

POSIX-PROGRAMMIERUNG UNTER UNIX

Standardkonforme Programme für UNIX-Plattformen entwickeln

Steve Graegert



für Harri & Rita

Oktober 2004, Revision 0.31b

Copyright Dieses Werk ist unter einem *Creative Commons Namensnennung-Keine kommerzielle Nutzung-Keine Bearbeitung 3.0 Deutschland* Lizenzvertrag lizenziert. Um die Lizenz anzusehen, gehen Sie bitte zu <http://creativecommons.org/licenses/by-nc-nd/3.0/de/> oder schicken Sie einen Brief an Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Jedoch dürfen Sie das Werk bzw. den Inhalt vervielfältigen, verbreiten und öffentlich zugänglich machen. Dabei sind folgende Bedingungen zu beachten:

Namensnennung

Sie müssen den Namen des Autors/Rechteinhabers in der von ihm festgelegten Weise nennen.

Keine kommerzielle Nutzung

Dieses Werk bzw. dieser Inhalt darf nicht für kommerzielle Zwecke verwendet werden.

Keine Bearbeitung

Dieses Werk bzw. dieser Inhalt darf nicht bearbeitet, abgewandelt oder in anderer Weise verändert werden.

Wobei gilt:

Verzichtserklärung

Jede der vorgenannten Bedingungen kann aufgehoben werden, sofern Sie die ausdrückliche Einwilligung des Rechteinhabers dazu erhalten.

Sonstige Rechte

Die Lizenz hat keinerlei Einfluss auf die folgenden Rechte:

1. Die gesetzlichen Schranken des Urheberrechts und sonstigen Befugnisse zur privaten Nutzung;
2. Das Urheberpersönlichkeitsrecht des Rechteinhabers;
3. Rechte anderer Personen, entweder am Lizenzgegenstand selber oder bezüglich seiner Verwendung, zum Beispiel Persönlichkeitsrechte abgebildeter Personen.

Hinweis

Im Falle einer Verbreitung müssen Sie anderen alle Lizenzbedingungen mitteilen, die für dieses Werk gelten. Am einfachsten ist es, an entsprechender Stelle einen Link auf diese Seite einzubinden.

Inhaltsverzeichnis

I Grundlagen	1
1 UNIX-Programmierung	3
1.1 Einführung	3
1.2 UNIX-Konzepte	4
1.2.1 Prozesse und Prozessverwaltung	4
1.2.2 Dateien und Dateibehandlung	6
1.2.3 Systemaufrufe (<i>system calls</i>)	7
1.3 UNIX-Programme entwickeln	8
1.3.1 Programme übersetzen: Überblick	9
1.3.1.1 C-Programme übersetzen	9
1.3.1.2 C-Compiler verwenden	10
1.3.1.3 Standardkonforme Programmübersetzung	11
1.3.2 Programmdesign und Fehlerbehandlung	12
1.3.3 Sicherheitsaspekte in der Programmierung mit C	15
1.3.3.1 Der Buffer-Overflow	15
1.3.3.2 Buffer-Overflow und Sicherheit	17
1.3.3.3 Kleine Checkliste für sicheres Programmieren	17
2 UNIX-Standards	19
2.1 Was ist POSIX?	20
2.2 POSIX-Dokumente	20
2.3 Die Single UNIX Specification	21
2.3.1 Solaris 9 und die Single UNIX Specification	21
2.4 Unzulänglichkeiten von POSIX/SUSv3	22
2.5 Die POSIX-Umgebung	22
2.5.1 Feature Test Macros	23
2.5.1.1 Das <code>_POSIX_SOURCE</code> Feature Test Macro	23
2.5.1.2 Das <code>_POSIX_C_SOURCE</code> Feature Test Macro	23
2.5.1.3 Das <code>_XOPEN_SOURCE</code> Feature Test Macro	24
2.5.2 Namensräume	24
2.5.3 Fehlerkennziffern	25
2.5.4 POSIX Datentypen	31
2.6 Ermittlung von Implementierungsdetails	34

3 Dateibehandlung	39
3.1 Dateien öffnen und erstellen	40
3.1.1 Die Funktion <code>open</code>	40
3.1.2 Die Funktion <code>creat</code>	41
3.1.3 Die Funktion <code>close</code>	42
3.2 Dateien lesen und schreiben	42
3.2.1 Die Funktion <code>lseek</code>	42
3.2.2 Die Funktion <code>read</code>	44
3.2.3 Die Funktion <code>write</code>	46
3.3 File Descriptors verwalten	50
3.3.1 Die Funktionen <code>dup</code> und <code>dup2</code>	50
3.3.2 Die Funktion <code>fcntl</code>	52
4 Arbeiten mit Dateien und Verzeichnissen	55
4.1 Informationen über Dateien und Verzeichnisse	55
4.1.1 Die Funktion <code>stat</code>	56
4.1.2 Die Funktion <code>utime</code>	59
4.2 Dateizugriffsrechte	61
4.2.1 Die Funktion <code>access</code>	62
4.2.2 Die Funktion <code>umask</code>	63
4.2.2.1 Warum brauchen wir <code>umask</code> ?	63
4.2.2.2 Gültigkeitsbereich von <code>umask</code>	64
4.2.3 Die <code>chmod</code> -Funktionen	65
4.2.4 Besitzrechte ändern	67
4.3 Dateien und Verzeichnisse verwalten	67
4.3.1 Dateisystemkunde	68
4.3.1.1 i-nodes in UFS	68
4.3.2 Hard Links erstellen und entfernen	69
4.3.3 Dateien entfernen	73
4.3.4 Dateien umbenennen	73
4.3.5 Die Funktionen <code>symlink</code> und <code>readlink</code>	73
4.3.6 Die Funktionen <code>mkdir</code> und <code>rmdir</code>	75
4.4 Verzeichnisfunktionen	75
4.5 Im Dateisystem navigieren	79
4.6 Temporäre Dateien	81
4.6.1 Temporäre Dateien erzeugen	82
4.7 Device Special Files	84

5 Die Standard E/A Bibliothek (stdio)	87
5.1 Einführung	87
5.1.1 Standard E/A-Streams	87
5.1.2 File Descriptors und Standard E/A-Streams	88
5.2 Eingaben, Ausgaben und Buffering	89
5.3 Streams öffnen	93
5.4 Stream-Operationen	96
5.4.1 Zeichenweise Verarbeitung	96
5.4.2 Zeilenweise Verarbeitung	98
5.4.3 Binäre E/A-Operationen	99
5.5 Positionsindikatoren von Streams steuern	103
5.6 Formatierte E/A-Operationen	104
5.6.1 Die Funktionen <code>scanf</code> , <code>fscanf</code> und <code>sscanf</code>	107
5.6.2 Format- und Typspezifizierer	109
5.7 Von langsamen und schnellen E/A-Operationen	110
5.8 Zugriffe synchronisieren	111
5.8.1 Locking mit einer Lock-Datei	112
5.8.2 Record Locking mit <code>fcntl(2)</code>	115
5.8.3 Kooperatives und vorgeschriebenes Locking	118
6 Systeminformationen und Systemdateien	123
6.1 Benutzer- und Gruppeninformationen	123
6.1.1 Die Benutzerdatenbank <code>/etc/passwd</code>	124
6.1.2 Die Gruppendatei <code>/etc/group</code>	126
6.1.3 Zusätzliche Gruppen-IDs von Prozessen	128
6.2 Netzerkinformationen	129
6.2.1 <code>/etc/hosts</code>	130
6.2.2 <code>/etc/networks</code>	132
6.2.3 <code>/etc/protocols</code>	135
6.2.4 <code>/etc/services</code>	136
6.3 Accounting	138
6.3.1 Der traditionelle Ansatz	138
6.3.2 Der standardisierte Ansatz	140
6.4 Systeminformationen und Systemzeit	142

7 Prozessverwaltung	147
7.1 Prozessbezeichner (PID)	149
7.2 Prozesse erzeugen	150
7.2.1 Die Funktion <code>fork</code>	150
7.2.2 Die Funktion <code>vfork</code>	154
7.2.3 Die Funktionsfamilie <code>exec</code>	156
7.2.4 Die Funktion <code>system</code>	164
7.3 Prozesse beenden	168
7.3.1 Die Funktionsfamilie <code>exit</code>	168
7.3.2 Die Funktionen <code>wait</code> und <code>waitpid</code>	169
7.4 Benutzer- und Gruppen-IDs von Prozessen ändern	173
7.5 Prozessgruppen	174
7.6 Sitzungen	175
7.7 Terminals	176
7.8 Benutzeridentifizierung	177
7.9 Laufzeitmessungen	178
7.10 Hintergrundprozesse und Daemons	179
7.10.1 Hintergrundprozesse	180
7.10.2 Die Prozessumgebung	182
7.11 Kommandozeilenparameter verarbeiten	184
7.11.1 Konventionen	184
7.11.2 Parameter mit <code>getopt</code> verarbeiten	185
7.11.2.1 Definieren der Optionen und Argumente	186
7.11.2.2 Extraktion der Optionen und Argumente	187
7.11.3 Subparameter mit <code>getsubopt</code> verarbeiten	189
7.11.4 Die GNU-Erweiterung <code>getopt_long</code>	193
8 Signalbehandlung	197
8.1 Die Bedeutung der Signale	198
8.2 Systemaufrufe und Signale	200
8.3 Unzuverlässige Signalbehandlung	200
8.4 Signale senden	206
8.4.1 Die Funktion <code>raise</code>	206
8.4.2 Die Funktion <code>kill</code>	207
8.4.3 Die Funktion <code>alarm</code>	209
8.4.4 Die Funktion <code>abort</code>	210
8.5 Auf Signale warten	212
8.5.1 Die Funktion <code>pause</code>	212
8.5.2 Die Funktion <code>sigsuspend</code>	213

8.6	Arbeiten mit Signalsätzen	214
8.6.1	Die Funktion <code>sigset</code>	215
8.6.2	Das <code>sigset</code> -Interface	216
8.6.3	Signalsätze bearbeiten	219
8.6.4	Die Funktion <code>sigprocmask</code>	221
8.7	Zuverlässige Signalverarbeitung	224
8.7.1	Die Funktion <code>sigaction</code>	224
8.7.2	Die Funktion <code>sigsuspend</code>	228
9	Das Terminal als Schnittstelle	235
9.1	Charakteristika	235
9.1.1	Eingabeverarbeitung	235
9.1.2	Der kanonische Eingabemodus	236
9.1.3	Der nicht-kanonische Eingabemodus	237
9.1.4	Sonderzeichen	239
9.2	Die Struktur <code>termios</code>	240
9.2.1	Eingabemodi	242
9.2.2	Ausgabemodi	242
9.2.3	Kontrollmodi	243
9.2.4	Lokale Modi	243
9.2.5	Beschreibung der Flags der <code>termios</code> -Struktur	243
9.2.6	Eingabezeichen	247
9.3	Die Funktionen <code>tcgetattr</code> und <code>tcsetattr</code>	247
9.4	Einstellung der Baudraten	251
9.5	Die Funktion <code>ctermid</code>	253
9.6	Die Funktionen <code>ttyname</code> und <code>isatty</code>	254
10	Interprozesskommunikation - IPC	255
10.1	Anonyme Pipes	255
10.2	Benannte Pipes (FIFOs)	263
10.2.1	Benannte Pipes erzeugen	263
10.2.1.1	Benannte Pipes über die Kommandozeile erstellen	263
10.2.1.2	Benannte Pipes über einen Systemaufruf erstellen	264
10.2.2	Benannte Pipes öffnen	264
10.2.3	Von FIFOs lesen und in FIFOs schreiben	264
10.2.4	Einrichtung einer Vollduplexkommunikation	265
10.2.5	Vor- und Nachteile benannter Pipes	265
10.2.6	Beispiel einer Halbduplexverbindung	266
10.2.7	Beispiel einer Vollduplexverbindung	267
10.3	Shared Memory	270

10.3.1	Shared Memory Segements anfordern	270
10.3.2	Shared Memory Segments und der Prozessadressraum	271
10.3.3	Shared Memory Segements verwalten	274
10.4	Message Queues	276
10.4.1	System V Message Queues	277
10.4.1.1	Von Bezeichnern und Schlüsseln	277
10.4.1.2	Die IPC-Struktur	278
10.4.1.3	Message Queues	278
10.4.1.4	Arbeiten mit Message Queues	279
10.4.2	POSIX Message Queues	285
10.4.2.1	Message Queues erzeugen	286
10.4.2.2	Nachrichten absetzen	287
10.4.2.3	Nachrichten empfangen	288
10.4.2.4	Warteschlange konfigurieren	288
10.4.2.5	Beispielprogramme	289
10.5	POSIX-Semaphores	294
10.5.1	Kritische Sektionen	295
10.5.2	Unbenannte (anonyme) Semaphores	297
10.5.3	Benannte Semaphores	298
10.5.4	Semaphores einsetzen	299
10.5.5	Semaphore Sets	302
11	POSIX-Threads	311
11.1	Theoretische Grundlagen	311
11.1.1	Thread-Implementierungen	313
11.1.1.1	Kernel-Level Threads	313
11.1.1.2	User-Level Threads	313
11.2	Die POSIX-Threads Bibliothek	314
11.2.1	Der Lebenszyklus von Threads	314
11.2.2	Scheduling	325
11.2.2.1	Thread-Scheduling	326
11.2.2.2	Context Switching	327
11.2.3	Thread-Synchronisierung	328
11.2.3.1	Mutual Exclusion Locks (<i>Mutexes</i>)	329
11.2.3.2	Semaphores	332
11.2.3.3	Condition Variables	336
11.2.3.4	Lese- und Schreibsperren	339

II Netzwerkprogrammierung 347

12 Der TCP/IP-Protokollstapel	349
12.1 Code-Konventionen	351
12.2 Referenzmodelle: OSI und TCP/IP	352
12.3 TCP, UDP und SCTP - Die Transportschicht	353
12.3.1 User Datagram Protocol (UDP)	353
12.3.2 Transmission Control Protocol (TCP)	354
12.3.3 Das Stream Control Transmission Protocol (SCTP)	359
12.3.3.1 Verbindungen auf- und abbauen	360
12.3.4 Portnummern und Sockets	362
13 Die BSD-Socket API	363
13.1 Datenstrukturen	363
13.1.1 Standard-Adress-Struktur	363
13.1.2 IPv4 und IPv6	364
13.1.3 Host Byte Order und Network Byte Order	365
13.2 Grundlagen	367
13.2.1 Das Transmission Control Protocol	367
13.2.1.1 Einen Socket erstellen (<code>socket</code>)	368
13.2.1.2 Den Socket mit einer lokalen Adresse verbinden (<code>bind</code>)	369
13.2.1.3 Ausstehende Verbindungen abholen (<code>listen</code>)	370
13.2.1.4 Etablierte Verbindungen akzeptieren (<code>accept</code>)	371
13.2.1.5 Clients stellen Verbindungen mit <code>connect</code> her	372
13.2.2 TCP-Server und Clients entwickeln	372
13.2.2.1 Ein Daytime-Server	373
13.2.2.2 Ein Daytime-Client	374
13.2.2.3 Ein Echo-Server	376
13.2.2.4 Mehrere Clients gleichzeitig verarbeiten	378
13.2.3 Das User Datagram Protocol	382
13.2.4 UDP-Server und Clients entwickeln	382
13.2.4.1 Der UDP Echo-Server	384
13.2.4.2 Der UDP Echo-Client	385
13.3 Asynchrone Sockets	388
13.3.1 <code>select</code> verwenden	389
13.3.2 <code>pselect</code> verwenden	395
13.3.3 <code>poll(2)</code> verwenden	398

14 Fortgeschrittene Socket-Programmierung	403
14.1 Alternative und spezielle E/A-Funktionen	403
14.1.1 <code>recv</code> und <code>send</code> verwenden	404
14.1.2 <code>readv</code> und <code>writenv</code> verwenden	405
14.1.3 <code>recvmsg</code> und <code>sendmsg</code> verwenden	406
14.1.3.1 Kontrollinformationen verarbeiten	408
14.1.3.2 Das <code>CMSG_LEN</code> -Makro	412
14.1.3.3 Das <code>CMSG_SPACE</code> -Makro	412
14.1.3.4 Das <code>CMSG_DATA</code> -Makro	412
14.1.3.5 Das <code>CMSG_ALIGN</code> -Makro	413
14.1.3.6 Das <code>CMSG_FIRSTHDR</code> -Makro	413
14.1.3.7 Das <code>CMSG_NXTHDR</code> -Makro	413
14.2 Signalgesteuertes I/O	414
14.3 UNIX Domain Sockets	415
14.3.1 Grundlagen	415
14.3.1.1 UNIX Domain Sockets erstellen	415
14.3.1.2 Stream-Sockets in der UNIX Domain	417
14.3.1.3 Datagram-Sockets in der UNIX Domain	418
14.3.2 Die Funktion <code>socketpair</code>	420
14.4 Broadcasting und Multicasting	421
14.4.1 Wie funktioniert Broadcasting	422
14.4.2 Broadcasting Clients entwickeln	423
14.4.2.1 Probleme mit <code>alarm</code> und <code>recvfrom</code>	425
III UNIX als Entwicklungsumgebung	429
15 Alles an Board	431
15.1 Überblick	431
15.2 Der Compiler	432
15.2.1 Wie funktioniert ein Compiler?	432
15.2.2 Bezug und Installation der GCC	433
15.2.2.1 GCC-Quellen installieren	433
15.2.2.2 Vorkompilierte GCC-Installation	434
15.2.2.3 Test der Installation	434
15.3 GNU <code>make</code> und Makefiles	435
15.3.1 Überblick	435
15.3.2 Verwendung von <code>make</code> und Makefiles	436
15.4 GNU <code>automake</code> und <code>autoconf</code>	437
15.4.1 <code>automake</code> und <code>autoconf</code> im GNU Build-System	437

16 Der GNU C Compiler	439
17 Makefiles entwickeln	441
18 Debugging mit gdb	443
A POSIX-Konstanten für Optionen und Limits	445
A.1 Laufzeit-Limits und -Werte (<code>sysconf</code>)	445
A.2 Laufzeit-Limits und -Werte (<code>pathconf</code>)	447
B Socketoptionen	449
B.1 Socketoptionen für <code>SOL_SOCKET</code>	451
B.2 Socketoptionen für <code>IPPROTO_IP</code>	453
B.3 Socketoptionen für <code>IPPROTO_ICMPV6</code>	456
B.4 Socketoptionen für <code>IPPROTO_IPV6</code>	457
B.5 Socketoptionen für <code>IPPROTO_IPV6</code> und <code>IPPROTO_IP</code>	460
B.6 Socketoptionen für <code>IPPROTO_TCP</code>	462
B.7 Socketoptionen für <code>IPPROTO_SCTP</code>	463
C Fehlerbehandlung	473
D Beschreibung der Hilfsfunktionen	475
D.1 Der Header <code><header.h></code>	475
D.2 Die Wrapper-Funktionen in <code>libsys.c</code> und <code>libsock.c</code>	479
D.2.1 Wrapper für UNIX-Systemaufrufe	479
D.2.2 Wrapper für Socketfunktionen	483
D.2.3 Wrapper für <code>pthread</code> -Funktionen	487
D.3 Verschiedene Source Codes	488

Abbildungsverzeichnis

1.1	Vereinfachte schematische Darstellung der Systemarchitectur von UNIX	3
1.2	Eine Prozesskette, die durch den Aufruf von <code>fork</code> erzeugt werden kann.	4
1.3	Zusammenhang zwischen Funktionsaufrufen und System Calls.	8
1.4	Stack-Layout nach Aufruf der Funktion <code>get_pass</code>	17
3.1	Beziehungen zwischen File Descriptor Table, System File Table und der i-node Table.	39
3.2	So beeinflussen positive und negative Offsets die Position in der Datei.	43
4.1	UFS-Blocklayout	69
4.2	Festplatte, Partitionen und Dateisysteme.	71
4.3	Verzeichniseinträge der i-nodes.	71
5.1	Mehrere Prozesse sperren jeweils einen eigenen Bereich der Datei.	112
6.1	Zusammenhang zwischen den <code>time</code> -Funktionen.	144
7.1	Zustandsdiagramm für Prozesse.	147
7.2	Verallgemeinertes Prozessabbild typischer UNIX-Prozesse.	148
7.3	Typischer Prozessbaum eines UNIX-Systems.	149
7.4	Eine Prozesskette (a) und ein Prozessbaum (b), erzeugt mit <code>fork</code>	154
7.5	Layout der Process Images und File Images	158
7.6	Prozessgruppen und Sitzungen: Die Verkettung <code>process1 process2</code> führt zu einer neuen Prozessgruppe innerhalb der Sitzung.	175
8.1	Zeitlicher Zusammenhang zwischen Signalbehandlung und Wiederherstellung der Signaldisposition.	206
8.2	Einfluss von <code>sigemptyset(3)</code> , <code>sigfillset(3)</code> und <code>sigdelset(3)</code> auf die Signalmaske.	214
9.1	Warteschlangen für Terminal E/A	236
9.2	Rolle der Line Discipline	239
10.1	Einfache Pipe: Datenfluß vom Parent zum Child.	257
10.2	Kommunikation zwischen zwei Children.	258
10.3	Halbduplex- und Vollduplexkommunikation mit FIFOs.	265
10.4	Vollduplexkommunikation mit FIFOs.	268

11.1 Beziehung zwischen Prozessen und Threads.	312
11.2 Single-Threading und Multi-Threading im Vergleich.	313
11.3 Beziehung zwischen Controller und Worker Thread.	316
11.4 Thread Cancellation anhand zweier Threads.	320
11.5 Grafische Darstellung von Beispiel 11.2.	322
11.6 Drei unterschiedliche Scheduling-Techniken.	326
12.1 Client-Server-Kommunikation: zwei Clients, ein Server.	349
12.2 Client und Server kommunizieren über ein Protokoll, das auf andere Protokolle aufsetzt. .	350
12.3 Das OSI-Modell und die IP Suite im Vergleich.	353
12.4 UDP-Datagram gekapselt in einem IP Datagram.	353
12.5 UDP-Header.	354
12.6 TCP-Paket gekapselt in einem IP Datagram.	354
12.7 TCP-Header.	355
12.8 Three-Way Handshake des TCP-Protokolls	357
12.9 Verbindungsabbau bei TCP.	358
12.10 Zustandsdiagramm von TCP.	358
12.11 SCTP-Datagram gekapselt in einem IP Datagram.	359
12.12 Vergleich zwischen TCP und SCTP.	359
12.13 SCTP-Paketformat.	360
12.14 SCTP Four-Way-Handshake	361
12.15 Abbau von SCTP-Verbindungen	361
12.16 SCTP-Zustandsdiagramm	362
13.1 Unterschied zwischen Little Endian und Big Endian.	366
13.2 Zwei Queues eines Sockets	370
13.3 Kommunikation zwischen Daytime-Client und -Server.	373
13.4 Kommunikation zwischen Echo-Client und Echo-Server.	376
13.5 udpclientserver	384
13.6 I/O Multiplexing mit <code>select(2)</code>	389
13.7 Organisation von File Descriptor Sets	390
14.1 Kapselung der cmsghdr-Struktur.	409
14.2 Unicasting und Broadcasting im Subnetz 192.168.1/24	423
15.1 Zusammenhänge zwischen automake und autoconf	438

Tabellenverzeichnis

1.1	UNIX-Standards und Makros	12
1.2	Sprachstandards und Makros	12
2.1	Standards in SunOS/Solaris	21
2.2	POSIX Standards und Feature Tests	23
3.1	Flags für den Zugriffsmodus.	40
3.2	Flags für den E/A-Modus.	41
3.3	Zulässige Werte für den Parameter <i>whence</i> von <code>lseek</code>	43
4.1	Flags zur Angabe der Dateizugriffsrechte.	61
4.2	Auflistung der <i>mode</i> -Bits für die Funktion <code>access</code>	62
5.1	Mögliche Werte des Parameters <i>mode</i> der Funktionen <code>fopen(3)</code> oder <code>fdopen(3)</code>	94
6.1	Auflistung der durch POSIX vorgeschriebenen Felder in <code>/etc/passwd</code>	124
6.2	Durch POSIX vorgeschriebene Felder der Gruppendaten <code>/etc/group</code>	126
7.1	Liste ausgewählter Umgebungsvariablen.	183
7.2	Übersicht der von <code> getopt(3)</code> verwendeten externen Variablen.	186
8.1	Liste aller Signale der wichtigsten UNIX-Derivate.	199
8.2	Liste aller sicheren POSIX.1-Funktionen	201
9.1	Flags der <code>termios</code> -Struktur	241
9.2	Baudraten und ihre Symbolischen Konstanten	243
9.3	Spezielle Eingabezeichen	248
10.1	IPC-Techniken in der UNIX-Welt.	256
10.2	Zugriffsrechte für SysV IPC.	279
11.1	POSIX-Funktionen zur Verwaltung und Anwendungen von RWLocks.	341
12.1	Von der IANA vergebene Portnummern.	362
14.1	Mögliche Anwendungen von Kontrollinformationen	410

14.2 Mögliche Anwendungen von Kontrollinformationen	421
B.1 Übersicht über die Socketoptionen und -Schichten. Erläuterung: S = Wert schreiben, L = Wert lesen, F = ist die Option ein Schalter?	450
B.2 ICMPv6 Type- und Code-Fields.	457

Listings

1.1	xcode/sample1.c - Einfaches Beispielprogramm mit rudimentärer Fehlerbehandlung	12
1.2	xcode/sample2.c - Verbessertes Beispielprogramm mit flexibler Fehlerbehandlung	13
2.1	xcode posix-test.c - Das POSIX-Feature Test Makro	23
2.2	xcode/printerrors.c - Ausgabe der Fehlerkennziffern und der passenden Meldungen.	25
2.3	xcode/pathconf.c - Anwendung von pathconf.	35
2.4	xcode/sysconf.c - Anwendung von sysconf.	36
3.1	xcode/lseek.c - Anwendung von lseek(2).	44
3.2	xcode/read.c - Anwendung von read(2).	45
3.3	xcode/cp.c - Lesen und schreiben von Dateien.	48
3.4	xcode/cp2.c - Bessere Implementierung des cp-Befehls.	49
3.5	xcode/dup2.c - Verwendung von dup2(2).	51
4.1	xcode/stat.c - Verwendung von stat.	57
4.2	xcode/paranoid-open.c - Sicheres Öffnen von Datei mit fstat und lstat.	58
4.3	xcode/utime.c - Einsatz von utime.	60
4.4	xcode/permissions.c - Anwendung der Funktion access.	63
4.5	xcode/umask.c - So wird umask verwendet.	65
4.6	xcode/link.c - Hard Links erzeugen.	70
4.7	xcode/unlink.c - Hard Links mit unlink entfernen.	72
4.8	xcode/ls.c - Einfache Implementierung von ls(1)	78
4.9	xcode/filesearch.c - Zeigt nur bestimmte Dateien im aktuellen Verzeichnis an.	78
4.10	xcode/changedir.c - Testprogramm für changedir	80
4.11	xcode/tmpnam.c - Einsatz von tmpnam.	82
4.12	xcode/showdev.c - Umgang mit Device Special Files.	85
5.1	xcode/setbuf.c - Aufruf von setbuf(3).	91
5.2	xcode/setbuf-race.c - Race Condition im Zusammenhang mit I/O-Buffering.	91
5.3	xcode/setvbuf.c - Aufruf von setvbuf(3).	92
5.4	xcode/fopen.c - Text einlesen und auf stdout ausgeben.	94
5.5	xcode/fdopen.c - Mit fdopen(3) eine Pipe erstellen.	95
5.6	xcode/inout.c - Eingaben von stdin nach stdout kopieren.	99
5.7	xcode/freadwrite.c - Binärdaten mit fread(3) und fwrite(2) lesen und schreiben.	101
5.8	xcode/sccanf.c - Mit scanf(3) formatierte Eingaben verarbeiten.	107

5.9	xcode/fscanf.c - Mit fscanf(3) formatierte Eingaben verarbeiten.	108
5.10	xcode/sscanf.c - Mit sscanf(3) formatierte Eingaben verarbeiten.	109
5.11	xcode/io-write.c - Schreiben großer Daten nach stdout.	111
5.12	xcode/file-locking.c - Locking mit Hilfe einer Lockdatei.	112
5.13	xcode/locksample.c - Einfache Anwendung des Record Locking.	116
5.14	xcode/lockdetect.c - Herausfinden, ob vorgeschriebenes Locking unterstützt wird.	119
6.1	xcode/getpwnam.c - Benutzerinformationen mit getpwnam(3) und getpwuid(3) abfragen.	125
6.2	xcode/getpwent.c - Benutzerinformationen mit getpwent(3) auflisten.	126
6.3	xcode/getgrnam.c - Gruppeninformationen mit getgrnam(3) auflisten.	127
6.4	xcode/getgroups.c - Gruppen-IDs mit getgroups auflisten.	129
6.5	xcode/gethostbyany.c - Hostinformationen mit gethostbyaddr(3) und gethostbyname(3) abfragen.	131
6.6	xcode/getnetent.c - Netzwerkinformationen aller Netze des lokalen Systems ausgeben.	133
6.7	xcode/getnetbyany.c - Netzwerkinformationen eines spezifischen Netzes des lokalen Systems ausgeben.	134
6.8	xcode/getprotoent.c - Informationen aller Protokolle von /etc/protocols ausgeben.	136
6.9	xcode/getservent.c - Informationen aller Dienst von /etc/services ausgeben.	138
6.10	xcode/utmp.c - Implementierung des who(1)-Kommandos mit Hilfe von getutent(3).	140
7.1	xcode/fork.c - Einfache Anwendung von fork(2) mit execvp.	151
7.2	xcode/process-chain1.c - Erzeugen einer Prozesskette.	153
7.3	xcode/process-chain2.c - Erzeugen einer Prozesskette.	154
7.4	xcode/vfork.c - Effiziente Erzeugung von Childs mit vfork.	155
7.5	xcode/process_image_c.c - Beispielprogramm für die Betrachtung von Process Images.	157
7.6	xcode/process_image_asm.c - Das Process Image in Assembler.	158
7.7	xcode/execle.c - Die Funktionen execle(3) und execlp(3) im Einsatz.	162
7.8	xcode/simpleshell.c - Einfache Shell mit Hilfe von fork, exec(3) und wait.	163
7.9	xcode/systemimpl2.c - Beispielimplementierung einer POSIX-konformen system(2)-Funktion	164
7.10	xcode/systemimpl.c - Beispielimplementierung einer POSIX-konformen system(2)-Funktion	165
7.11	xcode/system.c - Kommandos mit system(2) ausführen.	167
7.12	xcode/wait.c - Einsatz von wait(2)	170
7.13	xcode/wait2.c - Einsatz der WIF-Makros.	171
7.14	xcode/waitpid.c - So funktioniert WNOHANG mit waitpid(2)	172
7.15	xcode/getlogin.c - Informationen über den Benutzerprozess mit getlogin(2) abfragen.	177
7.16	xcode/times.c - Zeitmessungen mit times(2)	178
7.17	xcode/spinoff.c - Erstellung eines Hintergrundprozesses.	180
7.18	xcode/smallbiff.c - Beispielanwendung für einen Daemon	181
7.19	xcode/printenvimpl.c - Einfache Implementierung des printenv(7)-Kommandos.	183
7.20	xcode/getenv.c - Einsatz von getenv(3)	184

7.21	<code>xcode/getoptdemo.c</code> - Verwendung von <code>getopt(3)</code> zur Extraktion von drei Optionen und einem Argument.	187
7.22	<code>xcode/getopt_impl.c</code> - Beispielimplementierung von <code>getopt(3)</code>	188
7.23	<code>xcode/getopt_impl.h</code> - Header-Datei für <code>getopt(3)</code> und <code>getopt_long(3)</code>	189
7.24	<code>xcode/getsuboptdemo.c</code> - Verarbeiten von Unteroptionen mit <code>getsubopt(3)</code>	191
7.25	<code>xcode/getsubopt_impl.c</code> - Beispielimplementierung von <code>getsubopt(3)</code>	192
7.26	<code>xcode/getoptlongdemo.c</code> - Verwendung von <code>getopt_long</code>	195
8.1	<code>xcode/sigfunc.c</code> - Einfache Signalbehandlung mit <code>signal(2)</code>	203
8.2	<code>xcode/signal.c</code> - Unzuverlässige Signalbehandlung mit <code>signal(2)</code>	205
8.3	<code>xcode/raise.c</code> - Einsatz von <code>raise(2)</code>	207
8.4	<code>xcode/kill.c</code> - Einsatz von <code>kill(2)</code>	208
8.5	<code>xcode/alarm.c</code> - Einsatz von <code>alarm(3)</code>	210
8.6	<code>xcode/abortimpl.c</code> - POSIX-konforme Implementierung des <code>abort(3)</code> -Systemaufrufs.	211
8.7	<code>xcode/pause.c</code> - Beispielprogramm für <code>pause(3)</code>	212
8.8	<code>xcode/pauseimpl.c</code> - POSIX-konforme Implementierung der <code>pause(3)</code> -Funktion.	213
8.9	<code>xcode/sigset.c</code> - Sichere Signalauslieferung mit <code>sigset(3)</code>	216
8.10	<code>xcode/signalregion.c</code> - Beispiel zur Demonstration kritischer Regionen.	217
8.11	<code>xcode/sigprocmask.c</code> - Effektive Signalbehandlung mit <code>sigprocmask(2)</code> und Co.	222
8.12	<code>xcode/signalregion2.c</code> - Sichere Version des Beispiels 8.10 mit <code>sigaction(2)</code>	226
8.13	Beispiel für ein unerwünschtes Zeitfenster.	228
8.14	Bessere Variante: Auslöschung des Zeitfensters von Listing 8.13.	229
8.15	<code>xcode/sigsuspendimpl.c</code> - Beispielimplementierung für eine POSIX-konforme Variante von <code>sigsuspend</code>	229
8.16	<code>xcode/simpleshell2.c</code> - Erweitertes Beispiel 7.8 mit verbesserter Signalbehandlung.	231
9.1	<code>xcode/noncanon.c</code> - Umschalten in den nicht-kanonischen Modus.	237
9.2	<code>xcode/changekey.c</code> - Das INTR-Zeichen durch von STRG-C in STRG-G ändern.	249
9.3	<code>xcode/tcgetattr.c</code> - Flags in der <code>termios</code> -Struktur abfragen.	249
9.4	<code>xcode/term_echo.c</code> - Passwortabfrage bei deaktiviertem Echoing.	250
9.5	<code>xcode/term_speed.c</code> - Bestimmung der Ausgabegeschwindigkeit von <code>stdin</code>	252
9.6	<code>xcode/ctermidimpl.c</code> - Implementierung der POSIX-Funktion <code>ctermid</code>	253
9.7	<code>xcode/ctermid.c</code> - Anwendung von <code>ctermid(3)</code>	253
9.8	<code>xcode/isatty.c</code> - Anwendung von <code>isatty(3)</code> und <code>ttyname(3)</code>	254
10.1	<code>xcode/pipe-simplex.c</code> - Eltern schreiben gern ihren Kindern.	257
10.2	<code>xcode/pipe-who.c</code> - Geschwister schreiben sich untereinander auch gern.	258
10.3	<code>xcode/popen.c</code> - Anwendung von <code>popen(2)</code>	260
10.4	<code>xcode/popenimpl.c</code> - Beispielimplementierung der Funktionen <code>popen(2)</code> und <code>pclose(2)</code>	261
10.5	<code>xcode/echofilter.c</code> - Hauptprogramm für <code>myecho</code>	262
10.6	<code>xcode/myecho.c</code> - Hilfsprogramm <code>myecho</code>	262
10.7	<code>xcode/fifohdsrv.c</code> - Code des Servers.	266

10.8	xcode/fifohdcli.c - Code des Clients.	266
10.9	xcode/fifofdsrv.c - Code des Servers.	267
10.10	xcode/fifofdcli.c - Code des Clients.	269
10.11	xcode/shmserver.c - Server-Code. Erstellt ein Shared Memory Segment und füllt es mit Inhalt auf.	272
10.12	xcode/shmclient.c - Client-Code. Liest ein Shared Memory Segment aus.	273
10.13	xcode/shmctl.c - Anwendung von <code>shmctl(2)</code> .	275
10.14	xcode/mq1srv.c - Server-Code.	282
10.15	xcode/mq1srv.c - Client-Code.	283
10.16	xcode/mq1.h - Header für Server und Client.	285
10.17	xcode/mqrt_open.c - Code für <code>mqrt_open</code> .	289
10.18	xcode/mqrt_send.c - Code für <code>mqrt_send</code> .	291
10.19	xcode/mqrt_attr.c - Code für <code>mqrt_attr</code> .	292
10.20	xcode/mqrt_receive.c - Code für <code>mqrt_receive</code> .	293
10.21	xcode/critical_demo.c - Code für <code>critical_demo.c</code> .	295
10.22	xcode/sem_open.c - Beispielprogramm für den richtigen Umgang mit kritischen Sektionen.	301
10.23	xcode/semsetdemo1.c - Einfaches Demo zur Anwendung von Semaphore Sets.	303
10.24	xcode/semopdemo2.c - Beispielprogramm für die Anwendung von <code>semop(2)</code> .	308
11.1	xcode/thread_create.c - Threads erzeugen, auf sie warten und den Exit-Status abfragen.	316
11.2	xcode/threaded_demo.c - Threads erzeugen Threads erzeugen Threads.	320
11.3	xcode/pthread_mutex1.c - Anwendung von Mutexes in MT-Applikationen.	331
11.4	xcode/sem_pseudo1.c - Pseudocode für das Produzent/Konsumentproblem.	333
11.5	xcode/sem_philosophers.c - Lösung des Dining Philosophers Problems mit Hilfe von Threads und Semaphores.	334
11.6	xcode/pthread_cond1.c - Anwendung von Condition Variables.	337
11.7	xcode/pthread_rwlock2.c - Anwendung von Schreib-/Lesesperrren.	343
13.1	xcode/byteorder.c - Byteorder eines Systems herausfinden.	366
13.2	xcode/tcpdaytimesrv1.c - Ein einfacher Daytime-Server.	373
13.3	xcode/tcpdaytimecli1.c - Ein Daytime-Client.	375
13.4	xcode/tcpecho/tcpechosrv1.c - Der TCP Echo Server.	376
13.5	xcode/tcpecho/tcpechocli1.c - Der TCP Echo Client.	378
13.6	xcode/tcpecho/tcpechosrv2.c - Mehrere Clients mit Hilfe des <code>fork</code> -Systemaufrufs bedienen.	379
13.7	xcode/tcpecho/tcpechosrv3.c - Mehrere Clients mit Hilfe von Threads bedienen.	381
13.8	xcode/udpechosrv1.c - UDP-Version des Echo Servers.	385
13.9	xcode/udpechocli1.c - UDP-Version des Echo Clients.	385
13.10	xcode/udpechocli2.c - Ein UDP Echo Client mit <code>connect(2)</code> .	387
13.11	xcode/tcpecho/tcpechosrv4.c - TCP Echo Server mit <code>select(2)</code> .	392
13.12	xcode/tcpecho/tcpechocli2.c - TCP Echo Client mit <code>select(2)</code> .	394
13.13	xcode/pselect_impl.c - Einfache Implementierung von <code>pselect(2)</code>	397

13.14	xcode/poll.c - Einfaches Beispiel eines Clients mit poll(2).	400
14.1	xcode/readv.c - Anwendung von readv(2).	406
14.2	xcode/unix_domain/ud_server1.c - Ein einfacher Echo-Server mit UNIX Domain Sockets.	417
14.3	xcode/unix_domain/ud_client1.c - Ein einfacher Echo-Client mit UNIX Domain Sockets.	418
14.4	xcode/unix_domain/ud_server2.c - SOCK_DGRAM Echo-Server mit UNIX Domain Sockets.	418
14.5	xcode/unix_domain/ud_client2.c - SOCK_DGRAM Echo-Client mit UNIX Domain Sockets.	419
14.6	xcode/unix_domain/ud_socketpair.c - Einfaches Beispiel für die Anwendung von socketpair(2).	420
14.7	xcode/bcast/bcastclifun1.c - Die process_request-Funktion arbeitet nun mit Broadcasting .	424
14.8	xcode/bcast/bcastclifun2.c - Die process_request-Funktion mit einem Timer .	424
14.9	xcode/bcast/bcastclifun3.c - In der Schleife Signale blockieren (falsch) .	426
14.10	xcode/bcast/bcastclifun4.c - sigsetjmp(2) und siglongjmp(2) verwenden .	427
B.1	src/showmss.c - MSS programmatisch abfragen.	462
D.1	xcode/lib/header.h - Der Header <header.h>.	475
D.2	xcode/lib/libsys.c - Die Bibliothek lib/libsys.c.	479
D.3	xcode/lib/libsock.c - Die Bibliothek lib/libsock.c.	483
D.4	xcode/lib/libthread.c - Die Bibliothek lib/libthread.c.	487
D.5	xcode/lib/create_argv.c - Die Funktion lib/create_argv.c.	488
D.6	xcode/lib/get_block_size.c - Die Funktion lib/get_block_size.c.	489
D.7	xcode/lib/lltostr.c - Die Funktionen lltostr und ulltostr.	489
D.8	xcode/lib/make_tcp_socket.c - Die Funktion lib/make_tcp_socket.c.	490
D.9	xcode/lib/print_exit_status.c - Die Funktion lib/print_exit_status.c.	491
D.10	xcode/lib/readline.c - Die Funktion lib/readline.c.	492
D.11	xcode/lib/share_fd.c - Die Funktionen recv_fd und send_fd.	492

Vorwort

If you would not be forgotten, as soon as you are rotten, either write things worth reading or do things worth the writing.

BENJAMIN FRANKLIN (1706 - 1790)

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.

WILLIAM STRUNK JR., ELEMENTS OF STYLE

Dieses Buch ist für EntwicklerInnen, die lernen möchten, wie POSIX-konforme Programme erstellt und Legacy-Code portiert werden kann. Obwohl POSIX grundsätzlich auf allen Betriebssystemen implementierbar ist und weite Verbreitung fand, beschränke ich mich auf die Entwicklung unter UNIX. Es ist nicht einfach, die Balance zwischen einer schnöden Referenz und einem Handbuch zu finden, soll es doch beides sein. Ich bin stets bemüht, Beispiele zu finden, die den alltäglichen Anforderungen an ProgrammiererInnen gerecht werden und den Sachverhalt angemessen beleuchten.

Allzu oft sehen wir uns gezwungen, neue Themenbereiche schnell zu erfassen und anzuwenden, obwohl wir über wenig Hintergrundwissen verfügen. In solchen Situationen müssen wir uns selbst damit befassen und Inhalte möglichst effektiv aufnehmen, oftmals mit dem aktuellen Problem als Ausgangspunkt. Obwohl technische Referenzen und Handbücher die notwendigen Informationen liefern, fehlt es oftmals an der Vermittlung der Motivation, der Hintergründe oder einer Erklärung der jeweiligen Technik oder Technologie. Doch gerade diese Dinge helfen uns, den Zusammenhang zu verstehen und zu erfassen, warum wir das tun, was wir gerade tun.

Als praktische Referenz gedacht, halte ich mich mit der Erläuterung theoretischer Belange zurück und orientiere mich an der Praxis. Bis auf den Abschnitt 2.5 (Seite 22) *Die POSIX-Umgebung* (der zugegebener Maßen tatsächlich etwas theoretisch ist) versuche ich stets Informationen, die in den jeweiligen Manpages zu finden sind, wegzulassen und stattdessen Hintergrundwissen zu vermitteln.

Die Standarddokumente sind traditionell sehr ausführlich und präzise. Das führt bei den Lesern, genauer gesagt den Entwicklern, nicht selten zu Mißverständnissen, obwohl das Gegenteil erreicht werden soll. So werden Formulierungen wie beispielweise *sollte* (engl. *should*) und *muß* (engl. *shall*) verwendet, die besonders im Zusammenspiel mit Verneinungen Kopfschmerzen verursachen. Ein Ziel dieses Buches ist unter anderem die Verbesserung der Verständlichkeit so daß Sie sich nicht mit solchen Details herumschlagen müssen.

Voraussetzungen für das uneingeschränkte Verständnis dieses Buches sind fundierte Kenntnisse über die Programmiersprache C und das Betriebssystem UNIX. Sie sollten in der Lage sein, sich innerhalb des Systems sicher bewegen zu können und Programme mit ihrem bevorzugten Compiler zu übersetzen.

Wenn ich auf einzelne Kommandos und Systemaufrufe verweise, verwende ich die übliche Notation, die das Kommando nennt und, in Klammern gesetzt, die passende Manpage-Sektion anzeigt, beispielsweise `fopen(2)`, welches den Systemaufruf `fopen` und die Manpage-Sektion 2 nennt. Des Weiteren möchte ich darauf hinweisen, dass alle Erwähnungen einer Funktion im Text nicht mit den dazugehörigen Klammern genannt werden, auch wenn ein oder mehr Parameter für diese Funktion erforderlich sind. So würde, um beim Beispiel `fopen` zu bleiben, die Funktion zwei zusätzlich Parameter erfordern, nämlich: `char *filename` und `char *mode`.

Für aktuelle Informationen besuchen Sie die Website des Buches: X. Dort können Sie auch alle Beispiele herunterladen.

Aufbau des Buches und Konventionen

Im Rahmen der Besprechung von Systemaufrufen und Bibliotheksfunktionen wird der *Prototyp* einer Funktion vorgestellt. Zusammen mit den erforderlichen Headern und einer Beschreibung der Funktionsparameter sowie der Rückgabewerte ergibt sich die *Synopsis* einer Funktion und wird immer folgendermaßen hervorgehoben:

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, /* ... */ );
```

Rückgabewerte: File Descriptor oder -1 bei Fehler.

path

Pfad zu der Datei, die wir öffnen möchten.

oflags

Status-Flags und Zugriffsmodi, die für das Öffnen der Datei angewendet werden sollen.

Auf diese Weise sind Sie in der Lage, die wichtigsten Informationen schnell zu erfassen.

Beispiele, die häufig in diesem Text auftauchen sind immer mit dem Dateinamen der Quelldatei markiert, so dass Sie das Makefile benutzen können um die Beispiele bequem zu übersetzen. Hat das Beispiel den Dateinamen `test.c` so können sie einfach

```
% make test
cc test.c -o bin/test
```

eingeben und die übersetzte Datei befindet sich anschließend im Unterverzeichnis `bin/`.

Hin und wieder werden Hintergründe besprochen, die nicht zwingend für das Verständnis der UNIX-Programmierung erforderlich sind, aber durchaus interessante Einblicke in sonst unsichtbare Zusammenhänge liefern. Solche *Backgrounder* erkennen sie immer an dem Kasten und der anderen Schriftart.

Das ist ein Backgrounder

Das Thema wird als Überschrift hervorgehoben und mit einer anderen Schrift dargestellt. Pragmatiker werden Backgrounder des öfteren überspringen während insbesondere Einsteiger diese eingestreuten Ergänzungen sehr nützlich finden werden.

Gliederung

Kapitel 1

Dieses Kapitel wird Ihnen einen kleinen Einstieg in die UNIX-Programmierumgebung geben. Die Beispiele in diesem Kapitel greifen Themenbereiche auf, die wir erst im Laufe des Buches kennenlernen werden, sind aber durchaus dazu geeignet, sich mit der UNIX-Umgebung und den notwendigen Werkzeugen vertraut zu machen. LeserInnen, die bereits Erfahrung im Umgang mit Übersetzern aufweisen oder nicht zum ersten Mal unter UNIX programmieren, können diese Sektion ohne weiteres überspringen.

Kapitel 2

Das erste Kapitel liefert einen kurzen Abriß über die Entstehung des POSIX-Standards und gibt Einblicke in die Entwicklungsumgebung, die dieser Standard erzeugt. Erfahrene Entwickler finden wertvolle Informationen über Namensräume, Fehlerkennziffern (`errno`) und POSIX-Datentypen.

Kapitel 3

In diesem Kapitel behandeln wir die wichtigsten Aufrufe zur Dateibehandlung wie etwa `open`, `creat`, `close`, `lseek`, etc.

Kapitel 4

In diesem Abschnitt besprechen wir den Umgang mit Verzeichnissen und Dateien. Beispielsweise können wir mit Hilfe der Struktur `stat` alle Eigenschaften einer Datei abfragen und sie durch Hilfsfunktionen modifizieren.

Kapitel 5

Jede ANSI C konforme Implementierung bringt ihre eigene *Standard I/O Library* mit. Somit treffen große Teile dieses Kapitels auch auf nicht-UNICES zu. Hauptaufgabe der Bibliothek ist die Kapselung komplexer und fehleranfälliger Vorgänge wie etwa Buffering und Speicherallokierung oder auch die richtige Auswahl von Bockgrößen usw. Entwickler können sich so auf ihre Hauptaufgaben konzentrieren und müssen sich nicht mit Details auseinandersetzen. Dieses Kapitel zeigt Ihnen, wie es funktioniert

Kapitel 6

Viele Dateien eines UNIX-Systems sind für den reibungslosen Betrieb unabdingbar. Sie bestimmen, ob wir mit anderen Systemen kommunizieren dürfen, welche Dienste in welcher Reihenfolge gestartet werden usw. In diesem Kapitel befassen wir uns mit dem Umgang mit diesen Dateien.

Kapitel 7

Die Prozessverwaltung ist eine der wichtigsten Aufgaben des UNIX-Kernels. In diesem Kapitel lernen wir mehr über die Arbeitsweise und die Mechanismen, die in die Prozessverwaltung involviert sind.

Kapitel 8

Während Geräte Hardware-Interrupts an den Kernel senden können, um Zeit für die Durchführung Ihrer Aufgaben anzufordern, können wir auf die gleiche Weise unsere Applikationen steuern. Wir tun dies allerdings mit Software-Interrupts, auch Signale (*signals*) genannt. Kapitel 8 zeigt, wie wir Signale richtig behandeln.

Kapitel 9

In diesem Kapitel beschäftigen wir uns mit dem UNIX-Terminal als allgemeine Schnittstelle. Der Begriff *Terminal* hat viele Bedeutungen. Zum einen bezeichnen wir damit Geräte, die über ein Netzwerk mit einem Rechner verbunden sind und selbst keine Rechenkapazitäten besitzen (sog. *dumb terminals*), zum anderen sind Kommandoprozessoren, mit deren Hilfe wir die Befehle eingeben, auch mit einem Terminal verbunden. Und das, obwohl wir vielleicht noch nicht einmal an ein Netzwerk angebunden sind. Kapitel 9 erklärt warum das so ist.

Kapitel 10

In diesem Abschnitt betrachten wir verschiedene Ansätze, die es uns erlauben, Informationen mit anderen Prozessen auszutauschen. Die wichtigsten sind: Pipes, FIFOs (auch als *benannte Pipes* bekannt), Semaphoren, Shared Memory und Sockets.

Kapitel 11

Parallelität ist in der Computertechnik einer der wichtigsten Techniken modernen Softwaredesigns. Es existieren unterschiedliche Ansätze, dieses Problem zu lösen, zum Beispiel Shared Memory oder Message Passing. Alternativ stehen uns auch Threads zur Verfügung. Dabei existieren mehrere Ausführungspfade in ein und demselben Adressraum eines Prozesses. In diesem Kapitel werden wir lernen wie Threads erzeugt und verwaltet werden und wie sie uns helfen, einfache Probleme effizient zu lösen.

Vereinbarungen

Der vorliegende Text enthält viele Fachbegriffe, Referenzen auf weiterführende Literatur, Quelltext und andere Merkmale. Um die Lesbarkeit zu erhöhen unterliegen diese Dinge gewissen Vereinbarungen in Form von Formatierungen, die es Ihnen erleichtern, den Text zu erfassen. Dabei wurde folgendermaßen vorgegangen:

Normaler Fließtext wird in der Schriftart Roman in einer Größe von 10 Punkten dargestellt:

Einfacher Text, Schriftart Roman, 10 Punkte

Hervorhebungen, wichtige Unterscheidungen und andere Verdeutlichungen sind fettgedruckt:

Wichtiger Hinweis, Schriftart Roman, 10 Punkte, fettgedruckt

Abkürzungen, die voll ausgeschrieben sind, Eigennamen und englische Pendants werden kursiv dargestellt:

Abkürzung (*ausgeschriebene Bezeichnung*) und Fachbegriff (*englisches Pendant*)

Quelltexte sind in Courier, einer Schriftart mit fester Zeichenbreite, in der Größe von 10 Punkten verfaßt. Beispiel:

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     printf("Hello World!\n"); /* this is a comment */
5     return(0);
6 }
```

Teile, die Bezug auf den Quelltext nehmen, wie etwa Variablennamen, Funktionen und Datentypen werden innerhalb des Fließtextes mit der Schriftart Courier in der Größe 10 Punkte formatiert. Beispiel:

Die Funktion `fopen(2)` erfordert zwei Parameter: `filename` und `mode`, beide vom Typ `char *`.

Die Zahl in Klammern gibt die Manpage-Sektion an, hier Sektion 2, die genaue Informationen über die Anwendung der Funktion enthält.

Einzelne Abweichungen sind unbeabsichtigt und auf Fehler im Layout zurückzuführen. Sie können mit dem Author Kontakt (<graegerts@gmail.com>) aufnehmen und ihre Anmerkungen übermitteln.

Teil I

Grundlagen

Kapitel 1

UNIX-Programmierung

*An ostentatious man will rather relate a blunder or
an absurdity he has committed, than be debarred
from talking of his own dear person.*

JOSEPH ADDISON (1672 - 1719)

Zu Beginn greifen wir ein paar Grundlagen im Umgang mit der UNIX-Programmierumgebung auf. Die Beispiele in diesem Kapitel beziehen sich zum Teil auf Themenbereiche, die wir erst im Laufe des Buches kennenlernen werden, eignen sich aber durchaus dazu, sich mit der UNIX-Umgebung und den notwendigen Werkzeugen vertraut zu machen. LeserInnen, die bereits Erfahrung im Umgang mit Übersetzern aufweisen oder nicht zum ersten mal unter UNIX programmieren, können diese Sektion ohne weiteres überspringen.

1.1 Einführung

Die Architektur nahezu aller UNIX-Systeme ist in drei Ebenen gegliedert: dem Kernel, der zusammen mit den Gerätetreibern und den Systembibliotheken, die Anwendungsprogramme und Systemprogramme bedient, einer Shell auf der zweiten Schicht, die als Benutzerschnittstelle funktioniert (hier sind auch die Anwendungsbibliotheken zu finden, auf die fast alle Anwendungsprogramme zurückgreifen) und der Anwendungsschicht, die eine Ausführungsumgebung für Anwendungsprogramme bereitstellt. Die wichtigsten Elemente des UNIX-Bestriebssystems sind in Abbildung 1.1 dargestellt. Die Pfeile zeigen die Kommunikationsrichtung der einzelnen Module an.

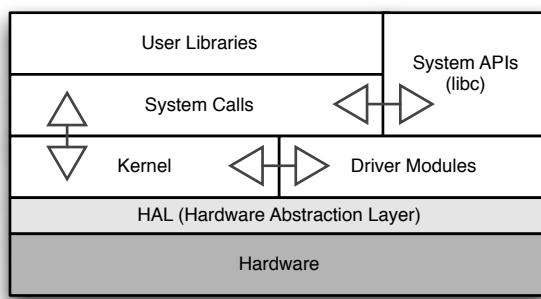


Abbildung 1.1: Vereinfachte schematische Darstellung der Systemarchitectur von UNIX.

Wenn wir es ganz genau nehmen, sind die Treiber nicht wirklich Bestandteil des Kernels, aber sehr eng mit ihm verwoben. Treiber kommunizieren mit dem Kernel über eine kleine, klar definierte Schnittstelle

und liefern grundlegende Dienste für beispielsweise Eingabegeräte oder Terminals. Auf der anderen Seite liefern Treiber wiederum wichtige Dienste für den Kernel, wie etwa Zugang zu wichtigen Geräten während der Startphase des Systems.

Jedes Betriebssystem bietet den Anwendungsprogrammen über sogenannte (*system calls*) Zugang zu den Systemdiensten. Jeder dieser Systemaufrufe erfüllt eine bestimmte Aufgabe, beispielsweise Ein- und Ausgaben oder Speicherverwaltung. Die Systemaufrufe des Kernels sind einer Bibliothek (in der Regel *libc*) gekapselt, um uns die Arbeit mit den Systemaufrufen zu erleichtern, denn die direkte Interaktion mit Systemaufrufen wäre weitaus aufwendiger und verringert die Portabilität erheblich.

Eine Shell dient zur Interaktion mit dem Betriebssystem. Hier setzen wir Kommandos ab oder betrachten die Ausgaben eines Programms. Bekannte Vertreter sind die Korn-Shell, Bourne-Shell oder C Shell. Überdies hinaus stellt die Shell unsere gesamte Benutzerumgebung mit Umgebungsvariablen, bestimmten Voreinstellungen und der Fähigkeit zur Prozesskontrolle bereit. Prozesse besprechen wir zum einen im nächsten Abschnitt und in Kapitel 7 *Prozessverwaltung* (Seite 147).

1.2 UNIX-Konzepte

Das gesamte Betriebssystem basiert auf zwei grundlegenden Einheiten: Dateien und Prozessen. Das wird besonders deutlich, wenn wir einen Blick auf die Systemaufrufe werfen, denn die meisten befassen sich mit einem von beiden.

1.2.1 Prozesse und Prozessverwaltung

Prozesse bilden den Rahmen, in dem Programme ausgeführt werden. Wird ein UNIX-Programm ausgeführt, so geschieht das im Kontext eines Prozesses. Der Kontext wird unter anderem durch gewisse Statusinformationen, einem Verweis auf den Scheduler, einem Stack, einem Daten- und einem Textbereich charakterisiert. Diese Informationen werden zusammenfassend als *Process Image* (Prozessabbild) bezeichnet. So ein Prozess kann eine exakte Kopie von sich selbst erstellen, indem er den Systemaufruf `fork(2)` ausführt. Das Verfahren wird häufig von Serveranwendungen verwendet, die eine Instanz von sich selbst erstellen, um eine Client-Anfrage zu verarbeiten. Jeder Prozess kann einen anderen erzeugen, der wiederum einen anderen hervorbringen kann, so daß eine Beziehung zwischen den Einzelprozessen besteht, die sich als Baumstruktur darstellen lässt. Abbildung 1.2 illustriert diese Konstellationen.

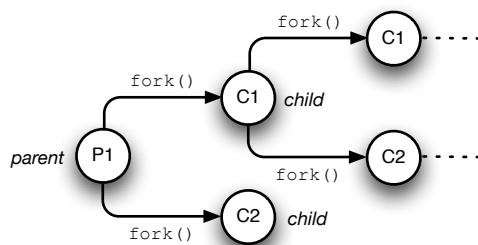


Abbildung 1.2: Eine Prozesskette, die durch den Aufruf von `fork` erzeugt werden kann.

Das sogenannte *Forking* beschreibt das Erzeugen eines neuen Kontextes und das Kopieren der Informationen, die in dem ursprünglichen Kontext enthalten waren. Nur einen vorhandenen Prozess einfach zu kopieren ist nicht besonders sinnvoll. Normalerweise wird er angewiesen ein anderes Programm zu starten, beispielsweise durch einen `exec(2)`-Systemaufruf, der wiederum selbst einen neuen Prozess erzeugt, dessen Process Image aber das der gerade von `fork(2)` erstellten Kopie vollständig überlagert. Der neue Prozess ist dann der Nachfahre des vorhandenen Prozesses. In der Regel spricht man dann von einem *Child Process* (Kindsprozess) und dem *Parent Process* (Elternprozess). Im Verlauf des Buches werden wir meist die Kurzform Parent und Child verwenden.

Neben Prozessen gibt es noch eine zweite Form einer Ausführungseinheit, die als *Thread* bezeichnet wird. Ein Thread ist eine abstrakte Einheit mit einem eigenen Ausführungspfad innerhalb eines

Prozesses. Er hat einen eigenen Stack, Programmzähler (*program counter*), etc. Durch Threads kann Gleichzeitigkeit und der Eindruck der Parallelität erreicht werden. Threads haben gewisse Vorteile gegenüber Prozessen, den das Betriebssystem muß viele Aufgaben erledigen um einen Prozess zu erzeugen und die Verwaltung relativ aufwendig, allerdings kann kein Thread außerhalb eines Prozesses existieren. Aus den genannten Gründen werden Prozesse als schwergewichtig und Threads als leichtgewichtig bezeichnet. Die Natur der Threads und die Interaktion mit dem Betriebssystem wird ausführlich in Kapitel 11 *POSIX-Threads* (Seite 311) erläutert.

Doch wie unterscheiden wir Prozesse, und woher wissen wir, ob wir es mit dem Parent oder Child zu tun haben? Das ist einfach: jedem Prozess wird eine eindeutige Bezeichnung zugeordnet, die *Process ID*. Zusammen mit der PPID (*parent process ID*) lässt sich die Vererbungskette genau bestimmen. Dabei ist das Dienstprogramm **ps(1)** sehr hilfreich.

```
% ps -eo pid,ppid,user,group,comm
  PID  PPID USER      GROUP      COMMAND
    1      0 root      root      init
    2      1 root      root      ksoftirqd/0
    3      1 root      root      events/0
    4      1 root      root      khelper
    9      1 root      root      kthread
   19      9 root      root      kacpid
   90      9 root      root      kblockd/0
  130      9 root      root      pdflush
  131      9 root      root      pdflush
  133      9 root      root      aio/0
```

Der **init**-Prozess hat immer PID 1 und alle anderen Prozesse stammen letztendlich von **init** ab, wenn auch nur indirekt.

Eine weitere wichtige ID, die jedem Prozess zugeordnet wird, ist die *User ID* (Benutzeridentifikation). Sie bestimmt, welche Berechtigungen dem Prozess in der Umgebung des Betriebssystems zur Verfügung stehen. Genau wie jeder Benutzer einer Gruppe angehört, ist das auch für jeden Prozess der Fall. Sie wird als *Group ID* (Gruppenidentifikation) genannt. Manchmal gehören einige Benutzer nicht nur einer sondern mehrerer Gruppen an. Diese Group IDs werden dann als *Supplementary Group IDs* (Ergänzende Gruppenidentifikationen) bezeichnet.

Um die ganze Sache noch etwas zu würzen, sind den Prozessen noch drei weitere Gruppen- und Benutzer-IDs zugeordnet: eine echte ID (*real*), eine effektive ID (*effective*) und eine mit sogenanntem SUID- und SGID-Bit (*saved user ID* und *saved group ID*). Alle drei haben eine spezielle und zum Teil subtile Bedeutung.

Die echte Benutzer- und Gruppen-ID beschreibt die Kennung mit der der Prozess läuft, also den Besitzer des Prozesses. Da Prozesse stets durch einen Benutzer gestartet werden entspricht die *echte ID* dem Benutzer, der den Prozess gestartet hat. Nun kommen die effektive Benutzer- und Gruppen-ID ins Spiel. Sie sind nach dem Start des Prozesses identisch. Um spezielle Operationen durchführen zu können, die über die Privilegien der Benutzer hinaus gehen, kann der Prozess mit Hilfe der beiden Funktionen **setuid(2)** und **setgid(2)** andere Benutzer- und Gruppen-IDs anfordern. Ab diesem Zeitpunkt läuft der Prozess *effektiv* mit anderen IDs als zuvor. Das gleiche kann auch erreicht werden, wenn ein Programm die SUID- oder SGID-Bits für einen bestimmten Benutzer oder eine Gruppe gesetzt hat. Dann werden die effektiven IDs automatisch beim Start des Programms verändert.

Ein Systemaufruf stellt eine Anfrage an das Betriebssystem zur Bereitstellung eines Dienstes dar. Dadurch wird die Verarbeitung des aktuellen CPU-Zyklus unterbrochen und die Kontrolle an das Betriebssystem übergeben. Jetzt kann das Betriebssystem zwischen den Prozessen umschalten. Das wird beispielsweise dann deutlich, wenn eine Anwendung das Betriebssystem über einen Systemaufruf auffordert Daten von einem Datenträger zu lesen. Wenn die Operation aus irgendwelchen Gründen relativ lang dauern kann, ordnet das Betriebssystem den Prozess in eine Warteschlange ein, die alle Prozesse enthält, welche auf das Gerät Zugriff erbeten. Anschließend wählt der Kernel einen anderen Prozess aus, der in dieser Zeit seine Aufgaben erledigen kann. Ist das Gerät nach einer gewissen Zeit wieder verfügbar, wird der aktuelle Ausführungspfad durch einen Interrupt unterbrochen und der nächste Prozess aus der Warteschlange ausgewählt und kann seine Operationen durchführen. Weitere

Informationen über diese und andere Mechanismen erhalten sie in Abschnitt 5.7 *Von langsamen und schnellen E/A-Operationen* (110).

Läuft der Prozess erst einmal, wird er einer Sitzung (*session*) zugeordnet. Eine Sitzung beginnt meistens mit dem Login. Der `login`-Prozess startet eine Shell, die eine Sitzung startet. Alle Prozesse, die innerhalb der Shell ausgeführt werden, gehören dann dieser Sitzung an. Zur Vereinfachung der Prozessverwaltung werden Prozesse innerhalb einer Sitzung noch in Prozessgruppen eingeteilt. Dabei unterscheiden wir zwischen Vordergrund- und Hintergrundprozessgruppen. Erstere hat direkten Zugang zum Terminal, kann also Benutzereingaben lesen oder selbst Meldungen ausgeben. Letztere haben diese Zugang nicht, laufen also im Hintergrund. Durch eine Einrichtung der Shell, die als *Job Control* bezeichnet wird, können Prozesse aus dem Hintergrund in den Vordergrund geholt werden und umgekehrt.

Ein Prozess ist immer der *Session Leader*. Nur er kann ein Terminal anfordern, über das Signale gesendet und empfangen werden können. Dieses Terminal, das sog. *controlling Terminal*, wird von allen Prozessen der Sitzung gleichermaßen geteilt und entspricht meist dem über das sich die BenutzerInnen eingeloggt haben. Das besondere an dem Controlling Terminal ist die Tatsache, daß nur dieses Signale (8 *Signalbehandlung*) an seine Prozessgruppe im Vordergrund senden kann.

Wenn ein Prozess gestartet wird, merkt er sich das Verzeichnis aus dem heraus es geschehen ist. Der Prozess darf das aktuelle Arbeitsverzeichnis (*current working directory*) selbstständig durch Aufruf von `chdir` ändern. Genauso verfügt er über ein Wurzelverzeichnis, das für die Auswertung absoluter Pfadangaben wichtig ist und mit `chroot` geändert wird.

1.2.2 Dateien und Dateibehandlung

Eingangs wurde erwähnt, daß fast jeder Systemaufruf in direktem oder indirektem Zusammenhang mit einer oder mehrerer Dateien steht. Dateien speichern Informationen und liefern eine Schnittstelle für den Zugriff auf Geräte, wie Festplatten, Terminals und Bandlaufwerke. Sogar Teile der Interprozesskommunikation (siehe Kapitel 10 *Interprozesskommunikation*) werden über Dateien geregelt, nämlich über sogenannte *Pipe Special Files*. Geräte unterscheiden wir über *Block Special Files* und *Character Special Files*. Mehr Informationen über Special Files finden Sie unter 4.7 *Device Special Files* (Seite 84).

Die Organisation von Dateien und Verzeichnissen unter UNIX ist einer Baumstruktur mit der Wurzel (/) ganz oben nachempfunden. Mehrere Dateien werden in Verzeichnissen organisiert, die selbst Verzeichnisse und Dateien enthalten können. Der Dateibaum kann allgemein als Dateisystem (*filesystem*) bezeichnet werden. Im Grunde sind Verzeichnisse auch Dateien, mit der Ausnahme, daß sie ein spezielles Flag aufweisen, welches anzeigt, daß es andere Dateien und Verzeichnisse aufnehmen darf. Das Format mit dem Dateien und Verzeichnissen beschrieben werden, wird durch das zugrundeliegende Dateisystem festgelegt. Das Dateiformat enthält mindestens den Dateinamen und eine eindeutige ID, die sogenannte *Inode Number*. Eigentlich ist die Inode Number eine Struktur, die Dateiattribute, wie Größe, Zugriffszeit und den Ort der Speicherung im Dateisystem enthält.

Jede Datei und jedes Verzeichnis verfügt über Zugriffsrechte, die bestimmen, ob BenutzerInnen die Datei oder das Verzeichnis lesen, schreiben oder ausführen dürfen. Nur Verzeichnisse, die ausführbar sind können durchsucht werden. Sie werden natürlich nicht wirklich ausgeführt, wie es mit Dateien der Fall ist (mehr Informationen im Abschnitt 4.2 *Dateizugriffsrechte*, Seite 61). Neben den Zugriffsrechten werden Dateien auch über einen Besitzer qualifiziert. So gehört eine Datei oder ein Verzeichnis immer genau einem Besitzer und einer Gruppe. So läßt sich eine Abstufung des Dateizugriffs realisieren, die entweder nur dem Besitzer, der Gruppe oder allen anderen Zugriff gewährt.

Problematisch im Umgang mit Dateien ist der gleichzeitige Zugriff durch mehrere BenutzerInnen. Das allgemeine Verständnis läßt sich etwa so zusammenfassen: wer zuerst kommt, kann die Datei öffnen, alle anderen müssen warten bis sie wieder freigegeben ist. Grundsätzlich nicht falsch, doch die Zusammenhänge sind etwas komplizierter. Die meisten Betriebssysteme haben einige ausgefeilte Mechanismen an Board, die uns bei der Synchronisierung des Zugriffs auf Dateien helfen. Es ist möglich eine Datei vollständig zu sperren oder nur teilweise. Letzteres Verfahren wird *Record Locking* genannt. Zwei unterschiedliche Ansätze sind verbreitet: *Advisory Locking* (empfohlenes Sperren) und *Mandatory Locking* (vorgeschriebenes Locking). Record Locking in Zusammenhang mit Advisory Locking sorgt dafür, daß

die Prozesse sich synchronisieren und die Zugriffe auf Segmente der Datei der Reihe nach geschehen. Beide Varianten besprechen wir in Abschnitt 5.8 *Zugriffe synchronisieren* (Seite 111).

Wir unterscheiden desweiteren zwischen Lese- und Schreibsperrern. Schreiben kann immer nur ein Prozess, lesen hingegen mehrere. Sobald eine Lesesperrre über die gesamte Datei oder nur ein Segment gesetzt ist, kann die Datei oder das Segment nicht gleichzeitig gelesen werden. Der Unterschied zwischen Advisory Locking und Mandatory Locking ist, daß ersteres darauf vertraut, daß sich die Prozesse an die Regeln halten. Es gibt keinen Mechanismus, der Prozesse davon abhält, nicht auf Bereiche zuzugreifen, die bereits gesperrt sind. Daher stammt die Bezeichnung „empfohlenes Locking“. Mandatory Locking ist ein Mechanismus, der von dem Betriebssystem bereitgestellt wird. Er wird über ein Zugriffsmodusbit gesteuert. Der Abschnitt 5.8 *Zugriffe synchronisieren* (Seite 111) befaßt sich ausführlicher mit diesem Thema.

1.2.3 Systemaufrufe (*system calls*)

Der Unterschied zwischen einem Systemaufruf und einer Schnittstelle für Anwendungsentwickler (API, *application programming interface*) ist, daß ersterer einen Dienst des Kernels über Software-Interrupts anfordert und letztere eine Funktionsdefinition, die bestimmt, wie beispielsweise ein verfügbarer Dienst angefordert werden kann.

Alle UNIX-Systeme werden mit unterschiedlichen Bibliotheken mit APIs für Entwickler ausgeliefert. Die wichtigste unter ihnen ist die *libc*-Bibliothek, die, neben anderen, auch Wrapper-Funktionen für Dienstauftrufe enthält. Wrapper-Funktionen haben nur den Zweck, Systemaufrufe durchzuführen. Jeder Systemaufruf, hat eine passende Wrapper-Funktion, die eine API definiert. Allerdings hat nicht jede API-Funktion auch einen passenden Systemaufruf. Das ist beispielsweise der Fall, wenn ein bestimmter Dienst im User Mode ausgeführt werden kann, wie etwa String-Operationen, die keine Kernel-Operation erfordern. Außerdem können innerhalb einer API-Funktion mehrere Systemaufrufe stattfinden.

Für uns ist wichtig, daß der POSIX-Standard nicht von Systemaufrufen sondern von API-Funktionen spricht. Systeme können alle notwendigen, vom Standard vorgeschriebenen, Funktionen vollständig im User Mode implementieren und trotzdem POSIX-konform sein. Aus Sicht der EntwicklerInnen ist dieser Unterschied bedeutungslos. Sie müssen nur wissen, welche Parameter eine POSIX-Funktion erwartet und welche Rückgabewerte geliefert werden. Für Kernel-Entwickler sieht das natürlich anders aus, denn Systemaufrufe finden vollständig im Kernel Mode und API-Funktionen im User Mode statt.

Die meisten Wrapper-Funktionen geben einen Fehlercode in Form eines Integers zurück. Dieser Code muß vom Systemaufruf (im Kernel Mode) an die Wrapper-Funktion (im User Mode) weitergegeben werden, damit der aufrufende Prozess den Code auslesen kann. Ein Systemaufruf kann durch ungültige Argumente, Fehler in der Hardware oder mangelnde Ressourcen fehlschlagen. Der spezifische Fehlercode wird in der *errno*-Variable (Abschnitt 2.5.3 geht auf dieses Thema genauer ein) gespeichert, die wiederum von der *libc* definiert wird.

Initiiert ein Prozess einen Systemaufruf, schaltet die CPU in den Kernel Mode und führt die Kernelfunktion aus. Normalerweise wird ein Systemaufruf durch das Auslösen eines Software-Interrupts angestoßen. Der Interrupt wiederum löst eine Ausnahme aus, die mit einem internen Vektor assoziiert ist. Dabei reicht der Prozess einen Parameter durch, der den auszuführenden Systemaufruf identifiziert. Alle Systemaufrufe geben einen Fehlercode zurück, der anzeigt, ob die Operation erfolgreich war (positiver Fehlercode oder 0) oder nicht (negativer Fehlercode). Die *errno*-Variable wird nicht vom Kernel gesetzt sondern durch die Wrapper-Funktion. Nachdem die Ausnahme ausgelöst wurde, ruft der System Call Handler die Kernelfunktion auf.

Folgende Operationen werden durchgeführt:

1. Der System Call Handler speichert den Inhalt der Register im Stack des Kernels (diese Operation ist vollständig in Assembler codiert).
2. Anschließend wird der Systemaufruf durch Ausführung der entsprechenden C-Funktion (auch *Service Routine* genannt) angestoßen.
3. Nach Ausführung der Service Routine kehrt der System Call Handler zurück und ruft eine spezielle Funktion auf, die den Fehlercode zurückgibt.

Abbildung 1.3 illustriert den Ablauf.

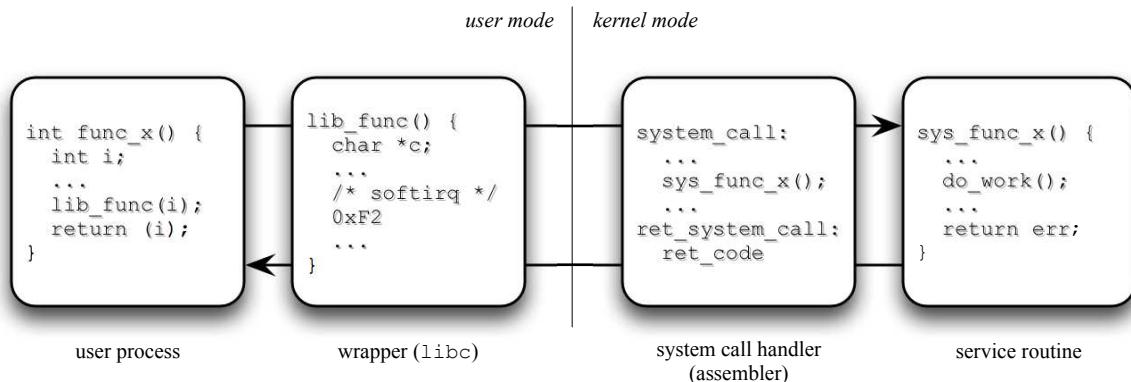


Abbildung 1.3: Zusammenhang zwischen Funktionsaufrufen und System Calls.

Mit Hilfe von Systemaufrufen und System Call Handlern wird ein Layer zwischen User Space und Kernel eingeführt, der Entwicklern die Arbeit erleichtert und die Portabilität von APIs erhöht. Für AnwendungsentwicklerInnen, an die sich dieser Text richtet, spielen nur die API-Funktionen eine Rolle.

1.3 UNIX-Programme entwickeln

Programme können mit dem zugrundeliegenden System nur interagieren, wenn sie gewissen Konventionen genügen. Die meisten UNIX-Programme sind in C oder C++ geschrieben, beispielsweise in ANSI C, K&R C oder ISO C++. Der Einstiegspunkt eines C-Programms ist die `main`-Funktion, die zwei optionale Argumente aufweisen kann:

```
int main(int argc, char *argv[])
{
    /* einige Statements */
}
```

Das Argument `argc` zeigt die Anzahl der Parameter an, die beim Start an das Programm übergeben wurden und weist immer mindestens den Wert 1 auf, da der Programmname selbst mitgezählt wird. Das Feld `argv` enthält Zeiger auf andere Zeichenketten, die alle übergebenen Parameter enthält. Die Position `argv[0]` enthält einen Zeiger auf den Namen des Programms.

Der Rückgabewert des Programms ist ein Integer der Aufschluß über den Erfolg des Aufrufs. Die Konvention besagt, daß ein Wert größer oder gleich 0 eine erfolgreiche Programmausführung darstellt und ein Wert kleiner 0 auf einen Fehler schließen lässt. Dieser Code wird bei Beendigung des Programms an den Parent über den Systemaufruf `exit(2)` weitergeleitet.

UNIX-Programme lassen sich grob in zwei Kategorien einteilen: *Daemons* und *Benutzerprogramme*. Benutzerprogramme werden in der Regel über die Shell gestartet, erfüllen eine bestimmte Aufgabe und kehren wieder zurück. Einige sind interaktiv und erwarten Benutzereingaben wie beispielsweise `cp`, `dd` oder `vi`.

Entweder laufen Programme im Vordergrund oder im Hintergrund. Programme im Vordergrund arbeiten stets synchron mit der Shell, sie wartet auf das Programm bis es fertig ist. Die Hintergrundprozesse laufen asynchron. Daemon-Prozesse sind nicht mit einer bestimmten Login-Shell verbunden und gehörigen aus diesem Grund auch keiner Session an. Sie sind nicht interaktiv und werden beispielsweise beim Systemstart aufgerufen oder durch ein Kommando, das in der Shell ausgeführt wurde.

1.3.1 Programme übersetzen: Überblick

Wir übersetzen unsere Programme üblicherweise mit einem C-Compiler. Fast alle UNIX-Systeme bringen einen C-Compiler mit oder liefern einen als Zusatzprogramm aus. Um ein C-Programm, das im Quelltext vorliegt, zu übersetzen, reicht ein einfacher Aufruf von:

```
% cc myprogram.c
```

Das Ergebnis ist eine ausführbare Datei mit dem Namen `a.out`, die im aktuellen Arbeitsverzeichnis abgelegt wird. Der Compiler selbst übernimmt noch einige Schritte, die wir gar nicht mitbekommen. Beispielsweise wird der Linker (`ld(1)`) ausgeführt, der den erzeugten Code mit all den notwendigen Bibliotheken verknüpft, damit ein ausführbares Image entsteht.

Der C-Compiler kennt eine Option, die das ausführbare Programm gleich richtig benennt, damit wir es nicht manuell mit `mv(1)` umbenennen müssen:

```
% cc myprogram.c -o myprogram
```

Dieser Aufruf erzeugt eine ausführbare Datei mit dem Namen `myprogram`. Der Compiler weiß, wie er den Code gegen die C-Bibliothek linken muß. Setzen wir in unserem Programm externe Bibliotheken ein, so müssen wir den Compiler darüber informieren, was wir mit dem Schalter `-l` erledigen können:

```
% cc myprogram.c -o myprogram -lXm -lXt -lX11
```

Alle Bibliotheken, die nach dem Schalter `-l` angeführt sind, werden dem Linker übergeben, wobei die Reihenfolge von Bedeutung ist. Beispielsweise ist `Xm` abhängig von `Xt` und `Xm` und `Xt` sind beide abhängig von `Xlib`, was durch Schalter `-lX11` dargestellt wird.

1.3.1.1 C-Programme übersetzen

Was C-Compiler angeht, so gibt es eine ganze Reihe von Unterschieden zwischen den UNIX-Derivaten. Mac OS X liefert beispielsweise:

```
% gcc --version
686-apple-darwin9-gcc-4.0.1 (GCC) 4.0.1 (Apple Inc. build 5484)
```

Das `gcc`-Kommando ist lediglich ein Link auf `/usr/bin/cc` wie ein kurzer Test zeigt:

```
% type cc
cc is /usr/bin/cc
% ls -li /usr/bin/cc
1176059 lrwxr-xr-x 1 root  wheel  7 Sep  1 16:22 /usr/bin/cc -> gcc-4.0
% type gcc
gcc is hashed (/usr/bin/gcc)
% ls -li /usr/bin/gcc
1176062 lrwxr-xr-x 1 root  wheel  7 Sep  1 16:22 /usr/bin/gcc -> gcc-4.0
% ls -li /usr/bin/gcc-4.0
1160170 -rwxr-xr-x 1 root  wheel  93088 May  3 11:07 /usr/bin/gcc-4.0
```

Wie wir sehen können ist `/usr/bin/gcc` ein symbolischer Link auf die aktuelle Compiler-Version.

Systeme mit eigenen Compiler-Implementierungen weichen in der Regel stark von der GNU Compiler Collection ab. Sie verwenden andere Schalter, geben andere Warnungen aus und unterstützen eventuell andere Standards. In dieser Sektion schauen wir uns einige wichtige Schalter und Warnungen der GNU Compiler Collection an.

1.3.1.2 C-Compiler verwenden

Die meisten UNIX-Derivate verwenden den `cc`-Befehl zum Aufruf des C-Compilers. FreeBSD und Linux erlauben auch die Verwendung von `gcc` zum gleich Zweck, da die GNU Compiler Collection bereits vorinstalliert ist. Kommerzielle Systeme unterscheiden hingegen zwischen `cc` und `gcc` um den eigenen Compiler von der GNU-Version zu differenzieren. HP beispielsweise liefert mit HP-UX einen Compiler mit, der nicht ANSI-konform ist. Alternativ kann ein ANSI-Compiler erworben werden.

Die Option `-c`

Der Schalter ist womöglich allen C-Compilern gemein und instruiert den Compiler eine übersetzte Objektdatei (Endung `.o`) zu erzeugen, sie allerdings nicht mittels Linking in eine ausführbare Datei umzuwandeln. Die Option ist nützlich um mehrere Quelldateien separat zu übersetzen und sie später om Linker binden zu lassen.

```
% cc test.c
```

In diesem Beispiel wird die Quelldatei `test.c` in einem Durchlauf direkt in die ausführbare Datei `a.out` übersetzt und gelinkt. Der Dateiname `a.out` ist Standard für alle anonym erstellten Module. Interessanterweise ist diese Tatsache auf das Verhalten der PDP-11 von DEC zurückzuführen als UNIX noch vollständig in Assembler geschrieben wurde.

Letztendlich können wir mit Hilfe des Schalters `-c` die einfache Erstellung von `a.out` auch in zwei Schritten durchführen:

```
% cc -o test.o
% cc test.o
```

Das Resultat ist das gleiche: `a.out`. Der Name der Ausgabedatei kann mit der Option `-o` gesteuert werden.

Die Option `-o`

Mittels `-o` können wir den Namen der Ausgabedatei bestimmen:

```
% cc test.c -o test
```

Dieses Kommando erstellt eine ausführbare Datei mit dem Namen `test`.

Die Option `-g`

Auch dieser Schalter ist den meisten Compilern bekannt und dient dazu, Debuginformationen für die ausführbare Datei zu erstellen. Mit Hilfe der Informationen können Variablennamen und andere Symbole im Debugger inspiziert werden, nachdem beispielsweise eine `core`-Datei aufgrund eines Programmabruchs erzeugt wurde. Verwenden Sie diese Option immer wenn Sie ein Programm interaktiv debuggen möchten. Vergessen Sie dabei aber nicht alle Module mit dem Schalter zu übersetzen.

Die Option `-D`

Die standardisierte Option `-D` erlaubt es uns, Makrosymbole an den Compiler zu übermitteln. Normalerweise wird das über ein Makfile gelöst, was aber nicht zwingend erforderlich ist.

```
% cc test.c -D_POSIX_C_SOURCE=199309L -o test
```

In diesem Beispiel definieren wir die Makrokonstante `_POSIX_C_SOURCE` mit dem Wert `199309L`. Somit erzwingen wir den Standard nach der die Übersetzung geschehen soll.

Die Option `-I`

Manchmal ist es notwendig, den Compiler anzugeben, neben den Standardverzeichnissen auch bestimmte Verzeichnisse nach Headern zu durchsuchen. Beispielsweise könnte es sein, daß wir Headerdateien für ein Projekt in speziellen Verzeichnissen speichern, beispielsweise `/usr/local/src/include`, so daß wir die Option `-I` verwenden können, wenn wir sie referenzieren möchten:

```
% cc test.c -I/usr/local/src/include -o test
```

Selbstverständlich können Sie beliebig viele Verzeichnisse angeben. Standardmäßig durchsucht der Compiler im Verzeichnis des Modules, anschließend werden die angegebenen Verzeichnisse durchsucht und schließlich `/usr/include`.

Die Option `-E`

Auch diese Option wird von den meisten Compilern unterstützt. Wird sie gesetzt gibt der Compiler den vorbereiteten (preprocessed) Code auf der Kommandozeile aus bevor er übersetzt wird. Das kann nützlich sein, um zu sehen, wie bestimmte Makros erweitert werden, was oftmals schwer zu debuggen ist. Mit Hilfe einer Umleitung kann der Code in eine Datei geschrieben werden:

```
% cc -c -E test.c > out.txt
```

Die Option `-O`

Optimierungen unterschiedlichster Intensität können mittels `-O` angegeben werden. Die wenigsten nicht-GNU-Compiler unterstützen diese Option. Der jeweilige Optimierungsgrad wird als konstante zwischen 0 und 4 angegeben wobei 0 keine Optimierung anzeigt und 4 den höchsten Optimierungsgrad. Beispiel:

```
% cc -c -O3 -O1 test.c
```

Wir haben zwei Optimierungsgrade spezifiziert: 3 und 1. Allerdings gewinnt immer die letzte Optimierungsangabe so daß wir mit Optimierungsgrad 1 und nicht mit 3 übersetzen.

Optimierungen und Option `-g` vertragen sich nicht

Die meisten C-Compiler erlauben nicht die gleichzeitige Angabe von `-O` und `-g`, da die Debug-informationen nicht optimiert werden können. Allerdings ist der GNU C Compiler in der Lage `-O1` und `-g` zu verarbeiten, aber keine höheren Optimierungsgrade.

Die Option `-W`

Warnungen sollten immer ausgegeben und nicht unterdrückt werden. Warnungen verhindern nicht den Übersetzungsvorgang, können aber später zu Problemen führen. Der Compiler ist in der Lage selbst subtile Nachlässigkeiten im Code zu identifizieren. In der Regel spezifizieren wir `-Wall` um über alle Probleme informiert zu werden. Möchten wir Warnungen zu spezifischen Problemen erhalten, können wir beispielsweise `-Wreturn-type` angeben um über das Fehlen oder Mismatches von Rückgabewerten informiert zu werden.

1.3.1.3 Standardkonforme Programmübersetzung

Viele UNIX-Derivate sind bemüht, gewissen C-Standards zu genügen. Zusätzlich unterstützen sie viele Erweiterungen, die von den Standards nicht abgedeckt werden. Darüber hinaus kann zwischen verschiedenen C-Standards gewählt werden, so daß sich die Frage stellt, wie wir wissen sollen, welchem Standard wir beide der Übersetzung von Programmen folgen sollen.

Unter UNIX können wir mehrere Feature Test Makros verwenden, um die Standards zu spezifizieren. Wird kein Standard angegeben folgt das System einer Standardeinstellung, allerdings ist es ratsam einen Standard zu wählen um Schwierigkeiten von Anfang an zu vermeiden, die beim Betrieb auf diversen UNIX-Derivaten auftreten könnten.

Der Header `<unistd.h>` definiert Makros, die anzeigen, welchen Standards das System genügt. Tabelle 1.1 listet die wichtigsten Makros, deren Werte und die jeweiligen Standards auf.

Möchten wir Sprachstandards definieren, so stehen uns die Makros aus Tabelle 1.2 zur Verfügung.

Um den aktuellsten Standard für ein System zu identifizieren, könnten wir folgende `#ifdef`-Sequenz verwenden:

Name	Makro	Standard
POSIX.1-1988	_POSIX_VERSION = 198808L	
POSIX.1-1990	_POSIX_VERSION = 199009L	ISO/IEC 9945-1:1990
POSIX.2	_POSIX2_C_VERSION = 199209L	ISO/IEC 9945-2:1993
POSIX.1b-1993	_POSIX_VERSION = 199309L	IEEE 1003.1b-1993
POSIX.1b-1993	_POSIX_VERSION = 199309L	IEEE 1003.1b-1993
POSIX.1-1996	_POSIX_VERSION = 199506L	IEEE 1003.1-1996
POSIX.1-2001	_POSIX_VERSION = 200112L	IEEE 1003.1-2001
XPG3	_XOPEN_VERSION = 3	X/Open Portability Guide 3 (1989)
XPG4	_XOPEN_VERSION = 4	X/Open Portability Guide 4 (1992)
SUS	_XOPEN_VERSION = 4 && _XOPEN_UNIX	X/Open Single UNIX Specification (UNIX95)
SUSv2	_XOPEN_VERSION = 500	X/Open Single UNIX Specification, Version 2 (UNIX98)
SUSv3	_XOPEN_VERSION = 600	Open Group Single UNIX Specification, Version 3 (UNIX03)

Tabelle 1.1: UNIX-Standards und Makros

Name	Makro	Standard
C89	--STDC__	ANSI X3.159-1989
C90	--STDC_VERSION__	ISO/IEC 9899:1990
C94	--STDC_VERSION__ = 199409L	ISO/IEC 9899-1:1994
C99	--STDC_VERSION__ = 199901L	ISO/IEC 9899:1999
C++98	--cplusplus = 199707L	ISO/IEC 14882:1998
C++/CLI	--cplusplus_cli = 200406L	ECMA-372
EC++	--embedded_cplusplus	Embedded C++

Tabelle 1.2: Sprachstandards und Makros

```

#define _if defined(_STDC__)
#define _define define
#define _if defined(_STDC_VERSION__)
#define _define define
#define _if (_STDC_VERSION__ >= 199409L)
#define _define define
#define _endif endif
#define _if (_STDC_VERSION__ >= 199901L)
#define _define define
#define _endif endif
#define _endif endif
#endif

```

In Abschnitt 2.5.1.2 beschäftigen wir uns ausführlich mit verschiedenen Standards und der Ihre Historie.

1.3.2 Programmdesign und Fehlerbehandlung

Wir wollen folgendes Programm heranziehen, um einige Punkte zum Design von UNIX-Programmen und der Fehlerbehandlung zu besprechen:

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 int main(int argc, char **argv) {
6     int i; /* counter variable */
7
8     /* at least one argument is needed */
9     if (argc == 1) {

```

```

10         printf("Usage: %s <arguments>\n", argv[0]);
11         exit(-1);
12     }
13
14     printf("Arguments given: %d\n", argc);
15
16     /* print arguments, one line each */
17     for (i = 0; i < argc; i++)
18         printf("argv[%d]: %s\n", i, argv[i]);
19
20     return (0);
21 }
```

Listing 1.1: xcode/sample1.c - Einfaches Beispielprogramm mit rudimentärer Fehlerbehandlung

Ein Aufruf könnte zu dieser Ausgabe führen:

```

% ./sample1
Usage: ./sample1 <arguments>
% /export/home/graegeerts/tmp/sample1
Usage: /export/home/graegeerts/tmp/sample1 <arguments>
% ./sample1 Test "abc 123"
Arguments given: 3
argv[0]: sample1
argv[1]: Test
argv[2]: abc 123
```

Es ist ein sehr kurzes Programm, das nichts weiter macht, als die ihm übergebenen Parameter auszugeben. Da ist nichts besonderes dran, lässt aber Spielraum für Verbesserungen, wie wir gleich sehen werden. Bemerkenswert ist in jedem Fall nur, daß wir die Anzahl der Parameter prüfen (`argc`), um die BenutzerInnen darüber aufzuklären, wie das Tool zu verwenden ist. Nur dumm, daß die Ausgabe des Programmnamens auch immer die gesamte Aufrufsyntax einschließt. Das zu ändern, wäre leicht, wir könnten den Namen des Werkzeugs doch einfach fest in den Code einbauen. Das bedeutet aber weniger Flexibilität und weniger Flexibilität möchten wir uns nicht leisten. Wir werden uns gleich einen Verbesserungsvorschlag ansehen.

Das Listing 1.2 zeigt eine verbesserte Variante.

```

1 #include "header.h"
2
3 int main(int argc, char *argv[]) {
4     int i;          /* counter variable */
5     char *basename; /* name of program */
6
7     /* fetch program name (see basename_ex() in plib.c) */
8     if ((basename = strrchr(argv[0], '/')) == 0)
9         basename = argv[0];
10    else
11        basename++;
12
13    /* at least one argument is needed */
14    if (argc == 1)
15        err_fatal("Usage: %s <arguments>\n", basename);
16
17    printf("Arguments given: %d\n", argc);
18
19    /* print arguments, one line each */
20    for (i = 0; i < argc; i++)
21        printf("argv[%d]: %s\n", i, argv[i]);
22
23    return (0);
```

```
24 }
```

Listing 1.2: xcode/sample2.c - Verbessertes Beispielprogramm mit flexibler Fehlerbehandlung

Das Programm rufen wir genau so wie zuvor auf:

```
% ./sample2
Usage: sample2 <arguments>
% /export/home/graegeerts/tmp/sample2
Usage: sample2 <arguments>
% ./sample2 Test "abc 123"
Arguments given: 3
argv [0]: sample1
argv [1]: Test
argv [2]: abc 123
```

Erste Verbesserung: in allen Programmen, die Parameter erwarten verwende ich den Vierzeiler von Zeile 11 bis 14:

```
if ((basename = strrchr(argv[0], '/')) == 0)
    basename = argv[0];
else
    basename++;
```

Die Funktion `strrchr(3)` sucht in dem String, der als erstes Argument übergeben wird (hier: `argv[0]`, nach dem letzten Auftreten des Zeichens, welches als zweites Argument übergeben wird (hier: `'/'`). Der Rückgabewert ist die Zeichenkette ab dem letzten Vorkommen des spezifizierten Zeichens. So lässt sich der Programmname aus der Zeichenkette in `argv[0]` extrahieren. Meistens handelt es sich dabei um den vollständigen Pfad oder um `./sample1`, was so viel bedeutet wie: führe das Programm `sample1` im aktuellen Verzeichnis aus. Der Programmteil eines Pfades wird als *Basename* bezeichnet.

Jetzt brauchen wir nur noch die Parameter der Reihe nach durchzugehen und auszugeben. Das ist mit einer `for`-Schleife schnell erledigt. □

Die in Listing 1.1 eingefügten Header gehören zu den am häufigsten verwendeten. Alle drei werden für fast jedes Programm benötigt. Es bietet sich an, die drei in einen eigenen Header zu verlegen, damit wir nicht mehr so viel tippen müssen. Unser Header für die folgenden Beispiele heißt `header.h`. Und wenn einer der Header zufälligerweise doch nicht benötigt wird, dann sorgt der Linker dafür, daß die überschüssige Teile abgeworfen werden. Ein zweiter Vorteil ist es eigene Definitionen (`#define`) und Deklarationen in dieser Headerdatei unterzubringen. Beispielsweise kann es passieren, daß wie immer mal wieder Datei mit immer den gleichen Zugriffsrechten öffnen müssen. Da ist es sinnvoll eine Definition folgender Art zu hinterlegen:

```
#define FMODE S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
```

Wenn wir das nächste mal `open(2)` aufrufen, brauchen wir nicht den ganzen Rattenschwanz von Optionen anzugeben, sondern einfach nur `FMODE`:

```
fd = open("myfile.c", FMODE);
```

Womit wir auch schon beim nächsten Punkt sind: Fehlerbehandlung. Es ist wichtig, stets den Rückgabewert einer Funktion zu prüfen, um herauszufinden, ob der Aufruf erfolgreich war oder nicht. Das erreichen wir am einfachsten, in dem wir die gesamte Anweisung in die Bedingung integrieren, wie das beispielsweise für den Aufruf von `strrchr(3)` geschehen ist.

```
if ((basename = strrchr(argv[0], '/')) == 0) { /* statements */ }
```

Durch diese Konstruktion erledigen wir zwei Schritte auf einmal: zuerst wird das Ergebnis der Variablen zugewiesen und anschließend vergleichen wir das Ergebnis und reagieren darauf entsprechend.

In Anhang C (Seite 473) stelle ich zwei Funktionen vor, die die Fehlerbehandlung ganz erheblich erleichtern. Um zu zeigen, wie sie angewendet werden, habe ich sie bei der Prüfung Parameterzahl eingesetzt:

```
if (argc == 1)
    err_fatal("Usage: %s <arguments>\n", basename);
```

Stimmt die Zahl der Parameter nicht mit der erwarteten Anzahl überein, so brechen wir das Programm ab und geben eine Fehlermeldung aus. Wenn der Fehler nicht so schwerwiegend ist, daß die Ausführung des Programms nicht gefährdet ist, verwenden wir `err_normal` und nicht `err_fatal`. Weitere Informationen dazu finden Sie in Anhang C *Fehlerbehandlung* (ab Seite 473).

1.3.3 Sicherheitsaspekte in der Programmierung mit C

An Systemprogramme mit langen Laufzeiten werden besondere Anforderungen gestellt: sie sollten Fehlertolerant sein, die Ressourcen richtig einteilen, etc. Für Programmierer solcher Software sind das keine unüberwindbaren Probleme erfordern aber ein hohes Maß an Aufmerksamkeit und technischem Hintergrundwissen.

Fast alle UNIX-Funktionen (ob System- oder Bibliotheksaufrufe) geben einen Fehlercode zurück, der dem Aufrüfer mitteilt, ob die Operation erfolgreich verlaufen ist oder nicht. Oftmals ist der Fehlercode auch gleichzeitig ein Ergebniswert. Beispielsweise könnte eine Funktion eine Zeichenkette verarbeiten und wenn die Manipulation erfolgreich war gibt sie den neuen String zurück, oder NULL, wenn ein Fehler aufgetreten ist. Die Prüfung des Fehlercodes ist Aufgabe des Entwicklers. C stellt hier keine Anforderungen. Versäumnisse dieser Art können zu subtilen Fehlern führen, denn wird der Code über den Punkt des Auftretens eines schwerwiegenden Fehlers weiter ausgeführt, kann der Fehler viel später zu Tage treten, als er im Ausführungspfad aufgetreten ist. Das Problem wird deutlich, wenn wir Daten in ein Array schreiben und dabei den zugewiesenen Speicherbereich überschreiten. Das C-Laufzeitsystem wird sich nicht darüber beschweren; es schreibt die Daten einfach in den Speicher und verändert möglicherweise andere Daten. Die Auswirkungen treten erst zu Tage wenn die betreffenden Daten verwendet werden. Fehler dieser Art sind schwer aufzuspüren und ziemlich gefährlich. Andere, neuere Programmiersprachen unterbinden das Verhalten durch Laufzeitprüfungen der Speicherbereiche.

Untersuchungen ([Chou01]) haben ergeben, daß die meisten Bugs durch fehlende Prüfung der Zeiger, die durch Funktionen zurücklefern, verursacht werden. Wie erwähnt geben einige Funktionen `null` anstelle eines Zeigers auf einen Wert zurück, um einen Fehler anzuzeigen. Wird das nicht überprüft, kommt es zu oben genannten Problemen.

1.3.3.1 Der Buffer-Overflow

Einer der am meisten verursachten und ebenso gefürchtetsten Fehler ist der Pufferüberlauf (*buffer overflow*). Selbst Software, die bereits viele Jahre erprobt und studiert wurde ist vor diesem Phänomen nicht sicher. Doch wie entstehen Buffer Overflows und wie können wir sie vermeiden?

Ein Pufferüberlauf entsteht, wenn ein Programm Daten in eine Variable kopiert, für die nicht genug Speicher bereit stehen. Folgender Code zeigt eine mögliche Konstellation, die schnell zu einem Pufferüberlauf führen kann:

```
char user_input[20];

printf("Bitte geben Sie ihren Namen ein: ");
scanf("%s", user_input);
```

Wir haben eine Puffer von 20 Bytes eingerichtet. Schreibt ein Benutzer mehr als 19 Zeichen (nur 19, denn der String wird NUL-terminiert) schreibt `scanf` über die Grenzen der Variablen im Hauptspeicher hinaus. Die erste Überlegung vieler Entwickler geht in Richtung Vergrößerung des Buffers, so daß auch lange Eingaben problemlos in den Buffer passen. Das ist nicht nur verschwenderisch sondern auch kurzsichtig. Natürlich ist die Wahrscheinlichkeit gering, daß ein Benutzer mehr als 999 Zeichen eingeibt, andererseits haben wir aber dafür aber zu viel Speicherplatz belegt. Besser wäre es, nur so viel in den Buffer zu schreiben, wie hineinpäßt. Etwa so:

```

char user_input[20];

printf("Bitte geben Sie ihren Namen ein: ");
fgets(user_input, sizeof(user_input), stdin);

```

In dieser Variante wird nur so viel in den Puffer geschrieben, wie auch tatsächlich hineinpaßt. Es gibt unzählige Funktionen, die keine Längenprüfung der Eingaben vornehmen, bevor sie in den Speicher geschrieben werden. Sie sind meist nur noch aus Gründen der Abwärtskompatibilität vorhanden. Die Manpages solcher Funktionen raten meist dringend von der Nutzung ab und verweisen auf Alternativen. `fgets(3)` ist so ein Beispiel. Mit `gets` können wir das gleiche erreichen, nur fehlt der zweite Parameter, der die Größe des Buffers angibt.

Werfen wir einen kleinen Blick auf die Interna, um zu verstehen, wie Buffer Overflows entstehen. Der meiste Code wird von Funktionen mit lokalen Variablen ausgeführt. Lokale Variablen werden automatisch auf dem Stack (dt. Stapel) verwaltet. Normalerweise wächst der Stack mit der Anzahl der Elemente von oben (dem oberen Speicherbereich) nach unten (dem unteren Speicherbereich) an. Wird eine Funktion aufgerufen, enthält der untere Teil des Stacks die übergebenen Argumente und die Rücksprungadresse. Weiter oben befinden sich die lokalen Variablen. Ruft eine Funktion wieder eine andere Funktion auf, passiert das gleiche noch einmal. Der Stack wächst und die Rücksprungadressen werden immer hinter den lokalen Variablen abgelegt.

Schreibt ein Programm nun über das Limit der Variablen auf dem Stack hinaus, kommt es zu einem Buffer Overflow. Entweder haben wir Glück und wir schreiben Bytes in unbenutzten Speicher oder nicht. Dann überschreiben wir möglicherweise Bytes anderer Variablen, Rücksprungadressen oder anderer Elemente, auf die wir eigentlich nicht zugreifen wollten. Das Verhalten des Programms kann vielfältig sein. Es kann das System direkt abstürzen, ein Core-Dump (das Process Image wird auf die Festplatte geschrieben) erzeugt werden oder anderes undefiniertes Verhalten auftreten.

Die Funktion `get_pass` fragt nach dem Benutzernamen und speichert ihn in `buf`. Stimmt der Name wird 1, andernfalls 0 zurückgegeben.

```

int get_pass(void) {
    int x = 0;
    char buf[10];

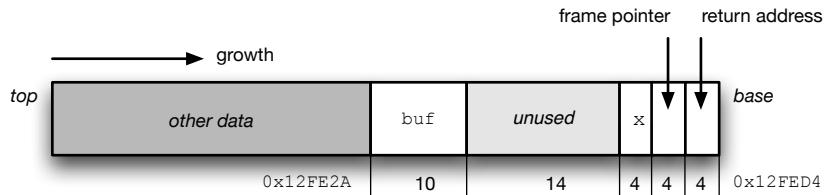
    printf("Bitte Namen eingeben: ");
    if (scanf("%s", buf) < 0) {
        printf("scanf() failed\n");
        exit(-1);
    }
    printf("[x]@%p, [buf]@%p\n", &x, buf);
    if (strcmp(buf, "steve") == 0)
        x = 1; /* true */
    else
        x = 0; /* false */

    return x;
}

```

Bei Aufruf der Funktion werden zwei lokale Variablen angelegt, so daß der Stack so aufgebaut sein könnte, wie in Abbildung 1.4. Der Unbenutzte Speicherbereich wird durch den Compiler angelegt, damit die Zeiger besser ausgerichtet werden können.

Beachten Sie, daß wir jetzt `scanf` einsetzen und keine Längenprüfung durchgeführt wird. Solange der eingegebene Name nicht länger als 23 Zeichen ist (10 Bytes + 14 Bytes) überschreiben wir zwar keine anderen Variablen. Bei 24 Zeichen überschreibt die NUL-Terminierung den Wert von `x` und bei mehr als 24 Zeichen würde `x` immer größer 0 sein und die Funktion immer 1 zurückgeben. Ist das Passwort so lang, daß einer der Zeiger überschrieben wird, könnte das Programm an einen Speicherbereich zurückspringen, der außerhalb des Adressraums des Prozesses liegt und damit evtl einen *Segmentation Fault* auslösen, der einen Core Dump zur Folge hätte.

Abbildung 1.4: Stack-Layout nach Aufruf der Funktion `get_pass`.

1.3.3.2 Buffer-Overflow und Sicherheit

Sichere Programme zu entwickeln ist, wie die Vergangenheit leider immer öfter gezeigt hat, nicht mehr nur eine Einstellungsfrage, sondern vielmehr auch mit Verantwortung verbunden. Der Buffer-Overflow spielt dabei eine nicht un wesentliche Rolle. Große Teile weit verbreiteter Software enthalten noch immer mögliche Quellen für Buffer Overflows. Das Problem ist nicht auf ein Betriebssystem beschränkt. Ob SunOS, QNX, Symbian OS oder Windows. Sie sind alle anfällig für diese Fehlerquelle. Solange Programme in C oder C++ geschrieben werden, besteht die Gefahr auch weiterhin.

Buffer-Overflows können ein System auf vielfältige Weise kompromittieren. Beispielsweise hat Robert Morris 1988 einen Internetwurm freigesetzt, der sich einen Buffer-Overflow im `finger`-Daemon zu Nutze gemacht hat. `finger` ist ein Program, mit dem man sich Informationen über Benutzer auf entfernten Systemen anzeigen lassen kann.

Hier wird deutlich, daß speziell präparierter Code einen Buffer-Overflow gezielt ausnutzen kann, in dem bestimmte Adressen mit eigenen Werten überschrieben werden, so daß Code ausgeführt wird, der eigentlich nicht Teil des Programms ist. Solche Exploits sind häufig auf vielen einschlägigen Websites zu finden und das Resultat erst kürzlich entdeckter Sicherheitslöcher.

Ein anderes Beispiel für einen Buffer-Overflow finden wir in Sun's XFS (*X Windows Font System*), das für den Export von Schriftinformationen in einem X Windows Netzwerk entworfen wurde. So war es einem Angreifer möglich, die Funktion `Dispatch` des `fs.auto`-Dienstprogramms mit präparierten Daten zum Überlaufen zu bringen, da es an einer Prüfung der übergebenen Daten mangelte. Auf diese Weise hätte das System zum Absturz oder sogar zur Ausführung beliebigen Codes gebracht werden können.

Aus diesen Ausführungen können wir drei Regeln ableiten:

1. Wenn wir Daten in Variablen speichern, insbesondere in Arrays, wie beispielsweise Zeichenketten, verwenden wir nur Funktionen, die auch eine Längenprüfung der zu schreibenden Daten vornehmen. Aktuelle APIs bieten stets eine Variante an, die diese Bedingung erfüllt.
2. Bei der Implementierung eigener Funktion sollten wir Punkt 1 immer beherzigen und selbst eine Prüfung der Daten vornehmen und eventuell einen Fehlercode zurückliefern, falls die Prüfung negativ ausfallen sollte.
3. Da viele Funktionen Zeiger auf Werte zurückliefern, sollten wir den Zeiger immer auf NULL prüfen, um herauszufinden, ob nicht doch ein Fehler aufgetreten ist. Dieses Konzept wird in allen Quelltexten in diesem Text praktiziert.

Behalten wir die drei Punkte immer im Hinterkopf, so besteht eine gute Chance, stabile und sichere Programme zu entwickeln.

1.3.3.3 Kleine Checkliste für sicheres Programmieren

Obwohl das Sicherheitsmodell von UNIX weitentwickelt ist, können wir uns nicht darauf verlassen, denn Programmierer sind oftmals nachlässig. Die meisten Sicherheitsprobleme sind auf Fehler in den Anwendungen zu finden, die auf unzureichende Fehlerbehandlung, mangelhafte Prüfung von Eingaben und anderen Praktiken abzuleiten sind. Betrachten wir einige Techniken, die sich in der Vergangenheit als praktikable Ansätze zur sicheren Anwendungsprogrammierung herausgebildet haben.

1. Prüfe immer alle Argumente. Immer wenn wir mit Argumenten und Parametern arbeiten, müssen wir darauf achten, sie einer ausreichenden Prüfung zu unterziehen:
 - Kommandozeilenparameter stets auf Vollständigkeit und Gültigkeit prüfen.
 - Argumente, die wir Systemaufrufen zuführen, sollten zuvor immer auf die richtigen Typen geprüft werden. Die Größe von Arrays sollte immer bekannt oder festgelegt sein. Das wird oftmals durch Definition von Konstanten (z.B. `#define NUM_ITEMS`) oder Längenparameter durchgeführt.
 - Prüfen Sie alle Daten, die wir aus einer Datei auslesen, um sicherzustellen, daß wir keine unerwünschten Elemente aufgreifen, z.B solche, die wir nicht verarbeiten können.
 - Im Zweifelsfall sollten Variablen in den erwarteten Typ umgewandelt werden, beispielsweise `signed int` in einen `unsigned int`, damit wir auf jeden Fall einen positiven Wert erhalten.
2. Verwenden Sie niemals Funktionen, die keine Längenprüfung der Eingaben vornehmen. Sie sind eine der häufigsten Quellen für Buffer Overflows. Beispiele sind: `fgets(3)` statt `gets(3)`, `strncpy(3)` statt `strcpy(3)`, `strncat(3)` statt `strcat(3)` usw.
3. Beachten Sie, daß ein String immer NUL-terminiert ist und damit eine Länge von `strlen + 1` aufweist. Der String `abc` ist daher nicht 3 Zeichen, sondern 4 Zeichen lang, auch wenn `strlen(3)` 3 zurückgibt.
4. Sollte nur ein kleiner Teil ihres Programms SUID-Rechte benötigen, sollten Sie diesen Teil in ein externes Programm verlagern und eine sichere Schnittstelle zur Interaktion definieren. Auf diese Weise verringern Sie die Angriffsfläche.
5. Verwenden Sie beim Zugriff auf Dateien immer absolute Pfadangaben.
6. Denken Sie immer an Race Conditions. Sie können sich in Deadlocks manifestieren und das Programm anhalten. Im Zusammenhang mit Deadlocks sollten Sie beachten, daß Ihr Programm mehrmals auf dem System laufen kann und damit File Locking beim Zugriff auf Dateien Pflicht ist.

Die folgenden beiden Hinweise sind vornehmlich für Anwendungen mit erhöhten Sicherheitsanforderungen nützlich, allerdings auch auf alle anderen Anwendungen übertragbar.

Sicheres Öffnen von Dateien, die bereits existieren:

1. Prüfen Sie mit `lstat(2)` ob die angeforderte Datei existiert.
2. Prüfen Sie, ob es sich bei dem Pfad um einen symbolischen Link handelt und stoppen Sie die Ausführung gegebenenfalls.
3. Rufen Sie nun `open(2)` ohne das Flag `O_CREAT` auf, um herauszufinden, ob der Aufruf erfolgreich durchgeführt werden kann.
4. Führen Sie `fstat` auf den File Descriptor aus, der von `open(2)` zurückgegeben wurde.
5. Wenn die Felder `st_ino` und `st_dev` übereinstimmen, ist alles in Ordnung und wir können den File Descriptor für alle weiteren Operationen verwenden.

Sicheres Erstellen von Dateien:

1. Führen Sie `lstat` auf den angeforderten Pfad aus und prüfen Sie auf den Fehlercode `ENOENT`.
2. Rufen Sie nun `open(2)` mit den Flags `O_CREAT` und `O_EXCL` auf.

Kapitel 2

UNIX-Standards

The spirit, the will to win, and the will to excel are the things that endure. These qualities are so much more important than the events that occur.

VINCE LOMBARDI (1913 - 1970)

UNIX und dessen Programmierung sind Themenbereiche, die in den letzten Dekaden so umfangreich und tiefgehend behandelt worden, wie kaum andere. Viele exzellente, einflußreiche Werke sind im Laufe der Zeit entstanden, die trotz ihres Alters nicht an Bedeutung oder Gültigkeit verloren haben. Obwohl es lange Zeit nicht danach aussah, bildeten sich nach und nach Konsortien und Gremien, die sich mit der Standardisierung von UNIX als Plattform auseinandersetzten. Die Gründe für diese Bemühungen sind so unterschiedlich wie offensichtlich. Zum einen wurde der Kokurrenzdruck auf Hersteller von UNIX-Betriebssystemen, wie beispielsweise Sun (*SunOS*) oder AT&T (*UNIX System V*), um nur einige zu nennen, immer größer, zum anderen sind Systemhäuser nicht länger bereit, die unterschiedlichen Plattformen abzudecken, was nur mit sehr hohem Mehraufwand gelang und sich in hohen Kosten für Kunden niederschlug.

Es liegt der Gedanke nahe, daß UNIX gleich UNIX ist, oder zumindest eine enge Verwandschaft bestehen müsse. Das ist nicht ganz falsch. Doch betrachtet man die Geschichte von UNIX, so stellte sich in den Jahren 1984 bis 1986 eine explosionsartige Vervielfachung der UNIX-Derivate ein, also solcher Varianten, die zwar UNIX-ähnlich sind, aber nur bis zu einem gewissen Grad. Und genau da gingen die Probleme los. Nachdem die Universität von Kalifornien zu Berkeley ihre eigene UNIX-Implementierung mit der Bezeichnung Berkeley Software Distribution (BSD) im Jahre 1983 teilweise öffentlich zugänglich machte (genauer gesagt, ging es um die Version 4.2BSD), stand es anderen Herstellern frei, neue Derivate auf Basis von BSD zu entwickeln. Aus diesem Zweig der Evolution stammen viele heute sehr weit verbreitete und außerordentlich erfolgreiche Betriebssysteme. Zu dieser Zeit war der Quellcode zwar nicht öffentlich zugänglich, doch mit einer Lizenzgebühr ließ sich das Problem lösen. Das größte Interesse allerdings galt dem Netzwerk-Code, der unabhängig von AT&T's System V entwickelt wurde.

Kleine Namenskunde

Die Bezeichnung UNIX (vollständig groß geschrieben) ist eine Marke der X/Open Group, früher Austin Group, heute OpenGroup, und nennt nur solche Systeme, die von dem Markeninhaber zertifiziert wurden (übrigens war auch SCO mal Besitzer dieser Marke). Die Bezeichnung Unix (erster Buchstabe groß geschrieben) identifiziert ein System als UNIX-ähnlich. UNX-Systeme sind nur solche, die einer Zertifizierung der OpenGroup stand halten und die offizielle UNIX 98 Zertifikat erhalten. Dazu gehören alle Unices der größeren Hersteller wie etwa Solaris, AIX oder Tru64 UNIX. Die Open Source Unices stehen den „großen“ Brüdern in der Regel in nichts nach, doch so eine Zertifizierung möchte bezahlt werden, was für die Communities meist nicht zu schultern ist. Daher sind beispielsweise Linux und Open|Net|FreeBSD zwar jeweils Unix aber eben nicht UNIX (oder: UNIX 98).

Mit der Vielfalt begannen sich die Hersteller durch das Verändern der Codebasis und das Hinzufügen neuer Funktionen von einander zu entfernen, mit dem Resultat, daß es kaum noch möglich war, einen

gemeinsamen Nenner zu finden, auf dessen Basis, investitionssichere Softwareentwicklung stattfinden konnte. UNIX als ganzheitlicher Ansatz einer modernen Betriebs- und Entwicklungsplattform zerfiel langsam. Erst durch die Standardisierungsbemühungen kam Bewegung in die längst verloren geglaubte Debatte um einen einheitlichen Standard. Das war die Geburtstunde von POSIX, dem *Portable Operating System Interface*. Ein Begriff, den Richard Stallman, einer der einflußreichsten Open Source Aktivisten, geprägt hat.

Doch der Reihe nach. Im Jahre 1988 wurde die erste Version des POSIX-Standards mit der Bezeichnung IEEE Std 1003.11988 durch die IEEE (*Institute for Electrical and Electronics Engineers*) veröffentlicht. POSIX ist eigentlich kein einzelner Standard, sondern vielmehr eine Familie von Spezifikationen, die inzwischen nicht nur Programmierschnittstellen abdecken. Die Standardisierungsgremien ISO und *International Electrotechnical Commission* (IEC) haben sie aufgegriffen und ihren Entwicklungen angeglichen.

2.1 Was ist POSIX?

Der Originalstandard deckte nur einen kleinen Teil von UNIX ab. 1994 veröffentlichte die *X/Open Foundation* (heute *The Austin Group*) einen weitaus umfassenderen Standard basierend auf System V, bekannt als Spec 1170. Leider bestanden gravierende Unterschiede zwischen 1170 und POSIX, so daß sich die Hersteller schwer taten, beide zu implementieren.

1998 formte sich aus der X/Open Foundation die Austin Group, die von Mitgliedern der Open Group, der IEEE und des ISO/IEC Joint Technical Committee vertreten wurde. Ziel war es, die verschiedenen Standards zu kombinieren und zu einer umfassenden, vollständigen Spezifikation zusammenzufassen. Schließlich wurde 2001 ein gemeinsames Dokument von der IEEE und der Open Group ratifiziert und von der ISO/IEC 2002 bestätigt. Diese Spezifikation ist als *Single Unix Specification* oder *IEEE 1003.1-2001* bekannt und ist in der aktuellen Version 3 unter <http://www.opengroup.com> frei verfügbar. Wenn wir diesen Standard im Text ansprechen, werden wir ihn einfach POSIX nennen als SUS bezeichnen.

2.2 POSIX-Dokumente

Der erste Wurf (IEEE Std 1003.1988) war eine 317 Seiten lange Abhandlung der Programmierschnittstelle eines UNIX-ähnlichen Kernels (auch *Kernel Interface* genannt). Als Programmiersprache war C das Mittel der Wahl denn schließlich basieren alle modernen Kernel auf ihr. Das Werk schrieb vor, wie mit Prozessen gearbeitet wird (beispielsweise mit Hilfe der Systemaufrufe `exec(3)` oder `fork`), wie der Kontext für solche Prozesse auszusehen hat (wie etwa Benutzer- und Gruppen-IDs), wie Dateien und Ordner verarbeitet werden sollen, wie der Zugriff auf Konfigurationsdateien (*system databases*) geregelt wird und wie die Ein- und Ausgabe mit Terminals funktioniert.

Im zweiten Durchgang (**IEEE Std 1003.11990**, 358 Seiten) wurden nur geringe Änderungen vorgenommen, doch die Ergänzung des Titels um *Part 1: System Application Program Interface (API) /C Language/* ließ erkennen, daß um mehr ging als nur die bloße Beschreibung des Kernel Interfaces.

1992 (**IEEE Std 1003.21992**) nahm der Umfang der Spezifikationen fast um den Faktor vier (immerhin um die 1250 Seiten) zu und kam in zwei Ausgaben daher: Teil eins kennen sie bereits als IEEE Std 1003.11990. Teil 2 nannte sich *Part 2: Shell and Utilities*. Neben der Definition einer Standardshell (in diesem Fall die Bourne Shell) wurden auch über 100 wichtige Programme für das UNIX-System spezifiziert, darunter `sed/awk`, `hostname` und `yacc`.

Als wäre das nicht schon genug, folgte nur ein Jahr später **IEEE Std 1003.1b1993**, gemeinhin als IEEE P1003.4 bekannt. Dieses rund 500 Seiten starke Dokument lieferte einige Ergänzungen und machte wichtige Aussagen für Entwickler von POSIX-konformen Echtzeitsystemen. Im Detail wurden Ergänzungen aus den Bereichen Datei E/A, asynchrone E/A-Operationen, Speicherverwaltung, Einsatz von Semaphoren, Scheduling, Zeitgeber und Nachrichtenwarteschlangen vorgenommen.

1996 führte die IEEE die Standards zusammen und veröffentlichte sie unter dem Namen **IEEE Std 1003.11996**. Sie enthielt neben der Grundbeschreibung (1003.1199) die wichtigen Teilbereiche *PThreads*

(1003.1c1995) und Echtzeitbehandlung (1003.1b1993). Die ISO/IEC-Bezeichnung für dieses Dokument lautete 99451: 1996. Es bildet die Grundlage aller POSIX-konformen Entwicklungen und wird im Allgemeinen als POSIX.1 bezeichnet.

Wenn Sie mehr über POSIX, die Menschen hinter dem Projekt oder den Entwicklungsstand erfahren möchten, lohnt sich ein Besuch der Website des *Portable Applications Standards Committee* (<http://www.pasc.org/standing/sd11.html>).

2.3 Die Single UNIX Specification

Momentan ist die Single UNIX Specification 3 Issue 6 (SUSv3) das Maß der Dinge. Sie vereint die meisten Einzelstandards zu einer einheitlichen Spezifikation für Unix-Systeme. POSIX ist somit nur eine Untergruppe innerhalb der SUSv3, aber alle Erweiterungen des Standards sind gekennzeichnet, so daß wir genau unterscheiden können, welche Elemente Teil der ursprünglichen Spezifikation sind.

Die SUSv3 ist ein Produkt aus der Zusammenarbeit zwischen der IEEE und der OpenGroup. Damit ist die SUSv3 gleichermaßen ein IEEE-Standard und ein Standard der OpenGroup. Wie auch schon POSIX definiert die Spezifikation verschiedene Vorgaben für unterschiedliche Ebenen in einem System, wie beispielsweise für Systemschnittstellen, Shells und Werkzeuge. Für unsere Besprechungen sind lediglich die Systemschnittstellen von Bedeutung.

Die Standards, Spezifikationen, Empfehlungen und sonstige Dokumente, welche letztendlich über die Jahre die SUSv3 geformt haben sind vielfältig und haben großen Einfluß auf die Entwicklungslinien der Betriebssysteme gehabt. So existieren Abkürzungen und Begriffe wie XPG3, XPG4, CAE, XNS4 und XNS5, die alle irgendwelche Teile oder ein Ganzes eines bestimmten Standards implementieren. Bevor wir im nächsten Abschnitt einen genauen Blick auf die POSIX-Umgebung und die Feature-Test-Makros werfen, lohnt es sich die historisch wichtigsten Standards exemplarisch vor Augen zu führen. Dazu eignet sich Solaris 9 aus verschiedensten Gründen ausgezeichnet.

2.3.1 Solaris 9 und die Single UNIX Specification

Solaris 9 implementiert folgende Standards und Empfehlungen, deren Bedeutung und Ober-/Untergruppe gleich im Anschluß erläutert wird: X/Open Common Applications Environment (CAE) Portability Guide Issue 3 (XPG3) und Issue 4 (XPG4), Single UNIX Specification (SUS, auch bekannt als XPG4v2) und die Single UNIX Specification, Version 2 (SUSv2). Sowohl XPG4 als auch SUS schließen die Networking Services Issue 4 (XNS4) und Issue 5 (XNS5) ein.

Die folgende Liste ist eine Zeitleiste der Standards und deren Implementierungen in SunOS und Solaris:

X/Open CAE	Beschreibung	Release
XPG3	Obermenge von POSIX.1-1988 mit den Utilities von SVr3.	SunOS 4.1
XPG4	Obermenge von POSIX.1-1990, POSIX.2-1992 und POSIX.2a-1992 mit Erweiterungen zum POSIX-Standard von XPG3.	Solaris 2.4
XPG4v2	Erste SUS als Obermenge von XPG4 mit BSD-Schnittstellen, die bereits weit verbreitet sind.	Solaris 2.6
XNS4	Definiert die Sockets-API und XTI-Schnittstellen.	Solaris 2.6
SUSv2	Obermenge von XPG4v2 (SUSv1) mit Unterstützung von POSIX.1c-1996 und ISO/IEC 9899 (C Standard) Zusatz 1.	Solaris 7
XNS5	Obermenge eines sauberen LP64-Derivates von XNS4.	Solaris 7
SUSv3	Obermenge von IEEE Std 1003.1-1996, IEEE Std 1003.2-1992 und SUSv2.	Solaris 8

Tabelle 2.1: Standards in SunOS/Solaris

Die XNS4-Spezifikation ist nur für LP32-Umgebungen und XNS5 für 64-Bit-Umgebungen geeignet. Das heißt natürlich nicht, daß vor Solaris 7 keine saubere 64-Bit-Unterstützung vorhanden war, sondern sie nicht standardisiert wurde. Seit Solaris 7 ist Sun's Unix offiziell als UNIX 98 zertifiziert. Es unterstützt

seit Version 2.0 die System V Interface Definition 3 (Ausgaben 1 bis 3), doch seit sie von den POSIX- und CAE-Definitionen abweicht, gibt es Unterschiedliche Ansichten über die Sinnhaftigkeit einer solchen Unterstützung.

2.4 Unzulänglichkeiten von POSIX/SUSv3

Trotz aller Bemühungen, oder gerade deshalb, hat der POSIX-Standard neue Probleme geschaffen, die es Entwicklern nicht leichter machen, portablen Code zu entwickeln. Neben einigen anderen sind wohl `uname(3)` und die Behandlung von IPC-Bezeichnern, die mit `sem_open(3)`, `mq_open(3)` und `shm_open(3)` verwendet werden, am meisten diskutiert worden.

So dient `uname(3)` zur Identifikation des Systems, indem es die Struktur `utsname` mit Systembezeichnung, Version und ein paar andere Informationen zurückgibt; allerdings sind die Member von `utsname` nicht standardisiert, was eine zuverlässige Verwendung unmöglich macht.

Was die IPC-Bezeichner angeht, so spezifiziert POSIX lediglich, dass der Bezeichner mit den Regeln des Dateisystems zur Spezifizierung von Pfadangaben vereinbar sein und der Bezeichner mit einem Slash (/) beginnen muß. Beginnt er nicht mit einem Slash oder enthält weitere Slashes, ist das Verhalten implementierungsabhängig.

Das Fehlen von Präzision in der Formulierung des Standards führt nun dazu, daß beispielsweise Solaris 2.6 keine weiteren Slashes erlaubt und Tru64 UNIX sogar den Bezeichner im Dateisystem anlegt, was POSIX weder verlangt noch ausdrücklich verbietet. Der Bezeichner `/myqueue` beispielsweise wird von Solaris verarbeitet, Tru64 UNIX macht das auch, aber wir benötigen Schreibrechte im angegebenen Verzeichnis, nämlich Root (/). Die Verwendung von `/tmp/myqueue` ist auch keine Alternative, weil Solaris dieses Format im Gegensatz zu Tru64 UNIX nicht unterstützt.

Entwicklern bleibt im Kontext von IPC-Anwendungen also nichts anderes übrig, als Direktiven mittels `#define` zu verwenden um den Bezeichner abhängig vom System zu definieren.

Es gibt weitere Beispiele dieser Art, die uns zeigen, daß POSIX noch nicht der Heilige Gral der Standardentwicklung ist.

2.5 Die POSIX-Umgebung

Bevor ich auf die Programmierung POSIX-konformer Anwendungen zu sprechen kommen, richte ich zunächst einige Worte an Entwickler, die POSIX-konforme Bibliotheken, Compiler und Ausführungsumgebungen erstellen wollen oder auf solche umstellen möchten. Es ist wichtig, einige Grundregeln für die für die Erstellung eigener POSIX-Umgebungen zu beachten um Probleme, Inkompatibilitäten und unabsichtigte Abweichungen vom Standard zu vermeiden, die die Portabilität Ihrer Software einschränken könnten.

Ein POSIX-kompatibles System muß mindestens den Basisstandard implementieren. Viele der recht interessanten Aspekte von POSIX sind nicht Teil dieses Basisstandards, aber als Erweiterungen (*extensions*) definiert. Solche konformen Systeme definieren das Symbol `_POSIX_VERSION` als `200112L` im Header `<unistd.h>`. Frühere Versionen des Standards wurde durch den Wert `199506L` repräsentiert.

Fast alle Ausgaben des IEEE-Standards 1003.1-2001 definieren bestimmte Symbole in ihren Header-Dateien, wobei andere Symbole nicht Bestandteil des Standards sind und dadurch mit denen anderer Anwendungen kollidieren könnten. Des Weiteren ist es anderen Standards nicht erlaubt, die gleichen Symbolnamen zu verwenden, solange durch klare Regelungen der Sichtbarkeit eine Überschneidung verhindert wird.

Viele Symbole dienen als *Feature Test Macros* und kontrollieren die Sichtbarkeit der Symbole, die in einem Header eingefügt werden sollen. Es ist zu erwarten, daß die Zahl dieser Makros in Zukunft weiter zunehmen wird. Wichtig ist, daß während der Übersetzung (*compilation*) eines Programms, welches ein spezifiziertes Testmakro definiert (`#define`), kein Header vor der Definition eingefügt (`#include`) werden sollte. Das trifft auch auf implementierungsspezifische Header, die solche *Feature Test Macros* verwenden, zu. Wird diese Regel nicht beachtet ist das Verhalten undefined. Alle Feature Test Macros beginnen mit einem Unterstrich („_“).

2.5.1 Feature Test Macros

Anwendungen verwenden *Feature Test Macros* um anzugeben, daß sie von Merkmalen Gebrauch machen möchten, die über die Definition des C-Standards hinausgehen. Möchten sie nur Schnittstellen eines bestimmten Standards nutzen, genügt es in der Regel die Feature Test Macros des jeweiligen Standards zu aktivieren.

2.5.1.1 Das `_POSIX_SOURCE` Feature Test Macro

Wenn wir diese Makro definieren, wird die gesamte Funktionalität des POSIX.1-Standards (IEEE Standard 1003.1) und die von ISO C aktiviert. Allerdings verliert die Definition jede Bedeutung wenn `_POSIX_C_SOURCE` mit einem positiven Integerwert definiert ist.

2.5.1.2 Das `_POSIX_C_SOURCE` Feature Test Macro

Eine POSIX-konforme Anwendung sollte sicherstellen, das das Feature Test Macro `_POSIX_C_SOURCE` definiert ist, bevor ein Header eingefügt wird. POSIX.1-1990 definierte `_POSIX_SOURCE`, welches mit POSIX.1b-1993 durch `_POSIX_C_SOURCE` ersetzt wurde. Verwendet eine Applikation einen IEEE 1003.1-Header (SUSv3) und ist `_POSIX_C_SOURCE` mit dem Wert 200112L definiert, so sind alle in SUSv3 definierten Symbole sichtbar, sobald ein solcher Header eingefügt wird. Zulässige aber nicht notwendige Symbole dürfen ebenfalls sichtbar sein.

Eigentlich müssen die Werte nicht immer ganz exakt definieren, vielmehr müssen wir auf folgende Regeln bei der Verwendung von `_POSIX_C_SOURCE` achten: Sobald das Makro mit einem positiven Integer definiert wurde, sind die minimalen POSIX-Funktionalitäten grundsätzlich in Kraft. Je höher der Wert, desto mehr Funktionalität erhalten wir. Ist der Wert ≥ 1 ist lediglich IEEE 1003.1-1990 (POSIX.1) verfügbar. Ist der Wert ≥ 2 erhalten wir Zugang zu den Funktionen von IEEE 1003.2-1992 (POSIX.2). Werte bis 199309L schalten POSIX.1b (IEEE 1003.1b-1993) und Werte bis 199506L schalten POSIX.1c (IEEE 1003.1-1996) ein. Momentan können wir alle Funktionen mit einem Wert von 200112L (IEEE 1003.1-2001) aktivieren. Höhere Werte schalten zusätzliche Erweiterungen in zukünftigen Standards ein und sollten vermieden werden.

Folgende Tabelle listet die Test Macros und den jeweiligen Standard auf.

Standard	Feature Test Macros
POSIX.1-1990	<code>_POSIX_SOURCE</code>
POSIX.1-1990, POSIX.2-1992	<code>_POSIX_SOURCE</code> und <code>_POSIX_C_SOURCE=2</code>
POSIX.1b-1993	<code>_POSIX_C_SOURCE=199309L</code>
POSIX.1c-1996	<code>_POSIX_C_SOURCE=199506L</code>
IEEE Std 1003.1-1996	<code>_POSIX_C_SOURCE=200112L</code>

Tabelle 2.2: POSIX Standards und Feature Tests

Listing 2.1: Das POSIX-Feature Test Makro

