 63
 Tasten
 Cursorsteuer- 929
 Funktions- 929
 tcdrain Funktion 911
 tcflag_t Datentyp 880
 tcflow Funktion 911
 tcflush Funktion 911
 tcgetattr Funktion 887
 tcgetpgrp Funktion 558
 TCP/IP 856
 tcsendbreak Funktion 911
 tcsetattr Funktion 887

tcsetpgrp Funktion 558
 tcsh Programm 11
 TC-Shell 11
 TELNET Programm 553
 telnetd Dämon 553
 telnetd Prozeß 553
 tempnam Funktion 209
 temporäre Dateien 207
 termcap 921
 Terminal
 Attribute 887
 Baudrate 908
 Fenstergröße 919
 Flags 882, 900
 Identifizierung 887
 Kontroll 557
 Modus 879, 912
 Steuerung 879
 Virtuell 985
 Terminal-Logins 549
 terminfo 921
 termios Struktur 880
 text segment 431
 Thread 62
 time Funktion 387
 time Kommando 32
 Time stamps (make) 1105
 TIME Variable 913
 time_t Datentyp 32, 52, 385
 Timer 633
 timer_bh Funktion 81
 timer_list Struktur 84
 timer_struct Struktur 84
 timer_table Array 84
 Timerinterrupt 80
 times Funktion 537
 timespec Funktion 636
 timeval Funktion 636
 timeval Struktur 80, 388, 633, 675
 timezone Struktur 388
 tm Struktur 385
 TMP_MAX Konstante 208
 TMPDIR Variable 209
 tmpfile Funktion 209
 tmpnam Funktion 208
 tms Struktur 537
 tolower Funktion 127
 touch Kommando 311, 1110
 toupper Funktion 127
 Trigraphs 104
 truncate Funktion 299, 305, 345

try_to_free_page Funktion 473
 ttymon Programm 552
 ttyname Funktion 892
 TZ Variable 396
 TZNAME_MAX Konstante 41, 44

U

UCHAR_MAX Konstante 130
 Übung
 autofahr.c 998
 buchmemo.c 1001
 gatter.c 1003
 musik1.c 997
 UID 269, 281
 uid_t Datentyp 52
 UINT_MAX Konstante 131
 UIO_MAX Konstante 697
 UIO_MAXIOV Konstante 697
 ulimit Kommando 441
 ULONG_MAX Konstante 131
 umask Funktion 278
 umask Kommando 280
 umount Funktion 332
 uname Funktion 378
 uname Kommando 379
 undefiniertes Verhalten 103
 ungetc Funktion 177
 Unix-Domain-Sockets 838
 Unix-Implementierungen 35
 Unix-Standardisierung 35
 unlink Funktion 296, 299, 343
 unlock_super Funktion 334
 unsetenv Funktion 430, 479
 unspezifiziertes Verhalten 103
 up Funktion 72
 update Dämon 704
 update_one_process Funktion 82
 update_process_times Funktion 82
 update_times Funktion 81
 update_wall_times Funktion 82
 User-ID 28, 269, 281
 USHRT_MAX Konstante 131
 usleep Funktion 635
 utime Funktion 308
 utimes Funktion 309
 utmp Datei 380
 utmp Struktur 380
 utsname Struktur 379

V

va_arg Makro 121
 va_end Makro 121
 va_list Datentyp 121
 va_start Makro 121
 Variable
 COLS 923
 environ 427
 errno 214
 LINES 923
 MIN 913
 sys_siglist 614
 TIME 913
 TZ 396
 Verhalten
 implementierungsdefiniertes 103
 lokal-spezifisches 103
 undefiniertes 103
 unspezifiziertes 103
 verwaiste Prozeßgruppe 565
 verwaister Prozeß 503
 Verweis
 symbolisch 12
 Verzeichnis 265
 Verzögerung 998
 vfork Funktion 498
 vfprintf Funktion 193
 vfree Funktion 463
 Vielbyte-Zeichen 105
 Virtual File System 283, 329
 Virtuelle Konsole 985
 Virtueller Adreßraum 457
 Virtuelles Terminal 985
 vm_area_struct Struktur 457
 vm_operations_struct Struktur 459
 vmalloc Funktion 463
 v-node 240
 void Datentyp 116
 volatile Schlüsselwort 118, 412
 Voll-Pufferung 200
 V-Operation 779
 voreingestellte Pufferung 201
 vprintf Funktion 193
 vsprintf Funktion 194
 VT_ACKAQQ Konstante 990
 VT_ACTIVATE Konstante 988
 VT_DISALLOCATE Konstante 988
 VT_GETMODE Konstante 987
 VT_GETSTATE Konstante 988
 VT_KIOCSOUND Konstante 990

VT_LOCKSWITCH Konstante 993
vt_mode Struktur 986
VT_OPENQRY Konstante 987
VT_RELDISP Konstante 990
VT_SETMODE Konstante 990
vt_state Struktur 986
VT_UNLOCKSWITCH Konstante 993
VT_WAITACTIVE Konstante 988

W

W_OK Konstante 277
wait Funktion 504
wait_queue Struktur 70
wait3 Funktion 515
wait4 Funktion 515
waitpid Funktion 504
wake_up Funktion 72
wake_up_interruptible Funktion 72
wake_up_process Funktion 72
WARTE_AUF_KIND Funktion 517, 645, 729
WARTE_AUF_PAPA Funktion 517, 645, 729
wc Kommando 217
wchar_t Datentyp 52, 105, 141
WCONTINUED Konstante 509
WCOREDUMP Makro 505
wcstombs Funktion 152
wctomb Funktion 152
who Kommando 380
WIFEXITED Makro 505
WIFEXITSTATUS Makro 505
WIFSIGNALED Makro 505
WIFSTOPPED Makro 505

winsize Struktur 919
WNOHANG Konstante 508
WNOWAIT Konstante 509
Working-Directory 13, 315
write Funktion 231, 347
write_inode Funktion 337
write_super Funktion 338
writev Funktion 695
WSTOPSIG Makro 505
WTERMSIG Makro 505
wtmp Datei 380
WUNTRACED Konstante 508

X

X/Open 35
X_OK Konstante 277
xchg Funktion 81
XOPEN_VERSION Konstante 44
XPG 35
xtime Variable 80
xxgdb 1062

Z

Zeilen-Pufferung 200
Zeitangaben 385
Zeiten einer Datei 307
Zeiten in Unix 32
Zeitmarken (make) 1105
Zeitzone 396
Zombieprozeß 503
Zugriffserlaubnis prüfen 276
Zugriffsrechte 224, 226, 267, 312