```

1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main(void) {
6
7 #ifdef _POSIX_SOURCE
8     printf("_POSIX_SOURCE = %ld\n", (long)_POSIX_SOURCE);
9 #endif
10 #ifdef _POSIX_SOURCE

```

```

11     printf("_POSIX_C_SOURCE = %ld\n", (long)_POSIX_C_SOURCE);
12 #endif
13 #ifdef _POSIX_SOURCE
14     printf("_POSIX_VERSION = %ld\n", (long)_POSIX_VERSION);
15 #endif
16
17     return EXIT_SUCCESS;
18 }

```

Listing 2.1: xcode/posix-test.c - Das POSIX-Feature Test Makro

Um zu erfahren, welche Werte ein System liefert, definieren wir beim Übersetzen `_POSIX_SOURCE`:

```

% cc -D_POSIX_SOURCE -o posix-test posix-test
% ./posix-test
_POSIX_SOURCE = 1
_POSIX_C_SOURCE = 198808
_POSIX_VERSION = 200112

```

Wie wir sehen können unterstützt unser System IEEE 1003.1-2001 (`_POSIX_VERSION = 200112`).

Alle Bezeichner des IEEE 1003.1-2001 Standards dürfen nur durch eine `#undef`-Direktive deaktiviert werden. Diese Direktiven sollten allen `#include`-Direktiven eines Headers folgen.

2.5.1.3 Das `_XOPEN_SOURCE` Feature Test Macro

Die OpenGroup (*Austin Group*) hat eigene Erweiterungen definiert, die als *XSI* gekennzeichnet sind (*XSI* steht für *X/Open System Interface*). Eine Anwendung, die diese Erweiterungen berücksichtigen möchte, muß das Feature Test Macro `_XOPEN_SOURCE` auf den Wert 600 festlegen, bevor ein Header eingefügt wird. Nur so wird sichergestellt, daß die Funktionalitäten von `_POSIX_C_SOURCE` und die Erweiterungen aktiviert werden. Das liegt daran, daß diese Ausgabe des Standards mit dem ISO C Standard abgestimmt ist und die Funktionalität von `_POSIX_C_SOURCE` auch durch `_XOPEN_SOURCE` abgedeckt werden. Wird es definiert, so werden alle Funktionalitäten der XPG-Serie aktiviert. Die XPG-Serie ist eine Obermenge von POSIX.1 und POSIX.2. Tatsächlich bewirkt `_XOPEN_SOURCE` eine automatische Definition der beiden Makros `_POSIX_C_SOURCE` und `_POSIX_SOURCE`.

Neben `_XOPEN_SOURCE` existiert auch `_XOPEN_SOURCE_EXTENDED`. Definieren wir es, stehen uns alle Funktionen zur Verfügung, die mit `_XOPEN_SOURCE` definiert werden und mehr. Diese Tatsache ist aber nicht immer nützlich, da es auch Funktionen sichtbar macht, die z.T. aus Sicherheitsgründen nicht mehr verwendet werden sollten. Beispielsweise wird `getwd(3)` auf vielen Systemen erst sichtbar, wenn `_XOPEN_SOURCE_EXTENDED` definiert wurde. Welche Prototypen beim Definieren des Makros sichtbar werden ist in jedem Fall implementierungsabhängig.

2.5.2 Namensräume

Alle Bezeichner in IEEE 1003.1-2001 sind, mit Ausnahme von `environ`, in mindestens einem der Header definiert. Sind `_POSIX_C_SOURCE` oder `_XOPEN_SOURCE` definiert, deklariert oder definiert jeder Header Bezeichner, die eventuell mit benutzerdefinierten Bezeichnern der Applikation kollidieren. Der Satz der Bezeichner, der für die Anwendung sichtbar ist, besteht aus genau denen aus den eingefügten Headern und zusätzlichen Bezeichnern, die für die Implementierung reserviert sind.

Es steht Implementierungen frei, eigene Member einer Struktur oder Union hinzuzufügen, ohne die Sichtbarkeit dieser Member mit einem Feature Test Macro zu überwachen, solange kein benutzerdefiniertes Makro gleichen Namens mit der korrekten Interpretation des Programms kollidiert. Einige Bezeichner sind für die Verwendung in Implementierungen reserviert. Das bewahrt unsere Applikationen vor sog. Namensraumverschmutzung (*namespace pollution*).

Bei Verwendung eines POSIX- und Standard-C-Headers ist sichergestellt, dass jedes vom Standard definierte Symbol mit einem gültigen Wert definiert ist.

Es ist selbstverständlich, daß die Präfixe `posix_`, `POSIX_` und `_POSIX_` ausschließlich für den Einsatz innerhalb des POSIX-Standards der IEEE reserviert sind.

2.5.3 Fehlerkennziffern

Die meisten Funktionen sind in der Lage, Fehlerkennziffern (*error numbers*) zurückzuliefern. Wie das geschieht und welche Ziffer bei welchem Fehler zurückgeliefert wird, kann der Funktionsbeschreibung, beispielsweise der Manpage, entnommen werden.

Einige Funktionen erlauben die Abfrage der Fehlerkennziffer durch das spezielle Symbol `errno`. Es ist in dem Header `<errno.h>` definiert und wird zu einem veränderbaren *lvalue* vom Typ `int` erweitert (*expanded*). Die POSIX-Spezifikation überläßt es der Implementierung, ob `errno` als Makro oder als Funktion bereitgestellt wird.

Sofern die Funktionsbeschreibung aussagt, daß `errno` zur Fehleranzeige verwendet wird, darf es ohne weiteres ausgelesen werden, denn es ist zum einen ein gültiger Rückgabewert der jeweiligen Funktion und zum anderen wird sie `errno` niemals auf den Wert 0 setzen. POSIX-konforme Implementierungen vermeiden es auch, daß Threads eines Prozesses die `errno`-Variable eines anderen Threads setzen können, oder durch Funktionsaufrufe beeinflußt werden.

Alle Funktionen, die einen Rückgabewert von 0 zurückliefern können, zeigen dadurch den Erfolg der durchgeführten Operation an. Tritt mehr als ein Fehler bei der Verarbeitung eines Funktionsaufrufes auf, kann jeder dieser Fehler als Rückgabewert in Frage kommen, da die Reihenfolge der Erkennung nicht spezifiziert ist.

Weiter unten finden Sie eine Liste der POSIX-Fehlerkennziffern. Implementierungen dürfen jederzeit weitere Kennziffern hinzufügen und Fehlerkennziffern zurückgeben, die von der beschriebenen Vorgehensweise abweichen. Die meisten Manpages der Funktionsbeschreibungen liefern eine ERRORS-Sektion, die eine genaue Aufschlüsselung der Fehlerquellen und der Kennziffern liefert. Andererseits dürfen POSIX-Implementierungen die Kennziffern nicht ändern.

Übrigens gibt es eine einfache Möglichkeit, sich alle Fehlerkennziffern und die dazugehörigen Meldungen eines Systems anzeigen zu lassen. Das folgende kleine Tool erfüllt genau diese Aufgabe.

Listing 2.2: Ausgabe der Fehlerkennziffern und der passenden Meldungen

```

1 #define _ALL_SOURCE
2
3 #include <stdio.h>
4 #include <errno.h> /* for _MAX_ERRNO */
5
6 int main(void) {
7     int i;
8     extern int errno;
9
10    for (i = 0; i < _MAX_ERRNO; i++) {
11        fprintf(stderr, "%3d", i);
12        errno = i;
13        perror(" ");
14    }
15
16    return (0);
17 }
```

Listing 2.2: `xcode/printerrors.c` - Ausgabe der Fehlerkennziffern und der passenden Meldungen.

Die folgenden symbolischen Namen identifizieren die zulässigen Fehlerkennziffern im jeweiligen Kontext der Funktionen. In Programmen sollten nur diese symbolischen Namen verwendet werden, da sie je nach Implementierung abweichen können. Alle Konstanten sind in der Header-Datei `<errno.h>` hinterlegt und dürfen nur dort ergänzt werden. Die eigentlichen Werte werden nicht durch den POSIX-Standard festgelegt.

E2BIG (*Argument list too long*)

Argumentliste ist zu lang. Die Summe der Bytes, die für die Argumentliste des neuen Process Images genutzt werden, ist zu lang. Sie überschreitet die Maximallänge von `{ARG_MAX}` Bytes. Eine andere Möglichkeit ist die Überschreitung des zur Verfügung stehenden Ausgabepuffers (*output buffers*) oder das Argument ist größer als das vorgeschriebene Maximum für dieses System.

EACCES (*Permission denied*)

Zugriff verweigert. Es wurde versucht, auf eine Datei zuzugreifen, wobei deren Zugriffsrechte nicht beachtet wurden.

EADDRINUSE (*Address in use*)

Adresse ist bereits in Gebrauch. Die angegebene Adresse im Speicher ist bereits in Gebrauch. Tritt beispielsweise auf, wenn `mmap` mit einer Adresse aufgerufen wird, die bereits von einem anderen Prozess oder Thread verwendet wird.

EADDRNOTAVAIL (*Address not available*)

Die Adresse ist nicht verfügbar. Die angegebene Adresse im Speicher ist nicht auf dem lokalen System verfügbar.

EAFNOSUPPORT (*Address family not supported*)

Die Adressfamilie wird nicht unterstützt. Die Implementierung unterstützt die angegebene Adressfamilie nicht, oder die angegebene Adresse ist keine gültige Adresse für diese Adressfamilie des spezifizierten Sockets. Dieser Fehler tritt üblicherweise auf, wenn beispielsweise ein `AF_INET` Socket mit einer OSI-Adresse ausgestattet werden soll.

EAGAIN (*Resource temporarily unavailable*)

Die Resource ist momentan nicht verfügbar. Hierbei handelt es sich um einen vorübergehenden Zustand, so daß spätere Zugriffe derselben Routine auf die Resource durchaus erfolgreich sein können.

EALREADY (*Connection already in progress*)

Verbindung wird bereits hergestellt. Für den Spezifizierten Socket wird bereits eine Verbindungsanfrage bearbeitet. Einige Funktionen, wie etwa `connect(2)` erfordern oftmals etwas mehr Zeit als gewöhnlich notwendig und geben sofort den Wert `EINPROGRESS` zurück, um Sie darüber zu informieren, daß die Anfrage bearbeitet wird, es aber erforderlich sein kann, erneut anzufragen, um den Vorgang abzuschließen. Wenn Sie versuchen, mit dem Socket zu arbeiten, bevor die Anforderung abgeschlossen wurde, erhalten Sie in der Regel einen `EALREADY`-Fehler. Das ist in der Regel ein Anzeichen für einen Bug in Ihrem Programm.

EBADF (*Bad file descriptor*)

Ungültiger File Descriptor. Der angegebene File Descriptor ist entweder außerhalb des gültigen Bereichs, referenziert eine noch nicht geöffnete Datei, oder eine read/write-Anfrage wird für eine Datei angefordert, die nur für schreiben oder lesen geöffnet ist.

EBADMSG (*Bad message*)

Ungültige Nachricht. Während eines Aufrufs von `read(2)`, `getmsg`, `getpmsg` oder `ioctl(2)` mit `I_RECVFD` auf ein STREAM-Gerät wurde eine Nachricht zurückgeliefert, die für ungültig für die empfangende Funktion ist. Im Fall von `read(2)` ist die erhaltene Nachricht keine Datennachricht. Wenn beim Aufruf von `getmsg` oder `getpmsg` ein File Descriptor zurückgeliefert wurde und keine Kontrollnachricht, tritt `EBADMESSAGE` auf. Ganz ähnlich verhält sich `ioctl(2)` für das Kommand `I_RECVFD` aufgerufen wurde, aber Kontroll- oder Dateninformationen anstelle eines File Descriptors zurückgegeben wurden. In allen anderen Fällen deutet der Fehler auf eine beschädigte Nachricht hin.

EBUSY (*Resource busy*) Die angeforderte Resource ist belegt. Es wurde versucht, eine Resource zuzugreifen, die momentan nicht verfügbar ist, beispielsweise wird sie gerade von einem anderen Prozess verwendet und würde bei gleichzeitigem Zugriff zu einem Konflikt führen.

ECANCELED (*Operation canceled*)

Vorgang abgebrochen. Eine asynchrone Operation wurde vor ihrer Fertigstellung abgebrochen.

ECHILD (*No child process*)

Kein Kindsprozess. Die Funktionen `wait(2)` oder `waitpid(2)` wurden von einem Prozess aufgerufen, der weder einen Kindsprozess aufweist oder auf einen solchen wartet.

ECONNABORTED (*Connection aborted*)

Verbindung abgebrochen. Die Verbindung wurde abgebrochen.

ECONNREFUSED

Verbindungsanfrage verweigert (*Connection refused*). Es wurde versucht, eine Verbindung zu einem Socket herzustellen, schlug aber fehl, weil kein Prozess mit diesem Socket assoziiert ist und auf Verbindungsanfragen wartet. Es ist auch möglich, daß die Warteschlange (*queue*) für Verbindungsanfragen bereits voll ist und das zugrundeliegende Protokoll keine Retransmissionen unterstützt.

ECONNRESET (*Connection reset*)

Verbindng zurückgesetzt. Die Verbindung mußte durch die Gegenstelle (*peer*) zurückgesetzt werden.

EDEADLK (*Resource deadlock would occur*)

Ein Deadlock auf eine Resource würde auftreten. Es wurde ein Versuch unternommen, eine Resource zu sperren, so daß ein Deadlock auf die Resource entstehen würde.

EDESTADDRREQ (*Destination address required*)

Zieladresse erforderlich. Es konnte keine Adresse der Gegenstelle angebunden werden.

EDOM (*Domain error*)

Domain-Fehler. Das Eingabeargument ist außerhalb der definierten Domäne der mathematischen Funktion. Eine Definition aus dem ISO C Standard.

EDQUOT

Reserviert.

EEXIST (*File exists*)

Datei existiert bereits. Es wurde eine bereits existierende Datei in einem unangemessenen Kontext referenziert. Beispielsweise würde dieser Fehler beim Aufruf von `link` zur Erzeugung eines Links auftreten, wenn bereits ein solcher oder eine gleichnamige Datei existiert.

EFAULT (*Bad address*)

Ungültige Adresse. Das System hat die Verwendung einer ungültigen Adresse als Parameter einer Funktion entdeckt. Es ist nicht garantiert, daß die Aufdeckung solcher logischer Fehler zuverlässig funktioniert. Kann der Fehler nicht entdeckt werden, könnte ein entsprechendes Signal erzeugt werden, daß die Zugriffsverletzung anzeigt.

EFBIG (*File too large*)

Datei zu groß. Die Größe einer Datei überschreitet das Maximum der Implementierung oder den zulässigen Offset, der dem File Descriptor zugeordnet ist.

EHOSTUNREACH (*Host is unreachable*)

Host nicht erreichbar. Das Zielsystem kann nicht erreicht werden. Beispielsweise könnte das Netzwerk beschädigt sein, weil ein Router oder ein entfernter Host nicht erreichbar ist.

EIDRM (*Identifier removed*)

Bezeichner wurde entfernt. Wird während einer XSI Interprozesskommunikation zurückgeliefert, wenn ein Bezeichner von dem System entfernt wurde.

EILSEQ (*Illegal byte sequence*)

Illegalle Bytesequenz. Ein Wide Character Code, der keinem gültigen Zeichen entspricht, wurde entdeckt oder eine Bytesequenz, die keinem gültigen Wide Character Code entspricht. Eine Definition des ISO C Standards.

EINPROGRESS (*Operation in progress*)

Vorgang in Bearbeitung. Zeigt an, daß eine asynchrone Operation noch nicht abgeschlossen wurde oder das `O_NONBLOCK` für einen Socket File Descriptor gesetzt wurde und die Verbindung nicht sofort etabliert werden kann.

EINTR (Interrupted function call)

Funktionsaufruf unterbrochen. Ein asynchrones Signal wurde von einem Prozess während der Ausführung einer unterbrechbaren Funktion empfangen. Wenn der Signal Handler normal zurückkehrt, kann der unterbrochene Funktionsaufruf dies anzeigen (zurückliefern).

EINVAL (Invalid argument)

Ungültiges Argument. Es wurde ein ungültiges Argument übergeben. Dieser Fehler tritt beispielsweise auf, wenn ein undefiniertes Signal an die Funktion `signal(2)` oder `kill(2)` übergeben wird.

EIO (Input/output error)

Eingabe-/Ausgabefehler. Es ist ein physikalischer Fehler bei der Ein- oder Ausgabe aufgetreten. Das passiert z.B. wenn zwei mal aufeinander folgend der gleiche File Descriptor für eine E/A-Operation verwendet wird. Wird eine andere Operation mit dem gleichen File Descriptor durchgeführt und ein Fehler angezeigt, so geht die ursprüngliche Fehleranzeige (also EIO) verloren.

EISCONN (Socket is connected)

Socket ist bereits verbunden. Der angegebene Socket ist bereits verbunden.

EISDIR (Is a directory)

Es ist ein Verzeichnis. Es wurde der Versuch unternommen, ein Verzeichnis im Schreibmodus zu öffnen.

ELOOP (Symbolic link loop)

Loop durch symbolische Links. Es wurde ein Loop zwischen symbolischen Links bei der Pfadauflösung entdeckt. Der Fehler tritt auch auf, wenn mehr als {SYMLOOP_MAX} symbolische Links während der Pfadauflösung auftauchen.

EMFILE (Too many open files)

Zu viele offene Dateien. Es wurde der Versuch unternommen, mehr Dateien zu öffnen, als File Descriptors für diesen Prozess zulässig sind.

EMLINK (Too many links)

Zu viele Verweise. Es wurde der Versuch unternommen, die maximale Anzahl von Verweisen (links) auf eine Datei zu überschreiten. Spezifiziert durch {LINK_MAX}.

EMSGSIZE (Message too large oder Inappropriate message buffer length)

Nachricht zu groß oder Ungültige Größe des Message Buffers. Eine Nachricht, die an einen Transport Provider gerichtet ist, überschritt die Größe des internen Nachrichtenpuffers (internal message buffer) oder eines anderen Netzwerklimits.

EMULTIHOP

Reserviert.

Mir ist nicht bekannt, wie viele Betriebssysteme (Symbian 9.x ist eines) diese Konstante einsetzen und welche Fehler im Detail damit behandelt werden. Die folgenden Beschreibungen sind deshalb mit Vorsicht zu genießen:

Teile eines Pfades erfordern mehrere Hops über mehrere entfernte Systeme, aber das Dateisystem unterstützt dies nicht. Des Weiteren kann der Fehler auftreten, wenn Benutzer versuchen, auf entfernte Ressourcen zuzugreifen, die nicht direkt verfügbar sind.

ENAMETOOLONG (Filename too long)

Dateiname zu lang. Die Länge eines Pfades überschreitet die Grenze von PATH_MAX oder eine Pfadanteil ist länger als NAME_MAX. Dieser Fehler kann auch auftreten, wenn Pfadnamensubstitution durch einen symbolischen Link auftritt, der einen längeren Pfad als {PATH_MAX} als Resultat aufweist.

ENETDOWN (Network is down)

Netzwerk ist nicht in Betrieb. Die lokale Netzwerkschnittstelle, die für das Erreichen der Gegenstelle verwendet werden soll, ist nicht in Betrieb (*down*).

ENETRESET (The connection was aborted by the network)

Die Verbindung wurde durch das Netzwerk abgebrochen.

ENETUNREACH (*Network unreachable*)

Netzwerk nicht erreichbar. Es existiert keine Route zum Netzwerk.

ENFILE (*Too many files open in system*)

Zu viele offene Dateien im System. Das System hat sein vordefiniertes Maximum an gleichzeitig geöffneten Dateien erreicht und kann zeitweise kein weiteren Anfragen in dieser Hinsicht erfüllen.

ENOBUFS (*No buffer space available*)

Kein Speicher für den Puffer verfügbar. Es wurden nicht ausreichend Ressourcen für einen Buffer zur Durchführung der Socket-Operation bereitgestellt. Das bedeutet in der Regel, daß das System nicht mehr über genügend freien Speicher verfügt.

ENODATA

Keine Nachricht verfügbar. Es ist keine Nachricht im STREAM-Kopf (*head*) in der Lese-Warteschlange verfügbar.

ENODEV (*No message available*)

Kein solches Gerät (*No such device*). Es wurde der Versuch unternommen, eine unpassende Funktion auf ein Gerät anzuwenden. Beispielsweise beim Versuch auf ein Nue-Lesen-Gerät zu schreiben.

ENOENT (*No such file or directory*)

Ungültiger Pfad oder Dateiname. Ein Teil eines Pfades existiert nicht oder der Pfadname ist ein Leerstring.

ENOEXEC (*Executable file format error*)

Ungültiges Format für ausführbare Dateien. Es wurde der Versuch unternommen eine Datei auszuführen, die zwar über die geeigneten Rechte verfügt, aber nicht in dem für die Implementierung korrekten Format zur Ausführung von Dateien vorliegt.

ENOLCK (*No locks available*)

Keine Locks verfügbar. Es wurde der systemspezifische Grenzwert für gleichzeitig geöffnete Dateien und Rekord Locks erreicht, so daß momentan keine weiteren zur Verfügung stehen.

ENOLINK

Reserviert. Meist deutet der Fehler auf Probleme mit der Gegenstelle hin.

ENOMEM (*Not enough space*)

Nicht genug Speicher. Das neue Process Image benötigt mehr Speicherplatz, als die Hardware unterstützt oder das Speichermanagement des Systems zuläßt.

ENOMSG (*No message of the desired type*)

Keine Nachricht diesen Typs. Die Nachrichtenwarteschlange enthält keine Nachricht dieses Typs. Tritt während der XSI Interprozesskommunikation auf.

ENOPROTOOPT (*Protocol not available*)

Protokoll nicht verfügbar. Die spezifizierte Protokolloption, die `setsockopt(2)` übergeben wurde, wird von der Implementierung nicht unterstützt.

ENOSPC (*No space left on a device*)

Kein Platz auf einem Gerät verfügbar. Es steht nicht genügend Speicherplatz auf dem Medium zur Verfügung. Tritt auf, wenn der Speicherplatz während einer Schreiboperation mit `write(2)` der Speicherplatz ausgeht.

ENOSR (*No STREAM resources*)

Keine STREAM Ressourcen. Es stehen nicht genügend Ressourcen für die Ausführung von STREAM-verwandten Funktionen zur Verfügung. Dies ist ein temporärer Zustand und kann sich ändern, sobald andere Prozesse ihre Ressourcen wieder freigeben.

ENOSTR (*Not a STREAM*)

Kein STREAM. Es wurde der Versuch unternommen, eine STREAM-Funktion auf einen File Descriptor anzuwenden, der nicht mit einem Stream-Gerät assoziiert ist.

ENOSYS (*Function not implemented*)

Funktion nicht implementiert. Es wurde der Versuch unternommen, eine Funktion auszuführen, die in dieser Implementierung nicht zur Verfügung steht.

ENOTCONN (*Socket not connected*)

Socket ist nicht verbunden. Der Socket ist nicht verbunden. Tritt auf, wenn der Socket mit einem verbindungsorientierten Protokoll assoziiert ist, aber nicht mit dem Endpunkt verbunden ist. Deutet meistens auf ein Problem in der Anwendung hin.

ENOTDIR (*Not a directory*)

Kein Verzeichnis. Ein Teil eines spezifizierten Pfades existiert, ist aber kein Verzeichnis, obwohl eins erwartet wurde.

ENOTEMPTY (*Directory not empty*)

Verzeichnis nicht leer. Es wurde zwar ein Verzeichnis spezifiziert, welches nicht leer war, obwohl ein solches erwartet wurde.

ENOTSOCK (*Not a socket*)

Kein Socket. Der angegebene File Descriptor referenziert keinen Socket.

ENOTSUP (*Not supported*)

Nicht unterstützt. Die Implementierung unterstützt dieses Merkmal der *Realtime Option Group* nicht.

ENOTTY (*Inappropriate I/O control operation*)

Unpassende E/A-Kontrolloperation. Es wurde der Versuch unternommen, eine Kontrollfunktion für eine reguläre Datei oder Gerätedatei auszuführen, für die diese Kontrollfunktion ungültig ist.

ENXIO (*No such device or address*)

Kein solche Datei oder Adresse. Ein- oder Ausgaben für eine Gerätedatei sind für ein Gerät bestimmt, das nicht existiert. Oder es wurde eine Anfrage gestellt, die über die Fähigkeiten des Geräts hinaus gehen. Der Fehler tritt beispielsweise auf, wenn ein Bandlaufwerk nicht online ist.

EOPNOTSUPP (*Operation not supported on socket*)

Vorgang für Sockets nicht erlaubt. Der Sockettyp (entweder Adressfamilie oder das Protokoll) unterstützt diese Operation nicht.

EOVERFLOW (*Value too large to be stored in data type*)

Wert zu groß für diesen Datentyp. Es wurde der Versuch unternommen, eine Operation durchzuführen, die als Resultat einen Datentyp hervorbringt, der größer als der darstellbare Maximalwert eines Datentyps ist.

EPERM (*Operation not permitted*)

Vorgang nicht zulässig. Es wurde der Versuch unternommen, eine Operation durchzuführen, die nicht den zulässigen Rechten eines Prozesses oder Benutzers entsprach.

EPIPE (*Broken pipe*)

Pipe unterbrochen. Es wurde versucht, eine Schreiboperation auf einen Socket, eine Pipe, oder einer FIFO-Warteschlange auszuführen, für die kein Prozess zum Auslesen zugeordnet ist.

EPROTO (*Protocol error*)

Protokollfehler. Es ist ein allgemeiner Protokollfehler aufgetreten. Dieser Fehler ist zwar an ein Gerät gebunden, deutet aber nicht auf Hardwarefehler hin.

EPROTONOSUPPORT (*Protocol not supported*)

Protokoll wird nicht unterstützt. Das Protokoll wird nicht von der angegebenen Adressfamilie oder nicht von der Implementierung unterstützt.

EPROTOTYPE (*Protocol wrong type for socket*)

Falsches Protokoll für diesen Socket. Der aktuelle Socket-Typ wird von dem spezifizierten Protokoll nicht unterstützt.

ERANGE (*Result too large or too small*)

Ergebnis zu groß oder zu klein. Das Ergebnis der Funktion ist zu groß (*overflow*) oder zu klein (*underflow*) um in dem verfügbaren Raum dargestellt zu werden. Definiert durch den ISO C Standard.

EROFS (*Read-only file system*)

Vom Dateisystem kann nur gelesen werden. Es wurde der Versuch unternommen, eine Datei oder ein Verzeichnis auf einem Dateisystem zu modifizieren, vom dem nur gelesen werden kann.

ESPIPE (*Invalid seek*)

Ungültiger Suchvorgang. Es wurde der Versuch unternommen, einen Offset für eine Pipe oder einen FIFO-Warteschlange zu setzen.

ESRCH (*No such process*)

Kein solcher Prozess. Es konnte kein Prozess mit der angegebenen Prozess-ID gefunden werden.

ESTALE

Reserviert.

ETIME (*Timeout*)

Timeout für den Aufruf von STREAM `ioctl(2)` (*STREAM ioctl timeout*). Der gesetzte Timer für einen Aufruf von `ioctl(2)` für STREAMs ist abgelaufen. Dieser Fehler ist geräteabhängig und könnte auf einen Hardware- oder Softwarefehler hindeuten. Alternativ kann der Timeout auch einfach nur zu kurz für die angeforderte Operation sein.

ETIMEDOUT (*Connection timed out*)

Verbindungzeit abgelaufen. Die zulässige Gesamtdauer für eine Verbindung ist abgelaufen. Wenn die Verbindung während der Ausführung einer Funktion, die diesen Fehler übermittelt hat, ist es nicht klar, ob die Funktion vollständig oder nur Teile des gewünschten Vorgangs abgeschlossen hat.

ETXTBSY (*Text file busy*)

Textdatei wird verwendet. Es wurde der Versuch unternommen, in eine Datei zuschreiben, die momentan ausgeführt wird oder eine Datei auszuführen, die momentan für das Schreiben geöffnet ist.

EWOULDBLOCK (*Operation would block*)

Vorgang würde blockieren. Eine Operation auf einem Socket, die als non-blocking (nicht blockierend) gekennzeichnet ist, würde blockieren. Das passiert beispielsweise, wenn keine Daten verfügbar sein, die Funktion aber eigentlich die Ausführungs aussetzen würde. Eine konforme Implementierung kann EWOULDBLOCK und EAGAIN die gleichen Werte zuweisen.

EXDEV (*Operation would block*)

Nicht korrekter Link. Es wurde versucht einen Hardlink auf eine Datei in einem anderen Dateisystem zu erzeugen.

2.5.4 POSIX Datentypen

POSIX definiert viele Datentypen, die für spezielle Verwendungszwecke in Funktionen vorgesehen sind. In der Regel handelt es sich dabei um `typedefs` konventioneller Datentypen. Sie wurde eingeführt, um eine möglichst große plattformunabhängigkeit sicherzustellen und gleichzeitig den Entwicklern die Arbeit zu erleichtern.

Die folgende Liste von Datentypen ist nicht vollständig, zeigt aber die wichtigsten. Implementierungen führen oft eigene Datentypen hinzu, deren Dokumentation der Funktionsbeschreibung zu entnehmen ist.

cc_t

Typ für Steuerzeichen von Terminals.

clock_t

Integer oder Float für Messung und Darstellung von Prozessorzeiten. Definiert durch den ISO C Standard.

clockid_t

Für bestimmte Timer zur Identifikation von *Clock IDs*.

dev_t

Arithmetischer Typ (z.B. Integer) für Gerätenummern.

DIR

Typ zur Darstellung eines Verzeichnis-Stroms (*directory stream*).

div_t

Struktur, die von der Funktion **div** zurückgegeben wird.

FILE

Struktur mit Informationen über eine Datei.

glob_t

Struktur, die für die Mustererkennung (*pattern matching*) in Pfaden verwendet wird.

fpos_t

Enthält Information über die Position des Cursors in einer Datei.

gid_t

Integer, der für Gruppen-IDs verwendet wird.

iconv_t

Wird für Konvertierungs-Descriptoren verwendet und von **iconv_open** zurückgeliefert.

id_t

Integer, der für allgemeine IDs Verwendung findet. Kann in jedem Fall mindestens den größten Wert von **pid_t**, **uid_t** oder **gid_t** aufnehmen.

ino_t

Vorzeichenloser Integer für Inodes.

key_t

Arithmetischer Typ für die XSI Interprozesskommunikation.

ldiv_t

Struktur, der von **ldiv** zurückgegeben wird.

mode_t

Integer zur Beschreibung von Dateiattributen.

mqd_t

Für die Verwendung von *Message Queue Descriptors*¹.

nfds_t

Integer für die Verwaltung der Anzahl von File Descriptors.

nlink_t

Integer für das Zählen von Links.

off_t

Vorzeichenbehafteter Integer zur Beschreibung von Dateigrößen.

pid_t

Vorzeichenbehafteter Integer für die Verwaltung von Prozess- und Prozessgruppen-IDs.

pthread_attr_t

Wird für die Verwaltung von Objekten für Thread-Attribute (*thread attribute object*) benötigt.

¹Die deutsche Übersetzung gefällt mir nicht: Nachrichtenwarteschlangen-Descriptoren. Daher bleibe ich bei dem englischen Begriff

pthread_cond_t

Wird für die Bedingungsvariablen (*condition variables*) in der Threadbehandlung eingesetzt.

pthread_condattr_t

Identifiziert ein Objekt eines Bedingungsattributs (*condition attribute object*).

pthread_key_t

Wird für die Verwaltung von Thread-spezifischen Datenschlüsseln verwendet.

pthread_mutex_t

Nur für den Einsatz von Mutexen (*mutexes*).

pthread_mutexattr_t

Wird für die Identifizierung eines Mutexattributobjekts (*mutex attribute object*) benötigt.

pthread_once_t

Ist für die dynamische Paketinitialisierung vorgesehen.

pthread_rwlock_t

Für den Einsatz als Lese-/Schreibschutz (*locks*) vorgesehen.

pthread_rwlockattr_t

Dient zur Beschreibung des Lese-/Schreibschutz (*read-write lock attributes*).

pthread_t

Identifiziert einen Thread.

ptrdiff_t

Vorzeichenbehafteter Integer zur Darstellung des Ergebnis einer Subtraktion zweier Zeiger.

regex_t

Struktur für die Verwendung in der Mustererkennung mit regulären Ausdrücken (enthält den regulären Ausdruck).

regmatch_t

Struktur für die Verwendung in der Mustererkennung mit regulären Ausdrücken (enthält das Ergebnis der Mustererkennung).

rlim_t

Vorzeichenloser Integer zur Darstellung von Beschränkungen (*limits*). Die Typen **int** und **off_t** können nach **rlim_t** ohne Verlust gecastet werden.

sem_t

Wird für die Ausführung von Operationen mit Semaphoren benötigt.

sig_atomic_t

Integer zur Beschreibung eines Objekts, auf das nur als atomare Einheit zugegriffen werden kann, auch wenn asynchrone Interrupts auftreten.

sigset_t

Integer, der zur Darstellung von Signalsätzen verwendet wird (siehe 8.6).

size_t

Vorzeichenloser Integer zur Beschreibung von Objektgrößen.

speed_t

Wird für Baudraten von Terminals eingesetzt.

ssize_t

Vorzeichenbehafteter Integer zum Zählen von Bytes und zur Fehleranzeige (z.B. -1).

suseconds_t

Vorzeichenbehafteter Integer zur Darstellung der Zeit in Mikrosekunden.

tcflag_t

Wird für die Darstellung von Betriebsmodi von Terminals verwendet.

time_t

Integer oder Fließkommatyp zur Darstellung der Zeit in Sekunden. Definiert durch den ISO C Standard.

timer_t

Wird für Timer-IDs verwendet und von `timer_create` zurückgeliefert.

uid_t

Integer zur Beschreibung von User-IDs.

useconds_t

Vorzeichenloser Integer zur Angabe der Zeit in Mikrosekunden.

va_list

Typ für das Behandeln von variablen Argumentlisten.

wchar_t

Integer der groß genug für die Aufnahme von Unicode-Zeichen ist.

wctype_t

Skalarer Typ zur Beschreibung einer Zeichenklasse (*character class descriptor*).

wint_t

Integer, der jeden gültigen Wert eines wchar_t oder WEOF aufnehmen kann.

wordexp_t

Struktur, die für *Word Expansion* benutzt wird. Beispielsweise wird das Tilde-Zeichen (~) als Heimatverzeichnis des Benutzers erweitert.

2.6 Ermittlung von Implementierungsdetails

Jedes System verfügt über bestimmte Kenngrößen, die Aufschluß über die Fähigkeiten des Systems geben. Die so genannten *Magic Numbers* sind implementierungsspezifisch. Während diese Größen früher fest kodiert wurden, haben sich im Laufe der Standardisierungsbemühungen, andere Techniken zur Speicherung und Abfrage der magischen Zahlen herauskristallisiert. Wir unterscheiden drei Merkmale eines Systems: Optionen zur Übersetzungszeit (*compile-time options*), Obergrenzen zur Übersetzungszeit (*compile-time limits*) und Laufzeitobergrenzen (*run-time limits*).

Einstellungen und Obergrenzen zur Übersetzungszeit werden in Headern abgelegt. Dort sind sie als Konstanten mit geläufigen Variablennamen definiert. Laufzeitobergrenzen, die sich nicht auf Dateien oder ein Verzeichnis beziehen, fragen wir mit der Funktion `sysconf(2)` ab. Laufzeitobergrenzen, die sich auf Dateien und Verzeichnisse beziehen, erhalten wir durch `pathconf(2)` bzw. `fpathconf(2)`.

```
#include <unistd.h>

long sysconf(int name);
long pathconf(const char *pathname, int name);
long fpathconf(int filedes, int name);
```

Rückgabewerte: Alle drei geben den jeweiligen Wert zurück, andernfalls -1.

name

Einer der _SC_ und _PC_ Konstanten aus Anhang A.

pathname

Pfad, dessen Limit ermittelt werden soll.

fd

File Descriptor eines bereits geöffneten Pfades, dessen Limits ermittelt werden sollen.

Die beiden Funktionen `pathconf(2)` und `fpathconf(2)` geben die aktuelle Einstellung (*configuration*) eines einstellbaren Limits einer Datei oder eines Verzeichnisses auf dem zugrunde liegenden System zurück. `pathconf(2)` erwartet einen Pfad als erstes Argument, während `fpathconf(2)` auf einen File Descriptor angewendet wird.

`pathconf(2)` und `fpathconf(2)` schlagen fehl, wenn:

- das `name`-Argument ungültig ist,
- die Variable, die mit `name` assoziiert ist, kein Limit für den Pfad oder File Descriptor aufweist,
- die Implementierung `pathname` zur Abfrage des Wertes von `name` benötigt, doch die Assoziation von `name` mit dem Pfad nicht unterstützt wird (beispielsweise `SYMLINK_MAX` mit `fd` oder `name` als symbolischen Link),
- der Prozess keine ausreichenden Rechte zur Abfrage besitzt,
- oder der Pfad nicht existiert.

Ist alles in Ordnung geben `pathconf(2)` und `fpathconf(2)` den aktuellen Variablenwert der Datei oder des Verzeichnisses zurück, ohne `errno` zu verändern.

Listing 2.3: Anwendung von `pathconf(2)`

Das folgende Beispiel zeigt die Anwendung von `pathconf(2)` am Beispiel der `_PC_PATH_MAX`-Konstante (maximale Anzahl von Bytes in einem Pfadnamen):

```

1 #include <errno.h>
2 #include "header.h"
3
4 int main(int argc, char *argv[]) {
5     long path_limit;
6     char *basename = basename_ex(argv[0]);
7     errno = 0;
8
9
10    if (argc != 2)
11        err_fatal("Usage: %s <arg>\n", basename);
12
13    if ((path_limit = pathconf(argv[1], _PC_PATH_MAX)) < 0) {
14        if (errno != 0)
15            err_fatal("pathconf failed!");
16
17        fputs("No limit returned", stdout);
18    } else {
19        printf("_PC_PATH_MAX: %ld\n", path_limit);
20    }
21
22    exit(0);
23 }
```

Listing 2.3: `xcode/pathconf.c` - Anwendung von `pathconf`.

Mit der Funktion `sysconf(2)` fragen wir aktuelle Werte verstellbarer Systemobergrenzen oder Einstellungen der Implementierung ab. Die vollständige Liste der Konstanten finden Sie in Anhang A (Seite 445). Ein POSIX-kompatibles System sollte mindestens diesen Satz von Konstanten definieren, die in den Headern `<limits.h>` oder `<unistd.h>` abgelegt sind. □

In folgendem Beispiel rufen wir `pathconf(2)` zweimal auf: einmal um herauszufinden, ob `_PC_NO_TRUNC` Dateinamen länger als `_PC_NAME_MAX` automatisch gekürzt werden und ein zweites mal um zu sehen, ob der Dateiname in `buf` zu lang für das Limit von `/tmp` ist:

```

#include <stdlib.h>      /* for EXIT_SUCCESS */
#include <stdio.h>
#include <fcntl.h>        /* for O_CREAT */
#include <unistd.h>       /* for _PC_* constants */
#include <string.h>        /* for strlen */
#include <errno.h>

extern int errno;

int main(int argc, char **argv)
{
    int fd;
    errno = 0; /* reset errno */

    /* buf has a strlen of 263; too long for most systems */
    char *buf = "ThisIsAVeryLongFileNameThatMayOrMayNotBeTruncated\
WhenBeingUsedToWriteAFileToTmpSoLetsFindOutusing\
pathconfWhichIsCalledTwiceInThisExampleToDetermine\
IfThisIsTheCaseByFirstCheckingForTruncationInTmp\
AndComparingBufLengthAgainstPCNAMEMAX.IfTooLongA\
MessageWillBePrinted";

    printf("buf is %d characters long\n", strlen(buf));

    if (pathconf("/tmp", _PC_NO_TRUNC) == -1) { /* _POSIX_NO_TRUNC in effect? */
        if (strlen(buf) > pathconf("/tmp", _PC_NAME_MAX)) { /* no */
            fprintf(stderr, "Filename too long.\n");
            /* handle error here */
        } else if (errno) {
            /* handle pathconf error */
        }
    }

    /* _POSIX_NO_TRUNC in effect for /tmp */
    if ((fd = open(buf, O_CREAT, 0644)) < 0) {
        /* handle open error */
    }

    return EXIT_SUCCESS;
}

```

□

Listing 2.4: Anwendung von `sysconf(2)`

Das folgende Beispiel demonstriert, wie Sie die maximale Länge von Argumenten für `exec(2)` (siehe Abschnitt 7.2.3 *Die Funktionsfamilie exec(2)* auf Seite 156) bestimmen.

```

1 #include <errno.h>
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     long value;
6     errno = 0;
7
8     if ((value = sysconf(_SC_ARG_MAX)) < 0) {
9         if (errno != 0)

```

```
10         err_fatal("sysconf error");
11
12     fputs("ARG_MAX not defined.", stdout);
13 } else {
14     printf("ARG_MAX = %ld\n", value);
15 }
16
17 return (0);
18 }
```

Listing 2.4: xcode/sysconf.c - Anwendung von sysconf.



Kapitel 3

Dateibehandlung

The first precept was never to accept a thing as true until I knew it as such without a single doubt.

RENE DESCARTES (1596 - 1650), 'LE DISCOURS DE LA METHODE,' 1637

In diesem Abschnitt behandeln wir die wichtigsten Aufrufe zur Dateibehandlung: `open(2)`, `creat(2)`, `close(2)` und `lseek(2)`. Für viele Operationen wird ein sog. *File Descriptor* benötigt oder zurückgeliefert. Der File Descriptor ist ein Integer, der die in der aktuellen Operation involvierten Datei eindeutig identifiziert. Wenn wir beispielsweise eine Datei öffnen möchten, so benutzen wir die Funktion `open(2)`, die einen File Descriptor der soeben geöffneten Datei zurückgibt. Haben wir anschließend die Arbeit an der Datei abgeschlossen, so können wir die Datei wieder schließen und den File Descriptor frei geben, indem wir einfach nur `close(fd)` aufrufen, wobei `fd` der File Descriptor ist, der gerade von `open(2)` geliefert wurde.

Abbildung 3.1 zeigt drei wichtige Datenstrukturen, die in die Verwaltung der File Descriptoren involviert sind. Öffnen wir eine Datei (hier: `myfile`), so wird uns vom Kernel ein File Descriptor zugewiesen (hier: 3). Alle offenen File Descriptors werden vom Prozess selbst verwaltet und in der *File Descriptor Table* gespeichert. Das gleiche macht auch der Kernel. Er speichert alle offenen File Descriptors des Systems in der *System File Table*, darunter auch unser Descriptor, der auf die Datei `myfile` zeigt. Der Speicherort wird in einer *i-node Table* im Speicher vorgehalten. Die beiden Buchstaben A und B zeigen, daß nicht nur unser Prozess (B), sondern auch ein anderer (B) die Datei `myfile` geöffnet hat. Beide Descriptoren zeigen auf den gleichen Eintrag in der i-node Table.

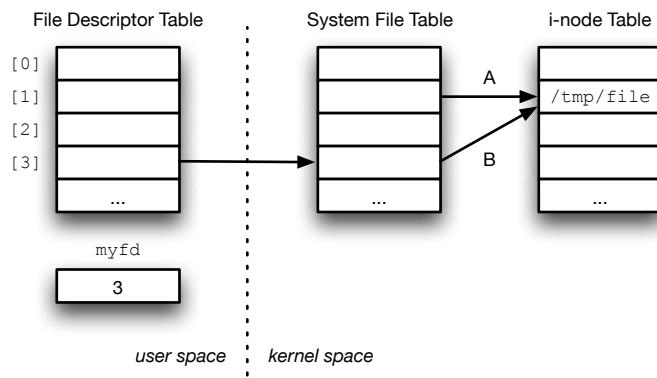


Abbildung 3.1: Beziehungen zwischen File Descriptor Table, System File Table und der i-node Table.

Genauere Informationen über den Umgang mit und die Natur der File Descriptors erfahren Sie im Laufe dieses Kapitels.

3.1 Dateien öffnen und erstellen

In fast alle Operationen in der UNIX-Umgebung sind entweder mit Dateien oder Prozessen verbunden. In diesem Abschnitt lernen wir die wichtigsten Funktionen zur Anbindung von Dateien an Prozesse kennen. Dazu zählen:

- open**
öffnet Dateien mit bestimmten Einstellungen.
- creat**
erzeugt neue Dateien (wird aber zu Gunsten von **open** nicht mehr verwendet).
- close**
schließt geöffnete Dateien und gibt den File Descriptor frei.

3.1.1 Die Funktion open

Die Funktion **open(2)** etabliert eine Verknüpfung zwischen einer Datei und einer Dateibeschreibung in Form eines *File Descriptors*. Sofern diese Datei noch nicht von dem Prozess geöffnet wurde, wird eine neue Dateibeschreibung, die der betreffenden Datei zugeordnet ist und ein File Descriptor, der die neue Dateibeschreibung referenziert, erzeugt und von **open(2)** zurückgegeben. Der File Descriptor wird dann für andere Operationen, die auf die Datei angewendet werden sollen, genutzt.

Beim Aufruf von **open(2)** gibt der Kernel den kleinsten verfügbaren File Descriptor für den aktuellen Prozess zurück. Der Offset, welcher die momentane Position des Cursors in der Datei angibt, wird durch **open(2)** an den Anfang der Datei gesetzt.

Die Synopsis von **open(2)** lautet folgendermaßen:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *file, int oflag, ... /* mode_t mode */);
```

Rückgabewert: Gibt einen File Descriptor zurück oder -1 bei Fehler..

Tabelle 3.1 listet die zulässigen Optionen für *oflag* auf, wobei immer nur eine der drei Konstanten angegeben werden darf. In Tabelle 3.2 sind die Flags für den gewünschten E/A-Modus zu finden.

<i>oflag</i>	Bedeutung
O_RDONLY	Öffnen zum Lesen (nur lesen)
O_WRONLY	Öffnen zum Schreiben (nur schreiben)
O_RDWR	Öffnen zum Lesen und Schreiben

Tabelle 3.1: Flags für den Zugriffsmodus.

In früheren System V Varianten wurde die **O_NODELAY**-Option eingeführt, die mit **O_NONBLOCK** identisch ist. Die Option **O_SYNC** ist nicht durch POSIX.1 definiert, wird allerdings von SYS V Implementierungen unterstützt.

Wenn wir **open(2)** auf einen symbolischen Link anwenden, wird die durch den Link spezifizierte Datei geöffnet und nicht der Link selbst. Wenn wir den symbolischen Link öffnen möchten, benötigen wir die Funktion **readlink** aus Abschnitt 4.3.5 auf Seite 73.

Um eine Datei namens **tmp.txt** im aktuellen Arbeitsverzeichnis zu öffnen könnten wir folgende Aufrufe verwenden, ja nachdem, was wir mit der Datei vor haben:

<i>oflag</i>	Bedeutung
O_APPEND	Inhalte werden am Ende der Datei angefügt.
O_CREAT	Erzeuge die Datei, wenn Sie noch nicht existiert. Diese Option erfordert, daß <i>mode</i> die Zugriffsrechte spezifiziert.
O_EXCL	Prüft, ob eine Datei bereits existiert und wenn nicht, wird sie in einer atomaren Operation erzeugt.
O_TRUNC	Sollte die Datei bereits existieren, wird sie beim Öffnen mit O_WRONLY oder O_RDWR auf Größe 0 zurückgesetzt.
O_NOCTTY	Wenn <i>pathname</i> ein Terminal referenziert, wird es nicht als steuerndes Terminal für diesen Prozess definiert.
O_NONBLOCK	Wenn <i>pathname</i> eine FIFO-Warteschlange, ein Block Special File oder ein Character Special File, werden nachfolgende E/A-Operationen <i>non-blocking</i> durchgeführt. Das gilt auch für das Öffnen der FIFO-Warteschlange und der Special Files.
O_SYNC	Warte bis physikalische E/A-Operationen fertig sind.

Tabelle 3.2: Flags für den E/A-Modus.

```
// öffnet die Datei zum Lesen und Schreiben im aktuellen Verzeichnis
fd = open("tmp.txt", O_RDWR);

// gleiche Funktion, diesmal mit vollqualifiziertem Pfad
fd = open("/var/log/tmp.txt", O_RDWR);
```

Mit folgenden Flags öffnen wir die Datei nur zum Anfügen von Daten (O_WRONLY | O_APPEND). Existiert sie jedoch noch nicht, wird sie erstellt (O_CREAT), allerdings nur mit Lese-, Schreib- und Ausführungsrechten für den Besitzer (S_IRWXU)

```
fd = open("tmp.txt", O_WRONLY | O_APPEND | O_CREAT, S_IRWXU);
```

3.1.2 Die Funktion creat

Wir erzeugen neue Dateien mit `creat(2)`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *file, mode_t mode);
```

Rückgabewert: File Descriptor oder -1 bei Fehler.

Durch die Optionen O_WRONLY, O_CREAT und O_TRUNC der Funktion `open(2)` ist `creat` nicht mehr notwendig. Das hat historische Gründe. Frühe UNIX-Versionen erwarteten als zweites Argument für `open(2)` entweder 0, 1 oder 2, die die jeweiligen Optionen als Zahlenkonstanten definierten. Es war deshalb nicht möglich, eine Datei zu öffnen, die nicht schon zuvor erzeugt wurde. Verschiedene Werte mit Hilfe logischer Operationen miteinander zu verknüpfen macht `creat(2)` überflüssig, denn `open(2)` erledigt alles für uns:

```
int fd; /* file descriptor */
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Mit `creat` würden wir das gleiche erreichen, wenn wir schreiben:

```
int fd; /* file descriptor */
fd = creat(pathname, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

Die Implementierung von `creat(2)` wurde aus Gründen der Kompatibilität zu älteren Systemen in den POSIX-Standard aufgenommen, ist aber redundant, weil ein Aufruf von `open(2)` grundsätzlich genügt.

Von der Verwendung von `creat(2)` wird grundsätzlich abgeraten, weil der Systemaufruf und ein nachfolgender Aufruf von `open(2)` nicht atomar sind. Das kann unschöne Seiteneffekte nach sich ziehen. Die Verwendung von zwei Systemaufrufen öffnet ein Zeitfenster welches anderen Prozessen erlaubt, auf die gleiche Resource zuzugreifen. Das Flag `O_APPEND` von `open(2)` eliminiert das Problem, da die Erzeugung der Datei und das Öffnen derselben atomar geschieht.

3.1.3 Die Funktion `close`

Die Funktion `close(2)` soll die Verbindung zwischen der Dateibeschreibung und dem File Descriptor selbst wieder aufheben. Aufheben bedeutet hier, daß der Descriptor für nachfolgende Aufrufe von `open(2)` oder anderen Funktionen, die File Descriptor zurückgeben, wieder zur Verfügung steht.

Wird der Aufruf von `close(2)` durch ein Signal unterbrochen wird -1 zurückgegeben und `errno` auf `EINTR` gesetzt. Der Zustand des File Descriptors ist dann undefiniert. Ist aber ein E/A-Fehler gemeldet worden, so enthält `errno` `EIO`.

```
#include <unistd.h>

int close(int fd);
```

Rückgabewert: Bei erfolgreicher Ausführung wird 0, andernfalls -1 zurückgegeben.

fd

File Descriptor auf den die Operation angewandt werden soll.

Nachdem die Datei geschlossen wurde, sind auch alle Ressourcen wieder freigegeben. Wird ein Prozess terminiert, so werden auch alle offenen Datei durch den Kernel geschlossen. Viele Programme machen sich dieses Verhalten zu Nutze und überlassen dem Kernel die Verwaltung der File Descriptors.

3.2 Dateien lesen und schreiben

Nachdem eine Datei geöffnet wurde, können wir Daten einlesen, schreiben oder eine bestimmte Position in der Datei aufsuchen. Folgende Funktionen sind für diese Aufgaben vorgesehen:

lseek
positioniert einen Cursor an einer bestimmten Position in der Datei.

read
liest Daten aus Dateien und speichert sie in einem Buffer.

write
schreibt Daten aus einem Buffer in Dateien.

3.2.1 Die Funktion `lseek`

Jeder geöffneten Datei ist ein aktueller Offset zugeordnet, der die momentane Position eines dateispezifischen Indikators repräsentiert. `read(2)`- und `write(2)`-Operationen beginnen immer am aktuellen Offset und inkrementieren diesen ständig um die Anzahl der gelesenen bzw. geschriebenen Bytes. Standardmäßig wird der Offset mit 0 initialisiert; `O_APPEND` positioniert den Offset am Ende der Datei.

Wir können die Position innerhalb einer Datei mit der Funktion `lseek(2)` explizit steuern.

```
#include <sys/stat.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Rückgabewert: Gibt neue Position in der Datei zurück oder -1 bei Fehler.

fd

File Descriptor auf den die Operation angewendet werden soll.

offset

Neue Position in der Datei.

whence

Bestimmt, wie die neue Position in der Datei ermittelt werden soll.

Die zulässigen Werte für *whence* sind in Tabelle 3.3 aufgeführt. Abbildung 3.2 fasst die Konstanten zusammen.

*

<i>whence</i>	Bedeutung
SEEK_SET	Setzt die neue Position in der Datei auf <i>offset</i> Bytes ausgehend vom Anfang der Datei.
SEEK_CUR	Setzt die neue Position in der Datei auf <i>offset</i> Bytes zur aktuellen Position (<i>offset</i> + aktuelle Position).
SEEK_END	Setzt die neue Position in der Datei auf die Dateigröße plus <i>offset</i> , wobei <i>offset</i> positiv oder negativ sein kann.

Tabelle 3.3: Zulässige Werte für den Parameter *whence* von *lseek*.

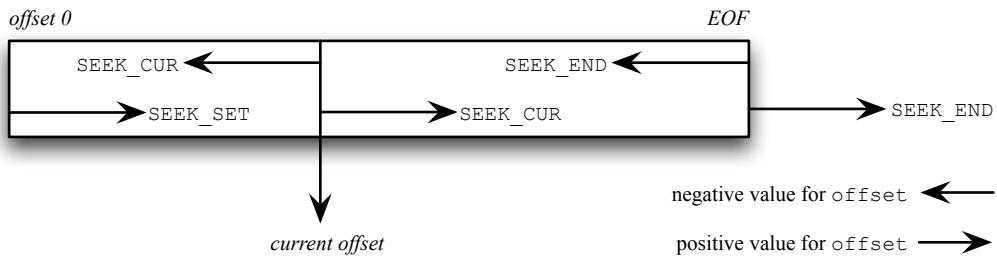


Abbildung 3.2: So beeinflussen positive und negative Offsets die Position in der Datei.

In vielen frühen SYSV-Varianten wurden die Konstanten SEEK_SET, SEEK_CUR und SEEK_END durch die Zahlen 0, 1 und 2 implementiert. Viele Programme werden heute noch auf diese Weise implementiert. Der Buchstabe *l* in *lseek(2)*, steht für *long integer*, wobei der Datentyp *long* erst mit UNIX Version 7 in C eingeführt wurde.

Normalerweise arbeiten wir immer mit einem positiven *offset*. Da negative Werte möglich sind, müssen wir darauf achten, daß wir auf -1 testen und nicht nur, ob der Rückgabewert kleiner 0 ist. Ein zweiter Punkt betrifft den Datentyp *off_t*, der vorzeichenbehaftet ist (*signed*). Dadurch ergibt sich eine maximale Dateigröße von 2^{31} Bytes auf die *lseek(3)* angewendet werden kann.

Der ISO C Standard schreibt die Implementierung der Funktionen *fgetpos(2)* und *fsetpos(2)* zur Abfrage bzw. das Setzen der Cursorposition in einer Datei vor. Sie funktionieren hervorragend mit sehr großen Dateien, da ein spezieller Datentyp (*fpos_t*) für die Verwaltung des Offsets angewendet wird.

Obwohl `lseek(3)` den Cursor hinter das Dateiende verschieben kann, führt die Funktion die Dateivergrößerung nicht automatisch durch. Erst wenn die POSIX-Funktionen `write(2)`, `truncate(2)` und `ftruncate(2)` sowie die ISO C Funktionen `fwrite`, `fprintf(3)`, usw. verwendet werden, wird die Größe der Datei verändert und gegebenenfalls mit 0 aufgefüllt. Ein ungültiger Offset verursacht `EINVAL` und kann implementierungsabhängig und geräteabhängig sein (beispielsweise bei der Arbeit mit Speicherbereichen). Ebenso kann ein negativer Offset für bestimmte Geräte einiger Implementierungen durchaus gültig sein.

Abschließend sei angemerkt, daß die Position in der Datei nur durch den Kernel verwaltet wird. Es findet keine E/A-Operation beim Aufruf von `lseek(2)` statt.

Listing 3.1: Anwendung von `lseek(2)`

```

1 #include <fcntl.h>
2 #include <sys/stat.h>
3 #include <sys/types.h>
4 #include "header.h"
5
6 int main(int argc, char **argv) {
7     char *basename = basename_ex(argv[0]);
8     int fd;
9     mode_t mode = S_IRUSR | S_IRGRP | S_IROTH;
10    off_t new_offset;
11
12    if (argc != 2)
13        err_fatal("Usage: %s <file>\n", basename);
14
15    if ((fd = open(argv[1], O_RDONLY, mode)) < 0)
16        err_fatal("open failed for %s.\n", argv[1]);
17
18    /* position cursor at byte 12 using fd */
19    if ((new_offset = lseek(fd, 12, SEEK_SET)) == -1)
20        err_fatal("lseek failed\n");
21    else
22        printf("New position: %ld\n", new_offset);
23
24    return(0);
25 }
```

Listing 3.1: xcode/lseek.c - Anwendung von `lseek(2)`.

Das Beispiel ist denkbar einfach. Zuerst öffnen wir eine Datei zum Lesen und positionieren den Cursor auf das 12. Byte innerhalb der Datei, die wir als Kommandozeilenparameter übergeben haben. Anschließend überzeugen wir uns vom Erfolg des Vorhabens. □

Wenn wir wissen möchten, an welcher Position sich der Cursor momentan befindet, rufen wir

```
off_t cur_pos;
cur_pos = lseek(fd, 0, SEEK_CUR);
```

auf. Hier machen wir uns die Eigenschaft zu Nutze, daß `lseek(2)` immer den aktuellen Offset zurück liefert. Wenn wir als neuen Offset 0 angeben, erhalten wir genau diese Information. Übrigens lässt sich auf die gleiche Weise herausfinden, ob eine Datei `lseek(2)`-fähig ist, was sich besonders im Zusammenhang mit Gerätedateien als nützlich erweist, die unterstützen nämlich kein Seeking und unterscheiden sich dadurch von anderen *Regular Files*.

3.2.2 Die Funktion `read`

Um Inhalte einer Datei einzulesen, können Sie die Funktion `read(2)` verwenden.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

Rückgabewert: Anzahl der gelesenen Bytes oder -1 bei Fehler.

fd

File Descriptor auf den die Operation angewendet werden soll.

buf

Speicherplatz zur Aufnahme der gelesenen Daten.

nbytes

Anzahl der zu lesenden Bytes.

Die Funktion `read(2)` versucht, `nbyte` Bytes aus der durch `fd` assoziierten Datei in `buf` zu lesen. Das Verhalten mehrfacher, paralleler Aufrufe von `read(2)` auf die gleiche Pipe, FIFO-Warteschlange oder Terminaldatei ist nicht spezifiziert. Wird keine Fehlererkennung durch den Anwender vorgenommen oder ist `nbytes` 0, so gibt `read(2)` nur 0 zurück und liefert keine anderen Ergebnisse. Dateien, die `lseek(2)`-Aufrufe unterstützen, wie etwa reguläre Dateien, werden immer ausgehend von der aktuellen Cursorposition des File Descriptors eingelesen. Dabei wird der Offset immer um die Anzahl der gelesenen Bytes erhöht. Unterstützt die Datei kein *Seeking*, wie z.B. Terminaldateien, wird die Datei immer von der aktuellen Position aus gelesen. Der Wert des Offsets des File Descriptors solcher Dateien ist allerdings undefiniert.

Das folgende Beispiel illustriert den Einsatz von `read(2)`.

Listing 3.2: Anwendung von `read(2)`.

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include "header.h"
5
6 int main(int argc, char *argv[]) {
7     char    buf[20]; /* we want the first 20 bytes */
8     char    *basename = basename_ex(argv[0]);
9     int      fd;
10    size_t   nbytes = sizeof(buf);;
11    ssize_t  bytes_read;
12
13    if (argc != 2)
14        err_fatal("Usage: %s <file>\n", basename);
15
16    if ((fd = open(argv[1], O_RDONLY)) < 0)
17        err_fatal("open failed for %s.\n", argv[1]);
18
19    bytes_read = read(fd, buf, nbytes);
20
21    if (bytes_read != nbytes) {
22        if (bytes_read < 0)
23            err_fatal("read failed for %s.\n", argv[1]);
24        else
25            err_normal("Short read for %s.\n", argv[1]);
26    }
27
28    printf("%s\n", buf); /* print what we've read so far */
29
30    return (0);

```

 31 }

Listing 3.2: xcode/read.c - Anwendung von read(2).

In der Regel wird die gelesene Anzahl der Bytes oder -1 im Fehlerfall zurückgegeben. Sollte der Buffer nicht vollständig gefüllt worden sein, entspricht der Wert der Anzahl der gelesenen Bytes. Es kann also vorkommen, daß die Anzahl der gelesenen Bytes geringer als der spezifizierte Wert (*nbytes*) ist. Die Gründe dafür sind vielfältig:

- Wird von einem Terminal gelesen, geschieht das meist nur Zeilenweise, was aber geändert werden kann, wie wir in Kapitel 9 sehen werden.
- Wenn wir eine reguläre Datei einlesen und das Ende der Datei erreicht wird bevor *nbytes* gelesen werden konnten, wird nur die tatsächlich gelesene Anzahl von Bytes zurückgegeben. Sind nur noch 50 Bytes auslesbar, wir fordern aber 200 Bytes an, so gibt **read(2)** auch nur 50 Bytes zurück.
- Lesen wir von einem Netzwerk-Socket oder STREAM, können Zwischenspeicher im Netzwerk eine Veränderung der Anzahl der zurückgegebenen Bytes verursachen, da wir keinen Einfluß auf das Buffering haben.
- Viele Record-orientierte Datenträger (z.B. Tapes) lesen nicht Byte-weise, sondern Blockweise, so daß auch hier die Anzahl der tatsächlich gelesenen Bytes von den von uns angeforderten abweichen kann.

Leider ist **read(2)** im Kontext von Netzwerkanwendungen nicht besonders zuverlässig, was weniger an dem Systemaufruf selbst, sondern vielmehr an der Signalbehandlung liegt. Im Rahmen der Netzwerkprogrammierung werden wir eine Funktion entwickeln, die eine relativ zuverlässige Nutzung von **read(2)** zuläßt. Wir werden im Laufe der weiteren Besprechungen immer wieder Systemaufrufe im Zusammenhang mit Signalen (Kapitel 8 *Signalbehandlung*) betrachten.

3.2.3 Die Funktion write

Daten schreiben wir mit der Funktion **write(2)**.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);
```

Rückgabewert: Anzahl der geschriebenen Bytes, -1 bei Fehler.

fd

File Descriptor auf den die Operation angewendet werden soll.

buf

Inhalte, die geschrieben werden sollen.

nbytes

Größe der zu schreibendenen Inhalte (beispielsweise `sizeof(writeme)`).

Im Normalfall wird *nbytes* zurückgegeben; wenn nicht, ist ein Fehler aufgetreten. Wie bereits bei **lseek(2)** erwähnt, beginnen die Schreiboperationen immer am aktuellen Offset, beispielsweise initialisiert durch den Aufruf von **open(2)**. Nach Abschluß der Schreiboperation wird der Offset um die Anzahl der geschriebenen Bytes (*nbytes*, wenn keine Fehler aufgetreten sind) inkrementiert.

Wie **read(2)** auch, versucht **write(2)**, mindestens *nbytes* aus **buf** in die durch **fd** assoziierte Datei zu schreiben. Wenn *nbytes* Null ist oder **fd** keine reguläre Datei referenziert, so ist das Resultat undefiniert.

Der Schreibvorgang in eine *reguläre Datei* oder in eine, die Seeking unterstützt, beginnt immer an der aktuellen Cursorposition des File Descriptors (sprich: dem Offset). Vor der Rückkehr der Funktion wird der Offset um die Anzahl der geschriebenen Bytes erhöht. Sollte der Cursor hinterher über das Dateiende hinausgehen, so wird die Länge der Datei an die Cursorposition angepasst. Bei allen anderen Dateien beginnt der Schreibvorgang immer an der aktuellen Position und der Wert des Cursors ist nicht definiert.

Wenn das Flag `O_APPEND` in `open(2)` mitgegeben wurde, wird der Cursor vor jedem Schreibvorgang an das Ende der Datei gesetzt, wobei keine Operationen während der Änderung des Offsets und des Schreibvorgangs stattfinden dürfen. Kann ein Aufruf von `write(2)` nicht vollständig durchgeführt werden, beispielsweise wenn die maximale Dateigröße für diesen Prozess erreicht wurde oder einfach nur das Medium voll ist, so werden nur so viele Daten geschrieben, wie Platz verfügbar ist. Sind beispielsweise nur noch 100 Bytes Platz auf dem Medium, so wird eine Anforderung von `write(2)`, 128 Bytes zu schreiben, einen Rückgabewert von 100 liefern und nicht 128. Ein nachfolgender Aufruf würde unmittelbar zu einem Fehler führen.

Ein Aufruf von `write(2)` kann aus mehreren Gründen unerwartete Ergebnisse liefern. Wie erwähnt, kann der Platz auf dem Medium ausgehen, was zu dem Fehlercode `SIGXFZS` führt. Andererseits kann der Vorgang durch Signale unterbrochen werden. Unterbricht ein Signal den Schreibvorgang bevor Daten geschrieben wurden, so wird -1 zurückgegeben und `EINTR` gesetzt. Der Vorgang kann auch nachdem die Daten teilweise erfolgreich geschrieben wurden, unterbrochen werden. In diesem Fall wird neben `EINTR` die Anzahl der geschriebenen Daten zurückgegeben und wir müssen angemessen darauf reagieren.

Schreibzugriffe auf eine Pipe oder FIFO-Warteschlange (Abschnitte 10.1 *Pipes* auf Seite 255 und 10.2 *Benannte Pipes* auf Seite 263) werden grundsätzlich genauso gehandhabt, wie bei regulären Dateien, allerdings mit folgenden Ausnahmen:

- Da kein Offset mit einer Pipe assoziiert ist, werden alle Daten an das Ende der Pipe geschrieben.
- Schreibvorgänge mit einer Größe von `PIPE_BUF` oder kleiner werden nicht durch Daten eines anderen Prozesses vermischt, der Schreibvorgang ist also *atomic*. Sollte `PIPE_BUF` überschritten worden sein, können die Daten in ungeregelten Grenzen mit denen anderer Prozesse durchmischt werden, auch wenn das Flag `O_NONBLOCK` gesetzt wurde.
- Ist das Flag `O_NONBLOCK` nicht gesetzt, so kann ein Schreibvorgang den Thread anhalten (*blocking*), doch bei erfolgreichem Abschluß werden `nbytes` zurückgegeben.
- Wenn `O_NONBLOCK` gesetzt ist, wird `write(2)` anders ausgeführt:
 - Die Funktion hält den Thread nicht an.
 - Ein Aufruf von `write(2)` über `PIPE_BUF` Bytes oder weniger hat folgende Auswirkungen: Wenn genügend Platz in der Pipe vorhanden ist, wird `write(2)` alle Daten in die Pipe schreiben und die Anzahl der Bytes zurückgeben. Andernfalls schreibt `write(2)` keine Daten und gibt -1 mit `EAGAIN` in `errno` zurück.
 - Sollen mehr als `PIPE_BUF` Bytes geschrieben werden, kann folgendes passieren:
 - * Wenn ein oder mehr Bytes geschrieben werden können, wird übertragen, so viel es geht und die Anzahl der geschriebenen Bytes zurückgegeben. Andernfalls werden keine Daten übertragen und -1 mit `EAGAIN` zurückgegeben. Wenn alle Daten, die zuvor in die Pipe geschrieben wurden, ausgelesen werden, werden mindestens `PIPE_BUF` Bytes gelesen.
 - * Wenn keine Daten geschrieben werden können, findet kein Datentransfer statt und es wird -1 zurückgegeben.

Was passiert, wenn wir versuchen in einen File Descriptor zu schreiben (keine Pipe oder FIFO-Warteschlange), der die Daten nicht sofort akzeptieren kann? Ist `O_NONBLOCK` nicht gesetzt, wird `write(2)` den aufrufenden Prozess solange anhalten, bis geschrieben werden kann. Ist `O_NONBLOCK` allerdings aktiv, so wird der Thread nicht angehalten und `write(2)` soll schreiben so viel es kann und anschließend die Anzahl der geschriebenen Bytes zurückgeben. Andernfalls wird -1 zurückgegeben und `EAGAIN` gesetzt.

Nach erfolgreichem Schreibvorgang aktualisiert `write(2)` die Felder `st_ctime` und `st_mtime` der Datei und deaktiviert die `S_ISUID` und `S_ISGID`-Flags bei regulären Dateien (siehe Funktion `stat` in Abschnitt 4.1.1 auf Seite 56).

Listing 3.3: Einfache Implementierung des cp-Befehls.

Das folgende Beispiel zeigt eine sehr einfache Implementierung des cp-Befehls.

```

1 #include <sys/stat.h>
2 #include <sys/types.h>
3 #include <fcntl.h>
4 #include "header.h"

5
6 int main(int argc, char *argv[]) {
7     char *basename;
8     struct stat statbuf; /* to find block size for best performance */
9     int fd_src, fd_dest, n;

10    /* find basename */
11    if ((basename = strchr(argv[0], '/')) == NULL)
12        basename = argv[0];
13    else
14        basename++;
15

16    /* sufficient arguments */
17    if (argc != 3)
18        err_fatal("Usage: %s <source> <destination>\n", basename);
19

20    /* open source file for reading */
21    if ((fd_src = open(argv[1], O_RDONLY)) < 0)
22        err_fatal("open error for %s", argv[1]);
23

24    /* call fstat() for source file to find block size */
25    if (fstat(fd_src, &statbuf) < 0)
26        err_fatal("Something is wrong: cannot stat %s.\n", argv[1]);
27

28    /* now that we got it, use it */
29    char buf[statbuf.st_blksize];
30

31    /* open destination file for writing */
32    if ((fd_dest = open(argv[2], O_WRONLY | O_TRUNC | O_CREAT)) < 0)
33        err_fatal("open error for %s", argv[2]);
34

35    /* read and write on the fly */
36    while ((n = read(fd_src, buf, sizeof(buf))) > 0)
37        if (write(fd_dest, buf, n) != n)
38            err_normal("write error");
39

40    if (n < 0)
41        err_fatal("read error");
42

43    return (0);
44 }

```

Listing 3.3: xcode/cp.c - Lesen und schreiben von Dateien.

Aufmerksamkeit verdient die Größe von `buf`. Der Einfachheit halber haben wir `fstat` bemüht, um die Blockgröße abzufragen. Es hat sich herausgestellt, daß die beste Performance erreicht wird, die Größe des Puffers der Blockgröße des zugrunde liegenden Dateisystems entspricht, der dem Member `st_blksize` der Struktur `stat` entnommen werden kann.

Leider hat diese einfache Implementierung einen schwerwiegenden Fehler, der für uns noch nicht offensichtlich ist. Was passiert, wenn der Aufruf von `read(2)` oder `write(2)` in der `while`-Schleife durch den Kernel unterbrochen wird. In diesem Fall hätten wir entweder nicht alle Daten gelesen oder geschrieben. Eine bessere Lösung finden wir in Listing 3.4. Augenmerk legen wir auf die `while`-Schleife, in der wir

aufzeichnen, wie viele Bytes wir gelesen und geschrieben haben und die Werte mit dem Datenaufkommen vergleichen, das hätte geschrieben werden müssen. Wenn ein Prozess unterbrochen wurde (`errno == EINTR`), dann können wir genau an der Stelle weiter machen, an der wir aufgehört haben.

Listing 3.4: Bessere Implementierung des cp-Befehls

```

1 #include <sys/stat.h>
2 #include <sys/types.h>
3 #include <fcntl.h>
4 #include "header.h"
5
6 int main(int argc, char *argv[]) {
7     char *basename, *bufptr;
8     struct stat statbuf; /* to find block size for best performance */
9     int fd_src, fd_dest, n, bytesread, byteswritten;
10
11    /* find basename */
12    if ((basename = strchr(argv[0], '/')) == NULL)
13        basename = argv[0];
14    else
15        basename++;
16
17    /* sufficient arguments */
18    if (argc != 3)
19        err_fatal("Usage: %s <source> <destination>\n", basename);
20
21    /* open source file for reading */
22    if ((fd_src = open(argv[1], O_RDONLY)) < 0)
23        err_fatal("open error for %s", argv[1]);
24
25    /* call fstat() for source file to find block size */
26    if (fstat(fd_src, &statbuf) < 0)
27        err_fatal("Something is wrong: cannot stat %s.\n", argv[1]);
28
29    char buf[statbuf.st_blksize]; /* now that we got it, use it */
30
31    /* open destination file for writing */
32    if ((fd_dest = open(argv[2], O_WRONLY | O_TRUNC | O_CREAT)) < 0)
33        err_fatal("open error for %s", argv[2]);
34
35    /* read and write on the fly */
36    while (bytesread = read(fd_src, buf, statbuf.st_blksize)) {
37        if ((bytesread == -1) && (errno != EINTR))
38            break; /* a fatal error occurred */
39        else if (bytesread > 0) {
40            bufptr = buf;
41            while (byteswritten = write(fd_dest, bufptr, bytesread)) {
42                if ((byteswritten == -1) && (errno != EINTR))
43                    break;
44                else if (byteswritten == bytesread)
45                    break; /* success */
46                else if (byteswritten > 0) {
47                    bufptr += byteswritten;
48                    bytesread -= byteswritten;
49                }
50            }
51            if (byteswritten == -1)
52                break;
53        }
54    }
55}
```

```

56     close(fd_dest);
57     close(fd_src);
58     return (0);
59 }
```

Listing 3.4: xcode/cp2.c - Bessere Implementierung des cp-Befehls.

In Sektion 4.1.1 besprechen wir `stat` und seine Member. □

3.3 File Descriptors verwalten

File Descriptor werden vom Kernel verwaltet und referenzieren Dateien oder Streams. Bei der Arbeit mit Dateien und Verzeichnissen kann es vorkommen, daß wir die File Descriptoren direkt beeinflussen müssen. Dazu stehen uns folgende Funktionen zur Verfügung:

`dup` und `dup2`
duplicieren File Descriptoren.

`fcntl(2)`
kann Eigenschaften von File Descriptoren verändern.

3.3.1 Die Funktionen `dup` und `dup2`

Oftmals ist es notwendig, einen File Descriptor zu duplizieren, um beispielsweise geöffnete Dateien den Standard-Streams `STDIN_FILENO`, `STDOUT_FILENO` und/oder `STDERR_FILENO` neu zuzuordnen. Um einen existierenden File Descriptor zu duplizieren, können Sie eine der folgenden Funktionen einsetzen.

```
#include <unistd.h>

int dup(int fd);
int dup2(int fd, int fd2);
```

Rückgabewert: beide geben einen neuen File Descriptor zurück oder -1 bei Fehler.

`fd`
File Descriptor den wir duplizieren möchten.

`fd2`
Neuer File Descriptor den wir von `fd` übernehmen möchten.

Der Rückgabewert von `dup(2)` ist der nächste freie File Descriptor. UNIX ordnet die File Descriptor 0, 1 und 2 standardmäßig `STDIN`, `STDOUT` und `STDERR` zu, die meistens von der Shell geöffnet werden. Damit wäre der nächste freie File Descriptor 3. Im Gegensatz zu `dup(2)` spezifizieren Sie den neuen File Descriptor als zweites Argument von `dup2(2)`. Sollte `fd2` identisch mit `fd` sein, wird automatisch `fd2` zurückgegeben, ohne die referenzierte Datei zu schließen. Nach der Ausführung von `dup(2)` oder `dup2(2)` verwenden die neuen File Descriptors die gleiche Dateitabelle (*file table*) und damit auch die gleichen Flags. Jede Dateitabelle hat ihren eigenen Satz von Flags, die Informationen über die File Descriptors verwalten.

Der Nutzen von `dup(2)` erschließt sich nicht sofort. Nehmen wir an, wir haben eine Datei zum Lesen und Schreiben geöffnet, und wir haben bereits einige Daten in die Datei geschrieben. Nun möchten wir die Daten den UNIX-Kommandos `sed(1)` oder `awk(1)` mittels `fork(2)` und `exec(2)` zuführen (Abschnitt 7.2 geht auf `fork(2)` und `exec(2)` ein). Beide Programme lesen einen Datenstrom von `stdin`, verarbeiten ihn und schreiben das Resultat nach `stdout`. Hätten wir `stdin` geöffnet wäre das kein Problem, denn ein Kindsprozess erbt den Descriptor, doch das ist offenbar nicht der Fall, denn wir können nicht in `stdin` schreiben. Wie ordnen wir nun die geöffnete Datei `stdin` unseres Kindsprozesses zu?

Eine Möglichkeit ist, `stdin` zuerst zu schließen und die Datei unmittelbar danach wieder zu öffnen. Wir machen uns dabei eine Eigenschaft von `open(2)` zunutze: sie liefert immer den kleinsten verfügbaren Descriptor zurück. Demnach ist die Datei anschließend `stdin` zugeordnet.

```
fd = open("datafile", O_RDWR | O_CREAT, 0644);

/* Some data are written in the file */

close(STDIN_FILENO);
fd = open("datafile", O_RDONLY); /* Now STDIN_FILENO points to "datafile" */

/* fork() und exec() here */
```

Leider verhindert diese Variante dem Elternprozess, `stdin` weiter zu verwenden, so daß es wahrscheinlich besser ist, das Schließen und erneute Öffnen im Kindprozess zu erledigen, da es auch den Descriptor der Datendatei erbt:

```
fd = open("datafile", O_RDWR | O_CREAT, 0644);

/* Some data are written in the file */

if (fork() == 0) { /* child code */
    close(STDIN_FILENO); /* only child's stdin is closed */
    fd = open("datafile", O_RDONLY);
    execvp("sed", "sed", (char *)0);
    perror("sed");
}
```

Im Abschnitt 7.2 besprechen wir `fork(2)` und `exec(2)` im Detail.

Wir müssen nicht `dup(2)` verwenden, um Descriptors zu duplizieren. Die Funktion `fcntl(2)` kann das auch:

```
dup(fd);
```

ist gleichbedeutend mit

```
fcntl(fd, F_DUPFD, 0);
```

und

```
dup2(fd);
```

ist identisch mit

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

Beachten Sie aber, daß `dup2(2)` eine atomare Operation ist, während der vorangegangene Weg zwei Funktionsaufrufe involviert. Dabei kann es vorkommen, daß die File Descriptors durch Signale (Signale besprechen wir in Kapitel 8) verändert werden, wenn sie genau zwischen den Funktionsaufrufen vorkommen.

Listing 3.5: Verwendung von `dup2(2)`.

```
1 #include <fcntl.h> /* mode flags for open() */
2 #include <unistd.h> /* for STDOUT_FILENO */
3 #include "header.h"
4
5 int main(int argc, char **argv) {
```

```

6     int fd;
7
8     if ((fd = open("myfile", O_RDONLY | MODE_FLAGS)) < 0)
9         err_fatal("open() failed");
10
11    if (dup2(fd, STDOUT_FILENO) < 0)
12        err_fatal("dup2() failed");
13
14    close(fd);
15    exit(0);
16 }
```

Listing 3.5: xcode/dup2.c - Verwendung von dup2(2).

Der Aufruf von `open(2)` veranlaßt das Betriebssystem, die Datei `myfile` zu öffnen und ihr den nächsten freien File Descriptor zuzuweisen. `dup2(2)` schließt den als zweites Argument (`STDOUT_FILENO`) übergebenen File Descriptor und kopiert den Eintrag der System File Table von `fd` in den Eintrag von `STDOUT_FILENO`. □

Der `inetd`-Superserver macht intensiven gebrauch von `dup2(2)`, denn jedesmal wenn ein Socket eines Dienstes, beispielsweise FTP, bereit zum Lesen ist, erstellt `inetd` mittels `fork(2)` eine Kopie von sich selbst und ordnet die File Descriptors 0, 1 und 2 neu an. Anschließend wird der ursprüngliche Socket Descriptor geschlossen und die einzigen drei Descriptoren, die im `inetd`-Child geöffnet sind entsprechen denen von Standard Input, Standard Output und Standard Error. Jedesmal wenn der Child-Prozess von einem der drei Descriptoren liest oder schreibt, wird automatisch der Socket verwendet.

3.3.2 Die Funktion `fcntl`

Die Eigenschaften bereits geöffneter Dateien können wir mit `fcntl(2)` modifizieren.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int args */);
```

Rückgabewerte: abhängig von cmd, oder -1 bei Fehler.

fd

File Descriptor auf den die Operation angewendet werden soll.

cmd

zeigt die Aktion an, die auf `fd` angewendet werden soll (erfordert evtl. ein drittes Argument, siehe weiter unten).

args

Spezifiziert das genaue Verhalten von `cmd` (siehe weiter unten).

Folgende Operationen sind für `fcntl(2)` vorgesehen:

F_DUPFD

Bereits zugewiesenen File Descriptor duplizieren. Der neue File Descriptor wird als Rückgabewert der Funktion geliefert. Dabei wird der kleinst mögliche Descriptor ermittelt, der meist größer als 2 ist, denn schließlich sind die Descriptoren 0 bis 2 mit dem Terminal verbunden. Allerdings verfügt der neue Descriptor über seine eigenen Flags und `FD_CLOEXEC` wird zurückgesetzt, was bedeutet, daß der Descriptor auch über einen Aufruf von `exec(3)` hinaus verfügbar ist.

F_GETFD

Abfragen der File Descriptor Flags für *fd*. Aktuell ist nur **FD_CLOEXEC** als Flag definiert. Die Flags werden als Rückgabewert der Funktion geliefert.

F_SETFD

Setzen der File Descriptor Flags für *fd*. Die neuen Flags werden durch das dritte Argument, *args*, als Integer übergeben.

F_GETFL

Ermitteln der Status Flags für *fd* als Funktionswert. Die Flags sind gleichbedeutend mit den Optionen von **open(2)**. Nachteilig ist der Umstand, daß die Flags für mode nicht unabhängig von einander getestet werden können, da sie oft aus historischen Gründen durch 0, 1 und 2 definiert sind und sich vollständig ausschließen (nur eines der drei Flags **O_RDONLY**, **O_WRONLY** und **O_RDWR** kann gestetzt werden). Sie müssen somit zuerst die Maske **O_ACCMODE** verwenden, um die Zugriffsbits abzufragen und das Ergebnis gegen eine der drei Werte vergleichen:

```
int access_mode, ret_val;
ret_val = fcntl(fd, F_GETFL, 0);
access_mode = ret_val & O_ACCMODE;
```

F_SETFL

Setzen der Status Flags auf die im dritten Argument (*args*) angegebenen Werte. Die einzigen Flags, die verändert werden können sind **O_APPEND**, **O_ASYNC**, **O_SYNC** und **O_NONBLOCK**.

F_GETOWN

Ermitteln der Prozess- oder Gruppen-ID, welche aktuell die asynchronen Signale **SIGIO** und **SIGURG** empfängt.

F_SETOWN

Setzen der Prozess- oder Gruppen-ID, um die asynchronen Signale **SIGIO** und **SIGURG** zu empfangen. Das positive *args*-Argument definiert die Prozess- oder Gruppen-ID. Ein negatives Argument wird als absoluter Wert interpretiert.

Wie bereits erwähnt, hängt der Rückgabewert der Funktion von dem Kommando (**cmd**) ab. Es wird im Fehlerfall immer -1 zurückgegeben; jeder andere (positive) Wert zeigt die erfolgreiche Durchführung der Operation an.

Kapitel 4

Arbeiten mit Dateien und Verzeichnissen

The most beautiful thing we can experience is the mysterious. It is the source of all art and science.

ALBERT EINSTEIN (1879 - 1955)

In diesem Kapitel besprechen wir den Umgang mit Verzeichnissen und Dateien. Mit Hilfe der Struktur `stat` können wir alle Eigenschaften einer Datei abfragen und sie durch Hilfsfunktionen modifizieren.

Das Kapitel ist in folgende Teilbereiche gegliedert:

- Informationen über Dateien und Verzeichnisse
- Dateizugriffsrechte
- Dateien und Verzeichnisse verwalten
- Spezifische Funktionen für Verzeichnisse und im Dateisystem navigieren
- Temporäre Dateien verwalten
- Device Special Files

4.1 Informationen über Dateien und Verzeichnisse

Dateien und Verzeichnisse weisen neben ihren Namen viele Eigenschaften auf, die wir für bestimmte Operationen benötigen, wie zum Beispiel die i-node-Nummer oder die Größe einer Datei. Mit Hilfe der folgenden Funktionen können wir jeden Aspekt der Datei genau beleuchten:

`stat`
ruft Informationen der Datei auf Dateisystemebene ab.

`utime`
verändert mit Dateien assoziierte Datums- und Zeitangaben.

4.1.1 Die Funktion stat

Um Informationen über Dateien abzufragen, können Sie die **stat**-Funktionen verwenden. Sie rufen die Informationen ab und speichern sie in einer **stat**-Struktur, deren Zeiger als zweites Argument übergeben wird. Es werden für **stat** keine Lese-, Schreib- oder Ausführungsrechte benötigt. Allerdings ist dieses Merkmal nicht zwingend durch den POSIX-Standard vorgegeben. Vielmehr kann es sogar vorkommen, daß die Existenz der Datei durch erweiterte Zugriffsrechte (beispielsweise ACLs) der Implementierung verleugnet wird. Wenn die Datei ein symbolischer Link ist, führt **stat** die Pfadauflösung durch und gibt Informationen über die jeweilige durch den Link referenzierte Datei aus. Letztlich werden alle Dateiinformationen in der Struktur **stat** gespeichert, die in <sys/stat.h> definiert ist.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *filename, struct stat *buf);
int lstat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Rückgabewert: alle drei Funktionen geben 0 zurück oder -1 bei Fehler.

filename

Pfadname, dessen Informationen abgefragt werden sollen.

fd

File Descriptor dessen Informationen abgefragt werden sollen.

buf

Struktur, die die ermittelten Informationen speichern soll.

Während die Funktion **stat** auf reguläre Dateien angewendet wird, steht **lstat** für symbolische Links und **fstat** für File Descriptor zur Verfügung. Beachten Sie aber, daß **lstat** nicht in POSIX 1003.1-1990 definiert wurde, aber in POSIX 1003.1a. Des Weiteren wird sie von allen SYS V Implementierungen unterstützt.

Das zweite Argument ist ein Zeiger auf eine **stat**-Struktur, die nach POSIX mindestens folgende Member enthält:

```
struct stat {
    mode_t   st_mode;          /* Dateityp und Zugriffsmodi */
    ino_t    st_ino;           /* i-node-Nummer (auch: Seriennummer) */
    dev_t    st_dev;           /* Geraetenummer (Dateisystem) */
    dev_t    st_rdev;          /* Geraetenummer fuer Special Files */
    nlink_t  st_nlink;         /* Anzahl der Links */
    uid_t    st_uid;           /* Benutzer-ID des Besitzers */
    gid_t    st_gid;           /* Gruppen-ID des Besitzers */
    off_t    st_size;          /* Dateigröße in Bytes */
    time_t   st_atime;         /* letzte Zugriffszeit */
    long     st_atimensec;     /* letzte Zugriffszeit in Nanosekunden */
    time_t   st_mtime;          /* Zeit der letzten Änderung */
    long     st_mtimensec;     /* Zeit der letzten Änderung in Nanosekunden */
    time_t   st_ctime;          /* Zeit der Dateierstellung */
    long     st_ctimensec;     /* Zeit der Dateierstellung in Nanosekunden */
    long     st_blksize;        /* optimale Blockgröße für EA-Operationen */
    long     st_blocks;         /* Anzahl der allokierten Blocks für die Datei */
};
```

POSIX empfiehlt den Einsatz des Datentyps **time_t** für die Felder **st_atime**, **st_mtime** und **st_ctime**. IRIX 6.5 setzt statt dessen auf **timespec_t**. Für eine feine Granularität dieser Angaben kann auch **st_atimensec** verwendet werden.

Der Member `st_mode` enthält den Dateityp und die Zugriffsmodi. `st_mode` kann einigen Makros, die in `<sys/stat.h>` definiert sind, zugeführt werden, um herauszufinden, um welchen Typ es sich tatsächlich handelt:

S_ISREG

Der am meisten verwendete Dateityp ist die reguläre Datei, die eine beliebige Information enthalten kann. Der Kernel unterscheidet nicht zwischen binären und Textdateien; jegliche Interpretation der Inhalte ist der Anwendung überlassen.

S_ISDIR

Verzeichnisse enthalten die Namen anderer Dateien und Zeiger auf Informationen. Jeder Prozess mit Leserechten für das betreffende Verzeichnis kann die Inhalte auslesen, doch nur der Kernel kann in das Verzeichnis schreiben.

S_ISCHR

Character Special File. Ein spezieller Dateityp, der für verschiedene Operationen auf einem System vorgesehen ist.

S_ISBLK

Block Special File. Ein Dateityp, der speziell für die Arbeit mit Festspeichern vorgesehen ist.

S_ISFIFO

FIFO-Warteschlange (first-in first-out). Wird meist für die Interprozesskommunikation verwendet. FIFOs sind auch als Pipes bekannt (Abschnitt 10.2 auf Seite 263).

S_ISLNK

Symbolische Links. Ein Dateityp, der auf eine andere Datei zeigt. Nicht in POSIX.1 oder SVR4 definiert.

S_ISSOCK

Socket. Sie dienen der Netzwerkkommunikation und der Kommunikation zwischen Prozessen auf Einzelplätzen. Nicht in POSIX.1 oder SVR4 definiert.

Solange Links auf eine Datei zeigen, wird immer der `st_ctime`-Member der Datei aktualisiert und nicht der des Links. Wird ein Link entfernt werden immer auch die `st_ctime`- und `st_mtime`-Member des Verzeichnisses, das den Link enthält, aktualisiert.

Listing 4.1: Verwendung von stat

Im folgenden Beispiel wollen wir uns durch `stat` den Dateityp anzeigen lassen.

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include "header.h"
4
5 int main(int argc, char **argv) {
6     struct stat stat_buf;
7     char *output;
8     char *basename = basename_ex(argv[0]);
9
10    if (argc != 2) /* sufficient arguments */
11        err_fatal("Usage: %s <file>\n", basename);
12
13    if (lstat(argv[1], &stat_buf) < 0)
14        err_fatal("lstat failed for %s", argv[1]);
15
16    if (S_ISREG(stat_buf.st_mode))
17        output = "regular file";
18    else if (S_ISDIR(stat_buf.st_mode))

```

```

19         output = "Directory";
20         /* insert tests for other types here */
21 #ifdef S_ISLNK /* not POSIX or SVR4 */
22     else if (S_ISLNK(stat_buf.st_mode))
23         output = "Link";
24 #endif
25 #ifdef S_ISSOCK /* not POSIX or SVR4 */
26     else if (S_ISSOCK(stat_buf.st_mode))
27         output = "Socket";
28 #endif
29     else
30         output = "Modus nicht bekannt";
31
32     printf("%s\n", output);
33     exit(0);
34 }
```

Listing 4.1: xcode/stat.c - Verwendung von stat.

Ein möglicher Aufruf des Programms könnte wie folgt aussehen:

```
% filemode ~/.vimrc
regular file
% filemode ~/bin
Directory
```

□

Die Bedeutung von `stat(2)`, `lstat(2)` und `fstat(2)` wird besonders deutlich, wenn wir außergewöhnliche Sicherheitsmaßnahmen ergreifen oder Sicherheitsstandards einhalten müssen. Das folgende Beispiel öffnet eine Datei mit `open(2)` und prüft mehrfach ab, ob bestimmte Sicherheitsmerkmale zutreffen, beispielsweise, ob die zu öffnende Datei nicht durch einen symbolischen Link umgeleitet wurde.

Listing 4.2: Sicheres Öffnen von Dateien

```

1  /*
2   * paranoid-open.c           Demonstrates a very secure way to open a file
3   *                           that already exists using open and stat/fstat.
4   */
5 #include "header.h"
6
7 int open_paranoid(char *, int, mode_t);
8
9 int main(int argc, char **argv) {
10     mode_t flags = S_IRUSR | S_IRGRP;
11
12     int fd = open_paranoid("paranoid-open.c", O_RDONLY, flags);
13     /* do secure stuff here */
14
15     return (0);
16 }
17
18 int open_paranoid(char *path, int oflag, mode_t mode) {
19     char *msg = "Will not proceed";
20     char *func = "open_paranoid()";
21     int fd;
22     struct stat buf1, buf2;
23
24     /* step 1: lstat the path, check that lstat succeeds */
25     if (lstat(path, &buf1) < 0) {
26         err_fatal("%s: lstat() failed", func);
```

```

27     }
28
29     /* step 2: is path a symlink? */
30     if (S_ISLNK(buf1.st_mode)) {
31         err_fatal("%s: path is a symbolic link. %s.\n", func, msg);
32     }
33
34     /* step 3: try to open path for reading, omitting O_CREAT of course */
35     if ((fd = open(path, oflag, O_RDONLY)) < 0) {
36         err_fatal("%s: open() failed for O_RDONLY", func);
37     }
38
39     /* step 4: fstat the fd returned by open */
40     if (fstat(fd, &buf2) < 0) {
41         err_fatal("%s: fstat() failed", func);
42     }
43
44     /* step 5: compare st_ino and st_dev fields if they match */
45     if (buf1.st_ino != buf2.st_ino || buf1.st_dev != buf2.st_dev) {
46         err_fatal("%s: inode and device do not match. %s", func, msg);
47     } else {
48         if ((fd = open(path, oflag, mode)) < 0) {
49             err_fatal("%s: open() failed", func);
50         }
51     }
52
53     return fd;
54 }
```

Listing 4.2: xcode/paranoid-open.c - Sicheres Öffnen von Datei mit fstat und lstat.

Das Verfahren besteht aus insgesamt 5 Einzelschritten:

1. Zuerst rufen wir mit `lstat` Informationen über die betreffende Datei ab. Das ist notwendig, denn im Fall einer symbolischen Links, kann die Datei zur Laufzeit ausgetauscht werden, so daß wir beim Aufruf von `open(2)` nicht die eigentliche Datei öffnen, sondern eine andere, beispielweise eine vom Angreifer prepaarierte Datei. `fstat(2)` folgt nämlich symbolischen Links.
2. Wenn wir den Zugriff auf symbolische Links im Allgemeinen verhindern möchten, müssen wir mit dem Makro `S_ISLNK` prüfen, ob wir es mit einem solchen zu tun haben.
3. Im nächsten Schritt versuchen wir, die Datei zu öffnen, allerdings ohne Angabe von `O_CREAT`.
4. Die mit dem File Descriptor assoziierte Datei kann mit `fstab` auf Übereinstimmung getestet werden. Dazu rufen wir diesmal `fstat` auf.
5. Nur wenn die Felder `st_ino` und `st_dev` mit den Angaben von `lstat` übereinstimmen, ist sicher gestellt, daß wir genau die Datei öffnen, die angefragt wurde.



4.1.2 Die Funktion utime

Um die Zeit des letzten Zugriffs bzw. der letzten Änderung von Dateien zu verändern, verwenden wir `utime`.

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *filename, struct utimbuf *buf);
```

Rückgabewert: Es wird bei Erfolg 0 und -1 bei Fehler zurückgegeben.

filename

Pfadname auf den die Operation angewendet werden soll.

buf

Nimmt die ausgelesenen Zeiten für *filename* auf.

POSIX.1 organisiert utimbuf folgendermaßen:

```
struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};
```

Das Verhalten und die notwendigen Zugriffsrechte zur Durchführung der Operation hängen vom **buf**-Argument ab:

- Ist **buf** ein NULL-Zeiger, werden die Zugriffs- und Modifikationszeit auf die aktuelle Zeit gesetzt. Dazu muß:
 1. die effektive Benutzer-ID des Prozesses der ID des Besitzers (*owner*) der Datei übereinstimmen,
 2. oder der Prozess muß Schreibrechte für *filename* aufweisen.
- Wenn **buf** nicht NULL ist, werden die Zugriffs- und Modifikationszeit auf die Werte in *buf* enthaltenen Member gesetzt. Die effektive Benutzer-ID des Prozesses muß mit der der Datei übereinstimmen oder der Prozess verfügt über Superuser-Privilegien.

Listing 4.3: Einsatz von utime

Das folgende Beispiel demonstriert den Einsatz von **utime**.

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <utime.h>
5 #include "header.h"
6
7 int main(int argc, char **argv) {
8     char           *basename = basename_ex(argv[0]);
9     struct stat    stat_buf;
10    struct utimbuf utim_buf;
11
12    if (argc != 2)
13        err_fatal("Usage: %s <file>\n", basename);
14
15    /* store original time values first */
16    if (stat(argv[1], &stat_buf) < 0)
17        err_fatal("stat failed for %s\n", argv[1]);
18
19    /* O_TRUNC updates the i-node */
20    if (open(argv[1], O_RDWR | O_TRUNC) < 0)
21        err_fatal("open failed for %s\n", argv[1]);
22
23    /* copy original time values */
24    utim_buf.actime = stat_buf.st_atime;
25    utim_buf.modtime = stat_buf.st_mtime;
```

```

26      /* update file with current time */
27      if (utime(argv[1], &utim_buf) < 0)
28          err_fatal("utime failed for %s\n", argv[1]);
29
30      return (0);
31  }

```

Listing 4.3: xcode/utime.c - Einsatz von utime.

Jede Datei hat eine Zugriffs- und Modifikationszeit. Das Beispiel speichert die beiden Zeiten der Datei, modifiziert sie durch Aufruf von `open(2)` und schreibt sie anschließend wieder zurück. Somit erscheint der Eindruck, als wäre sie nicht modifiziert worden. Beachten Sie, daß `O_TRUNC` die Datei auf 0 Bytes kürzt und alle darin enthaltenen Daten verloren gehen. Verwenden Sie also zu Sicherheit eine Testdatei. □

Jedem Prozess sind mindestens sechs oder mehr IDs zugeordnet, die darüber entscheiden, ob unser Prozess berechtigt ist, bestimmte Operationen durchzuführen:

Reelle User ID, reelle Gruppen ID

Sie bestimmen, wer wir tatsächlich sind, denn sie werden aus der Gruppen- (`/etc/group`) und Passwortdatei (`/etc/passwd`) ermittelt.

Effektive User ID, effektive Gruppen ID

Sie legen die Dateizugriffsrechte und erweiterte Gruppenzugehörigkeit fest. Beispielsweise können Benutzer mehreren Gruppen angehören

Gespeicherte Set-User-IDs und gespeicherte Set-Group-IDs

Diese Bits enthalten Kopien der effektiven Benutzer- und Gruppen-IDs wenn ein Prozess ausgeführt wird.

In Abschnitt 7.4 auf Seite 173 erfahren wir, wie Benutzer- und Gruppen-IDs von Prozessen geändert werden können.

4.2 Dateizugriffsrechte

Das Thema Zugriffsrechte und damit verbundene Auswirkungen auf die tägliche Arbeit beschäftigt nicht nur ProgrammierInnen sondern auch AnwenderInnen täglich und verdient besondere Aufmerksamkeit.

Im vorangegangenen Abschnitt wurde erwähnt, das `st_mode` aus der Struktur `stat` auch die Zugriffsrechte kodiert, ohne näher darauf einzugehen. Jeder Datei sind neun Zugriffsbits zugeordnet, die alle in Tabelle 4.1 zu finden sind.

<code>st_mode</code>	Bedeutung
<code>S_IRUSR</code>	Benutzer darf lesen
<code>S_IWUSR</code>	Benutzer darf schreiben
<code>S_IXUSR</code>	Benutzer darf ausführen
<code>S_IRGRP</code>	Gruppe darf lesen
<code>S_IWGRP</code>	Gruppe darf schreiben
<code>S_IXGRP</code>	Gruppe darf ausführen
<code>S_IROTH</code>	Andere dürfen lesen
<code>S_IWOTH</code>	Andere dürfen schreiben
<code>S_IXOTH</code>	Andere dürfen ausführen

Tabelle 4.1: Flags zur Angabe der Dateizugriffsrechte.

Alle aufgelisteten Zugriffsbits sind in `<sys/stat.h>` definiert. Die meisten System definieren weit mehr als diese neun Flags. POSIX schreibt jedoch lediglich die genannten vor.

Folgende Funktionen stehen uns für die Arbeit mit Dateizugriffsrechten zur Verfügung:

`access`

findet heraus, ob ein Prozess ausreichende Zugriffsrechte für bestimmte Dateien aufweist.

`umask`

ändert die Bitmaske des Prozesses, damit neue Dateien mit bestimmten Rechten ausgestattet werden.

`chmod`

verändert die Zugriffsrechte von Dateien.

`chown`

weist einer Datei einen anderen Besitzer zu.

4.2.1 Die Funktion `access`

Wenn die `open(2)`-Funktion ausgeführt wird, überprüft der Kernel die Zugriffsrechte auf Basis der effektiven Benutzer- und Gruppen-ID. Manchmal möchten wir aber wissen, ob dem Prozess auch Zugriff basierend auf den reellen IDs gestattet ist. Dazu können wir die `access`-Funktion verwenden. Auf diese Weise kann ein `setuid`-Programm herausfinden, ob der Benutzer des Prozesses auch tatsächlich die Rechte für den Zugriff auf die Datei aufweist.

Um herauszufinden, ob ein Prozess die reellen Zugriffsrechte auf Dateien hat, können Sie `access` einsetzen.

```
#include <unistd.h>

int access(const char *filename, int mode);
```

Rückgabewert: Gibt 0 bei Erfolg und -1 bei Fehler zurück.

`filename`

Pfad auf den die Operation angewendet werden soll.

`mode`

Bitweises OR der Zugriffsrechte aus Tabelle 4.2, die geprüft werden sollen.

Zulässige Flags für das `mode`-Argument der Funktion `access`:

<code>mode</code>	Bedeutung
<code>R_OK</code>	Prüfen, ob lesen erlaubt ist.
<code>W_OK</code>	Prüfen, ob schreiben erlaubt ist.
<code>X_OK</code>	Prüfen, ob Ausführen erlaubt ist.
<code>F_OK</code>	Prüfen, ob die Datei existiert.

Tabelle 4.2: Auflistung der mode-Bits für die Funktion `access`.

Listing 4.4: Anwendung der Funktion `access`

Auch wenn wir nur den Befehl `ls -l` ausführen brauchen, um herauszufinden, welche Zugriffsrechte wir haben, gilt es nicht für Prozesse, die Zugriff auf Dateien und Verzeichnisse anfordern. Das folgende kleine Tool zeigt die Rechte des Prozesses auf Basis der reellen User- und Group-IDs an.

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include "header.h"
5
6 int main(int argc, char **argv) {
7     char *basename = basename_ex(argv[0]);
8
9     if (argc != 2)
10         err_fatal("Usage: %s <file>\n", basename);
11
12     if (access(argv[1], R_OK) < 0)
13         err_normal("access failed for %s", argv[1]);
14     else
15         printf("%s: read() OK\n", argv[1]);
16
17     /* if SUID and GID are set we can open for reading */
18     if (open(argv[1], O_RDONLY) < 0)
19         err_normal("open failed for %s", argv[1]);
20     else
21         printf("%s: open() OK\n", argv[1]);
22
23     return (0);
24 }
```

Listing 4.4: xcode/permissions.c - Anwendung der Funktion access.

□

4.2.2 Die Funktion umask

Möglicherweise sind Sie bereits mit dem Shell-Kommando `umask` vertraut. Es erwartet als Parameter eine Bitmaske, die angeibt, mit welchen Zugriffsbits neue Dateien erstellt werden sollen. Beispielsweise wird in vielen Shellscrips das Kommando

```
umask 022
```

ausgeführt.

Das heißt nicht, daß die BesitzerInnen keine Rechte haben, aber die Gruppe und Andere (*other*) schreiben dürfen, sondern `umask` schaltet die Bits ab, die zuvor aktiviert waren. Effektiv werden die Dateien also mit den Rechten 755 ausgestattet und nicht 022. Auch wenn auf den ersten Blick so scheint, als wäre hier eine einfache Subtraktion durchgeführt worden, werden die Zugriffsbits durch ein logisches ODER (OR) miteinander verknüpft.

Immer wenn ein Prozess eine neue Datei erzeugt, tritt die Bitmaske in Kraft.

4.2.2.1 Warum brauchen wir `umask`?

Stellen wir uns vor, wir sind Informatikstudent, sitzen an einem öffentlichen Computer unserer Universität und erstellen ein Programm als Hausaufgabe, welches wir in einem Verzeichnis ablegen, das uns unser Professor freundlicherweise zur Verfügung gestellt hat. Das machen nicht nur wir so, sondern auch alle anderen Studenten. Nachdem wir das Programm mit `vim(1)` erstellt haben, speichern wir es im öffentlichen Ordner unseres Professors ab. Nun kann es sein, daß die Datei anschließend über Lese- und Schreibberechtigungen für den Besitzer, die Gruppe und alle anderen Anwender verfügt. Zu einem späteren Zeitpunkt nutzt ein anderer Student das gleiche System für seine Hausaufgabe und entdeckt unser Programm. Nicht nur daß er es einfach abschreibt, sondern er löscht es auch noch, so daß wir hinterher am Tag X nichts in der Hand haben.

Hätten wir `umask(1)` verwendet, wäre das nicht passiert. Mit `umask(1)` können wir dem System mitteilen, welche Rechte eine Datei haben soll, wenn sie neu erstellt wird. In unserem drastischen Beispiel können alle die Datei lesen und schreiben, sie hat also die Rechte 666 (`rw-rw-rw`). Bevor wir `vim(1)` gestartet haben, hätten wir mit `umask(1)` die Standrechte ändern können, so daß nur wir (die Besitzer) schreiben dürfen (644, `rw-r-r-`). Der entsprechende `umask(1)`-Befehl hätte gelautet:

```
% umask      // show current mask
000          // ouch!
% umask 022  // create files with 644
% umask      // verify
022          // OK now
```

In diesem Beispiel erledigen wir zwei Dinge: zuerst fragen wir die aktive Maske ab und hinterher setzen wir die Maske auf den Wert 022.

Nun stellt sich vielleicht die Frage, was die Werte 000 und 022 mit den Rechten 666 und 644 zu tun haben. Nun, mit `umask(2)` zeigen wir an, welche Bits für die Rechtevergabe nicht gesetzt, quasi maskiert, werden sollen. Geben wir nun 022 als Maske an, so daß der Wert 2 (für Schreiben) nicht für Group und World gesetzt wird. Die 0 für Owner steht an, daß wir hier keine Veränderung der Zugriffsbits wünschen. Eine Maske von 044 gibt an, daß wir nun Schreib- aber keine Leserechte gewähren (in unserem Fall keine sinnvolle Variante). Möchten wir verhindern, daß andere Studenten unsere Hausaufgabe nicht lesen oder überschreiben können, so wäre 066 (4 = Lesen + 2 = Schreiben) ausreichend.

`umask(2)` errechnet sich in C folgendermaßen:

```
permission = requested & (~umask_value);
```

Auch wenn es in den bisherigen Ausführungen so ausgesehen hat, als würden wir eine einfache Subtraktion durchführen, so handelt es sich hier doch um eine logische XOR-Verknüpfung. Dennoch können wir der Einfachheit halber von folgenden Regeln ausgehen:

- **Masken für Dateien:** 666 - gewünschte Rechte = Maske (Beispiel: 666 - 644 [`rw-r-r-`] = 022, 666 - 600 = 066)
- **Masken für Verzeichnisse:** 777 - gewünschter Rechte = Maske (Beispiel 777 - 755 [`rwxr-xr-x`] = 022)

4.2.2.2 Gültigkeitsbereich von `umask`

Der `umask`-Wert wird vom UNIX-Kernel auf Prozessebene verwaltet, was bedeutet, daß wir in der Shell `umask(1)` ausführen können, um die gewünschte Maske für den Shell-Prozess und damit die gesamte Sitzung festzulegen, selbst wenn wir neue Shell-Prozesse starten, da die Kindsprozesse die aktuelle Maske erben.

Allerdings können `umask(1)` und der Sysmtaufruf `umask(2)` nur auf Dateisystemobjekte und nicht auf IPC-Objekte angewendet werden. Mit anderen Worten eine Maske hat keine Gültigkeit für Semaphoren, Message Queues oder Shared Memory.

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t mode);
```

Rückgabewert: gibt die zuvor gültige Bitmaske zurück

`mode`

Bitmaske mit dem neuen Zugriffsmodus für neu erzeugte Dateien.

Listing 4.5: So wird umask verwendet

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include "header.h"
5
6 int main(void) {
7     /* reset mask (777) */
8     umask(0);
9     if (creat("test.tmp", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
10        err_fatal("creat failed");
11
12     /* apply new mask */
13     umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
14
15     if (creat("test2.tmp", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
16        err_fatal("creat failed");
17
18     return (0);
19 }
```

Listing 4.5: xcode/umask.c - So wird umask verwendet.

Wird eine Bitmaske durch einen Prozess verändert, so wird bei Beendigung des Prozesses die Maske wiederhergestellt:

```
% umask
022          // das ist die aktuelle Maske der Shell
% ./myumask // der Prozess setzt die Maske zurück und ersetzt sie
% umask      // der Prozess ist zurückgekehrt
022          // die alte Maske ist weiterhin gültig
```

□

4.2.3 Die chmod-Funktionen

In direktem Zusammenhang mit umask stehen die beiden Funktionen chmod und fchmod.

Um die Zugriffsrechte existierender Dateien zu verändern, können Sie eine der beiden folgenden Funktionen einsetzen.

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *filename, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Rückgabewerte: Beide Funktionen geben 0 zurück oder -1 bei Fehler.

filename

Pfadname auf den die Operation angewendet werden soll.

fd

File Descriptor auf den die Operation angewendet werden soll.

mode

Neue Zugriffsmaske für *fd* oder *filename*.

Neben den Konstanten 4.2 *Dateizugriffsrechte* stehen für `chmod` und `fchmod` noch die beiden Set-UID- und Set-GID-Konstanten `S_ISUID` und `S_ISGID` und das sogenannte „sticky bit“ `S_ISVTX` zur Verfügung. Außerdem dürfen auch Konstanten kombiniert werden: `S_IRWXU`, `S_IRWXG` und `S_IRWXO`.

Die grundsätzliche Anwendung der `chmod`- und `fchmod`-Funktion ist sehr einfach:

```
if (chmod("/somedir/somefile", S_IRUSR | S_IXUSR | S_IRGRP | S_IXGRP) < 0) {
    err_fatal("chmod() failed");
} else {
    /* do some stuff */
}
```

oder

```
if (fchmod(fd, (stat_buf.st_mode & ~S_IROTH) | S_IXUSR) < 0) {
    err_fatal("chmod() failed");
} else {
    /* do some stuff */
}
```

Bevor wir die neuen Zugriffsrechte setzen können, rufen wir zuerst das `st_mode`-Member von `stat` ab (Abschnitt 4.1.1 auf Seite 56) und erfragen die aktuellen Rechte. Im gleichen Zug schalten wir die Leseberechtigung für alle anderen Benutzer mit `S_IROTH` ab, und setzen explizit Ausführungsrechte für den Besitzer `S_IXUSR`.

Übrigens verändert `chmod/fchmod` nicht den Zeitstempel der Dateien, denn es wird nur die i-node aktualisiert. Was eine i-node ist und welche Rolle sie im Dateisystem spielt wird in im Rahmen der Besprechung der Funktion `link` (Abschnitt 4.3.2) erläutert.

Unter Umständen kann es vorkommen, daß `chmod` zwei der Zugriffsbits automatisch entfernt:

1. Wenn das „sticky bit“ (`S_ISVTX`) einer regulären Datei gesetzt wird, wir aber keine Superuser-Privilegien haben, deaktivierten traditionelle Systeme `S_ISVTX` in `mode` automatisch.
2. Es ist möglich, daß die Gruppen-ID einer neu erzeugten Datei nicht mit der Gruppen-ID des aufrufenden Prozesses übereinstimmt. Genauer gesagt: Wenn die Gruppen-ID der neuen Datei nicht mit der effektiven Gruppen-ID oder einer der erweiterten Gruppen-IDs des Prozesses übereinstimmt, bzw. der Prozess keine Superuser-Privilegien hat, wird `S_ISGUID` automatisch deaktiviert. Somit wird verhindert, daß Benutzer eine Datei mit einem `S_ISGUID`-Bit einer Gruppe erzeugen, der sie nicht angehören.

Kurze Geschichte des Sticky Bits

UNIX System V führte das Sticky Bit ein, um die Ausführung von Programmen zu beschleunigen. Das wurde erreicht, indem die `.text`-Segmente der Programme mit gesetztem Sticky Bit, nachdem sie im Swap-Bereich ausgelagert wurden, auch nach der Beendigung des Programms in diesem verblieben. Somit konnten die Programme deutlich schneller gestartet werden, denn der Kernel transferierte das Segment nach dem Start vom Swap-Bereich in den Prozessspeicherbereich. Dieser Vorteil war leider auch mit einem Nachteil verbunden: mußten die Programme aktualisiert werden, so waren die im Swap-Bereich verbliebenen `.text`-Segmente der Programme nicht mehr kompatibel, so daß zunächst das Sticky Bit entfernt, das Programm ausgeführt und wieder beendet werden mußte, damit der Cache geleert wurde. Anschließend aktualisierte man das Programm und setzte das Sticky Bit erneut.

Nur noch wenige aktuelle Betriebssysteme unterstützen dieses Verhalten: Mac OS X, HP-UX und NetBSD. Momentan planen Sun ebenso wie OpenBSD und FreeBSD, dieses traditionelle Verhalten abzuschaffen. Linux hat das Bit noch nie unterstützt.

Allerdings hat das Sticky Bit noch eine andere Funktion, wenn es mit Verzeichnissen verwendet wird: Elemente in Verzeichnissen mit gesetztem Sticky Bit können nur vom Besitzer oder dem Superuser geändert werden. Oftmals wird diese Funktion für das `/tmp`-Verzeichnis gesetzt, damit AnwenderInnen nicht die Dateien anderer löschen können. Sie ist seit 4.3BSD (1986) Teil der meisten UNIX-Systeme.

4.2.4 Besitzrechte ändern

Den Reigen der Funktionen zur Manipulation der Sicherheitsmechanismen schließen `chown(2)`, `fchown(2)` und `lchown(2)`.

Um die User-ID und die Gruppen-ID einer Datei zu verändern, stehen uns `chown(2)`, `lchown(2)` und `fchown(2)` zur Verfügung.

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Rückgabewert: Alle drei geben bei Erfolg 0 zurück oder -1 bei Fehler.

path

Pfadname auf den die Operation angewendet werden soll.

fd

File Descriptor auf den die Operation angewendet werden soll.

owner

Spezifiziert die zu setzende User-ID.

group

Spezifiziert die zu setzende Gruppen-ID.

Alle drei Funktionen verhalten sich grundsätzlich identisch, mit der Ausnahme, daß `lchown(2)` die Datei modifiziert, die durch den symbolischen Link referenziert wird. Es sei an dieser Stelle bemerkt, daß `lchown(2)` nur in SVR4-Implementierungen Einzug gehalten hat und nicht von POSIX, bzw. Berkeley-Systemen (wie etwa BSD) definiert wird. System V basierte Systeme erlauben es allen Usern Ihre UIDs und GIDs zu verändern sofern sie dazu berechtigt sind; Berkeley basierte Systeme hingegen, verbieten das Ändern der UIDs und GIDs völlig. POSIX erlaubt beide Möglichkeiten, abhängig vom Wert der Konstante `_POSIX_CHOWN_RESTRICTED`.

Ist letztere Variante der Fall, so ist das Verhalten von `chown(2)`/`fchown(2)` folgendermaßen geregelt:

1. Nur der Superuser, bzw. ein Prozess mit Superuser-Privilegien darf UIDs und GIDs ändern.
2. Ein Prozess oder Benutzer ohne Superuser-Privilegien kann GIDs und UIDs nur ändern, wenn
 - (a) der Prozess die betreffende Datei besitzt, d.h. die effektive UID stimmt mit der UID der Datei überein.
 - (b) oder *owner* gleicht der UID der Datei und *group* gleicht entweder der effektiven GID oder erweiterten GIDs des Prozesses.

Praktisch bedeutet es, daß Sie mit `_POSIX_CHOWN_RESTRICTED` die UID nicht verändern dürfen, aber die GID auf Gruppen umstellen dürfen denen Sie selbst angehören.

4.3 Dateien und Verzeichnisse verwalten

Zu den Standardoperationen vieler Programme gehört das Erstellen, Kopieren, Umbenennen und schließlich auch das Entfernen von Dateien. Tatsächlich basieren unzählige Funktionen und Programme auf Textdateien. Glücklicherweise stehen uns zur Verwaltung von Dateien und Verzeichnissen eine ganze Reihe von Funktionen zur Verfügung.

link

erstellt Hard Links, die meist zur Bereitstellung der gleichen Daten an verschiedenen Stellen im gleichen Dateisystem dienen.

unlink

entfernt Hard Links aus dem Dateisystem.

remove

entfernt Dateien und Verzeichnisse.

rename

benennt Dateien und Verzeichnisse um.

symlink

erstellt einen symbolischen Link, der im Grunde die gleiche Aufgabe wie ein Hard Link hat, jedoch nicht auf ein bestimmtes Dateisystem beschränkt ist.

readlink

öffnet einen symbolischen Link, um ihn zu bearbeiten.

Bevor wir auf die einzelnen Funktion eingehen, lohnt eine kleine Dateisystemkunde.

4.3.1 Dateisystemkunde

Ausgehend von einem der ältesten und populärsten Dateiesysteme, dem Universal File System (UFS), wollen wir uns anschauen wie i-nodes, Links und Datenblöcke miteinander zusammenhängen. UFS ist ein direkter Abkömmling des Fast Filesystem (FFS), welches bis zur heutigen Zeit lediglich von den BSD-basierten Systemen implementiert wurde. FreeBSD, OpenBSD, NetBSD und Solaris nutzen UFS standardmäßig während Mac OS X und Linux es als Option anbieten. Linux nutzte lange Zeit ext2fs welches ebenfalls konzeptionell auf UFS basiert.

4.3.1.1 i-nodes in UFS

Im UFS werden alle Informationen über eine Datei in einer speziellen Indexdatei, der i-node, gespeichert. Dazu gehört allerdings nicht der Name der Datei, welcher wiederum im Verzeichnis gespeichert ist. Die zugehörigen Daten einer Datei wiederum sind in Datenblöcken untergebracht. Es existieren zwei Arten von i-nodes: *On-Disk* und *In-Core i-nodes*. Erstere befinden sich auf dem Medium, letztere wird erzeugt, wenn eine Datei zum Lesen oder Schreiben geöffnet wird.

Die On-Disk i-node wird von einer Datenstruktur verwaltet, die von System zu System unterschiedlich ist. Allerdings sind die i-node-Strukturen aller System meist gleicher Länge beispielsweise 128 Bytes. POSIX schreibt keinesfalls vor, wie diese Strukturen auszusehen haben, vielmehr legt es fest, welche i-node-Informationen mittels `stat(2)` ausgegeben werden müssen. Im weiteren Verlauf gehen wir daher von einer fiktiven i-node-Struktur aus, was die Sinnhaftigkeit dieser Diskussion keineswegs schmälern wird.

```
struct inode {
    o_mode      i_mode          /* mode and type of file */
    short       i_nlink         /* number of links to file */
    addr32_t   i_db[NDADDR]     /* disk block addresses */
    addr32_t   i_ib[NIADDT]     /* indirect blocks */
    ...
}
```

i_mode

Zeigt den Typ der i-node an. Typischerweise gibt es vier i-node-Typen: Special Files (IFCHR, IFBLK, IFIFO, IFSOCK), symbolische Links (IFLNK), Verzeichnisse (IFDIR) und reguläre Dateien (IFREG). Die meisten Typen haben wir in der Besprechung von `stat(2)` (Abschnitt 4.1.1) kennengelernt. Meist wird ein weiterer Nulltyp verwendet um anzugeben, daß die betreffende i-node nicht verwendet wird. Den Special Files sind keine Datenblöcke zugeordnet.

i_link

Zähler (link count) für die Anzahl der Links zu einer Datei, oder anders ausgedrückt, die Anzahl der Namen in einem Namensraum, welche einer spezifischen Datei-ID zugeordnet sind. Eine reguläre Datei hat immer einen Link Count von 1, da nur ein Name in einem Namensraum zu dieser Datei-ID zugeordnet werden kann. Ein Verzeichnis hingegen hat standardmäßig einen Link Count von 2: einer für das Verzeichnis selbst und einer für den „.“-Eintrag innerhalb des Verzeichnisses. Jedes Unterverzeichnis führt zur Erhöhung des Link Count, sichtbar ausgedrückt durch den „..“-Eintrag.

i_db

Ein Array von 12 Zeigern zu Datenblöcken. Diese sind nur indirekte Datenblöcke. Auf einem Dateisystem mit einer Blockgröße von 8192 Bytes (8 KB) können diese bis zu 98.304 (96 KB) referenzieren. Besteht die Datei lediglich aus direkten Blöcken, kann der letzte Eintrag der Datei (nicht der des Arrays) fragmentiert sein. Übersteigt die Dateigröße die des i_db-Arrays so enthält die Blockliste der Datei ausschließlich Dateisystemblöcke in voller Größe. Der Zusammenhang wird in Abbildung X dargestellt.

i_ib

Ein Array von nur drei Zeigern. Obwohl es nur drei Zeiger sind, kann die Datei eine Größe von einem Terrabyte erreichen. Wie das funktioniert? Der erste Eintrag zeigt auf einen Block der 2048 Blockadressen speichert. Eine Datei mit nur einer Indirektion kann bis zu $8192 \times (12 + 2048)$ Bytes (16 MB) enthalten. Wird mehr Speicherplatz benötigt, kommt eine zweite Indirektion ins Spiel. Der Wert 12 stammt von den 12 Zeigern in i_db

Der zweite Eintrag im i_db-Array zeigt auf 2048 Blockadressen und jede dieser Adressen zeigt auf einen anderen Block mit 2048 Blockadressen, die schließlich auf die Datenblöcke zeigen. Mit zwei Indirektionen können wir $8192 \times 12 + 2048 + (2048 \times 2048)$ Bytes (32 GB) speichern und adressieren. Durch die dritte Indirektion erhalten $8192 \times 12 + 2048 + (2048 \times 2048) + (2048 \times 2048 \times 2048) = 70.403.120.791.552$ Bytes (64 TB). Da alle Adressen als Fragmente angesprochen werden müssen, also als 31-Bit-Zahl, beträgt die maximale Größe 2 TB. Es gibt allerdings Multiterrabyte-UFS-Implementierungen, die eine Minimalfragmentierung von 8 KB vorgibt, so dass $2^{31} \times 2^{10} \times 8K$ (16 TB) zur Verfügung stehen.

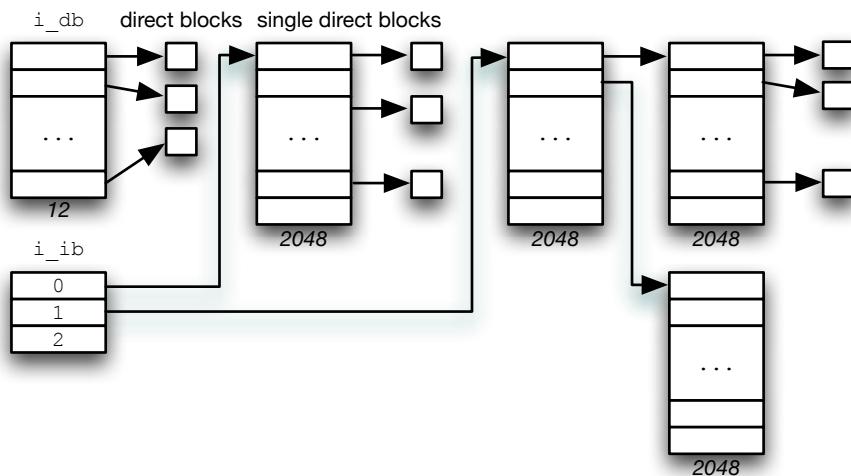


Abbildung 4.1: UFS-Blocklayout

4.3.2 Hard Links erstellen und entfernen

Dateien und Verzeichnisse können durch sogenannte Links (*Hard Links*) referenziert werden. Dabei zeigen die Links immer nur auf die betreffenden i-nodes. Wir kommen in Kürze auf i-nodes zurück. Um Links zu erzeugen, können Sie die Funktion `link` aufrufen. Mit `unlink` erreichen Sie das Gegenteil (Abschnitt 4.3.2 auf Seite 72).

Wir schreiben die Bezeichnung „i-node“ klein und mit Bindesstrich, weil wir diese Form in vielen klassischen Fachbüchern antreffen. Andere, ebenso gültige Schreibweisen, sind „inode“ und „I-Node“.

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

oldpath

Existierender Pfadname, der referenziert werden soll.

newpath

Neuer Pfadname, der auf die i-node von *oldpath* zeigt.

Wenn *newpath* bereits existiert, wird -1 zurückgegeben und *errno* entsprechend gesetzt. Die Erzeugung des Links und die Inkrementierung des Link-Zählers in der i-node muß als atomare Operation durchgeführt werden, da eine Unterbrechung zwischen den beiden Operationen den Link-Zähler beschädigen könnte, so daß die betreffende Datei irgendwann nicht mehr entfernt werden kann, weil der Link-Zähler nie 0 beträgt. Des Weiteren setzen viele Betriebssysteme voraus, daß sich *oldpath* und *newpath* auf dem gleichen Dateisystem befinden. POSIX erlaubt die Erstellung von Links über verschiedene Dateisysteme hinweg. Frühere Implementierungen überließen die Erzeugung von Links auf Verzeichnisse dem Superuser, da bei unsachgemäßem Einsatz Schleifen (*loops*) entstehen können, die das Dateisystem evtl. beschädigen. POSIX folgt dieser Philosophie, indem die Erstellung solcher Links grundsätzlich untersagt ist. Ein ähnliches, aber flexibleres Konzept wird mit *symbolischen Links* verfolgt (Abschnitt 4.3.5 auf Seite 73).

„The inode is the file“ — Kleine Dateisystemkunde

Für das Verständnis von Links und i-nodes ist es hilfreich zu verstehen, wie ein UNIX-Dateisystem konzeptionell aufgebaut ist und was der Unterschied zwischen einer i-node und einem Verzeichniseintrag ist, der auf eine i-node zeigt. Wir nehmen kein Bezug auf ein bestimmtes Dateisystem, sondern illustrieren die Zusammenhänge anhand eines verallgemeinerten Dateissystems.

Eine Festplatte ist in der Regel in mehrere kleine Einheiten, Partitionen, unterteilt, auf denen sich die Dateisysteme befinden. Abbildung 4.2 hilft uns bei der Erschließung dieser Thematik.

Die i-nodes sind Einträge mit fester Länge und enthalten fast alle Informationen einer Datei. Die beiden Verzeichnisse (*dir*) zeigen auf die gleiche i-node. Jede i-node hat einen Verweiszähler (link count), der anzeigt, wie viele Verzeichnisse auf diese i-node zeigen. Nur wenn der Verweiszähler 0 ist, kann die Datei gelöscht werden. Diese Links werden als *Hard Links* bezeichnet. In der i-node sind der Dateityp, die Zugriffsrechte, die Größe, Zeiger auf die Datenblocks (*data*), und einige andere Informationen enthalten. Diese Informationen fragen wir mit der *stat*-Funktion ab (Abschnitt 4.1.1 auf Seite 56).

Was ist nun mit Verweiszählern für ein Verzeichnis? Wenn wir ein Verzeichnis erzeugen (beispielsweise das Verzeichnis *mydir* in */tmp*), könnte es die i-node-Nummer 12345 und als Dateityp (*st_mode* in der *stat*-Struktur) „Verzeichnis“ erhalten. Der Verweiszähler eines Unterverzeichnisses beträgt dann immer 2 (das *(.)*-Verzeichnis kommt dazu). Die i-node mit der Nummer 34567 hat einen Verzeichniszähler mit dem Wert 3, denn sie wird von dem Eintrag, der das Verzeichnis selbst, dem Standardverzeichnis *(.)* und dem Verzeichniseintrag *(..)* des Verzeichnisses *mydir* referenziert

Listing 4.6 zeigt den Zusammenhang zwischen Links und Link Count.

Listing 4.6: Hard Links erzeugen

```
1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     char        *file_name = "link.file";
5     char        *link_name = "link.link";
```

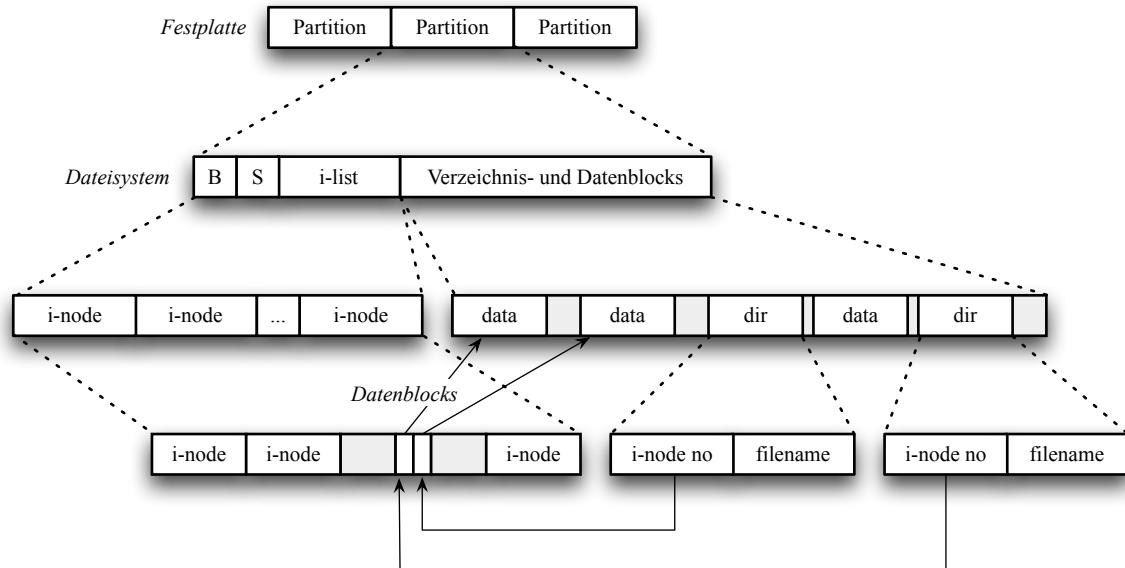


Abbildung 4.2: Festplatte, Partitionen und Dateisysteme.

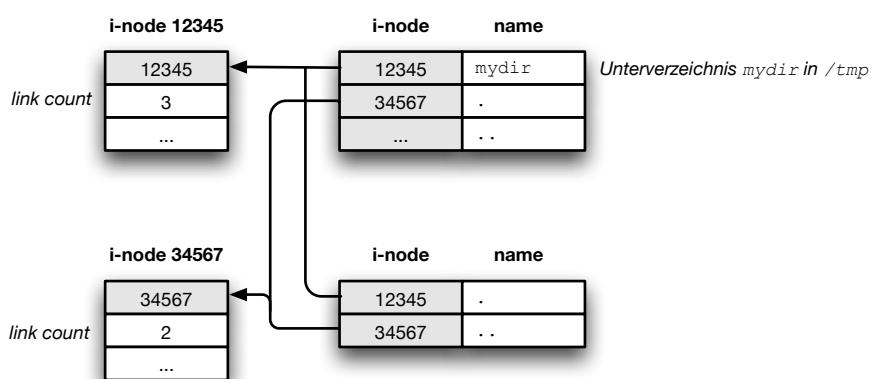


Abbildung 4.3: Verzeichniseinträge der i-nodes.

```

6     int          fd;
7     struct stat  info;
8
9     if ((fd = creat(file_name, S_IWUSR)) < 0)
10      err_fatal("creat() error");
11
12    close(fd);
13    puts("before link()");
14    stat(file_name, &info);
15    printf("\tnumber of links is %hu\n", info.st_nlink);
16
17    if (link(file_name, link_name) != 0) {
18      err_normal("link() error");
19      unlink(file_name);
20    } else {
21      puts("after link()");
22      stat(file_name, &info);
23      printf("\tnumber of links is %hu\n", info.st_nlink);
24      unlink(link_name);
25      puts("after first unlink()");
26      stat(file_name, &info);
27      printf("\tnumber of links is %hu\n", info.st_nlink);
28      unlink(file_name);
29    }
30  }
31 }
```

Listing 4.6: xcode/link.c - Hard Links erzeugen.



Um einen Link zu entfernen steht uns die Funktion `unlink` zur Verfügung.

```
#include <unistd.h>

int unlink(const char *pathname);
```

Rückgabewert: Es wird 0 bei Erfolg und -1 bei Fehler zurückgegeben.

pathname

Spezifiziert den zu entfernenden Link.

Auch hier müssen die beiden Operationen, Link entfernen und Link-Zähler der i-node dekrementieren, atomar durchgeführt werden. Tritt ein Fehler bei der Ausführung auf, wird ein Fehlercode zurückgegeben.

Listing 4.7 illustriert `unlink`.

Listing 4.7: Hard Links mit `unlink` entfernen

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     int   fd;
5     char *file_name = "unlink.file";
6
7     if ((fd = creat(file_name, S_IWUSR)) < 0) {
8         err_fatal("creat() failed");
9     } else {
10        close(fd);
11 }
```

```

12         if (unlink(fn) != 0)
13             err_fatal("unlink() failed");
14     }
15 }
```

Listing 4.7: xcode/unlink.c - Hard Links mit unlink entfernen.



4.3.3 Dateien entfernen

Links können auch mit der `remove`-Funktion entfernt werden.

```
#include <stdio.h>

int remove(const char *pathname);
```

Rückgabewert: Es wird 0 bei Erfolg und -1 bei Fehler zurückgegeben.

pathname

Pfadname auf den die Operation angewendet werden soll.

ANSI C spezifiziert die `remove`-Funktion zum Löschen der Datei. Der Name wurde von `unlink` auf `remove` geändert, da viele nicht-UNICES, die ANSI C implementieren, das Konzept der Links nicht kennen.

4.3.4 Dateien umbenennen

Um eine Datei umzubenennen können Sie die Funktion `rename` verwenden.

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

Rückgabewert: Es wird 0 bei Erfolg und -1 bei Fehler zurückgegeben.

oldpath

Existierender Dateiname, der in `newpath` umbenannt werden soll.

newpath

Neuer Dateiname für `oldpath`.

Nach ANSI C werden nur Dateien von `rename` verarbeitet. POSIX erweitert die Definition auch für Verzeichnisse. Sollte `newname` schon existieren, brauchen wir für den Zugriff die gleichen Berechtigungen, als würden wir die Dateien löschen wollen. Da wir `oldpath` ja eventuell löschen und neu mit `newpath` anlegen, brauchen wir neben den Schreibrechten auch Ausführungsberechtigungen für die Verzeichnisse in denen sich `oldname` und `newname` befinden.

4.3.5 Die Funktionen `symlink` und `readlink`

Bisher bezogen sich die Diskussionen der letzten Abschnitte auf Hard Links. Mit den beiden Funktionen `symlink` und `readlink` behandeln wir symbolische Links.

Um einen symbolischen Link zu erzeugen, können Sie `symlink` verwenden.

```
#include <unistd.h>

int symlink(const char *oldpath, const char *newpath);
```

Rückgabewert: Es wird 0 bei Erfolg und -1 bei Fehler zurückgegeben.

oldpath

Existierender Pfadname, der von **newpath** symbolisch referenziert wird.

newpath

Neuer Pfadname der **oldpath** symbolisch referenziert.

Obwohl **newpath** auf **oldpath** zeigt, ist es nicht notwendig, daß **oldpath** tatsächlich existiert, wenn **newpath** erzeugt wird. Wie mit Hard Links, dürfen Dateien und Verzeichnisse mehrere logische Namen in Form von symbolischen Links besitzen. Die Existenz eines Hard Links stellt sicher, daß die referenzierte Datei (oder das Verzeichnis) ebenfalls existiert. Eine solche Zusicherung gibt es bei symbolischen Links nicht. Tatsächlich muß **newpath** bei der Erzeugung des Links nicht einmal existieren.

Eine Anwendung von **symlink** zeigt der folgende Code-Ausschnitt:

```
if (symlink("/usr/local/bin/myprog", "/usr/bin/myprog") < 0)
    err_fatal("symlink() failed for /usr/bin/myprog");
else
    /* OK, perform some operations */
```

Hier erzeugen wir einen symbolischen Link (nämlich `/usr/bin/myprog`), der nach `/usr/local/bin/myprog` zeigt.

Zum Einlesen eines symbolischen Links ist die Funktion **readlink** notwendig, da **open(2)** dem symbolischen Link folgt und ihn selbst nicht öffnen kann.

```
#include <unistd.h>

int readlink(const char *path, char *buf, int bufsiz);
```

Rückgabewert: Anzahl der gelesenen Bytes und -1 bei Fehler.

path

Pfadname auf den die Operation angewendet werden soll.

buf

Speichert die abgerufenen Inhalte.

bufsiz

Größe des Buffers, der die abgerufenen Inhalte aufnimmt.

Diese Funktion vereint die Funktionen **open(2)**, **read(2)** und **close(2)**. Der folgende Code-Abschnitt illustriert die Anwendung:

```
int i;
char buf[PATH_MAX];

if ((i = readlink("path/to/symlink", buf, sizeof(buf)) < 0) {
    err_fatal("readlink() failed");
} else {
    buff[i] = 0;
    printf("symlink: %s\n", buf);
}
```

Beachte die Null-Terminierung des Buffers, denn **readlink** unterschlägt sie.

4.3.6 Die Funktionen `mkdir` und `rmdir`

Verzeichnisse erstellen wir mit `mkdir`.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
Rückgabewert: Es wird 0 bei Erfolg und -1 bei Fehler zurückgegeben.
```

pathname

Vollständige Pfadangabe des Verzeichnisses, das erstellt werden soll.

mode

Zugriffsrechte für das neu erstellte Verzeichnis (entweder oktal oder Makrokonstanten).

Das mit *pathname* spezifizierte leere Verzeichnis wird mit den in *mode* angegebenen Zugriffsrechten ausgestattet. Dabei kommt es oft vor, daß die gleichen Zugriffsbits wie für Dateien verwendet werden, wie z.B. lesen oder schreiben, doch die wichtigsten Rechte sind die Ausführungsrechte für Verzeichnisse, da wir sonst nicht in der Lage sind, die Inhalte des Verzeichnisses zu betrachten. Die beiden Verzeichnisse mit den Namen `.` und `..` werden automatisch erzeugt.

Folgender Code-Ausschnitt zeigt den Einsatz von `mkdir`:

```
if (mkdir("/tmp/mydir", 0755) < 0)
    err_fatal("mkdir() failed");
else
    /* do some stuff */
```

Die Benutzer-ID des Verzeichnisses wird auf die der effektiven Benutzer-ID des aufrufenden Prozesses, doch die Gruppen-ID wird auf die des Elternverzeichnisses gesetzt.

Bestehende, leere Verzeichnisse löschen wir mit `rmdir`.

```
#include <unistd.h>

int rmdir(const char *pathname);
Rückgabewert: Es wird 0 bei Erfolg und -1 bei Fehler zurückgegeben.
```

pathname

Spezifiziert den vollständigen Pfad des zu löschenen Verzeichnisses.

Durch diesen Funktionsaufruf wird der Link-Zähler der i-node auf 0 gesetzt und wenn kein anderer Prozess *pathname* referenziert wird auch der von *pathname* belegte Speicherplatz freigegeben.

Einige Plattformen, insbesondere HP-UX und IRIX erlauben nicht das Entfernen des aktuellen Arbeitsverzeichnisses (CWD) des Prozesses. Ein solcher Versuch wird mit `EINVAL` quittiert. Allerdings gestattet HP-UX das Entfernen des CWD durch einen anderen Prozess.

4.4 Verzeichnisfunktionen

Solange wir wissen, wo sich eine Datei oder ein Verzeichnis befinden, können wir mit den Funktionen des letzten Abschnitts so einiges anstellen. Wenn wir aber eine Datei finden möchten oder nur erfahren wollen, was sich in einem Verzeichnis befindet, benötigen wir andere Hilfsmittel.

Die folgenden vier Funktionen werden meist im Verbund eingesetzt:

opendir

öffnet ein Verzeichnis und fordert einen Verzeichnisstrom an.

readdir

liest den Inhalt eines Verzeichnisses ein.

rewinddir

positioniert den Cursor wieder an den Anfang des Verzeichnisstroms.

seekdir

positioniert den Cursor an einer bestimmten Position (ähnlich `lseek/fseek`).

closedir

schließt einen Verzeichnisstrom.

Jeder Benutzer mit Zugriffsberechtigungen für Verzeichnisse, darf diese lesen; schreiben darf aber nur der Kernel. Für ersteres sind mindestens Ausführungs berechtigungen (`S_ISXUSR`, `S_ISXGRP` oder `S_ISXOTH`) notwendig, für letzteres benötigen wir Schreibrechte (`S_ISWUSR`, `S_ISWGRP` oder `S_ISWOTH`). Zum Einlesen und Bearbeiten von Verzeichniseinträgen stehen einige Funktionen und Strukturen zur Verfügung, die wir uns in diesem Abschnitt etwas genauer ansehen.

POSIX.1 definiert folgende Funktionen und Strukturen für Verzeichnisse.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
long telldir(const DIR *dirp);
void seekdir(DIR *dirp, long location);
int closedir(DIR *dir);
```

*Rückgabewerte: opendir Zeiger auf eine dirent-Struktur oder NULL bei Fehler.
readdir Zeiger auf eine dirent-Struktur oder NULL am Ende des Verzeichnisses und bei Fehler.
telldir gibt die aktuelle Position im Stream oder -1 bei Fehler zurück.
rewinddir und closedir geben bei Erfolg 0 und bei Fehler -1 zurück.*

name

Pfadname, dessen Verzeichnisinformationen in DIR gespeichert werden sollen.

dirp

Zeiger auf eine DIR-Struktur, die den aktuellen Verzeichnisstrom spezifiziert.

dir

Zeiger auf eine DIR-Struktur, die Verzeichnisinformationen von *name* enthält.

location

Position (Offset) auf den die Position im Verzeichnisstrom gesetzt werden soll.

Alle UNICES definieren `dirent` in `<dirent.h>`. Die genaue Zahl der Member ist von der Implementierung abhängig. SYSV und BSD definieren `dirent` etwa so:

```
struct dirent {
    ino_t d_ino;           /* i-node number */
    char  d_name[NAME_MAX + 1]; /* null-terminated filename */
};
```

Linux hat noch einige weitere Eintäge vorgesehen und führte eine weitere Struktur für 64-Bit-Systeme ein:

```

struct dirent {
    long          d_ino;
    __kernel_off_t d_off;
    unsigned short d_reclen;
    char          d_name[256]; /* We must not include limits.h! */
};

struct dirent64 {
    __u64         d_ino;
    __s64         d_off;
    unsigned short d_reclen;
    unsigned char  d_type;
    char          d_name[256];
};

```

Beachten Sie, daß POSIX.1 nur das `name`-Member definiert.

Die `DIR`-Struktur wird intern verwendet, um grundlegende Verzeichnisinformationen zu verwalten, vergleichbar mit der Struktur `FILE` für Dateiinformationen. Sie wird durch die Standard I/O Bibliothek verwaltet. Nachdem `opendir` ein Verzeichnis eingelesen und die Informationen in `DIR` gespeichert hat, können die anderen drei Funktionen auf diese Struktur zu greifen.

Zusammenfassend haben die Funktionen folgende Aufgaben:

opendir

Öffnet ein Verzeichnisstrom (*directory stream*) für *name* und gibt einen Zeiger auf `DIR` zurück.

readdir

Liefert einen Zeiger auf eine `dirent`-Struktur zurück, die auf den nächsten Verzeichniseintrag zeigt, der durch *dir* dargestellt wird.

rewinddir

Setzt die aktuelle Position des durch *dir* repräsentieren Verzeichnisstrom auf den Anfang zurück.

telldir

Liefert die aktuelle Position in einem Verzeichnisstrom zurück. Damit können wir eine bestimmte Position festhalten und sie später, beispielsweise mit `seekdir` wieder herstellen.

seekdir

Positioniert den Cursor in einem Verzeichnisstrom an einer neuen Position. Sie könnte durch einen vorangegangenen Aufruf von `telldir` ermittelt worden sein.

closedir

Schließt den durch *dir* spezifizierten Verzeichnisstrom. Wenn `closedir` zurückkehrt, ist *dir* nicht mehr verfügbar.

Mit den beiden Funktionen `telldir` und `seekdir` können wir Positionen im Verzeichnisstrom jederzeit speichern und hinterher wieder herstellen:

```

DIR *dirp;
long offset;
... /* do some dir operations */
offset = telldir(dirp);
... /* some more dir operations */
seekdir(dirp, offset);

```

Auf einigen Systemen, beispielsweise IRIX, wird statt `long` als Datentyp für den Offset auch `off_t` verwendet, der meist per `typedef` auf `long` oder einen passenden Datentyp umdefiniert wurde.

Listing 4.8: Einfache Implementierung von ls(1)

Veranschaulichen wir die Verwendung der Funktionen anhand einer kleinen, sehr einfachen Implementierung des `ls(1)`-Kommandos.

```

1 #include <sys/types.h>
2 #include <dirent.h>
3 #include "header.h"
4
5 int main(int argc, char *argv[]) {
6     char             *basename;
7     DIR              *dir_ptr;
8     struct dirent   *dirent_ptr;
9
10    if ((basename = strrchr(argv[0], '/')) == NULL)
11        basename = argv[0];
12    else
13        basename++;
14
15    if (argc != 2)
16        err_fatal("Usage: %s <file>\n", basename);
17
18    /* initialize DIR structure */
19    if ((dir_ptr = opendir(argv[1])) == NULL)
20        err_fatal("open failed for %s", argv[1]);
21
22    /* traverse DIR structure and print entries */
23    while ((dirent_ptr = readdir(dir_ptr)) != NULL) {
24        printf("%s\n", dirent_ptr->d_name);
25    }
26
27    /* clean up what has been left behind */
28    closedir(dir_ptr);
29
30    return(0);
31 }
```

Listing 4.8: xcode/ls.c - Einfache Implementierung von ls(1)



Listing 4.9: Zeigt nur bestimmte Dateien im aktuellen Verzeichnis an

Das nachfolgende Beispiel sucht im aktuellen Verzeichnis nach Dateien, die als Parameter an das Programm übergeben werden.

```

1 #include <dirent.h>
2 #include <errno.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include "header.h"
6
7 static void lookup(const char *arg) {
8     DIR              *dirp;
9     struct dirent   *dp;
10
11    if ((dirp = opendir(".")) == NULL)
12        err_fatal("can't open .'");
13
14    do {
15        errno = 0;
16        if ((dp = readdir(dirp)) != NULL) {
17            if (strcmp(dp->d_name, arg) == 0) {
18                printf("%s found\n", arg);
19                if ((closedir(dirp)) < 0)
```

```

20             err_normal("closedir failed for %s\n", dp->d_name);
21         else
22             return;
23     }
24 }
25 } while (dp != NULL);
26
27 if (errno != 0) {
28     err_fatal("error reading directory");
29 } else {
30     printf("failed to find %s\n", arg);
31     if ((closedir(dirp)) < 0)
32         err_normal("closedir failed for %s\n", dp->d_name);
33     return;
34 }
35 }
36
37 int main(int argc, char *argv[]) {
38     int i;
39     char *basename = basename_ex(argv[0]);
40
41     if (argc < 2)
42         err_fatal("Usage: %s <file1> [file2] ...\\n", basename);
43
44     for (i = 1; i < argc; i++)
45         lookup(argv[i]);
46
47     return (0);
48 }
```

Listing 4.9: xcode/filesearch.c - Zeigt nur bestimmte Dateien im aktuellen Verzeichnis an.



4.5 Im Dateisystem navigieren

Bevor wir uns Gerätedateien (*special device files*) widmen steht noch die Besprechung der Funktionen `getcwd(3)`, `chdir` und `fchdir` aus. Jedem Prozess ist ein aktuelles Arbeitsverzeichnis (*current working directory, cwd*) zugeordnet. Alle Suchvorgänge für den Prozess beginnen in diesem relativen Pfad.

Mit den Funktionen `chdir` und `fchdir` können Sie das aktuelle Arbeitsverzeichnis eines Prozesses verändern.

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

Rückgabewert: Beide Funktionen geben 0 bei Erfolg und -1 bei Fehler zurück

path
Pfadangabe, die für den Prozess gesetzt werden soll.

fd
File Descriptor, der die neue Pfadangabe für den Prozess spezifiziert.

Entweder Sie legen das neue Arbeitsverzeichnis mit Hilfe einer Pfadangabe (`path`) oder über einen offenen File Descriptor (`fd`) fest, wobei letztere Variante nicht durch POSIX.1 definiert ist, sondern eine Erweiterung von SVR4 und BSD darstellt.

Die Funktionen verhalten sich auf den ersten Blick nicht genau so, wie Sie es vielleicht vermuten werden. Wenn wir ein Programm schreiben, daß das aktuelle Verzeichnis ändert und das Programm beendet wird, kehren wir in das aktuelle Arbeitsverzeichnis der aufrufenden Shell zurück, da es als übergeordneter Prozess ein eigenes Arbeitsverzeichnis aufweist. Wird ein Prozess beendet, so ist sein Arbeitsverzeichnis anschließend nicht mehr verfügbar. Mit dem Shell-Kommando `pwd` (*print working directory*) können wir das Arbeitsverzeichnis der Shell abfragen.

Folgendes Codefragment demonstriert die Anwendung von `chdir`:

```
if (chdir("/exports/home/steve") < 0) {
    printf("chdir failed");
}
```

Wenn wir nun folgende Session mit unserer Binary (`changedir.c`) unter Solaris 9 ausführen, sehen wir den Zusammenhang zwischen dem Arbeitsverzeichnis der Shell und des Prozesses.

```
% pwd
/usr/bin
% changedir /usr/local/bin // unser Programm wechselt das Verzeichnis
changedir: /usr/local/bin
% pwd
/usr/bin // der alte Pfad ist noch immer verfuegbar
```

Listing 4.10: Testprogramm für `changedir`

Um den Einsatz von `changedir` zu veranschaulichen, wollen wir uns an einem kleinen Programm versuchen, daß uns nach dem Aufruf von `changedir` anzeigen, daß wir tatsächlich das aktuelle Arbeitsverzeichnis gewechselt haben.

```
1 #include <unistd.h>
2 #include "header.h"
3
4 int main(int argc, char *argv[]) {
5     char *basename;
6     char *process_cwd;
7     char *pointer;
8     long path_limit;
9
10    if ((basename = strrchr(argv[0], '/')) == NULL)
11        basename = argv[0];
12    else
13        basename++;
14
15    if (argc != 2)
16        err_fatal("Usage: %s <dir>\n", basename);
17
18    /* find _PC_PATH_MAX for malloc() */
19    if ((path_limit = pathconf(argv[1], _PC_PATH_MAX)) < 0) {
20        if (errno != 0)
21            err_fatal("pathconf failed!");
22    }
23
24    if ((process_cwd = (char *)malloc((size_t)path_limit)) == NULL) {
25        err_fatal("malloc error for pathname");
26    }
27
28    if (chdir(argv[1]) < 0) {
29        printf("chdir failed");
30    } else {
31        if ((pointer = getcwd(process_cwd, (size_t)path_limit)) == NULL)
```

```

33         err_normal("getcwd failed\n");
34     printf("%s: %s [%ld]\n", basename, pointer, path_limit);
35 }
36
37     return(0);
38 }
```

Listing 4.10: xcode/changedir.c - Testprogramm für changedir

Offensichtlich hat die Änderung des Arbeitsverzeichnisses in unserem Programm nur begrenzte Wirkung. Kehrt der Prozess zurück, ist auch die Zuordnung des aktuellen Arbeitsverzeichnisses hinfällig. □

Mit der Funktion `getcwd(2)` fragen wir das aktuelle Arbeitsverzeichnis eines Prozesses ab.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Rückgabewert: Gibt buf zurück, oder NULL bei Fehler.

buf

Speichert die Pfadangabe des aufrufenden Prozesses ab.

size

Größe des allokierten Speichers für die Pfadangabe.

Das Argument `buf` muß groß genug sein, um den Pfad aufnehmen zu können. Einige Implementierungen erlauben `NULL` für `buf`, so daß `malloc(3)` die notwendigen Anzahl von Bytes dynamisch allokiert. Das ist für POSIX.1 und XPG3/4 nicht zulässig und sollte unbedingt vermieden werden. Ein Beispiel an dieser Stelle spare ich mir und verweise auf Listing 4.10.

4.6 Temporäre Dateien

Oftmals benötigen Prozesse Speicher für eine unbestimmte Menge von Daten. Gerade bei großem Datenaufkommen eignen sich temporäre Dateien. Sie werden in der Regel wieder freigegeben, wenn das Prozess die Dateien nicht mehr benötigt oder spätestens dann, wenn der Prozess beendet wird.

An der Erstellung sind eine Vielzahl von Funktionen beteiligt. Auch wenn von der Verwendung von `tmpnam` abgeraten wird, werfen wir einen Blick auf diese Funktion, denn wir treffen sie noch sehr häufig in altem Code an.

Insgesamt werden wir folgende Funktionen kennenlernen:

tmpnam

erzeugt einen String, der einen Pfad zu einer Datei repräsentiert, die noch nicht existiert.

mktemp und **mkstemp**

erstellen temporäre Dateien. Während `mktemp` einen Zeiger auf den Dateinamen zurückliefert, erhalten wir mit `mkstemp` einen File Descriptor. Warum das vorteilhaft ist, erfahren wir weiter hinten.

In diesem Abschnitt lernen wir, temporäre Dateien zu erstellen und alles wieder aufzuräumen, wenn wir fertig sind.

4.6.1 Temporäre Dateien erzeugen

Wir eröffnen den Reigen der Funktionen mit `tmpnam(3)`, von deren Anwendung abgeraten wird, da die Funktion nur einen gültigen Dateinamen erzeugt, nicht aber die Datei selbst. Das liegt in der Verantwortung der Applikation.

```
#include <stdio.h>

char *tmpnam(char *buf);
```

Rückgabewert: Zeiger auf einen gültigen Dateinamen oder NULL bei Fehler.

Wir übergeben der Funktion einen Buffer, in dem der Namen der temporären Datei abgelegt werden soll. Konnte kein Dateiname ermittelt werden, wird ein NULL-Zeiger zurückgegeben. Das könnte dann etwa so aussehen:

```
char buf [L_tmpnam], *ptr;
ptr = tmpnam(buf);
```

Die Standard E/A-Bibliothek erzeugt mit Hilfe des `P_tmpdir`-Makros einen temporären Dateinamen und legt ihn in `buf` ab. `buf` darf NULL sein, so das der Dateinamen im internen statischen Buffer von `stdio` abgelegt wird, dessen Adresse die Funktion zurückliefert. Diese Variante wurde auch in Listing 4.11 aufgegriffen.

`tmpnam` sollte nicht in neuem Code verwendet werden, da sie nur einen Dateinamen liefert, aber nicht die Datei selbst erzeugt. Zwischen Benennung einer temporären Datei und der Erzeugung liegt genug Zeit, die es einer anderen Applikation ermöglicht, in eine Datei mit gleichem Namen zu erzeugen, so daß eine klassische Race Condition entsteht.

Listing 4.11: Einsatz von `tmpnam`

```
1 #include "header.h"
2
3 int main(void) {
4     char *tmpfile, command[128];
5     FILE *file;
6
7     if ((tmpfile = tmpnam(NULL)) == NULL)
8         err_fatal("tmpnam() failed");
9
10    printf("Temp file: %s\n", tmpfile);
11
12    if ((file = fopen(tmpfile, "w")) == NULL)
13        err_fatal("open() failed for %s", tmpfile);
14
15    sprintf(command, "ls -la %s", tmpfile);
16    system(command);
17
18    fclose(file);
19    unlink(tmpfile);
20
21    return (0);
22 }
```

Listing 4.11: xcode/tmpnam.c - Einsatz von `tmpnam`.

Ein Testlauf zeigt uns, daß alles erwartungsgemäß abläuft:

```
% cc -o tmpnam tmpnam.c plibc.c
/tmp/ccuVmxwS.o(.text+0x19): In function `main':
: the use of `tmpnam' is dangerous, better use `mkstemp'
% ./tmpnam
Temp file: /tmp/filetFGBHs
-rw-r--r-- 1 steve    users          0 Feb 20 19:23 /tmp/filetFGBHs
```

Wie wir sehen, werden wir auch bei der Übersetzung darauf hingewiesen, daß wir auf `tmpnam` doch besser verzichten mögen.

Da wir `tmpnam` ohne Argument aufgerufen haben, können wir den temporären Dateinamen nur solange verwenden, bis wir `tmpnam` erneut aufrufen. Müssen wir mehrere temporäre Dateien anlegen, so sollten wir ihre Namen zwischenspeichern und daher `tmpnam` nicht ohne Argumente aufrufen.

Das Ergebnis ist zwar das gleiche, aber die Anwendung von `mktemp` unterscheidet sich doch etwas von `tmpnam`. Die Synopsis lässt diesen Schluss aber nicht zu:

```
#include <stdlib.h>

char *mktemp(char *template);
```

Rückgabewerte: Zeiger auf einen Dateinamen oder NULL bei Fehler.

Genau wie `tmpnam` erzeugt die Datei einen eindeutigen Dateinamen, allerdings auf Basis einer Vorlage, die als Argument übergeben wird. Die Vorlage hat immer das folgende Format: `/tmp/fileXXXXXX`. Die sechs Platzhalter werden mit Zufallswerten belegt und an den Aufrufer zurückgegeben. Praktisch könnte das ganze so aussehen:

```
char *template = "/tmp/fileXXXXXX", *ptr;
ptr = mktemp(template);
```

Die zurückgelieferte Adresse ist übrigens die gleiche, wie die des Arguments: `template`. Bei Fehler wird ein `NULL`-Zeiger zurück gegeben.

Kommen wir nun zu der Funktion, die mit einem wichtigen Nachteil der beiden zuletzt besprochenen aufräumt. Wir haben die Problematik bereits mehrfach angesprochen: `tmpnam` und `mktemp` erzeugen zwar einen eindeutigen Dateinamen, aber nicht die temporäre Datei, so daß im ungünstigsten Fall ein anderer Prozess in der Zwischenzeit eine Datei mit gleichem Namen anlegen könnte und damit unser temporärer Dateiname schon wieder hinfällig wäre, bevor wir ihn verwenden könnten. Die Funktion `mkstemp` ermittelt einen passenden Dateinamen und legt die Datei in einer einzigen Operation an und liefert einen File Descriptor auf die Datei zurück.

```
#include <stdlib.h>

int mkstemp(char *template);
```

Rückgabewerte: File Descriptor oder -1 bei Fehler.

Genau wie `mktemp` erwartet `mkstemp` eine Vorlage als Argument, das den gleichen Regeln der Namensgebung unterliegt. Die temporäre Datei wird standardmäßig mit den Rechten `S_IRUSR` und `S_IWUSR` ausgestattet, so daß nur der aktuelle Prozess Schreibrechte besitzt. Um eine temporäre Datei anzulegen, könnten wir prinzipiell folgendermaßen vorgehen:

```
char tempfile[PATH_MAX], writebuf[128];
int fd;
...
strcpy(tempfile, "/tmp/fileXXXXXX");
strcpy(writebuf, "Teststring");

if ((fd = mkstemp(tempfile)) < 0)
    err_fatal("mkstemp() failed for %s\n", tempfile);
```

```

if ((byteswritten = write(fd, writebuf, sizeof(writebuf)) < sizeof(writebuf) {
    err_fatal("write() failed");
...
close(fd);
unlink(tempfile);

```

Den erhaltenen File Descriptor verwenden wir anschließend zum Lesen und Schreiben der temporären Daten. Benötigen wir die temporäre Datei nicht mehr, schließen wir den Dateizugriff und löschen sie mit `unlink`.

Wenn EntwicklerInnen `mktemp` mit `mkstemp` ersetzen möchten, ist ein gutes Verständnis des zugrunde liegenden Codes notwendig. Betrachten wir folgenden Codeabschnitt, der eine temporäre Datei erzeugt und einen geöffneten Stream auf diese Datei zurückgibt.

```

char tempfile[15] = "";
FILE *sfp;

strlcpy(tempfile, "/tmp/ed.XXXXXX", sizeof tempfile);
if (mktemp(tempfile) == NULL || (sfp = fopen(tempfile, "w+")) == NULL) {
    fprintf(stderr, "%s: %s\n", tempfile, strerror(errno));
    return (NULL);
}

return (sfp);

```

Eine korrekte Konvertierung würde dann etwa so aussehen:

```

char tempfile[15] = "";
FILE *sfp;
int fd = -1;

strlcpy(tempfile, "/tmp/ed.XXXXXX", sizeof tempfile);
if ((fd = mkstemp(tempfile)) == -1 || (sfp = fdopen(fd, "w+")) == NULL) {
    if (fd != -1) {
        unlink(tempfile);
        close(fd);
    }

    fprintf(stderr, "%s: %s\n", tempfile, strerror(errno));
    return (NULL);
}

return (sfp);

```

Probleme, die auftreten können sind eher pragmatischer Natur. Hin und wieder wird `mktemp` recht früh aufgerufen, möglicherweise um eine variable frühzeitig zu initialisieren. Der Code, welcher `open(2)` oder `fopen(3)` für diese temporäre Datei aufruft, tritt viel später auf. In dem meisten Fällen wird `open(2)` ohne `O_CREAT` oder `O_EXCL` aufgerufen, so daß es zu einer Race Condition mit symbolischen Links kommen kann, was den Einsatz von `fdopen(3)` erfordert. Des Weiteren müssen wir vorsichtig sein, wenn Dateien geöffnet, geschlossen und gleich wieder geöffnet werden. Es entsteht immer ein kleines Zeitfenster, welches zu einer Race Condition führen kann. Letztlich dürfen wir nicht vergessen, die temporäre Datei auch im Fall eines Fehlers wieder zu entfernen (z.B. mit `unlink`).

4.7 Device Special Files

Während sich die bisherige Diskussion auf reguläre Dateien und Verzeichnisse beschränkte, befassen wir uns in diesem Abschnitt mit Special Device Files. Mit `stat` (Abschnitt 4.1.1) sind wir in der Lage, den Dateityp über die Member `st_dev` und `st_rdev` abzufragen. Oft führen gerade diese beiden Felder zu Verwirrungen. Dabei sind die Regeln recht einfach:

- Ein Dateisystem kann immer anhand einer *Major Device Number* und an einer *Minor Device Number* identifiziert werden. Sie werden durch den Datentyp `dev_t` kodiert. Major Device Numbers bezeichnen in diesem Zusammenhang das Gerät und die Minor Device Numbers das darauf eingerichtete Dateisystem.
- Beide Nummern können mit den Makros `major` und `minor` abgefragt werden. SYSV-Implementierungen und POSIX.1 definieren sie in `<sys/sysmacros.h>`. Unter BSD müssen Sie dafür nur `<sys/types.h>` einfügen. Aufgrund dieser Tatsache brauchen wir uns nicht darum zu kümmern, wie Device Numbers in `dev_t` organisiert sind.
- Das Member `st_dev` enthält für jede Datei eines Dateisystems die Gerätenummer des Geräts auf dem sich das Dateisystem befindet, und die i-node der betreffenden Datei.
- Nur Block Special Files und Character Special Files verfügen über ein `st_rdev`-Member. Sie enthalten die eigentliche Gerätenummer des Geräts

Anmerkungen

Frühere Systeme und auch 4.3+BSD speicherten Gerätenummern in einem 16-Bit Integer: 8 Bit für das Major Device und 8 Bit für das Minor Device. SVR4 und darauf basierende Systeme verwenden 32 Bit: 14 für das Major und 18 für das Minor Device. POSIX bestimmt das das Feld `dev_t` vorhanden sein muß, aber nicht wie sie abgespeichert werden müssen. Aus diesem Grund sind die Makros `major` und `minor` immer abhängig von der Implementierung und teilweise in verschiedenen Header-Dateien untergebracht.

Listing 4.12: Umgang mit Device Special Files

Das folgende kleine Programm zeigt die Major und Minor Device Numbers aller Dateien des als Parameter übergebenen Verzeichnisses an.

```

1 #include <dirent.h>
2 #include "header.h"
3
4 void lsdir(char *dirname);
5 void showdev(char *devpath);
6
7 int main(int argc, char *argv[]) {
8     char *basename = basename_ex(argv[0]);
9
10    if (argc != 2)
11        err_fatal("Usage: %s <path>\n", basename);
12
13    lsdir(argv[1]);
14 }
15
16 void lsdir(char *dirname) {
17     DIR          *pdir;
18     struct dirent *pdirent;
19
20     /* initialize DIR structure */
21     if ((pdir = opendir(dirname)) == NULL)
22         err_fatal("open failed for %s", dirname);
23
24     /* traverse DIR structure and print entries */
25     while ((pdirent = readdir(pdir)) != NULL) {
26         int path_len = strlen(dirname) + strlen(pdirent->d_name) + 2;
27         char full_path[path_len];
28         snprintf(full_path, path_len, "%s/%s", dirname, pdirent->d_name);
29         printf("%s: ", full_path);
30         showdev(full_path);

```

```
31     }
32
33     /* clean up what has been left behind */
34     closedir(pdir);
35 }
36
37 void showdev(char *devpath) {
38     struct stat stat_buf;
39
40     if (lstat(devpath, &stat_buf) < 0)
41         return;
42
43     printf("Device: %d:%d", major(stat_buf.st_dev), minor(stat_buf.st_dev));
44
45     if (S_ISCHR(stat_buf.st_mode))
46         printf(", [character] rdev: %d:%d\n",
47               major(stat_buf.st_rdev), minor(stat_buf.st_rdev));
48     else if (S_ISBLK(stat_buf.st_mode))
49         printf(", [block] rdev: %d:%d\n",
50               major(stat_buf.st_rdev), minor(stat_buf.st_rdev));
51 }
```

Listing 4.12: *xcode/showdev.c* - Umgang mit Device Special Files.



Kapitel 5

Die Standard E/A Bibliothek (stdio)

I hear and I forget. I see and I remember. I do and I understand.

CONFUCIUS

Jede ANSI C konforme Implementierung bringt ihre eigene *Standard I/O Library* mit. Somit treffen große Teile dieses Kapitels auch auf nicht-UNICES zu. Hauptaufgabe der Bibliothek ist die Kapselung komplexer und fehleranfälliger Vorgänge wie etwa Buffering und Speicherallokierung oder auch die richtige Auswahl von Bockgrößen usw. Entwickler können sich so auf ihre Hauptaufgaben konzentrieren und müssen sich nicht mit Details auseinandersetzen.

Die ursprüngliche Implementierung der Standard I/O Library geht auf Dennis Ritchie aus dem Jahr 1975 zurück, der eine vollständige Überarbeitung der *Portable I/O Library* von Mike Lesk vornahm. Alternativ existieren weitere erwähnenswerte Bibliotheken für Ein- und Ausgabeoperationen. Ein wichtiger Nachteil der Standard I/O Library lag in den vielen internen Datentransferoperationen und den Unterschieden in den Implementierungen auf verschiedenen Systemen. D. G. Korn und K. P. Vo [Korn90] haben viele Schwachstellen der ursprünglichen Version festgestellt. Beispielsweise konnte durch gezielte Optimierungen von `fgets(3)` und `fputs` die Anzahl der Kopieroperationen von vier auf zwei reduziert werden: (a) Transport der Daten nach einem `read(2)` oder `write(2)` vom Kernel in den Buffer und (b) vom Buffer in den Line Buffer. Die *Fast I/O Library* von AT&T aus dem Jahr 1990 umgeht das Problem, indem ein Zeiger auf den Speicherbereich zurückgegeben wird, der die eingelesenen Daten enthält. Korn's und Vo's *Safe/Fast String/File I/O Library* ist eine weitere Alternative zur Ein- und Ausgabe. Kern der SFIO ist die Ausdehnung des Streams-Konzepts auf Speicherbereiche und Verarbeitungsmodulen, die auf I/O Streams aufgesetzt werden können.

5.1 Einführung

Zentrale Bestandteile für die Ein- und Ausgabe unter UNIX sind *Streams* und *FILE*-Objekte. Wenn Sie eine Datei öffnen, erhalten Sie durch `open(2)` einen File Descriptor, der für alle nachfolgenden Ein-/Ausgabeoperationen verwendet wird. Man könnte auch sagen, wir erzeugen mit der Datei einen Stream durch die Standard I/O Library. Öffnen wir einen Stream, so erhalten wir von der Funktion `fopen(3)` einen Zeiger auf ein *FILE*-Objekt, das normalerweise als Struktur implementiert ist, die alle notwendigen Informationen zur Verwaltung des Streams enthält, wie zum Beispiel den verwendeten File Descriptor, die Buffer und deren Größen, usw. Anwendungsentwickler sind normalerweise nicht am Inhalt des *FILE*-Objekts interessiert, sondern übegeben den Zeiger an andere Funktionen, die letztendlich alle erforderlichen Operationen durchführen.

5.1.1 Standard E/A-Streams

Ein ist mit einer externen Datei, die durchaus auch ein Gerät repräsentieren kann, assoziiert. Wenn die Datei Positionierungsanfragen (*seeking*) unterstützt, wie es beispielsweise physikalische Dateien im

Gegensatz zu Terminals tun, so wird auch ein Cursor mit dem Stream assoziiert, welcher die aktuelle Position in der Datei anzeigen (zu Beginn immer Byte 0). Nur wenn die jeweilige Datei mit der Option `O_APPEND` geöffnet wurde, hängt es von der Implementierung ab, ob der Cursor am Anfang oder gleich am Ende der Datei positioniert wird. Alle E/A-Operationen entsprechen in etwa dem byteweisen lesen und schreiben mit `fgetc(3)` und `fputc(3)`.

Ungepufferte (*unbuffered*) Streams sind dazu vorgesehen, die Inhalte der Quelle dem Ziel so schnell wie möglich zu stellen. Alternativ können die Bytes, je nach Implementierung, auch zu Blöcken zusammengefaßt und anschließend ausgeliefert werden. Bei vollständig gepufferten Streams (*fully buffered*) werden alle Bytes in einen Puffer geschrieben und erst als Paket ausgeliefert, wenn er gefüllt ist. Ähnlich verhält es sich mit zeilenweise gepufferten (*line buffered*) Streams, mit dem Unterschied, dass die Bytes ausgeliefert werden, sobald ein Zeilenumbruch (*new line byte*) angetroffen wird. Des Weiteren werden die Bytes entweder wenn der Puffer voll ist oder Eingaben (die eine Transmission von Bytes erfordern) angefordert werden, übermittelt. Dieses Verhalten ist anhängig von der Implementierung und kann mit `setbuf(3)` und `setvbuf(3)` beeinflußt werden.

Jeder Datei kann der Stream entzogen werden, indem einfach mit `fclose(2)` geschlossen wird. Alle Inhalte der Ausgabestreams werden verworfen (*flushed*) bevor sie geschlossen werden. Der Wert des Zeigers auf das FILE-Objekt (auf das ich später zu sprechen komme) ist hinterher undefiniert (das gilt auch für die Standard-Streams).

Dateien dürfen mehrfach hintereinander durch das gleiche oder ein anderes Programm geöffnet und deren Inhalt abgerufen oder modifiziert werden (sofern sie Positionierungsanforderungen unterstützen). Wenn die `main`-Funktion zurückkehrt oder `exit(3)` aufgerufen wird, werden alle geöffneten Dateien geschlossen und die Ausgabestreams geleert. Diese Vorgabe ist für andere Ausführungspfade, wie sie beispielsweise durch `abort(3)` auftreten, nicht definiert und damit implementierungsabhängig.

5.1.2 File Descriptors und Standard E/A-Streams

Gehen wir nun auf die Interaktion von File Descriptors und den Standard E/A-Streams POSIX-kompatibler Implementierungen ein. Bei den folgenden Ausführungen handelt es sich um eine POSIX-Erweiterung des ISO C Standards. Die Besprechung ist sehr technisch und für das Verständnis der POSIX-Programmierung unter Unix nicht zwingend notwendig, erläutert aber die Hintergründe und Bemühungen, die zur Ausarbeitung des Standards geführt haben.

Geöffnete Dateien werden durch File Descriptors referenziert, die durch Funktionen wie `open(2)`, `fopen(3)` oder `pipe(2)`, bzw. `fpipe(2)` erzeugt werden. Oft wird bei File Descriptors auch von *Handles* geöffneter Dateien gesprochen. Eine geöffnete Datei kann durchaus mehrere Handles aufweisen.

Die Handles können mit Hilfe diverser Funktionen verwaltet werden, ohne die zugrunde liegende Dateibeschreibung zu beeinflussen. Einige Möglichkeiten Handles zu erzeugen sind `fcntl(2)`, `dup(2)`, `fdopen(3)` oder `fork(2)`; um sie zu zerstören könnten Sie beispielsweise `close(2)`, `fclose(2)` oder `exec(3)` verwenden.

Ein File Descriptor, der nie in einer Operation, die den Datei-Offset verändert (z.B. `read(2)`, `write(2)` oder `lseek`), eingesetzt wird, ist in der Regel auch kein Handle, könnte jedoch als Resultat von beispielsweise `fdopen(3)`, `dup(2)` oder `fork(2)` zu einem werden. Diese Ausnahme schließt File Descriptors von Streams, die mit `fopen(3)` oder `fdopen(3)` erzeugt wurden, nicht ein, solange sie jedoch nicht direkt von der Anwendung verwendet werden und den Datei-Offset verändern. Die Funktionen `read(2)` und `write(2)` verändert den Offset implizit, `lseek(3)` hingegen explizit.

Handles von Streams werden mit `fclose(2)` oder `freopen(2)` geschlossen (`freopen(2)` erzeugt einen neuen Stream, der nicht der gleichen Dateibeschreibung des Handles zugeordnet werden kann), oder implizit wenn der Prozess durch `exit(3)`, `abort(3)` oder ein Signal beendet wird (im Fall von `abort(3)` wird je nach Implementierung versucht, die offenen Streams mit `fclose(2)` zu schließen). File Descriptoren hingegen werden durch `close(2)`, `_exit(3)` oder die `exec`-Funktionen freigegeben.

Wenn ein Handle der aktive Handle werden soll, muß sichergestellt werden, das folgende Aktionen zwischen der letzten Nutzung des momentan aktiven Handles und der ersten Nutzung des neuen Handles ausgeführt werden:

- Jede Aktivität des Programms, die den Offset durch den früheren Handle beeinflussen, müssen ausgesetzt werden, bis der neue Handle aktiv ist.
- Beide Handles dürfen nicht in dem gleichen Prozess behandelt werden.

Beachten Sie, daß nach `fork(2)` zwei Handles existieren. Die Anwendung (Implementierung) muß sicherstellen, das beide vor dem Zugriff in einem Zustand sind, der es erlaubt einen der beiden als aktives Handle festzulegen. Somit bereitet sich die Applikation auf `fork(2)` genau so vor, als wenn es einen aktiven Handle ändern möchte. Wenn aber die einzige Aktion des Prozesses ein Aufruf von `exec(3)` oder `_exit(2)` (nicht `exit(2)`) ist, wird der Handle nie in diesem Prozess verwendet. Weitere Informationen zur Interaktion zwischen Handles und `fork(2)` finden Sie im Kapitel 7 *Prozessverwaltung*.

Wurde ein File-Offset für Positionierungsanforderungen verwendet, muß der zweite Handle durch die Implementierung mit Hilfe von `lseek(2)` oder `fseek(2)` an die richtige Stelle positionieren.

Die `exec(2)`-Funktionen können dafür sorgen, daß die geöffneten Streams zum Zeitpunkt des Aufrufs nicht mehr verfügbar sind, unabhängig davon, welche Streams oder File Descriptoren im neuen Prozessabbild (*process image*) verfügbar sind. Die `exec`-Funktionsfamilie wird in Abschnitt 7.3.1 besprochen.

Stream-Ausrichtung und Kodierungsregeln

Um dem ISO Standard IEC 9899:1999 zu entsprechen, wurde der Stream-Definition eine sogenannte Ausrichtung hinzugefügt. Nachdem ein Stream mit einer externen Datei referenziert wurde, aber noch keine E/A-Operation ausgeführt wurde, weist ein Stream noch keine Ausrichtung auf. Sollte eine Funktion auf dem Stream angewendet werden, die einen *Wide-Character* betrifft, ist der Stream von diesem Zeitpunkt an *Wide-Oriented*. Wide-Character sind Zeichen, deren Kodierung mehr als acht Bits umfaßt und somit für Zeichensätze von mehr als 256 Zeichen ausgelegt ist. Der Datentyp ist in der Regel `wchar_t` und breit genug (meist 16 Bit), um erweiterte Datensätze und den ASCII-Zeichensatz aufzunehmen. Genauso ist der Stream *Byte-Oriented*, wenn eine Byte-Operation auf den unausgerichteten Stream angewendet wird. Nur die beiden Funktionen `freopen(2)` und `fwide(3)` können die Ausrichtung verändern.

Ein erfolgreicher Aufruf von `freopen(2)` entfernt die Ausrichtung des Streams, er ist wieder unausgerichtet. Die drei Standard-Streams `stdin`, `stdout` und `stderr` sind bei Programmstart immer unausgerichtet.

Hat ein Stream erst einmal eine Orientierung erhalten, so können auf ihn keine gegensätzlichen Operationen angewendet werden. Ein Byte-Oriented Stream kann also keine Wide-Character-Operationen verarbeiten und umgekehrt. Alle anderen Stream-Operationen beeinflussen die Ausrichtung des Streams nicht und werden auch nicht von der Ausrichtung beeinflußt, es sei denn, folgende Einschränkung trifft zu: Nach einem Aufruf einer Positionierungsanforderung (beispielsweise durch `lseek(2)`) auf einen Wide-Oriented Stream, der den Cursor kurz vor das EOF-Zeichen (*end of file*) positioniert, kann eine Ausgabefunktion das Zeichen teilweise überschreiben. Somit ist der gesamte Inhalt, der nach den Bytes geschrieben wurde, undefiniert. Ein Zustand den die Implementierung abfangen sollte.

Jedem Wide-Oriented Stream ist ein `mbstate_t`-Objekt zugeordnet, der den aktuellen Status des Streams repräsentiert. Ein Aufruf von `fgetpos(3)` speichert eine Darstellung des Werte in diesem `mbstate_t`-Objekt als Teil des Wertes von `fpos_t`. Ein späterer Aufruf von `fsetpos(3)` mit dem gleichen gespeicherten `fpos_t`-Wert stellt den Wert des assoziierten `mbstate_t`-Objekts und die Position im Stream wieder her.

Implementierungen, die mehrere Kodierungen unterstützen, verbinden eine solche zusätzlich mit dem Stream. Sie kann durch die Einstellung von `LC_CTYPE` zum Zeitpunkt der Ausrichtung des Stream abgefragt werden. Genau wie die Ausrichtung, kann das Kodierungsschema eines ausgerichtete Streams nur durch einen Aufruf von `freopen(2)` verändert werden.

5.2 Eingaben, Ausgaben und Buffering

Standardmäßig sind drei Streams immer verfügbar: die Standardeingabe (*standard input*), Standardausgabe (*standard output*) und die Standardfehlerausgabe (*standard error*). Sie entsprechen den File Descriptors `STDIN_FILENO`, `STDOUT_FILENO` und `STDERR_FILENO`, die in der Regel den Nummern 0, 1 und 2 entsprechen. Die drei E/A-Streams werden normalerweise durch die FILE-Zeiger `stdin`, `stdout` und `stderr` referenziert.

Jedesmal, wenn `read(2)` oder `write(2)` aufgerufen wird, speichern die Funktionen das Ergebnis in einem Buffer. Dadurch soll die Anzahl der Aufrufe von `read(2)` und `write(2)` minimiert werden. Dabei spielt die richtige Größe des Buffers eine ganz entscheidende Rolle für die Leistungsfähigkeit von E/A-Operationen. Zum Teil versucht die Standard I/O Library automatisch die optimale Größe des Buffers für jeden zu allokalieren. Dabei unterscheiden wir zwischen drei unterschiedlichen Konzepten:

Vollständiges Buffering (*fully buffered*)

Die Ein- und Ausgabe findet erst statt, wenn der interne Buffer gefüllt wurde. In diesem Fall übernimmt die Standard I/O Library das Buffer-Management, indem sie `malloc(3)` aufruft.

Zeilenweises Buffering (*line buffered*)

Jedesmal wenn eine vollständige Zeile (bis '\n') in den Buffer gelesen wurde, findet die I/O-Operation statt. Somit sind wir in der Lage, jedes einzelne Zeichen gezielt auszugeben (z.B. mit der `putc`-Funktion). Diese Form des Buffering wird meist im Zusammenhang mit Terminals angewendet, denn E/A-Operationen finden immer nur dann statt, wenn eine ganze Zeile gelesen oder geschrieben wurde. Da die Größe des Buffers durch die Standard I/O Library festgelegt ist, kann es vorkommen, daß die Ein- oder Ausgabe stattfindet, bevor eine Zeile vollständig eingelesen oder geschrieben wurde, obwohl der Buffer noch nicht voll ist. Außerdem wird der *gesamte* Buffer geleert, wenn eine Eingabe, entweder von einem ungepufferten Stream oder von einem zeilenweise gepufferten Stream, durch die Standard I/O Library angefordert wird.

Kein Buffering (*unbuffered*)

In diesem Fall arbeitet die Standard I/O Library ohne Puffer. Das trifft zum Beispiel für `STDERR_FILENO` zu, so daß Fehlermeldungen ohne zeitliche Verzögerung ausgegeben werden.

Wie erwähnt, sind `STDIN_FILENO` und `STDOUT_FILENO` standardmäßig zeilenweise gepuffert, wenn sie mit einem Terminal verbunden sind und `STDERR_FILENO` ist ungepuffert. Gefällt uns dieses Verhalten nicht, so können wir dies jederzeit über die beiden Funktionen `setbuf(3)` und `setvbuf(3)` ändern.

Mit den beiden folgenden Funktionen beeinflussen Sie Puffermodus für einen Stream.

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);

Rückgabewert: setvbuf gibt 0 bei Erfolg und -1 bei Fehler zurück
```

`stream`

Zeiger auf einen FILE-Stream dessen Puffermodus geändert werden soll.

`buf`

Zeigt auf den Buffer mit BUFSIZ Bytes.

`mode`

Art des Puffermodus: `_IOFBF` (*fully buffered*), `_IOLBF` (*line buffered*) oder `_IONBF` (*unbuffered*).

`size`

Größe von `buf`.

Die Funktion `setvbuf(3)` kann beispielsweise nachdem `stream` mit einer geöffneten Datei assoziiert wurde, jedoch unbedingt bevor die erste E/A-Operation auf dem Stream stattgefunden hat, wobei ein fehlgeschlagener `setvbuf(3)` nicht dazu zählt. Das Argument `mode` legt fest, wie der Stream gebuffert wird:

`_IOFBF`

legt für die Ein- und Ausgabe vollständiges Buffering fest (*fully buffered*)

`_IOLBF`

legt für die Ein- und Ausgabe zeilenweises Buffering fest (*line buffered*)

`_IONBF`

alle E/A-Operationen finden ungepuffert statt (*unbuffered*)

Wenn `_IONBF` spezifiziert wird, ignoriert die Standard I/O Library die Argumente für `buf` und `size`. Für `_IOFBF` sind die Argumente für `buf` und `size` optional. Ist `buf` `NULL`, so wird automatisch ein Buffer der richtigen Größe durch `malloc(3)` allokiert. Dabei wird `st_blksize` der Struktur `stat` als optimale Größe gewählt. Sollte die Standard I/O Library keine passende Größe des Buffers ermitteln können wird auf die Konstante `BUFSIZ` herangezogen.

Listing 5.1 illustriert den Aufruf von `setbuf(3)`.

Listing 5.1: `setbuf(3)` verwenden.

```

1  #include "header.h"
2
3  int main(void) {
4      char buf[BUFSIZ];
5      char *string = "hello world\n";
6      FILE *stream;
7
8      if ((stream = fopen("myfile.dat", "wb")) == NULL)
9          err_fatal("fopen() failed");
10
11     setbuf(stream, buf);
12     fwrite(string, strlen(string), 1, stream);
13
14     if (ferror(stream))
15         err_fatal("fwrite() failed");
16
17     fclose(stream);
18
19     /* clear buffer */
20     memset(buf, 0, BUFSIZ);
21
22     if ((stream = fopen("myfile.dat", "rb")) == NULL)
23         err_fatal("fopen() failed");
24
25     fread(buf, strlen(string), 1, stream);
26
27     if (ferror(stream))
28         err_fatal("fread() failed");
29
30     fputs(buf, stdout);
31
32     fclose(stream);
33     return (0);
34 }
```

Listing 5.1: xcode/setbuf.c - Aufruf von `setbuf(3)`.



Wenn wir das Buffering ausschalten (Argument 2 von `setbuf(3)` ist `NULL`), können wir ganz interessante Effekte beobachten, wie Listing 5.2 zeigt.

Listing 5.2: Race Condition im Zusammenhang mit I/O-Buffering

```

1  #include "header.h"
2
3  static void output(char *);
4
5  int main(void) {
6      pid_t pid;
```

```

7     int i;
8
9     for (i = 0; i < 10; i++) {
10        if ((pid = fork()) < 0)
11            err_fatal("fork failed()");
12        else if( pid == 0)
13            output("Output from child\n");
14        else
15            output("Output from parent\n");
16    }
17
18    return(0);
19 }
20
21 static void output(char *str) {
22    int c;
23    char *ptr;
24
25    setbuf(stdout, NULL); /* disable buffering */
26
27    for(ptr = str; c = *ptr++;)
28        putc(c, stdout);
29 }
```

Listing 5.2: xcode/setbuf-race.c - Race Condition im Zusammenhang mit I/O-Buffering.

Wenn wir das Programm aufrufen, können Teile der Ausgabe etwa so aussehen:

```
% ./setbuf-race
...
Output from ch0output from parent
m child
Output from child
Output from parent
put from child
Output from child
Output from parent
child
...
```

Was ist hier passiert? Das Konzept sieht folgendermaßen aus: erzeuge 100 Prozesse so schnell wie möglich, die jeweils das Buffering für `stdout` deaktivieren und gib den String zeichenweise aus. Auch wenn wir `fork(2)` noch nicht besprochen haben (Abschnitt 7.2.1 gibt darüber mehr Aufschluß) erhalten wir einen Eindruck, was passiert, wenn mehrere Prozesse Zugriff auf einen ungepufferten Stream anfordern. Sobald wir das Buffering für den Stream aktivieren, tritt dieser Effekt nicht mehr auf. Probieren Sie es aus! □

Mit `setvbuf` haben wir erweiterte Einstellungsmöglichkeiten für das Buffering. Die Anwendung wird in Listing 5.3 gezeigt.

Listing 5.3: `setvbuf(3)` verwenden.

```

1 #include "header.h"
2
3 int main(void) {
4     FILE *input, *output;
5     char bufr[512];
6
7     input = fopen("file.in", "r+b");
8     output = fopen("file.out", "w");
9
10    if (setvbuf(input, bufr, _IOLBF, 512) != 0)
```

```

11         printf("failed to set up buffer for input file\n");
12
13     /*
14      * set up output stream for line buffering using space that
15      * will be obtained through an indirect call to malloc
16      */
17     if (setvbuf(output, NULL, _IOLBF, 132) != 0)
18         printf("failed to set up buffer for output file\n");
19
20     /* perform file I/O here */
21
22     /* close files */
23     fclose(input);
24     fclose(output);
25     return 0;
26 }
```

Listing 5.3: xcode/setvbuf.c - Aufruf von setvbuf(3).



5.3 Streams öffnen

Zum Öffnen von Streams stehen Ihnen drei Funktionen zur Verfügung.

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);

Rückgabewerte: Zeiger auf einen Stream oder NULL bei Fehler.
```

path
Pfadangabe die als Stream geöffnet werden soll.

filedes
File Descriptor der als Stream geöffnet werden soll.

mode
Modus des Streams.

stream
Spezifiziert den Stream, der erneut geöffnet werden soll.

Die Funktionen weisen folgende Unterschiede auf:

fopen
Öffnet einen Stream für die Datei.

fdopen
Verbindet einen Stream mit einem existierenden File Descriptor. Diese Funktion wird oftmals zum Öffnen einer Pipe oder zur Etablierung eines Kommunikationskanals.

freopen
Öffnet eine Datei für den spezifizierten Stream. Sollte sie bereits geöffnet sein, muß **stream** zuerst geschlossen werden. Typischerweise wird diese Funktion im Zusammenhang mit den vordefinierten Standardstreams **STDIN_FILENO**, **STDOUT_FILENO** und **STDERR_FILENO** angewendet.

mode	Bedeutung
r oder rb	Zum Lesen öffnen.
w oder wb	Auf 0 kürzen oder zum Schreiben erzeugen.
a oder ab	Zum Schreiben ans Ende der Datei öffnen oder zum Schreiben erzeugen.
r+ oder r+b oder rb+	Zum Lesen und Schreiben öffnen.
w+ oder w+b oder wb+	Auf 0 kürzen oder zum Schreiben und Lesen öffnen.
a+ oder a+b oder ab+	Zum Anfügen ans Ende öffnen oder erzeugen.

Tabelle 5.1: Mögliche Werte des Parameters mode der Funktionen fopen(3) oder fdopen(3).

Beachten Sie das fopen(3) und freopen durch ANSI C definiert werden, aber POSIX.1 fdopen(3) spezifiziert, da ANSI C keine File Descriptor kennt. Das Argument mode wird durch die Werte aus Tabelle 5.1 repräsentiert:

Der Buchstabe b zeigt an, daß wir binäre Daten schreiben wollen. Da der Kernel allerdings nicht zwischen Binär- und Textdaten unterscheidet ist die Angabe ohne Effekt, wurde aber zur Übereinstimmung mit ISO C eingeführt. Das type-Argument verhält sich bei fdopen(3) anders: es muß die durch filedes referenzierte Datei bereits geöffnet sein, so daß sie nicht auf 0 gekürzt wird, wenn w für type spezifiziert wurde. Das gleiche gilt natürlich auch für das a-Flag.

Beachten Sie beim Lesen *und* Schreiben folgende Einschränkungen:

- Ausgabeoperationen dürfen nicht direkt auf Eingabeoperationen folgen. Zwischendurch müssen Sie entweder rewind(3), fseek(3), fflush(3) oder fsetpos(3) anwenden.
- Eingabeoperationen dürfen direkt von Ausgabeoperationen folgen. Zwischendurch müssen Sie entweder rewind(3), fseek(3), oder fsetpos(3) anwenden oder eine Eingabeoperationen, die ein EOF aufwirft.

Listing 5.4: Text einlesen und auf stdout ausgeben

Das folgende Beispiel demonstriert die Verwendung von fopen(3). Bei der Ausführung übergeben wir dem Programm einen Dateinamen, deren Inhalt wir auf der Standardausgabe sehen möchten.

```

1 #include <sys/stat.h>
2 #include <stdio.h>
3 #include <errno.h>
4 #include "header.h"
5
6 /* use this function to obtain block size dynamically */
7 int get_block_size(int *size) {
8     struct stat stat_buf;
9
10    if (stat("/", &stat_buf) < 0) {
11        err_normal("stat failed\n");
12        return (-1);
13    } else {
14        *size = stat_buf.st_blksize;
15        return (0);
16    }
17 }
18
19 int main(int argc, char *argv[]) {
20     FILE *fp;
21     char *basename;
22     int block_size;
23
24     if ((basename = strrchr(argv[0], '/')) == NULL)

```

```

25     basename = argv[0];
26     else
27         basename++;
28
29     if (argc != 2)
30         err_fatal("Usage: %s <file>\n", basename);
31
32     if (get_block_size(&block_size) < 0)
33         err_fatal("get_buf_size failed\n");
34
35     char buf[block_size];
36
37     if ((fp = fopen(argv[1], "r")) == NULL) {
38         err_fatal("open failed");
39     }
40
41     while (fgets(buf, sizeof(buf), fp) != NULL) {
42         if (fputs(buf, stdout) == EOF) {
43             err_normal("output error");
44         }
45     }
46
47     return (0);
48 }
```

Listing 5.4: xcode/fopen.c - Text einlesen und auf stdout ausgeben.

Um einen möglichst optimalen Wert für die Puffergröße zu finden, habe ich die Funktion `get_block_size` geschrieben. Bevor wir also den Buffer einrichten (`buf`) fragen wir zuerst nach der richtigen Größe. □

Mit `fdopen` etablieren wir eine Kommunikation zwischen zwei Handles, eine Pipe. Pipes besprechen wir in Abschnitt 10.1. Dennoch wollen wir mit Listing 5.5 einen Blick auf `fdopen(3)` werfen.

Listing 5.5: Mit `fdopen(3)` eine Pipe erstellen

```

1 #include "header.h"
2
3 #define MAXLINE 512
4
5 int main(void) {
6     char line[MAXLINE];
7     FILE *fpin, *fpout;
8     pid_t child = -1;
9     int fd[2] = {0};
10    int i;
11
12    pipe(fd);
13
14    if((child = fork()) == -1) {
15        err_fatal("fork() failed");
16    } else if (child == 0) { /* child code */
17        close(fd[0]);
18
19        if ((fpout = fdopen(fd[1], "w")) == NULL)
20            err_fatal("fdopen() failed");
21
22        for(i = 0; i < 10; i++)
23            fprintf(fpout, "%s\n", "Read this!");
24
25        fclose(fpout);
26        return 0;
27 }
```

```

27     } else { /* parent code */
28         close(fd[1]);
29
30         if((fpin = fdopen(fd[0], "r")) == NULL)
31             err_fatal("fdopen() failed");
32
33         while(fgets(line, MAXLINE, fpin) != NULL)
34             printf("I'm reading... %d: %s", i++, line);
35
36         fclose(fpin);
37     }
38
39     return 0;
40 }
```

Listing 5.5: xcode/fdopen.c - Mit fdopen(3) eine Pipe erstellen.



Einen Stream schließen Sie mit der Funktion `fclose(3)`.

```
#include <stdio.h>

int fclose(FILE *stream);
```

Rückgabewert: 0 bei Erfolg und EOF bei Fehler.

Alle Daten im Buffer (auch wenn sie durch die Standardbibliothek eigenständig allokiert wurden) werden automatisch durch ein Aufruf von `flush(3)` gelöscht bevor der Stream geschlossen wird.

5.4 Stream-Operationen

Wir unterscheiden beim Lesen und Schreiben von Streams zwischen drei unterschiedlichen Formen des unformatierten I/O:

Zeichenweise Verarbeitung

Lesen und Schreiben einzelner Zeichen mit Hilfe der I/O-Funktionen, die das Buffering automatisch übernehmen.

Zeilenweise Verarbeitung

Lesen und Schreiben ganzer Zeilen mit Hilfe von `fgets(3)` und `fputs(3)`. Das Ende einer Zeile wird durch das EOL-Zeichen angezeigt. Das bedeutet, wir müssen stets die maximale Zeilenlänge spezifizieren, um Fehlverhalten zu verhindern.

Direkte Ein-/Ausgabe

Mit den Funktion `fread(3)` und `fwrite(3)` lesen und schreiben wir direkt. Dazu müssen wir die Größe der zu lesenden bzw. zu schreibenden Objekte spezifizieren. Beide Funktion sind insbesondere für binäre Ein-/Ausgaben geeignet.

5.4.1 Zeichenweise Verarbeitung

Die folgenden drei Funktionen sind für die zeichenweise Ein-/Ausgabe vorgesehen.

```
#include <stdio.h>

int getc(FILE *stream);
int fgetc(FILE *stream);
int getchar(void);
```

Rückgabewert: alle drei geben das nächste Zeichen zurück oder EOF bei Fehler.

stream

Stream auf den die Operation angewendet werden soll.

Während `fgetc(3)` als Funktion implementiert wird, finden wir `getc(3)` oft als Makro vor. In diesen Fällen kann es vorkommen, daß `getc(3)`, das Stream-Argument nicht korrekt interpretiert, wenn es Seiteneffekte aufweist. Beispielsweise würde ein Aufruf von `getc(*fp++)` nicht wie erwartet funktionieren, so daß bei dieser Konstellation zuvor `#undef getc` angebracht wäre. Alternativ könnten Sie auch `fgetc(3)` verwenden.

Wenn wir `stream` mit `fgetc(3)` lesen und das nächste Byte nicht EOF ist, ruft es das Zeichen als `unsigned int` ab, konvertiert es in einen `int`, gibt es zurück, und setzt den Cursor auf das nächste Byte. Möchten wir Multibyte-Zeichen abrufen, müssen wir `fgetc(3)` mehrfach aufrufen und ein vollständiges Zeichen zu erhalten.

Die Funktionsaufruf `getchar(3)` ist equivalent mit `getc(stdin)`. Wenn der Rückgabewert von `getchar(3)` in einem `char`-Datentyp gespeichert und anschließend mit der Integerkonstante `EOF` verglichen wird, könnte der Vergleich möglicherweise nie erfolgreich sein, da die Zeichenerweiterung von einem „schmalen“ `char` zu einem „breiten“ `int` implementierungsabhängig ist.

Alle Funktionen geben `signed int` zurück und nicht `char`, damit bei einem Fehler ebenfalls `EOF` zurückgegeben werden kann, das oftmals als `-1` implementiert ist. Das ist ungünstig, denn dadurch können wir keinen Fehlercode abfragen, so daß die beiden Hilfsfunktionen `ferror` und `feof` bemüht werden müssen. Mit ersterem erfahren wir den Fehlercode des Streams und mit letzterem können wir abfragen, ob das nächste Byte ein `EOF`-Zeichen ist.

```
#include <stdio.h>

int ferror(FILE *stream);
int feof(FILE *stream);
```

Rückgabewert: positive Werte (`true`) im Fehlerfall, sonst 0 (`false`)

stream

Stream auf den die Operation angewendet werden soll.

Selbstverständlich können Sie den assoziierten Fehlercode eines Streams zurücksetzen, indem Sie die Funktion `clearerr(3)` aufrufen.

```
#include <stdio.h>

void clearerr(FILE *stream);
```

stream

Stream auf den die Operation angewendet werden soll.

Nachdem ein Zeichen vom Stream gelesen wurde, können wir es durch `ungetc(3)` wieder „zurückschieben“. Das ist insbesondere dann sinnvoll, wenn wir herausfinden wollen, welches Zeichen als nächstes auftaucht, um darauf entsprechend reagieren zu können. Allerdings dürfen wir `EOF` nicht zurückschieben, können es aber vor `EOF` ein beliebiges Zeichen platzieren. Der nächste Aufruf von `getc(3)` oder `fgetc(3)` gibt dann genau dieses Zeichen und erst anschließend `EOF` zurück.

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Rückgabewert: gibt bei Erfolg c zurück oder EOF bei Fehler.

c

Zeichen, das zurückgeschoben werden soll.

stream

Stream auf den die Operation angewendet werden soll.

Zur zeichenweisen Ausgabe stehen uns die Funktionen `putc(3)`, `fputc(3)` und `putchar(3)` zur Verfügung.

```
#include <stdio.h>
```

```
int putc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int putchar(int c);
```

Rückgabewerte: Alle geben c zurück und EOF bei Fehler zurück.

c

Zeichen, das ausgegeben werden soll.

stream

Stream auf den die Operation angewendet werden soll.

Ähnlich wie `getc` ist `putc(3)` als Makro und `fputc(3)` als Funktion implementiert worden. Aus diesem Grund sollte das Stream-Argument keine Seiteneffekte aufweisen, wie z.B. in `putc(c, *fp++)`. In solchen Situationen ist `fgetc(3)` besser geeignet. Übrigens entspricht `putchar(c)` dem Funktionsaufruf `putc(c, stdout)`.

5.4.2 Zeilenweise Verarbeitung

Um ganze Zeilen zu verarbeiten stehen uns die beiden Funktionen `fgets(3)` und `gets(3)` zur Verfügung.

```
#include <stdio.h>
```

```
char *gets(char *buf);
char *fgets(char *buf, int bufsize, FILE *stream);
```

Rückgabewerte: Beide geben buf bei Erfolg und NULL bei Fehler oder EOF zurück.

buf

Buffer, der die eingelesene Zeile speichert.

bufsize

Größe des Buffers, der die Zeile speichert.

stream

Stream auf den die Operation angewendet werden soll.

Die Funktion `gets(3)` liest immer von `stdin` und `fgets(3)` liest von `stream`. Es wird stets vom Gebrauch von `gets(3)` abgeraten, da die Allokierung des Buffers automatisch vorgenommen wird, und durch die fehlende Angabe der Größe des Buffers ein Überlauf auftreten kann. Es gibt viele Beschreibungen, wie diese Tatsache ausgenutzt werden kann, wie es beispielsweise der Internetwurm aus dem Jahr 1988 getan hat. Auch wenn ANSI C die Unterstützung von `gets(3)` vorgibt, sollten Sie die Funktion nie verwenden.

Zur zeilenweisen Ausgabe stellt die Standardbibliothek die beiden Funktionen `fputs(3)` und `puts(3)` zur Verfügung.

```
#include <stdio.h>

int puts(const char *string);
int fputs(const char *string, FILE *stream);

Rückgabewerte: beide geben positive Werte bei Erfolg und EOF bei Fehler zurück.
```

string

Spezifiziert die auszugebende Zeichenkette als Null-terminierten String.

stream

Stream auf den die Operation angewendet werden soll.

Mit `puts` schreiben Sie den Null-terminierten String ohne das \0-Byte auf `stdout`. Anschließend wird ein Zeilenvorschub ausgegeben. Ähnlich wie `puts(3)` schreibt auch `fputs(3)` den Null-terminierten String ohne \0-Byte, allerdings in den spezifizierten Stream.

Listing 5.6: Eingaben von stdin nach stdout kopieren

Das folgende Beispiel kopiert Eingaben von `stdin` nach `stdout`. Die Funktion `get_block_size` ist in `plib.c` definiert und gibt lediglich den Wert des Members `st_blksize` der `stat`-Struktur für das Dateisystem zurück, auf dem sich der als erste Parameter übergebene Pfad befindet.

```
1 #include "header.h"
2
3 int main(int argc, char *argv[0]) {
4     int block_size;
5     char *basename = basename_ex(argv[0]);
6
7     if (get_block_size(&block_size, NULL) < 0)
8         err_fatal("%s: get_block_size failed\n", basename);
9
10    char buf[block_size];
11
12    while (fgets(buf, sizeof(buf), stdin) != NULL) {
13        if (fputs(buf, stdout) == EOF) {
14            err_fatal("%s: Output error\n", basename);
15        }
16    }
17
18    if (ferror(stdin)) {
19        err_fatal("%s: input error\n", basename);
20    }
21 }
```

Listing 5.6: xcode/inout.c - Eingaben von stdin nach stdout kopieren.

**5.4.3 Binäre E/A-Operationen**

Die bisher besprochenen Funktionen verarbeiten Zeichen, entweder zeichen- oder zeilenweise. Wenn wir binäre Datenstrukturen in Dateien speichern oder aus ihnen lesen möchten sind diese Funktionen ungeeignet. Zwar könnten wir mit `getc(3)` und `putc(3)` einzelne Elemente ausgeben oder lesen, doch würden beide bei der Verarbeitung von Nullbytes abbrechen, obwohl sich Nullbytes in unseren Datenstrukturen befinden dürfen. Zwei Funktionen, `fread(3)` und `fwrite(3)`, sollen uns beim Umgang mit Binärdaten unterstützen.

Zur Verarbeitung mit binären Daten können Sie die beiden folgenden Funktionen einsetzen.

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);

Rückgabewerte: Beide geben n zurück.
```

ptr

Zeiger auf eine Datenstruktur, die gelesen oder geschrieben werden soll.

size

Größe der zu lesenden oder zu schreibenden Datenstruktur.

n

Anzahl der zu schreibenden Elemente.

stream

Stream auf den die Operation angewendet werden soll.

Die Funktion **read(2)** liest **n** Elemente von **stream** in das Array **ptr**, dessen Größe durch **size** spezifiziert wird. Auf die gleiche Weise funktioniert **write(2)**. Für jedes Objekt, wird intern **size** mal **getc(3)** bzw. **putc(3)** aufgerufen und das Ergebnis in der eingelesenen Reihenfolge abgespeichert. Der Cursor des Streams wird jeweils um die Anzahl der gelesenen bzw. geschriebenen Bytes verschoben. Tritt ein Fehler auf, ist die Position des Cursors und, wenn nur ein Teil des Elements abgerufen oder geschrieben wurde, auch der Zustand des Elements undefined.

Das folgende Codesegment illustriert die Verwendung von **fread(3)**:

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...
```

Anwendungsbeispiele könnten etwa wie folgt aussehen: um fünf Elemente aus dem Bereich **array[4]** bis **array[9]** eines Arrays in **filedes** (ein beliebiger File Descriptor) zu speichern, rufen wir **fwrite(2)** etwa so auf:

```
int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

if (fwrite(&array[4], sizeof(int), 5, filedes) != 5) {
    printf("Write error.\n");
    exit(-1);
}
```

Ähnlich gehen wir für Strukturen vor:

```
struct {
    int id;
    float radius;
    int coords[3][2];
    char description[30];
} element;

if (fwrite(&element, sizeof(element), 1, filedes) != 1) {
    printf("Write error.\n");
    exit(-1);
}
```

Hier speichern wir eine Struktur (`element`) in `filedes` (ein beliebiger File Descriptor). In beiden Fällen prüfen wir den Erfolg der Schreiboperation, indem wir die Anzahl der zu schreibenden Elemente mit der tatsächlich geschriebenen vergleichen.

Bei der Arbeit mit binären Daten ergeben sich fundamentale Probleme. Die Daten sind meist nur auf Systemen der gleichen bzw. kompatiblen Architektur lesbar. In Zeiten der PDP-Serie war das mit Sicherheit kein Problem. Doch mit der Entstehung heterogener und vernetzter Umgebungen ist das inakzeptabel. Gründe für die Probleme finden wir in unterschiedlichen Optimierungen und Ausrichtungen (*alignments*) durch die Compiler und die unterschiedliche Darstellung von Bits verschiedener Architekturen (*big endian, little endian*).

Binärdaten plattformübergreifend lesen und schreiben

Um Binärdaten portabel zu verarbeiten gibt es unterschiedliche Ansätze, von denen aber nur wenige weite Verbreitung finden. Eine Möglichkeit für einzelne Programme ist die Entwicklung eines eigenen Dateiformats, daß die Beschreibung der Daten in binären Dateien genau definiert. Eine bessere und weniger proprietäre Variante ist XDR: *External Data Representation*.

External Data Representation ist ein abstrakter technischer Kommunikationsstandard, der von Sun Microsystems und anderen Firmen definiert wurde, um den Datenaustausch zwischen Servern und Clients hardwareunabhängig zu standardisieren. XDR ist eine Implementierung der Darstellungsschicht des OSI-Modell zur Netzwerkkommunikation und ist im RFC 1014 verbindlich festgeschrieben (1995 aktualisiert durch den RFC 1832).

Seine Hauptanwendung findet dieser Standard in der Kommunikation im SUN Network File System. Eine Reihe Programmiersprachen unterstützen das Lesen und Schreiben von XDR-Daten durch Bibliotheksfunktionen (s. z.B. `xdr_*`-Funktionen in der `libc` unter Unix für C, XDR-Modul für Perl, `xdrlib`-Modul für Python).

XDR definiert eine Repräsentation für die gebräuchlichsten Datentypen wie z.B. Integer, Strings oder Arrays, ist jedoch selbst untypisiert. Die XDR-Byte-Reihenfolge wird in den aktuellen Standards auf Big Endian festgelegt, was der Network Byte Order von TCP/IP entspricht. Eine XDR-Einheit entsprechen 4 Bytes. Gleitkommazahlen werden in einfacher und doppelter Genauigkeit nach dem IEEE 754-Standard kodiert.

Listing 5.7 zeigt ein Programm, das ein Integer-Array in eine Datei schreibt und wieder ausliest.

Listing 5.7: Binärdaten mit `fread(3)` und `fwrite(2)` lesen und schreiben

```

1 #include "header.h"
2
3 #define NUM_ROWS 5
4 #define NUM_COLS 5
5
6 void fill_array(int array[] []);
7 void print_array(int array[] []);
8
9 int main(int argc, char **argv) {
10     int my_array[NUM_ROWS][NUM_COLS] = {0};
11     FILE *stream;
12     char *file_name = "my_numbers.dat";
13     size_t osize = sizeof(int);
14     size_t ocount = NUM_ROWS * NUM_COLS;
15
16     fill_array(my_array);
17
18     printf ("Initial values of array:\n");
19     print_array(my_array);
20
21     if ((stream = fopen(file_name, "w")) == NULL)
22         err_fatal("fopen() failed for %s", file_name);
23
24     if ((fwrite (&my_array, osize, ocount, stream)) != ocount)
25         err_fatal("fwrite() failed.\n");
26     else
27         printf("Successfully wrote data to file.\n");
28

```

```

29     fclose(stream);
30
31     printf ("Clearing array...\n");
32     memset(my_array, 0, NUM_ROWS * NUM_COLS);
33
34     printf ("Now reading data from %s...\n", file_name);
35
36     if ((stream = fopen (file_name , "r")) == NULL)
37         err_fatal("fopen() failed for %s", file_name);
38
39     if ((fread(&my_array, osize, ocount, stream)) != ocount)
40         printf ("fread() failed");
41     else
42         printf ("Successfully read data from %s.\n", file_name);
43
44     printf ("Values read from file:\n");
45     print_array(my_array);
46
47     fclose (stream);
48     return (0);
49 }
50
51 void fill_array(int array[NUM_ROWS][NUM_COLS]) {
52     int row, column;
53
54     for (row = 0; row < NUM_ROWS; row++)
55         for (column = 0; column < NUM_COLS; column++)
56             array[row][column] = row + column;
57 }
58
59 void print_array(int array[NUM_ROWS][NUM_COLS]) {
60     int row, column;
61
62     for (row = 0; row < NUM_ROWS; row++) {
63         for (column = 0; column < NUM_COLS; column++)
64             printf("%d ", array[row][column]);
65
66         printf ("\n");
67     }
68 }
```

Listing 5.7: xcode/freadwrite.c - Binärdaten mit fread(3) und fwrite(2) lesen und schreiben.

Die Ausgabe des Programms sieht folgendermaßen aus:

```
% ./freadwrite
Initial values of array:
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
Successfully wrote data to file
Clearing array
Now reading data from my_numbers.dat
Successfully read data from my_numbers.dat
Values read from file:
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Wenn wir nun `cat(1)` aufrufen, um den Inhalt der Datei `my_numbers.dat` anzeigen zu lassen, sehen wir, daß die gespeicherten Zeichen nicht in Textform, sondern als Byte-Zeichenfolgen vorliegen:

```
% cat -v my_numbers.dat
^@^@^@^@^A^@^@^@^B^@^@^@^C^@^@^@^D^@^@^@^
^A^@^@^@^B^@^@^@^C^@^@^@^D^@^@^@^E^@^@^@^
^B^@^@^@^C^@^@^@^D^@^@^@^E^@^@^@^F^@^@^@^
^C^@^@^@^D^@^@^@^E^@^@^@^F^@^@^@^G^@^@^@^
^D^@^@^@^E^@^@^@^F^@^@^@^G^@^@^@^H^@^@^@^
```

Bei genauerer Betrachtung erkennen wir die Byte-Folgen (je 8 Bit) und die betreffende Zahl, die 32-Bit breit ist: `^@^@^@^@` repräsentiert 0, `^A^@^@^@` repräsentiert 1 usw. □

5.5 Positionsindikatoren von Streams steuern

Wenn wir die Position des Cursors in einem Stream verändern möchten, stehen uns zwei Möglichkeiten zur Verfügung:

1. Nach SVR7 verwenden wir `ftell(3)` und `fseek(3)`.
2. Auf anderen Systemen sollten wir auf die ANSI C Funktionen `fsetpos(3)` und `fgetpos(3)` zurückgreifen. Sie entsprechen der POSIX-Empfehlung und unterstützen Dateien oberhalb einer Größe von 2 Gigabyte.

Grundsätzlich erhöhen Sie die Portabilität Ihrer Programme durch Einsatz von `fgetpos(3)` und `fsetpos(3)`. Der Vollständigkeit halber gehen wir auch auf `fseek(3)` und `lseek(3)` ein.

Die Standardbibliothek stellt uns fünf Funktionen für den Umgang mit Streams bereit. Die ersten drei sind SVR7-Artefakte, die übrigen zwei von POSIX definiert.

```
#include <stdio.h>

long ftell(FILE *stream);
Rückgabewert: aktuelle Position des Cursors bei Erfolg, -1L bei Fehler.

int fseek(FILE *stream, long offset, int whence);
Rückgabewert: 0 bei Erfolg, -1 bei Fehler.

off_t lseek(int fildes, off_t offset, int whence);
Rückgabewert: aktuelle Position des Cursors bei Erfolg, (off_t)-1 bei Fehler.

void rewind(FILE *stream);
```

stream
Stream auf den die Operation angewendet werden soll.

offset
Versatz um den die Position des Cursors verändert werden soll.

whence
Gibt an, wie offset interpretiert werden soll.

Die Position des Cursors wird in Bytes angegeben, gezählt vom Anfang des Streams. `ftell(3)` gibt genau diese Position in Bytes zurück. Um den Cursor in einer binären Datei mit `fseek(3)` zu positionieren, müssen wir den Offset in Bytes spezifizieren. Der Parameter `whence` hat die hier gleiche Bedeutung wie von `lseek(3)`:

- SEEK_SET positioniert den Cursor am Anfang der Datei.
- SEEK_CUR bestimmt die neue Position ausgehend von der aktuellen Cursorposition.
- SEEK_END positioniert den Cursor ausgehend vom Ende der Datei.

Wenn Sie den Cursor eines Streams mit SEEK_END oder SEEK_CUR über das Dateiende hinaus positionieren, werden die übrigen Bytes bei der nächsten schreibenden Operation mit Nullen (NUL) aufgefüllt. Das Verhalten von `fseek(3)` ist bei Textdateien ein anderes. Die einzigen gültigen Werte für `whence` sind entweder 0 oder der Rückgabewert von `ftell(3)`. Übrigens wurde als Erweiterung des ISO C Standards die Funktion `ftello(3)` eingeführt, die sich absolut identisch wie `ftell(3)` verhält, allerdings einen Rückgabewert vom Typ `off_t` liefert.

Ein erfolgreicher Aufruf von `fseek(3)` setzt das EOF-Zeichen des Streams zurück und macht alle Auswirkungen von `ungetc(3)` oder `ungetwc(3)` des gleichen Streams unwirksam. Es ist durchaus möglich `fseek(3)` über das Ende des Streams hinaus zu positionieren, so daß nachfolgende Schreiboperationen an dieser Stelle durchgeführt werden und Leseoperationen in dieser Lücke Nullen (0) zurückgeben. Wird `fseek(3)` auf Geräte, die Seeking nicht unterstützen, angewendet, so ist der Offset des Streams undefined.

Wie bereits erwähnt, hat ANSI C andere Funktionen zur Positionierung des Cursors in Streams etabliert.

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

Rückgabewerte: beide geben 0 bei Erfolg und andere bei Fehler zurück.

stream

Stream auf den die Operation angewendet werden soll.

pos

Zeiger auf eine Variable vom Typ `fpos_t`.

`fgetpos(3)` speichert die Position des Cursors eines Streams in dem von `pos` referenzierten Speicherbereich. Dieser Wert kann später für ein Aufruf von `fsetpos(3)` verwendet werden.

5.6 Formatierte E/A-Operationen

Wie selbstverständlich haben wir `printf(3)` aus der Familie der `printf(3)`-Funktionsfamilie für formattiertes E/A eingesetzt. Zusammenfassend existieren acht Funktionen dieser Art: vier für Standardformatierung und vier mit Unterstützung von variablen Argumentlisten (`va_list`).

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);

Rückgabewerte: Anzahl der ausgegebenen Zeichen bei Erfolg, oder negativer Wert bei Fehler.

int sprintf(char *buf, const char *format, ...);
int snprintf(char *buf, size_t n, const char *format, ...);

Rückgabewerte: Anzahl der Zeichen, die in buf gespeichert wurden.
```

format

Auszugebende Zeichenkette.

stream

Stream auf den die Operation angewendet werden soll.

buf

Speicherplatz in dem die formatierten Zeichen gespeichert werden sollen.

n

Größe des Puffers **buf**.

Die Funktion **printf(3)** schreibt immer auf **stdin**, während **fprintf(3)** in den spezifizierten Stream schreibt. Mit **sprintf(3)** und **snprintf(3)** speichern Sie die formatierte Ausgabe in **buf**. Dabei wird das NUL-Byte automatisch an das Ende angehängt, ist aber nicht im Rückgabewert enthalten.

Die Formatierungsangaben sind folgendermaßen definiert:

[%] [Flags]<minimaler Platz><Punkt><Genauigkeit>[Typ]

wobei der Bezeichner (%) einer der folgenden Platzhalter sein kann:

%d/%i Decimal signed integer

%o Octal integer

%u Unsigned integer

%c Character

%s String (siehe weiter unten)

%f Double

%e/%E Double

%g/%G Double

%p Zeiger

%n Gesamtzahl der Zeichen dieses **printf(3)(3)**-Statements

%% Kein Argument erwartet

Der Bezeichner kann mit Hilfe der Flags genauer beschrieben werden:

- Linksbündig ausrichten

0 Felder mit 0 anstelle von Leerzeichen auffüllen

+ Vorzeichen einer Zahl mit ausgeben

blank Positive Zahlen immer mit einem Leerzeichen beginnen

Verschiedene Bedeutungen:

%#o Präfix 0 wird vorangestellt (oktal)

%#x Präfix 0x wird vorangestellt (hexadezimal)

%#X Präfix 0X wird vorangestellt (hexadezimal)

%#e Dezimalpunkt immer anzeigen

%#E Dezimalpunkt immer anzeigen

%#f Dezimalpunkt immer anzeigen

%#g Dezimalpunkt immer anzeigen, Nullen nach dem Dezimalpunkt werden nicht beseitigt.

%#G Dezimalpunkt immer anzeigen, Nullen nach dem Dezimalpunkt werden nicht beseitigt.

Das folgende kleine Beispiel zeigt die Anwendung von `fprintf(3)`

```
#include <stdio.h>

int main(void) {
    FILE * f;
    int n;
    char name [100];

    f = fopen ("names.txt", "w");

    for (n = 0 ; n < 3; n++) {
        puts("Please, enter a name: ");
        gets(name);
        fprintf(f, "Name %d [%-10.10s]\n", n, name);
    }

    fclose (f);
    return (0);
}
```

Die Ausgabe des Programms sieht das so aus:

```
% ./a.out
Please, enter a name: Steve
Name 1 [Steve      ]
Please, enter a name: Catherine
Name 2 [Catherine ]
Please, enter a name: Yoko
Name 3 [Yoko      ]
```

□

Die folgenden drei Funktionen sind den vorangegangenen drei ähnlich, allerdings werden die variablen Argumentlisten durch `arg` ersetzt.

```
#include <stdio.h>
#include <stdarg.h>

int vprintf(const char *format, va_list arg);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vsprintf(char *buf, const char *format, va_list arg);
int vsnprintf(char *buf, size_t n, const char *format, va_list arg);
```

Rückgabewerte: Anzahl der ausgegebenen Zeichen bei Erfolg, oder negativer Wert bei Fehler.

format

Auszugebende Zeichenkette.

stream

Stream auf den die Operation angewendet werden soll.

arg

Variable Argumentliste mit den gewünschten Formatierungsoptionen.

buf

Speicherplatz in dem die formatierten Zeichen gespeichert werden sollen.

n

Größe des Puffers buf.

Achten Sie darauf, den Header `<stdarg.h>` für den Datentyp `va_list` einzubinden, um Zugriff auf die variable Argumentliste nach ANSI C zu erhalten.

Formatierte Eingabeoperationen nehmen Sie mit den Funktionen `scanf(3)`, `fscanf(3)` und `sscanf(3)` vor.

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *buf, const char *format, ...);

Rückgabewerte: Anzahl der eingelesenen Zeichen oder EOF bei Fehler, bzw. Dateiende.
```

format
Angabe der Formatierungsoptionen

stream
Stream auf den die Operation angewendet werden soll.

buf
Speicherplatz für die formatierte Eingabe.

Die Funktion `fscanf(3)` liest von dem spezifizierten Eingabe-Stream, während `sscanf(3)` den String `buf` einliest und `scanf(3)` immer von `stdin` liest. Jede der drei Funktionen liest Bytes, interpretiert sie nach den in `format` spezifizierten Regeln und speichert das Ergebnis in `buf`. Das `format`-Argument zeigt die Art und Weise der Formatierung an (die Syntax bespreche ich weiter unten). Die variable Argumentliste listet die jeweiligen Variablen, in der die formatierten Daten gespeichert werden sollen, auf. Sind nicht genügend Argumente in der Liste, ist das Ergebnis undefiniert. Sind aber noch Argumente übrig, wenn alle Formatierungsangaben ausgewertet wurden, wird der Rest zwar ausgewertet, aber ignoriert.

5.6.1 Die Funktionen `scanf`, `fscanf` und `sscanf`

`scanf(3)` ist einfach in der Anwendung. Alle Zeichen, die wir einlesen, werden in das Format umgewandelt, das wir `scanf` als Argument übergeben. Möchten wir beispielsweise Zeichenketten einlesen, rufen wir `scanf("%s; string)` auf. Zahlen lesen wir mit `scanf("%d; myint)` oder einem passenden Format (z.B. `u` oder `l`) ein. Schauen wir uns ein Beispiel an.

Listing 5.8: Mit `scanf(3)` formatierte Eingaben verarbeiten

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     char string1[80], string2[80];
5     int age;
6
7     printf("Enter your name: ");
8     scanf("%s %s", string1, string2);
9     printf("Enter your age: ");
10    scanf("%d", &age);
11    printf("You (%s %s) are %d years old.\n", string1, string2, age);
12    printf("Enter a hexadecimal number: ");
13    scanf("%x", &age);
14    printf("You have entered %#x (%d).\n", age, age);
15
16    return (0);
17 }
```

Listing 5.8: xcode/scanf.c - Mit `scanf(3)` formatierte Eingaben verarbeiten.

Programmausgabe:

```
% ./scanftest
Enter your name: Steve Graegert
Enter your age: 27
You (Steve Graegert) are 27 years old.
Enter a hexadecimal number: 0x1B
You have entered 0x1B (27).
```

Der erste Fall ist der gerade besprochene: die Eingabe wird als String formatiert und in `string` gespeichert. Wenn wir zwei Formatierungen angeben, müssen wir auch zwei Zeichenketten liefern (hier: *Steve* und *Graegert*). Das gleiche gilt für die Eingabe eines Integers. Hexadezimale Zahlen lesen wir mit der Formatierungsangabe `x` ein, dabei steht es uns frei die hexadezimale (0x1B) oder einfache Notation (1B) zu verwenden.

Die Funktion `fscanf(3)` liest Daten ab der aktuellen Position im Stream und speichert in den jeweiligen Positionen des Formatspezifizierers (das 2. Argument). Die Positionen werden mit den jeweiligen Werten in der angeforderten Formatierung besetzt. Die Anzahl der Typspezifizierer muß mit der Anzahl der Argumente (der Anzahl der Variablen) übereinstimmen.

Listing 5.9: Mit `fscanf(3)` formatierte Eingaben verarbeiten.

```
1 #include <stdio.h>
2
3 int main (int argc, char **argv) {
4     char s[80];
5     float f;
6     FILE *stream;
7
8     stream = fopen ("file.txt", "w+");
9     fprintf(stream, "%f %s", 3.1416, "PI");
10    rewind(stream);
11    fscanf(stream, "%f", &f);
12    fscanf(stream, "%s", s);
13    fclose(stream);
14    printf("Input read: %f and %s \n", f, s);
15
16    return (0);
17 }
```

Listing 5.9: xcode/fscanf.c - Mit `fscanf(3)` formatierte Eingaben verarbeiten.

Programmausgabe:

```
% ./fcanftest
Input read: 3.141600 and PI
```

Das Programm erstellt eine Textdatei und schreibt einen Float-Wert und eine Zeichenkette in die Datei. Anschließend wird der Stream zurückgesetzt und beide Werte mit `scanf(3)` ausgelesen. □

Die Funktion `sscanf(2)` ist etwas komplizierter, aber umso leistungsfähiger. Sie untersucht einen String und extrahiert die Werte nach einem vorgegebenen Muster, das wir als eine Reihe von Formatspezifizierern definieren. Das Ergebnis speichert sie in den Argumenten. Die müssen selbstverständlich Variablen sein. Das folgende Code-Segment zeigt ein einfaches Beispiel:

```
float f;
int i;
char *string = "Float -3.6 und Integer 17";
sscanf(string, "Float %f und Integer %d", f, i);
```

Der Formatspezifizierer (Argument 2) besteht aus genau den gleichen Zeichen des Strings und enthält Formatierungen für die Stellen, die wir extrahieren möchten. Wichtig ist, das jedes Zeichen im Formatspezifizierer von Bedeutung ist, auch ein Leerzeichen. Das nächste Beispiel ist etwas komplizierter.

Listing 5.10: Mit `sscanf(3)` formatierte Eingaben verarbeiten

```

1 #include <stdio.h>
2
3 int main (int argc, char **argv) {
4     char *string = "Bart is 10 years old";
5     char s[20];
6     int i;
7
8     sscanf(string, "%s %*s %d", s, &i);
9     printf ("%s == %d\n", s, i);
10
11     return (0);
12 }
```

Listing 5.10: xcode/sscanf.c - Mit `sscanf(3)` formatierte Eingaben verarbeiten.

Programmausgabe:

```
% ./sscanftest
Bart == 10
```

Die Formatierung `%*` zeigt eine Auslassung an und bedeutet *lass' den String aus und speicher ihn nicht in einer Variable*. Das funktioniert nur mit Zeichenketten, die nicht durch ein Leerzeichen getrennt sind. Dafür bräuchten wir eine zweite Auslassung.

Ein paar weitere Beispiele für `sscanf`:

- `sscanf('Homer', 'H%s', a);`
Der Aufruf gibt 1 zurück und die Variable enthält `omer`.
- `sscanf('0815test', '%d%s', a, b);`
Der Aufruf gibt 2 zurück und `a` wird den Wert 815 und `b` wird `test` enthalten.
- `sscanf(str, "ein %s", str);`
Der Aufruf entfernt "ein" aus der Zeichenkette. Ist "ein" nicht der Anfang der Zeichenkette, wird `str` nicht verändert.

5.6.2 Format- und Typspezifizierer

Abschließend wollen wir uns noch einen kurzen Überblick über Format- und Typspezifizierer verschaffen. Formatspezifizierer haben immer das Format `[%]<*><Breite><Modifizierer>[Typ]`. Oftmals wird statt von Formatspezifizierern auch von Konvertierungsspezifizierern gesprochen.

Jeder Konvertierungsspezifizierer wird durch das %-Zeichen oder die Sequenz `%n$` eingeleitet und mit folgenden Zeichen in einer bestimmten Reihenfolge spezifiziert:

- optionaler Auslassungsoperator "(*)" (damit können Zeichenketten übersprungen werden)
- optionaler dezimaler Integer, der die Feldlänge definiert (maximale Anzahl der zu lesenden Zeichen).
- optionaler Längenmodifizierer, der die Größe des aufzunehmenden Objekts angibt
- ein Typspezifizierer, der den Zieltyp der Konvertierung bestimmt und anzeigt, wie die Daten gelesen werden sollen.

Folgende Längenmodifizierer stehen zur Verfügung:

`hh`

zeigt an, daß einer der Konvertierungsspezifizierer `d` (Dezimalwert zur Basis 10), `i` (Ganzahl zur Basis 10), `o` (Oktalwert), `u` (vorzeichenloser Wert), `x/X` (Hexadezimalwert), oder `n` auf ein Zeigerargument vom Typ `signed char` oder `unsigned char` angewendet werden soll.

`h`

zeigt an, daß einer der Konvertierungsspezifizierer `d`, `i`, `o`, `u`, `x/X` oder `n` auf ein Zeigerargument vom Typ `short` oder `unsigned short` angewendet werden soll.

`l`

zeigt an, daß einer der Konvertierungsspezifizierer `d`, `i`, `o`, `u`, `x/X` oder `n` auf ein Zeigerargument vom Typ `long` oder `unsigned long` angewendet werden soll; oder daß einer der Konvertierungsspezifizierer `a`, `A`, `e`, `E`, `f`, `F`, `g` oder `G` auf ein Zeigerargument vom Typ `double` angewendet werden soll; oder daß einer der Konvertierungsspezifizierer `c`, `s` oder auf ein Zeigerargument vom Typ `wchar_t` angewendet werden soll.

`ll`

zeigt an, daß einer der Konvertierungsspezifizierer `d`, `i`, `o`, `u`, `x`, `X` oder `n` auf ein Zeigerargument vom Typ `short` oder `unsigned short` angewendet werden soll.

Weitere Informationen finden Sie in der Manpage von `printf(3)`.

5.7 Von langsamen und schnellen E/A-Operationen

Wir haben in den vorangegangenen Ausführungen hin und wieder von langsamen und schnellen E/A-Operationen gesprochen. Die langsamen Operationen können für einen undefinierten Zeitraum im Zustand Blocking verweilen, während die anderen (schnellen) Operationen das nicht tun.

Das Lesen oder Schreiben von Dateien kann unendlich lange dauern, beispielsweise wenn die Daten nicht zur Verfügung stehen. Das ist besonders im Zusammenhang mit Pipes, Terminals oder Netzwerkoperationen der Fall. Auch das Öffnen bestimmter Datei kann unendlich lange dauern, beispielsweise wenn wir versuchen, ein Terminal anzufordern, daß selbst auf die Bereitschaft eines Modems wartet. Es gibt noch viele solcher Beispiele, die als langsame E/A-Operationen bezeichnet werden.

Eigentlich gehören E/A-Operationen, die den Zugriff auf Speichermedien wie Festplatten oder Disketten beinhalten, nicht in die Klasse der langsamen Operationen, denn sie blockieren den Aufrufer nur für einen bestimmten Zeitraum und nicht für immer (es sei denn, das Medium ist fehlerhaft oder der Controller des Geräts arbeitet nicht korrekt).

Wenn wir von E/A sprechen, die nicht blockieren, dann meinen wir beispielsweise Aufrufe von `read(2)`, `write(2)` oder `open(2)`, die sofort wieder zurückkehren, auch wenn die Anfrage nicht vollständig durchgeführt werden konnte. Versuchen wir eine Datei von der Größe eines Megabytes in einen Buffer von 512 Kilobytes zu lesen, kommt es zu einem *Short Read*, weil wir ja nur die Hälfte eingelesen haben und nicht die ganze Datei. Umgekehrt tritt ein *Short Write* auf, wenn nur ein Teil der zu schreibenden Daten in die Datei überführt werden konnte. In beiden Fällen, würden `read(2)` oder `write(2)` zurückkehren, sobald ein Problem auftritt.

Um E/A ohne Blockieren zu verwenden, brauchen wir bei einem Aufruf von `open(2)` nur die Option `O_NONBLOCK` anzugeben. Ist der File Descriptor bereits vorhanden, können wir `fcntl(2)` verwenden, um den Modus im nachhinein zu ändern.

Listing 5.11: Schreiben großer Daten nach stdout

Das folgende Beispiel zeigt, wie wir eine Datei einlesen und anschließend auf der Standardausgabe ausgeben. Die Größe der eingelesenen Datei beträgt rund zwei Megabyte und wurde zuvor mit einem kleinen Tool angelegt.

```

1 #include <termios.h>
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     int bytes_written, bytes_to_write, fd, max_input;
6     int writes = 0, errors = 0;
7     char *temp, large_buffer[20000];
8     char *basename = basename_ex(argv[0]);
9
10    if (argc < 2)
11        err_fatal("Usage: %s <file>\n", basename);
12
13    if ((fd = open(argv[1], O_RDONLY)) < 0)
14        err_fatal("%s: open() failed for %s.", basename, argv[1]);
15
16    if ((max_input = pathconf("/", _PC_MAX_INPUT)) < 0)
17        err_fatal("pathconf failed for MAX_INPUT");
18
19    bytes_to_write = read(fd, large_buffer, sizeof(large_buffer));
20    fprintf(stderr, "Bytes read: %d\n", bytes_to_write);
21
22    for (temp = large_buffer; bytes_to_write > 0;) {
23        errno = 0;
24        bytes_written = write(STDOUT_FILENO, temp, bytes_to_write);
25        writes++;
26
27        if (errno != 0)
28            errors++;
29
30        fprintf(stderr, "\n%s: bytes written: %d, errno = %d\n",
31                basename, bytes_written, errno);
32
33        if (bytes_written > 0) {
34            temp += bytes_written;
35            bytes_to_write -= bytes_written;
36        }
37    }
38
39    printf("Summary: total writes: %d, \
40           total errors: %d, \
41           MAX_INPUT:      %d\n", writes, errors, max_input);
42
43    exit(0);
44 }
```

Listing 5.11: xcode/io-write.c - Schreiben großer Daten nach stdout.

Das funktioniert mit der Standardausgabe sehr gut, obwohl wir sie auf O_NONBLOCK gesetzt haben. Das ist auch nicht verwunderlich, denn wir fordern nur so viel an, wie wir in unserem Buffer speichern können. Also wird nur ein einziger Aufruf von `write(2)` benötigt. Schreiben wir aber in ein anderes Terminal, daß nur eine begrenzte Buffergröße hat, werden mehrere aufeinander folgende Schreibvorgänge benötigt.

5.8 Zugriffe synchronisieren

Wollen mehrere BenutzerInnen auf die gleiche Resource, z.B. eine Datei, schreibend zugreifen, so gelingt dies in der Regel nur denen, die schneller sind als andere. Solange die Datei von BenutzerInnen verwendet wird, können andere diese Datei nicht benutzen. Dieser Vorgang wird als *File Locking* bezeichnet. Im Gegensatz dazu können auch nur bestimmte Bereiche innerhalb der Datei vor dem Zugriff anderer Prozesse geschützt werden. Diese Bereiche nennt man *Records*, so daß im Allgemeinen von *Record Locking*

gesprochen wird. Abbildung 5.1 faßt die Funktionsweise des Record Locking zusammen. Bezogen auf das Betriebssystem bedeutet Record Locking eigentlich nur, daß ein Prozess, der eine bestimmte Resource für sich beansprucht, andere Prozesse von der Verwendung derselben ausschließt.

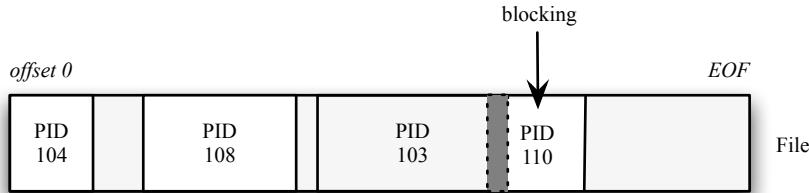


Abbildung 5.1: Mehrere Prozesse sperren jeweils einen eigenen Bereich der Datei.

Die unterschiedlichen UNIX-Derivate verfolgen verschiedene Ansätze beim Locking. So basiert das Locking der SYSV-Implementierungen auf `lockf(2)`, während BSD-basierte Systeme `fcntl(2)` oder `flock(2)` verwenden (`flock(2)` unterstützt kein Record Locking). POSIX geht erstgenannten Weg und wird Hauptbestandteil unserer Diskussionen sein.

`flock(2)` stammt aus der BSD-Familie und ist quasi auf allen UNIX-Plattformen verfügbar. Es ist sehr einfach zu implementieren und sehr effektiv auf Einzelplätzen aber fast nutzlos in Netzwerkumgebungen, beispielsweise NFS, denn `flock(2)` sperrt die gesamte Datei. `lockf(2)` hingegen ist nur ein vereinfachtes Interface für `fcntl(2)`.

5.8.1 Locking mit einer Lock-Datei

Lange Zeit, und heute noch in einigen Programmen zu finden, war das Locking über eine Lock-Datei durchaus üblich. Allerdings hat es sich in Mehrbenutzerumgebungen schnell als nachteilig erwiesen. Dennoch lohnt sich ein Blick auf diese Technik, um zu verstehen, welche Probleme mit neueren Lösungen behoben wurden.

Listing 5.12 zeigt ein vollständiges Programm, das 500 Datensätze schreibt und den Zugriff über eine Lock-Datei synchronisiert. Das Programm ist besonders im Hinblick auf den Schreibvorgang sehr ineffektiv und dient nur der Demonstration des Lock-Mechanismus. In der Besprechung des Beispiels gehen wir genauer auf die Funktionsweise ein.

Listing 5.12: Locking mit Hilfe einer Lockdatei

```

1 #include "header.h"
2
3 char *basename, *lock_file;
4
5 void create_lock(void) {
6     int fd;
7
8     do {
9         if ((fd = open(lock_file, O_WRONLY | O_CREAT | O_EXCL, 0666)) < 0)
10             if (errno == EEXIST)
11                 sleep(1);
12             else
13                 err_fatal("Could not acquire lock");
14     } while (fd < 0);
15
16     close(fd);
17 }
18

```

```

19 void release_lock(void) {
20     unlink(lock_file);
21 }
22
23 int main(int argc, char **argv) {
24     FILE *file;
25     int i, c, want_locking = 1;
26
27     if ((basename = strrchr(argv[0], '/')) == NULL)
28         basename = argv[0];
29     else
30         basename++;
31
32     /* dynamically create a suitable lock file */
33     if ((lock_file = (char *)malloc((size_t)strlen(basename) + 4)) == NULL) {
34         err_fatal("malloc failed");
35     } else {
36         strcpy(lock_file, basename);
37         strcat(lock_file, ".lck");
38     }
39
40     if (argc >= 2 && !strcmp(argv[1], "NOLOCK"))
41         want_locking = 0;
42
43     printf("Locking is%s enabled.\n", (want_locking ? "" : " not"));
44
45     /* fill in some arbitrary data --- EXTREMELY SLOOOOW! DEMO ONLY! */
46     for (i = 0; i < 500; i++) {
47         if (want_locking)
48             create_lock();
49
50         if ((file = fopen("file.dat", "r+")) == NULL) /* file exists? */
51             if (file == NULL && errno == ENOENT) /* no, create it */
52                 if ((file = fopen("file.dat", "w+")) == NULL) {
53                     if (want_locking)
54                         release_lock();
55                     err_fatal("fopen() failed");
56                 }
57
58         if (fseek(file, 0, SEEK_END) < 0)
59             err_fatal("fseek() failed");
60
61         fprintf(file, "Process [%ld]: Record #: %i\n", (long)getpid(), i);
62         for (c = ' ' ; c <= 'z' ; c++)
63             fputc(c, file);
64
65         fputc('\n', file);
66         fclose(file);
67
68         if (want_locking)
69             release_lock();
70     }
71
72     printf("Process [%ld] finished writing.\n", (long)getpid());
73     exit(0);
74 }
```

Listing 5.12: xcode/file-locking.c - Locking mit Hilfe einer Lockdatei.

Kern des Programms ist die `for`-Schleife der `main`-Funktion. Wir durchlaufen eine Schleife 500 mal, öffnen eine `dat`-Datei, schreiben einen Datensatz, schließen die Datei anschließend und geben die Lock-

Datei frei. Die Funktion `create_lock` prüft, ob die Lock-Datei vorhanden ist und wenn ja, ist gerade ein Prozess dabei, in die Datei zu schreiben, so daß wir ihn für eine Sekunde schlafen schicken.

Ein Testdurchlauf könnte etwa so aussehen:

```
% ./file-locking
Locking is enabled
Process [10501] finished writing.
%wc -l ./lockfile.dat
500
```

Standardmäßig ist Locking aktiv und nach einiger Zeit hat das Programm alle Datensätze geschrieben. Ein kurzer Blick genügt und wir wissen, daß alle Datensätze geschrieben wurden. Das war auch zu erwarten und ist natürlich kein Verdienst des Lockings, denn es war ja nur ein Prozess beteiligt.

Ein zweiter Durchlauf mit zwei Prozessen liefert das gleiche Ergebnis:

```
% ./file-locking& ./file-locking&
Locking is enabled
Locking is enabled
Process [10502] finished writing.
Process [10503] finished writing.
%wc -l ./lockfile.dat
1000
```

Prozess 10502 fordert die Lock-Datei erfolgreich an und darf als erstes schreiben, Prozess 10503 geht erstmal schlafen und versucht es nach einer Sekunde noch einmal. Auf diese Weise schreiben beide nach und nach ihre Datensätze in die Datendatei. Die Zugriffe können insgesamt ziemlich stark streuen.

Der letzte Durchlauf mit drei Prozessen ohne Locking illustriert die Notwendigkeit des Lockings:

```
% ./file-locking NOLOCK & ./file-locking NOLOCK & ./file-locking NOLOCK &
Locking is enabled
Locking is enabled
Locking is enabled
Process [10504] finished writing.
Process [10505] finished writing.
Process [10506] finished writing.
%wc -l ./lockfile.dat
1498
```

Wie wir sehen können sind ohne das Locking zwei Datensätze verloren gegangen. Es hätten auch mehr sein können. Möglicherweise wäre auch gar keiner verloren gegangen. Wir können es nicht vorhersagen, aber es besteht eine Wahrscheinlichkeit, daß es vorkommt. □

Der Mechanismus weist einige Nachteile auf:

- Durch den Aufruf von `sleep`, um herauszufinden, ob ein Prozess in die Datei schreiben kann, werden andere Prozesse unverältelmäßig lange vom Schreiben ihrer Daten abgehalten. Je mehr Prozesse Zugriff auf die Datei begehren, desto länger dauert es, bis sie schreiben dürfen.
- Der Lock-File-Mechanismus, der auf `O_CREAT | O_EXCL` beruht, könnte im Netzwerk nicht funktionieren wie zu erwarten wäre, denn das Zielbetriebssystem legt den Aufruf möglicherweise nicht atomar aus.
- Auch wenn immer nur eine Zeile geschrieben werden soll, wird die gesamte Datei gesperrt.

Wir sehen, daß es viel Raum für Verbesserungen gibt, die alle durch den Einsatz einer der Techniken der kommenden Abschnitte gelöst werden.

5.8.2 Record Locking mit fcntl(2)

Die `fcntl(2)`-Funktion haben wir bereits in zahlreichen Beispielen gesehen und wissen, daß wir damit auf vielfältige Weise Einfluß auf bestimmte Eigenschaften unterschiedlichster Datenstrukturen und Abläufe nehmen können, dazu gehört auch das Locking.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock /* flockptr */ *);
```

Rückgabewerte: abhängig von cmd, oder -1 bei Fehler.

Die Parameter kennen wir schon. Ein neuer ist aber doch dabei: `flockptr`. Die Struktur enthält alle Informationen über die Art der Sperrung, in welchem Bereich die Datei gesperrt ist, usw.

```
struct flock {
    short l_type;    /* type of lock; F_RDLCK, F_WRLCK, F_UNLCK */
    off_t l_start;   /* relative offset in bytes */
    short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t l_len;     /* size; if 0 then until EOF */
    pid_t l_pid;     /* process ID of the process holding the lock */
}
```

Der Typ der Sperrung ist in `l_type` gespeichert, `l_start` legt fest, wo die Sperrung innerhalb der Datei beginnt (gemessen vom Anfang der Datei), `l_whence` ist eine der Konstanten `SEEK_SET`, `SEEK_CUR`, oder `SEEK_END`, die anzeigen, daß die Sperrung am Anfang, ab der aktuellen Position in der Datei oder ab dem Ende beginnen soll. Mit dem Member `l_len` legen wir fest, wie viele aufeinanderfolgende Bytes gesperrt werden sollen und `l_pid` ist besetzt, wenn wir ein Lock einer Datei über das Kommando `F_GETLK` anfordern.

Wenn `l_len` ein positiver Wert ist, beträgt der gesperrte Bereich genau `l_len` Bytes beginnend bei `l_start` und endend bei `l_start + l_len - 1`. Locks dürfen über das Dateiende hinaus, aber nicht vor den Beginn einer Datei gesetzt werden. Ein Wert von 0 für `l_len` veranlaßt das System, eine Sperrung bis zum Dateiende zu setzen und automatisch zu verlängern, wenn der Datei Daten hinzugefügt werden. Wird hingegen der Wert 0 für `l_start` und `SEEK_SET` für `l_whence` angegeben, so unterliegt die gesamte Datei einer Sperrung.

Es existieren zwei Arten der Sperrung: `F_WRLCK` (*write lock*) und `F_RDLCK` (*read lock*). Mehrere Prozesse dürfen das gleiche Byte oder den gleichen Bereich sperren. Wobei aber immer nur ein Prozess eine exklusive Sperrung beanspruchen darf. Schreibsperrungen und Lesesperrungen schließen sich gegenseitig aus. Wenn also ein Prozess für einen bestimmten Bereich eine Sperrung erhalten hat, kann kein anderer Prozess diesen zum Schreiben sperren.

Die Sperrungen verwalten wir über drei Kommandos, die wir der `fcntl(2)`-Funktion übergeben:

F_GETLK

Mit diesem Kommando fragen wir ab, ob die Sperrung auf die `flockptr` zeigt, von einem anderen Prozess gesetzt ist. Wenn das der Fall ist, wird die Information in der `flock`-Struktur auf die `flockptr` zeigt mit der des existierenden Locks überschrieben. Gibt es keine Sperrung, so wird nur das Member `l_type` auf `F_UNLCK` gesetzt.

F_SETLK

Wenn wir eine Sperrung setzen wollen, verwenden wir dieses Kommando und übergeben dabei die in der `flock`-Struktur enthaltenen Informationen, auf die `flockptr` zeigt, an die `fcntl(2)`. Das gleiche Kommando verwenden wir auch, um bestehende Locks zu entfernen. Dazu setzen wir das Feld `l_type` einfach auf `F_UNLCK`.

F_SETLKW

Dies ist die blockierende Variante von F_SETLK. Kann momentan keine Sperrung des angegebenen Bereichs angefordert werden, wird der Prozess schlafen geschickt und durch jedes beliebige Signal aufgeweckt.

Zwei Dinge sollten unbedingt berücksichtigt werden. Wenn wir eine Sperrung erfragen (nachschauen, ob wir den angeforderten Bereich sperren können) und sie direkt im Anschluß setzen, sind daß zwei getrennte Operationen, zwischen denen ein Zeitfenster liegt. Es kann also passieren, daß ein anderer Prozess zwischen diesen beiden Operationen selbst eine Sperrung setzt. Sofern wir nicht F_SETLKW verwenden und so lange warten, bis wir eine Sperrung erhalten, müssen wir die Fehlercodes von F_SETLK einzeln abfragen und entsprechend reagieren. Ein anderer Punkt betrifft die Aufhebung von Sperrungen. Es ist ohne weiteres möglich, bestimmte gesperrte Bereich aufzuteilen. Wenn wir von Byte 10 bis Byte 20 eine Sperrung gesetzt haben und uns entschließen, Byte 15 freizugeben, so sind die Bereiche von Byte 10 bis 14 und Byte 16 bis 20 noch immer durch unseren Prozess gesperrt.

Listing 5.13: Einfache Anwendung des Record Locking

Das folgende kleine Programm erzeugt ein Lock zum Lesen für das erste Byte in der Datei. Anschließend wird ein Kindsprozess erzeugt, der entweder ein Lock zum Lesen oder Schreiben anfordern kann. Unterhalb des Listings sehen Sie das Ergebnis.

```

1 #include "header.h"
2
3 int get_lock_mode(char *c);
4 int lckset(int fd, int command, int type,
5            off_t start, int whence, off_t len);
6 int islockable(int fd, int type, off_t start,
7                 int whence, off_t len);
8
9 int main(int argc, char *argv[]) {
10     char *basename = basename_ex(argv[0]);
11     int fd, status, child_lock_mode;
12     pid_t pid, lock_pid;
13
14     if ((child_lock_mode = get_lock_mode(argv[2])) == -1)
15         err_fatal("Lock mode for child must be of type READ or WRITE.");
16     pthread_rwlock2
17     if (argc != 3)
18         err_fatal("Usage: %s <file> <child lock mode>", basename);
19
20     /* open a file for reading and writing */
21     if ((fd = open(argv[1], O_RDWR, S_IWUSR | S_IRUSR)) < 0)
22         err_fatal("open failed for %s", argv[1]);
23
24     /* acquire a lock */
25     if (lckset(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 1) < 0)
26         err_fatal("lckset failed");
27
28     printf("Parent (PID %d) acquired a lock.\n", getpid());
29
30     /* create a child and acquire a lock for it */
31     if ((pid = fork()) < 0) {
32         err_fatal("fork failed");
33     } else if (pid == 0) { /* child code */
34         /* let's see if we can acquire a lock */
35         if ((lock_pid = islockable(fd, child_lock_mode, 0, SEEK_SET, 1)) == 0) {
36             /* OK! Now get a lock already! */
37             if (lckset(fd, F_SETLK, child_lock_mode, 0, SEEK_SET, 1) < 0) {
38                 err_fatal("lckset in child failed");
39             }

```

```

40         printf("Child (PID %d) acquired lock.\n", getpid());
41     } else {
42         printf("File is locked by PID %d.\n", lock_pid);
43     }
44 } else { /* parent code */
45     while (1) /* waiting for child to return */
46         if (waitpid(pid, &status, WNOHANG) > 0)
47             exit(0);
48 }
49
50     return (0);
51 }
52
53 int get_lock_mode(char *c) {
54     if (strcmp(c, "READ") == 0)
55         return (F_RDLCK);
56     else if (strcmp(c, "WRITE") == 0)
57         return (F_WRLCK);
58     else
59         return -1;
60 }
61
62 int lckset(int fd, int command, int type,
63             off_t start, int whence, off_t len) {
64     struct flock lock;
65
66     lock.l_type = type;
67     lock.l_start = start;
68     lock.l_whence = whence;
69     lock.l_len = len;
70
71     return (fcntl(fd, command, &lock));
72 }
73
74 int islockable(int fd, int type, off_t start, int whence, off_t len) {
75     struct flock lock;
76
77     if (type == F_RDLCK) lock.l_type = F_RDLCK;
78     else if (type == F_WRLCK) lock.l_type = F_WRLCK;
79
80     lock.l_start = start;
81     lock.l_whence = whence;
82     lock.l_len = len;
83
84     if (fcntl(fd, F_GETLK, &lock) < 0) {
85         err_fatal("fcntl failed in islockable()");
86
87         if (lock.l_type == F_UNLCK)
88             return (0);
89         else
90             return(lock.l_pid);
91     }

```

Listing 5.13: xcode/locksmp.c - Einfache Anwendung des Record Locking.

Die Ausgabe unseres Programms könnte etwa so aussehen:

```
% ./locksmp testfile READ
Parent (PID 22107) acquired a lock.
Child (PID 22108) acquired lock.
% ./locksmp testfile WRITE
Parent (PID 22109) acquired a lock.
```

```
File is locked by PID 22109.
```

Das zweite Argument ist eine Testdatei, die wir für das Locking einsetzen wollen. Dazu können Sie einfach eine kleine Textdatei verwenden, die ein paar Zeichen enthält, da wir das Record Locking nur an den ersten beiden Bytes demonstrieren wollen. Der dritte Parameter gibt an, ob der Kindsprozess eine Lesesperre oder eine Schreibsperre anfordern soll. Im ersten Durchgang können sowohl Parent als auch Child eine Sperre anfordern, da beide nur Lesen möchten. Im zweiten Versuch, bei dem wir dem Child eine Schreibsperre verordnen, klappt das nicht mehr, denn wenn eine Lesesperre gesetzt ist, kann für den gleichen Bereich keine Schreibsperre angefordert werden. Das gilt natürlich auch umgekehrt. □

Das vorangegangene Beispiel macht eine wichtige Regel deutlich. Nur wenn ein Prozess keine Schreibsperre gesetzt hat, können mehrere Prozesse den gesperrten Bereich auch zum Lesen sperren. Sobald eine Schreibsperre im Spiel ist, kann kein Prozess auf diesen Bereich zugreifen, auch nicht lesend.

Ohne weiter auf diese Tatsache einzugehen, haben wir sehen können, daß Sperren nicht an den Child-Prozess vererbt werden. Beispiel 5.13 hat es gezeigt: der Child-Prozess kann selbst eine Sperre setzen, obwohl sein Parent den gleichen Record zum Lesen sperrt. Ein Child-Prozess wird als völlig eigenständiger Prozess behandelt und muß über die geerbten File Descriptors eigene Locks anfordern. Hintergrund dieses Mechanismus ist die Tatsache, daß Sperren vor gleichzeitigen Zugriffen unterschiedlicher Prozesse schützen sollen, denn sonst könnten Parent und Child gleichzeitig schreibend auf eine Datei zugreifen.

Alle Sperren eines Prozesses werden freigegeben, sobald der Prozess beendet wird. Das gleiche trifft auch für File Descriptors zu. Wird er geschlossen, sind die Sperren aufgehoben. Aus diesem Grund würden Locks, die durch das Duplizieren (z.B. `dup(2)` oder `fcntl(2)`) eines File Descriptors übertragen wurden, ebenfalls freigegeben.

5.8.3 Kooperatives und vorgeschriebenes Locking

Der POSIX-Standard unterscheidet zwischen zwei unterschiedlichen Arten des Locking: kooperatives Locking (*cooperative locking*) und vorgeschriebenes Locking (*mandatory locking*).

Kooperatives Locking ist in Umgebungen sinnvoll, in denen bestimmte Ressourcen nur von einer Gruppe von Prozessen verwendet wird und ausgeschlossen ist, daß andere Prozesse auf diese Ressourcen zugreifen. Diese Prozesse werden als kooperative Prozesse bezeichnet. Es gibt nur wenige Situationen, in denen kooperatives Locking ausreichend ist, denn selbst wenn wir nur eine einfache Textdatei mit einem Editor verändern möchten, die gerade von einem anderen Prozess verwendet wird, reicht kooperatives Locking nicht aus.

Im Gegensatz dazu steht das vorgeschriebene Locking. Es wird durch den Kernel implementiert, der darüber wacht, daß kein Prozess beim Lesen oder Schreiben einer Resource die aktiven Sperren anderer Prozesse verletzt. Der POSIX-Standard unterstützt nur das vorgeschriebene Locking ganzer Dateien und nicht einzelner Records, wie es beispielsweise SysV-Implementierungen tun.

Aktiviert wird das vorgeschriebene Locking durch das Setzen des SGID-Bits und das Ausschalten des Ausführungsbits für die Gruppe. Wie Prozesse auf Sperren anderer Prozesse reagieren hängt von der angeforderten Operation ab. Nur wenn ein Prozess keine Schreibsperre gesetzt hat, können andere Prozesse lesend auf diese Datei zugreifen. Das gilt für `read(2)`, `write(2)` oder `open(2)`, die blockieren. Im Idealfall ist die Resource irgendwann verfügbar und der Prozess erhält Zugriff. Ist die Option `O_NONBLOCK` gesetzt, warten anfragende Prozesse nicht und kehren sofort zurück, wenn ausstehende Locks den Zugriff verweigern. Dabei wird `errno` auf `EAGAIN` gesetzt. Eine Ausnahme bildet der Systemaufruf `open(2)`. Obwohl noch einige Locks ausstehen könnten ist der Aufruf erfolgreich, sofern nicht `O_CREAT` oder `O_TRUNC` gesetzt sind, da sie für einen schreibenden Zugriff auf die Datei sorgen. Zugegebenermaßen macht die Anzeige eines Fehlers für `O_CREAT` keinen Sinn, da die Datei erst erzeugt werden muß/soll und somit keine Sperre vorhanden sein kann.

Da sich die POSIX-Empfehlung nicht eindeutig für das vorgeschriebene Locking ausspricht und es auch nicht auf allen Systemen verfügbar ist, bleibt uns nichts anderes übrig, als herauszufinden, ob das vorgeschriebene Locking unterstützt wird. Listing 5.14 zeigt eine Funktion, die den Wert 1 zurückgibt, wenn es unterstützt wird oder 0 wenn nicht. Bei Fehler wird -1 zurückgegeben.

Listing 5.14: Herauszufinden, ob vorgeschriebenes Locking unterstützt wird

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <fcntl.h>
5 #include <signal.h>
6 #include <wait.h>
7 #include <unistd.h>
8 #include "header.h"
9
10 #define MODE S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH /* 644 */
11
12 int lock_detect(void);
13 int lckset(int fd, int command, int type,
14            off_t start, int whence, off_t len);
15 void signal_handler(int signum);
16 void cleanup(char *path);
17
18 static volatile sig_atomic_t signal_flag;
19 static sigset_t current_mask, backup_mask, sigall_mask;
20
21 int main(void) {
22     int i;
23
24     if ((i = lock_detect()) < 0)
25         err_fatal("Main: lock_detect failed");
26     if (i == 0)
27         printf("Main: No enforcement-mode locking available.\n");
28     if (i > 0)
29         printf("Main: Enforcement-mode locking enabled.");
30
31     return (0);
32 }
33
34 int lock_detect(void) {
35     int fd, flags, status;
36     pid_t pid;
37     char readbuf[5], *writeme = "lock", *testfile = ".lock";
38     struct stat bstat;
39
40     if ((fd = open(testfile, O_RDWR | O_TRUNC | O_CREAT, MODE)) < 0)
41         err_fatal("open failed");
42
43     if (write(fd, &writeme, 4) < 0)
44         err_fatal("write failed");
45
46     if (fstat(fd, &bstat) < 0)
47         err_fatal("fstat failed");
48
49     if (fchmod(fd, (bstat.st_mode & ~S_IXGRP) | S_ISGID) < 0)
50         err_fatal("chmod failed");
51
52     if (signal(SIGUSR1, signal_handler) == SIG_ERR)
53         err_fatal("signal failed for SIGUSR1");
54
55     if (signal(SIGUSR2, signal_handler) == SIG_ERR)
56         err_fatal("signal failed for SIGUSR1");
57
58     sigemptyset(&sigall_mask);
59     sigemptyset(&current_mask);
60

```

```

61     sigaddset(&current_mask, SIGUSR1);
62     sigaddset(&current_mask, SIGUSR2);
63
64     if (sigprocmask(SIG_BLOCK, &current_mask, &backup_mask) < 0)
65         err_fatal("sigprocmask failed");
66
67     if ((pid = fork()) < 0) {
68         err_fatal("fork failed");
69     } else if (pid > 0) { /* parent code */
70         /* create a write lock for file */
71         if (lckset(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
72             err_fatal("Parent: lckset failed.");
73
74         kill(pid, SIGUSR1);
75
76         if (waitpid(pid, NULL, 0) < 0)
77             err_fatal("Parent: waitpid failed");
78     } else if (pid == 0) { /* child code */
79         while (signal_flag == 0)
80             sigsuspend(&sigall_mask);
81
82         signal_flag = 0;
83
84         if (sigprocmask(SIG_SETMASK, &backup_mask, NULL) < 0)
85             err_fatal("Child: sigprocmask failed");
86
87         /* adjust flags for fd to non-blocking */
88         if ((flags = fcntl(fd, F_GETFL, 0)) < 0)
89             err_fatal("Child: fcntl failed for F_GETFL");
90
91         flags |= O_NONBLOCK;
92
93         if (fcntl(fd, F_SETFL, 0) < 0)
94             err_fatal("Child: fcntl failed for F_SETFL");
95
96         if (lckset(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) != -1)
97             printf("Child: Read lock OK\n");
98         else
99             printf("Child: Already locked, Error: %d\n", errno);
100
101        if (lseek(fd, 0, SEEK_SET) == -1)
102            err_fatal("seek failed");
103
104        if (read(fd, readbuf, 2) < 0)
105            return (1); /* enforcement-mode locking enabled */
106        else
107            return (0); /* no enforcement-mode locking */
108    }
109
110    cleanup(testfile);
111 }
112
113 int lckset(int fd, int command, int type,
114 off_t start, int whence, off_t len) {
115     struct flock lock;
116
117     lock.l_type = type;
118     lock.l_start = start;
119     lock.l_whence = whence;
120     lock.l_len = len;
121
122     return (fcntl(fd, command, &lock));

```

```
123 }
124
125 void cleanup(char *path) {
126     if (remove(path) < 0)
127         err_fatal("remove failed");
128 }
129
130 void signal_handler(int signum) {
131     signal_flag = 1;
132     return;
133 }
```

Listing 5.14: xcode/lockdetect.c - Herausfinden, ob vorgeschriebenes Locking unterstützt wird.

Kapitel 6

Systeminformationen und Systemdateien

Seize the moment of excited curiosity on any subject to solve your doubts; for if you let it pass, the desire may never return, and you may remain in ignorance.

WILLIAM WIRT (1772 - 1834)

Viele Dateien eines UNIX-Systems sind für den reibungslosen Betrieb unabdingbar. Sie bestimmen, ob wir mit anderen Systemen kommunizieren dürfen, welche Dienste in welcher Reihenfolge gestartet werden usw. In diesem Kapitel befassen wir uns mit dem Umgang mit diesen Dateien.

6.1 Benutzer- und Gruppeninformationen

Traditionell werden sehr viele Informationen des Systems, seiner Einstellungen und natürlich auch seiner Benutzer in Textdateien gespeichert. Manche sind einfach, andere wiederum recht komplex aufgebaut. Zu diesen Dateien gehören auch die Gruppen- und Passworddateien. In diesem Abschnitt lernen wir Funktionen kennen, die uns bei der Abfrage bestimmter Einträge aus den beiden Dateien behilflich sind.

getpwuid(3) und getpwnam(3)

dienen der Abfrage von Informationen über Benutzer auf Basis der Benutzer-ID oder des Benutzernamens.

getpwent(3), setpwent(3) und endpwent(3)

durchsuchen die Passworddatei sequenziell, setzen den Zeiger auf den Anfang der Datei zurück und schließen den Zugriff auf die Datei.

getgrgid(3) und getrgnam(3)

rufen Gruppeninformationen auf Basis der Gruppen-ID und des Gruppennamens ab.

getgrent(3), setgrent(3) und endgrent(3)

durchsuchen die Gruppendatei sequenziell, setzen den Zeiger auf den ersten Eintrag zurück und beenden den Zugriff auf die Gruppendatei.

initgroups(3), getgroups(3) und setgroups(3)

initialisieren die Gruppendatei, fragen zusätzliche Gruppen-IDs ab und setzen sie.

6.1.1 Die Benutzerdatenbank /etc/passwd

Die UNIX-Passwortdatei (unter POSIX.1 als **Benutzerdatenbank** bezeichnet) enthält einige Felder, die zur Identifizierung eines Benutzers benötigt werden. Sie sind alle in der Struktur `passwd` enthalten, die durch den Header `<pwd.h>` spezifiziert wird.

Beschreibung	Strukturelement	POSIX.1
Benutzername	<code>char *pw_name</code>	•
verschlüsseltes Passwort	<code>char *pw_passwd</code>	
numerische Benutzer-ID	<code>uid_t pw_uid</code>	•
numerische Gruppen-ID	<code>gid_t pw_gid</code>	•
Kommentar	<code>char *pw_gecos</code>	
Heimatverzeichnis	<code>char *pw_dir</code>	•
Benutzerspezifische Shell	<code>char *pw_shell</code>	•

Tabelle 6.1: Auflistung der durch POSIX vorgeschriebenen Felder in `/etc/passwd`.

Wie wir obiger Tabelle entnehmen können, verlangt POSIX.1 nur fünf der unter fast allen Unices vorhandenen Felder, sieht aber nur zwei Funktionen zur Abfrage von Datensätzen vor: eine für Benutzer-IDs und eine andere für Benutzernamen:

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);

Rückgabewerte: Beide geben einen Zeiger zurück oder NULL bei Fehler.
```

uid

Benutzer-ID, dessen Informationen abgefragt werden sollen.

name

Benutzername dessen Informationen abgefragt werden sollen.

Sinn und Zweck ist die schnelle Zuordnung von Benutzernamen bzw. IDs zu Objekten im Dateisystem. Denn jedesmal wenn auf solche Objekte zugegriffen wird, beispielsweise von `ls(1)`, muß auch `getpwuid(3)` aufgerufen werden, um die IDs oder Namen der Besitzer anzeigen zu können. Ebenso wird `getpwnam(3)` von `login(1)` verwendet, um den eingegebenen Benutzernamen mit dem in `/etc/passwd` zu vergleichen. Listing 6.1 zeigt wie es geht.

Die `passwd`-Struktur enthält die in Tabelle 6.1 genannten Felder:

```
struct passwd {
    char    *pw_name;          /* user name */
    char    *pw_passwd;         /* user password */
    uid_t   pw_uid;            /* user id */
    gid_t   pw_gid;            /* group id */
    char    *pw_gecos;          /* real name */
    char    *pw_dir;             /* home directory */
    char    *pw_shell;           /* shell program */
};
```

Sie wird auch von `getpwent` und Co. verwendet.

Listing 6.1: Benutzerinformationen mit `getpwnam(3)` und `getpwuid(3)` abfragen

Wir übergeben dem Programm entweder eine Benutzer ID oder einen Benutzernamen. Anschließend zeigen wir die Informationen aus der Passwortdatenbank an.

```

1 #include <pwd.h>
2 #include "header.h"
3
4 void print_info(struct passwd *p);
5
6 int main (int argc, char **argv) {
7     char          *basename = basename_ex(argv[0]);
8     struct passwd *pd;
9
10    if (argc < 2) {
11        err_fatal("Usage: %s [<username> | <uid>]\n", basename);
12    }
13
14    if (((pd = getpwnam(argv[1])) == NULL) &&
15        ((pd = getpwuid(atoi(argv[1]))) == NULL))
16        err_fatal("Could not get user info");
17    else
18        print_info(pd);
19
20    return (0);
21 }
22
23 void print_info(struct passwd *p) {
24     printf("      User name: %s\n", p->pw_name);
25     printf("      User id: %u\n", p->pw_uid);
26     printf("      Group ID: %u\n", p->pw_gid);
27     printf("      Initial directory: %s\n", p->pw_dir);
28     printf("Initial user program: %s\n", p->pw_shell);
29 }
```

Listing 6.1: xcode/getpwnam.c - Benutzerinformationen mit getpwnam(3) und getpwuid(3) abfragen.

Wenn weder `getpwuid` und `getpwnam` ein Ergebnis liefern ist die ID oder der jeweilige Name nicht vorhanden, was mit einem ENOENT quittiert wird:

```
% ./getpwnam graegerts
      User name: graegerts
      User id: 1000
      Group ID: 100
      Initial directory: /home/graegerts
      Initial user program: /bin/csh
% ./getpwnam 999
Could not get user info: No such file or directory
```

□

SysV und quasi alle Unices sowie deren Derivate definieren drei zusätzliche Funktionen zur Untersuchung der gesamten Passworddatei:

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent(void);

Rückgabewerte: Zeiger auf Struktur oder NULL bei Fehler oder Dateiende.

void setpwent(void);
void endpwent(void);
```

`getpwent(3)` rufen wir auf, um den nächsten Eintrag in `passwd` zu erhalten. Mit `setpwent(3)` setzen wir die den Cursor auf den Anfang zurück. Schließlich beenden wir die Arbeit mit `passwd` durch einen Aufruf von `endpwent(3)`.

Listing 6.2: Benutzerinformationen mit getpwent(3) auflisten

Das folgende kleine Programm ruft alle Benutzerinformationen des lokalen Systems aus der Passworddatenbank ab.

```

1 #include <pwd.h>
2 #include "header.h"
3
4 void print_info(struct passwd *p);
5
6 int main (int argc, char **argv) {
7     char          *basename = basename_ex(argv[0]);
8     struct passwd *pd;
9
10    if ((pd = getpwent()) == NULL)
11        err_fatal("getpwent() failed");
12
13    while (pd = getpwent())
14        print_info(pd);
15
16    endpwent();
17    return (0);
18 }
19
20 void print_info(struct passwd *p) {
21     printf("      User name: %s\n", p->pw_name);
22     printf("      User id: %u\n", p->pw_uid);
23     printf("      Group ID: %u\n", p->pw_gid);
24     printf("      Initial directory: %s\n", p->pw_dir);
25     printf("Initial user program: %s\n", p->pw_shell);
26 }
```

Listing 6.2: xcode/getpwent.c - Benutzerinformationen mit getpwent(3) auflisten.



6.1.2 Die Gruppendatei /etc/group

Die UNIX-Gruppenverwaltung wird mit Hilfe der Datei /etc/group verwaltet. Sie enthält folgende Felder, von denen allerdings nicht alle für POSIX.1 erforderlich sind:

Beschreibung	Strukturelement	POSIX.1
Gruppenname	char *gr_name	•
Verschlüsseltes Passwort	char gr_passwd	
Nummerische Gruppen-ID	int gr_gid	•
Array von Zeigern auf Benutzer-IDs	char **gr_mem	•

Tabelle 6.2: Durch POSIX vorgeschriebene Felder der Gruppendaten /etc/group.

Der letzte Eintrag ist ein Feld von Zeigern, das auf individuelle Benutzernamen zeigt, die Mitglied dieser Gruppe sind. POSIX.1 stellt auch für die Arbeit mit Gruppendateien nur zwei Funktionen zur Verfügung.

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

Rückgabewerte: Beide geben einen Zeiger vom Typ group oder NULL bei Fehler zurück.

gid

Gruppen-ID, über die Informationen abgefragt werden sollen.

name

Gruppenname, über die Informationen abgefragt werden sollen.

Die **group**-Struktur ist folgendermaßen definiert:

```
struct group {
    char      *gr_name;          /* group name */
    char      *gr_passwd;        /* group password (not defined in POSIX.1) */
    gid_t    gr_gid;            /* group id */
    char      **gr_mem;          /* group members */
};
```

POSIX.1 spezifiziert keine Gruppenpasswörter, so daß **gr_passwd** nicht definiert ist.

Listing 6.3: Gruppeninformationen mit **getgrnam(3)** auflisten

Das folgende kleine Beispiel ruft die Gruppeninformationen einer Gruppe, basierend auf einem Gruppennamen oder einer Gruppen-ID ab.

```
1 #include <grp.h>
2 #include "header.h"
3
4 int main (int argc, char **argv) {
5     char           *basename = basename_ex(argv[0]);
6     struct group *grp;
7
8     if (argc < 2) {
9         err_fatal("Usage: %s [<groupname> | <gid>]\n", basename);
10    } else {
11        if ( ((grp = getgrnam(argv[1])) == NULL) &&
12            ((grp = getgrgid(atoi(argv[1]))) == NULL) )
13            err_fatal("Could not get group info");
14        else {
15            printf("      Group name: %s\n", grp->gr_name);
16            printf("              GID: %u\n", grp->gr_gid);
17
18            int i;
19            for (i = 1; *(grp->gr_mem); i++, (grp->gr_mem)++)
20                printf("Group member %d: %s\n", i, *(grp->gr_mem));
21        }
22    }
23
24    return (0);
25 }
```

*Listing 6.3: xcode/getgrnam.c - Gruppeninformationen mit **getgrnam(3)** auflisten.*

Unabhängig davon, ob wir eine Gruppen-ID oder einen Gruppennamen angeben, erhalten wir immer die gleiche Ausgabe:

```
% ./getgrnam 100
Group name: users
    GID: 100
% ./getgrnam users
Group name: users
    GID: 100
```

□

Damit wir aber auch weiterhin SVR4 und 4.3+BSD bedienen können, stehen uns auf diesen Systemen noch drei weitere Funktionen zur Verfügung. Es wird von der Anwendung dieser Funktionen abgeraten, um Abhängigkeiten mit den Feldern in der Gruppendatei zu vermeiden. Ihnen sind `getgrgid(3)` und `getgrnam(3)` immer vorzuziehen.

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrent(void);
Rückgabewert: Zeiger auf eine Struktur oder NULL bei Fehler.

void setgrent(void);
void endgrent(void);
```

Um einen Record anzufordern, rufen wir `getgrent(3)` auf und lesen hinter her die `group`-Struktur aus. Somit sind wir in der Lage die gesamte Gruppendatei nach bestimmten Einträgen zu durchsuchen. Wie auch mit Passworddateien öffnet `setgrent(3)` die Gruppendatei und setzt den Cursor auf den Anfang der Datei zurück. Mit `endgrent(3)` schließen wir die Datei und geben alle assoziierten Ressourcen frei.

6.1.3 Zusätzliche Gruppen-IDs von Prozessen

Zu Zeiten von System 7 gehörte jeder Benutzer genau einer Gruppe an. Zwar konnten wir die Gruppen-ID eines Benutzers ändern, indem wir `chgrp(3)` aufriefen, jedoch gehörten wir weiterhin immer nur einer einzigen Gruppe an. Das Verhalten hat sich seit der Einführung von 4.2BSD geändert. Nun war es möglich, daß ein Benutzer mehreren Gruppen angehörte. In der Gruppendatei konnte ein Benutzer in bis zu 16 verschiedenen Gruppen eingetragen sein.

POSIX.1 beschreibt zusätzliche Gruppen-IDs als optional; fast alle Unices unterstützen dieses Feature vollständig. Die Konstante `NGROUPS_MAX` gibt die maximale Anzahl von Gruppen an, denen eine Benutzer angehören darf. Werden zusätzliche Gruppen-IDs nicht unterstützt ist `NGROUPS_MAX` 0.

Zwei Funktionen stehen uns für die Arbeit mit zusätzlichen Gruppen-IDs eines Prozesses zur Verfügung:

```
#include <sys/types.h>
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
Rückgabewert: Anzahl der zusätzlichen Gruppen-IDs oder -1 bei Fehler.

int setgroups(int ngroups, const gid_t grouplist[]);
Rückgabewert: 0 bei Erfolg und -1 bei Fehler.
```

`gidsetsize` Anzahl der Gruppen-IDs, die in `grouplist` gespeichert werden sollen.

`grouplist`
Feld mit den zusätzlichen Gruppen-IDs.

`ngroups`
Anzahl der Elemente des Arrays.

Die Funktion `getgroups(3)` füllt das Feld `grouplist` mit den Gruppen-IDs des aufrufenden Threads. Dabei werden nur bis zu `gidsetsize` Einträge in das Array geschrieben. Als Rückgabewert liefert sie die Anzahl der in `grouplist` gespeicherten Gruppen-IDs. Wenn `gidsetsize` 0 und `grouplist` NULL liefern, gibt die Funktion nur die Anzahl der zusätzlichen Gruppen-IDs zurück. Das ist beispielsweise hilfreich,

wenn wir einfach nur wissen möchten wie viele zusätzliche Gruppen-IDs mit einem Thread assoziiert sind.

Listing 6.4: Zusätzliche Gruppen-IDs mit getgroups(3) auflisten

Das folgende Programm ruft die Gruppen-IDs des laufenden Threads ab.

```

1  #include <grp.h>
2  #include "header.h"
3
4  int main (int argc, char **argv) {
5      struct group *grp;
6      gid_t        *grouplist;
7      int          numgroups;
8
9      /* get # of groups associated with calling thread */
10     numgroups = getgroups(0, NULL);
11     /* allocate buffer for group list */
12     grouplist = (gid_t *)malloc(sizeof(gid_t) * numgroups);
13
14     if (getgroups(numgroups, grouplist) == -1)
15         err_fatal("getgroups() failed\n");
16
17     printf("group\tID:\n-----\n");
18
19     if (numgroups > 0) {
20         while (numgroups--) {
21             grp = getgrgid(grouplist[numgroups]);
22             printf("%s\t%d\n", grp->gr_name, grouplist[numgroups]);
23         }
24     } else {
25         printf("numgroups = %d\n", numgroups);
26     }
27
28     return (0);
29 }
```

Listing 6.4: xcode/getgroups.c - Gruppen-IDs mit getgroups auflisten.



Mit `setgroups(3)` kann der Superuser die zusätzlichen Gruppen-IDs für den aufrufenden Thread laden, z.B. um ihn kurzfristig Zugriff auf andere Ressourcen zu ermöglichen. Dieser Vorgang erfordert Superuser-Privilegien. Die Liste `grouplist` referenziert und enthält `ngroups` Elemente.

6.2 Netzerkinformationen

Ebenso wie Benutzerinformationen werden auch fast alle Daten über die Netzwerkfähigkeiten eines Systems in Textdateien gespeichert. Viele BenutzerInnen eines UNIX-Systems haben meist schon mit mindestens einer dieser Dateien, nämlich `/etc/hosts`, auseinandergesetzt, um andere Rechner im Netzwerk mit ihrem Namen ansprechen zu können.

In diesem Abschnitt befassen wir uns mit den folgenden Konfigurationsdateien:

`/etc/hosts`

speichert IP-Adressen und Namen von Hosts, die dann leichter durch symbolische Namen im Netzwerk identifiziert werden können.

/etc/networks

listet alle bekannten Netzwerkadressen und deren Namen auf, damit sie leichter identifiziert werden können.

/etc/protocols

listet alle bekannten Protokollbezeichnungen des TCP/IP-Stapels auf.

/etc/services

enthält eine Liste von bekannten Diensten, die einem bestimmten Protokoll zugeordnet werden können.

6.2.1 /etc/hosts

Die Datei **/etc/hosts** wird in kleinen Netzwerken für die Zuordnung von IP-Adressen zu Namen oder umgekehrt eingesetzt. Traditionell stehen uns stehen zwei Funktionen zur Verfügung, um das programmatisch durchzuführen. Beide sind in SUSV als veraltet markiert, aber dennoch in vielen Codes enthalten.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
Rückgabewerte: Zeiger auf eine Struktur oder NULL bei Fehler.

void sethostent(int stayopen);
void endhostent(void);
```

name

Name oder IP-Adresse des Hosts, über den wir Informationen abfragen möchten.

addr

IP-Adresse des Hosts, über den wir Informationen abfragen möchten.

len

Länge von **addr**.

type

Bestimmt die Adressfamilie der IP-Adresse (IPv4 oder IPv6);

stayopen

Bestimmt, ob die Datei zwischen aufeinanderfolgenden Zugriffen geöffnet bleiben soll.

Um Hostinformationen über einen Namen abzufragen, wird die Funktion **gethostbyname(3)** mit dem Hostnamen als Argument aufgerufen. Haben Sie allerdings nur eine IP-Adresse, verwenden Sie **gethostbyaddr(3)** und übergeben die Adresse, die Länge der Adresse und die Adressfamilie als Parameter. Die Adressfamilie ist für IPv4 als **AF_INET** und für IPv6 als **AF_INET6** definiert. Das Domain Name System (DNS) wird oftmals für die Auflösung von Adressen und Host-Namen verwendet, wenn die lokalen Dateien keine Informationen enthalten. Auch wenn wir kein DNS im Netzwerk betreiben, können wir die Abfrage durchführen. Das System „weiß“ (meist konfiguriert durch **/etc/nsswitch**), wann das DNS oder die lokale Datenbank befragt werden muß.

Die Funktion **sethostent(3)** fordert die nächste Zeile der Datei an und **endhostent(3)** schließt den Socket für nachfolgende Abfragen.

Die Struktur **hostent** ist folgendermaßen in **<netdb.h>** definiert:

```
struct hostent {
    char *h_name;           /* official name of host */
    char **h_aliases;       /* alias list */
    int h_addrtype;         /* host address type */
    int h_length;           /* length of address */
    char **h_addr_list;     /* list of addresses */
}
```

Das Element `h_aliases` ist eine Liste mit alternativen Namen des Hosts, wenn Doppelbezeichnungen vorliegen. Das gleiche gilt auch für `h_addr_list`.

Listing 6.5: Hostinformationen mit `gethostbyaddr(3)` und `gethostbyname(3)` abfragen

Das folgende Testprogramm ruft die Hostinformationen für die als Parameter übergebenen IP-Adresse oder den Hostnamen ab.

```

1 #include <sys/socket.h>
2 #include <netinet/in.h>
3 #include <arpa/inet.h>
4 #include <netdb.h>
5 #include "header.h"
6
7 int main(int argc, char **argv) {
8     u_long             addr;
9     struct hostent *hp;
10    struct in_addr     ia;
11    struct in6_addr   ia6;
12    sa_family_t       af;
13    char              **p;
14    char              *basename = basename_ex(argv[0]);
15
16    if (argc < 2)
17        err_fatal("Usage: %s [<ip-address> | <hostname>]\n", basename);
18
19    /* determine given address family */
20    if (inet_pton(AF_INET, argv[1], &(ia.s_addr)))
21        af = AF_INET;
22
23    if (inet_pton(AF_INET6, argv[1], &(ia6.s6_addr)))
24        af = AF_INET6;
25
26    if (af != AF_INET && af != AF_INET6) {
27        printf("Not an IP address, trying to resolve by hostname...\n");
28
29        if ((hp = gethostbyname(argv[1])) == NULL)
30            err_fatal("gethostbyname() failed");
31    } else {
32        printf("Trying to lookup IP address...\n");
33
34        if (af == AF_INET) {
35            if ((hp = gethostbyaddr((char *)&ia.s_addr,
36                                    sizeof(ia.s_addr), AF_INET)) == NULL)
37                err_fatal("gethostbyaddr() failed");
38        } else {
39            if ((hp = gethostbyaddr((char *)&ia6.s6_addr,
40                                    sizeof(ia6.s6_addr), AF_INET6)) == NULL)
41                err_fatal("gethostbyaddr() failed");
42        }
43    }
44
45    for (p = hp->h_addr_list; *p != 0; p++) {
46        struct in_addr in;
47        char    **q;
48
49        memcpy(&in.s_addr, *p, sizeof(in.s_addr));
50        printf("%s\t%s", inet_ntoa(in), hp->h_name);
51
52        for (q = hp->h_aliases; *q != 0; q++)
53            printf(" %s", *q);
54

```

```

55         putchar( '\n' );
56     }
57
58     return (0);
59 }
```

Listing 6.5: xcode/gethostbyany.c - Hostinformationen mit gethostbyaddr(3) und gethostbyname(3) abfragen.

Mit `inet_ntop(3)` (presentation (Zeichenkette) **to** numeric (dezimal)) können wir herausfinden, ob eine Adresse angegeben wurde. Trifft das nicht zu (`inet_ntop` liefert 0) versuchen wir es mit `gethostbyname`, das bei Erfolg einen Zeiger auf eine `hostent`-Struktur zurückgibt. Ist `inet_ntop(3)` erfolgreich, befindet sich die IP-Adresse in binärer Form in `ia.s_addr` oder in `ia.s6_addr` mit IPv6. Diese Adresse können wir nun `gethostbyaddr(3)` übergeben und anschließend die Felder der `hostent`-Struktur aufschlüsseln. An die Aliase für den betreffenden Host gelangen wir, indem wir die Listen `h_aliases` und `h_addr_list` durchlaufen. Für den Hostnamen `www.google.de` und die Adresse `127.0.0.1` erhalten wir folgende Ausgaben:

```
% ./gethostbyany www.google.de
Not an IP address, trying to resolve by hostname...
216.239.59.103  www.l.google.com www.google.de www.google.com
216.239.59.104  www.l.google.com www.google.de www.google.com
216.239.59.147  www.l.google.com www.google.de www.google.com
216.239.59.99   www.l.google.com www.google.de www.google.com
% ./gethostbyany 127.0.0.1
127.0.0.1      localhost
```

□

6.2.2 /etc/networks

In `/etc/networks` sind Mappings für Netzwerke gespeichert. Sie wird ebenso wie `/etc/hosts` während des Bootvorgangs benötigt, wenn noch keine Server verfügbar sind, die das Mapping vornehmen könnten. Sie enthalten lediglich eine oder mehrere Netzwerkadressen und jeweils einen symbolischen Namen (Alias) für das betreffende Netzwerk. Die folgenden Funktionen lesen `/etc/networks`:

```
#include <netdb.h>

struct netent *getnetbyname(const char *name);
struct netent *getnetbyaddr(long net, int type);

Rückgabewerte: Beide geben Zeiger auf eine Struktur oder NULL bei Fehler zurück.

struct netent *getnetent(void);
void setnetent(int stayopen);
void endnetent(void);
```

name

Name oder IP-Adresse des Netzwerkes, dessen Informationen abgefragt werden sollen.

net

IP-Adresse des Netzwerkes, dessen Informationen wir abfragen möchten.

type

Adressfamilie von `net` (immer `AF_INET`).

stayopen

Bestimmt, ob die Datei zwischen aufeinanderfolgenden Zugriffen geöffnet bleiben soll.

`getnetbyname(3)` liefert die Struktur `netent` auf Basis des Namens oder der IP-Adresse, angegeben in `name`. Mit `getnetbyaddr(3)` fordern wir die Struktur `netent` mit Hilfe der in `net` definierten IP-Adresse an, die der in `type` angegebenen Adressfamilie, beispielsweise IPv4 (`AF_INET`) oder theoretisch auch IPv6 (`AF_INET6`), angehört. IPv6-Unterstützung bietet `getaddrinfo(3)`.

Ähnlich wie in den vorangegangenen Funktionen, fordert `getnetent(3)` die nächste Zeile der Datei an und speichert die Daten in der Struktur `netent`. Mit `setnetent(3)` wird die Position in der Datei wieder zurück gesetzt, und mit `endnetent(3)` die assoziierten Ressourcen wieder freigegeben.

Die Struktur `netent` ist folgendermaßen in `<netdb.h>` definiert:

```
struct netent {
    char      *n_name;           /* official network name */
    char      **n_aliases;       /* alias list */
    int       n_addrtype;        /* net address type */
    uint32_t  n_net;            /* network number */
}
```

Das Element `n_aliases` enthält eine nullterminierte Liste alternativer Netzwerknamen und `n_addrtype` die jeweilige Adressfamilie, die hier immer `AF_INET` ist. Beachten Sie daß in `n_net` die Adresse in NBO network byte order vorliegt.

Listing 6.6: Netzwerkinformationen aller Netze des lokalen Systems ausgeben

In diesem Beispiel bedienen wir uns `getnetent(3)` und durchlaufen alle Einträge, um alle Informationen der `netent`-Struktur auszugeben.

```
1 #include <arpa/inet.h>
2 #include <netdb.h>
3 #include "header.h"
4
5 int main(int argc, char **argv) {
6     char          **p, address[128];
7     struct netent *netbuf;
8
9     setnetent(0);
10    while ((netbuf = getnetent()) != NULL) {
11        static count = 0;
12
13        printf("/*** ENTRY %d ***/\n", ++count);
14        printf("    Network name: %s\n", netbuf->n_name);
15        printf("Network aliases:\n");
16
17        for (p = netbuf->n_aliases; *p; p++) {
18            printf("\tNetwork name: %s\n", *p);
19        }
20
21        switch (netbuf->n_addrtype) {
22            case AF_INET:
23                printf("    Address type: IPv4\n");
24                break;
25            case AF_INET6:
26                printf("    Address type: IPv6\n");
27                break;
28            default:
29                return (0);
30        }
31
32        memset((char *)address, 0, strlen(address));
33        int t = ntohl(netbuf->n_net); /* to host byte order */
34
```

```

35         if (inet_ntop(netbuf->n_addrtype, &t, address, 128) == NULL)
36             err_normal("inet_ntop() failed");
37         else
38             printf("Network address: %s\n", address);
39     }
40
41     endnetent();
42     return (0);
43 }
```

Listing 6.6: xcode/getnetent.c - Netzwerkinformationen aller Netze des lokalen Systems ausgeben.



Listing 6.7: Netzwerkinformationen eines spezifischen Netzes des lokalen Systems ausgeben

Diesmal rufen `getnetbyname(3)` und `getnetbyaddr(3)` auf, um alle Informationen bezüglich des angegebenen Netzwerks auszugeben. Den Namen oder die Netzwerkadresse übergeben wir als Parameter.

```

1 #include <sys/socket.h>
2 #include <netinet/in.h>
3 #include <arpa/inet.h>
4 #include <netdb.h>
5 #include "header.h"
6
7 void print_netent(struct netent *netbuf);
8
9 main(int argc, char **argv) {
10     struct netent *netbuf;
11     struct in_addr ia;
12     char        *basename = basename_ex(argv[0]);
13
14     if (argc < 2)
15         err_fatal("Usage: %s [<net-address> | <netname>]\n", basename);
16
17     if (!inet_pton(AF_INET, argv[1], &ia.s_addr)) { /* address not valid */
18         printf("Not a network address, trying to query by name...\n");
19
20         if ((netbuf = getnetbyname(argv[1])) == NULL)
21             err_fatal("getnetbyname() failed");
22         else
23             print_netent(netbuf);
24     } else {
25         printf("Got a network address, trying to query by addr...\n");
26
27         if (netbuf = getnetbyaddr(ia.s_addr, AF_INET))
28             err_fatal("getnetbyaddr failed");
29         else
30             print_netent(netbuf);
31     }
32
33     return (0);
34 }
35
36 void print_netent(struct netent *netbuf) {
37     char **p;
38     char    address[128];
39
40     printf("    Network name: %s\n", netbuf->n_name);
41     printf("Network aliases:\n");
```

```

43     for (p = netbuf->n_aliases; *p; p++)
44         printf("\tNetwork name: %s\n", *p);
45
46     printf("    Address type: %s\n",
47            (netbuf->n_addrtype == AF_INET ? "AF_INET" : "AF_INET6"));
48
49     memset((char *)address, 0, strlen(address));
50     int t = ntohl(netbuf->n_net); /* to host byte order */
51
52     if (inet_ntop(netbuf->n_addrtype, &t, address, 128) == NULL)
53         err_normal("inet_ntop() failed");
54     else
55         printf("Network address: %s\n", address);
56 }
```

Listing 6.7: xcode/getnetbyany.c - Netzwerkinformationen eines spezifischen Netzes des lokalen Systems ausgeben.

```
% ./getnetbyany loopback
Not a network address, trying to query by name...
    Network name: loopback
    Network aliases: local
        Address type: AF_INET
    Network address: 127.0.0.0
```



6.2.3 /etc/protocols

In `/etc/protocols` sind alle verfügbaren Protokolle des TCP/IP-Stacks des lokalen Systems. Es ist in der Regel nicht notwendig, Änderungen an dieser Datei vorzunehmen. Jedem DARPA-Protokoll ist eine interne Nummer zugeordnet, die im IP-Header hinterlegt wird. Stimmt der Protokollname nicht mit der DARPA-Nummer überein, arbeitet der Protokollstapel fehlerhaft.

Mit den beiden Funktionen `getprotobynumber(3)` und `getprotobyname(3)` können wir die Datei abfragen:

```
#include <netdb.h>

struct protoent *getprotobynumber(const char *name);
struct protoent *getprotobyname(int proto);

Rückgabewerte: Zeiger auf eine Struktur oder NULL bei Fehlerzurück.

struct protoent *getprotoent(void);
void setprotoent(int stayopen);
void endprotoent(void);
```

name

Name des Protokolls, dessen Informationen abgefragt werden sollen.

proto

Nummer des Protokolls, dessen Informationen abgefragt werden sollen.

stayopen

Gibt an, ob die Datei nach der Verarbeitung geöffnet bleiben soll oder nicht.

Der Parameter `name` gibt referenziert das entsprechende Protokoll durch den Namen (1. Spalte von `/etc/protocols`) während `proto` die Protokollinformationen mit Hilfe der DARPA-Nummer abfragt.

Beim Aufruf von `getprotoent(3)` wird die nächste Zeile der Datei angefordert und in der Struktur `protoent` gespeichert. Mit `setprotoent(3)` setzen Sie die Position in der Datei zurück und legen Sie fest, ob die Datei zwischen den Abfragen geöffnet bleiben soll oder nicht. Durch `endprotoent(3)` beenden Sie die Arbeit mit der Datei.

Die Struktur `protoent` ist in `<netdb.h>` folgendermaßen definiert:

```
struct protoent {
    char    *p_name;          /* Official name of the protocol. */
    char    **p_aliases;      /* A pointer alternative protocol names */
    int     p_proto;          /* The protocol number. */
}
```

Listing 6.8: Informationen aller Protokolle von /etc/protocols ausgeben

Das folgende Beispiel gibt alle Informationen über die in `/etc/protocols` aufgelisteten Protokoll aus. Dazu rufen wir `getprotoent(3)` solange auf, bis alle Protokolle abgefragt wurden.

```
1 #include <netdb.h>
2 #include "header.h"
3
4 void print_protoent(struct protoent *protobuf);
5
6 int main(int argc, char **argv) {
7     struct protoent *protobuf;
8
9     setprotoent(0);
10    while ((protobuf = getprotoent()) != NULL) {
11        print_protoent(protobuf);
12    }
13    endprotoent();
14
15    return (0);
16 }
17
18 void print_protoent(struct protoent *protobuf) {
19     char **p;
20     static int count;
21
22     printf("*** ENTRY %d ***\n", ++count);
23     printf("Protocol name: %s\n", protobuf->p_name);
24     printf("Protocol aliases:\n");
25
26     for (p = protobuf->p_aliases; *p; p++) {
27         printf("\t Alias: %s\n", *p);
28     }
29
30     printf("Protocol number: %u\n", protobuf->p_proto);
31 }
```

Listing 6.8: xcode/getprotoent.c - Informationen aller Protokolle von /etc/protocols ausgeben.



6.2.4 /etc/services

In der Datei `/etc/services` sind Zuordnungen zwischen benutzerfreundlichen ASCII-Bezeichnungen der Dienste und den Nummern nach RFC 1700. Die Lokation der Datei wird im Header `<netdb.h>` in der Konstante `_PATH_SERVICES` hinterlegt und zeigt gewöhnlich auf `/etc/services`. Beispielsweise ist dem

Dienst HTTP unter anderem das TCP-Protokoll auf Port 80 zugeordnet. Diese und andere Informationen sind in `/etc/services` zu finden.

POSIX stellt zwei Funktionen zum Zugriff auf die Felder in der Datei zur Verfügung, während ANSI C wieder drei weitere hinzugefügt hat.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);

Rückgabewerte: Zeiger auf eine Struktur oder NULL bei Fehler.

struct servent *getservent(void);
void setservent(int stayopen);
void endservent(void);
```

name

Name des Dienstes, dessen Informationen abgefragt werden sollen.

proto

Nummer des Protokolls, dessen Informationen abgefragt werden sollen.

stayopen

Gibt an, ob die Datei nach der Verarbeitung geöffnet bleiben soll oder nicht.

Die Funktion `getservbyname(3)` liefert einen Eintrag, der zu der Kombination von `name` und `proto` passt. Ist `proto` ein NULL-Zeiger, wird jede Übereinstimmung mit `servent.s_proto` zurückgeliefert. `getservbyport(3)` liefert die gleichen Informationen basierend auf der Kombination von `port` und `proto`. Mit den ANSI C Funktionen `setservent(3)` und `endservent` steuern wir die Verarbeitung der Datei.

Die Struktur `servent` ist in dem Header `<netdb.h>` folgendermaßen definiert:

```
struct servent {
    char    *s_name;      /* Official name of the service. */
    char    **s_aliases;  /* A pointer to alternative service names */
    int     s_port;       /* The port number at which the service resides */
    char    *s_proto;     /* The name of the protocol */
}
```

Um den richtigen Port für den Service HTTPS über TCP zu ermitteln, könnten wir folgenden Code verwenden:

```
struct servent *sp
int port;
...
if ((sp = getservbyname("https", "tcp")) == NULL)
    err_fatal("Could not query Service HTTPS for TCP\n");

port = sp->s_port;
```

Ist uns das Protokoll egal, wäre auch

```
struct servent *sp
int port;
char *proto_name;
...
if ((sp = getservbyname("https", NULL)) == NULL)
    err_fatal("Could not query Service HTTPS for any protocol\n");

port = sp->s_port;
proto_name = sp->s_proto;
```

ausreichend. Das erste passende Protokoll für den Service wird in `servent` gespeichert.

Auf einigen Systemen ist die Anzahl der Aliase auf 35 Einträge beschränkt, wird aber nicht durch POSIX spezifiziert und sollte bei Bedarf abgefragt werden, beispielsweise über die Konstante `ALIASES`.

Listing 6.9: Informationen aller Dienst von /etc/services ausgeben

Die Liste der Dienste ist sehr lang, so daß die Ausgabe mit `grep(1)` gefiltert werden sollte, um die richtige Information zu ermitteln.

```

1 #include <netdb.h>
2 #include "header.h"
3
4 void print_servent(struct servent *servbuf);
5
6 int main(int argc, char **argv) {
7     struct servent *servbuf;
8
9     setservent(0);
10    while ((servbuf = getservent()) != NULL) {
11        print_servent(servbuf);
12    }
13    endservent();
14
15    return (0);
16 }
17
18 void print_servent(struct servent *servbuf) {
19     char **p;
20     static int count;
21
22     printf("*** ENTRY %d ***\n", ++count);
23     printf("    Service name: %s\n", servbuf->s_name);
24     printf("Service aliases:\n");
25
26     for (p = servbuf->s_aliases; *p; p++) {
27         printf("\t Alias: %s\n", *p);
28     }
29
30     printf("    Service port: %u\n", servbuf->s_port);
31     printf("    Protocol name: %s\n", servbuf->s_proto);
32 }
```

Listing 6.9: xcode/getservent.c - Informationen aller Dienst von /etc/services ausgeben.



6.3 Accounting

Wir beschäftigen uns in diesem Abschnitt ausschließlich mit Login-Accounting. Unter UNIX werden dazu zwei Datenbanken geführt: zum einen `utmp` zur Überwachung aller aktiver Login-Sessions und `wtmp` zur Überwachung aller Logins und Logouts.

6.3.1 Der traditionelle Ansatz

Wie sollte es auch anders sein, es existieren wieder einmal unterschiedliche Ansätze für das Accounting: `utmp` und `utmpx`. Ersterer ist auf vielen SysV-basierten und einigen traditionellen 4.4BSD-Systemen zu

finden und wird hier der Vollständigkeit und aus Gründen der Portabilität besprochen. Letzterer wird weiter unten erläutert.

Die `utmp`-Datenbank ist nicht Teil der POSIX-Definition, die den XPG4.2-Ansatz verfolgt und stattdessen eine `utmpx`-Datenbank nutzt und gleich passende Funktionen mit identischer Funktionsweise liefert. Beides wird direkt im Anschluß an die `utmp`-Diskussion besprochen.

Mit jedem Login wird die `utmp`-Strukturen gefüllt, in `utmp` geschrieben und an die Datei `wtmp` angehängt. Beim Logout wird der `utmp`-Eintrag durch den `init`-Prozess entfernt und ein neuer Eintrag an `wtmp` angehängt. Zusätzlich definieren BSD-basierte Systeme auch eine `lastlog`-Struktur, die in gleichnamiger Datei gespeichert wird und Informationen über den letzten Login enthält.

Die in `<utmp.h>` definierte Struktur hat folgende Elemente, zumindest auf BSD-Systemen:

```
#define _PATH_UTMP      "/var/run/utmp"
#define _PATH_WTMP      "/var/log/wtmp"
#define _PATH_LASTLOG   "/var/log/lastlog"

#define UT_NAMESIZE     16
#define UT_LINESIZE     8
#define UT_HOSTSIZE    16

struct lastlog {
    time_t ll_time;           /* When user logged in */
    char ll_line[UT_LINESIZE]; /* Terminal line name */
    char ll_host[UT_HOSTSIZE]; /* Host user came from */
};

struct utmp {
    char ut_line[UT_LINESIZE]; /* Terminal line name */
    char ut_name[UT_NAMESIZE]; /* User's login name */
    char ut_host[UT_HOSTSIZE]; /* Host user came from */
    time_t ut_time;           /* When user logged in */
};
```

Es stehen uns einige Funktionen zur Manipulation der Login-Account-Daten zur Verfügung.

```
#include <utmp.h>

struct utmp *getutent(void);
struct utmp *getutid(struct utmp *ut);
struct utmp *getutline(struct utmp *ut);
struct utmp *pututline(struct utmp *ut);

Rückgabewerte: Zeiger auf eine Struktur oder NULL bei Fehler.

void setutent(void);
void endutent(void);
void utmpname(const char *file);
```

`ut`

Zeiger auf eine Struktur, die bearbeitet werden soll.

`file`

Dateiname für `utmp`, der für weitere Operationen verwendet werden soll.

`setutent(3)` setzt den Dateizeiger auf den Anfang zurück. Es wird empfohlen diese Funktion vor allen anderen Operationen aufzurufen. `endutent(3)` schließt alle verwendeten Ressourcen und sollte dann aufgerufen werden, wenn alle Operationen anderer Funktionen mit der Verarbeitung fertig sind. Mit `getutent(3)`

fragen Sie die nächste Zeile der Datei ab. Es wird eine Struktur mit den angeforderten Informationen zurückgegeben. Die Funktion `getutid(3)` führt eine vorwärts gerichtete Suche durch und liefert das Ergebnis zurück, dessen Felder mit dem Argument von `ut` übereinstimmen. Mit `getline(3)` wird eine vorwärts gerichtete Suche durchgeführt, die eine Struktur mit der ersten Übereinstimmung von `ut->ut_type == USER_PROCESS` oder `ut->ut_type == LOGIN_PROCESS`. Die Funktion `pututline(3)` schreibt die Struktur `ut` in die `utmp`-Datei und benutzt `getutid(3)`, um einen passenden Slot zu finden. Schlägt `getutid(3)` fehl, wird der Eintrag an das Ende der Datei angehängt.

Die Funktion `utmpname(3)` legt einen Dateinamen fest, der für nachfolgende Operationen verwendet werden soll. Die Datei muß spätestens zum Zeitpunkt der ersten Lese-/Schreiboperation existieren. Wird `utmpname(3)` nicht verwendet bevor andere Funktionen ausgeführt werden, so wird die Konstante `_PATH_UTMP` aus `<paths.h>` herangezogen. Normalerweise gibt es keinen Grund, diese Funktion direkt aus einer Anwendung heraus aufzurufen.

Listing 6.10: Implementierung des who(1)-Kommandos mit Hilfe von getutent(3)

Das folgende Beispiel durchläuft die `utmp`-Einträge vom Typ `USER_PROCESS` und gibt Logininformationen aus.

```

1 #include <stdio.h>
2 #include <utmp.h>
3
4 int main(void) {
5     struct utmp *buf;
6     char *a;
7     int i;
8
9     setutent();
10    buf = getutent();
11
12    while (buf != NULL) {
13        if (buf->ut_type == 7) {
14            printf("%-9s ", buf->ut_user);
15            printf("%-12s", buf->ut_line);
16            printf(" ");
17
18            a = (char *)ctime(&buf->ut_time);
19
20            for (i = 0; i < 16; i++)
21                printf("%c", a[i]);
22
23            if (strlen(buf->ut_host) != 0)
24                printf("(%s)\n", buf->ut_host);
25            else
26                putc('\n', stdout);
27        }
28
29        buf = getutent();
30    }
31 }
```

Listing 6.10: xcode/utmp.c - Implementierung des who(1)-Kommandos mit Hilfe von getutent(3).



6.3.2 Der standardisierte Ansatz

POSIX 1003.1-2001 definiert die `utmpx`-Struktur und Hilfsfunktionen mit gleicher Funktionsweise:

```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);

Rückgabewerte: Zeiger auf eine Struktur oder NULL bei Fehler.

void setutxent(void);
void endutxent(void);
```

id

ID nach der die Suche in der Datenbank durchgeführt werden soll.

line

Zeile nach der in der Datenbank gesucht werden soll.

utmpx

Struktur, die in die Datenbank geschrieben werden soll.

file

Dateiname für die neue Datenbankdatei.

Die Struktur ist in **utmpx.h** definiert:

```
struct utmpx {
    char          ut_user[] /* User login name. */
    char          ut_id[]   /* initialization process identifier. */
    char          ut_line[] /* Device name. */
    pid_t         ut_pid    /* Process ID. */
    short         ut_type   /* Type of entry. */
    struct timeval ut_tv    /* Time entry was made. */
}
```

Die **getutxent(3)**-Funktion liest den nächsten Eintrag aus der **utmpx**-Datenbank. Wenn die Datenbank noch nicht geöffnet ist, wird das automatisch erledigt. Wird das Ende der Datei erreicht, liefert die Funktion einen **NULL**-Zeiger zurück.

Mit **getutxid(3)** wird eine vorwärts gerichtete Suche ausgehend von der letzten Position in der Datenbank vorgenommen, bis ein Eintrag mit einer Übereinstimmung von **ut_type** mit **id->ut_type** gefunden wird, wenn **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME** angegeben wird. Wird hingegen in **id** entweder **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS** angegeben, liefert die Funktion einen Zeiger auf den ersten Eintrag, der mit einem dieser vier Typen und gleichzeitig dem Wert von **ut_id** übereinstimmt, zurück.

Um einen Eintrag vom Typ **LOGIN_PROCESS** oder **USER_PROCESS** zu finden bemühen wir **getutxline**. Sie führt eine vorwärts gerichtete Suche ausgehend von der letzten Position in der Datenbank durch und liefert einen Zeiger auf die Zeile zurück, deren **ut_line** und **line->ut_line** übereinstimmen.

Möchten wir eine Zeile in die **utmpx**-Datenbank schreiben, steht uns **pututxline(3)** zur Verfügung. Sie verwendet **getutxid(3)** um nach einer geeigneten Stelle zu suchen, falls sie sich nicht schon an einer solchen befindet. Es wird davon ausgegangen, daß wir bereits mit einer der **getutxid(3)**-Funktionen nach einer günstigen Position in der Datenbank gesucht wurde, bevor wir **pututxline(3)** aufrufen. Wenn dem nämlich so ist, wird **pututxline(3)** keine Suche durchführen, denn es weiß genau, wo wir uns befinden. Kann allerdings kein freier Slot gefunden werden, fügt sie einen neuen Eintrag am Ende der Datenbank ein. Da die Datenbank normalerweise nur vom Superuser bearbeitet werden darf, führt die Funktion ein **setuid(2)**-Programm auf, um den Eintrag zu prüfen und gegebenenfalls zu schreiben.

Den Eingabestrom setzen wir mit der Funktion **setutxent(3)** zurück. Das sollte vor jeder Suchanfrage geschehen, wenn die gesamte Datenbank durchsucht werden soll, denn die Suche beginnt nicht automatisch wieder am Anfang. Wenn wir mit der Arbeit an der Datenbank fertig sind, rufen wir **endutxent(3)**

auf, um den Stream zu schließen und alle Ressourcen freizugeben. `utmpxname(3)` funktioniert genauso, wie `utmpname(3)`, allerdings muss der neue Dateiname ein „x“ am Ende aufweisen, damit er leicht unterscheidbar ist.

6.4 Systeminformationen und Systemzeit

Benutzer loggen sich in Systeme ein und erfragen spezifische Informationen mit dem `uname(1)`-Kommando, das Auskunft über Betriebssystemversion und neben anderen auch über die Architektur des Systems gibt. Die Funktion gleichen Namens ermöglicht den programmatischen Ansatz.

```
#include <sys/utsname.h>
```

```
int uname(struct utsname *name);
```

Rückgabewert: negativ bei Fehler, sonst positiv.

buf

Zeiger auf eine Struktur, die mit den Systeminformationen gefüllt werden soll.

Die `utsname`-Struktur weist nach POSIX.1 mindestens folgende Member auf:

```
struct utsname {
    char sysname[]; /* name of the operating system */
    char nodename[]; /* name of this node */
    char release[]; /* current release of operating system */
    char version[]; /* current version of operating system */
    char machine[]; /* name of hardware type */
}
```

Dieser Konvention folgen neben SunOS, HP-UX, Linux und den 4.4BSD-Derivaten auch AIX und Irix, wobei HP den Member `char idnumber[SNLEN]` aufgenommen hat, der einen eindeutigen Code für die laufende Plattform beispielsweise als Seriennummer enthält. Linux definiert zusätzlich `char idnumber[SNLEN]` zur Anzeige einer Domain-Zugehörigkeit, was aber nur von geringem Nutzen ist, da ein Domainname nicht an Netzwerkarchitekturen gebunden ist und Hosts gleichzeitig Teil mehrerer Netzwerke sein können. Das gleiche trifft auf `nodename` zu, da der Kernel keinerlei Kenntnis über solche Dinge hat.

Die Systemzeit kann auf vielfältige Weise repräsentiert werden, beispielsweise in der Anzahl der verstrichenen Sekunden seit 1970 oder der Anzahl der Tick seit das System gestartet wurde. Standardmäßig wird die Zeit unter UNIX über die Anzahl der Sekunden seit dem 01.01.1970 00:00:00 UTC (`universal coordinated time`) errechnet. Diese als Kalenderzeit bezeichnete Angabe wird in dem Datentyp `time_t` gespeichert. Mit der Kalenderzeit kann Datum und Zeit gleichermaßen berechnet werden.

Die Funktion `time` liefert Datum und Zeit.

```
#include <time.h>
```

```
time_t time(time_t *tloc);
```

Rückgabewert: Zeitangabe oder -1 bei Fehler.

tloc

Wenn `t` nicht null ist, wird das Ergebnis in diesem Zeiger auf `time_t` abgelegt,

Beachten Sie, daß das Ergebnis immer als Rückgabewert geliefert wird auch wenn `tloc` nicht null ist. In einigen BSD-basierten System ist die `time`-Funktion so etwas wie eine Wrapper-Funktion, die eigentlich nur den Systemaufruf `gettimeofday(2)` auuftut. Der SVR4-Kernel initialisiert die Zeit durch `stime(2)`, das Superuser-Privilegien erfordert, während BSD-Systeme `settimeofday(2)` aufrufen. Die BSD-Funktionen bieten im Gegensatz zur ihren SVR4-Pendants Auflösungen im Millisekundenbereich.

Lesbare Zeit- und Datumsangaben erhalten wir von unterschiedlichsten Funktionen:

```
#include <time.h>

struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);

time_t mktime(struct tm *tm);

char *asctime(const struct tm *tm);
char *ctime(const time_t *timep);

size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

Rückgabewert: Zeiger auf tm-Struktur oder NULL bei Fehler.

Rückgabewert: Kalenderzeit oder -1 bei Fehler.

Rückgabewert: Zeiger auf nullterminierten String oder NULL bei Fehler.

Rückgabewert: Anzahl der Zeichen, die in s gespeichert wurden oder 0 bei Fehler.

`timep`

Zeiger auf eine Kalenderzeit, die aufgeschlüsselt werden soll.

`tm`

Zeiger auf eine Struktur, die in Kalenderzeit konvertiert werden soll.

`s`

Zeiger auf einen Buffer, der das Ergebnis aufnehmen soll.

`max`

Größe von des Buffers `s`.

`format`

Angabe der Darstellung des Zeichenkette.

`tm`

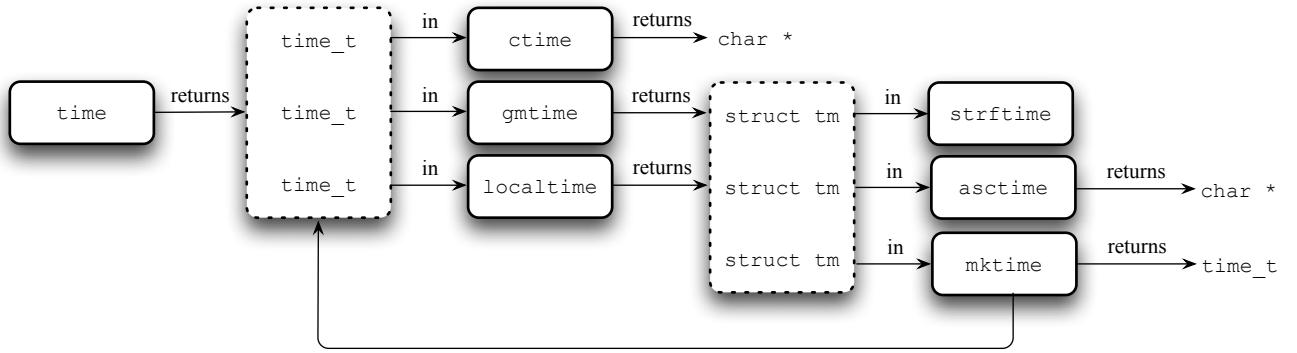
Zeiger auf eine Struktur mit aufgeschlüsselter Zeitangabe.

Die Funktionen `gmtime(3)` und `localtime(3)` konvertieren die als Argument übergebene Kalenderzeit in eine `tm`-Struktur, wobei erstere nach GMT und letztere nach lokaler Zeit konvertiert, wie sie in Zentraleuropa verwendet wird und auch die Sommer-/Winterzeit (`daylight saving time`) berücksichtigt. Mit Hilfe der Funktion `mktime(3)` konvertieren wir eine aufgeschlüsselte Zeit in Kalenderzeit, die als `time_t` zurückgegeben wird. Die Funktionen `asctime(3)` und `ctime(3)` erzeugen einen 26-Byte String, den `asctime(3)` aus der `tm`-Struktur und `ctime(3)` direkt aus der Kalenderzeit erzeugt. Schließlich können wir mit `strftime(3)` eine gezielte Formatierung der Zeitangabe vornehmen. Das Zusammenspiel der einzelnen Funktionen wird in Abbildung 6.1 dargestellt.

Formatangaben (Auszug aus `man strftime(3)`):

`%a` Wochentag abgekürzt, je nach lokaler Einstellung (`LOCALE`).

`%A` Voller Wochentag, je nach lokaler Einstellung (`LOCALE`).

Abbildung 6.1: Zusammenhang zwischen den `time`-Funktionen.

- %b Monat abgekürzt, je nach lokaler Einstellung (LOCALE).
- %B Voller Monat, je nach lokaler Einstellung (LOCALE).
- %c Bevorzugte Darstellung für Datum und Zeit, je nach lokaler Einstellung (LOCALE).
- %C Jahrhundert als zweistellige Zahl.
- %d Der Tag des Monats in dezimaler Darstellung.
- %D Entspricht %m/%d/%y. (Nur für den nordamerikanischen Raum).
- %e Wie %d, Tag des Monats in dezimaler Darstellung, vorangestellte 0 durch Leerzeichen ersetzt.
- %F Entspricht %Y-%m-%d (das ISO 8601 Datumsformat).
- %h Enspricht %b.
- %H Die Stunde als dezimaler 24-Stundenwert.
- %I Die Stunde als dezimaler 12-Stundenwert.
- %j Der Tag als dezimale Angabe.
- %k Enspricht %H nur das einstelligen Werten ein Leerzeichen vorangestellt wird.
- %l Enspricht %I nur das einstelligen Werten ein Leerzeichen vorangestellt wird.
- %m Der Monat als dezimaler Wert.
- %M Die Minute als dezimaler Wert.
- %n Neue Zeile.
- %p Entweder ‘AM’ oder ‘PM’ je nach Zeit und lokaler Einstellung (LOCALE). Mittag wird als ‘pm’ und Mitternacht als ‘am’ dargestellt.
- %P Wie %p aber in Kleinbuchstaben: ‘am’ oder ‘pm’ oder eine Zeichenkette nach lokaler Einstellung (LOCALE).
- %r Zeit in a.m- und p.m-Notation. Für POSIX-LOCALE entspricht das ‘%I:%M:%S %p’.
- %R Die Zeit in 24-Stundenangabe (%H:%M). Für Sekundenangaben siehe %T.
- %s Die Anzahl der Sekunden seit der Zeitrechnung (*epoch*) also beispielsweise, seit dem 01.01.1970 UTC.
- %S Die Sekunden in dezimaler Schreibweise
- %t Tabulatorzeichen.

- %T Die Zeit in 24-Stundenangabe mit Sekundenanteil (%H:%M:%S).
- %u Der Wochentag in dezimaler Darstellung mit Montag als 1 (siehe %w).
- %U Die Woche des aktuellen Jahres in dezimaler Darstellung, beginnend mit dem ersten Sonntag als ersten Tag der Woche 01 (siehe auch %V und %W).
- %V Die Woche in dezimaler Darstellung nach ISO 8601:1988. Hier ist Woche 1 die erste Woche des Jahres, die mindestens vier Tage lang ist und Montag der erste Tag der Woche (siehe auch %U und %W).
- %w Der Wochentag in dezimaler Darstellung (0 bis 6) mit Sonntag als 0 (siehe auch %u).
- %W Die Woche in dezimaler Darstellung mit Montag als ersten Tag der Woche 01.
- %x Die bevorzugte Darstellung des Datums je nach lokaler Einstellung (LOCALE), aber ohne Zeitangabe.
- %X Die bevorzugte Darstellung der Zeit je nach lokaler Einstellung (LOCALE), aber ohne Datumsangabe.
- %y Das Jahr in dezimaler Darstellung ohne Jahrhundertangabe.
- %Y Das Jahr in dezimaler Darstellung mit Jahrhundertangabe.
- %z Die Zeitzone als Stundenversatz (*offset*). Wird für RFC822-konforme Datumsangaben benötigt.
(heißt: "%a, %d %b %Y %H:%M:%S %z").
- %Z Die Zeitzone als Abkürzung.
- %+ Das Datum und die Zeit im `date(1)`-Format.
- %% Ein gedrucktes %-Zeichen.

Die Anwendung ist einfach, denn Sie entspricht im Wesentlichen der von `printf(3)` aus Abschnitt 5.6.

Kapitel 7

Prozessverwaltung

*Eh! Je suis leur chef, il fallait bien les suivre.
(Ah well! I am their leader, I really ought to follow them.)*

ALEXANDRE AUGUSTE LEDRU-ROLLIN

UNIX-Prozesse sind durch eine sog. *Process ID* (PID) eindeutig im System unterscheidbar und besitzen immer bestimmte Ausführungsechte, die durch die effektive und reale Benutzer-IDs sowie deren Gruppen-IDs beschrieben werden.

Prozesse befinden sich stets in einem genau definierten Zustand, beispielsweise angehalten oder laufend. Die möglichen Zustandsänderungen eines Prozesses sind in Abbildung 7.1 zusammengefaßt. Die Vierecke beschreiben die jeweiligen Zustände und die Pfeile die möglichen Zustandsänderungen. Ein Prozess kann nicht direkt in Zustände wechseln, die nicht durch Pfeile miteinander verbunden sind.

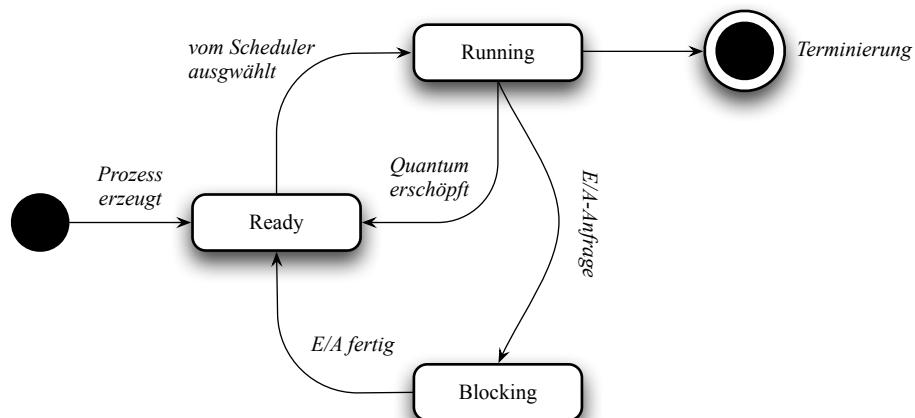


Abbildung 7.1: Zustandsdiagramm für Prozesse.

Sobald ein Programm für die Aktivierung als Prozess vorbereitet wird, befindet er sich im Zustand *New*. Wenn dieser Vorgang abgeschlossen ist, wird er in eine Warteschlange lauffähiger Prozesse eingetragen und hat damit den Zustand *Ready*. Wählt der Scheduler des Kernels den Prozess aus, beispielsweise weil er jetzt an der Reihe ist, so geht er in den Zustand *Running* über. Soll eine E/A-Operation durchgeführt werden, so blockiert (Zustand *Blocked*) der Prozess, bis diese Operation beendet ist und kehrt anschließend wieder in den Zustand *Ready* zurück. Unabhängig davon, ob der Prozess geplant oder ungeplant beendet wird, ist der letzte Zustand, den ein Prozess zu sehen bekommt, *Done*.

Eine E/A-Operation ist stets mit einem Systemaufruf verbunden, der einen Kontextwechsel (*context switch*) erfordert. Dabei übernimmt das Betriebssystem die Kontrolle über den Prozess, wechselt vom *User Mode* in den *Kernel Mode*. Die Information, die erforderlich ist, um einen Prozess in einem bestimmten Kontext auszuführen, wird als Prozesskontext (*process context*) bezeichnet. Das Betriebssystem zeichnet die jeweiligen Daten für einen der beiden Kontexte auf, so daß der Prozess den ursprünglichen Ausführungspfad wieder aufnehmen kann. Dinge wie Stack, Register, Sperren, der ausführbare Code und viele andere Informationen, werden zwischengespeichert und nach einem Kontextwechsel wieder hergestellt.

Um herauszufinden, in welchem Zustand sich ein bestimmter Prozess befindet, eignet sich der Befehl `ps(1)`. Um beispielsweise nach einem bestimmten Prozess in der Liste zu suchen, können sie

```
% ps -ef | grep myprocess
1000      15238  0.2  0.1    2056    976 pts/6      S+     18:05   0:00 myprocess
```

ausführen. Die mit *S+* gekennzeichnete Spalte zeigt den Zustand des Prozesses an, wobei *S* einen schlafenden Prozess bezeichnet, der sich in der **Foreground Process Group** befindet. Weitere Informationen über dieses Werkzeug erhalten Sie in den Manpages.

Läuft der Prozess erst einmal, nimmt er einen zusammenhängenden Bereich im Hauptspeicher ein, der als Prozessabbild (*process image*) bezeichnet wird. Das Image besteht aus einigen wichtigen Bereichen, die für unterschiedliche Aufgaben vorgesehen sind. Der Programmcode befindet sich im unteren Teil des Images (unten, von der Speicheradresse aus betrachtet) während initialisierte und statische Variablen in eigenen Bereichen untergebracht sind. Abbildung 7.2 zeigt ein einfaches Prozessabbild, wie es für ein typisches UNIX-Programm anzutreffen ist.

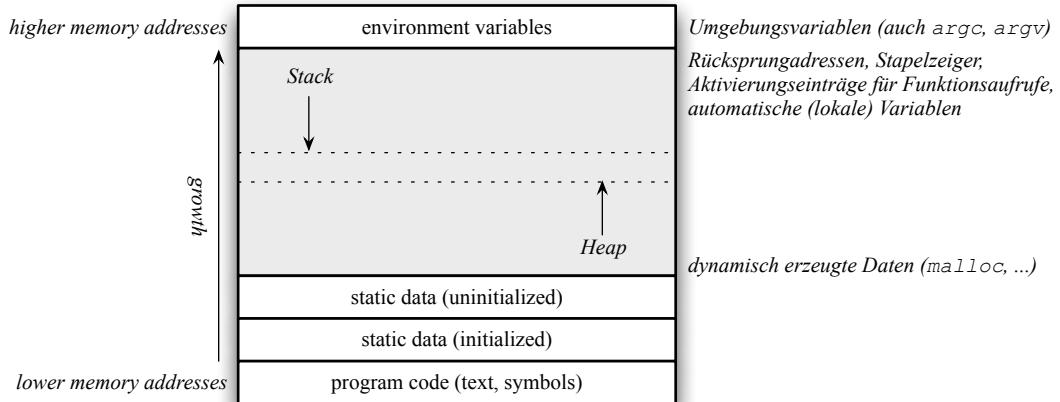


Abbildung 7.2: Verallgemeinertes Prozessabbild typischer UNIX-Prozesse.

Ein Aktivierungseintrag besteht aus einem Speicherbereich, der den Ausführungskontext eines Funktionsaufrufs vorhält. Bei jedem Funktionsaufruf wird ein neuer Aktivierungseintrag erzeugt, auf dem Stack platziert und wieder vom Stack entfernt, wenn die Funktion zurückkehrt. Der Eintrag enthält die Rücksprungadresse, die Werte der Argumente, die beim Funktionsaufruf kopiert wurden und einige Werte wichtiger CPU-Register. Auf Basis dieses Eintrages stellt der Kernel bei Rückkehr einer Funktion den Zustand des Prozesses wieder her.

Während der Stack streng nach dem LIFO-Prinzip (*last-in first-out*) aufgebaut ist, steht dem Prozess auch ein freier Speicherbereich zur Verfügung, der relativ unstrukturiert sein kann: der *Heap* (dt. Haufen oder Halde). Er wächst von unten nach oben und läuft dem Stack entgegen. Hier werden alle dynamisch erzeugten Datenstrukturen abgelegt, die beispielsweise von Funktionen der `malloc`-Familie erstellt wurden. Dieser Bereich wird nicht automatisch durch den Kernel organisiert, sondern bleibt so lange bestehen, bis der Prozess beendet wird. Die ProgrammiererInnen können auf dem Heap Platz schaffen indem sie nicht mehr benötigte Datenstrukturen mit `free(3)` freigeben. Eine wichtige Eigenschaft ist,

dass Datenstrukturen, die innerhalb einer Funktion von `malloc(3)` erzeugt wurden auch nach der Rückkehr der Funktion erhalten bleiben, allerdings nur, wenn der Prozess (hier: der Aufrufer) einen Zeiger auf diesen Speicherbereich verwaltet.

Statische Variablen sind in einem eigenen Speicherbereich untergebracht, je nach dem ob sie explizit bei ihrer Deklaration initialisiert wurden oder nicht. Das liegt daran, dass initialisierte Variablen Teil des ausführbaren Moduls auf dem Datenträger sind, und uninitialisierte Variablen nicht. Das gleiche trifft auch auf automatische (lokale) Variablen zu, denn sie werden erst initialisiert, wenn sie benötigt werden. Der Wert statischer Variablen ist schon beim Start des Prozesses bekannt.

7.1 Prozessbezeichner (PID)

Jeder Prozeß hat einen übergeordneten Prozess, der diesen quasi „hervorgebracht“ hat. Dadurch entsteht eine Baumstruktur in der jeder Prozess enthalten ist. Alle diese Prozesse haben eine positive ID größer 1. Der Prozess mit der ID 0 wird in der Regel dem Scheduler zugeordnet. Der Init-Prozess, die Mutter aller Prozesse, erhält immer die ID 1 und wird durch den Kernel erzeugt. Init (`/sbin/init`) steuert die Erzeugung neuer Prozesse und wird niemals, außer natürlich durch einen Neustart, beendet. Ein typischer Prozessbaum ist in Abbildung 7.3 dargestellt.

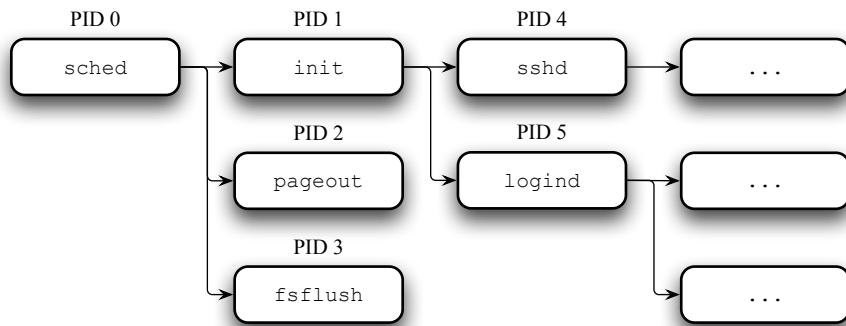


Abbildung 7.3: Typischer Prozessbaum eines UNIX-Systems.

Nachdem der Kernel `init` erzeugt hat, findet es in dem Verzeichnis `/sbin/init.d` einige Skripte, die in verschiedenen Runlevels in vordefinierter Reihenfolge gestartet werden. Gesteuert wird dieser Vorgang durch die Verzeichnisse nach Art von System V in `rc*`, die ihrerseits symbolische Links auf die Skripte in `/sbin/init.d` enthalten.

Es kann besonders auf modernen Systemen vorkommen, dass Benutzerprozesse eine ID jenseits von 100 oder gar 1000 erhalten, da der Kernel oftmals weitere Daemonen lädt, die zur Überwachung von Systemfunktionen notwendig sind. Beispielsweise laden die meisten SysV-basierten Systeme den sog. *Page Daemon* (`kpaged`), der für die Speicherverwaltung zuständig ist. Des Weiteren werden noch andere Prozesse für die Dateisysteme und Ereignisüberwachung des Kernels benötigt, auf die in diesem Text nicht eingegangen werden kann. Wir werden uns aber erneut mit dem Superprozess `init` auseinandersetzen, um zu erfahren, wie verweiste Prozesse zu Kindern von `init` werden.

Folgende Funktionen sind für die Abfrage der jeweiligen PIDs und den zugeordneten Benutzer- sowie Gruppen-IDs vorgesehen:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

Rückgabewert: Prozessnummer des aufrufenden Prozesses.

```
pid_t getuid(void);
```

Rückgabewert: Benutzer-ID des aufrufenden Prozesses.

```
pid_t geteuid(void);
```

Rückgabewert: Effektive Benutzer-ID des aufrufenden Prozesses.

```
pid_t getgid(void);
```

Rückgabewert: Gruppen-ID des aufrufenden Prozesses.

```
pid_t getegid(void);
```

Rückgabewert: Effektive Gruppen-ID des aufrufenden Prozesses.

getpid(2) ruft die PID des aufrufenden Prozesses ab, **getuid(2)** liefert die Benutzer-ID, **geteuid(2)** die effektive Benutzer-ID, **getgid(2)** die Gruppen-ID und **getegid(2)** die effektive Gruppen-ID des Aufrufers.

7.2 Prozesse erzeugen

Wie wir zu Beginn des Kapitels erfahren haben, kann nur ein laufender Prozess einen anderen erzeugen, sei es durch **fork(2)**, einen Aufruf von **exec(3)** oder **system(2)**. In diesem Abschnitt gehen wir auf die Besonderheiten beim Umgang mit Prozessen ein.

Folgende POSIX-Funktionen sind die Erzeugung von Prozessen involviert:

fork(2) und **vfork(2)**

erzugen Childs, wobei **fork(2)** eine vollständige Kopie eines Prozesses erzeugt und **vfork(2)** den Speicher des Parent nicht vollständig abbildet.

exec(2)

und ihre Verwandten erzeugen neue Prozesse ohne daß hinterher eine Parent-Child-Beziehung besteht.

system(2)

führt ein Kommando über die POSIX-Shell **sh(1)** aus; entspricht einem Aufruf von **/bin/sh -c <Kommando>**.

7.2.1 Die Funktion **fork**

Wie bereits eingangs erwähnt, kann ein Prozess nur erzeugt werden, wenn ein existierender Prozess einen anderen erzeugt, der dann als Kindsprozess (*child*) des erzeugenden Prozesses bezeichnet wird. Möchte ein Prozess einen anderen erzeugen, ruft er **fork(2)** auf.

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t fork(void);
```

Rückgabewert: 0 für Kindsprozesse, ID des Elternprozess oder -1 bei Fehler.

Wie aus der Beschreibung der Rückgabewerte ersichtlich wird, gibt **fork(2)** zwei Werte zurück. Der Kindsprozess gibt nach seiner Erzeugung immer 0 zurück, während der Elternprozess stets seine eigene ID zurückgibt. Warum das so ist, erfahren wir in Kürze.

Listing 7.1: Mit **fork(2)** Prozesse erzeugen.

Dieses Beispiel illustriert die Anwendung von **fork(2)**.

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include "header.h"
5
6 int main(void) {
7     int status;
8     pid_t pid;
9     char *command, *arguments[3];
10
11     /* create command and its arguments */
12     command = "/usr/bin/ls";
13     arguments[0] = "-l";
14     arguments[1] = "/";
15     arguments[2] = NULL;
16
17     /* allocate process space for ls command */
18     if ((pid = fork()) < 0)
19         err_fatal("fork error\n");
20
21     /* child code */
22     if (pid == 0)
23         execvp(command, arguments); /* execute command and its arguments */
24
25     /* parent code */
26     if (pid > 0)
27         waitpid(pid, NULL, 0); /* parent waiting for child to return */
28
29     return (0);
30 }

```

Listing 7.1: xcode/fork.c - Einfache Anwendung von `fork(2)` mit `execvp`.

Der Grund für zweifache Rückgabewerte liegt in der Tatsache begründet, daß ein Elternprozess mehrere Kinder haben kann. Des Weiteren gibt es keine Möglichkeit für den Elternprozess die IDs seiner Kindersprozesse zu erfahren. Durch `fork(2)` wird für Kindsprozesse 0 zurückgegeben, damit sichergestellt ist, daß der Kindsprozess nur einen Elternprozess hat. Dadurch ist der Kindsprozess andererseits in der Lage, die ID seines Elternprozesses mit `getppid(2)` abzufragen. Die Funktion `execvp(3)` ist erst einmal uninteressant für uns, sie besprechen wir in Abschnitt 7.2.3 auf Seite 156. □

Sowohl der Elternprozess als auch der Kindsprozess fahren mit der Programmausführung nach dem Aufruf von `fork(2)` fort, wobei der Kindsprozess eine vollständige Kopie des Prozessadressraums erhält, inkl. Stack, Heap und Daten. Allerdings operieren beide Prozesse in unterschiedlichen Addressbereichen, verwenden also nicht den gleichen Speicherbereich. Variablen die vor dem Aufruf von `fork(2)` initialisiert wurden, haben bis dahin in beiden Speicherbereichen die gleichen Werte, denn die laufenden Prozess Images wurden schließlich mittels `fork(2)` kopiert.

Moderne Implementierungen nehmen keine vollständige Verdopplung des Speicherbereichs vor, weil in vielen Fällen nach `fork(2)` ein Aufruf von `exec(3)` erfolgt. Stattdessen verwenden beide den gleichen Speicherbereich, doch sobald einer von beiden Änderungen (Schreibzugriffe) durchführt, wird der Kopievorgang angestoßen. Diese Technik wird als *copy on write* (COW) bezeichnet.

Ein wichtiger Aspekt im Umgang mit `fork(2)` ist die Handhabung assoziierter geöffneter Dateien der Eltern- und Kindsprozesse. Was passiert beispielsweise, wenn der Elternprozess drei Dateien durch einen File Descriptor referenziert und anschließend einen Kindsprozess erzeugt? Am einfachsten lässt sich die Sache anhand der drei Standardstreams `STDIN_FILENO`, `STDIN_FILENO`, `STDOUT_FILENO` und `STDERR_FILENO` erläutern. Eltern- und Kindsprozesse teilen sich den gleichen Dateioffset (*file offset*). Schreibt nun der Kindsprozess in `STDOUT_FILENO`, so muß der Dateioffset für den Elternprozess durch den Kindsprozess angepaßt werden. Somit kann der Kindsprozess in `STDOUT_FILENO` schreiben, während

der Elternprozess auf ihn wartet und anschließend selbst in `STDOUT_FILENO` schreibt. Dadurch wird die Ausgabe an die des Kindsprozesses angehängt. Würden beide die gleiche Datei referenzieren und keiner den anderen von Änderungen benachrichtigen, so würden die Daten vermischt werden.

Es gibt zwei Verfahrensweisen, wie mit File Descriptoren nach einem Aufruf von `fork(2)` umgegangen werden muß:

1. Der Elternprozess wartet auf den Kindsprozess bis er mit der Arbeit fertig ist. Anschließend werden alle gemeinsam genutzten File Descriptors entsprechend aktualisiert.
2. Nach `fork(2)` schließen Kinds- und der Elternprozess ihre nicht mehr benötigten File Descriptoren. Dadurch stehen beide in dieser Hinsicht nicht mehr in Beziehung zueinander.

Folgende Eigenschaften erbt der Kindsprozess von seinem Erzeuger:

- reelle Benutzer- und Gruppen-IDs, effektive Benutzer- und Gruppen-IDs
- zusätzliche Gruppen-IDs
- das verbundene Terminal
- das aktuelle Arbeitsverzeichnis
- SUID/GUID-Flags
- Session-ID
- das Wurzelverzeichnis
- `umask`-Einstellungen
- Umgebungsvariablen
- Resourcenbegrenzungen (`resource limits`)

Folgende Eigenschaften werden nicht vererbt:

- die Prozess-ID
- beide haben unterschiedliche Eltern
- die Werte von `tms_utime`, `tms_stime` und `tms_cutime` werden auf 0 gesetzt
- ausstehende Signale werden für den Kindsprozess verworfen
- die Warteschlange für ausstehende Signale wird für den Kindsprozess geleert.

Es gibt nur zwei Gründe, warum der Aufruf von `fork(2)` fehlschlagen könnte:

1. Die maximale Anzahl von Prozessen wurde erreicht. Dies deutet meist auf ein anderes, schwerwiegenderes Problem hin.
2. Die maximale Anzahl von effektiven Benutzer-IDs für Prozesse ist durch Resourcenbeschränkungen nicht mehr ausreichend.

Fast alle Serverprogramme machen intensiven Gebrauch von `fork(2)` oftmals in Verbindung mit einem direkten Aufruf von `exec(3)` (siehe Abschnitt 7.2.3). Der Elternprozess wartet auf Client-Anfragen und erzeugt zur Verarbeitung derselben einen Kindsprozess. Der Elternprozess kehrt nach `fork(2)` in den Ursprungsmodus zurück, um weitere Client-Anfragen zu befriedigen. Auch die Funktionalität von Shells basiert auf `fork(2)`. Sie selbst stellen den Elternprozess da und erzeugen nach der Eingabe durch den Benutzer einen Kindsprozess und warten auf die Fertigstellung. Durch Angabe eines kaufmännischen Und

(`&`) können die meisten Prozesse durch die Shell in den Hintergrund befördert werden, so daß der Elternprozess (also die Eingabeaufforderung selbst) wieder Befehle verarbeiten kann. Die sequenziellen Aufrufe von `fork(2)` und `exec(3)` werden in einigen Unices intern in einem einzigen Kommando kombiniert. Dadurch wird unmittelbar nach `fork(2)` ein `exec(3)`-Aufrufe getätig, ohne Operationen dazwischen zu zu lassen. Dieser Vorgang wird gemeinhin als *spawning* bezeichnet und kann mit `vfork(2)` (Abschnitt 7.2.2) realisiert werden.

Je nach Einsatz von `fork(2)` entsteht eine andere Beziehung zwischen Parent- und Child-Prozessen. Listing 7.2 erzeugt einen Child-Prozess, der jeweils einen eigenen Prozess erzeugt. Auf diese Weise entsteht eine Prozesskette, wie sie in Abbildung 7.4a zu finden ist.

Listing 7.2: Erzeugen einer Prozesskette.

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include "header.h"
4
5 int main(int argc, char **argv) {
6     char *basename = basename_ex(argv[0]);
7     pid_t pid;
8     int i, proc_num;
9
10    if (argc != 2)
11        err_fatal("Usage: %s <childs>\n", basename);
12
13    proc_num = atoi(argv[1]);
14
15    for (i = 0; i < proc_num; i++)
16        if ((pid = fork()) > 0)
17            break;
18        else if (pid < 0)
19            err_fatal("fork() failed");
20
21    printf("Process PID: %ld with parent %ld\n",
22           (long)getpid(), (long)getppid());
23
24    if (pid > 0)
25        waitpid(pid, NULL, 0); /* parent waiting for child to return */
26
27    exit(0);
28 }
```

Listing 7.2: xcode/process-chain1.c - Erzeugen einer Prozesskette.

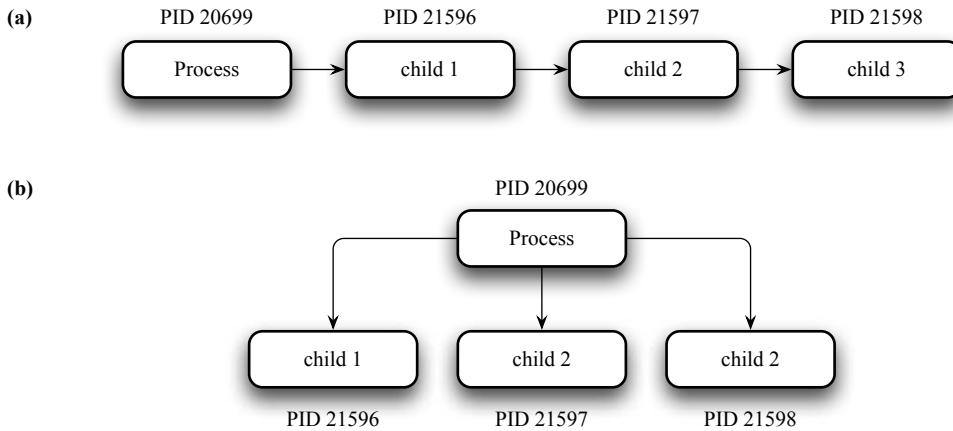
Bei jedem Durchgang der `for`-Schleife wird ein Prozess erzeugt und nur wenn die PID positiv ist (und das ist immer nur der Parent-Prozess) steigen wir aus der Schleife aus, denn wir kehren erst am Ende des Durchlaufs zum Parent zurück. Das sehen wir an der Ausgabe des Programms:

```
% ./process_chain1 3
Process PID: 21598 with parent 21597
Process PID: 21597 with parent 21596
Process PID: 21596 with parent 21595
Process PID: 21595 with parent 20699
```

Die Prozesskette wird in Abbildung 7.4a illustriert. Eine andere Beziehung, die weitaus häufiger anzutreffen ist, stellt sich wie ein Prozessbaum aus Abbildung 7.4b dar.

Listing 7.3 illustriert den Code, der zu einer solchen Konstellation führt. Der einzige Unterschied gegenüber Listing 7.2 ist, daß wir diesmal immer wenn ein Child erzeugt wurde aus der Schleife aussteigen. Auf diese Weise entstehen nur Childs, die alle von einem Parent abstammen.

Listing 7.3: Erzeugen einer Prozesskette

Abbildung 7.4: Eine Prozesskette (a) und ein Prozessbaum (b), erzeugt mit `fork`.

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     char *basename = basename_ex(argv[0]);
5     pid_t pid;
6     int i, proc_num;
7
8     if (basename = strrchr(argv[0], '/'))
9         basename = argv[0];
10    else
11        basename++;
12
13    if (argc != 2)
14        err_fatal("Usage: %s <childs>\n", basename);
15    else
16        proc_num = atoi(argv[1]);
17
18    for (i = 0; i < proc_num; i++)
19        if ((pid = fork()) <= 0)
20            break;
21
22    printf("Process PID: %ld with parent %ld\n",
23          (long)getpid(), (long)getppid());
24
25    if (pid > 0)
26        waitpid(pid, NULL, 0); /* parent waiting for child to return */
27
28    exit(0);
29 }

```

Listing 7.3: `xcode/process-chain2.c` - Erzeugen einer Prozesskette.

7.2.2 Die Funktion vfork

In manchen Situationen ist es notwendig, die Übertragung des elterlichen Speichers auf den Kindsprozess zu unterbinden. Das ist häufig der Fall, wenn `fork(2)` nur zur Ausführung eines `exec(2)`-Aufrufs benötigt wird. Der Kindsprozess läuft damit bis zum Aufruf von `exec(3)` oder `_exit(2)` im Adressbereich seines

Elternprozesses. Auf Systemen mit Unterstützung für virtuelles Memory Paging hat das einen großen Performancegewinn, nicht zuletzt auch durch den Einsatz von Copy On Write. Die `exec(2)`-Funktionsfamilie wird im nächsten Abschnitt besprochen.

Mit `vfork(2)` erzeugen wir einen Kindsprozess ohne den Adressbereich des Elternprozesses zu kopieren.

```
#include <unistd.h>

pid_t vfork(void);
Rückgabewert: 0 für Kindsprozesse, PID des Elternprozesses oder -1 bei Fehler.
```

Listing 7.4

Das folgende Programm demonstriert den Aufruf von `vfork(2)`, aber ohne `exec(2)`. Die Familie der `exec(2)`-Funktionen wird weiter hinten in Abschnitt 7.2.3 besprochen.

Listing 7.4: Effiziente Erzeugung von Childs mit vfork

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include "header.h"
4
5 int global_var = 0;
6
7 int main(void) {
8     int local_var = 10;
9     pid_t pid = -1; /* -1 not recommended, for demo only */
10
11    printf("Called main, not vforked yet\n");
12    printf("global_var: %d, local_var: %d, pid: %d\n",
13           global_var, local_var, pid);
14
15    if ((pid = vfork()) < 0) {
16        err_fatal("vfork failed\n");
17    } else if (pid == 0) { /* child code */
18        printf("vfork() OK\n");
19        global_var = 1;
20        local_var = 11; /* modify global (parent's) variable */
21        _exit(0); /* terminate child process only */
22    }
23
24    /* back in parents code */
25    printf("global_var: %d, local_var: %d, pid: %d\n",
26           global_var, local_var, pid);
27
28    return (0);
29 }
```

Listing 7.4: xcode/vfork.c - Effiziente Erzeugung von Childs mit vfork.

Beachten Sie, daß wir `_exit(2)` aufrufen und nicht `exit(2)`. Ersterer Funktionsaufruf sorgt dafür, daß nur der Kindsprozess beendet wird und nicht der Elternprozess, wie es mit letzterer Funktion geschehen würde.

Beatrachten wir die Bildschirmausgabe:

```
% ./vfork
Called main, not vforked yet
global_var: 0, local_var: 10, pid: -1
vfork() OK
global_var: 1, local_var: 11, pid: 2830
```

In der ersten Zeile sind wir gerade in `main` eingetreten und geben die Startwerte der drei vordefinierten Variablen an. Obwohl eingangs erwähnt wurde, daß `pid_t` eigentlich nur 0 annimmt, wenn ein Fehler bei `fork(2)/vfork(2)` aufgetreten ist. Dennoch habe ich die Variable `pid` mit -1 initialisiert, damit der Wer nicht bei der ersten Ausgabe undefiniert ist. Anschließend führen wir `vfork(2)` aus und verändern im Child Code die Werte der lokalen (Elternvariable) und der globalen Variable. Da sich Eltern- und Kindsprozess den gleichen Adressbereich teilen, haben beide gleichermaßen Zugriff auf die Variablen. Somit ist das Verhalten exakt so, wie wir es erwartet hatten. □

Die Verwendung von `vfork(2)` macht nur in Verbindung mit `exec(3)` Sinn, da die wesentlichen Unterschiede zwischen `fork(2)` und `vfork(2)` insbesondere durch die Einführung von *Copy On Write* erheblich verringert wurden. Ganz auf `vfork(2)` zu verzichten dürfte auch nicht schaden.

7.2.3 Die Funktionsfamilie `exec`

Im vorletzten Beispiel haben wir `exec(3)` eingeführt, ohne diese Funktionsfamilie zu besprechen. Sie finden meistens im direkten Zusammenhang mit `fork(2)` Anwendung. Durch `fork` wird ein anderer Prozess erzeugt und mit `exec(3)` ein anderes, externes Programm ausgeführt. Durch einen Aufruf von `exec(3)` wird der aufrufende Prozess (der Kindsprozess) vollständig durch das aufgerufene Programm ersetzt, inklusive Heap, Stack und Daten. Dabei ändert sich die PID des Aufrufers nicht, denn es wird kein neuer Prozess erzeugt. Abbildung 7.5 zeigt, wie das Process Image mit den Programmdaten überlagert wird. Das jeweilige Programm, welches zur Ausführung bestimmt ist, muß eine reguläre, ausführbare Datei sein und wird gemeinhin als *Process Image File* bezeichnet. Am Anfang des Kapitels haben wir das Thema schon einmal angeschnitten (Abbildung 7.2).

Wenn die Übersetzung des Quellcodes eines Programms erfolgreich abgeschlossen wurde, wird das Programm im letzten Schritt durch den Linker `ld(1)` eine ausführbare Datei, ein Process Image File, erstellt. Sie besteht aus folgenden Teilen:

Header

Der Header definiert die Größe der ausführbaren Datei und der anderen Sektionen und den Einstiegspunkt der Ausführung, wenn das Programm in das Process Image übertragen wurde. Das Format wird in den Dateien `/usr/include/a.out.h` und `/usr/include/filehdr.h` beschrieben. Wichtiger Bestandteil der ausführbaren Datei sind die so genannten Magic Numbers, die bestimmen, wie das File Image in ein Process Image verwandelt wird. Beispielsweise zeigt die magische Zahl 0410 an, daß die Instruktionen des Programms nur lesbar sind, während 0407 lesen und schreiben erlaubt. Sind die Instruktionen nur lesbar, so kann dieser Bereich mit anderen Prozessen geteilt werden, sofern sie das gleiche Process Image enthalten.

Text

Dieser Bereich enthält die Instruktionen, die ausgeführt werden, wenn das Programm gestartet wird.

Data

Der Datenbereich beinhaltet alle zu Beginn der Ausführung initialisierten Daten. Dazu gehören beispielsweise initialisierte Felder und Strukturen außerhalb von Subroutinen.

BSS

Hier finden alle nicht initialisierten Daten Platz. Wenn das Programm in einen Prozess überführt wird, initialisiert das Betriebssystem diese Daten zu 0. Da alle Daten initialisiert sind, wird nur die Größe des BSS-Bereichs in dem Image vermerkt, schließlich ist der Inhalt bekannt.

Relocation

Der Relocation-Teil eines Programms zeigt an, wie die Datei verändert werden muß, wenn sie gegen andere Dateien gelinkt wird. Nachdem ein ausführbares Programm erzeugt werden konnte, wird dieser Teil aus dem Image entfernt.

Symbol Table

Die Symboltabelle verbindet die Symbole (wie etwa Konstanten) mit den Positionen in der Datei. Diese Informationen ist besonders für das Debugging wichtig. Nach der Übersetzung eines Programms kann die Symboltabelle entfernt werden, beispielsweise mit dem Werkzeug `strip(2)`.

Wenn wir schon dabei sind, werfen wir noch schnell einen Blick auf die Bestandteile eines Process Image. Einige Bereich haben eine ähnliche Bedeutung wie die zuvor genannten, andere sind hinzugekommen.

Text

Der Text-Bereich entspricht genau dem der Image File. Die Größe dieses Bereiches ändert sich während der Ausführung nicht.

Data

Das Data-Segment eines Prozesses wird vor das BSS-Segment gesetzt und zuvor mit 0 initialisiert. Das System erlaubt stets Schreib- und Lesezugriffe auf das Data-Segment, dieser Bereich kann mit den beiden Systemaufrufen `brk(2)` und `sbrk(2)` verändert werden.

Stack

Der Stack wird während der Erzeugung des Prozesses etabliert und für die folgenden Aufgaben verwendet:

- Alle Variablen, die beim Aufruf von Subroutinen initialisiert werden, legt das System im Stack ab und entfernt sie wieder, wenn die Routine verlassen wird.
- Hier finden die Argumente Platz, die beim Aufruf des Programms übergeben werden.
- Die Umgebungsvariablen (siehe `sentenv(3)` und `export(8)`) sind Teil der Prozessumgebung und befinden sich als Schlüssel-Werte-Paare (Schlüssel = Wert) im Stack.

Das Betriebssystem verwaltet den Stack selbstständig. Er wird je nach Bedarf vergrößert oder verkleinert.

User Block

In diesem Bereich werden alle Informationen über den Prozess selbst hinterlegt. Sie befinden sich im Kernel-Space und ist normalen Prozessen nicht zugänglich. Der User Block bleibt konstant und wird beim Auslagern des Prozesses berücksichtigt.

Die Padding-Region zwischen Text und Data im Process Image wird für die hardware-basierte Speicherverwaltung eingerichtet, um der Ausrichtung der Speicherseiten zu entsprechen. Im Gegensatz dazu wird der Padding-Bereich zwischen BSS und Stack für die dynamische Erweiterung beider Bereiche benötigt.

Machen wir einen kleinen Ausflug in die Assemblerwelt, um zu verstehen, wie Process Images organisiert sind und warum wir die unterschiedlichen Sektionen benötigen.

Listing 7.5: Beispielprogramm für die Betrachtung von Process Images.

Das folgende kleine C-Programm zeigt einige Möglichkeiten, Variablen zu deklarieren und zu initialisieren. Je nach Vorgehensweise werden sie in unterschiedlichen Segmenten untergebracht und anders im Speicher verwaltet. Das eigentliche binäre Image ist im nächsten Listing abgebildet.

```

1 #include "header.h"
2
3 int x = 1; /* x placed in data */
4 int y;       /* y placed in bss */
5
6 int main(void) {
7     int z;      /* allocation of z on stack */
8     printf("variable x from data section has value %d\n", x);
9
10    y = 2;     /* code put in text */
11    printf("variable y from bss section has value %d\n", y);
12
13    z = 3;
14    printf("local variable z has value           %d\n", z);
15
16    return 0; /* retval in reg %o0 */

```

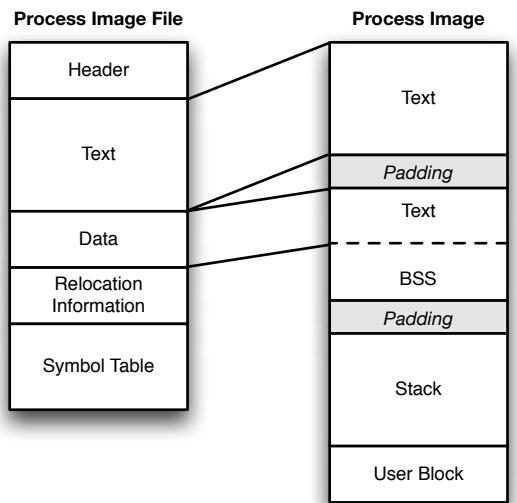


Abbildung 7.5: Layout der Process Images und File Images

17 }

Listing 7.5: *xcode/process_image_c.c* - Beispielprogramm für die Betrachtung von Process Images.**Listing 7.6: Assemblerversion des Beispielprogramms.**

Nach der Übersetzung des Programms aus Listing 7.5 erhalten wir ein Process Image, daß folgendermaßen in Assembler abgebildet wird.

```

1    .global main
2  main:
3      save %sp,-104,%sp
4
5      /* load value from initialized non-stack variable "x" */          */
6      set x,%o0
7      ld [%o0],%o1
8      set fmt1,%o0
9      call printf
10     nop
11
12     /* place value into uninitialized non-stack variable "y" */        */
13     mov 2,%o1
14     set y,%o0
15     st %o1,[%o0]
16
17     /* load value from non-stack variable "y" */                         */
18     set y,%o0
19     ld [%o0],%o1
20     set fmt2,%o0
21     call printf
22     nop
23
24     /* Place value into local variable "z" on stack */                   */
25     /* note that the address of "z" is not z but %fp-4 instead */        */
26     mov 3,%o0
27     st %o0,[%fp-4]
28
29     /* load value from local variable "z" on stack */                     */
30     /* note that the address of "z" is not z but %fp-4 instead */        */

```

```

31      ld    [%fp-4],%o1
32      set   fmt3,%o0
33      call  printf
34      nop
35
36      /* place return value in %io here, will be in %o0 for caller */
37      clr   %io
38
39      ret
40      restore
41
42      .section ".data"
43 x:   .word 1
44
45      .section ".bss"
46 y:   .skip 4
47
48      .section ".rodata"
49 fmt1:   .asciz "variable x from data section has value %d\n"
50 fmt2:   .asciz "variable y from bss section has value %d\n"
51 fmt3:   .asciz "local variable z has value %d\n"

```

Listing 7.6: xcode/process_image_asm.c - Das Process Image in Assembler.

Der Assembler kann (besser: muß) zwischen den Sektionen im Source Code umschalten, je nach dem welches Objekt bearbeitet wird. Das erledigt er mit sogenannten Pseudo-Ops, die keinen Code erzeugen und auch als Assembler-Direktiven bekannt sind:

.section ".text"

Der Assembler schaltet zur .text-Sektion um. Standardmäßig beginnt der Assembler immer in der .text-Sektion.

.section ".data"

Der Assembler schaltet zur .data-Sektion um.

.word wird gefolgt von einer Liste von Werten

.byte wird gefolgt von einer Liste von Werten

```

.byte "Hello, world"
.byte 'H','e','l','l','o','',' ','w','o','r','l','d'

```

.ascii Zeichenkette in Anführungszeichen

```

test:   .string "Hello, world!"
# Ergebnis: 0x48656C6C6F2C20776F726C6421.

```

.asciiz NUL-terminierte Zeichenkette in Anführungszeichen

```

test:   .string "Hello, world!\n"
# Ergebnis: 0x48656C6C6F2C20776F726C642100.

```

.section ".bss"

Der Assembler schaltet zur .bss-Sektion um.

.skip Setzt den Location Counter um die Anzahl der Bytes vor

```

.csect data[rw]
.skip 444,0
...
mine: # mine momentan bei Offset 0x1BC in csect data[rw].

```

```
section ".rodata"
```

Schreibgeschützte Sektion mit nur lesbaren initialisierten Werten.

```
.align
```

Variablen von der Größe eines Wortes (word, meist 16 Bit) müssen `.align 4` (32 Bit) ausgerichtet sein.

```
.global
```

Erlaubt die Referenzierung eines Symbols über die aktuelle Quelldatei hinaus.

Je nach Anwendung kann der Compiler und/oder Linker weitere Sektionen in das Process Image File integrieren, die ebenfalls in den Speicher als Process Image geladen werden. Lediglich der oben beschriebene grundlegende Aufbau muß eingehalten werden. Je nach Plattform und Software-Architektur kann die Struktur allerdings deutlich abweichen.

In C++ müssen die Konstruktoren der Bibliotheken (*initializer, startup code*) vor den Konstruktoren der Anwendungsprogramme aufgerufen werden. Ebenso müssen die Destruktoren (*finalizers*) der Programme vor denen der Bibliotheken aufgerufen werden. Dazu erzeugt der C++-Compiler unter Umständen vier zusätzliche Programmsektionen:

```
.init gefolgt von Zeigern auf die Initialisierer der Bibliotheken
.ctor gefolgt von Zeigern auf die Konstruktoren der Anwendung
.dtor gefolgt von Zeigern auf die Destruktoren der Anwendung
.fini gefolgt von Zeigern auf die Finalisierer der Bibliotheken
```

Eine weitere Besonderheit sind überladene Funktionsnamen, so daß der Compiler (und/oder der Linker) auf sog. *mangled function names* zurückgreifen muß, beispielweise:

```
void func(double d); // wird zu func_d kodiert
int func(float f); // wird zu func_f kodiert
```

Zu guter Letzt gibt es noch Templates, die für den Compiler nicht sichtbare oder überflüssige Routinen darstellen. In den meisten Fällen schickt der Compiler den Code für einen Probelauf an den Linker, um an dessen Fehlermeldungen zu erkennen, welche Routinen noch fehlen, damit sie anschließend für diese Version des Templates erzeugt werden können.

□

Insgesamt stehen uns sechs `exec(3)`-Funktionen zur Verfügung, die grundsätzlich alle jederzeit verwendet werden dürfen. POSIX definiert die folgenden Funktionen zur Ausführung von Programmen:

```
#include <unistd.h>

int execl(const char *path, ..., char * /*NULL*/);
int execv(const char *path, char *const argv[]);
int execle(const char *path, ..., char * /*NULL*/, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, ..., char * /*NULL*/);
int execvp(const char *file, char *const argv[]);
```

Rückgabewerte: Kehren bei Erfolg nicht zurück, geben bei Fehler -1 zurück.

path

Pfad zu dem Programm, das ausgeführt werden soll.

file

Dateiangabe, die verwendet werden soll, um einen Pfad zur Ausführung zu generieren. Enthält file einen Slash (/), so wird es als Pfadangabe interpretiert. Andernfalls wird die Ausführbare Datei im Pfad, durch die Umgebungsvariable PATH spezifiziert, gesucht.

arg
Argument für **path** oder **file**.

argv
Array von Argumenten für **path** oder **file**.

envp
Zeiger auf ein Array von Umgebungsvariablen.

Die **exec(2)**-Funktionen kehren nicht zurück, da der ursprüngliche Prozess das neue **Process Image** überlagern würde.

Zusätzlich zu dem Pfad und den Argumenten wird ein Zeiger auf ein Array von Umgebungsvariablen definiert. **path** und **environ** werden mit einem abschließenden NULL-Zeiger initialisiert, die beide nicht in **argc** gezählt werden. Das Feld **environ** ist für **execle(3)** vorgesehen, um Umgebungsvariablen in die Ausführung des Programms einzuflechten. Beispielsweise würde der Funktionsaufruf

```
char *env[] = {"DISPLAY=192.168.0.129:0.0", "LOGNAME=Steve", (char *)0};

if ((execle ("/bin/ls", "ls", "-l", (char *)0, env) < 0) {
    perror("execle failed\n");
    exit(errno);
}
```

zuerst die Umgebungsvariablen initialisieren und anschließend **execle(3)** mit einer variablen Argumentliste aufrufen. Beachten Sie, daß sowohl **env** als auch die Argumentliste durch einen NULL-Zeiger abgeschlossen werden müssen. Die Funktionen **exec1(3)**, **execlp(3)** und **execle(3)** fordern den Dateinamen des auszuführenden Programms als separates Argument der variablen Argumentliste an.

Alle Funktionen werden nach folgendem Schema aufgerufen:

exec1

```
#include <unistd.h>

/* execute ls command given the l argument */
if ((exec1 ("/bin/ls", "ls", "-l", (char *)0) < 0) {
    perror("exec1 failed\n");
    exit(errno);
}
```

execle

```
#include <unistd.h>

/* set enviroment variables and exec "ls -l" command */
char *env[] = {"DISPLAY=192.168.0.129:0.0", "LOGNAME=Steve", (char *)0};

if ((execle ("/bin/ls", "ls", "-l", (char *)0, env) < 0) {
    perror("execle failed\n");
    exit(errno);
}
```

execlp

```
#include <unistd.h>

/* execute "ls -l" command searching in the PATH environment variable */
if ((execlp ("ls", "ls", "-l", (char *)0, env) < 0) {
    perror("execlp failed\n");
    exit(errno);
}
```

execv

```
#include <unistd.h>

char *cmd[] = { "ls", "-l", (char *)0 };

/* execute "ls -l" given the command as an array */
if ((execv ("/bin/ls", cmd) < 0) {
    perror("execve failed\n");
    exit(errno);
}
```

execve

```
#include <unistd.h>

/* set environment variables and exec "ls -l" command */
char *env[] = {"DISPLAY=192.168.0.129:0.0", "LOGNAME=steve", (char *)0};

if ((execve ("/bin/ls", cmd, env) < 0) {
    perror("execve failed\n");
    exit(errno);
}
```

execvp

```
#include <unistd.h>

char *cmd[] = { "ls", "-l", (char *)0 };

/* execute "ls -l" given the command as an array */
if ((execvp ("ls", cmd) < 0) {
    perror("execve failed\n");
    exit(errno);
}
```

Oftmals ist **execve(3)** als Kernelfunktion implementiert, während alle anderen Bibliotheksfunktionen sind (wie beispielsweise unter Linux aber nicht Solaris), die ihrerseits wiederum auf **execve(3)** zurückgreifen. Es ist von der jeweiligen UNIX-Implementierung abhängig welche Variante zutrifft.

Listing 7.7: Einsatz von execle(3) und execlp(3)

```
1 #include "header.h"
2
3 char *env[] = {"MYVAR=myvar_contents", "PATH=/tmp", (char *)0};
4
5 int main(void) {
6     pid_t pid;
7
8     if ((pid = fork()) < 0)
9         err_fatal("fork failed\n");
10    else if (pid == 0) /* child code */
11        if (execle("/bin/ls", "ls", (char *)0, env) < 0)
12            err_fatal("execle failed\n");
13
14    if (waitpid(pid, NULL, 0) < 0)
15        err_fatal("waitpid failed\n");
16
17    if ((pid = fork()) < 0)
18        err_fatal("fork failed\n");
```

```

19     else if (pid == 0) /* child code */
20         if (execlp("/bin/ls", "another arg", (char *)0) < 0)
21             err_fatal("execlp failed\n");
22
23     return(0);
24 }
```

Listing 7.7: xcode/execl.c - Die Funktionen execle(3) und execlp(3) im Einsatz.



Bevor wir mit der Besprechung der Prozessverwaltung unter Unix fortfahren, möchte ich noch an einem ganzheitlichen Beispiel die Zusammenarbeit der drei wichtigen Funktionen `fork(2)`, `exec(3)` und `wait(2)` illustrieren.

Listing 7.8: Implementierung einer einfachen Shell

Das folgende Beispiel zeigt wie ein Kindsprozess mit `fork(2)` erzeugt, mit `exec(3)` externe Programme gestartet und mit `wait(2)` die korrekte Beendigung des Kindsprozesses beobachtet werden kann.

```

1 #include "header.h"
2
3 #define INPUT_BUFSIZ 512
4
5 /* prototype */
6 void exec_cmd(char *cmd);
7
8 int main(int argc, char **argv) {
9     char buffer[INPUT_BUFSIZ];
10
11    /* the initial user prompt */
12    printf("> ");
13
14    /* poll user input and exec commands */
15    while (fgets(buffer, INPUT_BUFSIZ, stdin) != NULL) {
16        exec_cmd(buffer);
17        printf("> ");
18    }
19
20    return (0);
21 }
22
23 void exec_cmd(char *cmd) {
24     pid_t pid;
25
26     /* create child and exec cmd */
27     if ((pid = fork()) < 0)
28         err_fatal("fork failed\n");
29
30     if (pid == 0) { /* child code */
31         execl("/sbin/sh", "sh", "-c", cmd, NULL);
32         err_fatal("execl failed\n");
33     }
34
35     /* parent waits for its child to exit */
36     if (waitpid(pid, NULL, 0) < 0)
37         err_fatal("waitpid failed\n");
38 }
```

Listing 7.8: xcode/simpleshell.c - Einfache Shell mit Hilfe von fork, exec(3) und wait.



7.2.4 Die Funktion system

Natürlich könnten wir davon ausgehen, die `exec(3)`-Funktionen auch für Shell-Kommandos zu verwenden, doch diesmal wollen wir nicht, daß ein Programm durch den Kernel gestartet wird, sondern ein sog. `Command Processor` die Ausführung übernimmt. Damit wir ein Shell-Kommando ausführen können, verwenden wir die `system(2)`-Funktion.

```
#include <stdlib.h>
```

```
int system(const char *command);
```

Rückgabewert: abhängig von command.

Wenn `command` ein NULL-Zeiger ist, findet die Funktion selbst heraus, ob ein Kommandoprozessor auf dem System vorhanden ist. Andernfalls dient `command` als Angabe eines gültigen Kommandoprozessors.

Die Umgebung des ausgeführten Kommandos verhält sich dann so, als wäre `fork(2)` zur Erzeugung eines Kindsprozesses genutzt worden, das seinerseits `exec(3)` mit dem sh-Interpreter ausführt:

```
execl("/bin/sh", "sh", "-c", command, (char *)0);
```

Wenn der Rückgabewert von `system(2)` nicht -1 ist, kann der Wert über die wait-Makros dekodiert werden (siehe Abschnitt 7.3.2). Beachte folgenden Zusammenhang: `system(2)` muß SIGINT und SIGQUIT ignorieren, aber SIGCHLD blockieren, während der Prozess auf seinen Kindsprozess wartet. Die Signalbehandlung für das ausgeführte Kommando wird durch die beiden Systemaufrufe `fork(2)` und `exec(3)` bestimmt. Wenn beispielsweise beim Aufruf von `system(2)` SIGINT abgefangen wird oder auf SIG_DFL steht, dann wird der Kindsprozess mit SIGINT auf SIG_DFL gestartet.

Das Ignorieren von SIGINT und SIGQUIT im Elternprozess soll Probleme bei der Abstimmung mit anderen Prozessen (beispielsweise, wenn zwei gleichzeitig versuchen, von dem gleichen Terminal zu lesen) verhindern, daß das ausgeführte Kommando eines dieser Signale ignoriert oder behandelt. Außerdem ist es immer die richtige Reaktion auf das Signal, wenn die BenutzerInnen interaktive Kommandos starten, die synchron ausgeführt werden. In beiden Fällen werden die Signale nur dem Kindsprozess zugestellt, nicht der Anwendung selbst. Allerdings könnte es eine Situation geben, in der das Ignorieren der Signale nicht den gewünschten Effekt hat: nämlich dann, wenn die Anwendung die `system(2)`-Funktion einsetzt, um für die BenutzerInnen unsichtbare Aufgaben zu erledigen. Wenn nun die Tastenkombination STRG-C (Unterbrechung) gedrückt wird, während `system(2)` diese Aufgabe erledigt, würden wir erwarten, daß die Anwendung beendet wird, doch nur das ausgeführte Kommando wird terminiert.

Die folgende Beispielimplementierung zeigt uns, daß es gar nicht so einfach ist, einen POSIX-konformen Systemaufruf für die `system(2)`-Funktion zu implementieren.

```

1 #include <signal.h>
2
3 int system(const char *cmd) {
4     int             stat;
5     pid_t          pid;
6     struct sigaction sa, sigint_mask, sigquit_mask;
7     sigset_t        saveblock;
8
9     if (cmd == NULL)
10        return (1);
11
12     sa.sa_handler = SIG_IGN;
13     sigemptyset(&sa.sa_mask);
14     sa.sa_flags = 0;
15
16     sigemptyset(&sigint_mask.sa_mask);
17     sigemptyset(&sigquit_mask.sa_mask);
18
```

```

19     sigaction(SIGINT, &sa, &sigint_mask);
20     sigaction(SIGQUIT, &sa, &sigquit_mask);
21     sigaddset(&sa.sa_mask, SIGCHLD);
22     sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
23
24     if ((pid = fork()) == 0) { /* child code */
25         sigaction(SIGINT, &sigint_mask, (struct sigaction *)0);
26         sigaction(SIGQUIT, &sigquit_mask, (struct sigaction *)0);
27         sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
28         execl("/bin/sh", "sh", "-c", cmd, (char *)0);
29         _exit(127);
30     }
31
32     if (pid == -1) { /* fork() failed */
33         stat = -1; /* errno comes from fork() */
34     } else { /* parent code */
35         while (waitpid(pid, &stat, 0) == -1) {
36             if (errno != EINTR){
37                 stat = -1;
38                 break;
39             }
40         }
41     }
42
43     sigaction(SIGINT, &sigint_mask, (struct sigaction *)0);
44     sigaction(SIGQUIT, &sigquit_mask, (struct sigaction *)0);
45     sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
46
47     return(stat);
48 }
```

Listing 7.9: xcode/systemimpl2.c - Beispielimplementierung einer POSIX-konformen system(2)-Funktion

Es folgt eine alternative Implementierung, die nur etwas länger ist, aber im Grunde das gleiche erledigt wie die erste. Der wesentliche Unterschied ist, daß die zweite Variante portabel ist. Desweiteren führt sie mehr Prüfungen durch, beispielsweise ob Ergebnisse, Fehlercodes und andere Elemente in korrekten Bereichen liegen.

```

1 #include <errno.h>
2 #include <signal.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/wait.h>
7
8 #define SHELL_PATH  "/bin/sh"    /* path of the shell. */
9 #define SHELL_NAME   "sh"        /* name to give it. */
10
11 static struct sigaction intr, quit;
12 static int                  sa_refcnt;
13
14 /* execute LINE as a shell command, returning its status. */
15 static int do_system(const char *line) {
16     int      status, save;
17     pid_t    pid;
18     struct   sigaction sa;
19     struct   sigaction intr, quit;
20     sigset_t omask;
21
22     sa.sa_handler = SIG_IGN;
23     sa.sa_flags   = 0;
```

```

24     sigemptyset (&sa.sa_mask);
25
26     if (sigaction(SIGINT, &sa, &intr) < 0)
27         goto out;
28
29     if (sigaction(SIGQUIT, &sa, &quit) < 0) {
30         save = errno;
31         goto out_restore_sigint;
32     }
33
34     /* we reuse the bitmap in the 'sa' structure. */
35     sigaddset(&sa.sa_mask, SIGCHLD);
36     save = errno;
37
38     if (sigprocmask(SIG_BLOCK, &sa.sa_mask, &omask) < 0) {
39         if (errno == ENOSYS)
40             errno = save;
41         else {
42             save = errno;
43             (void)sigaction(SIGQUIT, &quit, (struct sigaction *)NULL);
44             out_restore_sigint:
45                 (void)sigaction(SIGINT, &intr, (struct sigaction *)NULL);
46                 errno = save;
47         }
48     out:
49         return -1;
50     }
51
52     pid = fork ();
53
54     if (pid == (pid_t)0) { /* child code. */
55         const char *new_argv[4];
56         new_argv[0] = SHELL_NAME;
57         new_argv[1] = "-c";
58         new_argv[2] = line;
59         new_argv[3] = NULL;
60
61         /* Restore the signals. */
62         sigaction(SIGINT, &intr, (struct sigaction *)NULL);
63         sigaction(SIGQUIT, &quit, (struct sigaction *)NULL);
64         sigprocmask(SIG_SETMASK, &omask, (sigset_t *)NULL);
65
66         /* Exec the shell. */
67         execve (SHELL_PATH, (char *const *)new_argv, environ);
68         _exit(127);
69     } else if (pid < (pid_t)0) { /* fork failed. */
70         status = -1;
71     } else { /* parent code */
72         pid_t child;
73
74         do {
75             child = wait (&status);
76
77             if (child <= -1 && errno != EINTR) {
78                 status = -1;
79                 break;
80             }
81         } while (child != pid);
82
83         if (waitpid(pid, &status, 0) != pid)
84             status = -1;
85     }

```

```

86     save = errno;
87
88     if ((( sigaction(SIGINT, &intr, (struct sigaction *)NULL)
89             | sigaction(SIGQUIT, &quit, (struct sigaction *)NULL)) != 0)
90         || sigprocmask(SIG_SETMASK, &omask, (sigset_t *)NULL) != 0)
91     status = -1;
92
93     return status;
94 }
95
96
97 int system(const char *line) {
98     if (line == NULL)
99         return do_system("exit 0") == 0;
100
101    int result = do_system(line);
102
103    return result;
104 }
```

Listing 7.10: `xcode/systemimpl.c` - Beispielimplementierung einer POSIX-konformen `system(2)`-Funktion



Das folgende Beispiel demonstriert den Aufruf der `system(2)`-Funktion.

Listing 7.11: Kommandos mit `system(2)` ausführen

Zuerst prüfen wir, ob ein Command Processor verfügbar ist, der das Kommando `who(1)` ausführen kann. Anschließend wird `system(2)` ausgeführt und der Rückgabewert gespeichert. Am Ende geben wir den Rückgabewert der Operation.

```

1 #include <sys/wait.h>
2 #include "header.h"
3
4 int main(void) {
5     int ret_val;
6
7     /* see if we have an interpreter */
8     if ((ret_val = system((char *)0)) > 0) {
9         printf("Command processor found...\nexecuting...\n");
10
11     if ((ret_val = system("who")) < 0)
12         err_fatal("system failed\n");
13 }
14
15 printf("retval: %d\n", ret_val);
16
17 return (0);
18 }
```

Listing 7.11: `xcode/system.c` - Kommandos mit `system(2)` ausführen.

```
% ./systemdemo
Command processor found...
executing...
steve :0 Feb 16 17:35 (console)
steve pts/0 Feb 16 17:36
steve pts/1 Feb 16 17:36
retval: 0
```



7.3 Prozesse beenden

Ebenso wichtig wie das Erzeugen neuer Prozesse ist auch deren kontrollierte Terminierung. In der Regel erfüllen Prozesse (ob Child oder nicht) ihre Aufgabe und beenden sich mit einem Rückgabewert (beispielsweise durch ein `return`-Statement). In einer Parent-Child-Beziehung ist es Aufgabe des Parent, sich erst zu beenden wenn die Child-Prozesse zurückgekehrt sind. Auf diese Weise vermeiden wir die Entstehung von Zombieprozessen (mehr zu diesem Thema finden Sie in Abschnitt 7.3.2).

POSIX.1 definiert in diesem Zusammenhang folgende Funktionen:

`exit` und `_exit`

sorgen für eine saubere Terminierung von Parent (`exit(2)`) und Child(s) (`_exit(2)`).

`wait` und `waitpid`

warten auf einen Kindsprozess (`wait`) und auf Kindsprozesse mit bestimmten PIDs (`waitpid`).

7.3.1 Die Funktionsfamilie `exit`

Wie angedeutet existieren verschiedene Funktionen zum Beenden von Prozessen, die sich alle in wichtigen Details unterscheiden. Neben `exit(3)` haben wir im vorangegangenen Beispiel `_exit(2)` zum Beenden von Kindsprozessen kennengelernt.

Gundsätzlich unterscheiden wir zwischen drei Arten der „planmäßigen“ und zwei Arten von „unplanmäßigen“ Prozessterminierungen:

1. Einfache, planmäßige Prozessterminierung:

- (a) Das Schlüsselwort `return` in `main` veranlaßt den Kernel, den aufrufenden Prozess ordnungsgemäß zu beenden und damit alle belegten Ressourcen wieder frei zu geben. Sie entspricht einem Aufruf von `exit(3)`.
- (b) Die Funktion `exit(3)` sorgt für eine ganze Reihe weiterer Systemaufrufe, die ihrerseits durch `atexit(2)` aufgerufen werden und dafür sorgen, daß alle offenen Streams geschlossen werden. Dieses Verhalten wird von ANSI C nicht vorausgesetzt.
- (c) Mit Hilfe der `_exit(2)`-Funktion schließen wir explizit den aufrufenden Kindsprozess. Diese Funktion wird für alle Kindsprozesse durch `exit` automatisch aufgerufen und ist in POSIX.1 definiert. In den meisten Unices ist `exit(2)` eine Bibliotheksfunktion und `_exit(2)` ein Systemaufruf.

2. Unplanmäßige Prozessterminierung:

- (a) Erhält ein Prozess ein spezielles Signal (`SIGABRT`), entweder durch einen anderen Prozess, den Kernel oder durch sich selbst (beispielsweise durch Aufruf von `abort(3)`), so wird der Prozess unmittelbar beendet.
- (b) Jeder Prozess kann die `abort(3)`-Funktion auch selbst aufrufen und sich das `SIGABRT`-Signal selbst übermitteln.

Unabhängig von der Art der Terminierung, im Kernel wird immer der gleiche Code aufgerufen. Er sorgt für die Freigabe der belegten Ressourcen, wie etwa Speicherbereiche, File Descriptors und anderen spezifischen Details.

Wird ein Prozess durch `exit(2)` oder `_exit(2)` beendet, übermittelt der terminierte Prozess über das Argument von `exit(2)` den Grund der Terminierung an den Elternprozess und wird als *Exit Status* bezeichnet. Anders sieht es bei einer unplanmäßigen Prozessterminierung aus. Hier übernimmt der Kernel die Übermittlung des *Terminierungsstatus*. Dadurch müssen Prozesse, die den Code abfragen möchten die beiden Funktionen `wait(2)` oder `waitpid(2)` aufrufen. Beachten Sie, daß der Exitstatus als Argument für `exit(2)` oder `_exit(2)` übergeben wird, der Terminierungsstatus aber der Rückgabewert von `exit(2)` oder `_exit(2)` darstellt.

Besondere Aufmerksamkeit verdient eine spezielle Situation in der der Elternprozess aus irgendwelchen Gründen von dem Kindsprozess terminiert wird. Was passiert anschließend mit den Kindsprozessen? Diese „verwaisten“ (*orphaned*) Prozesse werden zu Kindsprozessen von `init`, dem Superprozess. Man spricht auch von Vererbung: `init` erbt die Kinder seines Kindsprozesses. Normalerweise wird einfach nur der Prozess der neue Elternprozess, dessen Eltern am nächsten liegen. Oftmals ist es `init`, solange die Kindsprozesse nicht selbst Kindsprozesse erzeugt haben.

Eine andere Situation tritt ein, wenn der Kindsprozess vor dem Elternprozess terminiert. Das klingt im ersten Moment völlig normal, ist aber nicht ganz so trivial, wie es scheint. Sollte der Kindsprozess vollständig verschwinden, besteht für den Elternprozess keine Möglichkeit mehr, den Terminierungsstatus zu erfahren. Für solche Fälle führt der Kernel Statistiken über alle Prozesse. Somit ist er in der Lage, Anfragen von `wait(2)` und `waitpid(2)` zu erfüllen. Unter UNIX werden Elterprozesse, die noch nicht auf ihre Kinder gewartet haben als *Zombies* bezeichnet.

Sollte der von `init` geerbte Prozess beendet werden, so wird `init` natürlich kein Zombie, sondern ruft automatisch `wait(2)` auf, um die PID des betreffenden Prozesses zu erfahren und verhindert so ein „verschmutzen“ des Systems durch Zombies.

7.3.2 Die Funktionen `wait` und `waitpid`

Alle Elternprozesse, deren Kinder entweder normal oder außerplanmäßig terminiert werden, erreicht ein `SIGCHLD`-Signal. Dabei handelt es sich um einen asynchronen Vorgang, da das Signal auftauchen kann, wenn der Elternprozess noch läuft. Der Empfänger des Signals kann selbst entscheiden, ob er es ignorieren, oder darauf reagieren möchte. Ist letzteres der Fall, wird eine Funktion definiert, die beim Eintreffen des Signals ausgeführt wird. Standardmäßig werden Singale ignoriert. Die Signalverarbeitung wird im Kapitel 8 *Signalbehandlung*.

Ein Prozess, der `wait(2)` oder `waitpid(2)` aufruft kann in den Zustand *Blocking* übergehen, solange die Kindsprozesse noch laufen. Andererseits könnte er auch sofort zurückkehren und den Terminierungsstatus des Kindes übermitteln.

Unter UNIX kehrt ein Prozess, der auf ein `SIGCHLD`-Signal reagiert sofort zurück. Wird `wait(2)` jedoch an beliebiger Stelle im Programmablauf aufgerufen, so geht der Prozess in den Zustand *Blocking* über.

Die beiden Funktion weisen folgende Synopsis auf:

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Rückgabewert: Child-PID oder -1 bei Fehler.

`status`

Zeiger auf einen Integer, in dem der Terminierungsstatus gespeichert wird.

`pid`

PID des Prozesses auf den gewartet werden soll.

`options`

Zeigt an, wie mit Terminierungen der Kindsprozesse umgegangen werden soll.

Während `wait(2)` blockiert, kann `waitpid(2)` durch Angabe der Optionen `WNOHANG` und/oder `WUNTRACED` davon abgehalten werden. Des Weiteren kehrt `wait(2)` zurück, sobald einer der Kindsprozesse terminiert wird, während `waitpid(2)` auf einen ganz bestimmten Prozess wartet. Ist ein Prozess bereits ein Zombie, so kehrt `wait(2)` sofort mit dem Terminierungsstatus des Prozesses zurück, andernfalls blockiert `wait(2)`.

Listing 7.12: Einsatz von `wait(2)`

```

1  #include "header.h"
2
3  int main(int argc, char **argv) {
4      pid_t pid;
5      int    status;
6
7      if ((pid = fork()) < 0)
8          err_fatal("fork() failed.");
9
10     if (pid == 0) /* child code */
11         printf("Child with PID %ld\n", (long)getpid());
12
13     if (pid > 0) { /* parent */
14         printf("Parent with PID %ld\n", (long)getpid());
15
16         if (wait(&status) != pid)
17             err_normal("wait() interrupted");
18         else
19             printf("status: %d\n", status);
20     }
21
22     return (0);
23 }
```

Listing 7.12: xcode/wait.c - Einsatz von wait(2).

Wir gehen wie üblich vor: erzeugen einen Child-Prozess, prüfen die PID und wenn das Child zurückkehrt, geben wir eine kleine Meldung aus. Kehrt `fork(2)` aber im Parent zurück, warten wir hier auf unser Child. Der Aufruf von `wait(2)` blockiert so lange, bis das Child zurückkehrt. Wenn der Aufruf wieder erwarten nicht die PID des Child zurückgibt, wurde der Systemaufruf von einem Signal unterbrochen. Mehr Informationen über Signale und Systemaufrufe erfahren Sie in Abschnitt 8.2 ab Seite 200. □

Der Parameter `options` ist eine Kombination logischer OR-Verknüpfungen folgender Konstanten:

WCONTINUED

Zeige den Status für den in `pid` fortgesetzten Kindsprozess an, sofern sein Status nicht mehr seit der letzten Fortsetzung abgefragt wurde. Optional in POSIX.1.

WNOHANG

Kehre sofort zurück, wenn keiner der Kindsprozesse beendet wurde.

WUNTRACED

Kehre sofort zurück, wenn für die Kindsprozesse kein Status gemeldet wurde.

POSIX.1 definiert nur die beiden Makros `WNOHANG` und `WUNTRACED`. Die meisten Unices und UNIX-ähnliche Systeme kennen unter Umständen ein oder zwei weitere. So hat `WCONTINUED`, was POSIX.1 als optional gekennzeichnet hat, erst mit Kernel 2.6.10 den Weg in Linux gefunden und Solaris kennt darüber hinaus auch noch `WNOWAIT`, was den betroffenen Prozess veranlaßt in einem Zustand zu verweilen, der es möglich macht, `wait(2)` oder `waitpid(2)` erneut mit dem gleichen Resultat aufzurufen.

Der in `status` gespeicherte Rückgabewert ist eine Bitmaske, die den genauen Status definiert. Mit Hilfe diverser Makros können wir diese Maske entschlüsseln. Wie auch die Funktion `wait(2)` sind die Makros in `<sys/wait.h>` definiert:

WIFEXIT(status)

Expandiert zu `wahr`, wenn der Kindsprozess „normal“ terminiert wurde. In diesem Fall werden die niederwertigen 8 Bits abgefragt, die durch den Kindsprozess an `exit` oder `_exit` übergeben werden.

WIFSIGNALED(status)

Expandiert zu `wahr`, wenn der Kindsprozess nicht „normal“ terminiert wurde. In diesem Fall kann das Makro `WTERMSIG(status)` ausgeführt werden, um das Signal, das zur unplanmäßigen Terminierung geführt hat, abzufragen.

WIFSTOPPED(status)

Expandiert zu `wahr` für den Kindsprozess, der gerade beendet wurde. Mit `WSTOPSIG(status)` fragen wir die Signalnummer ab, die zur Beendigung des Kindprozesses geführt hat.

Viele Systeme definieren oftmals mindestens ein weiteres Makro: `WCOREDUMP(status)`. Es gibt `true` zurück, wenn `WIFSIGNALED` ebenfalls `true` ist und durch das Signal ein Core Dump erzeugt wurde. Mit POSIX-konformen Shells können Core Dumps mittels `ulimit -c 0` deaktiviert und mit `ulimit -c 1` aktiviert werden. Die maximale Dateigröße von Core Dumps kontrollieren Sie mit `limit coredumpsize 10240`, was eine Dateigröße von maximal 10 MB zulässt.

Mit den Makros können wir Listing 7.12 leicht verbessern, um mehr über den Grund der Terminierung des Child zu erfahren (Listing 7.13).

Listing 7.13: Einsatz der WIF-Makros.

```

1 #include <sys/wait.h> /* for WIF macros */
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     pid_t pid;
6     int    status;
7
8     if ((pid = fork()) < 0)
9         err_fatal("fork() failed.");
10
11    if (pid == 0) /* child code */
12        printf("Child with PID %ld\n", (long)getpid());
13
14    if (pid > 0) { /* parent */
15        printf("Parent with PID %ld\n", (long)getpid());
16
17        if (wait(&status) != pid)
18            err_normal("wait() interrupted");
19
20        if (WIFEXITED(status))
21            printf("Child with PID %ld terminated normally. Status: %d\n",
22                   (long)pid, WEXITSTATUS(status));
23        else if (WIFSIGNALED(status))
24            printf("Child with PID %ld terminated by signal. Status: %d\n",
25                   (long)pid);
26    }
27
28    return (0);
29 }
```

Listing 7.13: xcode/wait2.c - Einsatz der WIF-Makros.



Bei der Verwaltung mehrerer Childs ist `wait(2)` unpassend, da wir nicht festlegen können, auf welchen Prozess wir warten möchten. Die Funktion `waitpid(2)` ist dazu besser geeignet.

Listing 7.14: Einsatz von waitpid(2)

Das folende Beispiel illustriert den Einsatz der Makros anhand von `WNOHANG`.

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include "header.h"
4
5 /* prototype */
6 void gen_random(void);
7
8 int main(void) {
9     pid_t pid;
10    int status, ret_val;
11
12    /* create a child process */
13    if ((pid = fork()) < 0)
14        err_fatal("fork failed\n");
15    else if (pid == 0) /* child code */
16        gen_random();
17
18    /* loop until child generated a random number and returns */
19    while(1) {
20        if (waitpid(pid, &status, WNOHANG) > 0) {
21            /* fetch value passed to exit */
22            ret_val = WEXITSTATUS(status);
23            printf("Random number generated by child: %d\n", ret_val);
24            exit(1);
25        }
26    }
27 }
28
29 void gen_random(void) {
30     time_t t; /* random seed */
31     srand((unsigned)time(&t));
32     unsigned int rand_num = random() % 100;
33     exit(rand_num); /* passing random number to parent */
34 }
```

Listing 7.14: xcode/waitpid.c - So funktioniert WNOHANG mit waitpid(2)

□

Betrachten wir `waitpid(2)` etwas genauer. Je nachdem welchen Wert das Argument für `pid` aufweist, wird `waitpid(2)` unterschiedlich interpretiert:

`pid == -1`

Warte auf alle Kindsprozesse.

`pid > 0`

Warte auf Prozess mit der ID `pid`.

`pid == 0`

Warte auf alle Kindsprozesse, deren Gruppen-PID dem Wert von `pid` entspricht.

`pid < -1`

Warte auf alle Kindsprozesse, deren Gruppen-PID dem absoluten Wert von `pid` entspricht.

Neben der Tatsache, daß `waitpid(2)` neben der PID des jeweiligen Prozesses auch den Terminierungsstatus liefert, sind folgende Unterschiede zwischen beiden Funktionen `wait(2)` und `waitpid(2)` relevant:

1. `waitpid(2)` erlaubt uns auf einen bestimmten Prozess zu warten, während `wait(2)` bei jedem terminierten Prozess zurückkehrt.

2. Mit `waitpid(2)` erhalten wir eine Variante von `wait(2)`, die den Aufrufer nicht in den Zustand *Blocking* versetzt.
3. `waitpid(2)` unterstützt Job control (siehe WTRUNCATED).

7.4 Benutzer- und Gruppen-IDs von Prozessen ändern

Wir haben in Abschnitt 6.1 bereits über Benutzer- und Gruppen-IDs von Prozessen gesprochen. Natürlich können wir nicht nur die jeweiligen IDs abfragen, sondern auch setzen. POSIX definiert folgende Funktionen zum setzen der Gruppen- und Benutzer-IDs.

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

uid

Benutzer-ID, die für den aufrufenden Prozess gesetzt werden soll.

gid

Gruppen-ID, die für den aufrufenden Prozess gesetzt werden soll.

Wenn der Prozess über entsprechende Rechte verfügt, setzt `setuid`/`setgid` sowohl die reelle Benutzer- als auch die effektive Benutzer-/Gruppen-ID, sowie das gespeicherte set-user-ID-Bit (SUID) des aufrufenden Prozesses. Hat der Prozess hingegen nicht die erforderlichen Rechte, `uid` entspricht aber der reellen Benutzer-/Gruppen-ID (oder das Set-User-ID-Bit ist gesetzt), wird die effektive Benutzer-/Gruppen-ID auf `uid` gesetzt. Die beiden anderen IDs und die zusätzlichen Gruppen-IDs bleiben unverändert.

Was ist der Unterschied zwischen den verschiedenen IDs?

Die reelle User ID (RUID) ist immer genau die mit der der Prozess erzeugt wurde. Hat Benutzer *graegerts* den Prozess gestartet, ist die RUID *graegerts*. Nur wenn diese ID 0 ist, kann der Prozess eine andere reelle User ID annehmen. Die effektive User ID (EUID) ist immer die der Prozess momentan verkörpert und wird herangezogen, um Zugriffsrechte von Ressourcen zu prüfen. Die effektive User ID (EUID) kann in die RUID verwandelt werden, wenn die EUID nicht 0 ist. Ist sie 0 kann die EUID quasi jede sein. Wenn das gestartete Process Image File das SUID-Bit gesetzt hat, wird das Image File mit der UID des Besitzers ausgeführt, andernfalls mit der RUID.

Während normale Programme wie `ls(1)` oder `cat(1)`, keine besonderen Rechte benötigen und die EUID in der Regel der RUID entspricht, gibt es Programme, die auch normalen Benutzern Zugriff auf geschützte Bereiche erlauben soll, wie beispielsweise das Programm `passwd(1)`. Es schreibt in die Datei `/etc/passwd`, die nur der Superuser schreiben kann. Ruft User *graegerts* nun `passwd(1)` auf startet es mit folgender Konfiguration: RUID = *graegerts*, EUID = *graegerts*, SUID = root. Nun ruft `passwd(1)` die Funktion `seteuid(0)` auf und zeigt mit 0 an, daß es die EUID auf 0 ändern möchte. Das geht nur, weil die SUID von `passwd(1)` selbst auch 0 ist. Die Einstellungen lauten nun: RUID = *graegerts*, EUID = root, SUID = root. Nur mit einer EUID von 0 kann `passwd(1)` in `/etc/passwd` schreiben. Übrigens treffen die Ausführungen auch auf Gruppen-IDs zu.

Nach diesen Ausführungen sollte klar sein, daß SUID-Programme ein Sicherheitsrisiko darstellen und außerordentlich gut entworfen sowie nur mit Bedacht eingesetzt werden sollten.

POSIX definiert zwei Funktionen zum Setzen der reellen und effektiven Benutzer- und Gruppen-IDs.

```
#include <sys/types.h>
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

ruid

Reelle Benutzer-ID, die für den aufrufenden Prozess gesteckt werden soll.

euid

Effektive Benutzer-ID, die für den aufrufenden Prozess gesteckt werden soll.

rgid

Reelle Gruppen-ID, die für den aufrufenden Prozess gesteckt werden soll.

egid

Effektive Benutzer-ID, die für den aufrufenden Prozess gesteckt werden soll.

Die Funktion **setreuid(2)/setregid(2)** setzt die reelle und effektive Gruppen-ID des aufrufenden Prozesses auf **ruid** und **euid**. Wenn **rgid/ruid -1** ist, wird die reelle Gruppen-/Benutzer-ID nicht geändert. Es darf nur ein Prozess mit ausreichenden Rechten, Änderungen der IDs vornehmen.

Schließlich hält POSIX noch zwei Funktionen zur Änderung der effektiven Benutzer- oder Gruppen-ID bereit:

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t uid);
int setegid(gid_t gid);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

uid

Effektive Benutzer-ID, die für den aufrufenden Prozess gesetzt werden soll.

gid

Effektive Gruppen-ID, die für den aufrufenden Prozess gesetzt werden soll.

Unprivilegierte Prozesse können ihre effektive Benutzer-/Gruppen-ID entweder auf die reelle Benutzer-/Gruppen-ID setzen oder auf die gespeicherte set-user-ID. Privilegierte Prozesse setzen nur ihre effektive Benutzer-/Gruppen-ID auf **uid/gid**.

7.5 Prozessgruppen

Zusätzlich zu den besprochenen IDs ist jedem Prozess auch eine Prozessgruppe zugeordnet. Eine Prozessgruppe enthält einen oder mehrere Prozesse und ihr ist eine eindeutige ID zugeordnet, die in einer Variable vom Typ **pid_t** abgelegt werden kann.

POSIX definiert zwei Funktionen zum Abfragen und Setzen von Prozessgruppen-IDs:

```
#include <unistd.h>

pid_t getpgrp(void);
```

Rückgabewert: Prozessgruppen-ID des aufrufenden Prozesses.

```
int setpgid(pid_t pid, pid_t pgid);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler.

pid

ID des Prozesses, dessen Prozessgruppe geändert werden soll.

gpid

Prozessgruppen-ID, auf die der in pid angegebene Prozess gesetzt werden soll.

Prozessgruppen sind für die Signaldistribution und Terminals von Bedeutung. Prozesse, die der gleichen Prozessgruppe wie das Terminal angehören, werden in den Vordergrund geholt und dürfen Signale lesen, während alle anderen in den Zustand *Blocking* übergehen, wenn sie versuchen zu lesen. Diese Funktionen werden meist von *Command Processors* wie `csh(1)` für die Prozesskontrolle genutzt. Die Aufrufe `TIOCGPGRP` und `TIOCSPGRP` beispielsweise werden benutzt, um die Prozessgruppe des Kontrollterminals zu ändern, wie in `termios(3)` beschrieben.

Sollte eine Prozessgruppe durch Beendigung eines Kindsprozesses verwaisen und ein Mitglied einer gerade verwaisten Prozessgruppe angehalten werden, wird das Signal `SIGHUP` (*hangup*), gefolgt von `SIGCONT` (*continue*) zu jedem Prozess der verwaisten Prozessgruppe gesendet.

7.6 Sitzungen

Eine oder mehrere Prozessgruppen werden zu Sitzungen (*sessions*) zusammengefaßt. Meist faßt die Shell Prozesse zu Gruppen zusammen, wenn sie beispielsweise durch *Pipes* miteinander verknüpft werden. Abbildung 7.6 illustriert diese Tatsache.

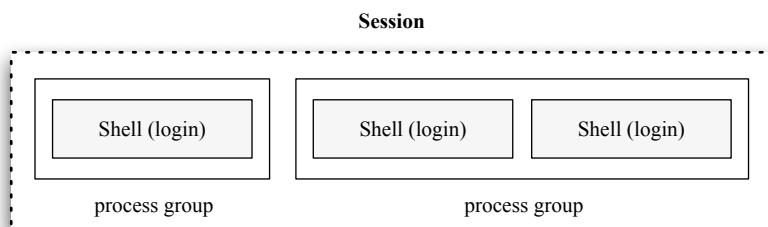


Abbildung 7.6: Prozessgruppen und Sitzungen: Die Verkettung `process1 / process2` führt zu einer neuen Prozessgruppe innerhalb der Sitzung.

Mit der POSIX-Funktion `setsid` erzeugen wir eine neue Session:

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

Rückgabewerte: ID der Prozessgruppe, oder -1 bei Fehler.

Ist der aufrufende Prozess kein Gruppenleiter, wird eine neue Session erzeugt. Dabei kann folgendes passieren:

- Der Aufrufer wird zum Sitzungsleiter (*session leader*) der neuen Sitzung. Dabei ist der Sitzungsleiter auch immer gleichzeitig der Erzeuger der aktuellen Sitzung und der einzige Prozess derselben.
- Der Aufrufer wird zum Leiter der Prozessgruppe (*process group leader*). Die ID der Prozessgruppe entspricht der Prozess-ID der Gruppenleiters.
- Dem Prozess ist kein Kontrollterminal zugeordnet. Wäre das vor dem Aufruf von `setsid` der Fall, würde diese Verbindung zuvor gelöst werden. Das ist wichtig, denn beispielsweise sollen Serverprogramme (*daemons*) keine Signale von dem Prozess erhalten, der sie gestartet hat. Da Child-Prozesse auch das Controlling Terminal vom Parent erben, rufen Daemons gleich zu Beginn `setsid(2)` auf, um diese Verbindung zum Parent zu lösen.

Die Funktion schlägt fehl, wenn der aufrufende Prozess bereits ein Prozessgruppenleiter ist. Um sicher zu stellen, daß es nicht der Fall ist, können Sie durch `fork(2)` einen Kindsprozess erzeugen und den Elternprozess beenden, denn das Kind wird die gewünschte Arbeit erledigen. Der Kindsprozess kann kein Prozessgruppenleiter sein, da die Prozessgruppen-ID von dem Elternprozess geerbt wird, aber die Prozess-ID des Kindes eine völlig neue ist.

Der Begriff Session ID ist verwirrend, da es keine solche ID gibt. Tatsächlich ist die Session ID immer die PID des Session Leaders. So gibt `getsid` nur die Prozessgruppen-ID des Session Leaders zurück.

Überdies fassen wir weitere wichtige Eigenschaften von Sitzungen und Prozessgruppen zusammen:

- Jede Sitzung *kann* mit einem *Kontrollterminal* verknüpft sein. Normalweise ist es das Gerät oder das PTY (*pseudo terminal*), das die Login-Sitzung gestartet hat. Ist ein Sitzungsleiter mit einem Kontrollterminal verknüpft, so wird er als kontrollierender Prozess (*controlling process*) bezeichnet.
- Prozessgruppen einer Sitzung können in zwei verschiedene Gruppen eingeteilt werden: Vordergrund- und Hintergrundprozesse. Ist eine Sitzung mit einem Kontrollterminal verknüpft, so verfügt sie über genau *einen* Vordergrundprozess, während alle anderen der Gruppe im Hintergrund operieren, auch der möglicherweise kontrollierende Prozess.
- Prozesse, die sich im Vordergrund befinden, können durch Eingabe einer Unterbrechung (z.B. CTRL-C oder CTRL-\) beendet werden. Dadurch wird allen Prozessen dieser Gruppe eine Unterbrechungsanforderung gesendet. Sollte das Terminal über ein Modem verbunden sein, wird der Sitzung das Signal `SIGHUP` (*hangup*) gesendet.

POSIX.1 überläßt die Anforderung an Kontrollterminals den Implementierungen.

Mit `getsid(2)` fragen wir die ID des Session Leaders ab:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Rückgabewerte: ID des Session Leaders, oder -1 bei Fehler.

`pid`

ID des Prozesses von dem wir wissen möchten, welche ID der Session Leader seiner Sitzung aufweist.

Um die ID des Session Leaders des Aufrufers abzufragen, setzen wir `pid` auf 0.

7.7 Terminals

In den meisten Shells, die *Job Control* implementieren, dürfen Prozesse von dem kontrollierenden Terminal aus in den Hintergrund und wieder in den Vordergrund geholt werden. Damit der Getätetreibers des Terminals weiß wohin die Daten gesendet werden müssen, wurden die beiden Funktionen `tcgetpgrp(2)` und `tcsetpgrp(2)` eingeführt. Sie werden beide Funktionen nur selten direkt aufrufen, es sei denn Sie entwickeln eine Shell oder terminalabhängige Software.

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp(int filedes);
```

Rückgabewerte: Gruppen-ID der Prozessgruppe für Vordergrundprozesse, -1 bei Fehler.

```
int tcsetpgrp(int filedes, pid_t pgid_id);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

filedes

Filedescriptor auf den, die Operation angewendet werden soll.

pgid_id

Gruppen-ID für Vordergrundprozesse.

Beide Funktionen sind nur erfolgreich, wenn `_POSIX_JOB_CONTROL` definiert ist.

7.8 Benutzeridentifizierung

Wir können zwar mit den Funktionen `getpwuid(2)` und `getuid(2)` herausfinden, welcher Benutzernname mit einem Prozess verbunden ist, doch was wenn ein Benutzer mehrere Login-Namen mit gleichen Benutzer-IDs besitzt? Die POSIX-Funktion `getlogin(2)` gibt einen Zeiger auf einen String, der den Login-Namen des Benutzers, welcher mit den Login-Aktivitäten des Kontrollterminals verbunden ist, enthält. Wird kein Nullzeiger zurückgegeben, enthält der referenzierte String den aktuellen Login-Namen, auch wenn mehrere Namen mit der gleichen Benutzer-ID assiziert sind.

Da `getlogin(2)` nicht wiedereintrittsfähig sein muß, ist Thread-Sicherheit nicht erforderlich.

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

```
char *getlogin(void);
```

Rückgabewerte: Zeiger auf den Login-Namen bei Erfolg, Nullzeiger bei Fehler.

Die Funktion wird nicht das gewünschte Ergebnis liefern, wenn der Prozess nicht mit einem Terminal verbunden ist, wie es bei Daemons (Hintergrundprozessen) der Fall ist.

Listing 7.15: Informationen über den Benutzerprozess mit `getlogin(2)` abfragen

```
1 #include <sys/types.h>
2 #include <pwd.h>
3 #include <unistd.h>
4 #include "header.h"
5
6 int main(void) {
7     char          *log; /* login name */
8     struct passwd *pw;  /* user data */
9
10    if ((log = getlogin()) == NULL || (pw = getpwnam(log)) == NULL)
11        err_fatal("Could not fetch user information.\n");
12    else
13        printf("Current user name (PID: %d): %s\n", getpid(), log);
14
15    return(0);
16 }
```

Listing 7.15: xcode/getlogin.c - Informationen über den Benutzerprozess mit `getlogin(2)` abfragen.

Bildschirmausgabe:

```
% ./getlogindemo
Current user name (PID: 19586): steve
```



7.9 Laufzeitmessungen

Wenn wir von Zeitmessungen sprechen, sind die drei unterschiedlichen messbaren Zeiten gemeint: Uhrzeit (*clock time*), Benutzerzeit (*user time*) und Prozesszeit (*process time*). Die Funktion `times` kann benutzt werden, um diese drei Größen abzufragen:

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Rückgabewert: Zeit in Millisekunden, bei Fehler sollte errno abgefragt werden.

`buf`

Zeiger auf eine Struktur in der die ermittelten Zeiten gespeichert werden sollen.

Die Struktur `tms` enthält folgende Elemente:

```
struct tms {
    clock_t tms_utime; /* user time */
    clock_t tms_stime; /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};
```

Die Zeiten terminierter Kindsprozesse werden in `tms_cutime` und `tms_cstime` des Elternprozess, sobald `wait(2)` oder `waitpid(2)` (siehe Abschnitt 7.3.2) die Prozess-ID des terminierten Prozesses liefern. Sollte ein Kindsprozess nicht auf seine Kinder gewartet haben, werden deren Zeiten nicht einbezogen.

`tms_utime`

Speichert die CPU-Zeit, die für die Ausführung von Instruktionen des aufrufenden Prozesses benötigt wurde (*user time*).

`tms_stime`

Speichert die CPU-Zeit, die für die Ausführung von Systeminstruktionen des aufrufenden Prozesses benötigt wurde (*system time*).

`tms_cutime`

Speichert die Summe von `tms_utime` und `tms_cutime` der Kindsprozesse (*child user time*).

`tms_cstime`

Speichert die Summe von `tms_stime` und `tms_cstime` der Kindsprozesse (*child system time*).

Eine sehr einfache, übersichtliche Variante zur Demonstration von `times(2)` finden wir in Listing 7.16.

Listing 7.16: Zeitmessungen mit `times(2)`

```
1 #include <limits.h>
2 #include <math.h>
3 #include <unistd.h>
4 #include <sys/times.h>
5 #include "header.h"
6
7 static pid_t pid;
8
9 static void exit_handler(void) {
10     double clockticks;
11     struct tms t;
```

```

13     if ((clockticks = (double)sysconf(_SC_CLK_TCK)) < 0)
14         err_fatal("sysconf() failed for _SC_CLK_TCK");
15
16     if (times(&t) < (clock_t)0)
17         err_fatal("times() failed");
18
19     printf("%s CODE:\n", (pid == 0 ? "CHILD" : "PARENT"));
20
21     printf("      User time: %8.3fs\n", t.tms_utime / clockticks);
22     printf("      System time: %8.3fs\n", t.tms_stime / clockticks);
23     printf(" Child user time: %8.3fs\n", t.tms_cutime / clockticks);
24     printf("Child system time: %8.3fs\n", t.tms_cstime / clockticks);
25 }
26
27 int main(void) {
28     int i, status;
29     float f;
30
31     /* install exit handler */
32     if (atexit(exit_handler))
33         err_fatal("atexit failed");
34
35     if ((pid = fork()) < 0)
36         err_fatal("fork failed");
37     else if (pid == 0) /* child code */
38         for (i = 1; i < 10000000; i++)
39             f = fmod(i, i * M_PI);
40     else
41         while (1)
42             if (waitpid(pid, &status, WNOHANG) > 0)
43                 exit(0);
44
45     return (0);
46 }
```

Listing 7.16: xcode/times.c - Zeitmessungen mit times(2)

Das Beispiel führte auf einer Alpha XP1000 (EV6) unter Tru64 UNIX zu folgender Ausgabe:

```
% ./times
CHILD CODE:
      User time: 0.150s
      System time: 0.000s
    Child user time: 0.000s
  Child system time: 0.000s
PARENT CODE:
      User time: 0.180s
      System time: 0.700s
    Child user time: 0.150s
  Child system time: 0.000s
```



7.10 Hintergrundprozesse und Daemons

Die Shell ist ein interaktiver Kommandoprozessor, der es uns ermöglicht mit dem Betriebssystem und seinen Einrichtungen zu interagieren. Sie liest Kommandos ein, führt sie aus, informiert uns über deren Status usw. Wir unterscheiden stets zwischen zwei Prozessvarianten: Vordergrund- und Hintergrundprozesse.

Vordergrundprozesse können wir mit der Tastenkombination CTRL-C beenden. Das funktioniert mit Hintergrundprozessen nicht, denn die Eingaben an der Konsole werden nicht an den Hintergrundprozess weitergeleitet (solange der zugehörigen Session kein Controlling Terminal zugewiesen wurde). Prozesse, die nur im Hintergrund laufen, werden auch als Daemons (Dämonen) bezeichnet. Beispiele für Daemons sind *named(8)* zur Auflösung von Domain-Namen und IP-Adressen oder *sshd(8)*, der Serverprozess für den sicheren Remote-Login. Ein Daemon wird erzeugt und läuft ohne Bezug zu anderen Prozessen im Hintergrund. Wir können also hinterher nicht mehr sagen, wer den Daemon erzeugt hat.

In diesem Kapitel werden wir lernen, Daemons zu entwickeln und befassen uns mit den Besonderheiten solcher Prozesse.

7.10.1 Hintergrundprozesse

Beginnen wir direkt mit der Besprechung eines kleinen Programms, das uns den Einstieg in die Materie erleichtern soll. Listing 7.17 zeigt, wie ein Prozess im Hintergrund ausgeführt wird und dann natürlich nicht mehr mit CTRL-C terminiert werden kann.

Listing 7.17: Erstellung eines Hintergrundprozesses.

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include "header.h"
4
5 int main(int argc, char *argv[])
6 {
7     pid_t pid;
8     char *basename = basename_ex(argv[0]);
9     char **argvp, *delimiter = " ";
10
11    if (argc != 2)
12        err_fatal("Usage: %s <string>\n", basename);
13
14    if ((pid = fork()) < 0)
15        err_fatal("fork() failed");
16    else if (pid == 0) { /* child code */
17        if (getpid() != getsid(0)) /* are we a session leader? */
18            if (setsid() < 0) /* become a session leader */
19                err_fatal("setsid() failed");
20
21        if (create_argv(argv[1], delimiter, &argvp) < 0)
22            err_fatal("create_argv() failed");
23
24        if (execvp(argvp[0], &argvp[0]) < 0)
25            err_fatal("execvp failed");
26    }
27
28    exit(0);
29 }
```

Listing 7.17: xcode/spinoff.c - Erstellung eines Hintergrundprozesses.

Nach einem **fork** prüfen wir, ob wir (der Child-Prozess) Session Leader ist, und wenn nicht, dann bemühen wir **setsid(2)**. Ein Session Leader hat kein Controlling Terminal, und kann daher nicht mit CTRL-C abgebrochen werden. Anschließend erzeugen wir mit Hilfe der Funktion **create_argv** (beschrieben in Listing D.5) ein Array bestehend aus Zeigern auf das auszuführende Kommando und seiner Parameter, das wir an **execvp(3)** übergeben.

Mit diesem Tool können wir das Kommando **ls -l** im Hintergrund ausführen, als ob wir es direkt in den Hintergrund geschickt hätten.

```
% ./spinoff "ls -l"
total 423
-rwxr-xr-x    1 graegerts users   1266 Nov  4 08:06 abortimpl.c
-rwxr-xr-x    1 graegerts users    868 Nov  4 08:06 changedir.c
-rwxr-xr-x    1 graegerts users    295 Nov  4 08:06 changekey.c
...
-rwxr-xr-x    1 graegerts users   912 Oct 15 05:55 utmp.c
-rwxr-xr-x    1 graegerts users   730 Nov  4 08:06 vfork.c
```

□

Daemons haben die Eigenschaft, nur einmal gestartet zu werden und so lange im Hintergrund zu laufen, bis das System heruntergefahren oder neu gestartet werden muß. Die ganze Hauptarbeit wird in einer Endlosschleife erledigt, wie wir in einem kleinen *biff*-Clone aus Listing 7.18 sehen können.

Listing 7.18: Beispielanwendung für einen Daemon.

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include "header.h"
5
6 #define MAIL_PATH "/var/spool/mail/"
7
8 int main(int argc, char **argv) {
9     pid_t          pid;
10    char          *basename = basename_ex(argv[0]);
11    char          mailbox[256], *user;
12    int           fd;
13    size_t         mbox_size = 0, cur_mbox_size;
14    struct stat   stat_buf;
15
16    if (argc != 2) {
17        err_fatal("Usage: %s <user>\n", basename);
18    } else {
19        strcpy(mailbox, MAIL_PATH);
20        user = argv[1];
21        strcat(mailbox, user);
22    }
23
24    printf("mailbox: %s\n", mailbox);
25
26    while (1) {
27        if (stat(mailbox, &stat_buf) < 0)
28            err_fatal("lstat() failed for %s", mailbox);
29
30        cur_mbox_size = (size_t)stat_buf.st_size;
31
32        if (cur_mbox_size > mbox_size) {
33            if (mbox_size != 0)
34                printf("You have new mail, %s%s!\n", argv[1], "\007");
35
36            mbox_size = cur_mbox_size;
37        } else if (cur_mbox_size < mbox_size)
38            mbox_size = cur_mbox_size;
39
40        sleep(5);
41    }
42
43    close(fd);
44    exit(0);
```

45 }

Listing 7.18: xcode/smallbiff.c - Beispielanwendung für einen Daemon

Das Prinzip ist einfach: Beim Start des Programms wird `smallbiff` ein Benutzername übergeben, der dem Mailbox-Pfad (`/var/spool/mail`) angefügt wird, so daß ein vollständiger Pfad zu Mailbox-Datei des Benutzers entsteht. Anschließend treten wir in die Endlosschleife ein, prüfen die Größe der Mailbox und gehen 5 Sekunden schlafen. Bei jedem Durchlauf vergleichen wir die Dateigröße erneut und geben bei Änderung einen Klang und eine Meldung aus.

```
% ./spinoff "./smallbiff graegerts"
mailbox: /var/spool/mail/graegeerts
% mail -s Bifftest graegerts
Hello. Hope your biff clone is running. Regards.
EOT                                // CTRL-D
% You have new mail, graegerts! // nach 5 Sekunden
%
```

□

Listing 7.18 zeigt, wie einfach Daemons gestrickt sein können. Allerdings fallen uns schnell einige Nachteile dieses Programms auf. Die Post der Benutzer eines Systems wird in der Regel im Verzeichnis `/var/spool/mail` abgelegt. Dieses Verzeichnis enthält eine Datei, die alle ungelesenen Nachrichten vorhält.

Befindet sich das Mail-Verzeichnis an einem anderen Ort, schlägt das Programm fehl. Allerdings gibt es eine standardkonforme Möglichkeit, dieses Verzeichnis programmatisch abzufragen. Des Weiteren findet keine Prüfung statt, ob der Benutzername gültig ist und mit den Rechten wird es auch nicht so genau genommen, denn es wird nicht geprüft, ob der Aufrufer berechtigt ist, die Mailbox anderer Benutzer zu beobachten (die Post lesen, können unberechtigte Benutzer nicht, da ein Aufruf von `open(2)` den Fehler *Permission denied* zurückliefern würde, aber es geht auch niemanden etwas an, wer wie viel Post bekommt).

Normalerweise würde es genügen, wenn `smallbiff` einfach nur die Existenz der Mailbox-Datei prüft. Ist sie vorhanden, befinden sich ungelesene Nachrichten im Postfach des Benutzers. Ich habe die Funktionalität erweitert und prüfe auch die Größenänderung der Datei, damit wir bei jeder neuen Nachricht eine Meldung erhalten.

Der POSIX.1-Standard spezifiziert die folgenden Umgebungsvariablen, die für die Behandlung von Benutzernachrichten relevant sind: `MAIL`, `MAILDIR`, `MAILPATH` und `MAILCHECK`. Die wichtigste Variable ist `MAIL`, denn sie enthält den vollständigen Pfad zur Mailbox des Benutzers. `smallbiff` sollte also eine der genannten Variablen verwenden, um die richtige Mailbox zu finden. Im nächsten Abschnitt werden wir eine verbesserte Variante von `smallbiff` kennenlernen.

7.10.2 Die Prozessumgebung

Ein gutes Programm versucht so anpassungsfähig wie möglich zu sein. Es gibt unterschiedliche Möglichkeiten, das zu erreichen, doch die einfachste und am weitesten verbreitete setzt auf *Umgebungsvariablen*. Sie enthalten system- und benutzerspezifische Informationen, die zahlreiche Funktionen haben. Tabelle 7.1 listet ein paar der wichtigsten Variablen auf.

Wenn Sie wissen möchten, welche Umgebungsvariablen momentan gesetzt sind, können sie den Befehl `printenv(1)` ausführen. Um zu wissen, ob eine bestimmte Variable gesetzt ist, lohnt sich eine Filterung der Ausgabe mit `grep(1)`, etwa so: `printenv | grep MAILDIR`.

Variable	Bedeutung
PATH	Pfadangaben zur Lokalisierung von ausführbaren Dateien.
LANG	Dient der Regionalisierung von Informationen.
TERM	Typ des Terminals für Ein- und Ausgaben.
USER	Aktueller Benutzername der Sitzung.
HOME	Heimatverzeichnis des aktuellen Benutzers.

Tabelle 7.1: Liste ausgewählter Umgebungsvariablen.

Das System hält eine Liste von Umgebungsvariablen in Form von Name/Werte-Paaren vor, die als *Environment List* bezeichnet wird. *Name* spezifiziert einen Variablenamen und *Wert* einen zugewiesenen Eintrag.

Die *Environment List* wird durch einen Zeiger auf das `environ`-Array repräsentiert und muß extern deklariert werden:

```
extern char **environ;
```

Wenn ein Programm über die `exec`-Funktionen `execle(3)` oder `execve(3)` gestartet wird, setzen wir die Environment List manuell. Die anderen `exec`-Varianten erben die Liste des Prozesses, der `exec(3)` aufgerufen hat.

Eine einfache Implementierung des `printenv(7)`-Kommandos finden wir in Listing 7.19.

Listing 7.19: Einfache Implementierung des `printenv(7)`-Kommandos.

```

1 #include "header.h"
2
3 extern char **environ;
4
5 int main(int argc, char **argv) {
6     int i;
7
8     for (i = 0; environ[i] != NULL; i++)
9         printf("Variable %d: %s\n", i, environ[i]);
10
11     return (0);
12 }
```

Listing 7.19: `xcode/printenvimpl.c` - Einfache Implementierung des `printenv(7)`-Kommandos.

Ist uns der Name einer Umgebungsvariablen bekannt, dessen Wert wir abfragen möchten, können wir `getenv(3)` einsetzen:

```
#include <stdlib.h>
```

```
char *getenv(const char *name);
```

Rückgabewerte: Zeiger auf den Wert der Umgebungsvariable oder `NULL` bei Fehler.

`name`

Name der Umgebungsvariable dessen Wert wir abfragen möchten.

Der Einsatz der Funktion ist denkbar einfach und wird in Listing 7.20 demonstriert.

Listing 7.20: Einsatz von `getenv(3)`

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     char *maildir, *value, *varname;
5     char *basename = basename_ex(argv[0]);
6
7     if (argc != 2)
8         err_fatal("Usage: %s <varname>\n", basename);
9     else
10        varname = argv[1];
11
12    if (getenv(varname) == NULL)
13        err_fatal("Variable \"%s\" name could not be found\n", argv[1]);
14    else {
15        value = getenv(varname);
16        printf("Value of variable \"%s\": %s\n", argv[1], value);
17    }
18
19    return (0);
20 }
```

Listing 7.20: xcode/getenv.c - Einsatz von getenv(3)

□

7.11 Kommandozeilenparameter verarbeiten

AnwenderInnen, die UNIX auf der Kommandozeile verwendet haben, wissen, daß fast alle Programme auf ihre Art und Weise Parameter verarbeiten. Sei es, ob wir `ls -la` aufrufen, um alle Dateien inklusive .-Verzeichnisse in einer langen Liste anzuzeigen, oder

```
find / -type f \(\ -perm -2 -o -perm -20 \) -exec ls -lg {} \; > ~/ww-files
```

um alle Dateien zu finden, die von allen Benutzern geschrieben werden können. Sie alle verwenden weitgehend das gleiche Format und ist Resultat einer Einrichtung, die sehr früh von den UNIX-Entwicklern aufgegriffen wurde und als `getopt`-Mechanismus bezeichnet werden kann. Er wird Gegenstand der Be- sprachungen in diesem Abschnitt sein.

7.11.1 Konventionen

Allgemein gilt für Programme, die Kommandozeilenparameter erwarten folgende Konvention:

```
command [-option [argument] [-option [argument [argn]]]] <file>
```

Die eckigen Klammern zeigen optionale, spitze Klammern Pflichtargumente an. Optionen, angezeigt durch den Bindestrich (-) folgen immer dem Kommando und die jeweiligen Argumente für eine Option direkt nach derselben. Es gibt natürlich immer Ausnahmen, im allgemeinen halten sich jedoch die meisten Tools an die Konventionen. Am einfachsten sind Kommandos, die nur ein Argument erwarten, wie etwa `ls`:

```
% ls -la a* // listet alle Dateien auf, die mit 'a' beginnen
```

Allerdings kann `ls(1)` auch mehrere Argumente aufnehmen, so daß wir beispielsweise nicht nur Dateien anzeigen können, die mit *a* beginnen, sondern auch solche, die mit *b* beginnen:

```
% ls -la a* b* // listet alle Dateien auf, die mit 'a' oder 'b' beginnen
```

`ls(1)` ist vielerleicht Hinsicht hervorragend dazu geeignet, sich mit Kommandozeilenparametern vertraut zu machen, da es die meisten Varianten bietet, die wir antreffen können und keine Schäden bei unsachgemäßen Gebrauch entstehen können.

Ohne es explizit zu erwähnen, haben wir eine Form der Parametrisierung verwendet, die auf den ersten Blick merkwürdig erscheint, uns aber in der Regel ein wenig Schreibarbeit abnimmt. So beschreibt `-la` nicht etwa eine Option namens 'la' sondern faßt die beiden Optionen `-l` und `-a` zusammen. Auf diese Weise können wir viele Optionen zusammenfassen, wie beispielsweise in '`ps -aux`' zur detaillierten Ausgabe der aktuellen Prozesse des Systems. Das geht schneller von der Hand als '`ps -a -u -x`'.

Einige Optionen erwarten ein oder mehrere Argumente. Weiter oben haben wir gesehen, daß `find` die Option `type` unterstützt, daß ein Argument erwartet. Hier haben wir `f` gewählt, daß angeht, daß wir Elemente vom Typ (`type`) Datei (`f, file`) suchen.

Schließlich hat sich im Rahmen der GNU-Software eine weitere Variante durchgesetzt. In vielen Fällen gibt es die kurze und die lange Notation von Argumenten. Beispielsweise können wir die Kurzhilfe von GNU `tar(1)` über die bekannte kurze Notation via `-h` oder die lange Notation `--help` aufrufen.

7.11.2 Parameter mit getopt verarbeiten

Sollen unsere eigenen Programme ebenfalls Kommandozeilenparameter verwenden, müssen wir uns der Bibliotheksfunktion `getopt(3)` bemühen, die alle Probleme für uns löst. Bisher haben wir nur mit Parametern gearbeitet, die Optionen ohne Argumente waren. Beispielsweise haben wir die Anzahl von Durchläufen für einen Test oder den Namen eines Verzeichnisses erfragt. Das sah dann etwa so aus:

```
% ./myprog 5 /tmp/test
```

Irgendwo in der `main`-Funktion haben wir dann die Werte verwendet, um auf die Ausführung Einfluß zu nehmen. Alle Parameter werden beim Start des neuen Prozesses in einem Array gespeichert, daß wir als Argument an `main` übergeben:

```
int main(int argc, char **argv) { ... }
```

Das `argv`-Array beinhaltet Zeiger auf `char`-Arrays, die einen der Parameter enthalten. Demnach wäre obigen Beispiel nach `argc` 3 und `argv[0]` enthält `./myprog`, während `argv[1]` und `argv[2]` die Werte 5 und `/tmp/test` enthielten.

Die Synopsis von `getopt(3)` lautet:

```
#include <unistd.h>

int getopt(int argc, char *const argv[], const char *optstring);
```

Rückgabewert: Nächstes Optionszeichen, das auf der Kommandozeile spezifiziert wurde.

`argc`

Anzahl der in `argv` enthaltenen Optionen (i.d.R. `argc` aus `main`).

`argv[]`

Array vom Typ `char *`, das alle auf der Kommandozeile gelieferten Optionen enthält.

`optstring`

Zeichenkette mit den zu verarbeitenden Optionen.

Direkt mit `getopt(3)` sind folgende Datenstrukturen verbunden:

```

extern char *optarg; /* zeigt auf die aktuell zu bearbeitende Option */
extern int optind; /* zeigt auf die naechste zu bearbeitende Option */
*/
extern int optopt; /* enthaelt Optionszeichen, die nicht erkannt wurden */
extern int opterr; /* zeigt an, ob ein Fehler aufgetreten ist */
extern int optreset; /* IEEE Std1003.2 (POSIX.2) Erweiterung */
*/

```

Die `getopt(3)`-Funktion durchsucht die Kommandozeilenoptionen, und gibt das aktuelle Optionszeichen zurück. Ist dieser Wert -1, wurde das Ende der Optionsliste erreicht. Ein Wert von ? zeigt eine nicht erkannte Option an. Wenn das `optstring`-Argument mit einem Doppelpunkt (':') beginnt, wird der Doppelpunkt zurückgegeben, falls kein Wert für die Option geliefert wurde. Weiter unten sehen wir einige Beispiele, die mehr Licht in das Dunkel bringen. Folgende Tabelle listet die Variablen und deren Bedeutungen im Zusammenhang mit `getopt(3)` auf.

Externe Variable	Beschreibung
<code>char *optarg</code>	Zeigt auf das Argument, welches für die aktuelle Option verarbeitet wird. Das geschieht nur mit Optionen, die Argumente erwarten. Wird beispielsweise die Option <code>-n myname.tar</code> bearbeitet, so zeigt optarg auf einen C String mit dem Inhalt <code>myname.tar</code> .
<code>int optind</code>	Zeigt auf das nächste zu bearbeitende Element im argv-Feld. Mit einer Initialisierung von 1 zeigt optind also auf das erste Argument, denn <code>argv[0]</code> ist der Name des Programms. Wenn das Ende der Optionen erreicht ist, zeigt optind auf das erste Argument. Wird beispielsweise das Kommando <code>tar -xvf myfiles.tar</code> bearbeitet, so hat optind am Ende den Wert 3, der dem Argument <code>myfiles.tar</code> der Option <code>-f</code> entspricht.
<code>int opterr</code>	Wird mit einem Wert von 1 initialisiert und wird von <code>getopt(3)</code> verarbeitet. Wenn der Wert <code>true</code> ergibt und eine Option nicht erkannt wurde, gibt <code>getopt(3)</code> eine Fehlermeldung aus und benennt die fehlerhafte Option.

Tabelle 7.2: Übersicht der von `getopt(3)` verwendeten externen Variablen.

Die externe Variable `optreset` ist eine Erweiterung des Standards IEEE Std1003.2 und wird momentan nur von FreeBSD unterstützt. Sie kann dazu verwendet werden, den internen Zustand von `getopt(3)` wieder zurück zu setzen, so daß die Funktion erneut auf einen anderen Satz von Kommandozeilenoptionen verwendet werden kann. Unter IRIX ab Version 6.5 können Sie dazu `getoptreset(3)` aufrufen. Allerdings ist es in beiden Fällen ratsam, zusätzlich `optind` auf 1 zu setzen, damit es auf das erste Argument in `argv` zeigt.

Der Rückgabewert von `getopt(3)` kann vier unterschiedliche Bedeutungen haben, die sich in der Synopsis nicht ausdrücken lassen:

- Es wird das gerade untersuchte Zeichen zurückgegeben.
- Es wird ein Fragezeichen (?) zurückgegeben, das eine nicht erkannte Option anzeigen.
- Es wird ein Doppelpunkt (':') zurückgegeben, der das Fehlen eines Arguments einer Option angezeigt.
- Es wird -1 zurückgegeben. Es gibt keine weiteren zu untersuchenden Optionen mehr.

7.11.2.1 Definieren der Optionen und Argumente

Bleiben wir beim `tar`-Kommando und schauen wir uns an, wie die Parameter `x`, `v` und `f` von `getopt(3)` verarbeitet werden. Jede Option, die von `getopt(3)` erkannt und verarbeitet werden soll, muß in dem Parameter `optstring` definiert werden:

```

int main(int argc, char **argv) {
    static char optstring[] = "xvf:";
    ... /* Aufruf von getopt() */
}

```

Der Doppelpunkt hinter `f` zeigt an, daß diese Option ein Argument erwartet. Dabei spielt die Reihenfolge der Definitionen keine Rolle, nur der Doppelpunkt muß unmittelbar der Option folgen, die ein Argument erwartet. Das Array `optstring` wird jedes mal durchsucht, wenn eine Option von `getopt(3)` angetroffen wird. Ist die untersuchte Option nicht in `opstring` enthalten wird ? zurückgeliefert, andernfalls untersucht `getopt(3)` das nächste Zeichen. Ist es ein Doppelpunkt (':') wird das Argument für diese Option aus `argv` extrahiert.

7.11.2.2 Extraktion der Optionen und Argumente

Die Optionen und Argumente im `argv`-Array werden in einer Schleife abgerufen und in entsprechenden Variablen zur weiteren Verarbeitung gespeichert. Listing 7.21 zeigt eine solche Schleife, die alle drei Optionen unseres `tar(1)`-Kommandos extrahiert:

Listing 7.21

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(int argc, char **argv) {
5     int extract = 0, verbose = 0; /* option variables */
6     int error, opt;             /* for getopt */
7     char optstring[] = "xvf:"; /* options */
8     char *filename;
9
10    while ((opt = getopt(argc, argv, optstring)) != -1) {
11        switch (opt) {
12            case 'x':
13                extract = 1;
14                printf("-x found\n");
15                break;
16            case 'v':
17                verbose = 1;
18                printf("-v found\n");
19                break;
20            case 'f':
21                filename = optarg;
22                printf("-f %s found\n", optarg);
23                break;
24            default:
25                error = 1; /* unknown option found */
26        }
27    }
28
29    for (; optind < argc; optind++)
30        printf("argv[%d] = %s\n", optind, argv[optind]);
31
32    return (0);
33 }
```

Listing 7.21: xcode/getoptdemo.c - Verwendung von `getopt(3)` zur Extraktion von drei Optionen und einem Argument.

Der Aufruf mit allen Optionen sähe dann so aus:

```
% ./getoptdemo -xvf ./myfile.tar /home/steve/tmp
-x found
-v found
-f ./myfile.tar found
argv[3] = /home/steve/tmp
```

Die **case**-Statements sorgen dafür, daß die Optionen den jeweiligen Variablen zur Weiterverarbeitung zugeführt werden. Aufmerksamkeit verdient die **for**-Schleife am Ende des Programms. Nach der Verarbeitung der Argumente in **optstring** zeigt **optind** genau auf das nächste Element der Kommandozeilenparameter, das nicht in **getopt**-Notation vorliegt. □

Eine Implementierung von **getopt(3)** ist gar nicht so schwer, auch wenn das zunächst den Anschein hat, wie Listing 7.22 zeigt:

Listing 7.22: Beispielimplementierung von getopt(3)

```

1 #include <getopt_impl.h>
2 #include "header.h"
3
4 extern int optind = 1;
5 extern int optopt = 0;
6
7 const char *optarg = NULL;
8
9 int getopt (int argc, char **argv, const char *options) {
10     static int pos = 1;
11     const char *p;
12
13     if (optind >= argc || argv[optind][0] != '-' || argv[optind][1] == 0)
14         return EOF;
15
16     optopt = argv[optind][pos++];
17     optarg = NULL;
18
19     if (argv[optind][pos] == 0) {
20         pos = 1;
21         optind++;
22     }
23
24     p = strchr(options, optopt);
25
26     if (optopt == ':' || p == NULL) {
27         err_normal("illegal option -- ", stdout);
28         goto error;
29     } else if (p[1] == ':') {
30         if (optind >= argc) {
31             err_normal("option requires an argument -- ", stdout);
32             goto error;
33         } else {
34             optarg = argv[optind];
35
36             if (pos != 1)
37                 optarg += pos;
38
39             pos = 1;
40             optind++;
41         }
42     }
43
44     return optopt;
45
46 error:
47     err_normal(optopt, stdout);
48     err_normal('\n', stdout);
49
50     return '?';
51 }
```

Listing 7.22: xcode/getopt_impl.c - Beispielimplementierung von getopt(3)

Nachdem wir geprüft haben, ob überhaupt noch Optionen in der getopt(3)-Syntax vorliegen, schieben wir den Optionszeiger in Zeile 16 um eine Position vor (nämlich um das Zeichen '-') und setzen ggf. die Position innerhalb der Argumentliste vor, bis wir wieder eine gültige Option finden (Zeilen 19 bis 22). getopt zeigt in jedem Fall auf die folgende Zeichenkette, die eine gültige Option sein kann oder nicht. Nun müssen wir eigentlich nur noch die Option mit strchr(3) aus der Optionsliste extrahieren und auf Gültigkeit prüfen, beispielsweise ob die Option selbst ein Argument erwartet. Ist alles in Ordnung schieben wir den Zeiger auf die nächste Option in der Liste (optind++) und setzen den internen Positionsähler pos für diese Option auf 1 zurück. Nachwie vor zeigt getopt auf die richtige Option.

Damit wir die Funktion als Bibliotheksfunktion einsetzen können, müssen wir einen Header erstellen, der zum Einen die Prototypen als extern exportiert und zum Anderen die Variablen korrekt definiert. Wie wir in Listing 7.23 sehen können bereiten wir auch gleich alles für getopt_long(3) vor.

Listing 7.23: Header-Datei für getopt(3) und getopt_long(3)

```

1  #ifndef _GETOPT_IMPL_H
2  #define _GETOPT_IMPL_H
3
4  extern char *optarg;
5  extern int optind;
6  extern int optarg;
7  extern int optopt;
8
9  struct option {
10     const char *name;
11     int has_arg;
12     int *flag;
13     int val;
14 };
15
16 extern int getopt(int argc, char *const *argv, const char *shortopts);
17 extern int getopt_long(int argc, char *const *argv, const char *shortopts,
18     const struct option *longopts, int *longind);
19 extern int getopt_long_only(int argc, char *const *argv,
20     const char *shortopts, const struct option *longopts, int *longind);
21
22 #endif /* _GETOPT_IMPL_H */

```

Listing 7.23: xcode/getopt_impl.h - Header-Datei für getopt(3) und getopt_long(3)

Der Header dürfte weitgehend selbsterklärend sein. Die Variablen und Funktionen sind extern deklariert weil sie als Vorwärtsdeklarationen fungieren und in anderen Übersetzungseinheiten definiert sind. □

7.11.3 Subparameter mit getsubopt verarbeiten

Oftmals erfordern Optionen weitergehende Einstellungen, die mittels Unteroptionen (Suboptions) verarbeitet werden. Einschlägige UNIX Manpages führen das Kommando `mount(1)` als passendes Beispiel für `getsubopt(3)` an, so daß wir uns dem anschließen und zur Grundlage unserer Diskussionen machen wollen:

```
mount -t nfs -o rw,hard,bg,wsize=1024 xp101:/usr /usr
```

Die Option `-o` leitet hier eine Unteroption ein, die aus den Optionen `rw`, `hard`, `bg` und `wsize` besteht. Letzteres erwartet ein Argument, während die anderen drei für sich selbst stehen.

Die Synopsis von `getsubopt(3)` lautet wie folgt:

```
#include <stdlib.h>
extern char *suboptarg;

int getsubopt(char **optionp, char *const *keylistp, char **valuep);
Rückgabewert: Index des übereinstimmenden Elements oder -1 wenn keines gefunden wurde.
```

optionp

Zeiger auf einen Zeiger der Zeichenkette, die untersucht werden soll.

keylistp

Array von Zeigern auf gültige Optionszeichenfolgen.

valuep

Zeiger auf den Wert der Option. Ist nach dem Aufruf von `subgetopt(3)` kein Wert für die Option vorhanden, ist der Wert `NULL`.

Das von `keylistp` referenzierte Array enthält die zu untersuchenden Optionen und hat, bezugnehmend auf das `mount(3)`-Kommando, folgenden Aufbau:

```
static char *myopts[] = {"rw", "hard", "bg", "wsize", NULL};
```

Die Anwendung von `getsubopt(3)` ist mit der von `getopt(3)` vergleichbar. Allerdings müssen wir beachten, daß die Unteroptionen als Argument für eine andere Option auftreten. Praktisch könnte das etwa so aussehen:

```
static char *myopts[] = {"rw", "hard", "bg", "wsize", NULL};
int c, wsize;
char *options, *value;
extern char *optarg;
...
while((c = getopt(argc, argv, "t:o:")) != -1) {
    switch (c) {
        case 't': /* process option -t */
        case 'o':
            options = optarg;
            while (*options != '\0') {
                switch(getsubopt(&options, myopts, &value)) {
                    case 1: /* process rw option */
                    ...
                    case 4: /* wsize=arg */
                        if (value != NULL) wsize = atoi(value);
                    ...
    }
}
```

Wir erkennen, daß wir `options` den Wert von `optarg` zuweisen (genauer gesagt: wir fragen die Adresse der Zeichenkette ab), daß die Zeichenkette mit den Unteroptionen von `-o` enthält. Anschließend führen wir diese Optionszeichenfolge `getsubopt(3)` zu.

Listing 7.24: Verarbeiten von Unteroptionen mit `getsubopt(3)`

Zur Demonstration der `getsubopt(3)`-Funktion ziehen wir ein fiktives Programm, namens `xfmt` heran, das Dateisysteme formatiert und zwei Parameter erfordert:

- `-t` zeigt den Typ des zu formatierenden Dateisystems an
- `-o` nimmt die Unteroptionen, wie Blockgröße und Optimierungsgrad für das Dateisystem auf

Folgendes Listing zeigt den Quelltext zur Verarbeitung der Unteroptionen:

```

1 #include "header.h"
2
3 int getsubopt(char **optionp, char *const *keylistp, char **valuep);
4
5 int main(int argc, char **argv) {
6     extern int      optind, optopt;
7     char           *subopts;
8     static char    optstring[] = "t:o:";
9     static char   *tokens[] = {"bsize", "opt", NULL};
10
11    int return_code = 0, arg_index;
12    char *valuep; /* pointer to subopt value */ */
13    int c; /* option character */ */
14
15    char *fstype; /* filesystem type */ */
16    int bsize, opt; /* block size, optimization level */
17    char *device;
18
19    while ((c = getopt(argc, argv, optstring)) != -1) {
20        switch (c) {
21            case 't':
22                fstype = optarg;
23                printf("Desired FS type: %s\n", fstype);
24                break;
25            case 'o':
26                subopts = optarg;
27                while (*subopts != 0) {
28                    switch ((arg_index = getsubopt(&subopts, tokens, &valuep))) {
29                        case 1: /* bsize */
30                            bsize = valuep ? atoi(valuep) : 1024;
31                            printf("Block size: %d\n", bsize);
32                            break;
33                        case 2: /* opt */
34                            opt = valuep ? atoi(valuep) : 0;
35                            printf("Optimization level: %d\n", opt);
36                            break;
37                        case -1:
38                            printf("Illegal suboption: %s%s%s\n",
39                                   subopts, valuep ? "=" : "", valuep ? valuep : "");
39                            break;
40                        default:
41                            /* Unknown suboption. */
42                            printf("Unknown suboption '%s'\n", valuep);
43                            break;
44                    }
45                }
46            break;
47        default:
48            return_code = 1; /* indicate wrong usage */
49        }
50    }
51
52    if (optind < argc)
53        device = argv[optind];
54    else
55        err_fatal("device name missing\n");
56
57    for (; optind < argc; optind++)
58        printf("argv[%d] = '%s'\n", optind, argv[optind]);
59
60    return return_code;
61

```

 62 }

Listing 7.24: `xcode/getsuboptdemo.c` - Verarbeiten von Unteroptionen mit `getsubopt(3)`.

Möchten wir das Slice auf `/dev/c0t1d1s4` formatieren, eine Blockgröße von 512 Byte und den eine Optimierung der Stufe 3 festlegen, würden wir `xfmt` mit folgenden Optionen aufrufen:

```
% ./xfmt -t ufs -o bsize=512,opt=3 /dev/c0t1d1s4
Desired FS type: ufs
Block size: 512
Optimization level: 3
argv[3] = /dev/c0t1d1s4
```

□

Auch in diesem Fall wollen wir nicht voranschreiten, ohne einen Blick auf eine Implementierung von `getsubopt(3)` geworfen zu haben. Listing 7.25 zeigt, wie es gehen kann.

Listing 7.25: Beispielimplementierung von `getsubopt(3)`

```
1 #include < getopt_impl.h>
2 #include "header.h"
3
4 char *suboptarg = NULL;
5
6 int getsubopt(char **optionp, char *const *tokens, char **valuep) {
7     int cnt;
8     char *p;
9
10    suboptarg = *valuep = NULL;
11
12    if (!optionp || !*optionp)
13        return (-1);
14
15    for (p = *optionp; *p && (*p == ',' || *p == '=' || *p == '\t'); p++) ;
16
17    if (!*p) {
18        *optionp = p;
19        return (-1);
20    }
21
22    /* save the start of the token, and skip the rest of the token. */
23    for (suboptarg = p;
24         **++p && *p != ',' && *p != '=' && *p != ' ' && *p != '\t'; ) ;
25
26    if (*p) {
27        /*
28         * If there's an equals sign, set the value pointer, and
29         * skip over the value part of the token. Terminate the
30         * token.
31        */
32        if (*p == '=') {
33            *p = '\0';
34
35            for (*valuep = p++; *p && *p != '=', &&
36                 *p != ' ' && *p != '\t'; p++) ;
37
38            if (*p)
39                *p++ = '\0';
40        } else
41            *p++ = '\0';
42    }
43}
```

```

42         /* Skip any whitespace or commas after this token. */
43         for (; *p && (*p == ',' || *p == ' ' || *p == '\t'); p++) ;
44     }
45
46     *optionp = p; /* set optionp for next round. */
47
48     for (cnt = 0; *tokens; tokens++, cnt++)
49         if (!strcmp(suboptarg, *tokens))
50             return(cnt);
51
52     return (-1);
53 }

```

Listing 7.25: xcode/getsubopt_impl.c - Beispielimplementierung von getsubopt(3)

Welche Werte wir erhalten, haben wir zuvor besprochen, doch was intern damit passiert ist einen Blick wert. Zuerst setzen wir die beiden Variablen `suboptarg` und `valuep` auf `NULL`. Das macht Sinn, denn wir wissen ja nicht in welcher „Runde“ wir uns befinden, `getsubopt(3)` quasi das erste mal oder bereits das dritte mal aufgerufen wurde. Außerdem benötigen wir `valuep` nicht zum lesen, so daß wir, falls wir keine Unteroptionen finden, eine saubere Umgebung hinterlassen.

Anschließend brauchen wir nur noch herauszufinden, wo die Argumente für eine Option beginnen, falls vorhanden, und sorgen dafür, daß der Zeiger `valuep` auf das erste Zeichend es Tokens steht, daß nicht erkannt wurde und nicht einen Zeiger auf den Wert enthält (Zeile 35). Nachdem wir das Argument der Unteroption verarbeitet haben ermitteln wir den Index des Tokens, das der letzten erkannten Unteroption entspricht; wurde beispielsweise zuletzt `bg` spezifiziert, wäre der Index 2. □

7.11.4 Die GNU-Erweiterung getopt_long

Anmerkungen

Die folgenden Ausführungen sind nur für Systeme mit GNU-Erweiterungen gültig und sind nicht durch den POSIX-Standard definiert.

Viele GNU-Programme erlauben die Angabe von langen Optionen. Sie zeichnen sich durch das Voranstellen von zwei Bindestrichen ('--') aus. Auch sie können Argumente aufnehmen oder für sich allein stehen. Sofern Sie die GNU-Version von `tar(1)` installiert haben, können Sie die Versionsnummer mit der langen Option `--version` abrufen:

```
% tar --version
tar (GNU tar) 1.15.1
```

Um diese Form der Optionszeichenketten zu verarbeiten, müssen wir die Bibliotheksfunktion `getopt_long\index{getopt_long}(3)` verwenden:

```
#ifdef __GNU_SOURCE
#include <getopt.h>
#else
#include <stdlib.h>

int getopt_long(int argc, char *const *argv, const char *optstring,
               const struct option *longopts, int *longindex);
```

Rückgabewert: Nächstes Optionszeichen, das auf der Kommandozeile spezifiziert wurde.

`argc`

Anzahl der zu verarbeitenden Optionen.

argv

Zeiger auf das Array mit Optionszeichenketten.

optstring

Enthält die zu untersuchenden Optionen in Kurzform.

longopts

Zeiger auf eine Struktur vom Typ **option**, das die langen Optionen mit ihren Einstellungen verwaltet.

longindex

Zeiger auf einen Integer, der den Index einer Variable in **longopts** angibt.

Die Struktur **options** weist folgende Member auf:

```
struct option {
    const char *name; /* langer name der Option */
    int has_arg; /* 1 mit Argument, 0 ohne Argument, 2 optional */
    int *flag; /* zeigt an, ob val zurueckgegeben werden soll */
    int val; /* Rueckgabewert oder Zeiger auf Variable */
};
```

Die **option**-Struktur muß vor dem Aufruf von `getopt_long\index{getopt_long}(3)` initialisiert werden, wobei alle Felder des letzten Elements im Array 0 sein müssen:

```
static struct option options[] = {
    {"help", 0, 0, 'h'}, /* name, has_arg, flag, val */
    {"version", 0, 0, 'v'},
    {"verbose", 0, &verbose_flag, 1},
    {"type", 1, 0, 0},
    {0, 0, 0}
```

Die Bedeutung der Member der **option**-Struktur, bezugnehmend auf oben abgebildete Struktur, lässt sich in wenigen Punkten zusammenfassen:

- **name** enthält immer die lange Option, beispielsweise `help`, die auf der Kommandozeile als `--help` ausgeschrieben wird.
- **has_args** zeigt an, ob die Option ein Argument erwartet oder nicht. Alternativ kann das auch offen bleiben, soll heißen, dass eine Option ein Argument erwarten kann, muß sie aber nicht. Das ist beispielsweise nützlich, wenn wir über Standardeinstellungen verfügen, die bei Bedarf übergangen werden können. **has_args** kann entweder numerisch oder über die symbolischen Konstanten `no_argument` (0), `required_argument` (1) und `optional_argument` (2) gesetzt werden.
- **flag** ist entweder 0 oder ein Zeiger auf eine Variable, die in Abhängigkeit **val** verwendet wird (dazu gleich mehr).
- **val** zeigt einen Wert an, der für die Variable von **flag** gesetzt werden soll, wenn **flag** nicht 0 ist.

Die Abhängigkeit zwischen den beiden Membern **flag** und **val** ist auf den ersten Blick etwas undeutlich. Wenn **flag** auf eine Variable zeigt, wird sie beim Antreffen der in **name** spezifizierten Option auf den in **val** definierten Wert gesetzt, egal ob dieser größer 0 oder 0 ist. Beispiel:

```
int verbose_flag = 0;
...
static struct option options [] = {
    {"verbose", 0, &verbose_flag, 1},
    {"quiet", 0, &verbose_flag, 0},
    {0, 0, 0}
```

In diesem Fall wird beim Antreffen der Option `--verbose` die Variable `verbose_flag` auf 1 gesetzt und beim Antreffen von `--quiet` auf 0 gesetzt.

Listing 7.26

Das folgende Beispiel zeigt wie wir lange Optionen mit `getopt_long(3)` auslesen. Wir können die Optionen `--verbose`, `--quiet`, `--append`, `--add` und `--delete` verwenden, wobei `delete` ein Argument erfordert.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <getopt.h>
4
5 int main(int argc, char **argv) {
6     int c, verbose_flag;
7
8     static struct option long_options [] = {
9         {"verbose", no_argument,      &verbose_flag, 1},
10        {"quiet",   no_argument,      &verbose_flag, 0},
11        {"add",     no_argument,      0, 'a'},
12        {"append",  no_argument,      0, 'b'},
13        {"delete",  required_argument, 0, 'd'},
14        {0, 0, 0, 0}
15    };
16
17    while (1) {
18        int option_index = 0; /* reset */
19        c = getopt_long(argc, argv, "abd", long_options, &option_index);
20
21        if (c == -1)
22            break;
23
24        switch (c) {
25        case 0:
26            if (long_options[option_index].flag != 0) break;
27
28            printf("Option %s ", long_options[option_index].name);
29
30            if (optarg)
31                printf ("with argument '%s' processed.", optarg);
32
33            printf("\n");
34            break;
35        case 'a':
36            printf("Option -a processed\n");
37            break;
38        case 'b':
39            printf("Option -b processed\n");
40            break;
41        case 'd':
42            printf ("Option -d with argument '%s' processed\n", optarg);
43            break;
44        case '?':
45            break;
46        default:
47            abort();
48        }
49    }
50
51    if (verbose_flag)
52        printf("Verbose flag set.\n");

```

```
53
54     while (optind < argc)
55         printf ("%s\n", argv[optind++]);
56
57     return (0);
58 }
```

Listing 7.26: xcode/getoptlongdemo.c - Verwendung von getopt_long.

□

Eine Implementierung von `getopt_long(3)` ist weitaus komplexer als die beiden „kurzen“ Varianten. Da sich ein Listing über fast sechs Seiten ziehen würde und es sich bei der Funktion ohnehin um eine GNU-Erweiterung handelt, erspare ich Ihnen eine ausführliche Besprechung und verweise für alle Neugierigen stattdessen auf die Quelltextdatei `xcode/getopt_long_impl.c`.

Kapitel 8

Signalbehandlung

You have to allow a certain amount of time in which you are doing nothing in order to have things occur to you, to let your mind think.

MORTIMER ADLER

Während Geräte Hardware-Interrupts an den Kernel senden können, um Zeit für die Durchführung Ihrer Aufgaben anzufordern, können wir auf die gleiche Weise unsere Applikationen steuern. Wir tun dies allerdings mit Software-Interrupts, auch Signale (*signals*¹) genannt. Es gibt unterschiedliche Signale mit jeweils eigenen Bedeutungen, beispielsweise **SIGINT**, das für den Vordergrundprozess generiert wird, wenn der Benutzer STRG-C drückt. Der Prozess wird standardmäßig beendet, sobald ihm das Signal zugestellt wurde. Prozesse können die meisten Signale gezielt ignorieren oder blockieren.

Frühe UNIX-Implementierungen hatten Probleme mit der Signalübermittlung. So konnten Signale einfach verloren gehen und zu Fehlverhalten führen. Des Weiteren hatten einige Prozesse Schwierigkeiten, bestimmte Signale zu deaktivieren, um kritische Regionen ungestört abarbeiten zu können. Mit der Einführung von SVR3 und 4.3BSD fanden *reliable signals* (zuverlässige Signale) Einzug in die UNIX-Welt und räumten mit den Problemen der ersten Implementierungen auf. POSIX greift diese Variante auf und sie wird Kernpunkt der Ausführungen in diesem Kapitel sein.

Wenn ein Signal an einen Prozess gesendet wurde, sagt man auch, daß es für den Prozess eingetragen (*posted*) oder für ihn generiert (*generated*) wurde. Zu diesem Zeitpunkt weiß nur der Kernel, daß es dieses Signal gibt. Wird der Prozess nun über das Auftreten des Signals informiert, so spricht man auch von der Zustellung des Signals (*signal delivery*). Dazwischen gibt es ein nicht definiertes Zeitfenster, was in direktem Zusammenhang zu den oben angesprochenen Problemen steht (mehr Informationen in Abschnitt 8.3).

Prozesse dürfen bis zu einem gewissen Grad selbst entscheiden, wie sie von einem Signal beeinflußt werden möchten. Das wird als Signaldisposition (*signal disposition*) bezeichnet. Bei Programmstart verfügt jeder Prozess über eine Standarddisposition (auch: *Standardeinstellung*, in der UNIX-Welt aber als Disposition bekannt), d.h. daß immer die mit dem Signal assoziierte Standardaktion ausgeführt wird. Viele Signale dürfen ignoriert werden, so daß der Prozess niemals über das Eintreffen eines Ereignisses informiert wird, andere dürfen niemals ignoriert werden und werden immer zugestellt. Jedes Signal verfügt über eine assoziierte Standardaktion. Meistens führt sie zur Beendigung (*termination*) des Prozesses oder es wird einfach ignoriert (*ignored*). Alternativ kann der Prozess für den Empfang eines oder mehrerer Signale eingestellt werden. Als Resultat reagiert der Prozess mit der Ausführung einer bestimmten Aktion durch die Signalbehandlung, die als Funktion (*signal handler*) realisiert wird. Es ist möglich, für jedes Signal eine andere Funktionen oder eine Funktion zur Bearbeitung mehrerer Signale zu definieren.

¹In diesem Kapitel nenne ich zu den jeweiligen Fachbegriffen oder Abläufen häufiger als sonst auch die englischen Begriffe, wie sie in den Manpages und der englischen Fachliteratur verwendet werden. Der Grund dafür liegt darin, daß gerade im Bereich der Signalbehandlung die Terminologie eine sehr wichtige Rolle für das Verständnis spielt und ich daher besonderen Wert auf die Erwähnung der Fachbegriffe lege.

Bei der Terminierung eines Prozesses wird, abhängig von der Art des Ereignisses, ein Core Dump erzeugt, der ein Abbild des Speicherbereichs des Prozesses enthält. Diese Datei ist eine binäre Datei, die sich anschließend im Ausführungsverzeichnis des Prozesses befindet.

Mit POSIX-konformen Shells können Core Dumps mittels `ulimit -c 0` deaktiviert und mit `ulimit -c 1` aktiviert werden. Die maximale Dateigröße von Core Dumps kontrollieren Sie mit `limit coredumpsize 10240`, was eine Dateigröße von maximal 10 MB zuläßt. Für weitere Informationen können sie `core(4)` konsultieren.

Prozesse dürfen auch Signale halten (auch *blockieren* genannt). Dabei handelt es sich nicht um eine Signaldisposition, denn das kann unabhängig von der Disposition geschen. Das Halten eines Signals bedeutet, daß es weiterhin im Zustand *Pending* (ausstehend) verbleibt, bis es freigegeben (*released*) wurde. Wird ein Signal mehrfach empfagen, zeichnet es der Kernel nur einmal auf. Er führt nur Buch darüber, daß ein Signal eingetroffen ist aber nicht wie oft. Signale sind also nicht dazu geeignet, zu zählen wie oft ein Ereigniss aufgetreten ist.

8.1 Die Bedeutung der Signale

Ein Signal kann als sehr kleine (besser: kurze) Nachricht verstanden werden, die an einen Prozess oder eine Prozessgruppe übermittelt wird. Dabei handelt es sich meist nur um eine Nummer, welche das Signal als solches identifiziert und gleichzeitig eine Bedeutung zuordnet. Eine ganze Reihe Makros sind zur Signalverarbeitung definiert und erhalten das Präfix `SIG`. Beispielsweise wird durch `fork(2)`, `vfork(2)` oder `clone(2)` das Signal `SIGCHLD` abgesetzt. Das Makro expandiert zum Wert 17 und sendet den Bezeichner des Signals an den Elternprozess, wenn eines seiner Kinder anhält oder terminiert wird. Ein anderes Beispiel ist das allgegenwärtige Signal `SIGSEGV` (*segmentation fault*), das den Wert 11 hat und einen Speicherzugriffsfehler anzeigt und an einen Prozess gesendet wird, der einen Speicherbereich falsch referenzierte.

Signale haben die folgenden beiden Aufgaben:

- Sie benachrichtigen Prozesse über spezifische Ereignisse.
- Sie veranlassen Prozesse eine Signalbehandlungsroutine auszuführen.

Beide Punkte schließen sich nicht aus, sondern ergänzen sich in bestimmten Fällen, denn oftmals muß ein Prozess eine bestimmte Funktion aufgrund eines Signals ausführen.

Die folgende Tabelle listet alle POSIX/ANSI-kompatiblen Signale und ihre Bedeutung auf.

Wie erwähnt, wird ein Signal erzeugt oder einem Prozess/Thread gesendet, wenn das Ereignis (`event`), welches das Signal erzeugt hat, erstmals auftritt. Beispiele solcher Ereignisse schließen Hardwarefehler, Zeitüberschreitungen, sowie Echtzeitsignale von `kill(2)`- oder `sigqueue`-Funktionsaufrufen ein. Unter bestimmten Umständen kann das gleiche Ereignis Signale für mehrere Prozesse erzeugen.

Zum Zeitpunkt der Signalerzeugung steht fest, ob das Signal für einen Prozess oder ein Thread bestimmt ist. Das heißt, Signale, die durch eine Aktion für einen bestimmten Thread, wie etwa Hardwarefehler, erzeugt wurden, sind stets für den Thread vorgesehen, der das Ereignis hervorgerufen hat. Das gleiche gilt auch für Signale, die für Prozesse bestimmt sind.

Jedem Signal ist eine Aktion zugeordnet, die bei Eintreffen eines bestimmten Signals ausgeführt werden soll. Es wird zwischen folgenden Begrifflichkeiten unterschieden: Ein Signal gilt als *ausgeliefert*, wenn die entsprechende Aktion durch den Prozess ausgeführt wird. Signale werden vom einem Prozess *akzeptiert*, wenn das Signal ausgewählt und durch die `sigwait(3)`-Funktionen von der Liste der aussehenden Signale entfernt wurde.

In dem Zeitraum zwischen Signalerzeugung und -akzeptanz befindet sich das Signal in dem Zustand *pending* (ausstehend). Diese Zeitspanne kann nicht durch eine Applikation abgefragt werden. Allerdings kann ein Signal von der Auslieferung abgehalten (*blocked*) werden. Solange die mit dem Signal verknüpfte

#	Name	Aktion	Beschreibung	Standard
1	SIGHUP	Abort	Hängt das Kontrollterminal oder den Prozess auf	POSIX
2	SIGINT	Abort	Unterbrechungsanforderung der Tastatur	ANSI
3	SIGQUIT	Dump	Beendigung durch die Tastatur	POSIX
4	SIGILL	Dump	Illegalen Instruktion	ANSI
5	SIGTRAP	Dump	Haltepunkt für Befehlsverfolgung (<i>trace</i>)	POSIX
6	SIGABRT	Dump	Unnormale Beendigung	ANSI
7	SIGBUS	Abort	Fehler auf dem Bus	4.2 BSD
8	SIGFPE	Dump	Floating Point Exception	ANSI
9	SIGKILL	Abort	Erzwungene Prozessterminierung	POSIX
10	SIGUSR1	Abort	Benutzerdefiniertes Signal 1	POSIX
11	SIGSEGV	Dump	Ungültige Speicherreferenz	ANSI
12	SIGUSR2	Abort	Benutzerdefiniertes Signal 2	POSIX
13	SIGPIPE	Abort	Schreibvorgang auf eine Pipe ohne Leser (<i>broken pipe</i>)	POSIX
14	SIGALARM	Abort	Echtzeituhr	POSIX
15	SIGTERM	Abort	Prozessterminierung	ANSI
16	SIGSTKFLT	Abort	Stack-Fehler	Alle
17	SIGCHLD	Ignore	Kindsprozess hat angehalten oder wurde terminiert	Sys V
18	SIGCONT	Continue	Fahre mit der Ausführung fort, falls zuvor angehalten	POSIX
19	SIGSTOP	Stop	Halte Ausführung des Prozesses an	POSIX
20	SIGTSTP	Stop	Halte Ausführung des Prozesses an, falls TTY	POSIX
21	SIGTTIN	Stop	Hintergrundprozess erfordert Eingabe	POSIX
22	SIGTTOU	Stop	Hintergrundprozess erfordert Ausgabe	POSIX
23	SIGURG	Ignore	Dringende Anfrage von einem Socket	4.2 BSD
24	SIGXCPU	Abort	CPU-Zeitlimit überschritten	4.2 BSD
25	SIGXFSZ	Abort	Dateigrößenlimit überschritten	4.2 BSD
26	SIGVTALRM	Abort	Virtueller Zeitgeber	4.2 BSD
27	SIGPROF	Abort	Zeitgeber des Profilers	4.2 BSD
28	SIGWINCH	Ignore	Fenstergröße neu bestimmen	4.3 BSD, Sun
29	SIGIO	Abort	E/A nun möglich	Sys V
29	SIGPOLL	Abort	Gleichbedeutend mit SIGIO	4.2 BSD
30	SIGPWR	Abort	Problem mit der Stromversorgung	Sys V
31	SIGSYS	Abort	Fehlerhafter Systemaufruf	4.2 BSD

Tabelle 8.1: Liste aller Signale der wichtigsten UNIX-Derivate.

Aktion eine andere außer Ignorieren (*ignore*) ist und das Signal für den Thread generiert wurde, wird das Signal weiterhin als ausstehend behandelt bis der Zustand aufgehoben (*unblocked*), es akzeptiert und durch `sigwait(3)` zurückgegeben wurde, oder die verknüpfte Aktion auf *ignore* gesetzt wurde. Der wesentliche Unterschied zwischen dem Ignorieren und dem Blockieren eines Signals ist der, daß ein Prozess, welcher ein bestimmtes Signal ignorieren möchte es auch niemals zu Gesicht bekommt, denn der Kernel liefert es erst gar nicht an den Prozess aus. Ein Blockiertes Signal ist nur vorübergehend für den Prozess unsichtbar. Nach Aufhebung der Blockade wird es sofort zugestellt.

Obwohl das Konzept der Signale recht intuitiv wirkt, ist die Implementierung im Kernel relativ komplex:

- Der Kernel muß sich „merken“, welche Signale von jedem Prozess blockiert werden.
- Findet ein Kontextwechsel statt (vom Kernelmodus in den Benutzermodus), muß der Kernel prüfen, ob ein Signal für einen Prozess eingetroffen ist. Das passiert beispielsweise bei nahezu jedem Interrupt durch den Zeitgeber.
- Des Weiteren muß der Kernel feststellen, ob das Signal ignoriert werden kann oder nicht. Das ist der Fall, wenn die folgenden Bedingungen zutreffen:
 - Der Zielprozess wird nicht durch einen anderen Prozess verfolgt (*tracing*)

- Das Signal wird nicht durch den Zielprozess blockiert.
- Das Signal wird durch den Zielprozess ignoriert, entweder weil der Prozess das Signal ausdrücklich ignoriert oder weil der Prozess nicht die Standardaktion des Signals auf eine andere gesetzt hat.

8.2 Systemaufrufe und Signale

Wenn Systemaufrufe und Signale zusammen auftreten, können zwei Schwierigkeiten auftreten: die erste betrifft die Wiederaufnahme von Systemaufrufen, die durch Signale unterbrochen wurden. Die zweite betrifft nicht wiedereintrittsfähige Systemaufrufe in Signal Handlern. Was passiert, wenn ein Signal während der Ausführung eines Systemaufrufs abgefangen wird? Das hängt vom Systemaufruf ab. Langsame Systemaufrufe, solche, die beispielsweise mit Terminals zu tun haben, können für eine unbestimmte Zeit blockieren (zum Beispiel wenn BenutzerInnen über einen langen Zeitraum keine Eingaben tätigen). Andere, die von Speichergeräten lesen, blockieren dagegen nicht sehr lange (schließlich dauert das Öffnen einer Datei nur wenige Millisekunden). Andere Systemaufrufe wie etwa `getpid(2)` blockieren überhaupt nicht. Die Möglichkeit, langsame Systemaufrufe unterbrechen zu können ist wichtig, damit der Kernel Signale an Programme ausliefern können, die relativ lange blockieren.

Wird ein langsamer Systemaufruf unterbrochen, liefert er beispielsweise -1 zurück und setzt `errno` auf `EINTR`. Das betreffende Programm muß dieses Problem abfangen, damit der Systemaufruf gegebenenfalls neu gestartet werden kann. Beispielsweise so:

```
int    fd, retval, size;
char *buf;

...
while (retval = read(fd, buf, size), retval == -1 && errno == EINTR);
if (retval == -1)
    /* Fehlerbehandlung */
...
```

Durch das Platzieren des Systemaufrufs in der `while`-Schleife wird der Aufruf neu gestartet, wenn `EINTR` auftritt. Mit der Funktion `sigaction(2)` (Abschnitt 8.7.1 auf Seite 225) können wir festlegen, daß langsame Systemaufrufe neu gestartet werden sollen. Diese Erweiterung ist zwar nicht Bestandteil von POSIX.1, wird aber auch nicht ausdrücklich verboten.

Folgende Regeln sind bei der Verwendung von Systemaufrufen zu beachten:

- Wenn wir nicht sicher sind, starten wir den Systemaufruf explizit neu.
- Wir stellen sicher, daß ein Systemaufruf in einem Signal Handler oder in einer ungefährdet aufgerufen werden kann (asynchrone Signale sicher verarbeiten kann).
- Unerwünschte Signale sollten wir stets Blockieren.

Tabelle 8.2 listet alle POSIX.1-Funktionen auf, die in der Lage sind, asynchrone Signale korrekt zu verarbeiten. Solche Funktionen werden oftmals als *async-safe* bezeichnet.

8.3 Unzuverlässige Signalbehandlung

In diesem Abschnitt lernen wir den traditionellen Mechanismus der Signalbehandlung aus SVR4 kennen. Aus dieser Zeit stammt auch der Begriff der *unzuverlässigen Signalbehandlung*. Das bedeutet, daß es in bestimmten Konstellationen zu Schwierigkeiten bei der richtigen Behandlung der Signale kommen kann. Wir gehen im weiteren Verlauf näher darauf ein. Als Reaktion auf die Schwierigkeiten wurden die *zuverlässigen Signale* (Abschnitt 8.7) eingeführt, die mit der Problematik aufräumten.

Es gibt genau drei Möglichkeiten, wie ein Signal behandelt werden kann:

_Exit	_exit	abort	accept
access	aio_error	aio_return	aio_suspend
alarm	bind	cfgetispeed	cfgetospeed
cfsetispeed	cfsetospeed	chdir	chmod
chown	clock_gettime	close	connect
creat	dup	dup2	execle
execve	fchmod	fchown	fcntl
fdatasync	fork	fpathconf	fstat
fsync	ftruncate	getegid	geteuid
getgid	getgroups	getpeername	getpgrp
getpid	getppid	getsockname	getsockopt
getuid	kill	link	listen
lseek(3)	lstat	mkdir	mkfifo
open	pathconf	pause	pipe
poll	posix_trace_event	pselect	raise
read	readlink	recv	recvfrom()
recvmsg	rename	rmdir	select
sem_post	send	sendmsg	sendto
setgid	setpgid	setsid	setsockopt
setuid	shutdown	sigaction	sigaddset
sigdelset	sigemptyset	sigfillset	sigismember
sleep	signal	sigpause	sigpending
sigprocmask	sigqueue	sigset	sigsuspend
socket	socketpair	stat	symlink
sysconf	tcdrain	tcflow	tcflush
tcgetattr	tcgetpgrp	tcsendbreak	tcsetattr
tcsetpgrp	time	timer_getoverrun	timer_gettime
timer_settime	times	umask	uname
unlink	utime	wait	waitpid
write			

Tabelle 8.2: Liste aller sicheren POSIX.1-Funktionen

- Das Signal wird ausdrücklich *ignoriert*.
- Es wird die **Standardaktion** ausgeführt, welche mit jedem Signal verknüpft ist (siehe Tabelle 8.1). Die durch den Kernel vordefinierten Aktionen hängen vom Signal ab und entsprechen einer der folgenden fünf:

Abort

Der Prozess wird beendet.

Dump

Der Prozess wird beendet und ein Speicherabbild (*core file*) des aktuellen Ausführungskontextes erzeugt.

Ignore

Das Signal wird ignoriert.

Stop

Der Prozess wird angehalten, quasi in den Zustand **TASK_STOPPED** versetzt.

Continue

Der Prozess setzt seine Ausführung fort (**TASK_RUNNING**), sofern er vorher in den Zustand **TASK_STOPPED** versetzt wurde.

- Das Signal wird abgefangen (*caught*) und die verknüpfte Funktion (**signal handler**) ausgeführt.

Nocheinmal: beachten Sie den Unterschied zwischen dem Ignorieren eines Signals und dem Blockieren. Solange das betreffende Signal blockiert ist, wird es nicht von dem Zielprozess empfangen (es steht dessen Zustellung noch aus), aber ignorierte Signale erreichen *nie* seinen Empfänger. Die einzigen Signale, die nicht ausdrücklich ignoriert oder abgefangen werden können, sind **SIGKILL** und **SIGSTOP**. Es wird immer ihre verknüpfte Standardaktion ausgeführt. Nur aus diesem Grund ist es privilegierten Benutzern überhaupt möglich, Prozesse zu beenden oder anzuhalten.

Der erste Einstiegspunkt für das Kennenlernen der Signalbehandlung ist die POSIX-Funktion **signal(2)**.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t(int));
```

oder

```
void (*signal(int sig, void (*func)(int)))(int);
```

Rückgabewerte: Wert von func oder SIG_ERR bei Fehler (siehe unten).

sig

Signal, das von **func** behandelt werden soll.

func

Signal Handler, der für das in **sig** spezifizierte Signal ausgeführt werden soll.

Der Prototyp der Funktion zeigt an, daß zwei Argumente erforderlich sind und ein Zeiger auf eine Funktion zurückgegeben wird, die keinen Rückgabewert hat. Das erste Argument ist ein Integer, der das Signal identifiziert. Als zweites wird die Funktion angegeben, die als Signal Handler fungieren soll. Diese Funktion wiederum erfordert ein Integerargument und gibt keinen Wert zurück. Die Adresse genau dieser Funktion wird als Wert von **sig** zurückgegeben und erfordert einen Integer als Argument, indiziert durch die Angabe von **(int)** als letztes Glied der Signatur.

Mit Hilfe der **typedef**-Direktive erhöhen wir die Lesbarkeit der Funktionsdeklaration erheblich. Beide Varianten sind korrekt und erfüllen die gleiche Aufgabe.

Der mit `signal(2)` verbundene Mechanismus ist eng mit dem klassischen Callback-Mechanismus verwandt: wir übergeben einer Funktion einen Funktionszeiger, der verwendet wird, um die damit referenzierte Funktion aufzurufen, wenn ein bestimmtes Ereignis auftritt. Für die `signal(2)`-Funktion bedeutet das:

```
/* define signal handler for SIGUSR1 */
void myhandler(int signal_number) { ... }

/*
 * Establish myhandler as signal handler fpr SIGUSR1 and
 * save original handler for restoring it later.
 */
sighandler_t handler = signal(SIGUSR1, myhandler);
...
/* restore original signal handler for SIGUSR1 */
signal(SIGUSR1, handler);
```

Wir übergeben der Funktion einen Funktionszeiger, der sich hinter dem Namen der Funktion (`myhandler`) verbirgt. Sie wird aufgerufen, wenn der Anwendung das in `signal(2)` spezifizierte Signal auftritt. Gleichzeitig speichern wir den ursprünglichen Signal Handler um ihn später wieder für das Signal zu installieren.

Das Argument `func` kann entweder die Konstante `SIG_IGN` (*ignore*), `SIG_DFL` (*default*) oder die Adresse einer Funktion sein, die aufgerufen werden soll, wenn das in `sig` angegebene Signal eintrifft.

In der Regel sind `SIG_IGN` und `SIG_DFL` als Makros in `<signal.h>` definiert:

```
#define SIG_DFL ((__sighandler_t *)0)
#define SIG_IGN ((__sighandler_t *)1)
```

Beide Makros expandieren zu kompatiblen Typen für die `signal(2)`-Funktion, hier ausgedrückt durch den Typ `__sighandler_t`.

Listing 8.1: Einfache Signalbehandlung mit `signal(2)`

Das folgende Beispiel zeigt die einfachste Möglichkeit einen Signal Handler zu etablieren. Dazu definieren wir die Funktion `sig_func`, die ausgibt, das der Prozess `SIGUSR1` erhalten hat.

```
1 #include <signal.h>
2 #include <unistd.h>
3 #include "header.h"
4
5 static void sig_func(int signal); /* signal handler */
6
7 int main(void) {
8     if (signal(SIGUSR1, sig_func) == SIG_ERR)
9         err_fatal("signal() failed");
10
11     while (1)
12         pause();
13 }
14
15 static void sig_func(int signal) {
16     if (signal == SIGUSR1)
17         printf("Signal SIGUSR1 received\n");
18     else
19         err_normal("Handling of SIGUSR1 failed, got: %d instead\n", signal);
20
21     return;
22 }
```

Listing 8.1: xcode/sigfunc.c - Einfache Signalbehandlung mit `signal(2)`.

Die Bildschirmausgabe zeigt uns, daß alles funktioniert wie wir es erwarten:

```
% ./signaldemo&
[1] 23733
% kill -USR1 23733
Signal SIGUSR1 received
% kill 23733
% kill -TERM 23733
```

Gehen wir das Programm nun Schritt für Schritt durch:

1-5

Zuerst importieren wir die notwendigen Header. Ohne `<signal.h>` haben wir kein Zugriff auf die Signalbehandlung und die Konstanten, die alle Signale repräsentieren. Die Datei `<unistd.h>` enthält den Prototyp für die `pause(3)`-Funktion. Der Prototyp des Signal Handlers aus Zeile 5 erwartet einen Integer als Argument (`signal`), der die Signalnummer, die wir behandeln wollen, angibt.

8-10

Obwohl die Abfrage, ob unser UNIX-System das Signal `SIGUSR1 (user defined)` verarbeiten kann nicht notwendig ist (das können nämlich alle) sollten wir stets prüfen, ob unser Vorhaben durchführbar ist. Kann das angeforderte Signal nicht behandelt werden, wird `SIG_ERR` zurückgegeben, was -1 entspricht.

12-14

Mit der Funktion `pause(3)` (Abschnitt 8.5.1) setzen wir den Prozess aus, bis ein Signal eintrifft. Damit das immer wieder geschieht, wurde der Aufruf in eine Endlosschleife integriert. Somit können wir beliebig oft versuchen, Signale an unseren Prozess zu schicken.

17-23

Unser Prozess wartet nun also auf das Eintreffen des Signals `SIGUSR1`. Sobald es ankommt, wird die in `signal(2)` angegebene Funktion aufgerufen. Sie prüft, ob das erwartete Ereignis eingetroffen ist (`signal == SIGUSR1`) und gibt bei Erfolg eine entsprechende Meldung aus. Sollte dennoch etwas nicht in Ordnung sein, so reagieren wir darauf und geben das statt dessen empfangene Signal aus. Letzteres sollte niemals passieren, wenn doch, liegt ein schwerwiegender Fehler vor.

Bei der Ausführung schicken wir das Programm in den Hintergrund (`&`) (andernfalls können wir nichts auf dem Terminal eingeben) und setzen das Signal `SIGHUP` mit dem Befehl `kill(1)` ab. □

Die unsichere Signalbehandlung können wir auf den meisten Systemen, die `signal(2)` implementieren, illustrieren, wie Listing 8.2 zeigt:

Listing 8.2: Unzuverlässige Signalbehandlung mit `signal(2)`

Wenn ein Signal Handler mit `signal(2)` installiert wird, wird unter SVR4 die Disposition des Signals auf `SIG_DFL` zurückgesetzt, bevor in den Signal Handler eingetreten wird. Soll der Signal Handler das mehrfache Auftreten des Signals behandeln, so muß er sich immer wieder neu installieren, damit nicht die Standardaktion für das Signal in Kraft tritt. Zwischen dem Zurücksetzen der Disposition des Handlers und der Laufzeit desselben liegt immer ein kleines Zeitfenster, in der das Signal erneut auftreten kann, dann allerdings mit dem assoziierten Standardverhalten. Wie bereits erwähnt beenden die meisten Signale einen Prozess.

Die unzuverlässige Signalbehandlung wurde nur unter System V Release 2 und früher verwendet. Seit Release 3 haben die zuverlässigen Signale *reliable signals* Einzug in die Unix-Welt gehalten und haben den traditionellen Mechanismus abgelöst. Die Funktion `signal(2)` ist nur aus Gründen der Abwärtskompatibilität vorhanden und wird intern auf einen sicheren Mechanismus umgelegt.

Um dieses Zeitfenster zu illustrieren, betrachten wir folgendes kleines Beispiel. Es reinstalliert den Handler immer wieder selbst, wartet aber zuvor eine Sekunde. Somit fällt es uns leichter, diesen Zusammenhang zu verstehen.

```

1 #include <signal.h>
2 #include <unistd.h>
3 #include "header.h"
4
5 int main(void) {
6     int i;
7     void handler(int);
8
9     signal(SIGINT, handler); /* install signal handler */
10    printf("PID: %d\n", (long)getpid());
11
12    /* pause 5 times; interrupted by signals */
13    for (i = 0; i < 5; i++) {
14        printf("Run %d\n", i);
15        pause();
16    }
17
18    return (0);
19}
20
21 void handler(int sig_num) {
22     printf("Caught SIGINT\n");
23     sleep(1); /* create a timing window */
24
25     signal(SIGINT, handler); /* reinstall signal handler */
26 }
27

```

Listing 8.2: xcode/signal.c - Unzuverlässige Signalbehandlung mit signal(2).

Wir beginnen mit der Installation des Handlers für das Signal SIGINT. Dann rufen wir fünf mal `pause(3)` auf, um die Ausführung des aufrufenden Prozesses auszusetzen bis ein Signal auftritt. Wenn `pause(3)` zurückkehren soll, dürfen wir das Signal nicht ignorieren (`pause(3)` besprechen wir in Abschnitt 8.5.1 auf Seite 212). Das ist verständlich, denn wenn das Signal die Terminierung des Prozesses zur Folge hat (und das ist bei CTRL-C [SIGINT] der Fall), so wird `pause(3)` nicht zurückkehren. Der Handler gibt eine Nachricht aus, die uns sagt, daß das Signal abgefangen wurde. Dann wird `sleep(3)` mit dem Argument 1 aufgerufen, das den Prozess veranlaßt, die Ausführung für eine Sekunde auszusetzen. Führen wir das Programm aus und drücken wir mehrfach STRG-C (*interrupt key*) so wird immer wieder die Meldung ausgegeben. Liegt der Zeitraum zwischen dem Drücken von STRG-C unter einer Sekunde, wird das Programm beendet, da dies die Standardaktion des Signals ist. Abbildung 8.1 illustriert den Zusammenhang.

Leider können wir keine aussagekräftigere Aufführung der unzuverlässigen Signalbehandlung mit `signal(2)` sehen, da neuere Implementierungen den Aufruf von `signal(2)` auf einen zuverlässigen Mechanismus, beispielsweise `sigaction(2)` (Abschnitt 8.7.1), umlegen. Zur Demonstration müßten wir auf eine System V Variante zurückgreifen, die noch keine zuverlässige Signalverarbeitung kennt.

Das erste Signal wird durch den installierten Signal Handler behandelt und alles läuft planmäßig (1). Bevor das Signal durch den Handler behandelt wird, setzt der Kernel die Disposition des Prozesses auf die Standardeinstellung zurück. Erst wenn der Handler zurückkehrt, reinstalliert der Kernel die ursprüngliche Disposition. Trifft nun ein Signal das zweite Mal ein, bevor der Kernel den Signal Handler wieder installiert hat (2), wird die assozierte Standardaktion (`SIG_DFL`) durchgeführt. In Listing 8.2 hätte das die Beendigung des Programms zur Folge.

Ein Nachteil der unzuverlässigen Signalverarbeitung ist, daß wir alle Signale ignorieren müssen, von denen wir nicht wollen, daß sie unseren Prozess beeinflussen. Das führt natürlich dazu, daß sie das Signal (oder sogar mehrere) einfach verpassen. Manchmal möchten wir das Signal nicht ignorieren, aber selbst entscheiden, wann wir es verarbeiten wollen. Das erreichen wir durch den Einsatz von *Signalsätzen*, wie sie in Abschnitt 8.6 besprochen werden.

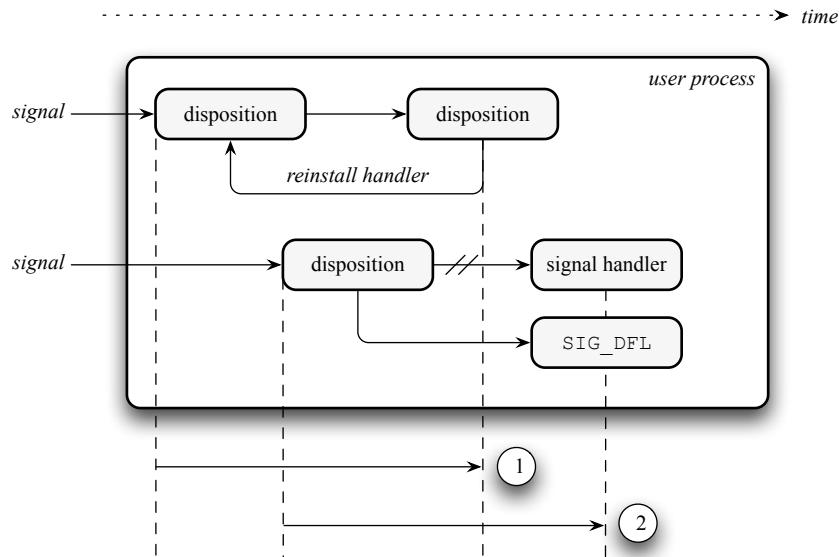


Abbildung 8.1: Zeitlicher Zusammenhang zwischen Signalbehandlung und Wiederherstellung der Signaldisposition.

8.4 Signale senden

Das Senden ist neben dem Abfangen von Signalen eine der wichtigsten (und ebenso primitiven) Kommunikationsformen zwischen Prozessen. Drei Funktionen sind dafür zuständig:

`raise`

Sendet ein Signal an den aktuellen Prozess, was dem Aufruf von `kill(getpid(), signal)` entspricht (Abschnitt 8.4.1).

`kill`

Sendet ein Signal an einen Prozess oder eine Prozessgruppe. Alternativ kann ein bestimmtes Signal auch an alle Prozesse einer Prozessgruppe gesendet werden (Abschnitt 8.4.2).

`alarm`

Richtet einen Timer für das Senden des Alarm-Signals ein. Beispielsweise könnte festgelegt werden, daß ein bestimmter Prozess in 5 Sekunden das `SIGALRM`-Signal erhält (Abschnitt 8.4.3).

8.4.1 Die Funktion `raise`

Mit der Funktion `raise` senden wir ein Signal an den ausführenden Prozess.

```
#include <signal.h>

int raise(int sig);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler.

`sig`

Signal, daß an den aufrufenden Prozess gesendet werden soll.

Wenn ein Signal Handler aufgerufen wird, kehrt `raise(3)` erst dann zurück, wenn der Handler mit seiner Ausführung fertig ist. Unterstützt die Implementierung Threads (was effektiv alle bekannten Unices tun), so entspricht der Aufruf von `raise(3)` auch:

```
pthread_kill(pthread_self(), sig);
```

wobei `pthread_self(3)` die ID des aufrufenden Threads abruft und `sig` das Signal, welches wir senden möchten repräsentiert. Das gleiche können wir auch mit der Funktion `kill(2)` erreichen:

```
kill(getpid(), sig);
```

Listing 8.3: Einsatz von `raise(2)`

```

1 #include <sys/types.h>
2 #include <signal.h>
3 #include "header.h"
4
5 static void signal_handler(int signal);
6 static void signal_alarm(int signal);
7
8 int main(void) {
9     int x = 0, y;
10
11     if (signal(SIGFPE, signal_handler) == SIG_ERR)
12         err_fatal("signal() failed");
13
14     y = 1 / x; /* causes SIGFPE */
15
16     exit(-1);
17 }
18
19 static void signal_handler(int signal) {
20     fprintf(stderr, "Oops! Caught SIGFPE. Terminating...\n");
21     raise(SIGTERM);
22
23     return;
24 }
```

Listing 8.3: xcde/raise.c - Einsatz von `raise(2)`.

Wir erzwingen eine Division durch 0, um `SIGFPE` zu provozieren und lassen die *Floating Point Exception* durch den Signal Handler behandeln. Dann senden wir uns selbst `SIGTERM`, um den Prozess aufgrund dieses schwerwiegenden Fehlers kontrolliert zu beenden. Der Signal Handler ist in einer solchen Situation der richtige Ort, um belegte Ressourcen freizugeben. □

8.4.2 Die Funktion `kill`

Die Funktion `kill(2)` kann ein Signal an einen Prozess oder eine Prozessgruppe senden.

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

Rückgabewert: 0 bei Erfolg oder -1 bei Fehler.

`pid`

ID des Prozesses oder der Prozessgruppe an die das in `sig` spezifizierte Signal gesendet werden soll.

`sig`

Signal, das an den in `pid` spezifizierten Prozess oder die Prozessgruppe gesendet werden soll.

Wenn das Argument für `sig` 0 ist, wird zwar eine Fehlerprüfung vorgenommen, doch kein Signal abgesetzt. Ein solcher Aufruf ist sinnvoll, um die Gültigkeit der in `pid` übergebenen Prozess-ID oder Prozessgruppen-ID zu prüfen (was wir uns in Listing 8.4 zu Nutze machen). Der Prozess, welcher ein Signal an `pid` senden möchte, muß über ausreichende Rechte verfügen. Hier müssen die reelle oder effektive Benutzer-ID der rellen oder set-user-ID des Empfängers entsprechen.

Das Verhalten von `kill(2)` ist von dem Wert von `pid` abhängig:

`pid > 0`

Das Signal wird dem Prozess mit der ID `pid` übermittelt.

`pid == 0`

Das Signal wird allen Prozessen übermittelt, dessen Gruppen-ID der Gruppen-ID des Senders übereinstimmt und gleichzeitig ausreichende Rechte vorhanden sind. Bestimmte Systemprozesse sind keine gültigen Empfänger solcher Signale.

`pid < 0`

Das Signal wird an alle Prozesse deren Gruppen-ID dem absoluten Wert von `pid` entspricht und für die der Sender ausreichende Rechte besitzt.

`pid == -1`

In diesem Fall wird das Signal `pid` an *alle* Prozesse gesendet, für die der Sender über ausreichende Rechte verfügt.

Eine besondere Situation tritt ein, wenn ein Signal zugestellt werden soll, `sig` aber nicht durch den Sender geblockt wird, bzw. ein anderer Thread die Blockierung von `sig` aufgehoben hat oder durch `sigwait` auf `sig` wartet. Dann wird entweder `sig` oder ein ausstehendes, nicht blockiertes Signal an den sendenden Thread ausgeliefert, bevor `kill(2)` zurückkehrt. Das entspricht dem Verhalten von `raise(2)`.

Die beschriebene Prüfung der Rechte wird nicht für Signale vom Typ `SIGCONT` durchgeführt, solange der Empfänger der gleichen Sitzungsgruppe wie der Sender angehört.

System V und andere Implementierungen wie Version 7 und 4.3BSD weisen eine unterschiedliche Semantik in der Rechteprüfung auf. POSIX hat sich der Implementierung von System V angeschlossen, d. h. ein Prozess, der das set-user-ID-Bit gesetzt hat, kann sich selbst nicht gegen Signale (oder zumindest nicht gegen `SIGKILL`) absichern, solange der Prozess seine reelle Benutzer-ID ändert. Das soll Benutzern, die eine Applikation starten, helfen, ihr Signale zu senden auch wenn die effektive Benutzer-ID geändert wurde. So wird der Applikation selbst mehr Schutz vor anderen Benutzerprogrammen geboten.

Listing 8.4: Einsatz von `kill(2)`

In diesem Beispiel senden wir einem Child-Prozess das Signal `SIGTERM`.

```

1 #include <sys/wait.h>
2 #include <signal.h>
3 #include "header.h"
4
5 int main(void) {
6     pid_t pid;
7     int status;
8
9     if ((pid = fork()) < 0)
10         err_fatal("fork() failed");
11
12     if (pid == 0) /* child code */
13         while (1) pause();
14
15     if (pid > 0) {
16         sleep(1); /* to avoid race condition */
17

```

```

18     if (kill(pid, 0) < 0)
19         err_fatal("kill() failed");
20     else
21         fprintf(stderr, "OK, child is running.\n");
22
23     /* kill child, now! */
24     if (kill(pid, SIGTERM) < 0)
25         err_fatal("kill() failed for SIGTERM\n");
26
27     /* wait for zombie child */
28     if (waitpid(pid, &status, WNOHANG) < 0)
29         err_fatal("waitpid() failed");
30
31     if (WIFSIGNALED(status))
32         fprintf(stderr, "Child terminated by signal.\n");
33 }
34
35     return (0);
36 }
```

Listing 8.4: xcode/kill.c - Einsatz von kill(2).

Der Child-Prozess wird schlafen geschickt, damit wir ihn gezielt beenden können (sonst wäre er schon von der Bildfläche verschwunden, bevor wir ihn selbst abschießen konnten). Um eine Race-Kondition zu vermeiden, warten wir im Parent eine Sekunde bevor wir mit `kill(pid, 0)` prüfen, ob der Child-Prozess läuft. Nun wird `kill(2)` mit dem Signal `SIGTERM` aufgerufen. Zwar wird der Child-Prozess beendet, doch wahrscheinlich wäre der Parent schneller mit der Ausführung fertig, als der Child-Prozess, so daß wir auf ihn warten müssen, um zu verhindern, daß `pid` als Zombie endet. □

8.4.3 Die Funktion alarm

Mit Hilfe der Funktion `alarm(3)` können Sie ein `SIGALRM`-Signal senden, nachdem ein in Sekunden angegebener Zeitraum verstrichen ist.

```
#include <unistd.h>

unsigned alarm(unsigned seconds);
```

Rückgabewert: 0 oder Anzahl der Sekunden bis zum zuvor gesetzten Alarm.

`seconds`

Anzahl der Sekunden, bis das Signal `SIGALRM` abgesetzt werden soll.

Die Funktion `alarm(3)` soll eine Anzahl von Sekunden (gemessen in Echtzeit) verstreichen lassen und dann das Signal `SIGALRM` senden. Der Scheduler des Kernels kann die Auslösung des Signals verzögern. Somit wird es möglicherweise nicht zu dem Zeitpunkt, an dem es ausgelöst wurde, verarbeitet.

Wenn `fork(2)` aufgerufen wird, werden alle ausstehenden Alarne im Kindsprozess verworfen. Ein *Process Image*, welches durch eine der `exec(3)`-Funktionen erzeugt wurde, erbt die verbleibenden Sekunden bis zur Auslösung des Signals des ursprünglichen Process Images.

Entwickler sollten beachten, daß der Rückgabewert von `alarm(3)` den Typ `unsigned` aufweist. Somit dürfen Sie, um streng POSIX-kompatibel zu bleiben, keine Werte übergeben die das garantierte Maximum von `UINT_MAX`, welches ANSI C mit 65535 festsetzt, überschreiten.

Listing 8.5: Einsatz von alarm(3)

```

1 #include <signal.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include "header.h"
5
6 void signal_handler(int sig);
7
8 static int pid;
9
10 int main(void) {
11     int status;
12
13     if (signal(SIGALRM, signal_handler) < 0)
14         err_fatal("signal() failed");
15
16     if ((pid = fork()) < 0)
17         err_fatal("fork() failed");
18
19     if (pid == 0) {
20         printf("Child loopin' around...");
21
22         while (1) {
23             printf("\nand again...");
24             sleep(1);
25         }
26     }
27
28     if (pid > 0) {
29         alarm(5);
30
31         if (waitpid(pid, &status, 0) < 0)
32             err_fatal("waitpid failed");
33
34         return (0);
35     }
36 }
37
38 void signal_handler(int sig) {
39     printf "...OK, that's enough! Bye!\n";
40
41     kill(pid, SIGTERM);
42 }
```

Listing 8.5: xcde/alarm.c - Einsatz von alarm(3).

Wir erzeugen einen Child-Prozess, der ständig eine Nachricht ausgibt und jedesmal für eine Sekunde schlafen geht. Der Parent teilt einen Alarm ein und konfiguriert den Signal Handler, der das nach fünf Sekunden eintreffende SIGALRM verarbeitet und den Child-Prozess kontrolliert beendet. Auch in diesem Beispiel gilt: `wait(2)` im Parent nicht vergessen! □

8.4.4 Die Funktion abort

Die `abort(3)`-Funktion signalisiert eine unplanmäßige Beendigung eines Programms. Unplanmäßig heißt hier nur, daß wir auf ein unplanmäßiges Ereignis reagieren wollen, welches uns nicht ermöglicht, mit der Ausführung des Programms fortzufahren.

```
#include <stdlib.h>

void abort(void);
```

Rückgabewert: kehrt niemals zurück.

Im Grunde macht `abort(3)` nichts weiter, als dem Prozess das Signal `SIGABRT` zu senden. Dieser sollte das Signal niemals ignorieren. Diese Funktion kehrt normalerweise nicht zurück. Nach ANSI C kehrt sie auch dann nicht zurück, wenn der mit `SIGABRT` assoziierte Handler zurückkehrt. Der Handler kann selbst nicht zurückkehren, wenn er `exit(2)`, `_exit(2)`, `longjmp(3)` oder `siglongjmp(3)` aufruft. POSIX geht sogar soweit, daß `abort(3)` alle Einstellungen, die das Signal blockieren oder ignorieren, übergehen werden sollten.

Der Hauptgrund für die Einführung eines solchen Systemaufrufs ist die Notwendigkeit, gewissen Aufgaben kontrolliert erledigen zu können, bevor der Prozess beendet wird. Im Signal Handler könnten beispielsweise die offenen File Descriptors geschlossen werden. Wenn sich der Prozess nicht selbst im Handler beendet, soll `abort(3)` das erledigen, sobald der Handler zurückkehrt.

Erledigt der Handler die notwendigen Aufräumarbeiten nicht selbst, so wird das System alle offenen Stream leeren (`flush`) und über `fclose(2)` schließen, wenn `abort(3)` den Prozess beendet. Tut es das nicht, verbleiben die Streams in der Regel in ihrem Zustand und sind undefiniert.

Listing 8.6: POSIX-konforme Implementierung des `abort(3)`-Systemaufrufs

```

1 #include <sys	signal.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include "header.h"
5
6 void abort(void) {
7     sigset_t          signal_mask;
8     struct sigaction sa;
9
10    sigaction(SIGABRT, NULL, &sa); /* reset disposition */
11
12    if (sa.sa_handler == SIG_IGN) { /* BAD BOY: SIGABRT should */
13        sa.sa_handler = SIG_DFL; /* never be ignored -- DFL */
14        sigaction(SIGABRT, &sa, NULL);
15    } else if (sa.sa_handler == SIG_DFL) {
16        fflush(NULL); /* NULL --> all open streams */
17    }
18
19    /* SIGABRT must neither be blocked nor ignored */
20    sigfillset(&signal_mask);
21    sigdelset(&signal_mask, SIGABRT); /* make sure it's handled */
22    sigprocmask(SIG_SETMASK, &signal_mask, (sigset_t *)NULL); /* OK, done!
   */
23
24    /* everything set up -- send the signal */
25    pthread_kill(pthread_self(), SIGABRT);
26
27    /* OK, the signal handler returned -- flush again */
28    fflush(NULL);
29
30    /* sa might have been altered, it shouldn't though */
31    sa.sa_handler = SIG_DFL;
32
33    /* restore disposition -- don't forget to call sigprocmask */
34    sigaction(SIGABRT, &sa, NULL);
35    sigprocmask(SIG_SETMASK, &signal_mask, (sigset_t *)0);
36
37    /* OK, try again -- this time we should not return... */
38    pthread_kill(pthread_self(), SIGABRT);
39
40    /* ...but if we do, give up */

```

```

41     exit(-1);
42 }

```

Listing 8.6: xcode/abortimpl.c - POSIX-konforme Implementierung des abort(3)-Systemaufrufs.

Wir müssen zunächst sicher sein, daß SIGABRT nicht ignoriert wird. Dazu setzen wir die Disposition des Signals zurück und installieren den Handler mit der Standardeinstellung (SIG_DFL). Damit wir uns nicht mit anderen Signalen herumschlagen müssen, lassen wir alle bis auf SIGABRT draußen (sigdelset(3)) und wenden die neue Maske an. Nun da wir vorbereitet sind, setzen wir das Signal ab und sehen nach, was passiert. Wenn wir wieder in `abort(3)` zurückkehren, ist das Signal behandelt worden, der Handler des Prozesses ebenfalls zurückgekehrt und überläßt also uns nun die Aufgabe, alle Streams ordnungsgemäß zu schließen. Wir dürfen nicht vergessen, die Signaldisposition zurückzusetzen, damit wir beim zweiten Versuch erfolgreich sind. □

8.5 Auf Signale warten

Signale sollen verhindern, daß das Warten auf bestimmte Ereignisse zu lasten der Rechenzeit geht, was beispielsweise der Fall ist, wenn wir eine `while`-Schleife verwenden, um den Zustand eines Indikators abzufragen. Naturgemäß wird die gesamte verfügbare Rechenzeit dafür aufgewendet. Signale sind da eine weitaus elegantere und wirtschaftlichere Möglichkeit, das gleiche zu erreichen.

Zwei Funktionen sind für diesen Zweck von Interesse:

pause(3)

Hält den aufrufenden Prozess oder Thread an, bis ein Signal eintrifft. Entweder führt das Signal zur Terminierung des Prozesses oder es sorgt für den Aufruf eines Signal Handlers.

sigsuspend

Ersetzt die aktuelle Signalmaske des Prozesses mit einer anderen und setzt die Ausführung dieses Prozesses aus, bis ein Signal empfangen wird.

8.5.1 Die Funktion pause

Manchmal ist es notwendig, einen Thread bis zum Eintreffen irgendeines Signals auszusetzen. Das erledigen wir mit `pause(3)`:

```
#include <unistd.h>

int pause(void);
```

Rückgabewert: -1 wenn ein Signal Handler zurückkehrt.

Beim Aufruf von `pause(3)` wird der aufrufende Prozess ausgesetzt, bis entweder ein Signal Handler ausgeführt oder der Prozess terminiert wird. Ist ersteres der Fall, kehrt `pause(3)` nach Ausführung des Signal Handlers zurück. Trifft letzteres zu, kehrt `pause(3)` gar nicht zurück.

Listing 8.7: Beispielprogramm für pause(3).

```

1 #include <signal.h>
2 #include "header.h"
3
4 void signal_handler(int);
5 void print_time(char *);
6
7 int main(int argc, char **argv) {
8     struct sigaction sigact;
```

```

9      sigemptyset(&sigact.sa_mask);
10     sigact.sa_flags = 0;
11     sigact.sa_handler = signal_handler;
12     sigaction(SIGALRM, &sigact, NULL);
13
14     alarm(5); /* fire SIGALRM in 5 seconds */
15
16     print_time("before pause");
17     pause();
18     print_time("after pause");
19
20     return (0);
21 }
22
23
24 void signal_handler(int sig_num) {
25     printf("Signal %d caught\n", sig_num);
26
27     return;
28 }
29
30 void print_time(char *msg) {
31     time_t t;
32
33     time(&t);
34     printf("Time %s: %s\n", msg, ctime(&t));
35 }
```

Listing 8.7: xcode/pause.c - Beispielprogramm für pause(3).



Oftmals wird `pause(3)` eingesetzt, um Timings zu steuern. Solche Szenarien basieren auf der Prüfung einer Bedingung in Verbindung mit einem Signal und wenn es nicht zutrifft, wird `pause(3)` aufgerufen. Wenn das Signal während der Prüfung und dem Aufruf von `pause(3)` auftritt, kann es passieren, daß der Prozess unendlich lange im Zustand Blocking verweilt. Mit den Funktionen `sigprocmask(2)` und `setsigmask(2)` kann dieses Problem vermieden werden.

Listing 8.8: POSIX-konforme Implementierung der pause(3)-Funktion

```

1 #include <errno.h>
2 #include <signal.h>
3 #include <unistd.h>
4
5 int pause(void) {
6     sigset(SIG_BLOCK, &set);
7
8     sigemptyset(&set);
9     sigprocmask(SIG_BLOCK, NULL, &set);
10
11     return setsigmask(&set); /* for errno */
12 }
```

Listing 8.8: xcode/pauseimpl.c - POSIX-konforme Implementierung der pause(3)-Funktion.



8.5.2 Die Funktion `sigsuspend`

Wenn `pause(3)` aufgerufen wird, während das der Thread das Signal blockiert, kommt das Signal niemals an und `pause(3)` kehrt nicht zurück. Angenommen wir würden ein Signal vor dem Aufruf von `pause(3)`

„unblocken“, dann könnte das Signal eintreffen, während die Blockierung aufgehoben und `pause(3)` ausgeführt wird. Wenn ein Signal eintrifft, während es blockiert ist, erhält der Prozess das Signal direkt nachdem die Blockierung aufgehoben wurde.

Das soeben beschriebene Problem ist sehr alt. Eine Möglichkeit wäre, die Blockierung des Signals aufzuheben und `pause(3)` aufzurufen, ohne daß dieser Vorgang unterbrochen werden kann. Beide Vorgänge müßten also in einer atomaren Operation ablaufen. Die Funktion `sigsuspend(2)` erledigt das für uns.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Rückgabewerte: kehrt nicht zurück, aber bei Fehler wird -1 zurückgegeben.

sigmask

Signalmaske, mit der ein Prozess ausgestattet und anschließend pausiert werden soll.

Der Aufruf von `sigsuspend(2)` setzt die Ausführung des Prozesses aus, bis ein Signal eintrifft, das entweder die Terminierung des Prozesses oder die Ausführung eines Signal Handlers zur Folge hat. Die Funktion kehrt nicht zurück, wenn ein Signal den Prozess beendet. Soll aber ein Signal Handler ausgeführt werden, so stellt `sigsuspend(2)` nach Ausführung des Signal Handlers die ursprüngliche Signalmaske wieder her. Wir greifen die Problematik nocheinmal in Abschnitt 8.7.2 auf. Signalsätze besprechen wir im nächsten Abschnitt.

8.6 Arbeiten mit Signalsätzen

Bisher haben wir nur die Verarbeitung einzelner Signale besprochen. Mehrere Signale verwalten wir mit sog. Signalsätzen (*signal sets*). Sie kommen insbesondere mit der Funktion `sigprocmask(2)` zum Einsatz, um dem Kernel mitzuteilen, wie mit bestimmten Signalen umgegangen werden soll. POSIX definiert einen neuen Datentyp für die Verwaltung von Signalsätzen: `sigset_t`.

Jeder Prozess verfügt über eine *Signalmaske*, die anzeigt, welche Signale blockiert und welche empfangen werden sollen. Alle Signale, die in der Maske enthalten sind, werden für diesen Prozess geblockt. Ausnahmen bilden wie immer `SIGKILL` und `SIGSTOP`, die weder ignoriert noch blockiert werden können. Ein Signalsatz, den wir einer Prozessmaske zuordnen, kann mit Hilfe der Funktionen `sigemptyset(3)`, `sigfillset(3)`, `sigaddset(3)` und `sigdelset(3)` modifiziert werden. Abbildung 8.2 illustriert, welchen Einfluß die Funktionen auf den Signalsatz haben.

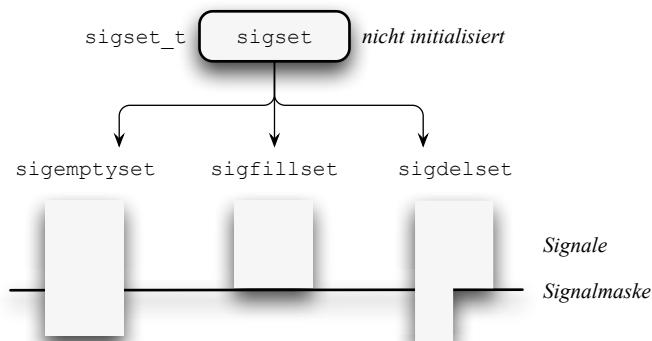


Abbildung 8.2: Einfluß von `sigemptyset(3)`, `sigfillset(3)` und `sigdelset(3)` auf die Signalmaske.

Die Funktion `sigemptyset(3)` schließt alle Signale aus der Signalmaske aus (jedes Signal wird dem Prozess zugestellt), während `sigfillset(3)` alle Signale in die Signalmaske aufnimmt, so daß sie alle von

dem Prozess blockiert werden. `sigdelset(3)` entfernt einzelne Signale aus der Maske, die dann entweder gemäß ihrer Standardaktion (`SIG_DFL`) oder durch einen Signal Handler verarbeitet werden. Den Typ `sigset_t` kommen wir im Abschnitt 8.7 im Zusammenhang mit der zuverlässigen Signalbehandlung noch einmal zu sprechen.

8.6.1 Die Funktion `sigset`

Als wir die `signal(2)`-Funktion besprochen haben wurde ein großer Nachteil deutlich: es gibt nur eine Möglichkeit, einen Prozess gegen bestimmte Signale immun zu machen. Alle anderen Signale müssen ignoriert werden, wenn der Prozess nicht auf sie reagieren möchte. Das bedeutet aber auch, daß er das Auftreten derselben vollständig versäumt. Manchmal ist es aber wünschenswert, die Auslieferung eines Signals zu verzögern, bis der Prozess wieder in der Lage ist, es zu behandeln. Aus diesem Grund wurde bereits mit System V Release 3 die `sigset(3)`-Funktion eingeführt. Obwohl der POSIX-Standard von der Anwendung von `sigset(3)` abrät, da die Funktion `sigaction(2)` (Abschnitt 8.7.1 auf Seite 224) zuverlässigere Mechanismen zur Kontrolle von Signalen aufweist, möchte wir dennoch einen genaueren Blick darauf werfen, da es noch viele Programme gibt, die mit der `sigset(3)`-Semantik arbeiten.

```
#include <signal.h>

void (*sigset(int sig, void (*disp)(int)))(int);
```

Rückgabewerte: Die letzte Signaldisposition oder `SIG_ERR` bei Fehler.

sig

Spezifiziert das Signal, das jedes außer `SIGKILL` oder `SIGSTOP` sein darf.

disp

Gibt die Disposition an, die entweder als `SIG_IGN` oder `SIG_DFL` bzw. Funktionsszeiger konfiguriert werden darf.

Wie unschwer zu erkennen ist, gleicht die Synopsis von `sigset(3)` der von `signal(2)`, doch die Funktion hat weit mehr zu bieten. So kann `disp` auch `SIG_HOLD` sein, durch die die vorhandene Disposition nicht verändert, aber zur **Signalmaske** des Prozesses hinzugefügt wird. Solange ein Signal durch die Signalmaske blockiert ist, wird es beim Eintreffen zwar gehalten, aber nicht dem Prozess zugestellt. Es bleibt im Zustand *pending*, bis das betreffende Signal von der Signalmaske entfernt worden ist. Erst dann werden ausstehende Signale auch zugestellt. Wir müssen mit unserer Wortwahl vorsichtig sein. Ein zugestelltes (*posted*) Signal ist für den Prozess nicht sichtbar; erst wenn es ausgeliefert (*delivered*) wird, nimmt es Einfluß auf den Prozess. In der Zeit zwischen Zustellung und Auslieferung ist das Signal ausstehend.

Die Funktion `sigset(3)` ist zusammen mit `sighold(3)`, `sigignore(3)`, `sigpause(3)` und `sigrelse` als Teil der *X/Open System Interface Extension* (XSI) aufgenommen worden und sollte ursprünglich die Signalbehandlung vereinfachen. Mit dem Einzug von `sigaction(2)` ist diese Funktionsfamilie auch schon wieder obsolet.

Die Funktion `sigset(3)` kann unterschiedliche Werte zurückgeben. Bei Erfolg wird entweder `SIG_HOLD` zurückgegeben, sofern `sig` zuvor gehalten wurde, oder sie gibt die letzte Disposition des Prozesses zurück, wie wir es bei `signal(2)` gesehen haben.

Neu ist außerdem ein kleines, aber wichtiges Detail. Bei Aufruf des Signal Handlers wird die Signaldisposition nicht verändert. Vielmehr wird vor Aufruf des Handlers die Signalmaske des Prozesses geändert, wenn ein Signal ausgeliefert wird. Kehrt der Signal Handler wieder zurück, wird die Signalmaske wieder in den ursprünglichen Zustand zurückversetzt. Diese Besonderheit verhindert die erneute Installation des Signal Handlers innerhalb des Handlers wie in Beispiel 8.2 gezeigt wurde und eliminiert das unsägliche Zeitfenster.

Listing 8.9: Zuverlässige Signalauslieferung mit `sigset(3)`

Um ein besseres Verständnis für die Arbeit mit `sigset(3)` zu erhalten, betrachten wir dazu ein kleines Beispiel. Es entspricht weitgehend dem aus Listing 8.2, jedoch verwenden wir dieses mal `sigset(3)`.

```

1 #include <signal.h>
2 #include "header.h"
3
4 int main(void) {
5     int i;
6     void handler(int);
7
8     /* install signal handler */
9     sigset(SIGINT, handler);
10
11    /* pause 5 times; interrupted by signals */
12    for (i = 0; i < 5; i++)
13        pause();
14
15    return (0);
16 }
17
18 void handler(int sig_num) {
19     printf("Caught SIGINT\n");
20     sleep(1); /* create a timing window */
21 }
```

Listing 8.9: xcode/sigset.c - Sichere Signalauslieferung mit sigset(3).

Wenn wir das Programm ausführen, erhalten wir genau fünf Meldungen, egal wie schnell wir CTRL-C drücken. Drücken wir die Tastenkombination mehrmals pro Sekunden, erhalten wir eine Nachricht und eine Sekunde später die nächste. Das liegt daran, daß das Signal gehalten wird während der Handler mit der Behandlung des ersten beschäftigt ist.

8.6.2 Das sigset-Interface

Zusammen mit der Einführung der `sigset(3)`-Funktion fanden auch einige zusätzliche Funktionen zur Verwaltung der Signale Einzug in SVR3:

```
#include <signal.h>

int sighold(int sig);
int sigignore(int sig);
int sigpause(int sig);
int sigrelse(int sig);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

`sighold` fügt das spezifizierte Signal der Signalmaske des Prozesses hinzu. Mit `sigrelease(3)` wird das Signal aus der Signalmaske entfernt. Die Funktion `sigignore(3)` setzt die Disposition auf den Zustand `SIG_IGN`. Mit `sigpause(3)` wird das Signal aus der Signalmaske entfernt und der Prozess veranlaßt, die Ausführung solange auszusetzen, bis irgendein Signal eintrifft.

Die vier genannten Funktionen sind Teil der *X/Open System Interface Extension* (XSI), das eine Obermenge von POSIX darstellt. Sie sind nicht durch den Standard definiert, aber Bestandteil der *Single UNIX Specification* (SUS).

Oftmals werden Signal Handler dazu verwendet, globale Datenstrukturen eines Programms zu verändern, die auch von dem Ausführungspfad des Programms manipuliert werden. Das kann zu Problemen führen, denn der Zustand der Strukturen kann dabei beschädigt werden, weil die Ausführungspfad von Signalen unterbrochen wird. Regionen, die Datenstrukturen asynchron verändern, werden als *kritische Regionen* (siehe Abschnitt 10.5.1) bezeichnet. Diese Code-Abschnitte müssen sicher abgearbeitet werden, ohne

von Signalen unterbrochen zu werden, die die Strukturen in einem inkonsistenten Zustand zurücklassen könnten. Diese Problematik betrachten wir im nächsten Beispiel.

Listing 8.10: Beispiel zur Demonstration kritischer Regionen

```

1 #include <sys/types.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <signal.h>
6
7 #define LIST_SIZE 10
8
9 struct list {
10     struct list *next;
11     int         value;
12 } list[LIST_SIZE];
13
14 struct list *head;
15
16 void signal_handler(int sig_num);
17 void print_list(int number);
18
19 int main(int argc, char **argv) {
20     int i, n;
21
22     if (argc == 2)
23         n = atoi(argv[1]);
24     else
25         n = 0;
26
27     /* install signal handler for SIGINT */
28     sigset(SIGINT, signal_handler);
29
30     /* create a linked list */
31     for (i = LIST_SIZE - 1; i >= 0; i--) {
32         list[i].value = i + 1;
33         list[i].next = head;
34         head = &list[i];
35     }
36
37     /* print the list, generating SIGINT when reaching element n */
38     print_list(n);
39
40     /* print list without generating signals */
41     print_list(0);
42
43     return (0);
44 }
45
46 void print_list(int number) {
47     struct list *plist;
48
49     /* print the value of each element in the list */
50     for (plist = head; plist != NULL; plist = plist->next) {
51         printf("%d ", plist->value);
52
53         /* if n-th element is reached generate signal */
54         if (plist->value == number)
55             kill(getpid(), SIGINT);
56     }

```

```

57     printf("\n");
58 }
59
60 void signal_handler(int sig_num) {
61     struct list *plist, *olist;
62
63     /* search the list and remove element 5 */
64     olist = NULL;
65     for (plist = head; plist != NULL; plist = plist->next) {
66         if (plist->value == 5) {
67             if (olist != NULL)
68                 olist->next = plist->next;
69             else
70                 plist->next = NULL;
71
72             break;
73         }
74
75         olist = plist;
76     }
77 }
```

Listing 8.10: xcode/signalregion.c - Beispiel zur Demonstration kritischer Regionen.

Die `printlist`-Funktion durchläuft die verkettete Liste und gibt jedes Element aus. Wenn wir ein bestimmtes Element erreichen, setzen wir das Signal SIGINT ab und machen mit der Ausgabe weiter. Der Signal Handler durchläuft ebenfalls die Liste und entfernt ein Element aus der Liste. Eine beispielhafte Bildschirmausgabe könnte etwa so aussehen:

```
% ./signalregion
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
% ./signalregion 4
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
% ./signalregion 5
1 2 3 4 5
1 2 3 4 5 6 7 8 9 10
% ./signalregion 6
1 2 3 4 5 6 7 8 9 10
1 2 3 4 6 7 8 9 10
```

Die Ausgabe ist recht aufschlußreich. Geben wir bei der ersten Ausführung des Programms kein Argument an, so wird die Liste zweimal vollständig ausgegeben. Beim zweiten Versuch setzen wir bei Erreichen von Element 4 ein Signal ab und sind in der Lage, auch dieses mal die Liste vollständig zu durchlaufen. Jetzt wird es interessant. Mit dem Argument 5 wird ein Signal beim Erreichen von Element 5 abgesetzt. Der Handler setzt ein und setzt den `next`-Zeiger auf `NULL` wenn das Element aus der Liste entfernt wird. Doch leider ist der Ausführungspfad in `printlist` gerade damit beschäftigt den `next`-Zeiger auszuwerten und nimmt fälschlicherweise an, daß die Liste zu Ende ist. Die zweite Ausgabe der Liste zeigt aber, daß sie in Takt ist. Durch eine Ausgabe der Liste mit Argument 6 sehen wir, daß die List vollständig bis auf das 6. Element ($n - 1$) ausgegeben wird, was auch in unserem Sinn ist, denn wir sind ja bereits über Element 5 hinausgegangen.

Das Problem kann (zumindest teilweise) gelöst werden, wenn wir beim Eintritt in die Funktion `printlist` am Anfang die Zeilen

```
...
void (*function)(int);
function = sigset(SIGINT, SIG_IGN);
...
```

eintragen, um das Auftreten von `SIGINT` zu ignorieren (`SIG_IGN`). Wir dürfen natürlich nicht vergessen, hinterher die ursprüngliche Disposition wiederherzustellen:

```
...
sigset(SIGINT, function);
...
```

Jetzt haben wir aber nicht genau das erreicht, was wir eigentlich möchten, denn das Signal wird nun vollständig ignoriert und die Liste wird immer, unabhängig von dem an das Programm übergebene Argument, vollständig ausgegeben. Eine bessere Lösung wäre, das Signal während der Ausgabe der Liste einfach nur zu halten. Dazu fügen wir einfach die Zeile

```
...
sighold(SIGINT);
...
```

am Anfang der Funktion ein und heben das Blockieren des Signals hinterher wieder auf:

```
...
sigrelse(SIGINT);
...
```

Der vollständige Code der `printlist`-Funktion würde dann so aussehen:

```
void print_list(int number) {
    struct list *plist;

    sighold(SIGINT); /* hold (block) signal */

    /* print the value of each element in the list */
    for (plist = head; plist; plist = plist->next) {
        printf("%d ", plist->value);

        /* if n-th element is reached generate signal */
        if (plist->value == number)
            kill(getpid(), SIGINT);
    }

    sigrelse(SIGINT); /* release signal */
    printf("\n");
}
```

□

8.6.3 Signalsätze bearbeiten

Die dritte Signalschnittstelle auf die wir zu sprechen kommen ist Bestandteil des POSIX-Standards. Sie wurde entworfen, die Nachteile, die beim Blockieren mehrerer Signale entstehen zu umgehen und um die Anzahl der definierbaren Signale zu erhöhen, da die Obergrenze fast schon erreicht wurde. Ursprünglich hat 4.2BSD das erste Problem gelöst, während System V keines der beiden Probleme lösen konnte. Die POSIX-Variante bildet einen gemeinsamen Nenner und bietet Zugang zu beiden Schnittstellen.

Mit den folgenden fünf Funktionen nehmen wir auf vielfältige Weise Einfluß auf die Signalsätze:

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

```
int sigismember(const sigset_t *set, int signum);
```

Rückgabewerte: 1 wenn true, 0 wenn false.

set

Signalsatz, auf den die Operation angewendet werden soll.

signum

Signal, das aus dem in **set** definierten Satz entfernt oder hinzugefügt werden soll.

Mit **sigemptyset(3)** initialisieren wir den Satz, auf den **set** zeigt, mit leeren Signalen. Dadurch schließen wir alle Signale aus. Die Funktion **sigfillset(3)** bewirkt das Gegenteil. Es werden alle Signale zum in **set** definierten Signalssatz hinzugefügt. Um ein individuelles Signal zu einem existierenden Satz hinzuzufügen, rufen wir **sigaddset(3)** auf und nennen als erstes Argument die Adresse des Signalsatzes und als zweites das jeweilige Signal, das wir hinzufügen wollen. Um es wieder aus dem Satz zu entfernen, steht uns **sigdelset(3)** zur Verfügung.

Die beiden erstgenannten Funktionen **sigemptyset(3)** und **sigfillset(3)** könnten als Makros implementiert werden:

```
#define sigemptyset(set) (*(set) = 0)
#define sigfillset(set)   (*(set) = ~(sigset_t)0, 0)
```

oder, wie es viele POSIX-kompatiblen UNICES praktizieren, als einfache Funktionen, etwa so:

```
int sigemptyset(sigset_t *set) {
    if (set == NULL) {
        errno = EINVAL;
        return -1;
    }

    memset(set, 0, sizeof(sigset_t));
    return 0;
}

int sigfillset(sigset_t *set) {
    if (set == NULL) {
        errno = EINVAL;
        return -1;
    }

    memset(set, 0xff, sizeof(sigset_t));
    return 0;
}
```

Die Implementierung der drei anderen Hilfsfunktionen **sigaddset(3)**, **sigdelset(3)** und **sigismember(3)** ist ebenso einfach, wie folgende Beispielimplementierungen zeigen:

```
#define SBITS      (sizeof (unsigned long int) * 8)
#define SSELECT(s)  ((s) / SBITS)
#define SSMASK(s)   (1 << ((s) % SBITS))

int sigaddset (sigset_t *set, int sig) {
    if (set == NULL || sig <= 0 || sig >= NSIG) {
        errno = EINVAL;
        return -1;
    }
```

```

    set->sigbits[SSELECT(signo)] |= SSMASK (signo);
    return (0)
}

int sigdelset (sigset_t *set, int sig) {
    if (set == NULL || sig <= 0 || sig >= NSIG) {
        errno == EINVAL;
        return -1;
    }

    set->sigbits[SSELECT(sig)] &= ~SSMASK (sig);
    return 0;
}

int sigismember (const sigset_t *set, int sig) {
    if (set == NULL || sig <= 0 || sig >= NSIG) {
        errno == EINVAL;
        return -1;
    }

    if (set->sigbits[SSELECT(sig)] & SSMASK (sig))
        return 1;

    return 0;
}

```

Auch wenn auf den ersten Blick Makros günstiger wären, verlangt POSIX die Überprüfung der Gültigkeit der Argumente, was mit Funktionen am einfachsten realisiert werden kann. Übrigens ist NSIG als 64, die größtmögliche Signumnummer + 1, in `<bits/signum.h>` definiert.

Die Funktion `sigpending(2)` hat folgende Synopsis:

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

Mit `sigpending(2)` fragen wir jene Signale ab, die zwar zugestellt aber noch nicht ausgeliefert werden. Sie befinden sich sozusagen im Zustand *pending*. Das Resultat wird in `set` gespeichert und kann hinterher beispielsweise als Argument für `sigismember(3)` verwendet werden.

```
#include <signal.h>

int sigismember(sigset_t *set, int signo);
```

Rückgabewerte: 0 (false) wenn `signo` nicht in `set` enthalten ist, andernfalls 1 (true).

Immer wenn wir wissen möchten, ob ein bestimmtes Signal in einem Signalsatz enthalten ist, befragen wir `sigismember(3)` und übergeben den zu untersuchenden Signalsatz mit `set` und mit `signo` das Signal, von dem wir wissen wollen, ob es in `set` enthalten ist.

8.6.4 Die Funktion `sigprocmask`

Eine Signalmaske stellt einen Satz von Signalen dar, die momentan für den Prozess nicht ausgeliefert werden. Die Maske verändern wir mit den zuvor besprochenen Funktionen. Sie blockiert die Signale nur, verwirft oder ignoriert sie nicht. Ein Prozess kann die Signalmaske mit `sigprocmask(2)` abfragen, verändern oder beides.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

how

Bestimmt, wie die aktuelle Signalmaske verändert werden soll.

set

Stellt ein Signalsatz dar, mit dem die aktuelle Signalmaske verändert werden soll.

oset

Speichert die vorangegangene Signalmaske.

Wenn *oset* kein NULL-Zeiger ist, wird die Signalmaske hier abgespeichert. Ist *set* kein NULL-Zeiger, wird die Maske auf die *set* zeigt, zur Modifikation der aktuellen Signalmaske verwendet und *how* gibt dabei an, wie sie verändert werden soll.

SIG_BLOCK

Die neue Signalmaske für den Prozess ist eine Vereinigung aus der vorangegangenen und der auf die *set* zeigt. Set beinhaltet also die zusätzlichen Signale für die neue Signalmaske.

SIG_UNBLOCK

Aus der aktuellen Signalmaske werden die in *set* übergebenen Signale entfernt.

SIG_SETMASK

Die neue Signalmaske ist in *set* enthalten.

Wenn *set* also ein NULL-Zeiger ist, wird die Signalmaske nicht verändert und *how* ist bedeutungslos. Befinden sich noch einige Signale im Zustand Pending, nachdem *sigprocmask(2)* ausgeführt wurde, so wird mindestens eines dieser Signale ausgeliefert bevor *sigprocmask(2)* zurückkehrt.

Da sich der Zusammenhang zwischen den einzelnen Funktionen nicht auf den ersten Blick erschließt, wollen wir dazu ein kleines Beispiel heranziehen und es detailliert besprechen.

Listing 8.11: Effektive Signalbehandlung mit *sigprocmask(2)* und Co.

```

1 #include <signal.h>
2 #include "header.h"
3
4 int main(int argc, char *argv[]) {
5     sigset_t current_mask, /* this mask is our initial set of signals */
6         backup_mask, /* this one's used to restore the set */
7         pending_mask; /* a mask of pending signals */
8
9     void signal_handler(int sig_num);
10
11    /* register a handler to catch SIGINT */
12    if (signal(SIGINT, signal_handler) == SIG_ERR)
13        err_fatal("Cannot handle SIGINT\n");
14
15    /* initialize empty set (all signals are excluded) */
16    sigemptyset(&current_mask);
17
18    /* add SIGINT to new set */
19    sigaddset(&current_mask, SIGINT);
20
```

```

21     /* enable current mask, store it in backup_mask and block SIGINT */
22     if (sigprocmask(SIG_BLOCK, &current_mask, &backup_mask) < 0)
23         err_fatal("sigprocmask failed for SIG_BLOCK\n");
24
25     /* this enables us to press the CTRL-C key for 10 seconds */
26     sleep(10);
27
28     /* store set of pending signals */
29     if (sigpending(&pending_mask) < 0)
30         printf("sigpending failed\n");
31
32     /* fetch list of pending signals */
33     if (sigismember(&pending_mask, SIGINT))
34         printf("\n%s: pending signal SIGINT found\n", argv[0]);
35
36     /* restore original signal mask to unblock SIGINT */
37     if (sigprocmask(SIG_SETMASK, &backup_mask, NULL) < 0)
38         err_fatal("procmask failed for SIG_SETMASK\n");
39
40     printf("%s: SIGINT restored, isn't blocked anymore\n", argv[0]);
41
42     exit(0);
43 }
44
45 void signal_handler(int sig_num) {
46     printf("Signal SIGINT caught\n");
47 }
```

Listing 8.11: xcode/sigprocmask.c - Effektive Signalbehandlung mit sigprocmask(2) und Co.

Das Beispiel könnte folgende Ausgaben generieren:

```
% ./sigprocmask // geht fuer 10 Sekunden schlafen
^C^C^C
./sigprocmask: pending signal SIGINT found
Signal SIGINT caught
./sigprocmask: SIGINT restored, isn't blocked anymore
```

Bemerkenswert ist, daß das Signal trotz mehrfachen Drückens von STRG-C nur einmal ausgeliefert wird. Sie werden sich erinnern, daß der Kernel zwar registriert, daß ein Signal mehrfach generiert wurde, assoziiert aber keinen internen Zähler, sodaß ein Signal *immer* nur einmal ausgeliefert wird.

Besprechen wir das Beispiel, um einen besseren Einblick in die Geschehnisse zu erhalten.

6-8

Da wir unterschiedliche Operationen mit unseren Signalsätzen durchführen möchten, definieren wir zunächst drei Signalsätze: einen zur Etablierung von **SIGINT**. Dieses Signal wollen wir der Maske **current_mask** hinzufügen. Den zweiten Satz, nämlich **backup_mask**, benötigen wir als temporären Speicherplatz, wenn wir die Maske verändern möchten. Nur so sind wir in der Lage, die ursprüngliche Signalmaske am Ende wiederherzustellen. Letztendlich benötigen wir noch einen dritten Satz, der das Ergebnis aus der Abfrage von **sigpending(2)** aufnimmt: **pending_mask**.

10

Die Routine für die Signalbehandlung kennen sie bereits. Es ist eine einfache Funktion, ohne Rückgabewert und einem Integer-Argument für die Signalnummer.

13-15

Nachwievor müssen wir den alten Bekannten **signal(2)** zur Registrierung der Behandlungsroutine einsetzen. In Abschnitt 8.7.1 besprechen wir die Funktion **sigaction(2)** und die gleichnamige Struktur. Mit ihrer Hilfe sind wir in der Lage, den POSIX-Mechanismus der verlässlichen Signale zu verwenden.

18-21

Bevor wir ein Signalsatz verwenden dürfen, müssen wir ihn mit `sigemptyset(3)` initialisieren. Anschließend fügen wir dem Satz das Signal mit `sigaddset(3)` hinzu, das wir behandeln wollen.

24-26

Um die Funktionen `sigpending(2)` und `sigismember(3)` zu demonstrieren, aktivieren wir den Signalsatz nicht einfach, sondern stellen ihn auf *blocking* (`SIG_BLOCK`) ein. Wenn das nicht gewünscht ist, würde die Option `SIG_SETMASK` genügen. Um die ursprüngliche Maske, also die in `current_mask` wieder herstellen zu können, geben wir als drittes Argument in `sigprocmask(2)` den temporären Signalsatz (`backup_mask`) an.

29

Nun schicken wir den Prozess für zehn Sekunden schlafen. In dieser Zeit können wir die Tastenkombination `CTRL-C` drücken und uns von dem Resultat überzeugen. Während dieser Zeit werden die eintreffenden Signale zwar zugestellt, aber nicht ausgeliefert, sie werden eben nur blockiert.

32-34

Da wir den Prozess schlafen geschickt haben, warten die Signale auf ihre Auslieferung. Natürlich wissen wir, daß es sich dabei um `SIGINT` handelt, doch wir können die ausstehenden Signale mit `sigpending()` abfragen. Das Ergebnis finden wir in der Maske, die wir als Parameter übergeben.

32-34

Mit der Funktion `sigispending(2)` befragen wir den Prozess (hier: `pending_mask`), ob er Signale aufweist, die noch nicht ausgeliefert wurden (sich also im Zustand *pending* befinden). Dazu übergeben den Signalsatz, der das Ergebnis speichern soll, als Argument.

37-39

Nun, da wir einen Signalsatz mit den ausstehenden Signalen abgefragt haben, können wir `sigismember(3)` verwenden, um nach einem bestimmten Signal, von dem wir wissen möchten, ob es ausstehend ist, zu suchen. Das erste Argument ist der zu untersuchende Signalsatz, das zweite Argument stellt das Signal dar, welches wir in dem Satz vermuten.

42-44

Erst wenn wir das Signal `SIGINT` aus dem Zustand Blocking herausholen, können wir es behandeln. Dazu stellen wir die ursprüngliche Signalmaske wieder her, indem `sigprocmask(2)` diesmal mit der Option `SIG_SETMASK` als erstes Argument und der gespeicherten Signalmaske als zweites Argument. `sigprocmask(2)` kann die aktuelle Signalmaske speichern, was mit der Angabe eines Zeigers auf eine `sigset_t`-Variable als drittes Argument erledigt werden kann. Da wir dieses mal, aber nur eine Maske setzen möchten, ist dieses Argument `NULL`. Es stellt sich Ihnen wahrscheinlich die Frage, warum wir nicht einfach die Option `SIG_UNBLOCK` verwenden, um die Blockierung des Signals aufzuheben. Offensichtlich wird sie doch von `sigprocmask(2)` unterstützt. Der Grund dafür ist weniger offensichtlich. Wenn wir eine Funktion schreiben, die auch von anderen Prozessen aufgerufen werden kann und wir in dieser Funktion ein Signal blockieren müssen, müssen wir die Option `SIG_SETMASK` verwenden und die Signalmaske auf den Ursprungszustand zurücksetzen, denn es könnte der Aufrufer das betreffende Signal ausdrücklich blockiert haben, bevor er die Funktion aufruft. Das ist manchmal im Zusammenhang mit der Funktion `system(2)` der Fall. □

8.7 Zuverlässige Signalverarbeitung

Jetzt haben wir lange die unzuverlässige Signalbehandlung (*unreliable signals*) und die unglückliche Semantik von `signal(2)` gesprochen. Außerdem haben wir die XSI-Variante (`sigprocmask(2)` und das `sigset(3)`-Interface) der zuverlässigen Signalbehandlung kennen gelernt. Es ist an der Zeit über den offiziellen, von POSIX favorisierten Nachfolger zu sprechen.

8.7.1 Die Funktion `sigaction`

Die Funktion `sigaction(2)` erlaubt es uns, die assoziierte Aktion eines Signals zu modifizieren oder zu untersuchen (oder beides gleichzeitig). Es sei an dieser Stelle ausdrücklich angemerkt, daß mit dem

Einzug von `sigaction(2)` und der gleichnamigen Struktur die Verwendung von `signal(2)` nicht mehr notwendig ist.

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler

sig

Stellt das Signal dar, welches wir modifizieren oder untersuchen wollen.

act

Ist der Zeiger nicht NULL, wollen wir das Signal modifizieren.

oact

Ist dieser Zeiger nicht NULL, wird die ursprüngliche Aktion des Signals zurückgeliefert.

Die Struktur `sigaction` weist folgende Member auf:

```
struct sigaction {
    void (*sa_handler)(int); /* pointer to handler or SIG_IGN/SIG_DFL */
    sigset_t sa_mask;        /* set of signals to be blocked */
    int     sa_flags;         /* flags to affect behavior of signal */
}
```

Wenn das `SA_SIGINFO`-Flag in `sa_flags` nicht gesetzt ist, entspricht das `sa_handler`-Feld die mit dem Signal assoziierte Aktion. Ist es auf einem System, das Signale in Echtzeit ausliefern kann (das bezieht sich auf die Erweiterung durch die X/Open Group) gesetzt, spezifiziert das Feld `sa_handler` die Funktion zur Signalbehandlung. Gleichermassen bedeutet die Präsenz des `SA_SIGINFO`-Flags, daß die Signale in `sa_mask` vor der Ausführung des Handlers der Signalmaske des Threads hinzugefügt wird. Das trifft nur zu, wenn `sa_handler` keine Behandlungsroutine enthält (also etwa nur die Makros `SIG_IGN` oder `SIG_DFL`). Ist das aber der Fall gelten die Flags in `sa_mask` für den Prozess und nicht für den Thread (Threads behandeln wir in Kapitel 11). Die ursprüngliche Maske wird bei Rückkehr des Signal Handlers wiederhergestellt. Dadurch sind wir in der Lage, bestimmte Signale zu blockieren, wenn ein Handler aufgerufen wird. Selbstverständlich ist das Signal, welches für den Prozess ausgeliefert wurde in der neuen Signalmaske enthalten. Das hat den Vorteil, daß immer wenn ein weiteres eintrifft und wir gerade ein Signal behandeln, dieses Signal solange geblockt wird, bis wir das erste Auftreten behandelt haben. Auch hier gilt die alte Regel: trifft das Signal während wir es blockieren mehrere male ein, so wird es nur einmal behandelt. Das kennen wir aus den vorangegangenen Besprechungen.

Wurde ein Signal Handler erst einmal mit seinen Attributen installiert, so bleibt er solange erhalten, bis wir die Konfiguration ausdrücklich ändern. Das ist eine der Anforderungen an POSIX-konforme Systeme und steht im Kontrast zu den unzuverlässigen Signalen betagter Unices.

Werfen wir einen Blick auf die Flags für das `sa_flags`-Feld der `sigaction`-Struktur:

SA_NOCLDSTOP

Wenn das Signal `SIGCHLD` entspricht, soll das Signal nicht generiert werden, wenn ein Kindsprozess durch Job Control gestoppt wird. Es wird aber natürlich generiert, wenn der Kindsprozess terminiert wird.

SA_ONSTACK

Wurde ein alternativer Signalstapel (*signal stack*) durch `signalstack` deklariert, wird das Signal dem Prozess auf diesem Stack zugestellt. Andernfalls wird das Signal dem aktuellen Stack zugestellt.

SA_RESETHAND

Ist das Flag gesetzt, wird die Disposition des Signals auf `SIG_DFL` zurückgesetzt und das `SA_SIGINFO`-Flag vor Eintritt in den Signal Handler entfernt.

SA_RESTART

Dieses Flag beeinflußt das Verhalten unterbrechbarer Funktionen (*interruptable functions*). Das bedeutet, daß solche als unterbrechbar gekennzeichnete Funktionen automatisch neugestartet werden. Dabei handelt es sich um eine ausgewählte Gruppe von Systemaufrufen.

SA_SIGINFO

Ist das Flag nicht gesetzt, so wird ganz normal in den Signal Handler eingetreten:

```
func(int signo);
```

mit `signo` als einziges Argument. In diesem Fall soll die Anwendung den in `sa_handler` spezifizierten Signal Handler benutzen, ihn aber nicht ändern. Ist das Flag gesetzt und das Signal abgefangen, wird der Handler mit folgender Synopsis aufgerufen:

```
void func(int signo, siginfo_t *info, void *context);
```

mit den beiden zusätzlichen Argumenten `info` und `context`. Das zweite Argument zeigt auf ein Objekt vom Typ `siginfo_t` und beinhaltet den Grund für die Generierung des Signals. Das dritte Argument kann in einen Zeiger auf ein Objekt vom Typ `ucontext_t` um auf den Kontext des empfangenden Prozesses, der durch die Signalzustellung unterbrochen wurde. In diesem Fall soll die Anwendung den Member `sa_sigaction` zur Beschreibung des Signal Handlers verwenden.

SA_NOCLDWAIT

Wenn dieser Schalter gesetzt und `sig` dem Signal `SIGCHLD` entspricht, werden Kindsprozesse eines Prozesses nicht in Zombies transformiert, wenn sie terminiert werden. Wartet der Prozess nun auf seine Kindsprozesse und weist selbst keine Zombies auf, soll er in den Zustand Blocking wechseln bis sich alle seine Kindsprozesse beenden. Die Funktionen `wait`, `waitid` und `waitpid(2)` liefern dann den Fehlercode `ECHILD`. Andernfalls werden diese Kindsprozesse in Zombies umgewandelt bis `SIGCHLD` auf `SIG_IGN` gesetzt wird.

SA_NODEFER

Wenn das Flag gesetzt und der Prozess `sig` behandelt, wird `sig` nicht zur Signalmaske des Prozesses hinzugefügt bevor in den Handler eingetreten wird. Das passiert nur, wenn `sig` in `sa_mask` eingetragen wurde. Andernfalls wird `sig` immer zur Signalmaske des Prozesses hinzugefügt.

Wenn ein Signal durch einen Handler, der durch `sigaction(2)` installiert wurde, abgefangen wird, kalkuliert das System eine neue Maske und installiert sie für den Zeitraum der Signalbehandlung (oder bis ein Aufruf von `sigprocmask(2)` oder `sigsuspend(2)` auftritt). Die Maske ist eine Kombination der Werte von `sa_mask` und der aktuellen Signalmaske. Das aktuell ausgelieferte ist nur enthalten, wenn nicht `SA_NODEFER` oder `SA_RESETHAND` gesetzt ist.

Problematisch ist ein kombinierter Aufruf von `signal(2)` und `sigaction(2)`. Wurde die vorangegangene Aktion für `sig` durch `signal(2)` konfiguriert, sind die Werte der `sigaction`-Member, die durch `oact` referenziert werden, undefiniert. Das soll heißen, daß insbesondere der Handler in `oact->sa_handler` nicht den gleichen Wert hat, der `signal(2)` übergeben wurde.

Listing: 8.12: Sichere Version des Beispiels 8.10 mit sigaction(2)

Greifen wir die Problematik aus Beispiel 8.10 auf und modifizieren den Code so, daß er nun sicher vor Unterbrechungen ist und die Nachteile der ersten Version ausräumt.

```

1 #include <sys/types.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <signal.h>
6
7 #define LIST_SIZE 10
8
9 struct list {
10     struct list *next;

```

```

11     int          value;
12 } list[LIST_SIZE];
13
14 struct list *head;
15
16 void signal_handler(int sig_num);
17 void print_list(int number);
18
19 int main(int argc, char **argv) {
20     int i, n;
21     struct sigaction sa;
22
23     if (argc == 2)
24         n = atoi(argv[1]);
25     else
26         n = 0;
27
28     /* install signal handler for SIGINT */
29     sa.sa_handler = signal_handler;
30     sigemptyset(&sa.sa_mask);
31     sa.sa_flags = 0;
32     sigaction(SIGINT, &sa, NULL);
33
34     /* create a linked list */
35     for (i = LIST_SIZE - 1; i >= 0; i--) {
36         list[i].value = i + 1;
37         list[i].next = head;
38         head = &list[i];
39     }
40
41     /* print the list, generating SIGINT when reaching element n */
42     print_list(n);
43
44     /* print list without generating signals */
45     print_list(0);
46
47     return (0);
48 }
49
50 void print_list(int number) {
51     struct list *plist;
52     sigset_t sigset, osigset;
53
54     /* clear signal set and add SIGINT, block it */
55     sigemptyset(&sigset);
56     sigaddset(&sigset, SIGINT);
57     sigprocmask(SIG_BLOCK, &sigset, &osigset);
58
59     /* print the value of each element in the list */
60     for (plist = head; plist != NULL; plist = plist->next) {
61         printf("%d ", plist->value);
62
63         /* if n-th element is reached generate signal */
64         if (plist->value == number)
65             kill(getpid(), SIGINT);
66     }
67     /* restore the original signal mask */
68     sigprocmask(SIG_SETMASK, &osigset, NULL);
69
70     printf("\n");
71 }
72

```

```

73 void signal_handler(int sig_num) {
74     struct list *plist, *olist;
75
76     /* search the list and remove element 5 */
77     olist = NULL;
78     for (plist = head; plist != NULL; plist = plist->next) {
79         if (plist->value == 5) {
80             if (olist != NULL)
81                 olist->next = plist->next;
82             else
83                 plist->next = NULL;
84
85             break;
86         }
87
88         olist = plist;
89     }
90 }
```

Listing 8.12: xcode/signalregion2.c - Sichere Version des Beispiels 8.10 mit sigaction(2).

□

8.7.2 Die Funktion sigsuspend

Im direkten Zusammenhang zu `sigaction(2)` steht die die Funktion `sigsuspend(2)`.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);

```

Rückgabewert: kehrt nie zurück, -1 bei Fehler.

Wir haben gesehen, wie die Signalmaske eines Prozesses so geändert werden kann, daß ausgewählte Signale blockiert und wieder durchgelassen werden können. Dadurch können wir kritische Regionen unseres Codes vor Unterbrechungen durch Signale schützen. Die Frage ist nun aber, was passiert, wenn wir ein Signal nicht mehr blockieren und dann gleich `pause(3)` aufrufen, um auf das zuvor blockierte Signal zu warten? Wir würden glauben, der folgende Code löse das Problem (hier: SIGINT):

```

sigset_t current_mask, backup_mask;

sigemptyset(&current_mask);
sigaddset(&current_mask, SIGINT);

if (sigprocmask(SIG_BLOCK, &current_mask, &backup_mask) < 0)
    err_fatal("sigprocmask failed for SIG_BLOCK");

/* enter critical section */
...
do_some_critical_stuff_here();
...
/* leave critical region */
if (sigprocmask(SIG_SETMASK, &backup_mask, (sigset_t *)NULL) < 0)
    err_fatal("sigprocmask failed for SIG_SETMASK");

pause();
...
do_other_stuff_here();
...
```

Listing 8.13: Beispiel für ein unerwünschtes Zeitfenster.

Es ist nicht ganz offensichtlich, aber zwischen dem zweiten Aufruf von `sigprocmask(2)` und `pause(3)` liegt ein Zeitfenster, in dem das Auftreten eines Signals (dummerweise genau das, welches wir gerade nicht mehr blockieren wollten) nicht bemerkt werden kann. Das Signal ist verloren. Um das zu vermeiden, müssten wir das Zurücksetzen der Signalmaske und den `pause(3)`-Aufruf in einer atomaren Operation durchführen. Genau zu diesem Zweck wurde `sigsuspend(2)` eingeführt.

```

sigset_t current_mask, backup_mask, allowall_mask;

sigemptyset(&allowall_mask); /* this one's used by sigsuspend() */
sigemptyset(&current_mask);
sigaddset(&current_mask, SIGINT);

if (sigprocmask(SIG_BLOCK, &current_mask, &backup_mask) < 0)
    err_fatal("sigprocmask failed for SIG_BLOCK");

/* enter critical section */
...
do_some_critical_stuff_here();
...
/* leave critical region and suspend execution until signal arrives */
if (sigsuspend(&allowall_mask) != -1) /* all signals are allowed here */
    err_fatal("sigsuspend failed");

if (sigprocmask(SIG_SETMASK, &backup_mask, (sigset_t *)NULL) < 0)
    err_fatal("sigprocmask failed for SIG_SETMASK");

pause();
...
do_other_stuff_here();
...

```

Listing 8.14: Bessere Variante: Auslösung des Zeitfensters von Listing 8.13.

Die Funktion wartet auf das Eintreffen eines Signals. Dazu ersetzt sie die aktuelle Signalmaske des aufrufenden Threads mit dem Signalsatz, auf den `sigmask` zeigt. Anschließend setzt sie den Thread aus, bis ein bestimmtes Signal ausgeliefert wird, dessen assoziierte Aktion entweder der Aufruf eines Signal Handlers oder die Terminierung des Threads ist. Ist letzteres der Fall, kehrt `sigsuspend(2)` nicht zurück. Andernfalls kehrt die Funktion zurück, sobald der Signal Handler zurückgekehrt ist. Dabei wird die Signalmaske wieder hergestellt, wie sie vor dem Aufruf von `sigsuspend(2)` vorhanden war. Das Verhalten wird etwas deutlicher, wenn wir eine Beispielimplementierung betrachten.

```

1 #include <errno.h>
2 #include <signal.h>
3 #include <unistd.h>
4
5 int sigsuspend(const sigset_t *set) {
6     sigset_t oset;
7     int save;
8
9     if (set == NULL) {
10         errno = EINVAL;
11         return -1;
12     }
13
14     if (sigprocmask(SIG_SETMASK, set, &oset) < 0)
15         return -1;
16
17     pause();
18     save = errno;
19
20     if (sigprocmask(SIG_SETMASK, &oset, (sigset_t *)NULL) < 0)

```

```

21         return -1;
22
23     errno = save;
24     return -1;
25 }
```

Listing 8.15: xcode/sigsuspendimpl.c - Beispielimplementierung für eine POSIX-konforme Variante von `sigsuspend`.

□

Im Beispiel 7.8 sahen wir, wie eine ganz einfache Shell programmiert werden kann. Doch leider begegnen wir dabei Problemen, die wir, weil das Konzept der Signale erst im Anschluß besprochen wurde, nicht erkannt haben.

Der Benutzer hat die Möglichkeit, das Programm mit Hilfe von STRG-C oder STRG-\ die Ausführung zu beenden. Egal welche Taste Sie drücken, das Signal wird an die Prozessgruppe im Vordergrund gesendet, die das Terminal verwendet. Daher würde das Signal nicht nur den Elternprozess, sondern auch den Kindsprozess beenden, sofern das Programm nicht im Hintergrund ausgeführt wird.

Das Problem könnte gelöst werden, wenn wir im Elternprozess `SIGINT` und `SIGQUIT` ignorieren, während die Signale für den Kindsprozess die Standardaktionen ausführen. Ein solchen Arrangement ist aber noch nicht besonders zufriedenstellend, denn das hätte zur Folge, daß die eingegebenen Daten, verloren wären, wenn wir noch keine neue Zeile am Ende eingegeben hätten (erinnern Sie sich, daß der Aufruf von `fgets(3)` in Beispiel 7.8 erst zurückkehrt, wenn wir entweder die Eingabetaste drücken oder `fgets(3)` ein EOF-Zeichen erhält.) Benutzerfreundlicher wäre es, wenn der Elternprozess den Prompt erneut anzeigen würde. Dazu müßten wir aber `SIGINT` und `SIGQUIT` behandeln. Das wäre außerdem auch recht bequem, denn die Disposition der Signale, die behandelt werden, wird während des Aufrufs von `exec(3)` auf die Standardeinstellung zurückgesetzt. Dadurch müssen wir im Child-Code keine speziellen Anpassungen vornehmen.

Und noch immer ist es nicht genug, einfach nur die Signale zu behandeln. Wenn nun ausgerechnet `fgets(3)` durch ein Signal unterbrochen wird, kehrt die Funktion zurück, ohne Daten gelesen zu haben und der Elternprozess wird beendet. Um das zu vermeiden, können wir einen entfernten Sprung (*non-local jump*) an genau die Stelle durchführen, bevor in die `while`-Schleife eingetreten wird. Die beiden Funktionen `sigsetjmp(3)` und `siglongjmp(3)` stellen quasi eine überregionale `goto`-Variante dar, denn wir dürfen auch außerhalb einer Funktion Sprünge durchführen.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

Rückgabewert: abhängig von der Aufrufart von `sigsetjmp(3)`, `siglongjmp` kehrt nach `env` zurück.

env

ist eine Struktur, die Zustandsinformationen der Ausführungsumgebung für den Sprung speichert.

savemask

Zeigt an, ob die Signalmaske des Prozesses und der Zustand des Schedulers auch gespeichert werden sollen. Ist `savemask` nicht 0 werden sie gespeichert, andernfalls nicht.

val

Die Funktion `siglongjmp` springt zu dem Punkt, die `sigsetjmp(3)` folgt, wie in `env` gespeichert. Für die Applikation sieht es so aus, als würde der durch `value` repräsentierte Wert zurückgegeben.

Gibt `sigsetjmp(3)` 0 zurück, wird davon ausgegangen, daß ein ganz normaler Rücksprung durchgeführt wurde. Ein anderer Wert ist nur durch einen Aufruf von `siglongjmp(3)` möglich. `sigsetjmp(3)` gibt immer 0 zurück, wenn `val` 1 ist.

Wie Sie aus den Ausführungen schon bemerkt haben, ist die Anwendung der beiden Funktionen nicht trivial. Kehrt `sigsetjmp(3)` mit einem Wert anders als 0 zurück, weisen alle Signal und statischen Variablen genau den Wert auf, den sie vor dem Aufruf von `siglongjmp(3)` hatten. Des Weiteren ist der Zustand der Variablen mit den Modifizierern `register` und `auto` (Standard) undefiniert. Sie sollten bei Bedarf als `volatile` deklariert werden. Aus diesen Gründen ist es ratsam, die beiden Funktionen nur selten einzusetzen, zumal im Allgemeinen von der Verwendung von `goto`-Statements in der strukturierten Programmierung abgeraten wird. Das trifft für nicht-lokale `goto`-Verzweigungen wohl noch viel mehr zu.

Listing 8.16: Erweitertes Beispiel 7.8 mit verbesserter Signalbehandlung

Zur Illustration der Funktionen `sigsetjmp(3)` und `siglongjmp(3)` ziehen wir einfach Beispiel 7.8 heran, daß die soeben erläuterten Nachteile aufweist.

```

1 #include <signal.h>
2 #include <setjmp.h>
3 #include "header.h"
4
5 #define INPUT_BUFSIZ 512
6
7 /* prototype */
8 void exec_cmd(char *cmd);
9 static void signal_handler(int signum);
10 static void dummy(int signum);
11
12 static sigjmp_buf env; /* structure to store execution environment */
13
14 int main(int argc, char **argv) {
15     char buffer[INPUT_BUFSIZ];
16     struct sigaction sa;
17
18     /* protect against siglongjmp() before sigsetjmp() is done */
19     if (sigsetjmp(env, 0))
20         err_fatal("sigsetjmp: interrupted\n");
21
22     /* the initial user prompt */
23     printf("> ");
24
25     sa.sa_handler = signal_handler;
26     sigemptyset(&sa.sa_mask);
27     sigaddset(&sa.sa_mask, SIGINT);
28     sigaddset(&sa.sa_mask, SIGQUIT);
29     sa.sa_flags = 0;
30
31     if (sigaction(SIGINT, &sa, NULL) < 0)
32         err_fatal("sigaction failed for SIGINT\n");
33
34     if (sigaction(SIGQUIT, &sa, NULL) < 0)
35         err_fatal("sigaction failed for SIGQUIT\n");
36
37     /*
38      * This is the real sigsetjmp(). If it returns nonzero value
39      * the signal handler has called siglongjmp(). Redisplay
40      * prompt in this case.
41     */
42     if (sigsetjmp(env, 1))
43         printf("\n> ");
44
45     /* poll user input and exec commands */
46     while (fgets(buffer, INPUT_BUFSIZ, stdin) != NULL) {
47         exec_cmd(buffer);
48         printf("> ");

```

```

49     }
50
51     return (0);
52 }
53
54 void exec_cmd(char *cmd) {
55     pid_t pid, wpid;
56     struct sigaction sig_int, osig_int;
57     struct sigaction sig_quit, osig_quit;
58
59     /* dummy */
60     sigemptyset(&sig_int.sa_mask);
61     sig_int.sa_handler = dummy;
62     sig_int.sa_flags = 0;
63
64     if (sigaction(SIGINT, &sig_int, &osig_int) < 0)
65         err_fatal("sigaction failed for SIGINT\n");
66
67     sigemptyset(&sig_quit.sa_mask);
68     sig_quit.sa_handler = dummy;
69     sig_quit.sa_flags = 0;
70
71     if (sigaction(SIGQUIT, &sig_quit, &osig_quit) < 0)
72         err_fatal("sigaction failed for SIGQUIT\n");
73
74     /* create child and exec cmd */
75     if ((pid = fork()) < 0) {
76         err_fatal("fork failed\n");
77
78         /* restore signal dispositions */
79         if (sigaction(SIGINT, &osig_int, NULL) < 0)
80             err_fatal("sigaction failed\n");
81
82         if (sigaction(SIGQUIT, &osig_quit, NULL) < 0)
83             err_fatal("sigaction failed\n");
84
85         return;
86     }
87
88     if (pid == 0) { /* child code */
89         execl("/sbin/sh", "sh", "-c", cmd, NULL);
90         err_fatal("execl failed\n");
91     }
92
93     /*
94      * Parent waits for its child to exit before
95      * restoring signal dispositions.
96      */
97
98     while ((wpid = waitpid(pid, NULL, 0)) != pid)
99         if ((wpid == -1) && (errno != EINTR))
100             err_fatal("waitpid failed\n");
101
102     if (sigaction(SIGINT, &osig_int, NULL) < 0)
103         err_fatal("sigaction failed\n");
104
105     if (sigaction(SIGQUIT, &osig_quit, NULL) < 0)
106         err_fatal("sigaction failed\n");
107 }
108
109 void signal_handler(int signum) {
110     /* flush the input and output buffers and jump back*/

```

```

111     fflush(stdout);
112     fflush(stdin);
113     siglongjmp(env, 1);
114 }
115
116 static void dummy(int signum) {}

```

Listing 8.16: xcode/simpleshell2.c - Erweitertes Beispiel 7.8 mit verbesserter Signalbehandlung.

Die Hauptfunktion (`main`) entspricht weitgehend der aus dem ursprünglichen Beispiel, mit der Ausnahme, daß wir einen Signal Handler für `SIGINT` und `SIGQUIT` installiert und zwei Aufrufe von `sigsetjmp(3)` hinzugefügt haben. Der erste Aufruf von `sigsetjmp(3)` schließt das Zeitfenster, das durch einen Aufruf von `siglongjmp(3)` mit einem uninitialisiertem Sprungpuffer auftaucht. Dieses Fenster würde zwischen dem Aufruf von `sigaction(2)` und dem Aufruf von `sigsetjmp(3)` mit `savemask = 1` entstehen. Der Fehler wird durch das Eintreffen des Signals in diesem Fenster verursacht. Es veranlaßt den Signal Handler, `siglongjmp(3)` aufzurufen, bevor der Sprungpuffer initialisiert werden konnte. Der zweite Aufruf von `sigsetjmp(3)` ist dafür verantwortlich, daß die BenutzerInnen einen neuen Prompt sehen, nachdem die Ausführung des Programms unterbrochen wurde. Nach dem Aufruf von `sigsetjmp(3)`, die 0 zurückgibt, treten wir in die `while`-Schleife ein. Wenn nun `SIGINT` oder `SIGQUIT` auftreten wird der Signal Handler (`signal_handler`) aufgerufen. Er lehrt die Buffer, damit sie sich in einem definierten Zustand befinden und ruft `siglongjmp(3)` auf, um in die Hauptfunktion zurückzuspringen. Da es diesmal so aussieht, als würde `sigsetjmp(3)` mit 1 zurückkehren, wird ein neuer Prompt ausgegeben. Während wir uns im Handler für ein Signal befinden, wird das andere blockiert.

Diese Version der einfachen Shell ist komplizierter als die erste, da wir die subtilen Interaktionen der Signale berücksichtigen müssen. Deshalb ist es auch notwendig, den Signal Handler `signal_handler` mit `dummy` auszutauschen. Wenn wir nämlich auf den Kindsprozess warten, möchten wir nicht, daß `siglongjmp(3)` in `main` zurückspringt, ohne den Terminierungsstatus des Kindsprozesses abzufragen, wenn er durch den Benutzer unterbrochen wurde. Wenn `signal_handler` hinterher noch installiert wäre, würde ein Zombie entstehen, der unser System solange verschmutzt, bis der Elternprozess terminiert wird. Wir müssen in einer Schleife auf den Kindsprozess warten, falls ein Signal `waitpid(2)` unterbricht. Gibt `waitpid(2)` -1 zurück und `errno` ist nicht auf `EINTR` gesetzt, wird eine Fehlernachricht ausgegeben und das Programm beendet. Natürlich stellen wir die ursprüngliche Disposition von `SIGINT` und `SIGQUIT` vor der Rückkehr des Elternprozesses aus `exec_cmd` wieder her. □

Kapitel 9

Das Teminal als Schnittstelle

All truths are easy to understand once they are discovered; the point is to discover them.

GALILEO GALILEI (1564 - 1642)

In diesem Kapitel beschäftigen wir uns mit dem UNIX-Terminal als allgemeine Schnittstelle. Der Begriff *Terminal* hat viele Bedeutungen. Zum einen bezeichnen wir damit Geräte, die über ein Netzwerk mit einem Rechner verbunden sind und selbst keine Rechenkapazitäten besitzen (sog. *dumb terminals*), zum anderen haben wir inzwischen gelernt, daß unser Kommandoprozessor, in den wir die Befehle eingeben, auch mit einem Terminal verbunden ist. Und das, obwohl wir vielleicht noch nicht einmal an ein Netzwerk angebunden sind.

Die Geschichte der Terminals ist wahrlich keine schöne. Die Gründe dafür sind sehr verschieden. Ganz sicher trug der Konkurrenzkampf unter den UNIX-Herstellern und unter den Herstellern von Hardware-Terminals dazu bei. Ein anderer Grund ist die Vielfalt der Aufgaben, die über ein Terminal erledigt werden können. Beispielsweise werden sie verwendet, um Netzwerkverbindungen mit anderen Rechnern herzustellen, Ausgaben auf einen Drucker um zuleiten oder Modems anzuschließen, die uns über Weitverkehrsnetzwerke mit anderen Systemen verbinden. Es handelt es sich bei den Terminals also um ein sehr vielseitiges Gebilde, daß sich da herauskristallisiert hat.

9.1 Charakteristika

Wenn wir eine Gerätedatei öffnen an das ein Terminal angeschlossen ist (beispielsweise `/dev/ttys0`), wartet der Thread solange, bis eine Verbindung hergestellt werden konnte. Normalerweise öffnen Anwendungsprogramme diese Dateien nicht eigenständig, sondern überlassen dies besonderen Programmen, die für die Anwendungen die Rolle der Standardeingaben, Standardausgaben oder Fehlerausgaben übernehmen.

Wie wir bereits von der Funktion `open(2)` wissen, wird ein Thread, der eine Terminaldatei öffnet in den Zustand *Blocking* versetzt, bis das Gerät bereit und in der Lage ist, Daten zu senden oder zu empfangen. Wird für das Modem das Flag `CLOCAL` nicht gesetzt, verbleibt der Prozess sogar so lange in diesem Zustand, bis eine Verbindung hergestellt wurde. Ist es aktiv oder `O_NONBLOCK` für `open(2)` angegeben, kehrt die Funktion sofort mit einem File Descriptor zurück, ohne auf das Modem, bzw. die Verbindung zu warten.

9.1.1 Eingabeverarbeitung

Ein Terminalgerät, das mit einer Gerätedatei (z.B. `/dev/ttys0`) verbunden ist, kann im Vollduplexmodus arbeiten, so daß Daten eintreffen können, während eine Ausgabe von Daten stattfindet. Jedem Gerät ist

eine *Queue* (Warteschlange) zugeordnet, in die eintreffende Daten geschrieben, bevor sie von dem Prozess gelesen werden. Die meisten Systeme geben ein Limit über die maximale Größe der Warteschlange vor, das über die `sysconf(3)`-Variable `MAX_INPUT` abgefragt werden kann. Für Eingabe und Ausgabe sind unabhängige Queues vorgesehen (Abbildung 9.1).

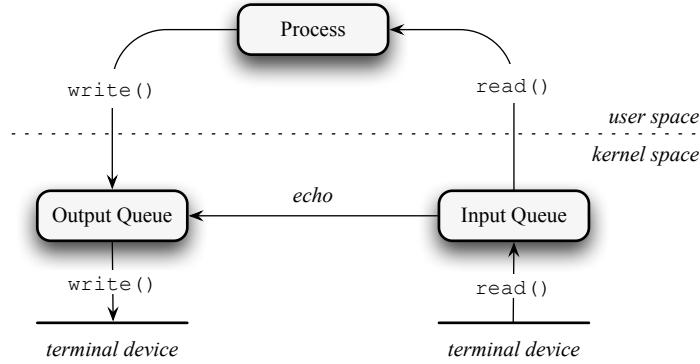


Abbildung 9.1: Warteschlangen für Terminal E/A

Uns stehen zwei unterschiedliche Eingabemodi zur Verfügung: der kanonische und nicht-kanonischen Modus (auf die beiden Modi kommen wir gleich zu sprechen). Des Weiteren werden Zeichen nicht einfach nur so eingelesen oder ausgegeben. Über Flags (die wir in `c_lflag` und `c_oflag` der `termios`-Struktur kodieren) legen wir fest, wie wir die Zeichen verarbeiten möchten. Beispiel für eine solche besondere Verarbeitung ist das Echoing, welches Vollduplexterminals ermöglicht eingelesene Zeichen auch gleich wieder in die Ausgabewarteschlange zu schreiben. Im Halbduplexbetrieb müssen die Terminals erst warten, bis eine bestimmte Anzahl von Zeichen vollständig eingelesen wurde und können sie dann wieder ausgeben. Die `termios`-Struktur besprechen wir in Abschnitt 9.2.

Die Art und Weise, wie die Daten von einem Terminal eingelesen werden hängt von den Optionen ab, die wir `open(2)` mit auf den Weg gegeben haben oder mit der `fcntl(2)`-Funktion gesetzt haben. Wurde das `O_NONBLOCK`-Flag bei `open(2)` gesetzt, wird die `read(2)`-Anfrage sofort bearbeitet, ohne daß der Aufrufer in den Zustand *Blocking* übergeht. Beim Lesen aus dem Terminal können drei unterschiedliche Szenarien auftreten:

1. Wenn genügend Daten zur Verfügung stehen, um die Anfrage vollständig zu erfüllen, kehrt `read(2)` erfolgreich zurück und zeigt die Anzahl der gelesenen Bytes an.
2. Wenn wiederum nicht genügend Daten zur Verfügung stehen, um die Anfrage vollständig zu erfüllen, kehrt `read(2)` zurück und zeigt die Anzahl der Bytes an, die maximal gelesen werden konnten (*short read*).
3. Stehen gar keine Daten zum Lesen zur Verfügung, kehrt `read(2)` mit -1 zurück und `errno` wird auf `EAGAIN` gesetzt.

Ob Daten zu einem bestimmten Zeitpunkt zur Verfügung stehen, hängt davon ab, ob im kanonischen oder nicht-kanonischen Format gearbeitet wird.

9.1.2 Der kanonische Eingabemodus

Im kanonischen Eingabemodus (auch *cooked mode* genannt) wird die Eingabe zeilenweise verarbeitet. Eine Zeile wird am Ende durch einen *Newline Character* (Wagenrücklauf), beispielsweise erzeugt durch das Drücken der Eingabetaste (`ENTER`), ein EOF-Zeichen (*end-of-file*) oder von einem EOL-Zeichen (*end-of-line*) abgegrenzt. Das bedeutet für die `read(2)`-Operation, daß sie nicht eher zurückkehren darf, bis eines der drei Zeichen angetroffen wurde. Es ist aber nicht zwingend notwendig, eine ganze Zeile zu lesen,

bis die Funktion zurückkehren darf, denn schließlich kann die Zeile länger sein, als die Größe des Buffers in den sie geschrieben werden soll. Daher ist es legitim, nur eine bestimmte Anzahl von Bytes, oder gar jedes Byte einzeln aus der Queue zu lesen. Die maximale Länge einer Zeile wird durch die Systemkonfiguration `MAX_CANON` beschrieben, die wir mittels `pathconf(3)` abfragen können. Ist `MAX_CANON` nicht definiert, existiert für dieses System kein Limit.

Interessant ist das Handling der Sonderzeichen wie beispielsweise `ERASE` und `KILL` (siehe Sonderzeichen weiter hinten im Text). Wird so ein Zeichen angetroffen, hat die Auswertung desselben nur Einfluß auf die Zeichen in der Eingabewarteschlange (*input queue*), die noch nicht durch ein NL-, EOF- oder EOL-Zeichen abgeschlossen wurde. Eine solche unabgegrenzte Zeichenkette stellt die aktuelle Zeile in der Warteschlange dar. Das `ERASE`-Zeichen löscht das letzte Zeichen in der Queue, während das `KILL`-Zeichen die gesamte aktuelle Zeile löscht. Die beiden Sonderzeichen werden aber selbst nicht in der Input Queue abgelegt.

9.1.3 Der nicht-kanonische Eingabemodus

Im nicht-kanonischen Eingabemodus (auch *raw mode* genannt) werden Bytes nicht zu einer Zeile zusammengefaßt und es wird auch keine Auswertung der `ERASE`- und `KILL`-Zeichen vorgenommen. Jedem Terminal, für das wir die Eingaben auswerten wollen, ist ein `c_cc`-Array zugeordnet, dessen Werte der beiden Member `MIN` und `TIME` darüber Aufschluß geben, wie die Bytes verarbeitet werden. Pikanterweise schreibt der POSIX-Standard nicht vor, ob die Option `O_NONBLOCK` von `open(2)` Vorrang vor `MIN` oder `TIME` hat. Es ist also möglich, daß, wenn `O_NONBLOCK` gesetzt ist, `read(2)` sofort zurückkehrt, unabhängig von den Werten für `MIN` oder `TIME`.

Listing 9.1: Umschalten in den nicht-kanonischen Modus

Das folgende Listing illustriert, wie der nicht-kanonische Modus arbeitet. Übersetzen Sie das Programm und geben Sie ein paar Zeichen ein. Erst wenn Sie ENTER drücken wird die Zeile ausgegeben. Die Bedeutung der Flags, der `tc`-Funktionen und andere Details besprechen wir im Laufe des Kapitels.

```

1 #include <termios.h>
2 #include "header.h"
3
4 /* Use this variable to remember original terminal attributes. */
5 struct termios saved_attributes;
6
7 void reset_input_mode(void) {
8     tcsetattr(STDIN_FILENO, TCSANOW, &saved_attributes);
9 }
10
11 void set_input_mode(void) {
12     struct termios tattr;
13     char *name;
14
15     /* Make sure stdin is a terminal. */
16     if (!isatty(STDIN_FILENO))
17         err_fatal("stdin is connected to a terminal.");
18
19     /* Save the terminal attributes so we can restore them later. */
20     tcgetattr(STDIN_FILENO, &saved_attributes);
21     atexit(reset_input_mode);
22
23     tcgetattr(STDIN_FILENO, &tattr);
24     tattr.c_lflag &= ~(ICANON | ECHO); /* Clear ICANON and ECHO. */
25     tattr.c_cc[VMIN] = 1;
26     tattr.c_cc[VTIME] = 0;
27     tattr.c_cc[VINTR] = 4;
28     tcsetattr (STDIN_FILENO, TCSAFLUSH, &tattr);
29 }
```

```

30
31 int main(int argc, char **argv) {
32     char c;
33
34     set_input_mode();
35     printf("Non-canonical mode set.\n");
36     printf("Enter characters and press ENTER (CTRL-D to quit):\n");
37
38     while (1) {
39         read(STDIN_FILENO, &c, 1);
40         putchar(c);
41     }
42
43     return (0);
44 }
```

Listing 9.1: xcode/noncanon.c - Umschalten in den nicht-kanonischen Modus.



Das Member MIN stellt die Mindestanzahl an Bytes dar, die empfangen werden sollten, wenn `read(2)` erfolgreich zurückkehrt. TIME ist ein Timer mit einer Auflösung von 1/10 Sekunde, der kurzlebige Datenübertragungen und Datenströme zeitlich begrenzt. Wenn MIN größer als MAX_INPUT ist, hängt es von dem System an, wie mit `read(2)`-Anfragen umgegangen wird. Es ist entscheidend in welcher Konstellation beide Werte auftreten. Danach richtet sich nämlich die Interaktion von Schreib- und Leseanfragen mit den zu erwartenden Daten.

Es gibt vier verschiedene Möglichkeiten, wie MIN und TIME auftreten können:

MIN > 0, TIME > 0

Hier fungiert TIMER als ein *Inter-Byte Timer*, der nach dem Empfang des ersten Bytes aktiviert wird. Die Eigenschaft eines Inter-Byte Timers ist, daß er nach jedem eingelesenen Byte wieder zurückgesetzt wird. Wenn nun MIN Bytes empfangen wurden, bevor der Timer abgelaufen ist, wird die `read(2)`-Anfrage erfolgreich abgeschlossen. Läuft er hingegen ab, bevor MIN Bytes empfangen wurden, werden die bis dahin eingelesenen Bytes an den Prozess übergeben (das entspricht weitestgehend dem Verhalten, wie wir es von `read(2)` kennen). Wenn der Timer abläuft, wird immer mindestens ein Byte zurückgeliefert, denn wenn nicht mindestens ein Byte empfangen wurde, wäre der Timer erst gar nicht gestartet worden. Das bedeutet allerdings auch, daß die `read(2)`-Anfrage solange blockiert (also im Zustand *Blocking* verweilt) bis beide Mechanismen, TIME und MIN, durch den Empfang eines Bytes oder eines Signals aktiviert werden.

TIME > 0, TIME = 0

Wird TIMER auf 0 gesetzt, ist er deaktiviert und nur MIN ist von Bedeutung. Eine ausstehende `read(2)`-Anfrage ist nur erfolgreich (bleibt im Zustand *Blocking*), wenn mindestens MIN Bytes empfangen wurden. Programme, die von dieser Konstellation ausgehen um Record-basiertes E/A durchzuführen, laufen Gefahr, unendlich lange in der `read(2)`-Operation zu verweilen.

MIN = 0, TIME > 0

Dieser Fall ist das genaue Gegenteil des vorangegangenen. Nun ist nur TIME signifikant. Der Wert fungiert als `read(2)`-Timer, der gestartet wird, sobald `read(2)` vom System verarbeitet wird. Die Anfrage liefert ein Ergebnis zurück, wenn das erste Byte gelesen wurde, oder der Timer abläuft. Wenn kein Byte nach Ablauf von TIME * 0,1 Sekunden nach Bearbeitung der `read(2)`-Anfrage empfangen wurde, gibt `read(2)` 0 zurück und zeigt damit an, daß die Anfrage fehl schlug.

MIN = 0, TIME = 0

In dieser Konstellation kehrt `read(2)` sofort zurück und liefert die Anzahl der zur Zeit verfügbaren Bytes zurück, ohne auf weitere Bytes zu warten. Wurde kein Zeichen gelesen, gibt `read(2)` 0 zurück.

All die beschriebenen Verfahren zur Verarbeitung der Ein- und Ausgaben werden über ein Modul geregelt, daß als *Line Discipline* bezeichnet wird. Es arbeitet auf Kernel-Ebene und sitzt zwischen den `read(2)/write(2)`-Systemaufrufen und den eigentlich Gerätetreibern (Abbildung 9.2).

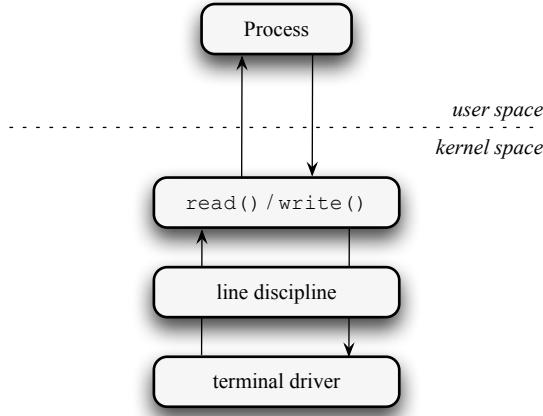


Abbildung 9.2: Rolle der Line Discipline

Schreibt ein Prozess ein oder mehr Bytes in ein Terminalgerät, wird die Ausgabe nach den Einstellungen des `c_oflags` der `termios`-Struktur vorgenommen (diese Struktur und seine Member wird weiter hinten besprochen). Das System verwendet dazu unter Umständen einen Puffer, so daß die `write(2)`-Anfrage erfolgreich abgeschlossen wurde, die Daten aber noch nicht wirklich an das Gerät übermittelt wurden.

9.1.4 Sonderzeichen

Einige Zeichen sind spezielle Funktionen zugeordnet, die sowohl Ausgabe als auch die Eingabe beeinflussen. Zwei haben wir bereits kennengelernt: `ERASE` und `KILL`. Die folgende Auflistung beschreibt die Besonderheiten dieser Zeichen.

INTR - interrupt

Das `INTR`-Zeichen wird bei der Eingabe ausgewertet und erkannt, wenn das `ISIG`-Flag gesetzt ist. Es erzeugt ein `SIGINT`-Signal, das der Prozessgruppe im Vordergrund gesendet wird, die das aktuelle Terminal kontrolliert. Bei der Verarbeitung wird das Zeichen verworfen.

QUIT

Das `QUIT`-Zeichen wird bei der Eingabe ausgewertet und erkannt, wenn das `ISIG`-Flag gesetzt ist. Es erzeugt ein `SIGQUIT`-Signal, das der Prozessgruppe im Vordergrund gesendet wird, die das aktuelle Terminal kontrolliert. Bei der Verarbeitung wird das Zeichen verworfen.

ERASE

Das `ERASE`-Zeichen wird bei der Eingabe ausgewertet und erkannt, wenn der kanonische Modus aktiviert ist. Es löscht das zuletzt eingelesene Zeichen, aber nicht über den Zeilenanfang hinaus. Wenn `ICANON` gesetzt ist, wird es bei der Verarbeitung verworfen.

KILL

Das `KILL`-Zeichen wird bei der Eingabe ausgewertet und erkannt, wenn der kanonische Modus aktiviert ist. Es löscht im Gegensatz zu `ERASE` die gesamte Zeile. Es wird der Verarbeitung verworfen.

EOF - end of file

Sonderzeichen bei der Eingabe, das erkannt wird, wenn der kanonische Modus aktiviert ist. Beim Empfang dieses Zeichens, werden alle Bytes die eingelesen werden sollen, dem Prozess übergeben ohne auf eine neue Zeile zu warten. Das Zeichen selbst wird bei der Verarbeitung verworfen. Gibt es keine eingelesenen Bytes (beispielsweise wenn `EOF` das erste Zeichen der Zeile ist) wird 0 zurückgeliefert.

NL - new line

Sonderzeichen für die Eingabe, das erkannt wird, wenn der kanonische Modus aktiviert ist. Es leitet eine neue Zeile ein und kann nicht geändert werden (wird also nicht bei der Verarbeitung verworfen).

EOL - end of line

Sonderzeichen für die Eingabe, das erkannt wird, wenn der kanonische Modus aktiviert ist. Es ist ein zusätzliches Trennzeichen für neue Zeilen, ähnlich NL.

SUSP - suspend

Wenn das ISIG-Flag gesetzt ist, wird ein SIGTSTP-Signal an alle Prozesse in der Prozessgruppe im Vordergrund gesendet. Das Zeichen wird bei der Verarbeitung verworfen.

STOP

Sonderzeichen, daß bei der Ein- und Ausgabe ausgewertet und erkannt wird, wenn das IXON- (*output control*) oder IXOFF- (*input control*) Flag gesetzt ist. Es kann dazu verwendet werden, die Ausgabe vorrübergehend auszusetzen. Es ist im Zusammenhang mit CRT-Terminals sinnvoll, um zu vermeiden, daß Ausgaben verloren gehen, bevor sie gelesen werden konnten. Das Zeichen wird bei der Verarbeitung verworfen.

START

Das START-Zeichen wird bei der Ein- und Ausgabe ausgewertet und erkannt, wenn das IXON- (*output control*) oder IXOFF- (*input control*) Flag gesetzt ist. Es kann dazu verwendet werden, die Ausgabe fortzusetzen, welche zuvor durch STOP angehalten wurde. Ist IXON gesetzt, wird das Zeichen bei der Auswertung verworfen.

CR - carriage return

Das CR-Zeichen wird nur bei der Eingabe erkannt, wenn der kanonische Modus aktiviert ist. Es repräsentiert den Wagenrücklauf. Wenn der kanonische Modus aktiviert und ICRNL, aber nicht IGNCR gesetzt ist, wird das Zeichen in NL übersetzt und hat dann den gleichen Effekt wie das NL-Zeichen.

Das NL- und CR-Zeichen kann nicht verändert werden. Einige Systeme erlauben das Ändern der START- und STOP-Zeichen. Im Gegensatz dazu ist zum Teil sogar erwünscht, die Zeichen INTR, QUIT, ERASE, KILL, EOF, EOL und SUSP zu ändern, wie wir in einem der Beispiele weiter hinten sehen werden. So kann das Verhalten an die eigenen Bedürfnisse angepaßt werden.

Beachten Sie, daß ein Sonderzeichen nicht nur durch seinen assoziierten Wert, sondern auch durch den Kontext in dem er auftritt erkannt wird. Beispielsweise könnte ein System Multibyte-Sequenzen eine andere Bedeutung zugeordnet haben, als die Bytes einzeln aufweisen. Sie werden nur erkannt, wenn das IEXTEN-Flag gesetzt wurde, andernfalls werden die Bytes nicht interpretiert.

9.2 Die Struktur `termios`

In den vorangegangenen Ausführungen haben wir immer wieder auf Flags verwiesen, die bestimmte Aufgaben erfüllen. Einige geben den Operationsmodus (kanonisch oder nicht-kanonisch) an, andere legen fest, ob und wie einzelne Sonderzeichen behandelt werden müssen. Diese Flags werden in der `termios`-Struktur gesetzt, die im Header `<termios.h>` definiert ist:

```
struct termios {
    tcflag_t c_iflag;      /* input modes */
    tcflag_t c_oflag;      /* output modes */
    tcflag_t c_cflag;      /* control modes */
    tcflag_t c_lflag;      /* local modes */
    cc_t     c_cc[NCCS] /* control characters */
}
```

Die beiden Datentypen `tcflag_t` und `cc_t` sind, zumindest was den POSIX-Standard angeht, immer `unsigned int`. Die Member und deren Werte sind in Tabelle 9.1 aufgeführt.

Member	Flag	Beschreibung	POSIX
c_iflag	BKREINT ICRNL IGNBRK IGNCR IGNPAR IMAXBEL INLCR INPCK ISTRIP IUCLC IXANY IXOFF IXON PARMRK	generiere ein SIGINT-Signal bei BREAK wandle CR in NL um ignoriere BREAK ignoriere CR ignoriere Zeichen mit Paritätsfehlern erzeuge Piepton, wenn Eingabe-Queue voll wandle NL in CR um schalte Paritätsprüfung ein streiche 8. Bit der Eingabezeichen weg wandle von Groß- in Kleinbuchstaben um jedes Zeichen startet Ausgabe neu schalte Flußkontrolle für Eingabe ein/aus schalte Flußkontrolle für Ausgabe ein/aus kennzeichne Paritätsfehler	• • • • • • • • • • • • • • •
c_oflag	BSDLY CRDLY FFDLY NLDLY OCRNL OFDEL OFILL OLCUC ONLCR ONLRET ONOCR ONOEOOT OPOST OXTABS TABDLY VTDLY	Verzögerung bei Backspace an/aus Verzögerung bei CR an/aus Verzögerung bei Seitenvorschub (<i>form feed</i>) Verzögerung bei NL wandle CR in NL um auffüllen ist DEL, andernfalls NUL benutze Füllzeichen für Verzögerung wandle von Klein- in Großbuchstaben um wandle NL in CR NL fungtiert wie CR bei CR keine Ausgabe in Spalte 0 verwerfe EOT (Ê) bei Ausgabe führe Ausgabeverarbeitung durch wandle TABs in Leerzeichen um Verzögerung für horizontale TABs Verzögerung für vertikale TABs	•
c_cflag	CCTS_OFLOW CIGNORE CLOCAL CREAD CTRS_IFLOW CSIZE CSTOPB HUPCL MDMBUF PARENB PARODD	CTS-Flußkontrolle bei Ausgaben ignoriere Kontrollflags ignoriere Statusleitungen des Modems schalte Empfänger ein RTS-Flußkontrolle bei Eingaben Anzahl der Bits pro Byte sende zwei Stopbits, andernfalls eines lege auf bei last close Flußkontrolle der Ausgabe durch den Träger schalte Paritätsprüfung ein ungerade Parität, andernfalls gerade	• • • • • • • • • •
c_lflag	ALTWERASE ECHO ECHOCTL ECHOE ECHOK ECHOKE ECHONL ECHOPRT FLUSHNO ICANON IEXTEN ISIG NOFLUSH NOKERNINFO PENDIN TOSTOP XCASE	verwende alternativen WERASE-Algorithmus schalte Echo ein gebe Kontrollzeichen als Zeichen aus lösche Zeichen visuell gebe das KILL-Zeichen aus lösche das KILL-Zeichen visuell gebe NL aus visuell Löschen im Hardcopy-Modus Output leeren (flush) kanonischer Eingabemodus erweiterte Eingabeverarbeitung schalte Signale ein, die vom Terminal generiert werden schalte Flushing nach Interrupt oder QUIT aus keine Kernelausgaben von STATUS gebe ausstehende Eingaben erneut ein sende SIGTTOU für Hintergrundausgaben kanonische Präsentation von Gross-/Kleinbuchstaben	• • • • • • • • • • • • • •

Tabelle 9.1: Flags der termios-Struktur

9.2.1 Eingabemodi

Die Werte des `c_iflags` (*input flags*) beschreiben die grundlegenden Einstellungen des Terminals für die Eingabe. Sie sind durch logische ODER-Operationen miteinander verknüpft. Die symbolischen Namen für alle Flags der `termios`-Struktur sind in Tabelle 9.1 zusammengefaßt und im Header `<termios.h>` definiert.

In der asynchronen, seriellen Datenübertragung tritt eine Abbruchbedingung (BREAK) ein, wenn eine Sequenz von 0-Bits angetroffen wird. Wenn `IGNBRK` gesetzt ist, wird `BREAK` bei der Eingabe ignoriert. Das Zeichen wird erst gar nicht in die Eingabewarteschlange geschrieben und daher nicht von Prozessen gelesen. Ist stattdessen `BRKINT` gesetzt (`IGNBRK` darf nicht gesetzt sein), leert die Abbruchbedingung die Ein- und Ausgabewarteschlangen und generiert gleichzeitig ein `SIGINT`-Signal, wenn das Terminal das Kontrollterminal der Prozessgruppe im Vordergrund ist. Wurde keine der beiden Flags gesetzt, wird die Abbruchbedingung als `0x00` eingelesen oder als `0xff 0x00 0x00`, wenn Paritätsfehler gekennzeichnet werden (das erledigt `PARMRK`). Solche fehlerhaften Bytes werden bei der Eingabe ignoriert. Normalerweise erfährt die Anwendung von dem Paritätsfehler, weil eine 3-Bytessequenz gesendet wird, die Aufschluß über den Fehler gibt. Das Format ist einfach: `0xff 0x00 [..]`, wobei `0xff 0x00` das Präfix darstellt und `[..]` das fehlerhafte Byte enthält. Das könnte zu Problemen führen, wenn `ISTRIP` (zum Streichen des achten Bits) gesetzt ist, denn `0xff` wäre zweideutig. Daher wird es als `0xff 0xff` der Applikation zugeführt. Anwendungen ignorieren die 3-Bytessequenz, wenn `IGNPAR` gesetzt ist.

Die Paritätsprüfung wird durch das Flag `INPCK` aktiviert. Die Paritätserzeugung und -erkennung und die Paritätsprüfung sind zwei unterschiedliche Dinge. Die Paritätserzeugung und -erkennung (*control mode*, Abschnitt 9.2.3) wird durch das `PARENB`-Flag (*parity enable*) und die Paritätsprüfung durch das Zusammenspiel der `INPCK`-, `IGNPAR` und `PARMRK`-Flags (*input mode*) kontrolliert. Ist `PARENB` gesetzt, wird der Gerätetreiber, der seriellen Schnittstelle veranlaßt, eine Parität für ausgehende Zeichen zu erzeugen und für eingehende Zeichen zu prüfen. Je nachdem ob `PARODD` aktiv ist oder nicht, wird eine ungerade (*odd parity*) oder eine gerade Parität (*even parity*) erzeugt. Trifft nun ein Byte mit einer falschen Parität ein, wird zuerst das `INPCK`-Flag geprüft und wenn es gesetzt ist, wird zusätzlich das `IGNPAR` geprüft, welches die Ignorierung des Paritätsfehlers veranlaßt. Werden die Bytes nicht ignoriert, muß `PARMRK` konsultiert werden, um herauszufinden, wie mit den fehlerhaften Bytes verfahren werden soll. Ob die Paritätsprüfung der Eingabe aktiviert ist oder nicht, hat nichts damit zu tun, ob die Paritätserkennung aktiviert ist oder nicht.

Die Bedeutung der anderen Flags wird weiter hinten in der Übersicht besprochen.

9.2.2 Ausgabemodi

Das `c_oflag` bestimmt, wie die Ausgabe durch die Schnittstelle des Terminals geregelt wird.

Das `OPOST`-Flag spezifiziert, ob ausgehende Daten nachbearbeitet werden sollen, damit beispielsweise Zeilen an das Terminal angepaßt werden, so daß die Ausgaben besser aussehen. Ist das Flag deaktiviert, wird die Ausgabe ohne Veränderungen übermittelt.

Das Zusammenspiel von `NL` und `CR` gibt immer wieder Rätsel auf. Wird das Flag `ONLCR` gesetzt, wird ein `NL`-Zeichen als ein `CR-NL`-Paar (in dieser Reihenfolge) übermittelt. Ist hingegen `OCRNL` gesetzt, wird ein `CR`-Zeichen auch nur als ein `NL`-Zeichen übermittelt. Anders funktioniert das Flag `ONLRET`. Wenn es aktiviert ist, übernimmt das Zeichen die Aufgabe eines Wagenrücklaufs (*carriage return*). Dadurch wird der Spaltenzeiger auf 0 gesetzt. Andernfalls übernimmt `NL` nur die Funktion eines Zeilenvorschubs (*line feed*) und der Spaltenzeiger bleibt unverändert.

Die Verzögerungsbits (z.B. `CRDLY` oder `FFDLY`) legen fest, wie lange die Transmission ausgesetzt werden soll, um langwierige Vorgänge des Terminals, wie etwa mechanische Justierungen, zu erlauben. Denken Sie daran, daß Terminals vielfältige Aufgaben übernehmen können, wie beispielsweise die Druckeranbindung. Naturgemäß sind diese Geräte nicht die schnellsten, gerade weil manchmal mechanische Einstellungen vorgenommen werden müssen. Zurück zu den Verzögerungen. Die Verzögerung kann auf zwei unterschiedliche Arten bewerkstelligt werden. Eine Möglichkeit besteht in der Verwendung von Füllzeichen, was für Geräte mit langsamen Baudaten durchaus praktikabel ist. Die andere Möglichkeit ist eine zeitliche Verzögerung, die besonders für schnelle Geräte mit hohen Baudaten geeignet ist. Ist `OFDEL` gesetzt, wird

DEL als Füllzeichen benutzt, andernfalls NUL. Ein VTDLY (*vertical TAB delay*) oder ein FFDLY (*form feed delay*) soll mindestens zwei Sekunden dauern und wird von den meisten Implementierungen unterstützt. NLDLY (*newline delay*) dauert etwa 0,1 Sekunden und wenn **ONLRET** gesetzt ist dauert die Verzögerung genau so lange wie ein Wagenrücklauf, der in drei Kategorien eingeteilt ist. Typ 1 ist abhängig von der aktuellen Spalte, Typ 2 dauert etwa 0,1 Sekunden und Typ 3 um die 0,15 Sekunden. Wenn **OFILL** gesetzt ist, übermittelt Typ 1 zwei und Typ 2 vier Füllzeichen. Ganz ähnlich funktionieren auch die anderen Verzögerungen auf die ich in der Übersicht weiter hinten eingehe.

9.2.3 Kontrollmodi

Mit dem **c_flag** nehmen wir Einfluß auf die Hardwareeinstellungen des Terminals. Nicht alle Einstellungen stehen auf allen Geräten zur Verfügung. Welche Möglichkeiten wir haben und wie sie sich auswirken, entnehmen wir der Auflistung und der Erklärung der Optionen weiter hinten. An dieser Stelle möchte ich einige etwas weniger offensichtliche Zusammenhänge erläutern.

Laut Tabelle 9.1 wird mit dem **CSIZE**-Flag die Anzahl der zu sendenden und empfangenden Bits pro Byte festgelegt. Folgende Werte sind möglich: **CS5** (5 Bits), **CS6** (6 Bits), **CS7** (7 Bits) und **CS8** (8 Bits). Die Paritätsbits sind darin nicht enthalten.

Neben den verschiedenen Flags für Ein- und Ausgaben werden auch die Baudraten in der **termios**-Struktur gespeichert. Allerdings greifen wir auf diese Informationen nicht direkt zu, sondern über die Funktionen **cfgetspeed(3)**, **cfgtospeed(3)**, **cfsetspeed(3)** und **cfsetospeed(3)**.

Die Baudraten und ihre Symbole sind in Tabelle 9.2 zusammengefaßt.

Symbol	Beschreibung	Symbol	Beschreibung
B0	Auflegen (<i>hang up</i>)	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134.5 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

Tabelle 9.2: Baudraten und ihre Symbolischen Konstanten

Wenn das Objekt für das die Kontrollmodi gesetzt werden, keine asynchrone serielle Verbindung ist, werden einige Einstellungen ignoriert. Beispielsweise würde der Versuch, die Baudrate einer Netzwerkverbindung zu einem Terminal eines anderer Hosts zu setzen fehlschlagen, weil sie nur zwischen direkt verbundenen Terminals gesetzt werden muß.

9.2.4 Lokale Modi

Das **c_lflag** muß für unterschiedliche Einstellungen herhalten. Beispielsweise, wie das *Echoing* arbeiten soll und ob kanonischer oder nicht-kanonischer Modus verwendet wird.

Ist das **ECHO**-Flag gesetzt werden alle Eingaben wieder direkt an das Terminal zurückgegeben. So können wir also sehen, was wir in das Terminal eingegeben haben oder dem Terminal zugeführt wurde. Wenn **ICANON** und **ECHOE** gesetzt sind, wird beim Antreffen von **ERASE** versucht, das letzte Zeichen im Terminal zu löschen. Ebenso sorgt **ECHOK** zusammen mit **ICANON** für das Löschen der zuletzt eingegebenen Zeile.

9.2.5 Beschreibung der Flags der **termios**-Struktur

Folgende Liste zeigt die möglichen Werte für die jeweiligen Member der **termios**-Struktur. Das Format lautet: **Wert/Konstante (Struktur-member)**.

ALTWERASE (c_lflag)

Mit diesem Flag wird für WERASE der alternative Algorithmus für das Löschen des letzten Zeichens verwendet. Dabei wird nicht bis zum ersten Leerzeichen gelöscht, sondern bis zum Antreffen des nächsten nicht alphanumerischen Zeichens.

BRKINT (c_iflag)

Wird ein BREAK empfangen und ist BRKINT, aber nicht IGNBRK gesetzt, so werden die Queues der Ein- und Ausgabe gelöscht und das SIGINT-Signal abgesetzt. Es ist an die Prozessgruppe im Vordergrund des Kontrollterminals gerichtet. Sind weder IGNBRK noch BRKINT gesetzt, wird BREAK als gewöhnliche 1-Bytessequenz oder mit aktivierter Paritätserkennung als 3-Bytessequenz.

BSDLY (c_oflag)

Bestimmt, ob die Verzögerung für die Löschentaste aktiviert werden soll. Die gültigen Werte sind entweder BS0 oder BS1.

CCTS_OFLOW (c_cflag)

Flußkontrolle über CTS (*clear to send*). Wird in einigen BSD-Derivaten nicht verwendet.

IGNORE (c_cflag)

Sorgt dafür, daß die Flags des Kontrollmodus ignoriert werden.

CLOCAL (c_cflag)

Zeigt an, daß die Statusleitungen ignoriert werden, was meistens darauf hin deutet, daß kein Modem im Spiel ist, daß Gerät also lokal angebunden ist. Ist das Flag nicht gesetzt, blockiert ein Aufruf von `open(2)` für ein Terminal solange, bis das Modem antwortet.

CRDLY (c_oflag)

Bestimmt die Maske für den Wagenrücklauf (*carriage return*). Gültige Werte sind CR0 (kein Verzögerung) oder abhängig von der Spaltenposition, CR1 Verzögerung für Terminal 300, CR2 Verzögerung für Terminal 37 oder CR3 Verzögerung für Concept 100.

CREAD (c_oflag)

Wenn das Flag aktiviert ist, zeigt es an, daß der Empfänger bereit ist und Zeichen gesendet werden können.

CRTS_IFLOW (c_cflag)

Flußkontrolle über RTS (*ready to send*).

CSIZE (c_cflag)

Zeigt an, wie viele Bits eines Bytes für das Senden und den Empfang verwendet werden. Gültige Werte sind: CS5 (5 Bits), CS6 (6 Bits), CS7 (7 Bits) oder CS8 (8 Bits). Die Werte berücksichtigen keine zusätzlichen Paritätsbits, wenn vorhanden.

CSTOPB (c_cflag)

Zeigt an, wie viele Stopbits gesendet werden sollen. Wenn es gesetzt ist, werden zwei Bits gesendet, andernfalls nur eines.

ECHO (c_lflag)

Wenn es gesetzt ist, werden die Eingabezeichen gleich wieder an das Terminal zurückgesendet. Das kann sowohl im kanonischen als auch nicht-kanonischen Modus geschehen.

ECHOCTL (c_lflag)

Bestimmt, ob Kontrollzeichen auch wie bei ECHO zurück an das Terminal gesendet werden sollen. Dazu muß auch ECHO gesetzt sein. Die Zeichen werden, abgesehen von TAB, NL, START und STOP als $\text{^}X$ ausgegeben, wobei X das Zeichen repräsentiert, indem der oktale Wert 100 zu dem Kontrollzeichen addiert wird. Das heißt, daß STRG-A als $\text{^}A$ oder DEL als $\text{^}? (Oktal 177)$ ausgegeben wird. Diese Option gilt sowohl für den kanonischen als auch den nicht-kanonischen Modus.

ECHOE (c_lflag)

Wenn das Flag im kanonischen Modus (`ICANON`) gesetzt wird, wird das letzte Zeichen in der Zeile gelöscht. Das übernimmt meistens der Terminaltreiber mit der 3-Bytessequenz BACKSPACE-SPACE-BACKSPACE.

ECHOK (*c_lflag*)

Wenn dieses Flag im kanonischen Modus gesetzt ist, löscht Das KILL-Zeichen die ganze Zeile oder druckt alternativ ein NL-Zeichen zu Anzeige der gelöschten Zeile.

ECHOKE (*c_lflag*)

Wenn dieses Flag im kanonischen Modus gesetzt ist, wird das KILL-Zeichen ausgegeben, indem jedes Zeichen der Zeile gelöscht wird.

ECHONL (*c_lflag*)

Gibt im kanonischen Modus das NL-Zeichen aus, selbst wenn ECHO nicht gesetzt ist.

ECHOPRT (*c_lflag*)

Wenn das Flag zusammen mit ECHO und ICANON gesetzt ist, löscht das ERASE-Zeichen das Zeichen und zeigt die Löschung auf dem Terminal an.

FFDLY (*c_oflag*)

Verzögerung für den Zeilenvorschub (*form feed*). Gültige Werte sind FF0 oder FF1.

FLUSHO (*c_lflag*)

Wird gesetzt und leert die Ausgabe wird, wenn wir das DISCARD-Zeichen eingeben, wird ausgelöscht, wenn wir das Zeichen erneut eingeben.

HUPCL (*c_cflag*)

Wenn es gesetzt ist und der letzte offene Prozess das Terminal schließt wird die Verbindung des Modems geschlossen.

ICANON (*c_lflag*)

Aktiviert den kanonischen Modus (siehe 9.1.2).

ICRNL (*c_iflag*)

Wenn es gesetzt und IGNCR nicht gesetzt ist, wird ein CR- beim Empfang in ein NL-Zeichen umgewandelt.

IEXTEN (*c_lflag*)

Das Flag aktiviert die implementierungsspezifischen Spezialzeichen und schließt sie in die Verarbeitung mit ein.

IGNBRK (*c_iflag*)

Wenn es gesetzt ist, wird BREAK einfach ignoriert (siehe BRKINT).

IGNCR (*c_iflag*)

Ist das Flag gesetzt wird ein CR-Zeichen bei der Eingabe ignoriert. Nur wenn das IGNCR nicht, aber ICRNL gesetzt ist, kann ein CR in NL umgewandelt werden.

IGNPAR (*c_iflag*)

Wenn das Flag gesetzt ist werden Bytes mit einem Paritäts oder Rahmenfehlern (*framing errors*) ignoriert. Das trifft nicht für das BREAK-Zeichen zu.

IMAXBEL (*c_iflag*)

Gibt einen Piepton aus, wenn die Eingabewarteschlange voll ist.

INLCR (*c_iflag*)

Wandelt ein NL-Zeichen bei der Eingabe in ein CR-Zeichen um.

INPCK (*c_iflag*)

Führt eine Paritätsprüfung bei der Eingabe durch (siehe 9.2.1).

ISIG (*c_lflag*)

Prüft Eingabezeichen gegen solche, die Terminalsignale erzeugen (QUIT, SUSP, DSUSP und INTR) und wenn der Vergleich erfolgreich ist, wird das entsprechende Signal generiert.

ISTRIP (*c_iflag*)

Wenn das Flag gesetzt ist, werden die Eingababytes vor der Verarbeitung auf 7 Bits gekürzt.

IUCLC (*c_iflag*)

Das Flag sorgt für die Übersetzung von Großbuchstaben in Kleinbuchstaben.

IXANY (*c_iflag*)

Jedes Zeichen kann die Ausgabe anstoßen.

IXOFF (*c_iflag*)

Wenn das Flag gesetzt ist, wird die Start/Stop-Eingabekontrolle aktiviert. Wenn der Terminaltreiber feststellt, daß die Eingabewarteschlange voll wird, sendet er ein STOP-Zeichen. Dieses Zeichen wird von dem sendenden Gerät erkannt und hält das Gerät an. Wenn die Zeichen der Eingabewarteschlange verarbeitet wurden, sendet der Terminaltreiber ein START-Zeichen, daß den Sender veranlaßt mit der Datenübertragung fortzufahren.

IXON (*c_iflag*)

Wenn das Flag gesetzt ist, wird die Start/Stopausgabekontrolle aktiviert. Wenn der Terminaltreiber ein STOP-Zeichen empfängt, wird die Ausgabe angehalten. Das nächste empfangene START-Zeichen veranlaßt den Treiber zur Fortsetzung der Ausgabe. Ist das Flag nicht aktiviert, werden die beiden Zeichen als einfache Zeichen von dem Prozess gelesen.

MDMBUF (*c_cflag*)

Die Flußkontrolle erfolgt nach den Regeln des Carrier-Flags des Modems.

NLDLY (*c_oflag*)

Verzögerung für das NL-Zeichen. Gültige Werte sind entweder NL0 oder NL1.

NOFLSH (*c_lflag*)

Wenn der Terminaltreiber ein SIGINT- oder SIGQUIT-Signal absetzt werden sowohl Eingabe- als auch die Ausgabewarteschlangen geleert. Wird ein SIGSUSP generiert, wird die nur Eingabewarteschlange geleert. Ist dieses Flag gesetzt, findet keine Leerung der Queues statt, wenn ein solches Signal erzeugt wird.

NOKERNINFO (*c_lflag*)

Wenn das Flag gesetzt ist, wird das STATUS-Zeichen davon abgehalte, Informationen über die Prozessgruppe im Vordergrund zu schreiben. Dennoch wird, unabhängig von dem Flag, das SIGINFO-Signal an die Prozessgruppe im Vordergrund gesendet.

OCRNL (*c_oflag*)

Wandelt bei der Ausgabe ein CR-Zeichen in ein NL-Zeichen um.

OFDEL (*c_oflag*)

Wenn das Flag gesetzt ist, wird DEL als Füllzeichen verwendet, andernfalls NUL.

OFILL (*c_oflag*)

Wenn das Flag gesetzt ist, wird entweder DEL oder NUL als Füllzeichen für Verzögerungen anstelle einer zeitgesteuerten Verzögerung verwendet (siehe BSDLY, CRDLY, FFIDLY, NLDLY, TABDLY und VTDLY).

OLCUC (*c_oflag*)

Wandelt bei der Ausgabe Klein- in Großbuchstaben um.

ONLCR (*c_oflag*)

Wenn gesetzt wird NL in NL-CR umgewandelt.

ONLRET (*c_oflag*)

Wenn das Flag gesetzt ist, übernimmt NL die Funktion von CR.

ONOCR (*c_oflag*)

Ein CR wird nicht in Spalte 0 ausgegeben.

ONOEOF (*c_oflag*)

Wenn es gesetzt ist, wird das EOT-Zeichen (D) bei der Ausgabe verworfen. Das ist für manche Terminals notwendig, um die Tastenkombination STRG-D als Hangup (Auflegen) zu interpretieren.

OPOST (`c_oflag`)

Wenn das Flag gesetzt ist, wird implementierungsspezifische Ausgabeverarbeitung ermöglicht.

OXTABS (`c_oflag`)

Wenn das Flag gesetzt ist, werden TABs in Leerzeichen umgewandelt.

PARENB (`c_cflag`)

Schaltet die Paritätsgenerierung für ausgehende Zeichen und die Paritätsprüfung für eingehende Zeichen ein (siehe 9.2.1).

PARMRK (`c_iflag`)

Wenn dieses Flag gesetzt und `IGNPAR` nicht gesetzt ist, wird ein Zeichen mit Parität als 3-Bytesquenz gelesen und bei Paritätsfehler oder Rahmenfehler als fehlerhaftes Zeichen gekennzeichnet an die Applikation weitergeleitet.

PARODD (`c_cflag`)

Wenn das Flag gesetzt ist, wird sowohl für eingehende als auch ausgehende Zeichen eine ungerade Parität ermittelt. Andernfalls wird eine gerade (*even*) Parität generiert.

PENDIN (`c_lflag`)

Alle Zeichen die noch nicht von einem Prozess gelesen wurden, werden erneut ausgegeben, sobald das nächste Zeichen eingeht.

TABDLY (`c_oflag`)

Verzögerung für horizontale TABs. Gültige Werte sind TAB0, TAB1, TAB2 oder TAB3.

TOSTOP (`c_lflag`)

Wenn das Flag auf Systemen mit Job Control gesetzt ist, wird das `SIGTTOU`-Signal an die Prozessgruppe im Hintergrund gesendet, die gerade versucht auf das eigene Kontrollterminal zu schreiben. Standardmäßig werden alle Prozesse der Prozessgruppe angehalten.

VTDLY (`c_oflag`)

Verzögerung für vertikale TABs. Werte sind VT0 oder VT1.

XCASE (`c_lflag`)

Wenn das Flag zusammen mit `ICANON` gesetzt ist, wird davon ausgegangen, daß dieses Terminal nur mit Großbuchstaben arbeitet, so daß alle Eingaben in Kleinbuchstaben umgewandelt werden (dieses Flag ist überflüssig, da solche Terminals nicht mehr existieren).

9.2.6 Eingabezeichen

Nachdem wir in den vorangegangenen Abschnitten viel über Steuerzeichen, Flags, E/A-Modi gesprochen haben, wollen wir die theoretische Besprechung der Terminals abschließen. Dazu werfen wir nun einen Blick auf die speziellen Eingabezeichen, die bisher nur erwähnt, aber nicht genauer untersucht haben.

Die speziellen Eingabezeichen haben eine besondere Bedeutung bei der Eingabe, deren Verhalten teilweise über die Flags der `termios`-Struktur beeinflussen können. Tabelle 9.3 faßt die Zeichen und ihre Bedeutung zusammen.

Bis auf die Zeichen `\r` und `\n` können wir alle verändern, wie wir in einem der Beispiele sehen werden. Abhängig von der Implementierung können auch `START` und `STOP` geändert werden. Kernpunkt ist das Array `c_cc` der `termios`-Struktur. Wir legen die Zeichen in diesem Array mit ihren symbolischen Namen fest, die alle mit einem V beginnen.

9.3 Die Funktionen `tcgetattr` und `tcsetattr`

Mit den beiden Funktionen `tcgetattr(3)` und `tcsetattr(3)` fragen wir die Eigenschaften eines Terminals ab oder setzen sie.

Zeichen	Beschreibung	c_cc	Feld	Flag	Wert	POSIX
CR	Wagenrücklauf	-	c_lflag	ICANON	\r	•
DISCARD	Verwerfe Ausgaben	VDISCARD	c_lflag	IEXTEN	\0	
DSUSP	verzögertes Aussetzen (<i>suspend</i>)	VDSUSP	c_lflag	ISIG	\Y	
EOF	End of file	VEOF	c_lflag	ICANON	\D	•
EOL	End of line	VEOL	c_lflag	ICANON		•
EOL2	End of line 2	VEOL2	c_lflag	ICANON	\D	
ERASE	letztes Zeichen löschen	VERASE	c_lflag	ICANON	\H	•
INTR	Unterbrechung (SIGINT)	VINTR	c_lflag	ISIG	?/C	•
KILL	letzte Zeile löschen	VKILL	c_lflag	ICANON	\U	•
NL	Zeilenvorschub	-	c_lflag	ICANON	\n	•
QUIT	Beenden (SIGQUIT)	VQUIT	c_lflag	ISIG	\n	•
REPRINT	Ausgaben erneut ausgeben	VREPRINT	c_lflag	ICANON	\R	
START	mit Ausgabe fortfahren	VSTART	c_iflag	IXON/IXOFF	\Q	•
STATUS	Statusanfrage	VSTATUS	c_lflag	ICANON	\T	
STOP	Ausgabe anhalten	VSTOP	c_iflag	IXON/IXOFF	\S	•
SUSP	SIGTSTP absetzen	VSUSP	c_lflag	ISIG	\Z	
WERASE	Ein Wort löschen	VWERASE	c_lflag	ICANON	\W	

Tabelle 9.3: Spezielle Eingabezeichen

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termios_p);
int tcsetattr(int filedes, int optact, const struct termios *termios_p);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

filedes

File Descriptor auf ein Terminalgerät, dessen Eigenschaften abgefragt oder geändert werden sollen.

optact

Aktionen die nach dem Setzen von Einstellungen angewendet werden sollen.

termios_p

Zeiger auf eine **termios**-Struktur.

Der Parameter **optact** legt Aktionen fest, die bestimmen, wie die Einstellungen durch **tcsetattr(3)** angewendet werden sollen:

TCSANOW

Die Einstellungen werden sofort umgesetzt.

TCSADRAIN

Bevor die Einstellungen umgesetzt werden, sollen allen ausstehenden Ausgaben zuvor nach **filedes** geschrieben werden.

TCSAFLUSH

Bevor die Einstellungen umgesetzt werden, sollen allen ausstehenden Ausgaben zuvor nach **filedes** geschrieben und alle Eingaben, die zwar bisher empfangen, aber nicht gelesen wurden, sollen verworden werden.

Listing 9.2: Das INTR-Zeichen durch von STRG-C in STRG-G ändern.

```

1 #include <termios.h>
2 #include <unistd.h> /* for STDIN_FILENO */
3 #include "header.h"
4
5 int main(int argc, char *argv[]) {
6     struct termios t;
7
8     if (isatty(STDIN_FILENO) == 0)
9         err_fatal("we are not connected to stdin");
10
11    if (tcgetattr(STDIN_FILENO, &t) < 0)
12        err_fatal("tcgetattr failed");
13
14    t.c_cc[VINTR] = 7; /* 7 is CTRL-G (A = 1, B = 2, ... Z = 26 */
15
16    if (tcsetattr(STDIN_FILENO, TCSANOW, &t) < 0)
17        err_fatal("tcsetattr failed");
18
19    return (0);
20 }
```

Listing 9.2: xcode/changekey.c - Das INTR-Zeichen durch von STRG-C in STRG-G ändern.

Zunächst wollen wir uns davon überzeugen, daß die Standardeingabe mit einem Terminal verbunden ist. Dazu verwenden wir die Funktion `isatty`, die `true` (1) zurückliefert, wenn der übergebene Zeiger auf einen Stream mit einem Terminalgerät verbunden ist. Anschließend initialisieren wir die `termios`-Struktur mit `tcgetattr(3)` und greifen auf das `c_cc`-Array zu. Wir übergeben dem Index `VINTR` den Zahlenwert 7, der dem 7. Zeichen im Alphabet (G) entspricht. Damit können wir das `SIGINT`-Signal nun über die Tastenkombination `STRG-G` absetzen. Dazu müssen wir die Änderungen aber erst mit der Funktion `tcsetattr(3)` speichern. □

Das nächste Beispiel zeigt, wie wir gezielt einzelne Flags abfragen und setzen können.

Listing: 9.3: Flags in der termios-Struktur abfragen

Wir wollen nun herausfinden, ob die beiden Flags `ISTRIP` und `IXANY` in dem Member `c_iflag` gesetzt sind. Dazu initialisieren wir zuerst wieder eine `termios`-Struktur und fragen sie dann nach den gewünschten Flags.

```

1 #include <termios.h>
2 #include <unistd.h>
3 #include "header.h"
4
5 int main(int argc, char **argv) {
6     struct termios t;
7     int          istrong_set, ixany_set;
8
9     if (tcgetattr(STDIN_FILENO, &t) < 0)
10        err_fatal("tcgetattr failed");
11
12    istrong_set = t.c_iflag & ISTRIP;
13    ixany_set   = t.c_iflag & IXANY;
14
15    printf("ISTRIP flag %s\n", istrong_set ? "set" : "not set");
16    printf("IXANY  flag %s\n", ixany_set   ? "set" : "not set");
17
18    t.c_iflag &= ~IXANY;
19
20    if (tcsetattr(STDIN_FILENO, TCSANOW, &t) < 0)
21        err_fatal("tcsetattr failed");
```

```

23     return (0);
24 }
```

Listing 9.3: xcode/tcgetattr.c - Flags in der termios-Struktur abfragen.

Da die Flags immer logisch geodert (*OR*) werden, müssen wir die gleichen Mechanismen zur Abfrage oder zum setzten der Flags anwenden. Mit dem Bitoperator `&` fragen wir nach einem Flag. Um eines zu entfernen ist das Statement

```
t.c_iflag &= ~IXANY;
```

angebracht. Um zu sehen, ob unser Beispiel richtig funktioniert, können wir das Tool `stty(1)` verwenden:

```
% stty -a | grep iflags // vor der Ausfuehrung unseres Beispielprogramms
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
% ./tcgetattr // Beispielprogramm ausfuehren
ISTRIP flag not set
IXANY flag set
% stty -a | grep iflags // nach der Ausfuehrung unseres Beispielprogramms
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff -ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
```

Richten wir unser Augenmerk auf die Zeile `iflags`, vor und nach der Ausführung unseres Beispielprogramms. Ein Minuszeichen vor dem Flag zeigt an, daß es momentan nicht gesetzt ist, andernfalls ist es gesetzt. Übrigens dient das Programm `stty(1)` dazu, die aktuellen Konfigurationseinstellungen des Terminals auf der Standardausgabe anzuzeigen und zu setzen. □

Listing 9.4: Passwortabfrage bei deaktiviertem Echoing

Das folgende Beispiel zeigt, wie wir das Echoing deaktivieren, damit das Password bei der Abfrage nicht automatisch angezeigt wird. Hier verwenden wir ein `ioctl(2)`-Kommando, um die `termios`-Struktur zu befüllen.

```

1 #include <sys/ioctl.h>
2 #include <termios.h>
3 #include "header.h"
4
5 struct termio lock;
6 struct termio save;
7 char password[82];
8
9 int main(void) {
10     ioctl(STDIN_FILENO, TCGETA, &lock);
11     ioctl(STDIN_FILENO, TCGETA, &save);
12
13     printf("Enter password:");
14
15     while (checkpw())
16         printf("\r\nSorry, try again\r\n\r\n");
17
18     ioctl(STDIN_FILENO, TCSETA, &save);
19     printf("\nYour password is %s", password);
20
21     return (0);
22 }
23
24 int checkpw(void) {
25     int i;
26     char *p;
```

```

27     lock.c_iflag &= ~(BRKINT);
28     lock.c_lflag &= ~(ECHO | ISIG);
29     ioctl(STDIN_FILENO, TCSETA, &lock);
30
31     printf("Enter password: ");
32     if ((i = read(STDIN_FILENO, password, sizeof(password) - 1)) < 0)
33         i = 0;
34
35     password[i] = '\0';
36     printf("\r\nYou entered: ");
37
38     for (p = password; p < password + i; p++)
39         if (*p >= ' ' && *p < 127)
40             printf("%c", *p);
41         else
42             printf("^%c", *p == 127 ? '?' : *p + '@');
43
44     printf("\r\n");
45     return strcmp(password, "open\n");
46 }

```

Listing 9.4: xcode/term_echo.c - Passwortabfrage bei deaktiviertem Echoing.

□

9.4 Einstellung der Baudraten

Im Abschnitt 9.2.3 haben wir erfahren, daß wir die Baudraten eines Terminals mit Hilfe von vier unterschiedlichen Funktionen abfragen und setzen können:

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);
speed_t cfgetospeed(const struct termios *termios_p);

Rückgabewert: beide geben die Baudrate zurück.

int cfsetispeed(struct termios *termios_p, speed_t speed);
int cfsetospeed(struct termios *termios_p, speed_t speed);

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.
```

termios_p

ist eine Struktur vom Typ **termios**, die für das aktuelle Terminal initialisiert wurde.

speed

gibt an, welche Geschwindigkeit für das Terminal eingestellt werden soll. Ausgedrückt als symbolische Konstante.

Wie Sie vielleicht schon bemerkt haben wird der Term *Baud* anstelle von Bits pro Sekunde verwendet. Ein Baud entspricht nicht einem Bit pro Sekunde. Der Begriff ist in diesem Zusammenhang historisch gewachsen und wurde beibehalten. Die vier Funktionen **cfgetospeed(3)**, **cfgetispeed(3)**, **cfsetospeed(3)** und **cfsetispeed(3)** erwarten keine Zahlenwerte als Geschwindigkeitsangabe sondern eine der symbolischen Konstanten B0 bis B38400 wie in Tabelle 9.2 aufgelistet. Es gibt zwei Gründe dafür:

1. Früher wurde die Baudrate kodiert in der Struktur abgespeichert und ließ keinen direkten Zugriff zu.

2. Ein wichtigerer Punkt war die Portabilität, denn nur bestimmte Baudraten waren tatsächlich auf andere Systeme übertragbar.

Traditionell wurde die Baudrate durch ein Flag in dem Feld `c_cflag` kodiert. Wir können davon ausgehen, daß es auch noch heute der Fall ist, sollten uns aber nicht darauf verlassen, beispielsweise indem wir `c_cflag` einfach kopieren. Konsultieren Sie daher die Dokumentation des Systems um herauszufinden, in welchem Feld die Baudrate festgelegt wird. Der POSIX-Standard empfiehlt aber die Nutzung des `c_cflag`-Feldes.

Listing 9.5: Bestimmung der Ausgabegeschwindigkeit von `stdin`

Das folgende Beispiel illustriert wie die Baudrate eines Terminals mit `cfsetospeed(3)` ausgelesen werden kann.

```

1 #include <sys/types.h>
2 #include <termios.h>
3 #include <unistd.h>
4 #include <stdio.h>
5
6 int main() {
7     struct termios t;
8     speed_t      baudRate;
9     char         *inputSpeed = "unknown";
10
11     tcgetattr(STDIN_FILENO, &t);
12
13     /* Get the input speed. */
14     baudRate = cfgetispeed(&t);
15
16     /* Print input speed. */
17     switch (baudRate) {
18         case B0:      inputSpeed = "none"; break;
19         case B50:     inputSpeed = "50 baud"; break;
20         case B110:    inputSpeed = "110 baud"; break;
21         case B134:    inputSpeed = "134 baud"; break;
22         case B150:    inputSpeed = "150 baud"; break;
23         case B200:    inputSpeed = "200 baud"; break;
24         case B300:    inputSpeed = "300 baud"; break;
25         case B600:    inputSpeed = "600 baud"; break;
26         case B1200:   inputSpeed = "1200 baud"; break;
27         case B1800:   inputSpeed = "1800 baud"; break;
28         case B2400:   inputSpeed = "2400 baud"; break;
29         case B4800:   inputSpeed = "4800 baud"; break;
30         case B9600:   inputSpeed = "9600 baud"; break;
31         case B19200:  inputSpeed = "19200 baud"; break;
32         case B38400:  inputSpeed = "38400 baud"; break;
33     }
34
35     printf("Input speed = %s\n", inputSpeed);
36
37     return (0);
38 }
```

Listing 9.5: `xcode/term_speed.c` - Bestimmung der Ausgabegeschwindigkeit von `stdin`.



9.5 Die Funktion ctermid

Normalerweise sind alle Unix-Terminals über den Gerätenamen `/dev/tty` ansprechbar. Einige Implementierungen weichen evtl. von dieser Notation ab, liefern aber stets einen Link auf die richtige Gerätedatei. Um herauszufinden, wie das Terminal heißt bietet uns POSIX eine Funktion an, die eine Abfrage zur Laufzeit ermöglicht.

```
#include <stdio.h>

char *ctermid(char *s);
```

*Rückgabewert: abhängig von *s*

Wenn `s` ein NULL-Zeiger ist wird automatisch ein ausreichend großes, statisches char-Array durch das System allokiert und der Name des Terminals darin gespeichert. Ist `s` nicht NULL, wird davon ausgegangen, daß es auf ein ausreichend großes Array zeigt, in dem der Name des Terminals gespeichert werden kann. Die Größe des Arrays beträgt in der Regel `L_ctermid` Bytes. Die Konstante ist in `<stdio.h>` definiert. Dann der Name des Terminals aus irgendwelchen Gründen nicht ermittelt werden, wird ein Leerstring zurückgegeben.

Die Implementierung einer solchen Funktion ist denkbar einfach und auf allen Systemen (Implementierungen) nahezu identisch:

```
1 #include <stdio.h> /* for L_ctermid */
2 #include <string.h> /* for strcpy() */
3
4 char *ctermid(char *s) {
5     static char name[L_ctermid];
6
7     if (s == NULL) s = name;
8
9     return strcpy(s, "/dev/tty");
10 }
```

Listing 9.6: xcode/ctermidimpl.c - Implementierung der POSIX-Funktion ctermid.

Listing 9.7: Anwendung von ctermid(3)

Das folgende Beispiel verwendet `ctermid(3)`, um auf das aktuelle Terminal zuzugreifen und dann eine Nachricht an das interaktive Ausgabegerät zu senden.

```
1 #include "header.h"
2
3 int main(int argc, char *argv[]) {
4     FILE *termfile; /* open the terminal for writing */
5
6     /* assign name of interactive terminal to termfile. */
7     if ((termfile = fopen(ctermid(NULL), "w")) == NULL) {
8         err_fatal("fopen failed");
9     }
10
11    /* print message to interactive terminal */
12    fprintf(termfile, "This is a test message\n");
13
14    fclose(termfile);
15    return (0);
16 }
```

Listing 9.7: xcode/ctermid.c - Anwendung von ctermid(3).



9.6 Die Funktionen `ttyname` und `isatty`

Um herauszufinden, ob ein File Descriptor mit einem Terminal verbunden ist, können wir `isatty(3)` verwenden, wohingegen wir den Pfad zu einem Terminal, das mit einem File Descriptor verbunden ist, mit `ttyname(3)` abfragen.

```
#include <unistd.h>

int isatty(int fildes);
Rückgabewert: 1, wenn fildes mit einem Terminal verbunden ist, 0 bei Fehler

char *ttyname(int fildes);
Rückgabewert: Zeiger auf einen String oder NULL bei Fehler
```

Listing 9.8: Anwendung von `isatty(3)` und `ttyname(3)`

Das folgende kleine Beispiel zeigt, wie wir mit `isatty(3)` die drei Standard File Descriptor abfragen, um uns zu vergewissern, daß sie mit einem Terminal verbunden sind. Zum Test, übergeben wird dem Programm eine Datei, die zum Lesen geöffnet wird und damit natürlich nicht mit einem Terminal verbunden ist.

```
1 #include <unistd.h>
2 #include <string.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include "header.h"
6
7 int main(int argc, char **argv) {
8     int fd, i;
9     char message[128];
10    char *basename = basename_ex(argv[0]);
11
12    if (argc != 2)
13        err_fatal("Usage: %s <file to open>\n", basename);
14
15    if ((fd = open(argv[1], O_RDONLY)) == -1)
16        err_fatal("fopen failed.");
17
18    for (i = 0; i <= fd; i++) {
19        printf("FD %d: %s: %s\n",
20               i,
21               isatty(i) ? "is a TTY, connected to: " : " is not a TTY",
22               isatty(i) ? ttyname(i) : "");
23    }
24
25    return (0);
26 }
```

Listing 9.8: xcode/isatty.c - Anwendung von `isatty(3)` und `ttyname(3)`.

Die Ausgaben sehen aus, wie wir sie erwartet haben:

```
% isatty ./testfile
FD 0: is a TTY, connected to: /dev/pty0
FD 1: is a TTY, connected to: /dev/console
FD 2: is a TTY, connected to: /dev/pty1
FD 3: is not a TTY
```



Kapitel 10

Interprozesskommunikation - IPC

An education isn't how much you have committed to memory, or even how much you know. It's being able to differentiate between what you know and what you don't.

ANATOLE FRANCE

Die Kommunikation mit anderen Prozessen beschränkte sich in bisherigen Diskussionen nur auf Signale oder File Descriptoren, die wir anderen Prozessen beim Aufruf von `exec(3)` oder `fork(2)` weitergegeben haben. Eigentlich sind Signale auch keine Form der Kommunikation, denn sie informieren nur über bestimmte Ereignisse, enthalten aber selbst keinerlei zusätzliche Informationen.

In diesem Abschnitt betrachten wir verschiedene Ansätze, die es uns erlauben, Informationen mit anderen Prozessen auszutauschen. Die wichtigsten sind: Pipes, FIFOs (auch als *benannte Pipes* bekannt), Semaphoren, Shared Memory, Sockets und STREAMS (nicht zu verwechseln mit Streams der Standard-E/A-Bibliothek).

Von den genannten sechs Varianten sind nur fünf durch POSIX standardisiert. STREAMS sind nur auf einigen Systemen verfügbar. Um maximale Portabilität zugewährleisten sollten wir von STREAMS Abstand nehmen (auch wenn es sich um eine sehr leistungsfähige Technik handelt). Während Pipes, FIFOs, Semaphoren und Shared Memory nur lokale Bedeutung haben, sind STREAMS und Sockets tatsächlich in der Lage, Informationen zwischen Prozessen auf mehreren Hosts auszutauschen.

Tabelle 10.1 listet die Unterschiede der genannten Techniken auf.

10.1 Anonyme Pipes

Alle UNIX-Systeme stellen Pipes zu Verfügung. Sie sind seit Unix System II Bestandteil des AT&T UNIX (mehr Informationen zur Geschichte und Entstehung finden sie unter [Tho74] und [Rit79]).

Trotzdem sind die Nachteile seit dem die gleichen geblieben: Pipes können Informationen nur in eine Richtung übermitteln (Simplexkommunikation) und nur zwischen Prozessen, die den gleichen Vorfahren aufweisen, beispielsweise zwischen zwei Childs oder zwischen Parent und Child.

Doch für welche Anwendungen benötigen wir Pipes eigentlich? Der Begriff Interprozesskommunikation ist recht abstrakt. Die meisten UNIX-Programme erledigen (natürlich mit einigen Ausnahmen) nur eine kleine Aufgabe. Aber das machen sie gut. Mit Hilfe von Pipes sind wir in der Lage, mehrere kleine Programme miteinander zu verbinden, um eine größere Aufgabe zu erledigen. Das Grundprinzip ist einfach: wir verwenden die Ausgaben eines Programms als Eingabe für ein anderes. Betrachten wir als Beispiel die beiden Kommandos `ls` und `wc -l`. Ersteres gibt alle Dateien und Verzeichnisse im aktuellen Arbeitsverzeichnis aus und letzteres zählt die Anzahl der Zeilen. Um nun herauszufinden, wie viele Dateien und Verzeichnisse im aktuellen Ordner enthalten sind rufen wir einfach

IPC-Technik	Beschreibung
Anonyme Pipes	Richten eine Einwegekommunikation zwischen zwei verwandten Prozessen ein.
Benannte Pipes (FIFO)	Richten einen Zweiwegekommunikationskanal zwischen mehreren nicht miteinander verwandten Prozessen ein.
Signale	Benachrichtigung über asynchrone Ereignisse wie Software- oder Hardware-Interrupts
Shared Memory	Zur Erzeugung eines Speicherbereiches, der in den Adressraum zweier oder mehrerer Prozesse eingebettet (<i>mapping</i>) wird, so daß sie den Inhalt ansprechen und falls erforderlich verändern können.
Semaphoren	Software-Objekte, die für den synchronisierten Zugriff auf begrenzte Ressourcen verwendet werden.
Locks & Mutexes	Software-Objekte, die für den synchronisierten Zugriff auf eine Resource oder Code-Sequenz verwendet werden.
Message Queues	Software-Objekte für den Austausch von Nachrichten in festgelegter Reihenfolge.
Sockets	Stellt virtuelle Verbindungen (<i>connections</i>) zwischen Prozessen her, die sich auch auf unterschiedlichen Systemen befinden können.
STREAMS	Dient zum Aufbau von Kommunikationsdiensten zwischen Prozessen auf dem gleichen oder unterschiedlichen Systemen

Tabelle 10.1: IPC-Techniken in der UNIX-Welt.

```
% ls | wc -l
```

auf. Die Dateiliste, die wir normalerweise auf der Standardausgabe sehen würden, wird nun als Eingabe für das Programm `wc` verwendet. Das Ergebnis können wir wieder auf der Standardausgabe sehen. Das ganze funktioniert auch mit mehreren Befehlen; eine Grenze gibt es hier nicht. Beispiel:

```
% sort myfile | pr -3 | lp
```

Hier wird der Inhalt der Datei `myfile` sortiert, das Ergebnis vernünftig aufbereitet (`pr(1)`) und anschließend ausgedruckt (`lp(1)`).

Dieses Konzept ist auch für Entwickler interessant, denn möglicherweise existiert ein Programm, das eine gewünschte Aufgabe für uns erledigt, die wir uns zu Nutzen machen können.

Natürlich existiert zur Erzeugung von Pipes ein Systemaufruf:

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler.

`filedes[2]`

Integer-Array mit File Descriptoren, die für die Ein- und Ausgabe verwendet werden.

Einer der beiden Integer in `filedes` wird zum Lesen geöffnet, der andere zum Schreiben. Das heißt, die Ausgaben von `filedes[1]` werden als Eingaben an `filedes[2]` gesendet. Wichtig ist, daß die Optionen `O_NONBLOCK` und `O_CLOEXEC` beim Aufruf von `open(2)` für beide File Descriptoren nicht gesetzt wird. Natürlich können wir die Einstellungen nachträglich mit `fcntl(2)` anpassen.

Meistens werden Pipes im direkten Zusammenhang mit dem `fork(2)`-Systemaufruf benötigt, um einen Kommunikationskanal zwischen Parent und Child zu öffnen. Eine solche Verbindung ist in Abbildung 10.1 dargestellt. Dabei ist die Reihenfolge in der `fork(2)` und `pipe(2)` aufgerufen werden von Bedeutung.

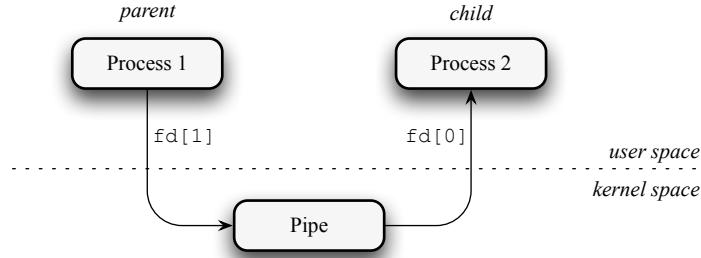


Abbildung 10.1: Einfache Pipe: Datenfluß vom Parent zum Child.

Listing 10.1: Einwege-Pipe zwischen Parent und Child

Das folgende Beispiel einzeugt einen einfachen Kanal zwischen Parent und Child. Der Elternprozess schreibt seinem Kind eine kleine Nachricht.

```

1 #include <unistd.h>
2 #include "header.h"
3
4 int main(void) {
5     int bytes_read, fd[2];
6     pid_t pid;
7     char buf[BUFSIZ], text[] = "This is your parent writing!\n";
8
9     if (pipe(fd) < 0)
10         err_fatal("pipe() failed");
11
12    if ((pid = fork()) < 0)
13        err_fatal("fork() failed");
14    else if (pid > 0) { /* parent code */
15        close(fd[0]); /* close reading end */
16        write(fd[1], text, sizeof(text));
17    } else if (pid == 0) { /* child code */
18        close(fd[1]); /* close writing end */
19
20        if ((bytes_read = read(fd[0], buf, BUFSIZ)) < bytes_read)
21            err_normal("short read");
22        else
23            write(STDOUT_FILENO, buf, bytes_read);
24    }
25
26    return (0);
27 }
```

Listing 10.1: *xcode/pipe-simplex.c* - Eltern schreiben gern ihren Kindern.

Damit ein Parent seinem Child schreiben kann, muß der Parent zuerst `fd[0]` (lesen) schließen und anschließend der Child-Prozess `fd[1]` (schreiben) schließen. □

Wenn wir versuchen, von einer Pipe zu lesen, dessen schreibendes Ende bereits geschlossen wurde, so liefert `read(2)` stets 0 zurück. Umgekehrt wird `SIGPIPE` abgesetzt, wenn wir versuchen, zu schreiben, aber das lesende Ende geschlossen wurde. Der Aufruf von `read(2)` kehrt zurück und `errno` zeigt `EPIPE` an.

Normalerweise verwenden nur zwei Prozesse die gleiche Pipe. Wie wir gleich sehen werden, ist es auch möglich, die File Descriptoren zu duplizieren, so daß rein technisch mehrere Schreiber/Leser die gleiche Pipe nutzen können. Versuchen nun mehrere Prozesse gleichzeitig in die Pipe zu schreiben, müssen wir wissen, wie der Buffer des Kernels verwaltet wird, damit die Daten in der richtigen Reihenfolge im

Buffer landen. Die Konstante `_PC_PIPE_BUF` gibt Aufschluß über die Größe des Puffers. Solange maximal `_PC_PIPE_BUF` Bytes oder weniger geschrieben werden, ist sichergestellt, daß der Schreibvorgang nicht von anderen Prozessen unterbrochen wird. Überschreiten wir die Grenze, kann das nicht mehr ausgeschlossen werden. Grundsätzlich wird der Kernel-seitige Buffer in einer atomaren Operation gefüllt.

Listing 10.2: Geschwister schreiben sich untereinander auch gern

Betrachten wir nun ein Beispiel, in dem zwei Children miteinander kommunizieren. Hier erzeugt der Elternprozess zwei Kinder und verknüpft sie so, daß dem einen Child die Ausgaben des anderen übergeben werden (Abbildung 10.2).

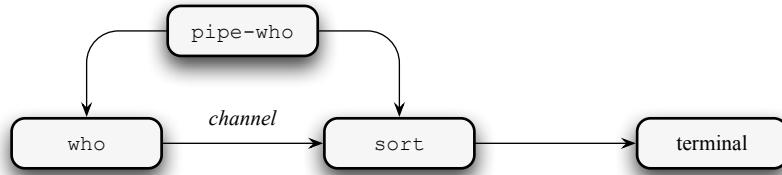


Abbildung 10.2: Kommunikation zwischen zwei Children.

```

1 #include <unistd.h>
2 #include <wait.h>
3 #include "header.h"
4
5 int main(int argc, char *argv[]) {
6     int pidA, pidB, rc = 0, fd[2], ret_val, status;
7
8     if (pipe(fd) < 0)
9         err_fatal("pipe failed");
10    else if ((pidA = fork()) < 0)
11        err_fatal("fork failed (A)");
12    else if (pidA > 0) { /* parent */
13        if ((pidB = fork()) < 0) { /* fork another child */
14            err_fatal("fork failed");
15        } else if (pidB > 0) { /* parent */
16            if (close(fd[0]) < 0) /* read end */
17                err_fatal("close failed for fd[0] (B)");
18            else if (close(fd[1]) < 0)/* write end */
19                err_fatal("close failed for fd[1] (B)");
20            else if (waitpid(pidB, &status, WNOHANG) > 0) {
21                /* fetch value passed to exit */
22                ret_val = WEXITSTATUS(status);
23                printf("Child returned: %d\n", ret_val);
24                exit(1);
25            }
26        } /* child code (B) */
27        if (close(STDIN_FILENO) < 0)
28            err_fatal("close failed for STDIN_FILENO");
29        else if (dup(fd[0]) < 0)
30            err_fatal("dup failed for fd[0]");
31        else if (close(fd[0]) < 0)
32            err_fatal("close failed for fd[0]");
33        else if (close(fd[1]) < 0)
34            err_fatal("close failed for fd[1]");
35        else if (execl("/bin/sort", "sort", 0) < 0)
36            err_fatal("execl failed for sort");
37    }
  
```

```

38     } else { /* child code (A) */
39         if (close(STDOUT_FILENO) < 0)
40             err_fatal("close failed for STDIN_FILENO");
41         else if (dup(fd[1]) < 0)
42             err_fatal("dup failed for fd[0]");
43         else if (close(fd[0]) < 0)
44             err_fatal("close failed for fd[0]");
45         else if (close(fd[1]) < 0)
46             err_fatal("close failed for fd[1]");
47         else if (execl("/usr/bin/who", "who", 0) < 0)
48             err_fatal("execl failed for who");
49     }
50
51     return (0);
52 }
```

Listing 10.2: xcode/pipe-who.c - Geschwister schreiben sich untereinander auch gern.

Ein Child führt den UNIX-Befehl `who` aus und das andere Kind wartet auf das Ergebnis um damit den UNIX-Befehl `sort` zu versorgen. Das entspricht dem Kommando

```
% sort | who
```

in der Shell. □

Pipes zu erzeugen ist eine wiederkehrende Aufgabe, die nicht besonders erbauend ist und nur Zeit kostet, wie im vorangegangenen Beispiel gesehen. Deshalb wurden uns zwei Funktionen spendiert, die uns die Arbeit erleichtern: `popen(2)` und `pclose`. Folgende Aufgaben werden von den beiden Funktionen erledigt:

- `popen(2)` erzeugt eine Pipe, führt ein `exec(3)` durch und gibt einen Zeiger auf einen der beiden Standard-E/A-Streams (`STDIN_FILENO` oder `STDOUT_FILENO`) zurück.
- `pclose(2)` schließt den Eingabestrom `STDIN_FILENO` und wartet auf die Beendigung des durch `exec()` ausgeführten Kommandos.

Die Funktionen weisen folgende Synopsis auf:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

Rückgabewert: Zeigen auf einen Stream oder NULL bei Fehler.
```

```
int *pclose(FILE *stream);

Rückgabewert: Status von command oder -1 bei Fehler.
```

command

Bestimmt den auszuführenden Befehl. Wird an `exec(3)` übergeben.

type

Zeigt an, in welcher Richtung die Streams verbunden werden: `r` zum Schreiben, `w` zum Lesen.

stream

Zeigt an, welcher Stream geschlossen werden soll.

Das Argument `type` ist entscheidend für die Kommunikationsrichtung. Ist `type „r“` wird `command` mit `STDIN_FILENO` verbunden. Ist `type „w“` wird `command` mit `STDOUT_FILENO` verbunden.

Listing 10.3: Anwendung von popen(2)

Das folgende kleine Beispiel illustriert die Anwendung von `popen(2)` indem wir eine Datei laden, dessen Inhalt wir an das Programm `wc(1)` übergeben, das uns anzeigt, wie viele Zeilen in der Datei enthalten sind.

```

1 #include <sys/wait.h>
2 #include "errors.h"
3
4 #define MAX_BUF 1024
5
6 int main(int argc, char **argv) {
7     char buf[MAX_BUF];
8     char *basename = basename_ex(argv[0]);
9     FILE *strin, *strout;
10
11    if (argc != 2)
12        err_fatal("Usage: %s <path>\n", basename);
13
14    if ((strin = fopen(argv[1], "r")) == NULL)
15        err_fatal("fopen failed for %s", argv[1]);
16
17    if ((strout = popen("wc -l", "w")) == NULL)
18        err_fatal("popen failed");
19
20    while (fgets(buf, MAX_BUF, strin) != NULL)
21        if (fputs(buf, strout) == EOF)
22            err_fatal("fputs failed");
23
24    if (ferror(strin))
25        err_fatal("fgets failed");
26
27    if (pclose(strout) == -1)
28        err_fatal("pclose failed");
29
30    return (0);
31 }
```

Listing 10.3: xcode/popen.c - Anwendung von `popen(2)`.

Als Parameter übergeben wir dem Programm die zu untersuchende Datei:

```
% ./popendemo popendemo.c
32
```

Das Ergebnis ist die Anzahl der Zeilen in der Datei `popendemo.c`. □

Listing 10.4: Beispiimplementierung der Funktionen `popen(2)` und `pclose(2)`

Das Listing 10.4 zeigt eine Implementierung der Funktionen `popen(2)` und `pclose(2)`. Sie basieren auf den bisher erworbenen Kenntnissen und zeigen uns wichtige Details, die für das Verständnis von Pipes von Bedeutung sind. Immer wenn `popen(2)` aufgerufen wird, initiieren wir `fork(2)` und ordnen die jeweiligen File Descriptoren den Standardstreams für Parent und Child zu. Das erledigen wir mit `dup(2)`. Wenn als Modus „r“ angegeben wurde, verbinden wir `fd[1]` mit `STDOUT_FILENO`, andernfalls `fd[0]` mit `STDIN_FILENO`. Die Funktion `popen(2)` ist wenig spektakulär. Wir schließen den als Argument übergebenen File Descriptor und warten anschließend auf den Child-Prozess. Aufmerksame LeserInnen könnten nun anmerken, daß wir in `pclose(2)` gar nicht prüfen, ob `popen(2)` jemals aufgerufen wurde. Das läßt sich einfach realisieren, indem eine globale Variable eingeführt wird, die die Child-PID speichert, so daß wir diesen Fall abfangen könnten. Das haben wir uns hier gespart, da ohnehin -1 zurückgegeben wird, wenn der Aufruf von `wait` fehlschlägt.

```

1 #include <errno.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 FILE *popen (const char *command, const char *mode) {
8     int fd[2];
9     FILE *file; /* return value */
10
11     if (mode[0] != 'r' && mode[0] != 'w') {
12         errno = EINVAL; /* invalid mode string */
13         return NULL;
14     }
15
16     if (pipe(fd))
17         return NULL; /* error in pipe */
18
19     switch (fork())
20     {
21     case -1 : /* error in fork */
22         close(fd[0]);
23         close(fd[1]);
24         return NULL;
25
26     case 0 : /* child code */
27         if (mode[0] == 'r') {
28             dup2(fd[1], 1); /* dup pipe to STDOUT_FILENO */
29         } else {
30             dup2(fd[0], 0); /* dup pipe to STDIN_FILENO */
31         }
32         close(fd[0]); /* close other ends */
33         close(fd[1]);
34
35         execl("/bin/sh", "sh", "-c", command, (char *) NULL);
36         _exit(1); /* if we get here, there was an error in exec */
37
38     default : /* parent code */
39         if (mode[0] == 'r') { /* open reading end of pipe */
40             close(fd[1]);
41             if (!(file = fdopen(fd[0], mode)))
42                 close (fd[0]);
43         } else { /* open writing end of pipe */
44             close(fd[0]);
45             if (!(file = fdopen(fd[1], mode)))
46                 close (fd[0]);
47         }
48     }
49
50     return file;
51 }
52
53 int pclose (FILE *file) {
54     int status;
55
56     fclose(file);
57
58     if (wait(&status)) /* wait for child to terminate */
59         return -1; /* error in wait */
60
61     if (WIFEXITED(status))

```

```

62         return WEXITSTATUS(status);
63
64     errno = ECHILD;
65     return -1;
66 }

```

Listing 10.4: xcode/popenimpl.c - Beispielimplementierung der Funktionen `popen(2)` und `pclose(2)`.

Listing 10.5: Hauptprogramm für myecho

Abschließend betrachten wir ein Beispiel, indem der Eingabe ein Programm vorgeschaltet wird, das nichts anderes macht, als die Eingabe erneut auszugeben, quasi als Echo.

```

1 #include <sys/wait.h>
2 #include "errors.h"
3
4 #define MAX_BUF 1024
5
6 int main(int argc, char **argv) {
7     char buf[MAX_BUF];
8     char *basename = basename_ex(argv[0]);
9     FILE *strin;
10
11    if (argc != 2)
12        err_fatal("Usage: %s <filter>\n", basename);
13
14    if ((strin = popen(argv[1], "r")) == NULL)
15        err_fatal("popen failed");
16
17    while (1) {
18        if (fgets(buf, MAX_BUF, strin) == NULL)
19            break;
20
21        if (fputs(buf, stdout) == EOF)
22            err_fatal("fputs failed");
23    }
24
25    /* cleanup */
26    if (pclose(strin) < 0)
27        err_fatal("pclose failed");
28
29    exit(0);
30 }

```

Listing 10.5: xcode/echofilter.c - Hauptprogramm für myecho.

Unser Hilfsprogramm zur Ausgabe des Echos liest von der Eingabe und gibt sie wieder aus.

```

1 #include "errors.h"
2
3 int main(void) {
4     int c;
5
6     while ((c = getchar()) != EOF)
7         if (putchar(c) == EOF)
8             err_fatal("putchar failed in myecho");
9
10    return(0);
11 }

```

Listing 10.6: xcode/myecho.c - Hilfsprogramm myecho.

Ausgabe von Listing 10.5 und 10.6:

```
% bin/echofilter bin/myecho
This is a test. // ENTER
This is a test. // STRG-D
```

Anders als wir es von den Beispielprogrammen der Standard-E/A-Bibliothek her kennen, bei denen wir einfach alles was von der Standardeingabe kam ohne Verzögerung an die Standardausgabe weitergegeben haben, müssen wir in diesem Beispiel erst die Eingabe beispielsweise mit D (STRG-D) beenden, denn es wird immer das gesamte Ergebnis an das verknüpfte Programm (hier: `myecho`) geliefert. □

10.2 Benannte Pipes (FIFOs)

Benannte Pipes (**named pipes**) haben einen gravierenden Nachteil anonymer Pipes auf: nun können auch Prozesse, die nicht miteinander verwandt sind kommunizieren. Dabei wird eine Warteschlange nach dem FIFO-Prinzip (**first-in first-out**) eingesetzt um Eingaben mehrerer Prozesse zu verwalten (daher wird oft einfach von FIFOs gesprochen). Tatsächlich handelt es sich bei benannten Pipes um eine Halb-Duplex-Kommunikation.

Benannte Pipes werden durch einen Zugriffspunkt im Dateisystem identifiziert. Das ist eine der wichtigsten Eigenschaften benannter Pipes, denn weil sie im Dateisystem verankert sind, können nicht miteinander verwandte Prozesse kommunizieren. Daraus können wir schließen, daß die involvierten Prozesse wahrscheinlich einfach nur die benannte Pipe öffnen und über die File Descriptoren darauf zugreifen. Im Gegensatz zu anonymen Pipes bleiben FIFOs auch nach Beendigung des Prozesses erhalten und sind nicht an die Lebensdauer die Prozesse gebunden. FIFOs können mit einem Funktionssufruf erzeugt und entfernt oder bei Bedarf auch über die Kommandozeile gelöscht werden.

FIFOs sind ebenso einfach zu handhaben wie anonyme Pipes, denn jener Prozess, der die FIFO lesend öffnet ist der Konsument der Pipe und der oder die Prozesse, die in die FIFO schreiben sind Produzenten.

Eine wichtige Eigenschaft benannter Pipes ist die Tatsache, daß ein Prozess standardmäßig blockiert, wenn er versucht, sie zu öffnen und kein Konsument oder Produzent vorhanden ist. Der Prozess, der eine FIFO zum Lesen öffnet, geht in den Zustand *Blocking* über, bis ein Prozess das andere Ende zum Schreiben öffnet und umgekehrt.

10.2.1 Benannte Pipes erzeugen

Es gibt zwei Möglichkeiten eine benannte Pipe zu erzeugen: von der Kommandozeile aus oder über einen Systemaufruf. Uns interessiert natürlich letztere Variante, wir wollen uns aber dennoch beide Variante anschauen.

10.2.1.1 Benannte Pipes über die Kommandozeile erstellen

Auf der Kommandozeile können wir zwei unterschiedliche Befehle verwenden, um benannte Pipes zu erzeugen: `mknod(1)` und `mkfifo(1)`. Um eine Datei mit dem Namen `mypipe` zu erzeugen, würden wir entweder

```
% mknod mypipe p
```

oder

```
% mkfifo mypipe
```

verwenden. Selbstverständlich dürfen wir auch einen absoluten Pfadnamen angeben, denn FIFOs sind nicht an eine bestimmte Position im Dateisystem gebunden.

Wir überprüfen die erfolgreiche Erstellung unserer FIFO in dem wir einfach den Befehl `ls -lF mypipe` eingeben:

```
% ls -lF mypipe
prw-r--r-- 1 graegerts staff      0 Oct 2  9:01 mypipe
```

Das pām Anfang der Ausgabe zeigt an, daß die Datei ein Special File vom Typ *Pipe* ist. Außerdem können wir auch die Zugriffsrechte sehen, die der einer normalen Datei entsprechen.

10.2.1.2 Benannte Pipes über einen Systemaufruf erstellen

Eine benannte Pipe erstellen wir mit der Funktion `mkfifo(3)`.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Rückgabewert: 0 bei Erfolg oder -1 bei Fehler.

pathname

Pfad im Dateisystem. Dort wird die Pipe angelegt.

mode

Zugriffsmodus und -rechte für die Pipe (siehe Tabellen 3.1 und 3.2).

Die Funktion wird in der Regel mit dem Modus von `O_CREAT | O_EXCL` aufgerufen, der dafür sorgt, daß die Pipe erzeugt wird und einen Fehler ausgibt, wenn sie schon existiert. Desweiteren werden Benutzer- und Gruppen-ID auf die effektiven IDs des Prozesses gesetzt. Ist das `S_ISGID`-Bit des übergeordneten Verzeichnisses gesetzt, so wird es der Pipe vererbt.

10.2.2 Benannte Pipes öffnen

Das Öffnen einer FIFO zum Lesen und Schreiben funktioniert genau so wie bei regulären Dateien. Uns stehen die Funktionen `open(2)` und `fopen(3)` aus der Standard-E/A-Bibliothek zur Verfügung. Ist der Aufruf erfolgreich erhalten wir im Fall von `open(2)` einen File Descriptor auf die Pipe und im Fall von `fopen(3)` einen Zeiger auf eine FILE-Struktur. Aus Sicht der AnwenderInnen kann eine Pipe nach ihrer Erstellung fast wie eine reguläre Datei behandelt werden.

10.2.3 Von FIFOs lesen und in FIFOs schreiben

Wie das Öffnen benannter Pipes, wird auch das Lesen und Schreiben mit Hilfe der Standardfunktionen `read(2)` und `write(2)` vorgenommen. Diese Operationen sind standardmäßig blockierend, denn jedes Ende wartet auf das andere, damit entweder gelesen oder geschrieben werden kann.

Eine benannte Pipe kann von ein und dem selben Prozess nicht gleichzeitig zum Schreiben und Lesen geöffnet werden. Er kann immer nur Lesen oder Schreiben und die Pipe verbleibt so lange in diesem Modus bis sie ausdrücklich geschlossen wird. Falls ein Prozess von einer Pipe lesen möchte, jedoch keine Daten geliefert werden, geht dieser Prozess in den Zustand Blocking über. Das gleiche geschieht auch beim Schreiben, wenn auf der anderen Seite kein Prozess lesen möchte. Wie Sie vielleicht schon vermutet haben, kann ein solches Verhalten mit der Option `O_NONBLOCK` unterbunden werden. Naturgemäß ist Seeking (siehe Abschnitt 3.2.1 `seek`) mit benannten Pipes nicht möglich.

10.2.4 Einrichtung einer Vollduplexkommunikation

Obwohl eingangs von einer Halbduplexkommunikation (nur Senden oder Empfangen zu einem bestimmten Zeitpunkt) gesprochen wurde, ist es dennoch möglich eine Vollduplexkommunikation einzurichten, indem einfach eine zweite Pipe hinzugenommen wird. Das verkompliziert die Sache natürlich und wir müssen aufmerksam sein, damit keine Deadlocks auftreten, denn die Reihenfolge, in der Konsumenten oder Produzenten geöffnet werden, ist von Bedeutung.

Nehmen wir an, wir hätten gerade die beiden Pipes `Pipe1` und `Pipe2` erzeugt. Um nun einen Vollduplexkommunikationskanal aufzubauen, gehen wir folgendermaßen vor:

1. Der Produzent öffnet `Pipe1` zum Lesen und `Pipe2` zum Schreiben.
2. Der Konsument öffnet wiederum `Pipe1` zum Schreiben und `Pipe2` zum Lesen.

In anderen Konstellationen kann es zu Deadlocks kommen. Die beiden Kommunikationsformen (halb- und vollduplex) sind in Abbildung 10.3 dargestellt.

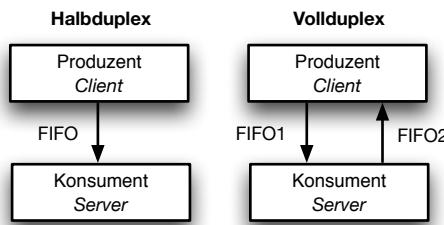


Abbildung 10.3: Halbduplex- und Vollduplexkommunikation mit FIFOs.

10.2.5 Vor- und Nachteile benannter Pipes

In der Einführung haben wir einen Vorteil bereits kennengelernt: die Prozesse müssen nicht mehr miteinander verwandt sein, um Daten auszutauschen.

Andere Vorteile wären beispielsweise:

- die einfache Anwendung,
- benannte Pipes müssen nicht synchronisiert werden,
- Pipes können Zugriffsrechte zugeordnet werden um die Sicherheit der Kommunikation zu erhöhen.
- ein Aufruf von `write(2)` wird atomar ausgeführt,
- und die Thread-Sicherheit der Funktion `mkfifo(3)`.

Bei aller Euphorie weisen benannte Pipes auch einige Einschränkungen auf:

- Benannte Pipes stehen nur Prozessen des selben Systems zur Verfügung.
- FIFOs können nur auf einem lokalen Dateisystem erstellt werden und nicht auf NFS-Shares.
- Es wird zur Kommunikation ein Byte-Stream verwendet, der keine Abgrenzung der Datensätze ermöglicht.

10.2.6 Beispiel einer Halbduplexverbindung

Bevor wir zur Vollduplexverbindung kommen, betrachten wir zunächst die einfache Variante. Dazu erzeugt der Produzent (oftmals auch als Server bezeichnet) eine benannte Pipe, öffnet sie zum Lesen und wartet darauf, daß der Konsument (auch Client genannt) die Daten schreibt. Sobald der Client die Daten gesendet hat, verwandelt der Server die Großbuchstaben in Kleinbuchstaben, die auf `STDOUT_FILENO` ausgegeben werden. Der Client öffnet die gleiche Pipe zum Lesen und empfängt die Zeichenkette, die der Server abgesetzt hat.

Listing 10.7: Server-Code

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <ctype.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include "header.h"
7
8 #define PIPE_NAME    "/tmp/fifohd"
9 #define MAX_BUF      512
10
11 int main(void) {
12     int fd, result, nread, i = 0;
13     char buf[MAX_BUF];
14
15     /* step 1: create a named pipe */
16     if (mkfifo(PIPE_NAME, PIPE_MODE) < 0)
17         err_fatal("mkfifo failed in server");
18
19     /* step 2: open pipe for reading */
20     if ((fd = open(PIPE_NAME, O_RDONLY)) < 0)
21         err_fatal("open of %s failed in server", PIPE_NAME);
22
23     /* step 3: read from first pipe */
24     if ((nread = read(fd, buf, MAX_BUF)) == 0)
25         err_fatal("Short read");
26
27     printf("SERVER: string received: %s\n", buf);
28
29     while (i < nread) { /* conversion */
30         buf[i] = tolower(buf[i]);
31         i++;
32     }
33
34     printf("SERVER: converted: %s\n", buf);
35
36     if (remove(PIPE_NAME) < 0) /* cleaning up */
37         err_fatal("could not remove pipe");
38
39     return (0);
40 }
```

Listing 10.7: xcode/fifohdsrv.c - Code des Servers.

Listing 10.8: Client-Code

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <ctype.h>
```

```

4 #include <fcntl.h>
5 #include <unistd.h>
6 #include "errors.h"
7
8 #define PIPE_NAME    "/tmp/fifohd"
9
10 int main(int argc, char **argv) {
11     char *basename = basename_ex(argv[0]);
12     int fd, written;
13
14     if (argc != 2)
15         err_fatal("Usage: %s <string to send>", basename);
16
17     if ((fd = open(PIPE_NAME, O_WRONLY)) < 0)
18         err_fatal("open failed in client");
19
20     if ((written = write(fd, argv[1], sizeof(argv[1]))) == 0) {
21         err_fatal("read failed in client");
22
23         if (written != strlen(argv[1]))
24             err_normal("short write: %d", written);
25     }
26
27     return (0);
28 }
```

Listing 10.8: xcode/fifohdcli.c - Code des Clients.

Wir starten den Server im Hintergrund mit dem Befehl

```
% fifohdsrv&
```

und starten den Client und übergeben den zu sendenden String mit

```
% fifohdcli "PROGRAMMING IS FUN!"
SERVER: string received: PROGRAMMING IS FUN!
SERVER: converted: programming is fun!
```

□

10.2.7 Beispiel einer Vollduplexverbindung

Wir nehmen das vorangegangene Beispiel auf und modifizieren es so, daß eine Zweiwegeverbindung entsteht. Der Server erzeugt zwei benannte Pipes: über eine wird gelesen, über die andere empfangen. Zuerst öffnet der Server eine Pipe zum Lesen und die zweite Pipe, um dem Client zu schreiben. Anschließend wartet der Server am lesenden Ende auf Daten den Clients. Sobald Daten verfügbar sind wird die übermittelte Zeichenkette konvertiert. Der Client wiederum öffnet die erste Pipe zum Schreiben und die zweite zum Lesen der Daten, die der Server schreibt. Das Szenario wird in Abbildung 10.4 dargestellt.

Listing 10.9: Server-Code

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <ctype.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include "header.h"
7
8 #define PIPE_NAME_READ    "/tmp/fifofd1"
```

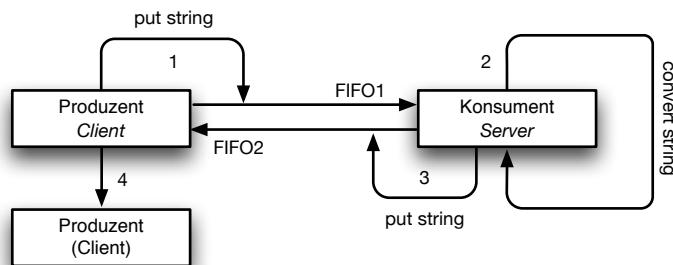


Abbildung 10.4: Vollduplexkommunikation mit FIFOs.

```

9  #define PIPE_NAME_WRITE "/tmp/fifofd2"
10
11 int main(void) {
12     int fdread, fdwrite, result, nread, written, i = 0;
13     char buf[MAX_BUF];
14
15     umask(0); /* reset mask for process */
16
17     /* step 1: create first named pipe */
18     if (mkfifo(PIPE_NAME_READ, PIPE_MODE) < 0)
19         err_fatal("mkfifo failed for %s", PIPE_NAME_READ);
20
21     /* step 2: create second named pipe */
22     if (mkfifo(PIPE_NAME_WRITE, PIPE_MODE) < 0)
23         err_fatal("mkfifo failed for %s", PIPE_NAME_WRITE);
24
25     /* step 3: open first pipe for reading */
26     if ((fdread = open(PIPE_NAME_READ, O_RDONLY)) < 0)
27         err_fatal("open of %s failed in server", PIPE_NAME_READ);
28
29     /* step 4: open second pipe for writing */
30     if ((fdwrite = open(PIPE_NAME_WRITE, O_WRONLY)) < 0)
31         err_fatal("open of %s failed in server", PIPE_NAME_WRITE);
32
33     /* step 5: read from first pipe */
34     if ((nread = read(fdread, buf, MAX_BUF)) == 0)
35         err_fatal("Short read");
36
37     printf("SERVER: string received: %s\n", buf);
38
39     while (i < nread) { /* conversion */
40         buf[i] = tolower(buf[i]);
41         i++;
42     }
43
44     /* step 6: write to second pipe */
45     if ((written = write(fdwrite, buf, strlen(buf))) == 0) {
46         err_fatal("write failed in server");
47
48         if (written != strlen(buf))
49             err_normal("short write: %d", written);
50     }
51
52     if (remove(PIPE_NAME_READ) < 0) /* cleaning up */
53         err_fatal("could not remove %s", PIPE_NAME_READ);
54
55     if (remove(PIPE_NAME_WRITE) < 0) /* cleaning up */
56         err_fatal("could not remove %s", PIPE_NAME_WRITE);

```

```

57         return (0);
58     }
59 }
```

*Listing 10.9: xcode/fifofdsrv.c - Code des Servers.***Listing 10.10: Client-Code**

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <ctype.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include "header.h"
7
8 #define PIPE_NAME_READ    "/tmp/fifofd1"
9 #define PIPE_NAME_WRITE   "/tmp/fifofd2"
10
11 int main(int argc, char **argv) {
12     char *basename = basename_ex(argv[0]);
13     char buf[MAX_BUF];
14     int fdread, fdwrite, written, nread;
15
16     if (argc != 2)
17         err_fatal("Usage: %s <string to send>\n", basename);
18
19     /* step 1: open first pipe for writing */
20     if ((fdwrite = open(PIPE_NAME_WRITE, O_WRONLY)) < 0)
21         err_fatal("open failed for %s", PIPE_NAME_WRITE);
22
23     /* step 2: open first pipe for writing */
24     if ((fdread = open(PIPE_NAME_READ, O_RDONLY)) < 0)
25         err_fatal("open failed for %s", PIPE_NAME_READ);
26
27     /* step 3: write to first pipe */
28     if ((written = write(fdwrite, argv[1], strlen(argv[1]))) == 0) {
29         err_fatal("write failed in server");
30
31         if (written != strlen(argv[1]))
32             err_normal("short write: %d", written);
33     }
34
35     /* step 4: read from second pipe */
36     if ((nread = read(fdread, buf, MAX_BUF)) == 0)
37         err_fatal("Short read");
38
39     printf("CLIENT: string received: %s", buf);
40     return (0);
41 }
```

Listing 10.10: xcode/fifofdcli.c - Code des Clients.

Wir führen Server und Client folgendermaßen aus:

```
% fifofdsrv&
% fifofdcli "PROGRAMMING IS FUN!"
SERVER: string received: PROGRAMMING IS FUN!
CLIENT: converted: programming is fun!
```



10.3 Shared Memory

Shared Memory (gemeinsam genutzter Speicher) ist eine einfache Möglichkeit, Daten mehreren Programmen (Prozessen) zur Verfügung zu stellen. Ein Prozess richtet einen bestimmten Bereich (Segment) des zur Verfügung stehenden Arbeitsspeichers für den Datenaustausch ein und andere Prozesse, die erforderliche Rechte aufweisen, können auf diesen zugreifen und möglicherweise auch verändern.

Shared Memory ist eine Technik, die besonders in IO-intensiven Anwendungen genutzt wird, um die Schreib- und Lesezugriffe auf Dateien zu verringern, beispielsweise bei Datenbankanwendungen, die große Datendateien in den Hauptspeicher laden, um mehreren Threads Zugriff zu gewähren, ohne den Overhead einer Schreib- und Leseanforderung.

Es stehen uns vier unterschiedliche Funktionen für die Erzeugung, Verwaltung und Freigabe von Shared Memory zur Verfügung:

- shmget**
Allociert ein neues Shared Memory Segment.
- shmat**
Verknüpft ein Shared Memory Segment mit dem Adressraum des aufrufenden Prozesses.
- shmdt**
Löst ein Shared Memory Segment aus dem Adressraum des aufrufenden Prozesses.
- shmctl**
Dient zur Verwaltung eines Shared Memory Segments.

Die Funktionsprototypen sind in `<sys/shm.h>` definiert. Die Struktur `shmid_ds` identifiziert das Shared Memory Segment und wird von einigen der genannten Funktionen als Parameter erwartet. Sie weist folgende Member auf:

```
struct shmid_ds {
    struct ipc_perm shm_perm /* Operation permission structure. */
    size_t        shm_segsz /* Size of segment in bytes. */
    pid_t         shm_lpid  /* Process ID of last shared mem op. */
    pid_t         shm_cpid  /* Process ID of creator. */
    shmat_t       shm_nattch /* Number of current attaches. */
    time_t        shm_atime /* Time of last shmat(). */
    time_t        shm_dtime /* Time of last shmdt(). */
    time_t        shm_ctime /* Time of last change by shmctl(). */
}
```

10.3.1 Shared Memory Segements anfordern

Mit der Funktion `shmget(2)` erzeugen wir ein neues Shared Memory Segment oder fordern ein bereits existierendes an.

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

Rückgabewerte: Shared Memory Segment ID oder -1 bei Fehler.

key

Repräsentiert einen eindeutigen Schlüssel des Shared Memory Segments.

size

Bestimmt die Größe des Shared Memory Segments. Der Wert wird zu einem Vielfachen von `sysconf(PAGE_SIZE)` aufgerundet.

shmflg

Zeigt an, wie das neue Segment erzeugt, bzw. das existierende Segment angefordert werden soll.

Wenn der Schlüssel durch die Konstante `IPC_PRIVATE` repräsentiert wird, kann das Segment nur von Parent und Child gemeinsam genutzt werden. Die Flags bestimmen, wie der Aufruf von `shmget(2)` ausgeführt werden soll. Das Argument kann mit folgenden Konstanten kombiniert werden:

IPC_CREAT

Ist dieses Flag gesetzt, wird ein neues Segment erzeugt. Andernfalls lokalisiert die Funktion das Segment, dessen Schlüssel dem Argument `key` entspricht.

IPC_EXCL

Wird in Verbindung mit `IPC_CREAT` verwendet und stellt sicher, daß der Aufruf fehlschlägt, wenn das Shared Memory Segment bereits existiert.

mode_flags

Geben die Zugriffsrechte des Shared Memory Segments an.

Das folgende Codesegment illustriert die Anwendung von `shmget(2)`:

```
#include <sys/types.h> /* for key_t */
#include <sys/shm.h>

/* key to be passed to shmget() */
key_t key = ftok("/home/steve/dummy.key", 255);

/* shmflg to be passed to shmget() */
int shmflg = 0666 | IPC_CREAT;

/* size to be passed to shmget() */
int size = 4096;

/* return value from shmget() */
int shmid;

if ((shmid = shmget(key, size, shmflg)) == -1)
    err_fatal("shmget failed");
else
    fprintf(stderr, "shmget return ID: %d\n", shmid);
```

Beim Aufruf von `fork(2)` wird das Segment an den Kindsprozessen vererbt. Aufrufe von `exec(3)` und `exit(3)` entfernen das Shared Memory Segment automatisch aus dem Adressraum des Prozesses.

10.3.2 Shared Memory Segments und der Prozessadressraum

Nachdem ein Shared Memory Segment erzeugt worden ist, kann es mit Hilfe der Funktion `shmat(2)` in den Adressraum des Prozesses aufgenommen (engl. *to attach*) werden. Bei jedem erfolgreichen Aufruf von `shmat(2)` wird der Zähler `shm_nattch` der `shmid_ds`-Struktur erhöht und das Feld `shm_atime` aktualisiert.

Die `shmat(2)`-Funktion weist folgende Synopsis auf:

```
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Rückgabewert: Startadresse des Shared Memory Segments oder -1 bei Fehler.

shmid

Identifiziert das Segment, daß dem aufrufenden Prozess zugeordnet werden soll.

shmaddr

Startadresse des Prozessadressraums an der das Segment angefügt werden soll. Ist es NULL, so wird die erste verfügbare Adresse verwendet.

shmflg

Bestimmen, wie das Segment in den Adressraum des Prozesses aufgenommen werden soll.

Das **shmflg**-Argument wird durch eine der folgenden Konstanten charakterisiert:

SHM_RND

Das Segment soll an die von **shmaddr** spezifizierte Adresse geheftet werden. Dabei ist zu beachten, daß die Adresse auf ein Vielfaches von **SHMLBA** abgerundet wird (also **shmaddr** - ((**uintptr_t**) **shmaddr** % **SHMLBA**)).

SHM_RDONLY

Bindet das Shared Memory Segment im Nur-Lesen-Modus ein. Der Prozess muß dazu Leserechte aufweisen. Ist das Flag nicht gesetzt, wird das Segment zum Lesen und Schreiben eingebunden und der Prozess muß entsprechende Rechte aufweisen.

Da es sich bei dem Rückgabewert um einen (**void ***)-Zeiger handelt, kann er wie ein Zeiger verwendet werden, der von Funktionen der **malloc(3)**-Familie zurückgegeben wird. Einige Systeme fordern eine spezielle Ausrichtung (*alignment*) des Segments, daß durch **shmflg** spezifiziert werden kann. Um maximale Portabilität zu gewährleisten, sollte das Argument **shmaddr** stets NULL sein, damit das System eine geeignete Adresse für die Einbindung des Segments auswählt.

Selbstverständlich kann ein eingebundenes Segment wieder aus dem Adressraum eines Prozesses entfernt (engl. *to detach*) werden. Dazu verwenden wir die Funktion **shmdt(2)**.

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Rückgabewert 0 bei Erfolg oder -1 bei Fehler.

shmaddr

Startadresse des eingebundenen Shared Memory Segments, daß von dem Adressraum des Prozesses gelöst werden soll.

Die durch das Argument **shmaddr** spezifizierte Adresse muß mit dem von **shmat(2)** zurückgelieferten Wert übereinstimmen. Andernfalls schlägt der Funktionsaufruf fehl.

Listing 10.11: Server-Anwendung, die ein Shared Memory Segment erstellt

Zuerst wird ein Schlüssel mit der Funktion **ftok(3)** erzeugt und anschließend **shmget(2)** aufgerufen, um ein Shared Memory Segment zu registrieren. Damit wir es mit Inhalten füllen können, müssen wir es dem Adressraum unseres Prozesses hinzufügen. Jetzt kann es gefüllt und vom Client ausgelesen werden.

```
1 #include <sys/types.h>
2 #include <sys/shm.h>
3 #include <stdio.h>
4 #include "header.h"
5
```

```

6  #define SHM_SIZE 128
7  #define SHM_PERM (IPC_CREAT | 0666)
8
9  int main(void) {
10    int segment_id;
11    key_t key;
12    char *segment, *str, c;
13
14    /* obtain a unique key */
15    key = ftok("/home/steve/dummy.key", 255);
16
17    if ((segment_id = shmget(key, SHM_SIZE, SHM_PERM)) < 0)
18      err_fatal("segmentget failed in client");
19
20    /* attach the segment to process address space */
21    if ((segment = shmat(segment_id, NULL, 0)) == (char *)-1)
22      err_fatal("segmentat failed");
23
24    /* fill in the shared memory segment */
25    str = segment;
26
27    for (c = 'A'; c <= 'z'; c++)
28      *str++ = c;
29    *str = (char)NULL;
30
31    while (*segment != '*')
32      sleep(1);
33
34    return (0);
35 }

```

Listing 10.11: xcode/shmserver.c - Server-Code. Erstellt ein Shared Memory Segment und füllt es mit Inhalt auf.

Listing 10.12: Client-Anwendung, die ein Shared Memory Segment ausliest

Obwohl der Server-Prozess beendet wurde, bleibt das Shared Memory Segment erhalten, bis das System neugestartet wird. Auf diese Weise kann der Client gestartet werden und den Inhalt des Segments auslesen.

```

1  #include <sys/types.h>
2  #include <sys/shm.h>
3  #include <stdio.h>
4  #include "header.h"
5
6  #define SHM_SIZE 128
7  #define SHM_PERM (IPC_CREAT | 0666)
8
9  int main(void) {
10    int segment_id;
11    key_t key;
12    char *segment, *str;
13
14    /* get key for segment created by server */
15    key = ftok("/home/steve/dummy.key", 255);
16
17    /* acquire the shared memory segment */
18    if ((segment_id = shmget(key, SHM_SIZE, SHM_PERM)) < 0)
19      err_fatal("segmentget failed in server");
20
21    /* attach the segment to process address space */
22    if ((segment = shmat(segment_id, NULL, 0)) == (char *)-1)

```

```

23         err_fatal("segmentat failed");
24
25     /* read content of shared memory segment */
26     for (str = segment; *str != (char)NULL; str++)
27         printf("%c", *str);
28
29     putchar('\n');
30
31     /* finally , detach shared memory segment */
32     if (shmdt(segment) == -1)
33         err_fatal("shmdt failed in client");
34     else
35         if (shmctl(segment_id, IPC_RMID, NULL) < 0)
36             err_fatal("shmctl failed in client");
37
38     return (0);
39 }
```

Listing 10.12: xcode/shmclient.c - Client-Code. Liest ein Shared Memory Segment aus.

Eine beispielhafte Ausführung beider Programme könnte wie folgt aussehen:

```

% ./shmserver&
[1] 4413
% ipcs -m
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
0xffffffff 0        steve      666        128        1
% ./shmclient
ABCDEFGHJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
% ipcs -m
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
0xffffffff 0        steve      666        128        1          dest
% kill -SIGTERM 4413
% ipcs -m
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
```

Der Server wird als Hintergrundprozess gestartet und erzeugt ein Shared Memory Segment, das sogleich in den Adressraum des Prozesses eingebunden wird. Ob die Operation erfolgreich war, können wir mit dem Kommando `ipcs(8)` überprüfen. Wir sehen, wurde das Segment mit einer Größe von 128 Bytes genau 1 mal referenziert (nattach). Der Client liest den Inhalt des Segments aus und beendet sich selbst, allerdings nicht ohne das Segment vorher vom Prozessadressraum zu lösen und anschließend über die Funktion `shmctl(2)` (die in Kürze besprochen wird) zu zerstören (erkennbar an dem Wert `dest` in der Spalte `status`). Da der Server noch immer im Hintergrund läuft, wird das Segment entgültig entfernt sobald der Serverprozess endet. □

Um dem Systemadministrator ihres Systems (auf dem Sie evtl. keine Superuser-Rechte besitzen) auch weiterhin gewogen zu sein, sollten sie stets dafür sorgen, ihre IPC-Resourcen vollständig aufzuräumen, denn sie sind auf den meisten Systemen beschränkt. Wenn Sie es versäumen, ein Shared Memory Segment zu zerstören, ist ein Neustart des Systems erforderlich, um alle verwaisten Segmente zu entfernen. Wenn mehrere Benutzer auf diesem System arbeiten, werden sie sich dafür auf ihre ganz eigene Weise bedanken.

10.3.3 Shared Memory Segements verwalten

Zur Verwaltung der globalen IPC-Resourcen steht uns der Befehl `ipcs(8)` zur Verfügung. Shared Memory Segments verwalten wir programmatisch mit der Funktion `shmctl(2)`.

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
Rückgabewert: 0 bei Erfolg und -1 bei Fehler.
```

shmid

ID des Shared Memory Segments, das wir verwalten wollen.

cmd

Gibt an, was wir mit dem durch **shmid** spezifizierte Shared Memory Segment anstellen möchten.

buf

Zeiger auf eine **shmid_ds**-Struktur, das die zurückgelieferten Informationen aufnimmt.

Mögliche Kommandos für den Parameter **cmd** sind:

IPC_STAT

Kopiert die Kernelinformationen aus der mit **shmid** verknüpften Datenstruktur in die **shmid_ds**-Struktur auf die **buf** zeigt. Der Aufrufer muß Leserechte für dieses Segment aufweisen.

IPC_SET

Schreibt Werte einiger Member der **shmid_ds**-Struktur auf die **buf** zeigt. Dadurch wird der **shm_ctime**-Eintrag in **shmid_ds** aktualisiert. Das Kommando erfordert, daß die effektive User ID des aufrufenden Prozesses mit dem Besitzer des Shared Memory Segments übereinstimmen. Ausgenommen ist nur der Superuser.

IPC_RMID

Markiert das Segment als veraltet und erlaubt die Freigabe. Der Kernel entfernt das Segment erst, wenn kein Prozess das Segment referenziert (**shmid_ds.shm_nattch** = 0). Nur der Besitzer des Segments ist autorisiert, dieses Kommando auszuführen.

Beispiel 10.13: Shared Memory Segments mit **shmctl(2)** administrieren

```

1 #include <sys/types.h>
2 #include <sys/shm.h>
3 #include <stdio.h>
4 #include "errors.h"
5
6 #define SHM_SIZE 128
7 #define SHM_PERM (IPC_CREAT | 0666)
8
9 int main(void) {
10     int segment_id;
11     key_t key;
12     char *segment, *str, c;
13     struct shmid_ds buffer;
14
15     /* obtain a unique key */
16     key = ftok("/home/steve/dummy.key", 255);
17
18     if ((segment_id = shmget(key, SHM_SIZE, SHM_PERM)) < 0)
19         err_fatal("segmentget failed in client");
20
21     /* attach the segment to process address space */
22     if ((segment = shmat(segment_id, NULL, 0)) == (char *)-1)
23         err_fatal("segmentat failed");
24
25     /* print some infos about the segment */

```

```

26     if (shmctl(segment_id, IPC_STAT, &buffer) < 0)
27         err_fatal("shmctl failed");
28
29     printf("    Segment ID: %d\n segment size: %d\n# of attaches: %d\n",
30           segment_id, buffer.shm_segsz, buffer.shm_nattch);
31
32     /* first , detach segment */
33     if (shmdt(segment) < 0)
34         err_fatal("shmdt failed");
35
36     /* destroy IPC resource */
37     if (shmctl(segment_id, IPC_RMID, NULL) < 0)
38         err_fatal("shmctl failed for IPC_RMID command");
39
40     return (0);
41 }
```

Listing 10.13: xcode/shmctl.c - Anwendung von shmctl(2).



Das Kommando IPC_SET setzt die Informationen, welche in **buffer** enthalten sind. Allerdings dürfen nur die Felder **uid**, **gid** und **mode** der **shm_perms**-Struktur geändert werden.

10.4 Message Queues

Die IPC-Techniken Shared Memory, Semaphores, Mutexes und Message Queues werden im Allgemeinen unter der Bezeichnung *System V IPC* zusammengefaßt, da sie zum ersten Mal im AT&T-UNIX System V aufgetaucht sind. In diesem Abschnitt beschäftigen wir uns mit Message Queues.

POSIX bringt im Rahmen der Echtzeiterweiterung auch Unterstützung für Message Queues mit. Da sie auf fast allen UNIX-Systemen verfügbar sind, befassen wir uns aber zuvor mit System V Message Queues und kommen dann zur POSIX-Variante (Abschnitt 10.4.2 *POSIX Message Queues*).

Ein Vergleich beider Techniken zeigt, wie verschieden die Ansätze sind. Zwar können beide zur Kommunikation zwischen Threads und Prozessen eingesetzt werden, doch gibt es erhebliche Unterschiede in der Syntax und Implementierung:

- Die POSIX-Funktionen sind nicht Teil des Kernels, sondern der libc-Bibliothek und arbeiten somit im User Space. Dadurch ergibt sich für System V Message Queues natürlich der Nachteil, daß jeder Funktionsaufruf einen Kontextwechsel (*context switch*) erfordert, was den Overhead gegenüber POSIX Message Queues erhöht.
- Beide Implementierungen erhalten die Queues nach einem Neustart des Systems, doch ist nicht sichergestellt, daß die erhaltenen Nachrichten gespeichert werden. Grundsätzlich ist es nicht empfehlenswert, sich darauf zu verlassen. Wir sollten also stets prüfen, ob eine Queue noch vorhanden ist und bei Bedarf neu erstellen.
- Anfragen an Message Queues können entweder blockierend (*blocking*) oder nicht-blockierend (*non-blocking*) durchgeführt werden. Bei POSIX Message Queues ist das eine Eigenschaft des Message Descriptors, während diese Einstellung bei System V Message Queues mit jedem Funktionsaufruf angegeben werden muß.
- Die Nachrichtengröße von POSIX Message Queues kann individuell angepaßt werden, während die Nachrichtengröße von System V Message Queues immer gleich ist.
- Mit POSIX Message Queues können wir über den Eingang neuer Nachrichten informiert werden, was mit System V Message Queues nicht möglich ist.

- POSIX Message Queues unterstützen Nachrichtenprioritäten, so daß stets die älteste Nachricht mit der höchsten Priorität abgeholt wird. System V Messages kann zwar auch eine Klasse zugeordnet werden, allerdings ist nicht sichergestellt, daß wir immer die älteste Nachricht dieser Klasse erhalten. Letztendlich handelt es sich um Warteschlangen innerhalb von Warteschlangen.

Das Einsatzgebiet von Message Queues beider Implementierungen ist vielfältig. Die einfachste Möglichkeit ist sicherlich die Anwendung in einem Client/Server-Model, in der ein Client (Produzent) Nachrichten an einen Server (Konsument) sendet.

Mit Message Queues läßt sich die Arbeit auf Prozesse oder Threads verteilen, in dem Prozesse, die eine bestimmte Aufgabe erfüllen, benachrichtigt werden, wenn es etwas zu tun gibt. Sie „lauschen“ quasi an der Message Queue und bei Erhalt der Nachricht erledigen sie ihre Aufgaben.

10.4.1 System V Message Queues

Eine Nachrichtenwarteschlange (*message queue*) arbeitet ähnlich wie eine benannte Pipe, verfügt aber über ein paar sehr nützliche Merkmale, wie beispielsweise die Bearbeitung der Nachrichten außerhalb der vorgeschriebenen Reihenfolge, was mit FIFOs nicht möglich ist. Das bedeutet also, daß Nachrichten in der Warteschlange in der Reihenfolge platziert werden, in der sie eintreffen.

Prozesse sind in der Lage neue Message Queues zu erzeugen, oder sich mit einer existierenden zu verbinden. Auf diese Weise können Prozesse miteinander kommunizieren und Nachrichten austauschen.

Während anonyme Pipes nur so lange existieren wie der erzeugende Prozess, muß eine Message Queue explizit zerstört werden. Es ist ratsam, die `ipcs(8)`-Kommandos zu verwenden, um herauszufinden, ob es noch vorhandene Message Queues gibt. Nicht mehr benötigte Warteschlangen entfernen wir mit `ipcrm(8)`.

10.4.1.1 Von Bezeichnern und Schlüsseln

Jedesmal, wenn wir eine Warteschlange erzeugen, erstellt der Kernel eine so genannte IPC-Struktur, der eine eindeutige ID (*Bezeichner*) zugeordnet ist, mit der sie im gesamten System identifiziert werden kann. Sie wird benötigt, um die Warteschlange zu verwalten. Die ID können wir mit dem `msgget(2)` Systemaufruf abfragen. Im Gegensatz dazu gibt es Schlüssel (*keys*), die eine Warteschlange identifiziert. Wir können uns nur mit einer bestimmten Warteschlange verbinden, wenn wir den Schlüssel kennen. Im weiteren Verlauf dieses Abschnittes gehen wir noch genauer auf die Beziehung zwischen Bezeichner und Schlüssel ein.

Die ID ist ein Integer, verwaltet durch den Kernel. Wird eine neue IPC-Struktur erzeugt, erhöht sich auch der Zähler. Der Wert bleibt auch dann erhalten, wenn eine Struktur entfernt wird. Der Schlüssel ist vom Typ `key_t`, definiert in `<sys/types.h>`, aber nicht Teil der IPC-Struktur.

Die Unterscheidung zwischen ID und Schlüssel ist notwendig, da Client- und Serverprozesse nicht notwendigerweise miteinander verwandt sind und wir die ID nicht an andere Prozesse senden können, ohne eine andere IPC-Technik zu verwenden. Der Schlüssel hingegen kann im Dateisystem oder in einem Header abgelegt werden so daß er anderen Prozessen zur Verfügung steht.

Wir müssen uns über folgende Zusammenhänge im Klaren sein, wenn wir Message Queues reibungslos einsetzen wollen:

- Der Server kann eine neue IPC-Struktur erzeugen, indem er als Schlüssel die Konstante `IPC_PRIVATE` angibt und die zurückgelieferte ID der Struktur abspeichert, so daß der Client später darauf zugreifen kann. Dazu eignet sich das Dateisystem am Besten. Nachteil ist, daß wir auf das Dateisystem zurückgreifen müssen, um den Key abspeichern oder abfragen zu können. Der Vorteil von `IPC_PRIVATE` ist, daß die ID an den Child-Prozess durch `fork(2)` vererbt wird.
- Client und Server können sich auf einen Schlüssel einigen, so daß der Server eine Struktur mit diesem Schlüssel erzeugt und der Client sich mit der Queue durch diesen Schlüssel verbinden kann. Nachteil ist, daß der Schlüssel bereits einer existierenden IPC-Struktur zugeordnet sein kann und

somit ein Fehler bei der Erstellung der Struktur zurückgegeben wird. Diese Fehler müssen von Server und Client behandelt werden. Der Server muß den Schlüssel löschen, einen neuen erzeugen und ihn dem Client mitteilen, was nicht besonders elegant ist.

- Eine recht praktikable Lösung besteht in der Verwendung der Funktion `ftok(3)`, die einen mit hoher Wahrscheinlichkeit noch nicht existierenden Schlüssel aus einer Pfadangabe und beispielsweise einer Projekt-ID erzeugt. Nun gibt es zwei Möglichkeiten: entweder legen wir den Schlüssel in einem Header ab, der vom Client importiert wird, oder er wird in einer Datei platziert, die von Server und Client ausgelesen wird.

In weiteren Verlauf greifen wir die letzte Variante auf, da sie viele Vorteile bietet.

10.4.1.2 Die IPC-Struktur

Die vielfach erwähnte mysteriöse IPC-Struktur enthält eine besondere Struktur zur Definition von Zugriffsrechten: `ipc_perm`.

```
struct ipc_perm {
    uid_t uid      /* Owner's user ID. */
    gid_t gid      /* Owner's group ID. */
    uid_t cuid     /* Creator's user ID. */
    gid_t cgid     /* Creator's group ID. */
    mode_t mode    /* Read/write permission. */
    ulong seq      /* Sequence number */
    key_t key      /* Key to the IPC structure */
}
```

Fast alle Felder werden bei der Erzeugung der IPC-Struktur automatisch gesetzt. Die ersten vier Felder können wir hinterher mit Hilfe der Funktionen `msgctl(2)`, `semctl(2)` oder `shmctl(2)` verändert werden.

10.4.1.3 Message Queues

Eine Nachrichtenwarteschlange besteht aus einer verketteten Liste von Nachrichten, die über eine `msqid_ds`-Struktur verwaltet werden:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* permission structure */
    struct msg *msg_first; /* pointer to 1st message on queue */
    struct msg *msg_last; /* pointer to last message on queue */
    ulong msg_cbytes; /* number of bytes on queue */
    ulong msg_qnum; /* number of messages on queue */
    ulong msg_qbytes; /* maximal number of bytes on queue */
    pid_t msg_lspid; /* PID of last msgsnd() */
    pid_t msg_lrpid; /* PID of last msgrecv() */
    time_t msg_stime; /* time of last msgsnd() */
    time_t msg_rtime; /* time of last msgrecv() */
    time_t msg_ctime; /* time of last queue change */
}
```

Eine neue Warteschlange wird mit der Funktion `msgget(2)` erzeugt. Neue Nachrichten platzieren wir in der Queue mit `msgsnd(2)`. Sie werden stets an das Ende der Queue gestellt. Vorhandene Nachrichten holen wir von der Queue mit `msgrecv(2)` ab. Jeder Nachricht ist ein Typ, eine Länge und die Anzahl der Bytes zu geordnet.

10.4.1.4 Arbeiten mit Message Queues

Bevor wir mit Nachrichtenwarteschlangen arbeiten können, müssen wir zuerst eine Queue erzeugen. Das erledigen wir mit der Funktion `msgget(2)`.

```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

Rückgabewert: ID der Message Queue oder -1 bei Fehler.

key

Schlüssel, den wir für das Verbinden mit einer Nachrichtenwarteschlange benötigen.

msgflag

Zugriffsrechte für die Message Queue.

Die möglichen Werte für den Parameter `msgflg` sind in Tabelle 10.2 für alle IPC-Varianten zusammengefaßt.

Zugriffsrechte	Message Queues	Semaphores	Shared Memory
Benutzer darf lesen	MSG_R	SEM_R	SHM_R
Benutzer darf ändern	MSG_W	SEM_W	SHM_W
Gruppe darf lesen	MSG_R > 3	SEM_R > 3	SHM_R > 3
Gruppe darf ändern	MSG_W > 3	SEM_W > 3	SHM_W > 3
Andere dürfen lesen	MSG_R > 6	SEM_R > 6	SHM_R > 6
Andere dürfen ändern	MSG_W > 6	SEM_W > 6	SHM_W > 6

Tabelle 10.2: Zugriffsrechte für SysV IPC.

Abhängig von den Argumenten `key` und `msgflg` wird eine neue Queue erzeugt oder eine existierende Queue referenziert. Die Regeln lauten:

- Ist der Wert von `key` auf `IPC_PRIVATE` gesetzt oder `key` momentan nicht einer Queue diesen Typs zugeordnet, aber `IPC_CREAT` in `msgflg` gesetzt, so wird eine neue Queue erzeugt.
- Stimmt der Wert in `key` mit dem einer existierenden Queue überein, so wird diese Queue referenziert. `key` muß mit dem beim Aufruf mit `IPC_CREAT` angegebenen Schlüssel übereinstimmen.
- Es ist nicht möglich `IPC_PRIVATE` für `key` und `IPC_CREAT` in `msgflg` anzugeben, denn `IPC_PRIVATE` erzeugt immer eine neue Queue. Eine solche Queue kann nur über den Bezeichner referenziert werden, der von `msgget(2)` zurückgeliefert wird.

Der Aufruf vom `msgget(2)` sorgt für die Initialisierung der `ipc_perm`-Struktur. Das Feld `mode` wird auf die Werte von `msgflg` gesetzt. Des Weiteren wird die Struktur `msqid_ds` initialisiert, wobei die Member `msg_cbytes`, `msg_qnum`, `msg_lspid`, `msg_stime` und `msg_rtime` auf 0 gesetzt werden. `msg_ctime` wird auf die aktuelle Systemzeit und `msg_qbytes` auf das Limit des Systems gesetzt.

Message Queue ID und Keys

Oft ist die Verwirrung groß, wenn wir von Keys und Message Queue IDs sprechen. Wozu benötige ich denn einen Schlüssel und warum gibt es auch noch eine ID? Der Schlüssel ist die Eingangstür für Benutzerprozesse. Er hat im Kernel Mode keine Bedeutung. Ganz anders die Message Queue ID: sie ist nur dem Kernel bekannt und hat im User Mode keine Bedeutung. Anwendungsprogramme zeigen mit dem Key an, daß sie eine Queue anfordern wollen, deren ID mit Key assoziiert ist. Aus diesem Grund müssen wir vor der Arbeit mit einer Message Queue den Schlüssel erzeugen und können hinterher nur noch über die Message Queue ID mit der Warteschlange interagieren.

Der Parameter `key` muß bei der Erzeugung einer neuen Queue eindeutig sein. Würden wir den Schlüssel manuell setzen, ist nicht sichergestellt, daß er eindeutig ist. Aus diesem Grund kombinieren wir den Aufruf mit `ftok(3)`, welcher einen Schlüssel erzeugt. Das könnte etwa so aussehen:

```
key = ftok("/home/steve/keyfile", 'z');
queue_id = msgget(key, MSG_R | MSG_W | MSG_R >> 3 | MSG_R >> 6 | IPC_CREAT);
```

Um auf eine existierende Message Queue Einfluß zu nehmen, verwenden wir die Funktion `msgctl(2)`:

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
Rückgabewert: 0 bei Erfolg und -1 bei Fehler.
```

`msqid`

ID der Nachrichtenwarteschlange, die wir konfigurieren möchten.

`cmd`

Operation, die auf die mit `msqid` referenzierte Queue angewendet werden soll.

`buf`

Zeiger auf eine `msqid_ds`-Struktur, in der die Informationen abgelegt werden oder bereits platziert wurden. Abhängig von `cmd`.

Die Funktionalität von `msgctl(2)` entspricht der von `ioctl(2)`. Das Argument `cmd` bestimmt hier auch die Arbeitsweise:

`IPC_STAT`

Ruft die `msqid_ds`-Struktur der Queue ab, die durch `msqid` referenziert wird. Sie wird in der Struktur abgespeichert, auf die `buf` verweist.

`IPC_SET`

Setzt vier Felder der `msqid_ds`-Struktur referenziert durch `buf` der betreffenden Queue. Dieses Kommando kann nur von Prozessen ausgeführt werden, dessen effektive Benutzer-ID der von `msg_perm.cuid` oder `msg_perm.uid` entspricht. Ausgenommen sind natürlich Prozesse mit Superuser-Privilegien. Folgende Felder der `msqid_ds`-Struktur werden auf die der Queue übertragen: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` und `msg_perm.qbytes` (nur Root).

`IPC_RMID`

Entfernt die spezifizierte Nachrichtenwarteschlange auf dem System erhält die Daten aber. Dadurch sind andere Prozesse in der Lage, die Daten weiterhin zu nutzen. Versucht ein Prozess zum Zeitpunkt der Löschung der Queue auf sie zu, wird EIDRM als Fehler geliefert. Auch hier muß die effektive Benutzer-ID des Prozesses der von `msg_perm.cuid` oder `msg_perm.uid` entsprechen.

Nach all den Vorbereitungen platzieren wir Nachrichten auf der Message Queue mit `msgsnd(2)`:

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
Rückgabewert: 0 bei Erfolg und -1 bei Fehler.
```

`msqid`

Message Queue in der wir die Nachricht, die durch `msgp` referenziert wird, platzieren möchten.

`msgp`

Zeiger auf eine Struktur, in der sich die Nachricht befindet.

msgsz

Größe der zu platzierenden Nachricht, die durch `msgp` referenziert wird.

msgflg

Kann entweder mit `IPC_WAIT` oder `IPC_NOWAIT` gesetzt werden.

`IPC_NOWAIT` sorgt dafür, daß die Nachricht sofort platziert wird, auch wenn beispielsweise die maximale Anzahl von Nachrichten erreicht wurde oder die Queue voll ist. In diesem Fall wird `EAGAIN` als Fehlercode geliefert. `IPC_WAIT` wartet gegebenenfalls bis die Bedingungen zum Platzieren der Nachricht erfüllt sind und der Aufrufer blockiert.

Die Nachrichten werden immer am Ende der Queue platziert. Sie sind in einer Struktur untergebracht, die als erstes Feld den Typ als `long` angibt und als zweites den Nachrichteninhalt:

```
struct mymessage {
    long mtype;      /* message type */
    char mtext[1];   /* message text */
}
```

Das Feld `mtype` ist nicht negativ und kann zur Identifikation von Nachrichten beim Empfang mit `msgrcv(2)` verwendet werden. `mtext` kann eine Zeichenkette beliebiger Länge sein.

Die Strukturdefinition ist etwas irreführend, denn eigentlich können wir jede beliebige Struktur in der Nachrichtenwarteschlange platzieren, solange das erste Feld vom Typ `long` ist. Der folgende Aufruf von `msgsnd(2)` ist daher völlig legal:

```
key_t key;
int queue_id;

/* some arbitrary message */
struct some_message {
    long type;
    char *title;
    int id;
};

/* fill in some data */
struct some_message buf = {0, "Jean's Cookbook", 123};

/* obtain a unique key for our message queue */
key = ftok("/home/steve/keyfile", 'z');

/* create message queue and store queue id */
queue_id = msgget(key, MSG_R | MSG_W | MSG_R >> 3 | MSG_R >> 6 | IPC_CREAT);

/* place the message on the queue */
if (msgsnd(queue_id, &buf, sizeof(buf), IPC_NOWAIT) < 0) {
    err_fatal("msgsnd failed");
}
```

Bevor wir uns ein vollständiges Programm zur Demonstration anschauen, wollen wir noch klären, wie wir Nachrichten von der Queue abfragen. Dazu steht uns die Funktion `msgrcv(2)` zur Verfügung:

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
Rückgabewert: Anzahl der Bytes, die in der Queue platziert wurden oder -1 bei Fehler.
```

msqid

Nachrichtenwarteschlange von der wir Nachrichten abrufen möchten.

msqp

Zeiger auf eine Nachrichtenstruktur zur Aufnahme der Daten.

msgsz

Größe der Nachrichtenstruktur in der wir die Nachricht speichern wollen.

msgtyp

Zeigt an, welche Nachricht wir abrufen möchten (siehe weiter unten).

msgflg

Wenn das Flag auf **MSG_NOERROR** gesetzt ist erhalten wir keine Mitteilung darüber, ob die Nachricht größer als **msgsz** ist. Sie wird einfach abgeschnitten. Andernfalls wird **E2BIG** zurückgegeben und die Nachricht in der Warteschlange gelassen.

Das **msgtyp**-Argument bestimmt, welche Nachricht wir von der Queue nehmen möchten. Ist der Wert 0 so wird die erste Nachricht abgeholt. Beträgt er größer 0 wird die erste Nachricht, dessen **type**-Feld den gleichen Wert wie **msgtyp** aufweist, abgeholt. Andernfalls rufen wir die erste Nachricht ab, die den kleinsten absoluten Wert von **msgtyp** aufweist.

Um Nachrichten nach dem FIFO-Prinzip (*first-in first-out*) abzuholen, verwenden wir einen **msgtyp**-Wert ungleich 0. Das macht beispielsweise Sinn, wenn Nachrichten mit Prioritäten ausgestattet sind.

Eine Angabe von **IPC_NOWAIT** bzw. **IPC_WAIT** hat die gleiche Auswirkung wie bei **msgsnd(2)**. Ist beispielsweise keine Nachricht in der aktuellen Queue so wird bei **IPC_NOWAIT** der Fehlercode **ENOMSG** zurückgeliefert, während der aufrufende Prozess bei **IPC_WAIT** auf das Eintreffen neuer Nachrichten des angegebenen Typs wartet. Des Weiteren könnte auch die Fehler **EIDRM** und **EINTR** auftreten, wenn die Queue von dem System entfernt oder der aufrufende Prozess durch ein Signal unterbrochen wird.

Das Empfangen von Nachrichten funktioniert fast genauso wie das Senden:

```
key_t key;
int queue_id;

/* some arbitrary message */
struct some_message buf;

/* obtain a unique key for our message queue */
key = ftok("/home/steve/keyfile", 'z');

/* create message queue and store queue id */
queue_id = msgget(key, MSG_R | MSG_W | MSG_R >> 3 | MSG_R >> 6);

/* fetch message with type of 0 from queue */
msgrcv(queue_id, &buf, sizeof(buf), 0, 0); /* no flags here */
```

Kommen wir nun zu einem Beispiel

Listing 10.14: Server-Code

Der Server erzeugt eine neue Message Queue und läuft in eine Hauptschleife, in der die Nachrichten des Clients abgerufen, in Großbuchstaben umgewandelt und auf gleichem Wege wieder in der Queue platziert werden. Der Client holt sie nach einer kurzen Wartezeit wieder ab.

```
1 #include "mq1.h"
2
3 void signal_handler(int sig_num);
4
5 int quit = 0;
6
7 int main(int argc, char **argv) {
8     int i, queue_id;
```

```

9      size_t len;
10     char *ptr;
11     struct msgbuf msg_buf;
12
13     signal(SIGTERM, signal_handler);
14
15     /* create a queue for messages */
16     if ((queue_id = msgget(KEY, 0666 | IPC_CREAT)) == -1)
17         err_fatal("msgget failed in server");
18
19     printf("server up and running\n");
20
21     while (!quit) {
22         /* fetch messages sent by clients, ignore others */
23         if ((len = msgrcv(queue_id, &msg_buf,
24                            (size_t)sizeof(msg_buf), -1L, 0)) == (size_t)-1)
25             err_normal("msgrcv failed in server");
26         break;
27     }
28
29     printf("Input from client: %s\n", msg_buf.text);
30
31     /* change message text to UPPER CASE */
32     for (ptr = msg_buf.text; *ptr != '\0'; ptr++)
33         *ptr = toupper(*ptr);
34
35     msg_buf.type = msg_buf.id; /* set type to client PID */
36     //sleep(1); /* sleep, then return to sender */
37
38     if (msgsnd(queue_id, &msg_buf, len, 0) == -1) {
39         if (errno != EINTR)
40             err_fatal("msgsnd failed in server");
41         break;
42     }
43 }
44
45     fprintf(stderr, "server stopped\n");
46
47     /* cleanup */
48     if (msgctl(queue_id, IPC_RMID, NULL) == -1)
49         err_fatal("msgctl failed for IPC_RMID in server");
50
51     return (0);
52 }
53
54 void signal_handler(int sig_num) {
55     quit = 1;
56     return;
57 }
```

Listing 10.14: xcode/mq1srv.c - Server-Code.

Listing 10.15: Client-Code

Der Client platziert die als Argument übergebene Nachricht in der Message Queue. Die Nachrichtenstruktur enthält ein `type`-Feld und ein `ID`-Feld. ersteres ist für den Server interessant. Letzteres transportiert die PID des Clients, der die Nachricht gesendet hat. So können wir mit dem Server mehrere Clients bedienen.

```

1 #include "mq1.h"
2
3 void signal_handler(int sig_num);
4
5 int quit = 0;
6
7 int main(int argc, char **argv) {
8     int i, queue_id;
9     size_t len;
10    char *ptr;
11    struct msgbuf msg_buf;
12
13    signal(SIGTERM, signal_handler);
14
15    /* create a queue for messages */
16    if ((queue_id = msgget(KEY, 0666 | IPC_CREAT)) == -1)
17        err_fatal("msgget failed in server");
18
19    printf("server up and running\n");
20
21    while (!quit) {
22        /* fetch messages sent by clients, ignore others */
23        if ((len = msgrcv(queue_id, &msg_buf,
24                           (size_t)sizeof(msg_buf), -1L, 0)) == (size_t)-1)
25        {
26            err_normal("msgrcv failed in server");
27            break;
28        }
29
30        printf("Input from client: %s\n", msg_buf.text);
31
32        /* change message text to UPPER CASE */
33        for (ptr = msg_buf.text; *ptr != '\0'; ptr++)
34            *ptr = toupper(*ptr);
35
36        msg_buf.type = msg_buf.id; /* set type to client PID */
37        //sleep(1); /* sleep, then return to sender */
38
39        if (msgsnd(queue_id, &msg_buf, len, 0) == -1) {
40            if (errno != EINTR)
41                err_fatal("msgsnd failed in server");
42            break;
43        }
44
45        fprintf(stderr, "server stopped\n");
46
47        /* cleanup */
48        if (msgctl(queue_id, IPC_RMID, NULL) == -1)
49            err_fatal("msgctl failed for IPC_RMID in server");
50
51        return (0);
52    }
53
54 void signal_handler(int sig_num) {
55     quit = 1;
56     return;
57 }
```

Listing 10.15: `xcode/mq1srv.c` - Client-Code.

Listing 10.16: Header-Code

Der Header enthält die Nachrichtenstruktur und den Schlüssel der Nachrichtenwarteschlange.

```

1 #include <ctype.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <signal.h>
5 #include <sys/msg.h>
6 #include <unistd.h>
7 #include "header.h"

8
9 extern int errno;

10 struct msgbuf {
11     long type;
12     long id;
13     char text[256];
14 };
15
16
17 #define KEY    1029

```

Listing 10.16: xcode/mq1.h - Header für Server und Client.

Ein Testlauf könnte etwa so aussehen:

```
% mq1srv &
Server up and running...
% mq1cli "Testnachricht"
Input from client: Testnachricht
Output for client: TESTNACHRICHT
client ended
```



10.4.2 POSIX Message Queues

POSIX.4 definiert Erweiterungen, denen ein Echtzeitsystem entsprechen muß. In diese Kategorie fallen viele verschiedene Techniken, die alle zusammen ein System für die Echtzeitverarbeitung qualifizieren:

- Process Memory Locking
- Memory Mapped Files und Shared Memory
- Priority Scheduling
- Signalverarbeitung in Echtzeit
- Zeitgeberunterstützung
- Interprozesskommunikation
- Synchrone und asynchrone E/A-Operationen

In diesem Kapitel sind wir nur an der Interprozesskommunikation in Form von Message Queues interessiert und wollen uns anschauen, wie wir Nachrichtenwarteschlangen mit den Echtzeit-Weiterungen verarbeiten.

Mit POSIX.4 sind auch viele neue Funktionen für die Bearbeitung von Message Queues eingeführt worden:

mq_close

Schließt eine Nachrichtenwarteschlange, indem die Assoziation zwischen Queue und Queue Descriptor entfernt wird.

mq_getattr

Ruft Informationen und Attribute einer Warteschlange ab.

mq_notify

Registriert den aufrufenden Prozess mit der Warteschlange, um über das Eintreffen neuer Ereignisse informiert zu werden.

mq_open

Etabliert eine Verbindung zwischen dem aufrufenden Prozess und einer Warteschlange.

mq_receive

Ruft die älteste Nachricht mit der höchsten Priorität von der Nachrichtenwarteschlange ab.

mq_send

Platziert eine neue Nachricht in der Warteschlange.

mq_setattr

Setzt Attribute für die Message Queues.

mq_timedreceive

Empfang einer Nachricht innerhalb einer bestimmten Zeitspanne.

mq_timedsend

Platzieren einer Nachricht in der Warteschlange innerhalb einer bestimmten Zeitspanne.

mq_unlink

Entfernt die Nachrichtenwarteschlange von dem Systems.

Nachrichtenwarteschlangen sind einem implementierungsspezifischen Namensraum zugeordnet, beispielsweise einem Pfad im Dateisystem. Des Weiteren verfügen Echtzeitnachrichten über ein Prioritätsfeld und werden gemäß dieses Feldes extrahiert. Das Platzieren und Abholen von Nachrichten kann blockierend (`IPC_WAIT`) oder nicht-blockierend (`IPC_NOWAIT`) geschehen. Die Übertragung und der Empfang von Nachrichten wird nicht synchronisiert, das heißt der Sender wartet nicht auf den Empfänger, bis er die Nachricht erhalten hat, also von der Queue entfernt hat. Der Typ `mqd_t` wird wie alle weiteren Funktionen und Strukturen für die Echtzeiterweiterung in `<mqueue.h>` definiert. Er hat die gleiche Bedeutung wie ein File Descriptor.

Im Zusammenhang mit POSIX Message Queues wird die Struktur `mq_attr` in `<mqueue.h>` definiert und hat folgende Member:

```
struct mq_attr {
    long      mq_flags;    /* message queue flags */
    long      mq_maxmsg;   /* maximum number of messages */
    long      mq_msgsize;  /* maximum message size */
    long      mq_curmsgs;  /* number of messages currently queued */
};
```

Sie wird von einigen Funktionen benötigt.

10.4.2.1 Message Queues erzeugen

Um eine Warteschlange zu erzeugen, verwenden wir die Funktion `mq_open(3)`:

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
```

Rückgabewert: Message Queue Descriptor oder -1 bei Fehler.

name

Je nach Implementierung wird die Warteschlange durch eine Pfadangabe oder einen gewöhnlichen String identifiziert.

oflag

Zeigt den gewünschten Zugriffsmodus an (siehe weiter unten).

Normalweise wird als Name der Queue immer eine Pfadangabe gewählt, beispielsweise `/var/tmp/queue1`. Um sicherzustellen, daß auf dem Zielsystem die gleiche Konvention gilt, sollten stets die Manpages konsultiert werden. Mit Hilfe des *name*-Arguments wird die Warteschlange später durch andere Prozesse referenziert.

Der Parameter *oflag* wird logisch mit den Konstanten aus folgender Liste verknüpft, wobei nur ein Zugriffsbit gleichzeitig mit anderen Einstellungen kombiniert werden kann:

O_RDONLY

Öffnet die Warteschlange nur für den Empfang von Nachrichten. Der zurückgelieferte Descriptor kann im Zusammenhang mit `mq_receive(3)` verwendet werden.

O_WRONLY

Öffnet die Warteschlange nur zum Platzieren von Nachrichten. Somit kann der zurückgelieferte Descriptor nur für die Funktion `mq_send(3)` aber nicht `mq_receive(3)` verwendet werden.

O_RDWR

Öffnet die Warteschlange zum Platzieren und Empfangen von Nachrichten.

Zusammen mit den Zugriffsbits können auch noch die folgenden Einstellungen spezifiziert werden:

O_CREAT

Erzeugt eine neue Message Queue. Diese Option erfordert neben den Zugriffsbits (*oflag*) auch noch einen zweiten, der ein Zeiger auf eine `mq_attr`-Struktur darstellt. Ist der Zeiger ein NULL-Zeiger, so wird eine entsprechende Struktur mit implementierungsspezifischen Standardwerten initialisiert.

O_EXCL

Ist dieses Bit zusammen mit **O_CREAT** gesetzt, so schlägt `mq_open(3)` fehl, wenn die in *name* spezifizierte Warteschlange bereits existiert. Aufgrund der Thread-Sicherheit findet die Prüfung auf Existenz und das Erzeugen einer neuen Queue in einer atomaren Operation statt.

O_NONBLOCK

Zeigt an, daß `mq_send(3)` oder `mq_receive(3)` nicht auf die Verfügbarkeit der Ressourcen warten sollen, sondern stattdessen mit dem Fehlercode `EAGAIN` zurückkehren.

10.4.2.2 Nachrichten absetzen

Nachrichten platzieren wir in der Warteschlange mit `mq_send(3)`:

```
#include <mqqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

mqdes

Message Queue Descriptor der Queue in der wir die Nachricht platzieren wollen.

msg_ptr

Zeiger auf einen String, der die zu übermittelnde Nachricht enthält.

msg_len

Länge der in **msg_ptr** spezifizierten Nachricht.

msg_prio

Angabe der Priorität, die zwischen 0 und MQ_PRIO_MAX liegen muß.

Die Nachricht wird direkt aus dem Buffer des Aufrufers kopiert, so daß er nach einer erfolgreichen Platzierung sofort wiederverwendet werden kann. Wurde beim Aufruf von **mq_open(3)** das Flag **O_NONBLOCK** angegeben kehrt die Funktion sofort zurück und zeigt evtl. **EAGAIN** an. Andernfalls blockiert **mq_send(3)**.

10.4.2.3 Nachrichten empfangen

Der Empfang von Nachrichten findet mit **mq_receive(3)** statt.

```
#include <mqueue.h>
```

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
```

Rückgabewerte: Anzahl der platzierten Bytes oder -1 bei Fehler.

mqdes

Message Queue Descriptor der Queue von der wir die Nachricht abholen wollen.

msg_ptr

Zeiger auf einen String zur Aufnahme der Nachricht.

msg_len

Größe des Buffers in dem die Nachricht gespeichert werden soll.

msg_prio

Zeiger auf einen Integer in der die Priorität der Nachricht gespeichert werden soll.

Die zurückgelieferte Nachricht ist immer die älteste mit der höchsten Priorität.

In vielen Anwendungen gibt es Prozesse oder Threads, die nur Nachrichten verarbeiten. Da macht es Sinn, diese Threads oder Prozesse auszusetzen, wenn es nichts zu tun gibt. Manchmal gibt es auch Prozesse oder Threads, die mehrere Aufgaben gleichzeitig erfüllen. Sie können nicht ohne weiteres ausgesetzt werden, so daß es sinnvoll ist, sie zu beanachrichtigen, wenn es wieder Nachrichten gibt, die bearbeitet werden müssen. Eine Möglichkeit wäre, **O_NONBLOCK** zu setzen und die Warteschlange regelmäßig abzufragen. Noch besser aber wäre der Einsatz der POSIX-Funktion **mq_notify(3)**, mit der Prozesse mit einer Warteschlange registriert werden können und unterrichtet werden, wenn eine Nachricht in der Queue eintrifft. Dazu wird auf eine **sigevent_t**-Struktur zurückgegriffen, mit deren Hilfe entweder ein Signal oder eine Callback-Funktion registriert werden kann.

10.4.2.4 Warteschlange konfigurieren

Mit Hilfe der zwei Funktionen **mq_getattr(3)** und **mq_setattr(3)** können wir die Konfiguration einer Nachrichtenschlange abfragen oder ändern:

```
#include <mqueue.h>
```

```
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat);
```

Rückgabewerte: 0 bei Erfolg und -1 bei Fehler.

mqdes

Message Queue Descriptor der Queue deren Eigenschaften wir anpassen wollen.

mqstat

Zeiger auf eine **mq_attr**-Struktur, die entweder die Einstellungen enthält, die gesetzt werden sollen oder die Einstellungen enthält, nachdem sie mit **mq_getattr(3)** abgefragt wurden.

omqstat

Zeiger auf eine **mq_attr**-Struktur in der die ursprünglichen Einstellungen der Message Queue abgespeichert werden sollen. Ist der Zeiger ein NULL-Zeiger, wird das Argument ignoriert.

10.4.2.5 Beispielprogramme

Im folgenden werden wir insgesamt vier Testprogramme kennenlernen, die den Einsatz der wichtigsten Funktionen erläutern. Die Quellen gehen auf eine Entwicklung von Silicon Graphics zurück und wurden freundlicherweise für diesen Text zur Verfügung gestellt. Betrachten wir zunächst, wie wir die Programme einsetzen können und besprechen wir hinterher den Quelltext.

Mit **mqrt_open** erzeugen wir eine Queue mit bestimmten Eigenschaften.

```
% mqrt_open -p 0664 -b 128 -m 32 -c -x /var/tmp/queue1
Flags: 0x0
Maximum messages: 32
Maximum message size: 128
Current number of messages: 0
```

In diesem Fall wird eine Queue mit den Zugriffsrechten 0664, einer maximalen Nachrichtenlänge von 128 Bytes, die maximal 32 Nachrichten aufnehmen kann und in dem Namensraum **/var/tmp/queue1** mit den Flags **O_CREAT** (-c) und **O_EXCL** (-x) angelegt wird.

Listing 10.17

```

1  /*
2   * Program to test mq_open(3)
3   *
4   * Usage: mq_open [-p <perms>] [-b <bytes>] [-m <msgs>] [-c] [-x] <path>
5   *
6   *      -p <perms>    access mode to use when creating, default 0600
7   *      -b <bytes>     maximum message size to set, default MQ_DEF_MSGSIZE
8   *      -m <msgs>      maximum messages on the queue, default MQ_DEF_MAXMSG
9   *      -f <flags>     flags to use with mq_open, including:
10  *                      c      use O_CREAT
11  *                      x      use O_EXCL
12  *      <path>         the pathname of the queue, required
13  *
14  * Numeric arguments can be given in any form supported by strtoul(3).
15  */
16
17 #include <mqueue.h> /* message queue */
18 #include "header.h"
19
20 #define MQ_DEF_MSGSIZE 512
21 #define MQ_DEF_MAXMSG 16
22
23 int main(int argc, char **argv) {
24     mode_t perms = 0600;
25     int     oflags = O_RDWR; /* to combine flags */
```

```

26     int      rd = 0, wr = 0; /* store -r and -w options */
27     mqd_t    mqd;           /* queue descriptor */
28     int      c;
29     char    *path;
30     struct   mq_attr  buf;
31
32     buf.mq_msgsize = MQ_DEF_MSGSIZE;
33     buf.mq_maxmsg = MQ_DEF_MAXMSG;
34
35     while ((c = getopt(argc, argv, "p:b:m:cx")) != -1) {
36         switch (c) {
37             case 'p': /* permissions */
38                 perms = (int)strtoul(optarg, NULL, 0);
39                 break;
40             case 'b': /* message size */
41                 buf.mq_msgsize = (int)strtoul(optarg, NULL, 0);
42                 break;
43             case 'm': /* max messages */
44                 buf.mq_maxmsg = (int)strtoul(optarg, NULL, 0);
45                 break;
46             case 'c': /* use O_CREAT */
47                 oflags |= O_CREAT;
48                 break;
49             case 'x': /* use O_EXCL */
50                 oflags |= O_EXCL;
51                 break;
52             default: /* Oops! Wrong args specified */
53                 return -1;
54         }
55     }
56
57     if (optind < argc)
58         path = argv[optind]; /* first non-option argument */
59     else
60         err_fatal("Pathname for queue missing\n");
61
62     if ((mqd = mq_open(path, oflags, perms, &buf)) < 0)
63         err_fatal("mqopen() failed");
64     else
65         if (mq_getattr(mqd, &buf) != 0)
66             err_fatal("mq_getattr() failed");
67         else
68             printf("\tFlags: 0x%x\n      \
69                   \tMax messages: %d\n \
70                   \tMessage size: %d\n \
71                   \tCurrent message: %d\n",
72                   buf.mq_flags, buf.mq_maxmsg,
73                   buf.mq_msgsize, buf.mq_curmsgs);
74     return (0);
75 }
```

Listing 10.17: xcode/mqrt_open.c - Code für mqrt_open.

Wir verwenden `getopt(3)` um die Argumente zu extrahieren und rufen anschließend `mq_open(3)` mit den jeweiligen Einstellungen auf. Mit einem Aufruf von `mq_attr(3)` können wir überprüfen, ob alles wie gewünscht funktioniert.

Nun können wir versuchen, eine Nachricht mit `mqrt_send` in der Queue zu platzieren.

```
% mqrt_send -b 129 /var/tmp/queue1
mq_send(): Inappropriate message buffer length
```

Der Aufruf schlägt fehl, weil wir nur Nachrichten mit einer Länge von 128 Bit platzieren dürfen (was beim Aufruf von `mqrt_open` festgelegt wurde).

Der zweite Versuch gelingt und wir überprüfen den Erfolg mit einem Aufruf von `mqrt_attr`. Das Argument `-p 7` legt eine Priorität von 7 für diese Nachricht fest.

```
% mqrt_send -b 128 -p 7 /var/tmp/queue1
```

Listing 10.18

```

1  /*
2   * Program to test mq_send(3)
3   *
4   * mq_send [-p <priority>] [-b <bytes>] [-c <count>] [-n] <path>
5   *
6   *      -p <priority> priority code to use, default 0
7   *      -b <bytes>     size of the message, default 64, min 32
8   *      -c <count>      number of messages to send, default 1, max 9999
9   *      -n              use O_NONBLOCK flag in open
10  *      <path>          path to queue, required
11  *
12  * The program sends <count> messages of <bytes> each at <priority>.
13  * Each message is an ASCII string containing the time and date and
14  * a serial number 1..<count>. The minimum message is 32 bytes.
15  */
16
17 #include <mqueue.h> /* message queue */
18 #include <time.h>
19 #include "header.h"
20
21 int main(int argc, char **argv) {
22     char          *path;
23     int           oflags = O_WRONLY; /* to combine flags */
24     mqd_t         mqd;             /* descriptor */
25     unsigned int msg_prio = 0;    /* priority */
26     size_t        msglen = 64;    /* message size */
27     int           count = 1;      /* number of messages to send */
28     char          *msgptr;
29     int           c;
30
31     while ((c = getopt(argc, argv, "p:b:c:n")) != -1) {
32         switch (c) {
33             case 'p': /* priority */
34                 msg_prio = strtoul(optarg, NULL, 0);
35                 break;
36             case 'b': /* bytes */
37                 msglen = strtoul(optarg, NULL, 0);
38                 if (msglen < 32)
39                     msglen = 32;
40                 break;
41             case 'c': /* count */
42                 count = strtoul(optarg, NULL, 0);
43                 if (count > 9999)
44                     count = 9999;
45                 break;
46             case 'n': /* use nonblock */
47                 oflags |= O_NONBLOCK;
48                 break;
49             default: /* unknown or missing argument */
50                 return -1;
51         }

```

```

52     }
53
54     if (optind < argc)
55         path = argv[optind];
56     else
57         err_fatal("Pathname for queue missing\n");
58
59     msgptr = calloc(1, msglen);
60
61     if ((mqd = mq_open(path, oflags)) < 0) {
62         err_fatal("mq_open failed for O_WRONLY");
63     } else {
64         char stime[26];
65         const time_t tm = time(NULL); /* current time value */
66         ctime_r(&tm, stime);           /* formatted time string */
67         stime[24] = '\0';             /* drop annoying \n */
68
69         for (c = 1; c <= count; c++) {
70             sprintf(msgptr, "%05d %s", c, stime);
71
72             if (mq_send(mqd, msgptr, msglen, msg_prio)) {
73                 err_normal("mq_send failed");
74                 break;
75             }
76         }
77     }
78
79     return (0);
80 }
```

Listing 10.18: xcode/mqrt_send.c - Code für mqrt_send.

Um herauszufinden, ob die Nachricht erfolgreich in die Nachrichtenwarteschlange eingereiht wurde, rufen wir `mqrt_attr` auf:

```
% mqrt_attr /var/tmp/queue1
Flags: 0x0
Maximum messages: 32
Maximum message size: 128
Current number of messages: 1
```

Listing 10.19

```

1 #include <mqueue.h>      /* message queue */
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     char *basename = basename_ex(argv[0]);
6     mqd_t mqd;           /* descriptor */
7     struct mq_attr buf;  /* buffer for getattr */
8
9     if (argc != 2)
10         err_fatal("Usage: %s <queue>\n", basename);
11
12     if ((mqd = mq_open(argv[1], O_RDONLY)) < 0)
13         err_fatal("mq_open failed");
14     else
15         if (mq_getattr(mqd, &buf) != 0)
16             err_fatal("mq_getattr() failed");
17     else
```

```

18     printf("\tFlags: 0x%x\n"
19         "\tMax messages: %d\n"
20         "\tMessage size: %d\n"
21         "\tCurrent message: %d\n",
22         buf.mq_flags, buf.mq_maxmsg,
23         buf.mq_msgsize, buf.mq_curmsgs);
24     return (0);
25 }

```

Listing 10.19: xcode/mqrt_attr.c - Code für mqrt_attr.

Versuchen wir nun, eine zweite Nachricht zu platzieren, um die Funktionalität des Empfangs zu prüfen.

```
% mqrt_send -p 19 /var/tmp/queue1
% mqrt_attr /var/tmp/queue1
    Flags: 0x0
    Maximum messages: 32
    Maximum message size: 128
    Current number of messages: 2
```

Beim Abrufen der platzierten Nachrichten mit `mqrt_receive` stellen wir fest, daß sich die Reihenfolge geändert hat, denn Nachrichten mit höherer Priorität werden zuerst abgerufen:

```
% mqrt_receive -c 2 /var/tmp/queue1
1: priority 19, length 63, text 00001, time: Fri Jun 14 09:19:12 2003
2: priority 7, length 128, text 00001, time: Fri Jun 14 09:17:15 2003
```

Listing 10.20

```

1  /*
2   * Program to test mq_receive
3   *
4   * mq_receive [-c <count>] [-n] [-q] <path>
5   *   -c <count>      number of messages to request, default 1
6   *   -n              use O_NONBLOCK flag on open
7   *   -q              quiet, do not display messages
8   *   <path>          path to message queue, required
9   *
10  * The program calls mq_receive <count> times or until an error occurs.
11 */
12
13 #include <mqueue.h> /* message queue stuff */
14 #include "header.h"
15
16 int main(int argc, char **argv) {
17     char          *path;
18     int           oflags = O_RDONLY; /* to combine flags */
19     int           quiet = 0;        /* -q option */
20     int           count = 1;       /* number of messages to request */
21     mqd_t         mqd;           /* descriptor */
22     char          *msgptr;        /* -> allocated message space */
23     unsigned int  msg_prio;      /* received message priority */
24     int           c, ret;
25     struct        mq_attr obuf;  /* output of mq_getattr(): mq_msgsize */
26
27     while ((c = getopt(argc, argv, "c:nq")) != -1) {
28         switch (c) {
29             case 'c': /* count */
30                 count = strtoul(optarg, NULL, 0);
31                 break;

```

```

32         case 'q': /* quiet */
33             quiet = 1;
34             break;
35         case 'n': /* nonblock */
36             oflags |= O_NONBLOCK;
37             break;
38         default: /* unknown or missing argument */
39             return -1;
40     }
41 }
42
43 if (optind < argc)
44     path = argv[optind];
45 else
46     err_fatal("Pathname for queue missing\n");
47
48 mqd = mq_open(path, oflags);
49
50 if (-1 != mqd) {
51     if (!(mq_getattr(mqd, &obuf))) { /* get max message size */
52         msgptr = calloc(1, obuf.mq_msgsize);
53
54         for( c = 1; c <= count; c++) {
55             ret = mq_receive(mqd, msgptr, obuf.mq_msgsize, &msg_prio);
56
57             if (ret >= 0) { /* got a message */
58                 if (!quiet) {
59                     if (isascii(*msgptr)) {
60                         printf("%d: priority %ld, length %d,
61                             text %-32.32s\n",
62                             c, msg_prio, ret, msgptr);
63                     } else {
64                         printf("%d: priority %ld, length %d,
65                             (nonascii)\n",
66                             c, msg_prio, ret);
67                     }
68                 }
69             } else { /* an error on receive, stop */
70                 err_normal("mq_receive failed");
71                 break;
72             }
73         }
74     } else {
75         err_fatal("mq_getattr failed");
76     }
77 } else {
78     err_fatal("mq_open failed for O_WRONLY");
79 }
80
81 return (0);
82 }

```

Listing 10.20: xcode/mqrtr_receive.c - Code für mqrtr_receive.



10.5 POSIX-Semaphores

Das Prinzip des gegenseitigen Ausschlusses (in der Fachsprache *Mutual Exclusion* genannt) wird immer dann angewendet, wenn der Zugriff auf eine begrenzte Zahl von Ressourcen, die mehreren Prozessen oder

Threads zur Verfügung stehen, gesteuert werden soll. Spezielle Objekte, Locks (auch Mutexes genannt) und Semaphoren, stehen uns zur Synchronisierung der Zugriffe zur Verfügung. Mit ihrer Hilfe können wir

- sicherstellen, daß immer nur ein Thread oder Prozess auf eine bestimmte Datenstruktur zugreift,
- Aktivitäten synchronisieren, so daß Prozesse oder Threads überprüfen können, ob eine Datenstruktur von einem Thread oder Prozess verwendet wird.

Bisher haben wir erfahren, wie wir Daten zwischen Prozessen austauschen können, in dem wir *Shared Memory* verwenden und den Zugriff auf Dateien mit Hilfe des *File Locking* koordinieren. Wenn wir aber von Ressourcen sprechen sind in der Regel globale Datenstrukturen, Geräte oder Schnittstellen gemeint.

Semaphores¹, ein Begriff, den der niederländische Forscher E. W. Dijkstra geprägt hat, sind seit 1993 in POSIX.1b definiert. Ob ein System POSIX-Semaphores unterstützt, kann mit der Konstante `_POSIX_SEMAPHORES` geprüft werden. Der Datentyp eines Semaphore ist `sem_t`. Zur Handhabung von Semaphores sind ausschließlich atomare Funktionen vorgesehen.

Ein UNIX-Semaphore, der über eine Ressource wacht, verwaltet einen internen Zähler und benachrichtigt interessierte Prozesse, wenn sich der Zähler ändert. Der Zähler zeigt an, ob ein Prozess die Ressource kontrolliert und ob ein Thread auf dieses Semaphore wartet. Die denkbar einfachste Variante ist ein binäres Semaphore, das die beiden Werte 1 und 0 annehmen kann. Mutexes sind solche binären Semaphores, die vor allem im Zusammenhang mit Threads (siehe Kapitel 11 *POSIX-Threads*) Anwendung finden.

Andere Semaphores können bis zu n Instanzen einer bestimmten Resource verwalten. Nehmen wir an, uns stehen drei Server für die Transaktionsverarbeitung für Clients zur Verfügung. Also initialisieren wir das Semaphore mit einem Wert von 3. Jedesmal wenn sich ein Client mit dem Server verbindet, wird das Semaphore um 1 dekrementiert. Ein Wert von 0 zeigt an, daß alle Ressourcen aufgebraucht sind und kein weiterer Client bedient werden kann. Sobald der Server eine Clientanfrage verarbeitet hat, wird eine Verbindung frei, das Semaphore um 1 inkrementiert, so daß sich neue Clients mit dem Server verbinden können. Wurden alle Clients bedient, hat das Semaphore seinen ursprünglichen Wert angenommen.

Wenn ein Semaphore dekrementiert wird, spricht man auch vom *Warten auf das Semaphore*. In der Informatik ist diese Operation als *wait*, *down* oder *P* bekannt. Die gegenteilige Operation, das Inkrementieren eines Semaphores, wird als *Benachrichtigung eines Semaphores* (*notification*), bzw. *signal*, *up* oder *V* bezeichnet.

10.5.1 Kritische Sektionen

Eine *kritische Sektion* ist ein Codesegment, das so ausgeführt werden muß, daß immer nur ein Ausführungs-Thread innerhalb dieser Sektion aktiv sein kann. Wird beispielsweise in diesem Segment eine gemeinsame Resource gelesen oder modifiziert ist dieses Segment Teil einer kritischen Sektion, insbesondere wenn es andere Threads gibt, die ebenfalls Zugriff auf diese Resource anfordern.

Beispiel 10.21 zeigt ein Programm, daß eine benutzerdefinierte Zahl von Child-Prozessen erzeugt, die alle ein paar Informationen auf der Standardausgabe ausgeben. Es enthält eine kritische Sektion, die alle Child-Prozesse ausführen, so daß Unregelmäßigkeiten bei der Ausgabe auftreten können.

Listing 10.21: Demonstration einer kritischen Sektion

```

1 #include <unistd.h>
2 #include <sys/wait.h>
3 #include <sys/types.h>
4 #include "errors.h"
5

```

¹Semaphore: ein Anglizismus, der nur schwer übersetzt werden kann. Aus diesem Grund bleibe ich im weiteren Verlauf dabei und verweise auf *das Semaphore* und *die Semaphores*, um der englischen Grammatik zu folgen. Das gleiche gilt auch für den Begriff Mutex: *der Mutex* und *die Mutexes*

```

6  #define BUF_SIZ 1024
7
8  int main(int argc, char **argv) {
9      char buffer[BUF_SIZ], *c;
10     char *basename = basename_ex(argv[0]);
11     pid_t childpid = 0, child_ret;
12     int waitval = 1000, i, num_proc, status, dummy = 0;
13
14     if (argc != 2)
15         err_fatal("Usage: %s <num_processes>\n", basename);
16
17     num_proc = atoi(argv[1]);
18
19     for (i = 1; i < num_proc; i++) {
20         if ((childpid = fork()) > 0) {
21             printf("Some output to illustrate misbehavior\n");
22             break;
23         }
24     }
25
26     sprintf(buffer, BUF_SIZ,
27             "Process: %d\tPID: %ld\tparent PID: %ld\tchild PID: %ld\n",
28             i, (long)getpid(), (long)getppid(), (long)childpid);
29
30     c = buffer;
31
32     /* START CRITICAL SECTION */
33     while (*c != '\0') {
34         fputc(*c, stdout);
35         c++;
36
37         for (i = 0; i < waitval; i++)
38             dummy++;
39     }
39     /* END CRITICAL SECTION */
40
41     while (((child_ret = wait(&status)) == -1) && (errno == EINTR));
42
43     return 0;
44 }
```

Listing 10.21: `xcode/critical_demo.c` - Code für `critical_demo.c`.

Der erste Teil des Beispiels ist schnell erklärt. Zuerst holen wir die Anzahl der zu erzeugenden Child-Prozesse aus dem Kommandozeilenargument ab und rufen anschließend `fork(2)` auf, der `argv[1]` Prozesse erzeugt (hier: 5). Jetzt legen wir nur noch schnell einen Buffer für die Ausgabe an und kommen dann zur kritischen Sektion des Programms. Hier geben wir jedes Zeichen des Buffers einzeln aus und tauchen in eine kleine Schleife ein, die nur eine geringfügige Verzögerung zur Folge hat. Da alle Child-Prozesse diese Sektion durchlaufen und damit konkurrierend auf die Standardausgabe zugreifen, ist das Ergebnis nicht genau das, was wir erwarten würden:

```
% ./critical_demo 5 | erste Ausfuehrung |
Some output to illustrate misbehavior
Process: 1      PID: 17988      parent PID: 4335      child PID: 17989
Process: 5      PID: 17992      parent PID: 17991      child PID: 0
Some output to illustrate misbehavior
Process: 4      PID: 17991      parent PID: 17990      child PID: 17992
Some output to illustrate misbehavior
Process: 3      PID: 17990      parent PID: 17989      child PID: 17991
Some output to illustrate misbehavior
Process: 2      PID: 17989      parent PID: 17988      child PID: 17990
```

```
% ./critical_demo 5 | zweite Ausfuehrung |
Some output to illustrate misbehavior
Process: 1      PID: 17988      parent PID: 4335      child PID: 17989
Process: 5      PID: 17992      parent PID: 17991      child PID: 0
Some output to illustrate misbehavior
Process: 4      PID: 17991      parent PID: 17990      child PID: 17992
Some output to illustrate misbehavior
Some output to illustrate misbehavior
Process: 3      PID: 17990      parent PID: 17989      child PID: 17991
Process: 2      PID: 17989      parent PID: 17988      child PID: 17990
```

Wir können sehen, daß die Reihenfolge der Ausgabe durch die jeweiligen Child-Prozesse nicht der Reihenfolge der Erzeugung durch `fork(2)` entspricht. Auch nicht beim zweiten mal. Das liegt ganz einfach daran, daß mehrere Prozesse CPU-Zeit anfordern und der Scheduler des Kernels entscheidet, wann welcher Prozess an der Reihe ist. □

Der Umgang mit Semaphores findet eigentlich in drei Schritten statt:

1. Eintritt in die sog. *Entry Section*. In diesem Teil fordern wir ein Semaphore an.

```
while (sem_wait(ptrsem) < 0)
    if (errno != EINTR)
        err_fatal("sem_wait() failed");
```

Dieser Teil des Quellcodes wird auch als *Gatekeeper* bezeichnet, da der Eintritt in die kritische Sektion nur gestattet werden kann, wenn es die Entry Section (also der Gatekeeper) zuläßt.

2. Eintritt in die kritisch Sektion.
3. Direkt nach dem Austritt aus der kritischen Sektion wird in die sog. *Exit Section* übergegangen in der das involvierte Semaphore angepaßt wird.

```
if (sem_post(ptrsem) < 0)
    err_fatal("sem_post() failed");
```

10.5.2 Unbenannte (anonyme) Semaphores

Jedes System, das Semaphores nach der X/Open-Erweiterung implementiert, kann mit unbenannten (anonymen) und benannten Semaphores umgehen. Der Unterschied zwischen beiden Varianten ist der gleiche wie zwischen anonymen Pipes und benannten Pipes: während unbenannte Semaphores nur zwischen Parent und Child, bzw. Child und Child genutzt werden können, stehen benannte Semaphores auch anderen Prozessen außerhalb des gleichen Vererbungszweiges zur Verfügung, da sie (genau wie benannte Pipes) über Pfadnamen referenziert werden.

Es ist einfach, einen Semaphore zu definieren:

```
#include <sys/sem.h>
sem_t semaphore;
```

Damit deklarieren wir die Variable `semaphore` vom Typ `sem_t`. Bevor wir sie verwenden können, muß sie initialisiert werden, was mit der Funktion `sem_init(2)` erledigt werden kann.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

sem

Semaphore, das wir initialisieren möchten.

pshared

Zeigt an, ob das Semaphore auch von anderen Prozessen verwendet werden kann.

value

Wert mit dem das Semaphore initialisiert werden soll.

Wenn **pshared** nicht 0 ist, kann das Semaphore auch von anderen Prozessen über Funktionen wie **sem_wait(2)**, **sem_trywait(2)**, **sem_post(2)**, und **sem_destroy(2)** verwendet werden. Steht uns nur eine Einheit einer bestimmten Resource zur Verfügung kann **sem** mit einem **value** von 1 initialisiert werden:

```
sem_t my_semaphore;

if (sem_init(&my_semaphore, 0, 1) < 0)
    err_fatal("sem_init failed");
```

In diesem Fall kann das Semaphore nur von Threads dieses Prozesses genutzt werden (**pshared** = 0). Unter Umständen können wir auf mehrere Einheiten einer Resource zurückgreifen, beispielsweise wie bei einer Druckerwarteschlange, die über mehrere Slots verfügt, so wäre es sinnvoll mehr als einem Thread oder Prozess Zugang zu dieser Resourcen zu gewähren. Dazu könnte **value** auf 5 gesetzt werden, so daß 5 Threads oder 5 Prozesse (**pshared** != 0) gleichzeitig auf jeweils einen Slot der Druckerwarteschlange zugreifen können.

Sollen mehrere Prozesse mit unterschiedlichen Addressräumen das Semaphore nutzen, muß es in einem Shared Memory Segment abgelegt werden, auf das diese Prozesse Zugriff haben, da es nur im Speicher existiert.

Vor Beendigung einer Anwendung sollten alle unbenannten Semaphores zerstört werden. Das erreichen wir mit der Funktion **sem_destroy(2)**:

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

sem

Semaphore, das neutralisiert werden soll.

Die Freigabe eines Semaphores geschieht auf die gleiche Weise wie das Initialisieren:

```
sem_t my_semaphore;
...
if (sem_destroy(&my_semaphore) < 0)
    err_fatal("sem_destroy failed");
```

Warten ein oder mehrere Threads auf das Semaphore, das wir freigeben möchten, so ist das Verhalten der Funktion undefiniert, kann also je nach System anders gehandhabt werden. Linux, Solaris und HP-UX beispielsweise, liefern **EBUSY** zurück und zeigen damit an, daß es noch Threads gibt, die auf dieses Semaphore warten. AIX liefert dagegen **EACCESS** und Irix **EACCESS** zurück.

10.5.3 Benannte Semaphores

Prozesse, die über keinen gemeinsamen Speicher, beispielsweise in einer Parent-Child-Beziehung oder Shared Memory (siehe Abschnitt 10.3 *Shared Memory*) verfügen können den Zugriff synchronisieren,

indem sie benannte Semaphores verwenden. Der Schlüssel eines benannten Semaphores ist eine Zeichenkette, die als gültiger Pfad im System aufgelöst werden kann. Allerdings sollten sie beachten, das POSIX weder zwingend vorschreibt, daß der Pfad gültig ist, noch erläutert, ob der Pfad für andere Funktionen, die Pfadnamen verarbeiten, sichtbar ist. Grundsätzlich ist es empfehlenswert, stets einen absoluten Pfad (beginnend mit einem Slash '/') zu wählen, um sicherzustellen, daß der Namensraum eines Semaphore für alle Prozesse der gleiche ist.

Wir öffnen und erzeugen benannte Semaphores mit der Funktion `sem_open(2)`.

```
#include <semaphore.h>

int *sem_open(const char *name, int oflag, ...);
```

Rückgabewert: Adresse des geöffneten oder erzeugten Semaphores oder SEM_FAILED bei Fehler.

name

Bezeichner des zu erzeugenden oder zu öffnenden Semaphore.

flags

Legt fest, ob das Semaphore erzeugt oder mit Hilfe der Funktion ein Semaphore geöffnet werden soll.

mode

Bestimmt die Zugriffsrechte des Semaphores.

value

Initialer Wert des Semaphore.

Ist das `O_CREAT`-Flag in `oflag` gesetzt, so sind zwei weitere Parameter erforderlich: `mode` und `value`; ersteres ist vom Typ `mode_t` und bestimmt die Zugriffsrechte des Semaphore, die mit denen von `umask(2)` verknüpft werden. Letzteres stellt den initialen Wert des Semaphore dar. Wurden `O_CREAT` und `O_EXCL` für `flags` gesetzt, so wird die Funktion einen Fehlercode zurückliefern, falls das Semaphore bereits existiert.

Um das Semaphore `/tmp/sem` zu öffnen, oder zu erstellen, falls es noch nicht existiert, würden wir folgenden einfachen Aufruf verwenden:

```
sem_t *my_sem;
my_sem = sem_open("/tmp/sem", O_CREAT, S_IRUSR | S_IWUSR, 5);
```

Es wurde mit einem Wert von 5 initialisiert und nur der Besitzer kann das Semaphore `my_sem` für Aufrufe von `sem_wait`, `sem_trywait`, `sem_post`, `sem_getvalue`, `sem_close`, etc. verwenden.

10.5.4 Semaphores einsetzen

Bevor wir in eine kritische Sektion eintreten können, müssen wir am Gatekeeper vorbei. Dieser prüft mit der Funktion `sem_wait(2)` oder `sem_trywait(2)`, ob uns Zugang zur kritischen Sektion gewährt werden kann oder nicht. Wenn der Wert des Semaphore 0 ist, wird der aufrufende Thread blockieren, bis die Blockierung durch einen Aufruf von `sem_post(2)` oder durch ein Signal wieder aufgehoben wird. Ganz ähnlich funktioniert `sem_trywait(2)`, allerdings liefert die Funktion -1 (`EAGAIN`) zurück, wenn der Versuch, das Semaphore zu dekrementieren, fehlschlägt, und blockiert den Aufrufer nicht.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

Rückgabewerte: 0 bei Erfolg und -1 bei Fehler.

sem

Semaphore dessen Wert dekrementiert werden soll.

Nachdem eine kritische Sektion verarbeitet wurde, müssen wir dafür sorgen, daß wartende Threads, also jene, die beim Aufruf von **sem_wait(2)** blockieren, darüber benachrichtigt werden, daß sie in die kritische Sektion eintreten können. Dieses Signal setzen wir mit der Funktion **sem_post(2)** ab.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

sem

Semaphore dessen Wert inkrementiert werden soll.

Wartet kein Thread auf **sem**, wird **sem_post(2)** den Wert des Semaphore inkrementieren. Wartet auch nur ein Thread auf **sem** (weil er beispielsweise von **sem_wait(2)** blockiert wird), so hebt die Funktion die Blockierung wieder auf.

Kombinieren wir die drei Funktionen **sem_open(2)**, **sem_wait(2)/sem_trywait(2)** und **sem_post(2)**, haben wir alle notwendigen Mittel, eine kritische Sektion zu schützen:

```
if ((ptrsem = sem_open("/tmp/my_sem", O_CREAT, S_IRUSR | S_IWUSR, 5)) < 0)
    err_fatal("sem_open() failed");

while (sem_wait(ptrsem) < 0)
    if (errno != EINTR)
        err_fatal("sem_wait() failed");

/* critical section */

if (sem_post(ptrsem) < 0)
    err_fatal("sem_post failed");
```

Möchte ein Prozess oder Thread wissen, welchen Wert ein Semaphore momentan aufweist, so kann die Funktion **sem_getvalue(2)** bemüht werden. Das funktioniert mit benannten und unbenannten Semaphores.

```
#include <semaphore.h>
```

```
int sem_getvalue(sem_t *sem, int *sval);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

sem

Semaphore dessen Wert ermittelt werden soll.

sval

Zeiger auf einen Integer in dem der Wert des Semaphore gespeichert werden soll.

Ganz offensichtlich wird **sval** für die Rückgabe des Wertes verwendet. Dabei müssen wir aber einiges beachten, denn zum Einen gibt die Funktion den Wert des Semaphores zu einem *undefinierten* Zeitpunkt zurück, so daß es nicht unbedingt der Wert ist, den das Semaphore hat, wenn die Funktion zurückkehrt. Zum Anderen kann der Wert 0 oder negativ sein, was bedeutet, daß immer der absolute Wert die Anzahl der wartenden Threads (oder Prozesse) repräsentiert.

Um Semaphores zu schließen verwenden wir die Funktion **sem_close(2)**.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Rückgabewert: 0 bei Erfolg oder -1 bei Fehler.

Die `sem_close(2)`-Funktion gibt alle mit dem Semaphore assoziierten Ressourcen frei, beeinflusst den Zustand des Semaphores aber nicht. Der Zähler bleibt erhalten, denn es könnten ja noch andere Prozesse diesen Semaphore verwenden.

Betrachten wir ein Beispielprogramm, das den Einsatz der beschriebenen Funktionen illustriert.

Listing 10.22: Umgang mit kritischen Sektionen

```

1 #include <semaphore.h>
2 #include "header.h"
3
4 #define SEM_NAME "/mysem"
5
6 int main ( int argc,  char *argv[] ) {
7     int i, n = 3;
8     pid_t chldpid;
9     char buffer[256];
10    char *c;
11    char *basename = basename_ex(argv[0]);
12    //pid_t chldpid;
13    sem_t *mylock;
14
15    if ((argc != 2) || ((n = atoi(argv[1])) <= 0))
16        err_fatal("Usage: %s <number-of-processes>\n", basename);
17
18    if (sem_unlink(SEM_NAME))
19        err_normal("unlink() failed");
20
21    mylock = sem_open(SEM_NAME, O_CREAT, 0777, (unsigned)1);
22
23    if (mylock < 0)
24        err_fatal("sem_open() failed");
25
26    for (i = 0;  i < n; i++) {
27        chldpid = fork();
28
29        if (chldpid != 0)
30            break;
31    }
32
33    sprintf(buffer, "i: %d process ID: %d parent ID: %ld child ID: %ld\n",
34            i, (long)getpid(), (long)getppid(), (long)chldpid);
35
36    c = buffer;
37
38    /* synchronized critical section - it will cause printout to be serialized */
39    if (sem_wait(mylock) < 0)
40        perror("sem_wait() failed");
41
42    /* begin critical section */
43    while (*c != '\0') {
44        fputc(*c, stderr);
45        c++;
46    }
47    /* end of critical section */
```

```

48     /* exit critical section */
49     if (sem_post(mylock) < 0)
50         err_fatal("sem_post() failed");
51
52     return (0);
53 }

```

Listing 10.22: xcode/sem_open.c - Beispielprogramm für den richtigen Umgang mit kritischen Sektionen.

Der erste Schritt sieht die Anforderung eines benannten Semaphores durch die Funktion `sem_open(2)` vor. Anschließend rufen wir `sem_wait(2)` auf und zeigen damit an, daß wir Zugriff auf die kritische Sektion begehren. Gibt es noch keinen Prozess der auf das Semaphore wartet, wird das Semaphore dekrementiert und wir treten in die kritische Sektion ein. Hinterher inkrementieren wir das Semaphore durch einen Aufruf von `sem_post(2)`, so daß andere Prozesse die Gelegenheit erhalten in die kritische Sektion eintreten zu können. □

10.5.5 Semaphore Sets

Neben den Semaphores, die wir in den letzten Abschnitten kennengelernt haben, existieren noch *Semaphore Sets*. Sie sind im Grunde nur Arrays von Semaphores und funktionieren so ähnlich wie die Integer-basierten, traditionellen Semaphores. Ein Unterschied besteht darin, daß wir einen ganzen Satz von Semaphores mit nur einem Funktionsaufruf manipulieren können.

Um einen Semaphore Set zu erzeugen, verwenden wir die Funktion `semget(3)`.

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Rückgabewert: Positiver Semaphore-Bezeichner oder -1 bei Fehler.

key

Ist `key` auf `IPC_PRIVATE` gesetzt oder kein Semaphore mit `key` verbunden und `IPC_CREAT` in `semflg` gesetzt, wird ein neuer Satz von Semaphores, nämlich `nsems`, erzeugt.

key

Anzahl der zu erzeugenden Semaphores in diesem Satz.

semflg

Zeigt an, wie der Semaphore-Satz erzeugt werden soll. Die Präsenz von `IPC_CREAT` und `IPC_EXCL` führt zu einem Fehler, wenn das durch `key` referenzierte Semaphore bereits existiert.

Jedem Semaphore ist, wie bei allen anderen bisher besprochenen IPC-Strukturen auch, eine `semid_ps`-Struktur zugeordnet:

```

struct semid_ds {
    struct ipc_perm sem_perm; /* operation permission struct */
    struct sem *sem_base; /* ptr to first semaphore in set */
    ushort sem_nsems; /* # of semaphores in set */
    time_t sem_otime; /* last semop time */
    time_t sem_ctime; /* last change time */
};

```

Die `sem`-Struktur wird zur Verwaltung eines einzelnen Semaphores in einem Satz benötigt und intern als eine verkettete Liste gespeichert.

```

struct sem {
    ushort  semval; /* semaphore value */
    pid_t   sempid; /* pid of last operation */
    ushort  semncnt; /* # awaiting semval > cval */
    ushort  semzcnt; /* # awaiting semval = 0 */
};

```

Um einen Satz von Semaphores zu erzeugen, könnten wir folgendermaßen vorgehen:

```

int sem_id;

if ((sem_id = semget(IPC_PRIVATE, 3, S_IRUSR | S_IWUSR)) == -1)
    err_fatal("semget failed for IPC_PRIVATE");

```

Damit erzeugen wir ein Semaphore Set mit drei Elementen, das nur von Prozessen in einer Parent-Child- oder Child-Child-Beziehung sichtbar ist. Andere Prozesse können nicht auf diesen Semaphore über key zugreifen.

Das nächste Beispiel verschafft uns Zugriff auf ein Semaphore Set mit dem Schlüssel 1001 mit nur einem Element. Wenn der Satz noch nicht existiert, wird er angelegt, erkennbar an dem Flag IPC_CREAT.

```

#define PERM (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

key_t key = 1001;
int sem_id;

if ((sem_id = semget(key, 1, PERM | IPC_CREAT)) == -1)
    err_fatal("semget failed for %d", (int)key);

```

Alternativ könnten wir auch noch das Flag IPC_EXCL spezifizieren, um sicherzustellen, daß ein Semaphore Set mit dem Schlüssel key auch tatsächlich neu erzeugt wird und die Funktion fehlschlägt, wenn der Satz bereits existiert.

Listing 10.23: Anwendung von Semaphore Sets

```

1 #include <sys/sem.h>
2 #include <sys/stat.h> /* for permission flags */
3 #include <fcntl.h>
4 #include "header.h"
5
6 #define PERM      (0644)
7 #define KEY_FILE  ("/tmp/semset.sem")
8
9 int main(int argc, char **argv) {
10     key_t key;
11     char *basename = basename_ex(argv[0]);
12     int sem_id;
13
14     if (open(KEY_FILE, O_CREAT, 0644) < 0)
15         err_fatal("open failed for %s\n", KEY_FILE);
16
17     if ((key = ftok(KEY_FILE, 127)) < 0)
18         err_fatal("ftok failed for %s", KEY_FILE);
19
20     if ((sem_id = semget(key, 1, PERM | IPC_CREAT)) < 0)
21         err_fatal("semget failed for %s", (int)key);
22     else
23         printf("semget returned ID: %d\n", sem_id);
24
25     if (unlink(KEY_FILE) < 0)
26         err_fatal("unlink failed for %s", KEY_FILE);

```

```

27     return (0);
28 }
29 }
```

Listing 10.23: xcode/semsetdemo1.c - Einfaches Demo zur Anwendung von Semaphore Sets.

Das Programm erzeugt zuerst eine Datei, leitet einen Schlüssel für die Erzeugung eines Semaphore Sets ab und verwendet ihn dann für `semget(2)`. Am Ende wird noch die ID des Semaphore Sets ausgegeben:

```
% ./semsetdemo1
semget returned ID: 32769
```

Natürlich erhalten wir mit dem Werkzeug `ipcs(8)` einen Überblick über die Semaphores im System:

```
% ipcs -s
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
0x40021476 32769      graegerts  644        1
```

An dieser Stelle möchte ich darauf hinweisen, daß unser Demoprogramm sehr nachlässig mit den begrenzten Ressourcen umgeht. Besitzen wir keine Superuser-Rechte, kann es nach einiger Zeit vorkommen, daß Sie einen Anruf eines leicht angesäuerten Admins erhalten, der ihre zuvor angelegten IPC-Datenstrukturen manuell entfernen muß. Wir sollten stets darauf achten, am Ende des Programms die Ressourcen wieder freizugeben. □

Jetzt haben wir zwar ein Semaphore Set angefordert, dürfen es aber eigentlich noch nicht verwenden, denn zuvor müssen wir es initialisieren, was wir mit dem Multitalent `semctl(2)` erledigen können.

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

Rückgabewerte:

semid

Bezeichner des Semaphore Sets das wir verwalten wollen.

semnum

Nummer des Semaphores im Set das wir verwalten wollen.

cmd

Kommando, das wir auf das Semaphore **semnum** in **semid** anwenden möchten.

Die Sysnopsis deutet die mögliche Existenz eines weiteren Parameters an, der aber abhängig vom Kommando ist. Es handelt sich dabei um eine Union, welche die Applikation eigenständig erstellen muß und folgende Elemente aufweist:

```
union semun {
    int             val;      /* value for SETVAL */
    struct semid_ds *buf;    /* buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;   /* array for GETALL, SETALL */
} arg;
```

Die verfügbaren Kommandos für **cmd** werden im Folgenden aufgelistet.

GETVAL

Gibt den Wert von **semval** zurück. Erfordert Leserechte.

SETVAL

Setzt den Wert von `semval` zu dem von `arg.val` der `senum`-Union. Dabei wird das Feld `sem_ctime` der `semid_ds`-Struktur aktualisiert.

GETPID

Gibt den Wert von `sempid` der `sem`-Struktur zurück. Erfordert Leserechte.

GETNCNT

Gibt den Wert von `semncnt` der `sem`-Struktur. Erfordert Leserechte.

GETZCNT

Gibt den Wert von `semzcnt` der `sem`-Struktur. Erfordert Leserechte.

GETALL

Gibt den Wert von `semval` für jeden Semaphore in diesem Satz und speichert das im Feld `arg.array` mit `arg` als vierter Argument von `semctl(2)`.

SETALL

Setzt alle Semaphores im Semaphore Set auf den Wert im Feld `arg.array`, das als vierter Argument von `semctl(2)` übergeben wird. Erfordert Rechte zum Ändern.

IPC_STAT

Platziert den aktuellen Wert jedes Member der `semid_ds`-Struktur, die mit `semid` assoziiert ist in der Struktur, auf die `arg.buf` zeigt, wobei `arg` als vierter Argument von `semctl(2)` übergeben wird. Erfordert Leserechte.

IPC_SET

Setzt den Wert der folgenden Member der `semid_ds`-Struktur, die mit `semid` assoziiert ist, auf die Werte, welche in den Member der Struktur, auf die `arg.buf` zeigt, wobei `arg` als vierter Argument von `semctl(2)` übergeben wird. Nur die Member

```
sem_perm.uid
sem_perm.gid
sem_perm.mode
```

werden gesetzt. Das Kommando kann nur ausgeführt werden, wenn die effektive Benutzer-ID des Prozesses mit den Werten der Member `sem_perm.cuid` oder `sem_perm.uid` der `semid_ds`-Struktur übereinstimmt.

IPC_RMID

Entfernt das Semaphore mit der ID `semid` aus dem System und zerstört gleichzeitig die mit ihm verbundene Datenstruktur `semid_ds`. Das Kommando kann nur ausgeführt werden, wenn die effektive Benutzer-ID des Prozesses mit den Werten der Member `sem_perm.cuid` oder `sem_perm.uid` der `semid_ds`-Struktur übereinstimmt.

Die Initialisierung eines Semaphore Sets kann eine Hilfsfunktion übernehmen, um unnötige Schreibarbeit zu vermeiden. Das könnte etwa so aussehen:

```
int Sem_init(int sem_id, int sem_num, int sem_value) {
    union semun {
        int             val;
        struct semid_ds *buf;
        unsigned short  *array;
    } buf;

    buf.val = sem_value;
    return (semctl(sem_id, sem_num, SETVAL, buf));
}
```

Neben der Verwaltung von Semaphore Sets können wir auch bestimmte Operationen auf diese Anwenden. Das erledigen wir mit der Funktion `semop(2)`.

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

Rückgabewert: 0 bei Erfolg oder -1 bei Fehler.

semid

Identifiziert den Semaphore-Satz auf den die Operation angewendet werden soll.

sops

Ein Feld von Strukturen des Typs `sembuf`.

nsops

Anzahl der Elemente des `sops`-Feldes.

Die `sembuf`-Struktur weist folgendes Layout auf:

```
struct sembuf {
    short sem_num; /* number of the semaphore element */
    short sem_op;  /* particular element operation to be performed */
    short sem_flg; /* flags to specify options for the operation */
};
```

Die `semop(2)`-Funktion erkennt nur zwei Flags für `sembuf.sem_flg`: `SEM_UNDO` und `IPC_NOWAIT`. Die Bedeutung der beiden Flags steht im direkten Zusammenhang mit den Werten von `semval`. `SEM_UNDO` sorgt dafür, daß die Semaphore-Operation rückgängig gemacht wird, wenn der Prozess beendet wird. Nur wenn die in `sops` enthaltenen Operationen atomar und gleichzeitig durchgeführt werden können, kehrt die Funktion erfolgreich zurück. Es gibt genau drei Operationen, unterschieden durch -1, 0 und 1. Jede Operation wird auf das `sem_num`-te Element des Semaphore Sets angewendet.

Die Anwendung der Operationen auf Semaphore Sets unterliegt folgenden Regeln (frei nach SUS; für eine einfache Darstellung siehe weiter unten):

1. Wenn `sembuf.sem_op` einen negativen Wert aufweist und der aufrufende Prozess Rechte zum Ändern des Semaphore Sets besitzt, müssen folgende Punkte beachtet werden:
 - Wenn `sem.semval` größer oder gleich dem absoluten Wert von `sembuf.sem_op` ist, wird der absolute Wert von `sembuf.sem_op` von `sem.semval` subtrahiert. Ist `(sembuf.sem_flg & SEM_UNDO)` nicht null, wird der absolute Wert zum `sembuf.sem_undo.semadj`-Wert (*process undo count*) des jeweiligen Semaphores addiert.
 - Ist `sem.semval` kleiner als der absolute Wert von `sembuf.sem_op` und `(sembuf.sem_flg & IPC_NOWAIT)` ist nicht null, wird `semop(2)` sofort zurückkehren.
 - Ist `sem.semval` kleiner als der absolute Wert von `sembuf.sem_op` aber `(sembuf.sem_flg & IPC_NOWAIT)` null, wird `semop(2)` `semncnt` des über `semid` referenzierten Semaphores inkrementiert und die Ausführung der Funktion solange ausgesetzt, bis eine der folgenden Bedingungen zutrifft:
 - Der Wert von `sem.semval` wird größer oder gleich dem absoluten Wert von `sembuf.sem_op`. Wenn das passiert, wird `semncnt` des spezifizierten Semaphores dekrementiert, der absolute Wert von `sembuf.sem_op` von `sem.semval` subtrahiert. Wenn `(sembuf.sem_flg & SEM_UNDO)` nicht null ist, wird der absolute Wert von `sembuf.sem_op` zum Wert von `sem.undo.semadj` (*process undo count*) des spezifizierten Semaphores addiert.
 - Der Semaphore-Bezeichner (`semid`) auf den der aufrufende Thread wartet wird vom System entfernt. Wenn das passiert, wird `EIDRM` gesetzt und -1 zurückgegeben.
 - Der aufrufende Thread empfängt ein Signal, das abgefangen werden soll. Wenn das passiert, wird der Wert von `semncnt` des spezifizierten Semaphores dekrementiert. Der aufrufende Thread setzt die Ausführung fort, so wie es durch `sigaction(2)` definiert ist.

2. Wenn `sembuf.sem_op` ein positiver Integer ist und der aufrufende Prozess genügend Rechte für das Ändern des Semaphore Sets aufweist, wird der Wert von `sembuf.sem_op` dem von `sem.semval` hinzugefügt. Wenn auch (`sembuf.sem_flg & SEM_UNDO`) gesetzt ist wird der Wert von `sembuf.sem_op` von `sem_undo.semadj (process undo count)` des spezifizierten Semaphores subtrahiert.
3. Wenn `sembuf.sem_op` 0 ist und der aufrufende Prozess Leserechte besitzt, kann folgendes geschehen:
 - Wenn `sem.semval` 0 ist, kehrt `semop(2)` sofort zurück, denn es wird eine sog. *wait-for-zero*-Operation durchgeführt.
 - Wenn `sem.semval` nicht 0 ist, aber (`sembuf.sem_flg & IPC_NOWAIT`) gesetzt ist, schlägt `semop(2)` fehl und keine der Operationen wird durchgeführt.
 - Wenn `sem.semval` nicht 0, aber (`sembuf.sem_flg & IPC_NOWAIT`) 0 ist, wird `semop(2)` `semzcnt` des spezifizierten Semaphores inkrementieren und die Ausführung des aufrufenden Threads solange aussetzen, bis eine der folgenden Bedingungen zutrifft:
 - Der Wert von `sem.semval` wird 0, so daß der Wert von `semzcnt` des spezifizierten Semaphores inkrementiert wird.
 - Der Semaphore-Bezeichner (`semid`) auf den der aufrufende Thread wartet wird vom System entfernt. Wenn das passiert, wird `EIDRM` gesetzt und -1 zurückgegeben.
 - Der aufrufende Thread empfängt ein Signal, das abgefangen werden soll. Wenn das passiert, wird der Wert von `semzcnt` des spezifizierten Semaphores dekrementiert. Der aufrufende Thread setzt die Ausführung fort, so wie es durch `sigaction(2)` definiert ist.

Ungefähr so finden wir die Beschreibung zu `semop(2)` in der Single Unix Specification. Doch was bedeutet das im Klartext? Versuchen wir diese Erkenntnisse allgemein zusammen zu fassen:

- Ein positiver Integer von `sem_op` inkrementiert das Semaphore um diesen Wert.
- Ein negativer Integer von `sem_op` dekrementiert das Semaphore um diesen Wert. Ein Versuch das Semaphore auf einen Wert kleiner 0 zu setzen, führt dazu, daß der aufrufende Thread (oder Prozess) entweder blockiert wird, oder daß `semop(2)` fehl schlägt, je nachdem ob `IPC_NOWAIT` gesetzt ist oder nicht.
- Ist der Wert von `sem_op` 0, bedeutet es daß der aufrufende Thread (oder Prozess) auf dieses Semaphore wartet bis es 0 erreicht.

Die beiden Flags `IPC_NOWAIT` und `SEM_UNDO` haben folgende Bedeutung:

`IPC_NOWAIT`

Es kann für jede der drei Operationen im `sops`-Feld gesetzt werden. Sollte eine der Operationen nicht durchgeführt werden können, kehrt die Funktion sofort wieder zurück, wobei keine Änderungen am Semaphore durchgeführt werden. Die Funktion schlägt fehl, wenn sie versucht, einen Semaphore um mehr Schritte als den aktuellen Wert zu dekrementieren, oder wenn sie versucht einen Semaphore auf 0 zu testen, das nicht 0 ist.

`SEM_UNDO`

Erlaubt die Rücknahme von Operationen in diesem `sops`-Array, wenn der Prozess beendet wird.

Praktisch könnte das etwa so aussehen:

```
struct sembuf sem[3];
...
setsemop(sem, 0, -1, 0);           /* decrement element 0 (P, up) */
setsemop(sem, 1, 1, 0);           /* increment element 1 (V, down) */
setsemop(sem, 2, 0, IPC_NOWAIT);  /* wait for zero */
```

Das folgende Beispiel illustriert die Anwendung von `semop(2)` anhand einer Parent-Child-Beziehung.

Listing 10.24: Anwendung von `semop`

```

1  #include <sys/types.h>
2  #include <sys/sem.h>
3  #include "errors.h"
4
5  #define SEM_FLAG (IPC_CREAT | 0666)
6  #define KEY      (9876)
7
8  void setsembuf(struct sembuf *sem_buf, int sem_num, int sem_op, int sem_flg);
9
10 int main(int argc, char **argv) {
11     int             i, j, semid;
12     pid_t          pid;
13     int             nsems = 1;
14     int             nsops;
15     struct sembuf sops[2];
16
17     if ((semid = semget(KEY, nsems, SEM_FLAG)) < 0)
18         err_fatal("semget failed");
19     else
20         printf("semget OK: semid = %d\n", semid);
21
22     if ((pid = fork()) < 0)
23         err_fatal("fork failed");
24
25     if (pid == 0) { /* child code */
26         i = 0;
27
28         while (i < 3) {
29             nsops = 2;
30
31             setsembuf(&(sops[0]), 0, 0, SEM_UNDO); /* wait for zero */
32             setsembuf(&(sops[1]), 0, 1, SEM_UNDO | IPC_NOWAIT);
33
34             printf("Child ruft semop(%d, &sops, %d) auf:\n", semid, nsops);
35
36             for (j = 0; j < nsops; j++) {
37                 printf("\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
38                 printf("sem_op = %d, ", sops[j].sem_op);
39                 printf("sem_flg = %#o\n", sops[j].sem_flg);
40             }
41
42             if ((j = semop(semid, sops, nsops)) == -1) {
43                 err_fatal("semop failed");
44             } else {
45                 printf("\tsemop kehrt zurck: %d\n", j);
46                 printf("Child bernimmt Kontrolle: %d/3 mal\n", i + 1);
47
48                 sleep(5); /* DO Nothing for 5 seconds */
49                 nsops = 1;
50
51                 setsembuf(&(sops[0]), 0, -1, SEM_UNDO | IPC_NOWAIT);
52
53                 if ((j = semop(semid, sops, nsops)) == -1)
54                     err_fatal("semop failed");
55                 else
56                     printf("Child gibt Kontrolle auf: %d/3 mal\n", i + 1);
57

```

```

58             sleep(5); /* allow parent to catch semaphore change */
59         }
60         ++i;
61     }
62 } else { /* parent code */
63     i = 0;
64
65     while (i < 3) {
66         nsops = 2;
67
68         setsembuf(&(sops[0]), 0, 0, SEM_UNDO); /* wait for zero */
69         setsembuf(&(sops[1]), 0, 1, SEM_UNDO | IPC_NOWAIT);
70
71         printf("Parent ruft semop(%d, &sops, %d) auf:\n", semid, nsops);
72
73         for (j = 0; j < nsops; j++) {
74             printf("\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
75             printf("sem_op = %d, ", sops[j].sem_op);
76             printf("sem_flg = %#o\n", sops[j].sem_flg);
77         }
78
79         if ((j = semop(semid, sops, nsops)) == -1) {
80             err_fatal("semop: semop failed");
81         } else {
82             printf("\tsemop: semop kehrt zurck: %d\n", j);
83             printf("Parent bernimmt Kontrolle: %d/3 mal\n", i + 1);
84
85             sleep(5);
86             nsops = 1;
87
88             setsembuf(&(sops[0]), 0, -1, SEM_UNDO | IPC_NOWAIT);
89
90             if ((j = semop(semid, sops, nsops)) == -1)
91                 perror("semop: semop failed");
92             else
93                 printf("Parent bernimmt Kontrolle: %d/3 mal\n", i + 1);
94
95             sleep(5); /* allow child to catch semaphore change */
96         }
97         ++i;
98     }
99 }
100 }
101
102 void setsembuf(struct sembuf *sem_buf, int sem_num, int sem_op, int sem_flg) {
103     sem_buf->sem_num = (unsigned short)sem_num;
104     sem_buf->sem_op = (short)sem_op;
105     sem_buf->sem_flg = (short)sem_flg;
106
107     return;
108 }
```

Listing 10.24: xcode/semopdemo2.c - Beispielprogramm für die Anwendung von semop(2).

Nachdem ein wir das Semaphore angefordert haben, erzeugen wir einen neuen Prozess, so daß am Ende ein Parent und ein Child das Semaphore gemeinsam verwenden. Jeder Prozess führt jeweils die gleichen Operationen durch:

- er ruft das gleiche Semaphore desselben Sets ab (`sops.sem_num = 0`).
- er testet das Semaphore des Sets und übernimmt die Kontrolle, sobald es verfügbar ist. Das wird durch das Setzen des `sem_op`-Feldes auf 1 erreicht.

- hat er die Kontrolle übernommen, wird eine Meldung ausgegeben und ein paar Sekunden gewartet. An dieser Stelle würden wir kritische Operation durchführen anstatt einfach nur zu warten.
- Nun gibt der Prozess die Kontrolle ab, indem `sem_op` auf -1 gesetzt wird. Die Pause ist notwendig, um sicherzustellen, daß der andere Prozess das Semaphore abgreifen kann, bevor der gleiche Prozess es noch einmal ausliest.

□

Kapitel 11

POSIX-Threads

Men are generally idle, and ready to satisfy themselves, and intimidate the industry of others, by calling that impossible which is only difficult.

SAMUEL JOHNSON (1709 - 1784)

Parallelität ist in der Computertechnik einer der wichtigsten Techniken modernen Softwaredesigns. Es existieren unterschiedliche Ansätze, dieses Problem zu lösen, zum Beispiel Shared Memory oder Message Passing. Alternativ stehen uns auch Threads zur Verfügung. Dabei existieren mehrere Ausführungspfade in ein und demselben Adressraum eines Prozesses. In diesem Kapitel werden wir lernen wie Threads erzeugt und verwaltet werden und wie sie uns helfen, einfache Probleme effizient zu lösen.

11.1 Theoretische Grundlagen

Im Zusammenhang mit Threads wird auch oft von *Lightweight Processes* (dt. leichtgewichtige Prozesse) gesprochen, die eine spezielle Implementierungstechnik zur Herstellung einer Verbindung zwischen User Space und Kernel Space darstellen (im Abschnitt 11.2.2 *Thread-Scheduling* ab Seite 326 erläutern wir die Rolle der LWPs genauer). Herkömmliche Prozesse verfügen über einen Adressraum für das Process Image und kontrollieren einige Ressourcen (oder zumindest Teile) wie etwa Dateien oder Geräte. So ein Prozess läuft in der Regel immer nur entlang eines einzigen Ausführungspfades, der durch andere Prozesse unterbrochen werden kann. Aus diesem Grund ist jedem Prozess stets ein Ausführungszustand (*execution state*) und eine Priorität (*dispatching priority*) zugeordnet, damit der Scheduler des Kernels die Prozesse gerechter einordnen und den Ausführungszustand nach der Fortsetzung wieder herstellen kann.

Moderne Betriebssysteme betrachten beide Elemente als unabhängige Einheiten. Der Besitz von Ressourcen sowie der Zugriff auf Prozessoren, andere Ausführungspfade, Dateien und E/A-Geräten wird in der Regel in einer Struktur, die wir gemeinhin als *Prozess* oder *Task* bezeichnen zusammengefaßt. Innerhalb eines Prozesses sind einzelne Bestandteile, wie etwa der Zustand (Blocked, Running, usw.), der Thread-Kontext und Ausführungsstapel in Strukturen, die wir als Threads kennenlernen werden, enthalten. Diese bilden eine sog. *Dispatching Unit*. Alle Threads eines Prozesses teilen sich diese Bestandteile, denn wenn ein Thread ein bestimmtes Element im Speicher verändert, sehen alle anderen Threads dieses Prozesses die Änderung.

Richtig angewandt sind Threads dem Konzept traditioneller Prozesse überlegen, denn

- es wird weniger Zeit benötigt, einen Thread zu erzeugen, denn sie arbeiten im gleichen Adressraum wie der Prozess,
- ein Thread kann sehr schnell terminiert werden,
- das Umschalten zwischen einzelnen Threads des gleichen Prozesses geht sehr schnell,

- die Kommunikation zwischen Threads ist sehr effektiv, da im Vergleich zu Prozessen deutlich weniger Overhead notwendig ist, schließlich teilen sich Threads alle die gleichen Ressourcen.

Es wurde schon mehrfach angedeutet: wir unterscheiden stets zwischen zwei Modellen: *Single-Threading* und *Multi-Threading* (MT). Einige Betriebssysteme, wie etwa MS-DOS arbeiten nach dem Single-Threading-Modell. Das traditionelle UNIX System V kann zwar mit mehreren Prozessen arbeiten, jedoch nur ein Thread pro Prozess. Moderne Betriebssysteme, wie SunOS, IBM's AIX, HP-UX, Linux, die BSD-Derivate und viele andere können mit mehreren Threads pro Prozess umgehen und entsprechen dem heutigen Standard. Die Beziehung zwischen Prozessen und Threads wird in Abbildung 11.1 illustriert.

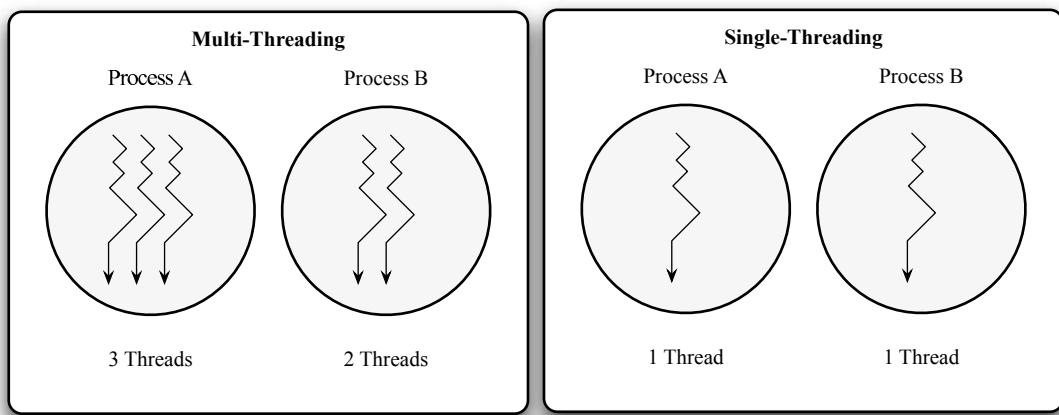


Abbildung 11.1: Beziehung zwischen Prozessen und Threads.

Für Entwickler stellt sich die Frage, warum sie auf Threads setzen sollten. Dazu einige Anregungen:

Programme reagieren besser

Viele Aktivitäten eines Programms, die nicht voneinander abhängig sind, können in Threads verwandelt werden, so daß sie nicht sequenziell, sondern parallel ablaufen können. Das ist insbesondere für grafische Benutzeroberungen (*Graphical User Interfaces*) sinnvoll, da die GUI-Elemente nicht auf eine bestimmte Aktivität warten müssen um auf neue Benutzereingaben reagieren zu können. Somit kann das GUI bereits neue Aktionen verarbeiten, so daß die Antwortzeiten verringert werden und nicht der Eindruck entsteht, daß die Anwendung eingefroren ist.

Mehrprozessorsysteme werden besser ausgenutzt

Anwendungen, die parallel arbeiten, wissen nicht, wie viele Prozessoren an der Arbeit beteiligt sind. Die Anwendung skaliert mit der Anzahl der Prozessoren. Numerische Algorithmen und Anwendungen mit einem hohen Maß an Parallelität, wie Matrizenberechnungen, laufen auf Mehrprozessorsystemen schneller, wenn sie mit Hilfe von Threads implementiert wurden.

Threads helfen die Programmstruktur zu verbessern

Viele Programme können verbessert werden, wenn unabhängige Teile in Threads aufgegliedert werden. Sie können viel leichter an neue Anforderungen angepaßt werden. Ein monolithischer Thread ist wesentlich unflexibler.

Threads benötigen weniger Systemressourcen

Wenn mehrere Prozesse auf die gleichen Ressourcen, beispielsweise mit Hilfe von Shared Memory, zugreifen ist das nicht besonders effektiv. Das liegt ganz einfach daran, daß jeder Prozess einen eigenen Adressraum besitzt und das Betriebssystem die Zustände, und damit auch die Ausführungspfade, aller Prozesse verwalten muß. Der Aufwand, diese Informationen zu verarbeiten kann durch Threads reduziert werden, denn sie erfordern viel weniger Overhead als Prozesse. Aus diesem Grund sprechen wir auch von *Lightweight Processes*.

Abbildung 11.2 verdeutlicht, welche Informationen der Prozess und der Thread verwalten müssen. Jeder Thread weist nur einen spezifischen *Thread Control Block* auf. Würden wir die gleiche Anwendung hingegen mit Prozessen implementieren, würden wir immer einen vollständigen Process Control Block, User Space Block und den jeweiligen Ausführungs-Thread erstellen, was mit einem unverhältnismäßigen Ressourcen-Aufwand verbunden wäre.

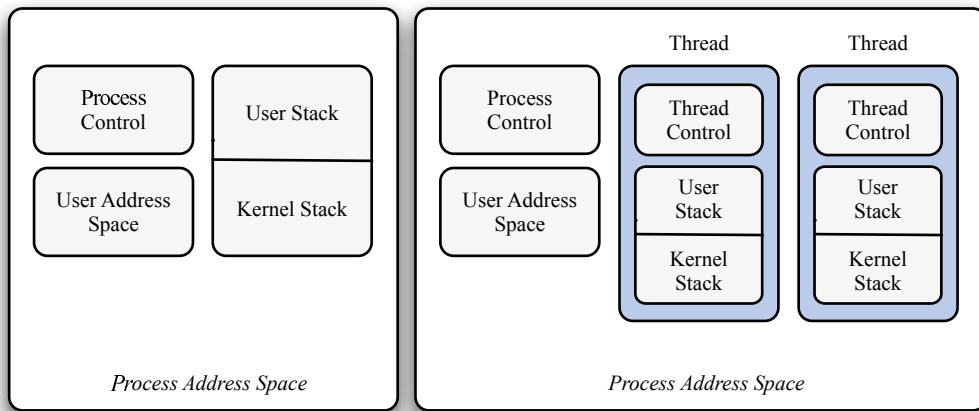


Abbildung 11.2: Single-Threading und Multi-Threading im Vergleich.

11.1.1 Thread-Implementierungen

Es gibt zwei Ebenen der Implementierung von Threads: auf Kernel-Ebene und auf User-Ebene. Erstere arbeiten im Kernel-Space und sind durch Systemaufrufe erreichbar, letztere im User-Space und basieren auf Thread-Bibliotheken. Beide Varianten haben Vor- und Nachteile die wir gleich zusammenfassen.

11.1.1.1 Kernel-Level Threads

Die gesamte Thread-Verwaltung findet im Kernel Space statt. Es existiert also keine externe Bibliothek für den Zugriff auf diese Einrichtung, wir können wohl aber über System Calls eingreifen. Der Kernel hält Kontextinformationen des Prozesses vor und das Umschalten zwischen den Threads wird durch den Scheduler des Kernels vorgenommen.

Vorteile

- Der Kernel kann mehrere Threads eines Prozesses für mehrere Prozessoren gleichzeitig einteilen (*scheduling*).
- Interne Kernel-Routinen können mit Threads implementiert werden.

Nachteile

- Die Umschaltung zwischen Threads innerhalb desselben Prozesses involviert immer den Kernel.

11.1.1.2 User-Level Threads

Auf dieser Ebene ist der Kernel nicht direkt involviert. Genau genommen weiß der Kernel gar nicht, wie viele Threads es momentan überhaupt gibt. Die gesamte Thread-Verwaltung findet im User Space durch die Thread-Bibliothek statt, aber, wie wir später sehen werden, gibt es eine Einrichtung im Kernel,

die eine Verknüpfung zwischen Kernel Space und User Space herstellt. Dennoch erfordert das Thread-Switching von User-Level Threads keine Eingriffe des Kernels, denn es treten keine Context Switches auf. Context Switches besprechen wir in Abschnitt 11.2.2.2 ab Seite 327.

Denken Sie aber immer daran, daß der Prozess nachwievor die Threads verwaltet und daß ein Systemaufruf durch einen Thread, den gesamten Prozess blockiert, aber die Thread-Bibliothek davon nicht Notiz nimmt. Daraus folgt, daß Thread-Zustände unabhängig von den Prozesszuständen verwaltet werden.

Vorteile

- Thread-Umschaltungen involvieren nicht den Kernel.
- Das Scheduling kann anwendungsspezifisch durchgeführt werden.
- User-Level Threads können quasi auf allen Plattformen implementiert werden.

Nachteile

- Systemaufrufe aus Threads heraus blockieren den gesamten Prozess und damit auch alle anderen Threads in diesem Prozess.
- Der Kernel kann Threads nicht auf mehrere Prozessoren verteilen. Aus diesem Grund können Threads des gleichen Prozesses nicht gleichzeitig auf mehrere Prozessoren ausgeführt werden.

11.2 Die POSIX-Threads Bibliothek

Die POSIX-Threads Bibliothek ist in sich geschlossen. Sie besteht aus etwa 100 Funktionsaufrufen und bildet die Grundlage für die Thread-Programmierung. Alles was nicht in dieser Bibliothek enthalten ist, wird entweder durch einen Systemaufruf oder eine Funktion der Standardbibliothek abgedeckt.

Beim Studium der Bibliothek wird Ihnen vielleicht auffallen, daß es nicht besonders viele Features enthalten sind, die Sie evtl. erwarten würden. Das ist auch beabsichtigt. Dinge wie Spin Locks, Mutexes mit Prioritätsvererbung, Deadlock-Recovery und so weiter können mit wenig Aufwand erstellt werden. Wir haben es hier mit einer minimalen Implementierung zu tun, die nur die nötigsten Funktionen bereitstellt, um maximale Leistung und höchste Portabilität zu erzielen. Alles was darüber hinaus geht, müssen wir selbst erstellen.

11.2.1 Der Lebenszyklus von Threads

Jedes Programm startet, indem seine `main`-Funktion aufgerufen wird. Wenn das Programm gegen die Threads-Bibliothek gelinkt ist, können wir ein oder mehrere Threads erzeugen, wie wir sie benötigen. Das einfachste wäre, die `pthread_create`-Funktion aufzurufen. Ihr übergeben wir eine Funktion, die unter Kontrolle des Threads ausgeführt werden soll. Zusätzlich dürfen wir auch Argumente übergeben, die für den Aufruf der Funktion notwendig sind.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void*), void *arg);
```

Rückgabewert: 0 bei Erfolg oder ungleich 0 bei Fehler.

`thread`

Speichert den Bezeichner (ID) des Threads an diesem Speicherplatz.

`attr`

Enthält die Attribute, die auf den neuen Thread angewendet werden sollen.

start_routine

Auszuführende Funktion des Threads (mit `arg` als Argument für diese Funktion).

arg

Argument für die in `start_routine` spezifizierte Funktion.

Der folgende Codeabschnitt zeigt, wie wir Threads erzeugen:

```
pthread_t      thread;
pthread_attr_t attr;

if (pthread_attr_init(&attr))
    err_fatal("pthread_attr_init failed");

if (pthread_create(&thread1, &attr, thread_func, NULL))
    err_fatal("pthread_create failed for thread_func1");
```

Bevor wir `pthread_create(3)` aufrufen können, müssen wir die Attribute des Threads initialisieren. Das erledigt die Funktion `pthread_attr_init(3)` für uns.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

Rückgabewert: 0 bei Erfolg oder ungleich 0 bei Fehler.

attr

Zeiger auf ein Attributobjekt, das wir initialisieren wollen.

Attributobjekte enthalten wichtige Informationen zur internen Verwaltung der Threads, wie beispielsweise Stack-Größen oder Prioritäten. Während letztere durchaus portierbar sind, müssen die Stack-Größen auf jeder Plattform angepaßt werden. Durch die Verwendung von Attributobjekten müssen die Anpassungen nur an einer Stelle der Thread-Bibliothek vorgenommen werden und nicht in den Implementierungen der Funktionen selbst.

Threads beenden wir auf die gleiche Weise, wie sie gestartet werden, nämlich durch Aufruf der `pthread_exit(3)`-Funktion. Im Gegensatz zu Prozessen, besteht zwischen Threads keine Beziehung. Das bedeutet, daß jeder Thread andere erzeugen kann, aber hinterher keine Parent-Child-Beziehung existiert.

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

Rückgabewert: Die Funktion kann nicht zurückkehren.

value_ptr

Rückgabewert des Threads. Dieser kann von anderen Threads mit Hilfe von `pthread_join(3)` abgefragt werden.

Jedem Thread ist ein Bezeichner (ID) zugeordnet, der zur Kontrolle diverser Aspekte der Thread-Verwaltung, wie etwa Scheduling, Signalbehandlung, etc., eingesetzt wird. Der Bezeichner ist vom Typ `pthread_t`, der als *opaque* (dt. unsichtbar) eingestuft ist¹.

Manchmal müssen wir auf einen bestimmten Thread warten. Insbesondere dann, wenn das Ergebnis der Verarbeitung dieses Threads für die weitere Ausführung des Programms notwendig ist. Dazu könnten

¹ Unsichtbare Datentypen sind für Anwender unsichtbar definiert und daher nicht ohne weiteres zu bestimmen. Aus diesem Grund kann es auf unterschiedlichen Plattformen zu Abweichungen des verwendeten Datentyps kommen, so daß wir mit Castings vorsichtig sein müssen.

wir die Funktion `pthread_join(3)` für alle gewünschten Threads aufrufen. Es sollte nur der *Controller*-Thread auf andere Threads warten. Ein Controller ist immer der Thread, der andere Threads (*Worker*) erzeugt; somit ist er auch für die Verwaltung dieser Thread verantwortlich. Abbildung 11.3 illustriert den Zusammenhang zwischen Controller und Worker Threads. Selbstverständlich können Worker Threads auch ihrerseits andere Threads erzeugen und müssen sie in ihrer Eigenschaft als Controller verwalten. Ein solches Szenario sehen wir in Beispiel 11.2.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

Rückgabewert: 0 bei Erfolg oder ungleich 0 bei Fehler.

thread

Thread auf dessen Terminierung wir warten möchten.

value_ptr

Zeiger auf einen Zeiger vom Typ `void` zur Ablage des Terminierungstatus von `thread`.

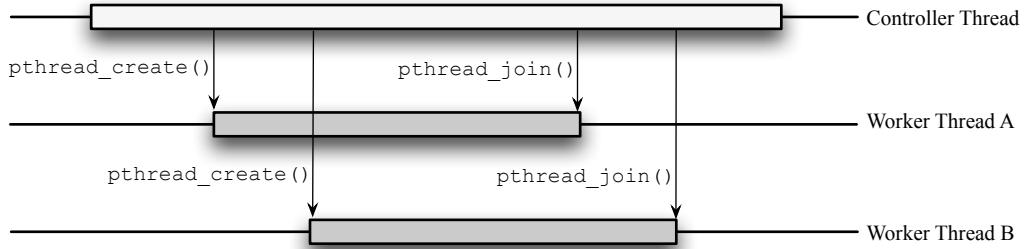


Abbildung 11.3: Beziehung zwischen Controller und Worker Thread.

Eine andere, bessere, Möglichkeit liegt im Einsatz einer der Synchronisierungsmechanismen, die wir in Abschnitt 11.2.3 *Thread-Synchronisierung* ab Seite 328 kennenlernen. Anstatt auf einen Thread zu warten, könnten wir auch den Status der Threads abfragen. Zur Vermeidung von Deadlocks ist es nicht wichtig, ob der wartende Thread zuerst `pthread_join(3)` aufruft oder der beendende Thread `pthread_exit(3)` zuerst aufruft. Wenn `start_routine` zurückkehrt entspricht das einem impliziten Aufruf von `pthread_exit(3)` mit dem Rückgabewert von `start_routine` als Exit-Code.

In Beispiel 11.1 fragen wir die ID des aktuellen Threads mit der Funktion `pthread_self(3)` ab. Sie liefert einen numerischen Wert zurück, der in dem Argument `thread` beim Aufruf von `pthread_create(3)` abgelegt wird, entspricht.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Rückgabewert: ID des aufrufenden Threads.

Schauen wir uns zunächst ein Beispiel an, um ein besseres Gefühl für die Anwendung der kennengelernten Funktionen zu bekommen.

Listing 11.1: Threads erzeugen, auf sie warten und den Exit-Status abfragen

```
1 #include <pthread.h>
2 #include "header.h"
```

```

3
4 void *thread_func(void *arg);
5 int mytime(void);
6
7 int time_zero;
8 pthread_t thread1, thread2, last_thread_id;
9 pthread_attr_t attr;
10
11 int main(int argc, char **argv) {
12     void *status;
13     time_zero = time(NULL);
14
15     if (pthread_attr_init(&attr))
16         err_fatal("pthread_attr_init failed");
17
18     printf("[%2d] Main thread started\n", mytime());
19
20     printf("[%2d] Starting thread 1...\n", mytime());
21     if (pthread_create(&thread1, &attr, thread_func, NULL))
22         err_fatal("pthread_create failed for thread_func1");
23
24     printf("[%2d] Starting thread 2...\n", mytime());
25     if (pthread_create(&thread2, &attr, thread_func, NULL))
26         err_fatal("pthread_create failed for thread_func1");
27
28     if (pthread_join(thread1, &status))
29         err_normal("pthread_join failed for thread1");
30
31     printf("[%2d] Thread 1 returned status %d\n", mytime(), (int)status);
32
33     if (pthread_join(thread2, &status))
34         err_normal("pthread_join failed for thread2");
35
36     return (0);
37 }
38
39 void *thread_func(void *arg) {
40     int i;
41     void *status;
42     pthread_t thread_id = pthread_self();
43
44     printf("[%2d] Thread %d started\n",
45            mytime(), (int)thread_id);
46
47     for (i = 0; i < 5; i++) {
48         printf("[%2d] value %d\tthread id %d\n",
49                mytime(), i, (int)thread_id);
50         sleep(1);
51     }
52
53     pthread_exit((void *)1);
54 }
55
56 /* returns time in seconds */
57 int mytime(void) {
58     return (time(NULL) - time_zero);
59 }
```

Listing 11.1: xcode/thread_create.c - Threads erzeugen, auf sie warten und den Exit-Status abfragen.

Übersetzung und Ausführung:

```
% make thread_create
cc -lpthread plib.c nlib.c thread_create.c -o bin/thread_create
% ./thread_create
```

Programme, die auf POSIX Threads setzen machen von der *libpthread*-Bibliothek Gebrauch, so daß wir gegen sie mit dem Schalter `-lpthread` linken müssen. Die drei globalen Objekte `thread1`, `thread2` und `attr` sind zentrale Bestandteile dieses kleinen Programms, daß nichts weiter tut, als zwei Threads zu erzeugen, die Integer-Werte auf `stdout` ausgeben. Wir unternehmen folgende Schritte, um Threads zu erzeugen und sie mit minimalem Aufwand zu verwalten:

1. Attributobjekt(e) initialisieren

Wir rufen die Funktion `pthread_attr_init(3)` auf, um ein oder mehrere Attributobjekte zu initialisieren, da sonst `pthread_create(3)` fehlschlagen würde. Ein Attributobjekt kann für mehrere Threads verwendet werden. Wird es nicht mehr benötigt, kann es mit `pthread_attr_destroy(3)` zerstört und anschließend wieder mit `pthread_attr_init(3)` initialisiert werden. Der POSIX-Standard legt nicht fest, wie sich `pthread_attr_init(3)` verhält, wenn es auf bereits initialisierte Attributobjekte angewendet wird.

2. Threads erzeugen

Wir rufen `pthread_create(3)` auf um einen neuen Thread zu erzeugen und gleichzeitig zu starten. Zwischen den beiden Vorgängen (Erzeugen und Starten des Threads) wird nicht unterschieden, da wir zwei Funktionsaufrufe benötigen würden, obwohl die zugrunde liegende Applikation diese Form der Synchronisierung möglicherweise nicht benötigt. Ein anderer Grund liegt in der Tatsache, daß ein weiterer Thread-Zustand (*Thread erzeugt, aber nicht gestartet*) eingeführt werden müßte, was die Fehlersuche erschweren und eine Aufwendige Anpassung der `pthread_xxx`-Funktionen nach sich ziehen würde.

3. Die Worker Threads verwalten

Nachdem die erzeugten Threads ihre Aufgaben erledigt haben, kehren die assoziierten Funktionen (die `start_routine`-Argumente) zurück. Auf diesen Moment können wir mit der Funktion `pthread_join(3)` warten. Würden wir nicht auf die beiden Threads (`thread1` und `thread2`) warten, so würde das Programm vorzeitig zurückkehren und zwar bevor einer der beiden Threads auch nur einen Teil seiner Aufgaben erledigt hat. Diesen Effekt können Sie direkt beobachten, wenn Sie die Zeilen 29 bis 35 auskommentieren und das Programm erneut ausführen.

Die Schleife, die einen länger andauernden Arbeitsablauf simulieren soll wird jeweils durch einen Aufruf von `sleep(3)` angehalten, um dem anderen Thread Gelgenheit zu bieten, seine Arbeit zu erledigen. Nun können wir zwar sehen, wie sich die beiden Threads abwechseln, doch letztlich greifen wir durch `sleep(1)` in den asynchronen Ablauf ein. Wenn wir `thread_func` nur geringfügig modifizieren, können wir uns davon überzeugen, daß die beiden Threads in der Tat zum Zuge kommen.

```
pthread_t last_thread_id;
...
int main(void) {....}
...
void *thread_func(void *arg) {
    ...
    for (i = 0; i < 5000000; i++) {
        if (thread_id != last_thread_id) {
            last_thread_id = thread_id;
            printf("%d ", (int)last_thread_id);
        }
    }
    ...
}
```

Die Ausgabe des modifizierten Programms könnte etwa so aussehen:

```
...
32771 16386 16386 32771 16386
...
```

Wir sehen, daß ein Thread zwei mal und der andere drei mal nach `stdout` schreibt. Würden wir das gleiche noch einmal versuchen, so kann das nun völlig anders aussehen:

```
...
16386 32771
...
```

Das System entscheidet selbstständig, wieviel Zeit ein Thread für seine Ausführung erhält. □

Jetzt haben wir so viel über das Thread-Management gesprochen und fragen uns: Sollen wir nun auf Threads warten oder nicht? Es mag zwar nicht offensichtlich sein, doch die Antwort lautet: fast nie. Warum ist das so? Stellen wir ein paar Überlegungen an: Warum sollten wir überhaupt daran interessiert sein, daß sich ein Thread beendet? Weil wir auf das Ergebnis warten, welches wir für einen anderen Thread benötigen. Durch das Zusammenführen (`pthread_join(3)`) dieses Threads nehmen wir implizit an, daß er mit seiner Aufgabe fertig ist, wenn er sich beendet. Das kann natürlich wahr sein, aber wäre eine Implementierung nicht sauberer, wenn wir nicht auf den Thread selbst warten, sondern die Synchronisationsobjekte verwenden, die uns POSIX-Threads bieten? Diese Einrichtungen lernen wir in Abschnitt 11.2.3 *Thread-Synchronisierung* kennen.

Das gleiche Argument trifft auch für die Rückgabe von Statusinformationen zu. Nicht der Thread gibt einen Status zurück, sondern die assoziierte Funktion, die bei der Thread-Erzeugung übergeben wurde. Dieser Status kann auch ohne `pthread_join(3)` in Erfahrung gebracht werden.

Es gibt Situationen in denen wir tatsächlich wissen wollen, daß der Thread beendet wurde, nämlich dann, wenn wir den Stack des Threads selbst allokiert haben und diesen Speicher freigeben müssen. Hier ist ein Aufruf von `pthread_join(3)` absolut notwendig.

Es sei angemerkt, daß viele Programme diesem Prinzip nicht folgen und dennoch sehr gut funktionieren. Wenn wir aber neue Programme erstellen, sollten wir über diese Punkte zumindest nachdenken.

Wenn ein Thread `exit(3)` aufruft, wird der gesamte Prozess beendet und all seine Ressourcen freigegeben. Tritt ein Fehler im Thread (also in der `start_routine`-Funktion) auf, so wird `pthread_exit(3)` implizit aufgerufen, so daß nur der betreffende Thread beendet wird. Das Prinzip kennen wir bereits von der `main`-Funktion (Abschnitt 7.3.1).

Viele POSIX-kompatible Implementierungen (dazu gehören auch Win32- und SunOS-Threads) weisen Funktionen auf, die Threads zwingen kann, ihre Ausführung für unbestimmte Zeit auszusetzen und hinterher wieder aufzunehmen. Sie wurden für den Einsatz von Debuggern, die vollständige Kontrolle über den Worker Thread benötigen, entworfen. Jede andere Anwendung wäre auch von geringem Nutzen, da ausgesetzte Threads Ressourcen halten könnten, die der Controller Thread evtl. benötigen könnte. Diese Funktionen sind nicht Bestandteil der POSIX-API, werden aber oftmals zusätzlich von den Herstellern hinzugefügt.

Wie aus den bisherigen Ausführungen hervorgegangen ist, kann ein Thread einen anden auffordern, sich zu beenden. Der POSIX-Standard spricht von *Cancellation* und viele Implementierungen sprechen *Thread Termination* oder *Killing a Thread*. Abbildung 11.4 verdeutlicht den Ablauf einer solchen Aktion.

Um einen Thread aufzufordern, sich zu beenden, verwenden wir `pthread_cancel(3)`.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Rückgabewert: 0 bei Erfolg oder ungleich 0 bei Fehler.

thread
Thread, den wir auffordern möchten, sich zu beenden.

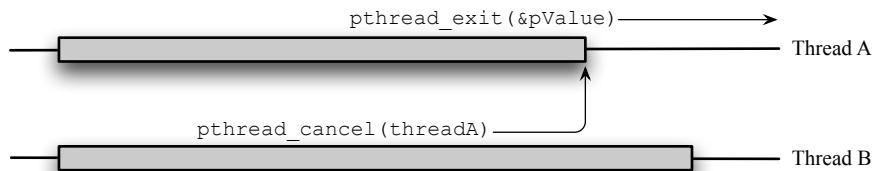


Abbildung 11.4: Thread Cancellation anhand zweier Threads.

Listing 11.2: Threads erzeugen Threads erzeugen Threads

Das folgende Beispiel zeigt, wie `pthread_create(3)`, `pthread_join(3)`, `pthread_cancel(3)` und `pthread_exit(3)` richtig eingesetzt werden.

```

1 #include <pthread.h>
2 #include "header.h"
3
4 pthread_t      thread1, thread2, thread3, thread_main;
5 pthread_attr_t thread_attr;
6 int            time_zero;
7
8 int main(int argc, char **argv) {
9     pthread_t tid;
10    time_zero = time(NULL);
11    thread_main = pthread_self();
12
13    if (pthread_attr_init(&thread_attr))
14        err_normal("pthread_attr failed; fallback to default");
15
16    printf("[%2d] Main\t\tThread [%s] started \n",
17           mytime(), thread_name(thread_main));
18    sleep(1);
19
20    Pthread_create(&thread1, &thread_attr, thread_func1, NULL);
21    printf("[%2d] Main\t\tThread [%s] created\n",
22           mytime(), thread_name(thread1));
23    sleep(5);
24
25    printf("[%2d] Main\t\tCancelling thread [%s]\n",
26           mytime(), thread_name(thread2));
27    pthread_cancel(thread2);
28    sleep(1);
29
30    printf("[%2d] Main\t\tThread [%s] exiting...\n",
31           mytime(), thread_name(thread_main));
32
33    pthread_exit((void *) NULL);
34 }
35
36 /* returns time in seconds */
37 int mytime() {
38     return (time(NULL) - time_zero);
39 }
40
41 void *thread_func1(void *arg) {
42     pthread_t thread_id = pthread_self();
43
44     printf("[%2d] thread_func1\tIn thread [%s] now\n",
45           mytime(), thread_name(thread_id));

```

```

46     sleep(1);
47
48     Pthread_create(&thread3, &thread_attr, thread_func3, NULL);
49     printf("[%2d] thread_func1\tThread [%s] created\n",
50            mytime(), thread_name(thread3));
51     sleep(1);
52
53     printf("[%2d] thread_func1\tThread [%s] is exiting...\n",
54            mytime(), thread_name(thread_id));
55     pthread_exit((void *)11);
56 }
57
58 void *thread_func2(void *arg) {
59     pthread_t thread_id = pthread_self();
60     void *status;
61
62     printf("[%2d] thread_func2\tIn thread [%s] now\n",
63            mytime(), thread_name(thread_id));
64     pthread_cleanup_push(cleanup, (void *)0);
65     pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
66     pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
67     sleep(1);
68
69     /* do some useful stuff */
70     int i;
71     printf("[%2d] thread_func2\tSome work in thread [%s]:",
72            mytime(), thread_name(thread_id));
73
74     for (i = 0; i < (int)arg; i++)
75         printf(" %d ", i);
76     fputc('\n', stdout);
77
78     printf("[%2d] thread_func2\tJoining thread [%s] now...\n",
79            mytime(), thread_name(thread3));
80     sleep(1);
81
82     if (pthread_join(thread3, &status))
83         err_normal("pthread_join() failed for thread [%s]",
84                    thread_name(thread3));
85
86     printf("[%2d] thread_func2\tThread [%s] returned status %d\n",
87            mytime(), thread_name(thread3), (int)status);
88     sleep(1);
89
90     /* thread 2 should get cancelled before this runs. */
91     printf("[%2d] thread_func2\tThread [%s] exiting...\n",
92            mytime(), thread_name(thread_id));
93
94     pthread_cleanup_pop(0);
95     pthread_exit((void *)22);
96 }
97
98 void *thread_func3(void *arg) {
99     void *status;
100    pthread_t thread_id = pthread_self();
101
102    printf("[%2d] thread_func3\tJoining thread [%s] now...\n",
103           mytime(), thread_name(thread1));
104    sleep(1);
105
106    if (pthread_join(thread1, &status))
107        err_normal("pthread_join failed for thread [%s]",
```

```

108         thread_name(thread1));
109
110     printf("[%2d] thread_func3\tThread [%s] returned status %d\n",
111            mytime(), thread_name(thread1), (int)status);
112     sleep(1);
113
114     Pthread_create(&thread2, &thread_attr, thread_func2, (void *)5);
115     printf("[%2d] thread_func3\tThread [%s] created\n",
116            mytime(), thread_name(thread2));
117     sleep(1);
118
119     printf("[%2d] thread_func3\tThread [%s] exiting...\n",
120            mytime(), thread_name(thread_id));
121
122     pthread_exit((void *)33);
123 }
124
125 /* simple wrapper for lazy people */
126 void Pthread_create(pthread_t *thread,
127                     pthread_attr_t *attr,
128                     void *(*start_routine)(void *), void *arg) {
129     int retval;
130
131     if (pthread_create(thread, attr, start_routine, arg))
132         err_fatal("Pthread_create failed\n");
133 }
134
135 void cleanup(void *arg) {
136     char *name = (char *)thread_name(pthread_self());
137
138     printf("[%2d] thread_func2\tThread [%s] cancelled! \n",
139            mytime(), name);
140 }
```

Listing 11.2: xcode/threaded_demo.c - Threads erzeugen Threads erzeugen Threads.

Da wir es hier mit einer etwas umfangreicheren Konstellation zu tun haben, soll uns Abbildung 11.5 bei der Analyse dieses Beispiels behilflich sein.

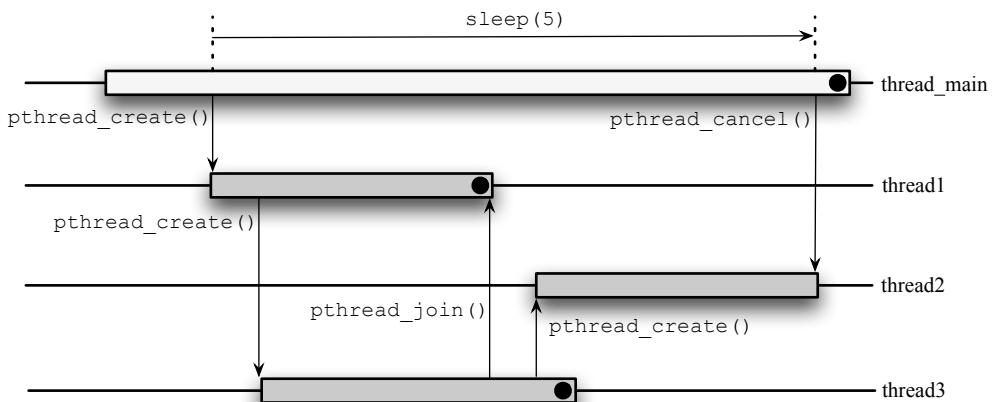


Abbildung 11.5: Grafische Darstellung von Beispiel 11.2.

Übersetzung und Ausführung:

```
% cc -o threaded_demo threaded_demo.c plibc.c -L -lpthread
```

```
% ./threaded-demo
[ 0] Main      Thread [Thread ID 0] started
[ 1] Thread 1  In thread [Thread ID 1] now
[ 1] Main      Thread [Thread ID 1] created
[ 2] Thread 3  Joining thread [Thread ID 1] now...
[ 2] Thread 1  Thread [Thread ID 2] created
[ 3] Thread 1  Thread [Thread ID 1] is exiting...
[ 3] Thread 3  Thread [Thread ID 1] returned status 11
[ 4] Thread 2  In thread [Thread ID 3] now
[ 4] Thread 3  Thread [Thread ID 3] created
[ 5] Thread 3  Thread [Thread ID 2] exiting...
[ 5] Thread 2  Some work in thread [Thread ID 3]: 0  1  2  3  4
[ 5] Thread 2  Joining thread [Thread ID 2] now...
[ 6] Main      Cancelling thread [Thread ID 3]
[ 6] Thread 2  Thread [Thread ID 3] cancelled!
[ 7] Main      Thread [Thread ID 0] exiting...
```

Aus Abbildung 11.5 geht hervor, daß Threads andere Threads erzeugen und auf sie warten. Während die Threads `thread_main`, `thread1` und `thread3` sich selbst durch einen Aufruf von `pthread_exit(3)` selbst beenden, wird `thread2` von `thread_main` aufgefordert, sich zu beenden. □

Zur besseren Lesbarkeit der Ausgabe verwenden wir eine kleine Hilfsfunktion: `thread_name()`.

```
#include "header.h"

char *thread_name(pthread_t tid)
```

Rückgabewert: Name des Threads passend zur Thread-ID.

`tid`
Thread-ID dessen Namen wir erfahren möchten.

Der Quelltext wird der Vollständigkeit halber abgebildet:

```
1  char *thread_name(pthread_t tid) {
2      char                     s[100];
3      thread_name_t           *n;
4      static int               tid_count = 0;
5      static pthread_mutex_t   lock = PTHREAD_MUTEX_INITIALIZER;
6      static thread_name_t   *thread_names = NULL;
7
8      if (pthread_equal(NULL_TID, tid))
9          tid = pthread_self();
10
11     pthread_mutex_lock(&lock);
12
13     for (n = thread_names; n != NULL; n = n->next) {
14         if (pthread_equal(n->tid, tid))
15             goto Label;
16     }
17
18     n = (thread_name_t *)malloc(sizeof(thread_name_t));
19     n->tid = tid;
20     sprintf(s, "Thread ID %d", tid_count);
21     tid_count++;
22     n->name = (char *)malloc(strlen(s) + 1);
23     strcpy(n->name, s);
24     n->next = thread_names;
25     thread_names = n;
26 Label:
```

```

27     pthread_mutex_unlock(&lock);
28     return(n->name);
29 }
```

Die Struktur `thread_name_t` ist in der Datei `<header.h>` definiert:

```

typedef struct thread {
    pthread_t      tid; /* thread ID */
    char           *name; /* thread description */
    struct thread *next; /* next thread in list of thread names */
} thread_name_t;
```

Die Funktionsweise ist schnell erklärt. Die Funktion verwaltet eine Liste von Threads und ihrer Namen. Existiert der Thread bereits in der Liste, so wird der Name, passend zur Thread-ID (`tid`) zurückgegeben, andernfalls wird der Name erzeugt und der Liste samt ID hinzugefügt. Auf diese Weise wird statt einer schnöden Zahl der String „Thread ID X“ (X ist ein positiver Integer beginnend mit 0) zurückgeliefert.

Da wir uns bereits mit der Erzeugung und der Terminierung von Threads beschäftigt haben, wollen wir uns nun auf die vier Funktionen `pthread_setcanceltype(3)`, `pthread_setcancelstate(3)`, `pthread_cleanup_push(3)` und `pthread_cleanup_pop(3)` konzentrieren.

Mit den ersten beiden Funktionen legen wir fest, wann und ob ein Thread asynchron beendet werden darf. Wenn wir den *Cancel State* eines Threads auf `PTHREAD_CANCEL_ENABLE` setzen erlauben wir anderen Threads ihn zu beenden. Im Gegenzug entziehen wir anderen Threads diese Möglichkeit mit dem Flag `PTHREAD_CANCEL_DISABLE`.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Rückgabewert: 0 bei Erfolg, anderer Wert bei Fehler.

state

Cancel State der für den aktuellen Thread gesetzt werden soll.

oldstate

Zeiger auf einen Speicherbereich zur Ablage des ursprünglichen Cancel States. Wenn wir den ursprünglichen Wert nicht festhalten wollen, dürfen wir auch `NULL` übergeben.

Der *Cancel Type* legt fest, wie ein Thread beendet werden kann. Wenn wir von einer asynchronen Terminierung eines Threads sprechen, heißt es, daß der betreffende Thread sofort bei Eintreffen der Anforderung (also einem Aufruf von `pthread_cancel(3)`) beendet werden soll. Das entspricht der Standardeinstellung und wird durch das Flag `PTHREAD_CANCEL_ASYNCHRONOUS` repräsentiert. `PTHREAD_CANCEL_DEFERRED` hat den gegenteiligen Effekt und verzögert die Terminierung des Threads bis zum nächsten so genannten *Cancellation Point*, was nichts weiter heißt als daß der Thread koordiniert durch `pthread_exit(3)` oder durch die Rückkehr der `start_routine()`-Funktion beendet wird.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Rückgabewert: 0 bei Erfolg, anderer Wert bei Fehler.

type

Cancel Type der für den aktuellen Thread gesetzt werden soll.

oldstate

Zeiger auf einen Speicherbereich zur Ablage des ursprünglichen Cancel Types. Wenn wir den ursprünglichen Wert nicht festhalten wollen, dürfen wir auch `NULL` übergeben.

Jeder neu erzeugte Thread weist den Cancel Type PTHREAD_CANCEL_ASYNCHRONOUS und den Cancel State PTHREAD_CANCEL_ENABLE auf. Ganz klar, daß wir die beiden Funktionen in Beispiel 11.2 nicht aufrufen müssen. Wenn wir aber den Cancel State auf PTHREAD_CANCEL_DEFERRED und den Cancel Type auf PTHREAD_CANCEL_DISABLE setzen

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

stellen wir folgenden Effekt fest:

```
... // bis zu diesem Zeitpunkt läuft alles normal
[ 5] Thread 2  Joining thread [Thread ID 2] now...
[ 6] Main      Cancelling thread [Thread ID 3]
[ 6] Thread 2  Thread [Thread ID 2] returned status 33 // das ist neu!
[ 7] Main      Thread [Thread ID 0] exiting...
[ 7] Thread 2  Thread [Thread ID 3] exiting...
```

Entscheident ist die zweite Zeile aus Sekunde 6, denn nun wir die Abbruchanfrage (*cancellation request*) ignoriert, so daß der Status der von `pthread_exit(3)` dieses Threads zurückgegeben wird. Kombinieren wir PTHREAD_CANCEL_DEFERRED mit PTHREAD_CANCEL_ENABLE so passiert wiederum etwas anderes: Der Cancellation Request wird so lange verzögert, bis der Thread (genauer gesagt die `start_routine`-Funktion) `pthread_exit(3)` aufruft, allerdings wird nicht der Status zurückgegeben, sondern die Ausführung lediglich bis zu diesem Cancellation Point fortgesetzt und dann abgebrochen, wie es für `pthread_cancel(3)` üblich ist.

11.2.2 Scheduling

Es gibt genau drei Varianten des Scheduling um Threads den Zugriff auf Kernel-Resourcen, und damit indirekt auch auf CPUs zu gewähren. Die Entwickler der Thread-Bibliotheken müssen sich in der Regel für eine der drei entscheiden. Zwei von ihnen setzen auf mehrere *Light Weight Processes* (LWPs):

- Mehrere Threads werden auf einen auf einen LWP umgesetzt (*many-to-one*)
- Jedem Thread ist ein LWP zugeordnet (*one-to-one*)
- Mehrere Threads werden auf mehrere LWPs umgesetzt (*many-to-many*)

Alle drei Modelle (Abbildung 11.6) haben ihre Existenzberechtigung und bringen unterschiedliche Vorteile und Nachteile mit. Es gibt viele Hersteller, die jeweils eines der drei vorgestellten Modelle bevorzugen.

Many-to-One-Modell

Die erste Technik ist als *Many-to-One* oder auch *Co-Routing* bekannt, bei dem mehrere Threads auf einen LWP geschaltet werden. Alle notwendigen Operationen finden im User Space statt und sind somit sehr schnell. Leider profitieren Mehrprozessorsysteme nicht von dieser Tatsache, da die Threads alle auf einen LWP geschaltet werden. Ein weiterer Nachteil liegt darin, daß der gesamte Prozess blockiert, wenn ein Thread einen blockierenden Systemaufruf durchführt.

One-to-One-Modell

Für jeden Thread wird ein LWP allokiert und auf ihn geschaltet. Damit skaliert das Modell auch auf Mehrprozessorsystemen sehr gut. Auch das Problem eines blockierten Threads des Many-to-One-Modells ist damit aus der Welt geschafft (siehe Abschnitt 5.7). Da das Erzeugen eines LWP immer mit einem Systemaufruf verbunden ist, entsteht ein kleiner Leistungsgenpass. Das gleiche trifft aus diesem Grund auch auf das Scheduling und die Synchronisierung zu.

Many-to-Many-Modell

Eine unbestimmte Anzahl von Threads wird auf eine unbestimmte Anzahl von LWPs geschaltet. Die Anzahl der LWPs kann kleiner oder gleich der Anzahl der Threads sein, abhängig von der Anzahl der laufenden Threads. Die Thread-Erzeugung findet, genau wie das Scheduling und die

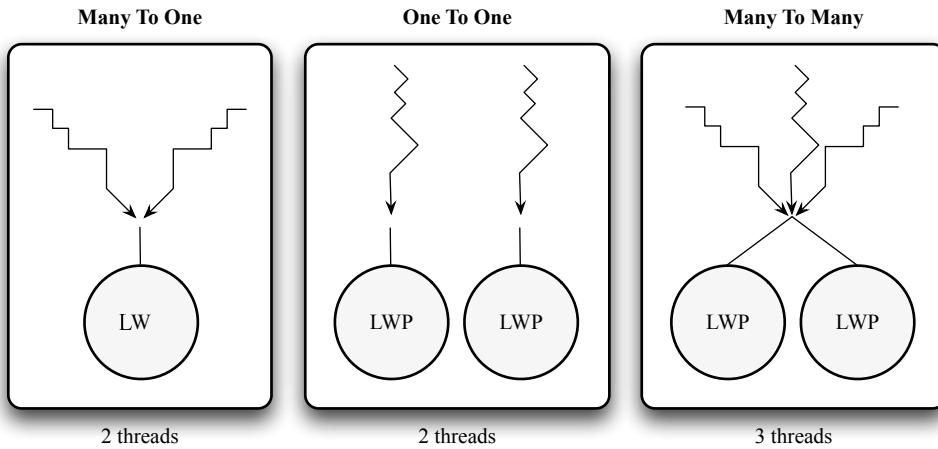


Abbildung 11.6: Drei unterschiedliche Scheduling-Techniken.

Synchronizierung, fast vollständig im User Space statt. Die Anzahl der LWPs kann für jede CPU eingestellt werden und daher als Tuning-Option für spezielle Anwendungen herhalten. Praktisch setzt kein Hersteller dieses Modell ein.

Interessanterweise existiert eine Mischform, die das One-to-One-Modell einbindet und als Two-Level-Modell bekannt ist. Damit können auf Anfrage individuelle Threads auf einen LWP geschaltet werden. Es hat sich herausgestellt, daß diese Variante wohl die beste Lösung ist und beispielsweise von Digital UNIX (ehemals OSF/1, heute Tru64Unix), IRIX und Solaris eingesetzt wird.

Ist ein Thread ein LWP oder ist ein LWP ein Thread?

Eingangs des Kapitels wurde erwähnt, daß Threads auch als *Lightweight Processes* (LPWs) bezeichnet werden können. Im Rahmen der Diskussion des Schedulings von Threads, scheint eine genauere Klärung dieses Begriffes notwendig zu sein.

Ein LWP kann als virtuelle CPU betrachtet werden, die für die Ausführung von Code zur Verfügung steht. Jeder LWP wird vom Kernel separat eingeteilt (*scheduled*) und kann unabhängig von anderen LPWs Systemaufrufe durchführen, aber auch jeweils eigene Fehler, wie etwa Seitenfehler (*page faults*), auslösen. Mehrere LWPs können desweiteren auf mehrere CPUs verteilt werden und parallel laufen. Den LWPs wird abhängig von ihrer *Scheduling Class* und *Priorität* jeweils einen Teil der verfügbaren CPU-Ressourcen zugeteilt. Da das Scheduling für jeden LWP einzeln vorgenommen wird, zeichnen sie jeweils eigene Kernel-Statistiken (User Time, System Time, Page Faults, etc.) auf. Daraus kann gefolgert werden, daß ein Prozess mit zwei LWPs doppelt so viel Rechenzeit erhält als ein Prozess mit nur einem LWP. Auch wenn das eine sehr stark vereinfachte Formulierung ist, wird hier deutlich, daß der Kernel nicht die Prozesse einteilt, sondern die LWPs.

Einige Eigenschaften von LWPs werden nicht direkt an Threads (aus Sicht der EntwicklerInnen) weitergegeben; dennoch können wir von ihnen profitieren, während alle Schnittstellen und Fähigkeiten weiter verwendet werden, indem wir den Threads beispielsweise mitteilen, daß sie über einen gewissen Zeitpunkt an die LWPs gebunden sind, was als *Contention Scope Scheduling* bezeichnet wird.

Fazit: LPWs stellen eine Implementierungstechnik dar, die Gleichzeitigkeit und Parallelität auf Kernel-Ebene erlaubt und durch die Thread-Schnittstellen im User Space reflektiert wird. Aus diesem Grund greifen wir nie direkt auf die LWPs zu, da es sich nur negativ auf die Portabilität der Software auswirken würde.

11.2.2.1 Thread-Scheduling

Wir können aus den letzten Ausführungen die Erkenntnis mitnehmen, daß zwei verschiedene Arten des Thread-Scheduling existieren: prozessbasiertes Scheduling und das globale Scheduling. Ersteres haben

wir als das Many-To-Many-Modell (*process contention scope scheduling*) und letzteres als One-To-One-Modell (*system contention scope scheduling*) kennengelernt. Beide Varianten sind namentlich nur in POSIX definiert.

Process Contention Scope Scheduling (PCSS) heißtt, daß das gesamte Scheduling im User Space stattfindet. Die Thread-Bibliothek hat die volle Kontrolle über alle Scheduling Mechanismen und entscheidet, wie ein Thread auf einen LWP geschaltet wird. Das setzt den Einsatz des One-To-One- oder Many-To-Many-Modell voraus. Beim *System Contention Scope Scheduling* (SCSS) entscheidet der Kernel, wie das Scheduling ausgeübt wird. Global verwaltete Threads weisen eine Priorität und eine Policy auf, die das kernel-basierte Scheduling weiter verfeinern.

In der Geschichte der Informatik hat sich das Thema Scheduling zu einem der unbequemsten entwickelt. Bisher konnte noch kein Mechanismus gefunden werden, der allen Anforderungen gerecht wird. Dabei spielt es keine Rolle, ob es um das Prozess-Scheduling oder Thread-Scheduling geht. Wir sehen uns in dieser Thematik zwei Situationen gegenüber, mit denen alle Probleme illustriert werden können. Im ersten Fall existieren zwei Threads, die beide völlig unabhängig voneinander arbeiten. Folglich nutzen beide ihre CPU-Zyklen optimal aus. Eine andere Situation stellt sich dar, wenn zwei Threads von den Ergebnissen des anderen abhängig sind. Das klassische Zeitscheibenmodell macht hier nicht sehr viel Sinn, da beispielsweise ein Thread sein Quantum nicht aufbrauchen kann, so lange er auf die Ausgaben des anderen wartet. Dynamisch zugewiesene Prioritäten wären hier besser geeignet.

PCS-Scheduling wird durch die Threads-Bibliothek vorgenommen in dem sie einen ungebundenen Thread auswählt und an einen LWP heftet.

Nachwievor ist das LWP-Scheduling eine globale Angelegenheit und Sache des Kernels. Es funktioniert völlig unabhängig vom lokalen Scheduling. Das mag befremdlich erscheinen, schließlich haben wir es hier offensichtlich mit einer Zwei-Ebenen-Struktur zu tun. Diese Tatsache kann aber vernachlässigt werden, denn wir müssen uns nur mit den lokalen Algorithmen des Schedulings beschäftigen.

Ein Thread im System Contention Scope (SCS) ist nichts weiter als ein Thread, der dauerhaft an einen LWP gebunden ist. Das bedeutet, daß der Thread immer läuft, wenn der LWP läuft und der LWP immer nur diesen spezifischen Thread betreibt. Dadurch kann der Thread aber niemals in den Zustand *Runnable* übergehen, denn entweder er wartet (*sleeping*), ist ausgesetzt (*suspended*) oder läuft (*active*). Sobald er Zeit anfordert, erhält er sie auch und übergeht dadurch den Zustand *Runnable*.

11.2.2.2 Context Switching

Ein Thread wird veranlaßt, den Kontext zu wechseln, wenn eines der folgenden Ereignisse eintrifft:

- **Aussetzung (*preemption*)**
Tritt ein Ereignis ein, das einen Thread mit hoher Priorität veranlaßt, in den Zustand *Runnable* (nur PCS) überzugehen, so wird der aktuell laufende Thread mit niedriger Priorität ausgesetzt und der andere Thread auf den LWP geschaltet. Auslöser könnte die Freigabe einer Sperre oder das Ändern der Prioritäten der beiden Threads sein.
- **Synchronisierung**
Der häufigste Grund für einen Context Switch ist der fehlgeschlagene Versuch, eine Sperre anzufordern. Wenn das nicht klappt, weil ein anderer Thread die Sperre bereits erfolgreich angefordert hat, wird der andere Thread schlafen geschickt und in die Warteschlange für diese Sperre eingereiht.
- **Zeitscheibenzuordnung**
Unterstützt das PCS eines Systems das Zeitscheibenmodell (z.B. Tru64Unix), kann ein Thread sein Quantum aufbrauchen, so daß automatisch ein anderer Thread an der Reihe ist.
- **Explizite Freigabe (*yielding*)**
Wir können den Thread zwingen, den Prozessor freizugeben, so daß er einen anderen Thread auswählen kann. Das erledigen wir mit einem Aufruf der Funktion `sched_yield` im Code des jeweiligen Threads.

Während das Yielding und die Synchronisierung vollständig im User Space ablaufen kann, erfordert das Aussetzen eines Threads einen Systemaufruf und damit auch mehr Overhead.

Der Zustand eines Systems manifestiert sich in den Daten der internen Prozessorspeicher, der Register und der MMU (*Memory Management Unit*). Bei jedem Context Switch wird der Zustand der CPU und des Prozesses gespeichert und bei Rückkehr in den anderen Kontext wieder hergestellt. Prozesse und Threads wechseln den Kontext auf diese Weise. Welche Rolle dabei LWPs spielen scheint unklar. Anders ausgedrückt: LWPs sind in diesem Prozess völlig unsichtbar.

PCS-Threads wechseln den Kontext, ohne den Kernel zu beanspruchen. Dennoch funktioniert die Prozedur wie gerade beschrieben. Muß ein Thread den Kontext wechseln, so wird der Scheduler aufgerufen, der die CPU auffordert, ihre Register in der Thread-Struktur des Threads zu speichern. Anschließend lädt er die Prozessstruktur des anderen Threads und kehrt zurück. Das ganze kann im User Mode geschehen und ist daher sehr schnell.

Zusammenfassung

Wir brauchen uns in der Regel keine Gedanken über das Scheduling machen, da viele Konzepte für die meisten Anwendungen überdimensioniert sind. In der Regel erledigt der Scheduler des Herstellers alles für uns und bedient sich dabei ausgefeilter Mechanismen. Wir haben uns mit dieser Thematik deshalb intensiv auseinandergesetzt, weil wir so die Natur der Threads und die Illusion der Gleichzeitigkeit besser verstehen können.

11.2.3 Thread-Synchronisierung

Die Gleichzeitigkeit der Threads bereitet uns Probleme, wenn mehrere Threads versuchen, auf die gleiche Datenstruktur zuzugreifen. Insbesondere dann, wenn ein Thread Daten schreibt und ein anderer liest. Es muß sichergestellt werden, daß beide Threads auf gültige Daten zugreifen, der Zugriff quasi synchronisiert wird. Das folgende Codebeispiel zeigt, warum wir Threads synchronisieren müssen:

```
struct some_data my_data = get_data(db_name, table_name);
... /* modify my_data */
if (insert_record(db_name, table_name, my_data) < 0)
    err_fatal("insert_record() failed");

if (commit_changes(db_name, table_name) < 0)
    err_fatal("commit_changes() failed");
```

Wir fordern einen Datensatz mit `get_data` an, modifizieren ihn, schreiben ihn in eine temporäre Tabelle (`insert_record`) und bestätigen anschließend unsere Änderungen mit `commit_changes`.

Wenn nun zwei Threads das gleiche Versuchen, kann schnell ein Teil der Information verloren gehen, denn was passiert wenn ein anderer Thread ebenfalls `get_data` aufruft und Daten modifiziert? Es gewinnt der Thread, der zuletzt die Daten schreibt. Die Informationen des anderen Threads sind für immer verloren.

Als wir in Kapitel 10 *Interprozesskommunikation* den Mechanismus der Semaphores (Seite 294) besprochen haben, stellten wir fest, daß wir eine Möglichkeit benötigen, bestimmte Operationen atomar, also ohne Unterbrechungen jeder Art durchführen zu können. Das erledigen zwei atomare Instruktionen in der Prozessorhardware, sogenannte *Set/Test Instructions*. Sie laden ein Datenwort in ein Register und setzen es auf einen Wert, beispielsweise 1. Das geschieht in einer einzigen Instruktion, die nicht unterbrochen werden kann. Hat das Datenwort den Wert 0, so kann es auf 1 gesetzt werden und der Thread ist *Besitzer* dieser Sperre. Andernfalls wird der Wert nicht verändert und der Besitzer bleibt Besitzer der Sperre.

Teile unseres Codes sind diesen Effekten ausgesetzt. Und zwar immer dann, wenn wir auf globale Datenstrukturen zugreifen. Diese Codeteile sind kritische Sektionen (Abschnitt 10.5.1 auf Seite 295) und müssen besonders geschützt werden. Das oben gezeigte Codebeispiel ist eine kritische Sektion. Ein effektiver Schutzmechanismus sind Sperren, die wie ein Pförtner den Zugang zu einer kritischen Sektion erlauben oder verhindern.

Die Synchronisierung wird in der Regel durch spezielle Datenstrukturen im Speicher realisiert. POSIX definiert drei Techniken, die sich solcher Strukturen bedienen. Zusätzlich steht uns noch `pthread_join` zur Verfügung, desse Anwendung wir bereits in Listing 11.2 kennen gelernt haben. Was Anwendungen mit mehreren Threads (MT-Anwendungen) angeht, wollen wir, im Gegensatz zu Single-Threaded Anwendungen gleich zwei Dinge erreichen: wir wollen unsere Daten schützen (das erledigen Sperren) und verhindern, daß ein Thread vom Scheduler ausgewählt wird, der eigentlich gar nichts zu tun hat (das machen wir mit Bedingungen, Semaphores und ähnlichem).

11.2.3.1 Mutual Exclusion Locks (*Mutexes*)

Die einfachste Möglichkeit, Zugriffe zu synchronisieren, liegt in der Verwendung von Mutexes. Sie lassen immer nur einen Besitzer einer kritischen Sektion zu. Wer zuerst kommt, besetzt den Mutex und nach Abarbeitung der kritischen Sektion wird es für den nächsten Thread freigegeben. Sobald ein Mutex besetzt ist, schlägt ein weiterer Versuch in die kritische Sektion vorzudringen fehl. Vor dem Eintritt in eine kritische Sektion rufen wir `pthread_mutex_lock` auf und wenn wir fertig sind, lösen wir die Sperre mit `pthread_mutex_unlock`. Versuchen Threads während der Verarbeitung der kritischen Sektion Zugriff darauf zu erlangen, werden sie schlafen geschickt und in eine Warteschlange aufgenommen, die nach dem FIFO-Prinzip abgearbeitet wird. Der nächste Thread in der Schlange wird aufgeweckt und kann erneut versuchen, die kritische Sektion zu verarbeiten.

Die kritische Sektion aus dem gezeigten Codeausschnitt würde mit Mutexes etwa so aussehen:

```

pthread_mutex_t      *mymutex;
struct some_data     my_data = get_data(db_name, table_name);

pthread_mutex_init(mymutex, NULL);
pthread_mutex_lock(&mymutex);

/* modify my_data */
if (insert_record(db_name, table_name, my_data) < 0)
    err_fatal("insertNewRecord() failed");

if (commit_changes(db_name, table_name) < 0)
    err_fatal("commit_changed() failed");

pthread_mutex_unlock(&mymutex);

```

Die Synopsis von `pthread_mutex_lock` und `pthread_mutex_unlock` lautet folgendermaßen:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

mutex

Zeiger auf ein Mutex vom Typ `pthread_mutex_t` dessen Besitzer wir werden wollen oder das wir lösen möchten.

Bevor wir ein Mutex verwenden können, müssen wir es initialisieren.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler.

mutex

Zeiger auf ein Mutex vom Typ `pthread_mutex_t` das wir initialisieren möchten.

attr

Attribute des Mutex. Um auf die Standardwerte zurückzugreifen, übergeben wir `NULL`.

Wenn uns die Standardattribute der Mutexes ausreichen, können wir das POSIX-Makro `PTHREAD_MUTEX_INITIALIZER` verwenden, um statische Mutexes zu initialisieren. Das Resultat ist das gleiche, als wären sie dynamische allokiert worden, mit dem Unterschied, daß keine Fehlerprüfung durchgeführt wird.

Der folgende Codeausschnitt illustriert die Anwendung der drei Funktionen:

```
static pthread_once_t my_once_mutex = PTHREAD_ONCE_INIT;
static pthread_mutex_t my_mutex;

void init_mutex() {
    pthread_mutex_init(&my_mutex, NULL);
}

void worker_method() {
    pthread_once(&my_once_mutex, init_mutex);
    pthread_mutex_lock(&my_mutex);
    /* perform some work */
    pthread_mutex_unlock(&my_mutex);
}
```

Die statische Variante läßt sich ganz ähnlich codieren:

```
static pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void worker_method() {
    pthread_mutex_lock(&my_mutex);
    /* perform some work */
    pthread_mutex_unlock(&my_mutex);
}
```

Der Typ `pthread_once_t` ist für die Einmalverwendung von Mutexes durch einen einzigen Thread vorgesehen und wird daher auch mit einem anderen Makro initialisiert. Passend dazu gibt es die Funktion `pthread_once`, die immer nur einmal während der Laufzeit einer Anwendung ausgeführt wird.

```
#include <pthread.h>

int pthread_once(pthread_once_t *once, void (*init_routine)(void));
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

once

Zeiger auf ein Mutex vom Typ `pthread_once_t` das wir initialisieren möchten.

init_routine

Zeiger auf eine Funktion, die zur einmaligen Initialisierung des Mutex aufgerufen werden soll.

Ein Mutex darf niemals von einem anderen Thread als dem, der den Mutex gesetzt hat, gelöst werden. Der Versuch, das zu tun, muß nicht unbedingt zu einem Fehler während der Laufzeit führen, muß aber in jedem Fall vermieden werden.

Unter bestimmten Umständen möchten wir verhindern, daß ein Thread schlafen geschickt wird, wenn der Versuch, eine kritischen Sektion zu betreten, nicht erfolgreich ist. In diesem Fall können wir die Funktion `pthread_mutex_trylock` verwenden, die entweder 0 bei Erfolg (wir sind jetzt Besitzer der Sperre) oder EBUSY bei Fehler zurückgibt.

Listing 11.3: Anwendung von Mutexes in MT-Applikationen.

Der folgende Code berechnet das Skalarprodukt², das häufig in der Vektorrechnung Anwendung findet. Die Daten sind in einer globalen Struktur abgelegt, wobei jeder Thread mit einem anderen Teil der Daten beschäftigt ist. Der Hauptthread wartet, bis alle Threads ihre Arbeit beendet haben und gibt das Ergebnis anschließend aus.

```

1 #include <pthread.h>
2 #include <malloc.h>
3 #include "header.h"
4
5 #define VECLEN    100
6
7 typedef struct {
8     double      *a;
9     double      *b;
10    double      sum;
11    int       veclen;
12 } data_t;
13
14 data_t          data;
15 pthread_t        *threads;
16 pthread_mutex_t my_mutex;
17
18 void *calc_skalar(void *arg) {
19     int i, start, end, len, offset = (int)arg;
20     double mysum = 0, *x, *y;
21
22     len = data.veclen;
23     start = offset * len;
24     end   = start + len;
25     x = data.a;
26     y = data.b;
27
28     for (i = start; i < end ; i++)
29         mysum += (x[i] * y[i]);
30
31     pthread_mutex_lock(&my_mutex);
32     data.sum += mysum;
33     pthread_mutex_unlock(&my_mutex);
34     pthread_exit((void*)0);
35 }
36
37 int main(int argc, char **argv) {
38     int          i, status, num_threads;
39     double      *a, *b;
40     char        *basename = basename_ex(argv[0]);
41     pthread_attr_t attr;
42
43     if (argc != 2)
44         err_fatal("Usage: %s <num-threads\n", basename);
45
46     if ((num_threads = atoi(argv[1])) <= 0)
```

²Das Skalarprodukt zweier Vektoren X und Y wird definiert durch: $X \cdot Y = |X||Y|\cos\theta$ mit θ als Winkel zwischen den beiden Vektoren. Das Skalarprodukt stellt die geometrische Interpretation der Länge der Projektion von X auf den Einheitsvektor \hat{Y} dar, wenn sich die beiden Vektoren schneiden.

```

47     err_fatal("Number of threads must be > 0\n");
48
49     threads = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
50
51     /* Assign storage and initialize values */
52     a = (double *)malloc(num_threads * VECLEN * sizeof(double));
53     b = (double *)malloc(num_threads * VECLEN * sizeof(double));
54
55     for (i = 0; i < VECLEN * num_threads; i++) {
56         a[i] = 1;
57         b[i] = a[i];
58     }
59
60     data.veclen = VECLEN;
61     data.a = a;
62     data.b = b;
63     data.sum = 0;
64
65     pthread_mutex_init(&my_mutex, NULL);
66
67     pthread_attr_init(&attr);
68     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
69
70     for(i = 0; i < num_threads; i++) {
71         printf("creating thread #%d\n", i);
72         pthread_create(&threads[i], &attr, calc_skalar, (void *)i);
73     }
74
75     pthread_attr_destroy(&attr);
76
77     for(i = 0; i < num_threads; i++) {
78         printf("waiting for thread #%d\n", i);
79         pthread_join(threads[i], (void **)&status);
80     }
81
82     printf("Sum = %f \n", data.sum);
83     free(a);
84     free(b);
85     pthread_mutex_destroy(&my_mutex);
86     pthread_exit(NULL);
87 }
```

Listing 11.3: xcode/thread_mutex1.c - Anwendung von Mutexes in MT-Applikationen.

Die Funktion `calc_scalar` wird bei Erstellung des Threads aktiviert. Alle Daten werden für die Berechnung aus der globalen Struktur `data_t` gelesen und die Ausgaben in die gleiche Struktur geschrieben. Wir übergeben dem Thread ein Argument, das die Nummer des Threads repräsentiert und gleichzeitig für die Berechnung des Skalarprodukts verwendet wird. Bevor das Ergebnis in die Struktur geschrieben wird sperren wir sie und lösen die Sperre hinterher. □

11.2.3.2 Semaphores

Anders als Mutexes erlauben Semaphores, mehrere Besitzer einer Resource, die mehrfach zur Verfügung steht. Wenn wir beispielsweise fünf Einheiten einer bestimmten Ressource haben, können wir fünf verschiedenen Threads zeitgleich Zugriff auf jede Einheit gewähren. Alle anderen Threads müssen warten, bis ein anderer seine Resource frei gibt.

Der Begriff *Semaphore* stammt aus dem Eisenbahnverkehr anfang des 19. Jahrhunderts. Damals waren Gleise teuer und Dampflokomotiven exotisch. Meist verfügten die Strecken nur über ein Gleis, so daß immer nur ein oder mehrere Züge in eine Richtung fahren konnten. Um anzugeben, daß bereits

ein Zug in einer bestimmten Richtung auf dem Gleis unterwegs war, wurden Semaphores installiert. Sie sahen aus wie kleine Metalfähnchen, die im 45 oder 90 Grad Winkel ausgerichtet wurden sobald ein Zug an ihnen vorbei gefahren ist. War ein Zug schon unterwegs, wußte der Zugführer anhand der Stellung der Semaphores, daß er warten muß.

Ein Semaphore ist im Grunde ein einfacher Zähler, der mit einem beliebigen Betrag größer 0 initialisiert werden kann. Immer wenn ein Thread versucht, Zugriff auf eine Ressource zu erlangen, wird der Wert dekrementiert, vorausgesetzt er hat nicht schon 0 erreicht. In diesem Fall schlägt der Versuch fehl. In Abschnitt 10.5 *POSIX-Semaphores* (ab Seite 294) haben wir Semaphores bereits im Rahmen der Interprozesskommunikation besprochen.

Im Gegensatz zu allen anderen Synchronisierungsmechanismen sind POSIX-Semaphores *async-safe*, d.h. wir dürfen sie ohne Bedenken in Signal Handlern aufrufen, um beispielsweise andere Threads aufzuwecken. Das ist mit anderen Techniken nicht gefahrlos möglich.

Wenn wir uns die Synopsis von `sem_wait` anschauen (Sektion 10.5.4) stellen wir fest, daß die Funktion -1 zurückgeben kann und `errno` auf `EINTR` setzt. Das bedeutet, daß `sem_wait` durch ein Signal unterbrochen wurde und der Zähler nicht erfolgreich dekrementiert werden konnte. Ganz klar, daß wir das in unserem Programm berücksichtigen müssen. Die Lösung ist, `sem_wait` in einer Schleife laufen zu lassen und den Rückgabewert zu testen. Ein einfacher Wrapper erledigt das für uns:

```
void sem_wait_ex(sem_t *semaphore) {
    while (sem_wait(semaphore) < 0, errno == EINTR) ;
}
```

Eine typische Anwendung von Semaphores ist das Produzent/Konsumentproblem aus Listing 11.4. Ein Thread holt ständig Daten ab und speichert sie in einer Liste, während ein anderer Thread die Elemente von der Liste abholt und verarbeitet. Eleganterweise wird die Anzahl der Nachrichten (Datensätze) in der Liste im Semaphore verwaltet, so daß die ProgrammiererInnen sich niemals mit diesem Wert auseinandersetzen müssen, denn die Aufrufe des Konsumenten blockieren, wenn keine Elemente mehr in der Liste enthalten sind. Ist das Semaphore mit einem Wert von fünf initialisiert worden, so kann der Produzent fünf Elemente auf einmal in die Liste schreiben und der Konsument genau fünf mal `sem_wait_ex` aufrufen. Der sechste Aufruf blockiert, bis eine weitere Nachricht in der Liste platziert wird.

```

1 mydata_t *get_data() {
2     mydata_t *data;
3     data = (mydata_t *) malloc(sizeof(mydata_t));
4     data->d_data = read_data_from_source();
5     return(data)
6 }
7
8 void process_data(mydata_t *data) {
9     process(data->d_data);
10    free(data);
11 }
12
13 produzent() {
14     mydata_t *data;
15
16     while (1) {
17         data = get_data();
18         add(data);
19         sem_post(&num_data);
20     }
21 }
22
23 konsument() {
24     mydata_t *data;
```

```

25
26     while (1) {
27         sem_wait_ex(&num_data);
28         data = remove();
29         process_data(data);
30     }
31 }
```

Listing 11.4: xcode/sem_pseudo1.c - Pseudocode für das Produzent/Konsumentproblem.

Intern wird Das Semaphore durch ein Mutex geschützt, da es sich hierbei um eine gemeinsam genutzte Datenstruktur handelt.

Das Folgende Beispiel zeigt, wie Threads und Semaphores zu Synchronisierung verwendet werden können.

Listing 11.5: Lösung des Dining Philosophers Problems mit Hilfe von Threads und Semaphores

In der Informatik wird gern das sog. Dining Philosophers Problem herangezogen, um den synchronisierten Zugriff auf begrenzte Ressourcen zu demonstrieren. Das Problem basiert auf folgenden Annahmen: n Philosophen sitzen an einem Runden Tisch und wechseln zwischen den beiden Aktionen Essen und Nachdenken. Möchte ein Philosoph essen, muß er zwei benachbarte Stäbchen, die jeweils links und rechts neben dem Teller liegen, aufnehmen, jedoch immer nur nacheinander, nicht beide gleichzeitig. Demnach versucht Philosoph n das Stäbchen i links und das Stäbchen $i + 1$ rechts neben den Teller aufzunehmen und zu essen. Das Problem ist aber, daß sich ein Philosoph das Stäbchen mit seinem Nachbarn teilen muß, denn es gibt genau so viele Stäbchen wie Teller. Er kann also nur essen, wenn die Philosophen $n - 1$ und $n + 1$ nicht essen, sondern gerade nachdenken.

```

1 #include <sys/types.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include "header.h"
5
6 #define NUM_PHILOSOPHERS 5
7
8 /* prototypes */
9 pthread_t create_thread(void *(*func)(void *), void *param);
10 void *philosopher(void *id);
11 int left(int i);
12 int right(int i);
13
14 sem_t chopsticks[NUM_PHILOSOPHERS]; /* one semaphore per chopstick */
15
16 int main(void) {
17     int ids[NUM_PHILOSOPHERS], i;
18
19     for (i = 0; i < NUM_PHILOSOPHERS; i++)
20         sem_init(&chopsticks[i], 0, 1);
21
22     pthread_t tids[NUM_PHILOSOPHERS];
23
24     for (i = 0; i < NUM_PHILOSOPHERS; i++) {
25         ids[i] = i;
26         tids[i] = create_thread(philosopher, &ids[i]);
27     }
28
29     for (i = 0; i < NUM_PHILOSOPHERS; i++)
30         pthread_join(tids[i], NULL);
31
32     return (0);
33 }
```

```

34
35 pthread_t create_thread(void *(*func)(void *), void *param) {
36     pthread_t tid;          /* thread ID */
37     pthread_attr_t attr;    /* attributes for new thread */
38
39     pthread_attr_init(&attr);
40     pthread_create(&tid, &attr, func, param);
41
42     return tid;
43 }
44
45 /*
46  * Return number of the chopstick to the
47  * right of philosopher i.
48 */
49 int right(int i) {
50     return (i + 1) % NUM_PHILOSOPHERS;
51 }
52
53 /*
54  * Return number of the chopstick to the
55  * left of philosopher i.
56 */
57 int left(int i) {
58     return i;
59 }
60
61 /* The philosopher eats and thinks. */
62 void *philosopher(void *id) {
63     int me = *(int *)id;
64
65     int i;
66     for (i = 0 ; i < 20 ; i++) {
67         /* pick up chopsticks */
68         printf("[%d] waiting to pick up chopstick %d\n", me, right(me));
69         sem_wait(&chopsticks[right(me)]);
70         printf("[%d] has picked up chopstick %d\n", me, right(me));
71
72         // sleep(1); /* uncomment to illustrate deadlock potential */
73
74         printf("[%d] waiting to pick up chopstick %d\n", me, left(me));
75         sem_wait(&chopsticks[left(me)]);
76         printf("[%d] has picked up chopstick %d\n", me, left(me));
77
78         /* eat */
79         printf("[%d] eating...%d\n", me, i);
80         sleep(rand() % 10);
81
82         /* put down chopsticks */
83         printf("[%d] puts down chopstick %d\n", me, right(me));
84         sem_post(&chopsticks[right(me)]);
85
86         printf("[%d] puts down chopstick %d\n", me, left(me));
87         sem_post(&chopsticks[left(me)]);
88
89         /* think */
90         printf("[%d] thinking...%d\n", me, i);
91         sleep(rand() % 10);
92     }
93
94     printf("[%d] philosopher complete\n", me);
95 }
```

Listing 11.5: xcode/sem_philosophers.c - Lösung des Dining Philosophers Problems mit Hilfe von Threads und Semaphores.

Jeder Philosoph erhält 20 mal die Gelegenheit, zu essen und nachzudenken. Diese Vorgänge dauern zwischen einer und zehn Sekunden (`sleep(rand() % 10)`). Dazu muß er aber zuerst das rechte und dann das linke Stäbchen (engl. *chopstick*) aufnehmen. Erst wenn jeder Philosoph genug gegessen und nachgedacht hat, wird das Programm beendet. Ohne die Verwendung von Semaphores wäre das Programm anfällig für Deadlocks, denn wenn jeder Philosoph versucht das rechts liegende Stäbchen aufzunehmen, wartet jeder auf seinen Nachbarn, so daß keiner der Philosophen jemals genug ist und nachdenkt. Die Semaphores erlauben den kontrollierten Zugriff auf die Stäbchen. □

11.2.3.3 Condition Variables

In manchen Fällen möchten wir Threads nur ausführen, wenn eine Bedingung erfüllt ist. Die Bedingung kann beliebig formuliert werden, so lange sie sich im Programm zu einem gültigen Ausdruck formen läßt.

Mutexes sind für einfache Synchronisierungen ausreichend, solange nur der gegenseitige Ausschluß erforderlich ist. Sind aber mehrere Teilnehmer an der Synchronisierung beteiligt eignen sich Bedingungen besser. Nehmen wir an, ein Produzent wartet wenn der Schreibpuffer voll und ein Konsument wenn der Lesepuffer leer ist. Ein solches Problem läßt sich mit *Condition Variables* hervorragend lösen.

Während Mutexes nur den Zugriff auf kritische Datenstrukturen koordinieren, geschieht das mit Condition Variables auf Basis der Werte in den Daten.

In der Informatik gibt es sogenannte *Monitors*, die ebenfalls für die Lösung des Prozent/Konsumentenproblem in Betracht kommen, allerdings Compiler-Unterstützung erfordern. Da POSIX-Threads als Bibliothek im User Mode implementiert wurde, wird das Prinzip mit Condition Variables implementiert.

Eine Bedingung wird durch den POSIX-Datentyp `pthread_cond_t` repräsentiert und mit Hilfe der beiden Funktionen `pthread_cond_init` und `pthread_cond_destroy` verwaltet.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.
```

cond

Zeiger auf eine Variable vom Typ `pthread_cond_t`, die wir initialisieren oder freigeben möchten.

attr

Attribute, die der Bedingung zugeordnet werden sollen. Reichen die Standardwerte aus, übergeben wir `NULL`.

Die beiden Funktionen lassen sich leicht einsetzen. Die Schnittstelle entspricht der anderer `pthread`-Funktionen:

```
pthread_cond_t my_condition;

pthread_cond_init(&my_condition, NULL);
/* do some work */
pthread_cond_destroy(&my_condition);
```

Und für statische Bedingungen gilt das gleiche wie für Mutexes. Sie lassen sich mit einem Makro initialisieren:

```
pthread_cond_t my_condition = PTHREAD_COND_INITIALIZER;
```

Die ganze Sache funktioniert etwa so:

1. Ein Thread sperrt einen Mutex (ist in diesem Moment Besitzer des Mutexes) und prüft die Bedingung unter dem Schutz des Mutexes. So ist sichergestellt, daß kein anderer Thread die Bedingung beeinflussen kann, solange er nicht Besitzer des Mutexes ist.
2. Ist die Bedingung wahr, führt der Thread die seine Aufgabe aus und löst die Sperre, wenn er fertig ist. Ist die Bedingung falsch, wird der Mutex automatisch gelöst und der Thread geht schlafen, während er auf die Bedingung wartet.
3. Wird die Condition Variable durch einen anderen Thread verändert, feuert er `pthread_cond_signal` und weckt den schlafenden Thread auf. Anschließend sperrt der Thread den Mutex erneut und prüft die Bedingung ein weiteres mal.

Wir sind gezwungen, die Bedingung jedes mal zu prüfen, wenn wir aufgeweckt werden, denn erstens könnte der andere Thread die Bedingung nicht geprüft haben, bevor er das Signal³ zum Aufwachen gesendet hat und zweitens könnte die Bedingungsvariable verändert worden sein, bevor der Thread nach Empfang des Signals zu laufen begann. Um alle schlafenden Threads aufzuwecken, können wir auch `pthread_cond_broadcast` verwenden.

Die Synopsis der beiden Funktionen lautet folgendermaßen:

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Rückgabewert: 0 bei Erfolg und errno bei Fehler.

cond

Condition Variable die mit den Threads assoziiert ist, die wir aufwecken möchten.

Die erste Funktion weckt immer nur den Thread auf, der zuerst beim Versuch, die Bedingung zu prüfen, schlafen geschickt wurde. Letztere weckt alle auf.

Listing 11.6 zeigt, wie wir Condition Variables einsetzen können um mehrere Threads zu synchronisieren.

Listing 11.6: Anwendung von Condition Variables

Die Applikation erzeugt zwei Threads, die jeweils eine Variable bis zu einem Grenzwert inkrementieren. Ein dritter Thread beobachtet die Variable und wartet bis der Grenzwert erreicht wurde. Einer der Worker-Threads signalisiert dem Beobachter, daß eine Änderung der Bedingung stattgefunden hat.

```
1 #include <pthread.h>
2 #include "header.h"
3
4 #define NUM_THREADS 3
5 #define TOTAL_COUNT 10
6 #define COUNT_LIMIT 12
7
8 void *inc(void *);
9 void *watch(void *);
10
11 int count = 0;
```

³Das „Signal“ ist hier nicht jenes, wie wir es von der Signalverarbeitung unter UNIX kennen, sondern eine Einrichtung der POSIX-Threads nur für diesen Zweck.

```

12 int threads[NUM_THREADS] = {0, 1, 2};
13 pthread_mutex_t my_mutex;
14 pthread_cond_t my_condition;
15
16 int main(int argc, char **argv) {
17     int i, rc;
18     pthread_t threads[NUM_THREADS];
19     pthread_attr_t attr;
20
21     /* initialize mutex and condition variable objects */
22     pthread_mutex_init(&my_mutex, NULL);
23     pthread_cond_init(&my_condition, NULL);
24     pthread_attr_init(&attr);
25
26     /* for maximum portability */
27     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
28
29     pthread_create(&threads[0], &attr, inc, (void *)&threads[0]);
30     pthread_create(&threads[1], &attr, inc, (void *)&threads[1]);
31     pthread_create(&threads[2], &attr, watch, (void *)&threads[2]);
32
33     /* wait for all threads to complete */
34     for (i = 0; i < NUM_THREADS; i++)
35         pthread_join(threads[i], NULL);
36
37     printf ("Main(): Waited on %d threads. Done.\n", NUM_THREADS);
38
39     /* clean up */
40     pthread_attr_destroy(&attr);
41     pthread_mutex_destroy(&my_mutex);
42     pthread_cond_destroy(&my_condition);
43     pthread_exit (NULL);
44 }
45
46 void *inc(void *idp) {
47     int j, i, *my_id = idp;
48     double result = 0.0;
49
50     for (i = 0; i < TOTAL_COUNT; i++) {
51         pthread_mutex_lock(&my_mutex);
52         count++;
53
54         if (count == COUNT_LIMIT) {
55             pthread_cond_signal(&my_condition);
56             printf("inc(): thread %d, count = %d --> threshold reached.\n",
57                   *my_id, count);
58         }
59         printf("inc(): thread %d, count = %d, unlocking mutex\n",
60               *my_id, count);
61         pthread_mutex_unlock(&my_mutex);
62
63         /* Do some work so threads can alternate on mutex lock */
64         for (j = 0; j < 1000; j++)
65             result += (double)random();
66     }
67
68     pthread_exit(NULL);
69 }
70
71 void *watch(void *idp) {
72     int *my_id = idp;
73

```

```

74     printf("Starting watch(): thread %d\n", *my_id);
75
76     /* protect access to counter */
77     pthread_mutex_lock(&my_mutex);
78     while (count < COUNT_LIMIT) {
79         pthread_cond_wait(&my_condition, &my_mutex);
80         printf("watch(): thread %d Condition signal received.\n", *my_id);
81     }
82
83     pthread_mutex_unlock(&my_mutex);
84     pthread_exit(NULL);
85 }
```

Listing 11.6: xcode/pthread_cond1.c - Anwendung von Condition Variables.

Die `main`-Funktion initialisiert zuerst alle benötigten Datenstrukturen und erzeugt drei Threads von denen zwei mit der Funktion `inc` und der dritte mit der Funktion `watch` gestartet werden.

Tritt ein Thread in `inc` ein, fordert er eine Sperre an, incrementiert die Variable, prüft ob die Bedingung erfüllt ist, und wenn ja, weckt er den dritten Thread auf, und wenn nicht, löst er die Sperre und gibt dem anderen Thread Gelegenheit, die Variable zu verändern.

Der Beobachter startet mit `watch_count` und sperrt zuerst die kritische Sektion. Solange die Bedingung nicht zutrifft geht er schlafen und wartet darauf, aufgeweckt zu werden. Wenn die Grenze `COUNT_LIMIT` erreicht wurde, bevor der wartende Thread in die Routine eintreten konnte, wird die Schleife übersprungen, um zu verhindern, daß `pthread_cond_wait` niemals zurückkehrt. □

11.2.3.4 Lese- und Schreibsperren

Nicht immer sind Mutexes sinnvoll. Manchmal arbeiten wir mit einer globalen Datenstruktur, die viel gelesen und wenig geschrieben wird. Bei jedem lesenden Zugriff eine Sperre anzufordern und hinterher wieder freizugeben ist unwirtschaftlich, denn was spricht gegen den lesenden Zugriff anderer Threads auf diese Struktur?

Lese- und Schreibsperren (*readers/writers locks*, RWLocks) erlauben den gleichzeitigen Lesezugriff aber Schreibzugriffe werden serialisiert.

RWLocks sind weniger performant als Mutexes, sind jedoch effektiver, wenn die zu sperrende Datenstruktur häufiger gelesen, als geschrieben wird. Es gibt zwar keine Regel für den Einsatz von RWLocks, doch bei mehreren hundert Elementen, die nur selten Schreibzugriffen unterliegen, sind RWLocks gegenüber Mutexes zu bevorzugen.

Das Prinzip der RWLocks funktioniert folgendermaßen: der erste Thread, der versucht zu lesen, erhält die Sperre. Alle nachfolgenden erhalten die Sperre ebenfalls und dürfen die Daten gleichzeitig lesen. Kommt ein schreibender Thread ins Spiel, wird er schlafen geschickt, bis alle Lese-Threads fertig sind. Alle weiteren Schreiber werden nach Prioritäten geordnet in die Warteschlange eingereiht. Die Leser werden immer hinter den Schreibern eingeordnet, so daß schreibende Threads immer Vorrang vor lesenden Threads haben.

Der erste Schritt vor Verwendung von RWLocks ist die Initialisierung des Attributs:

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

Rückgabewert: 0 bei Erfolg und errno bei Fehler.

`attr`

Zeiger auf ein Attribut vom Typ `pthread_rwlockattr_t`, das wir initialisieren möchten.

Wie immer ist das Ergebnis nicht definiert, wenn wir versuchen, ein bereits initialisiertes Attribut erneut zu initialisieren.

Ebenso einfach wie die Initialisierung ist auch die Freigabe eines RWLock-Attributs:

```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Rückgabewert: 0 bei Erfolg und errno bei Fehler.

attr

Zeiger auf ein Attribut vom Typ `pthread_rwlockattr_t`, das wir frei geben möchten.

Die Anwendung der beiden Funktionen folgt dem gleichen Schema wie es alle anderen Pthreads-Funktionen auch tun.

Das RWLock kann mit anderen Threads im gleichen Adressraum und mit Threads anderer Prozesse gemeinsam genutzt werden. Diese Eigenschaft wird im Attribut des RWLocks codiert und mit den beiden Funktionen `pthread_rwlockattr_getpshared` und `pthread_rwlockattr_setpshared` abgefragt oder gesetzt.

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(
    const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_setpshared(
    pthread_rwlockattr_t *attr, int pshared);
```

Rückgabewerte: 0 bei Erfolg und errno bei Fehler.

attr

Zeiger auf ein RWLock-Attribut vom Typ `pthread_rwlockattr_t`, dessen Gültigkeitsbereich des assoziierten RWLocks wir abfragen oder setzen wollen.

pshared

Flag, das den Gültigkeitsbereich des RWLocks angibt, bzw. Zeiger, der den Speicherort des Flags angibt.

Die folgenden beiden Flags sind für `pshared` gültig:

PTHREAD_PROCESS_SHARED

Erlaubt die gemeinsame Nutzung des RWLocks durch alle Threads, die gemeinsamen Zugriff auf den Speicherbereich haben, in dem das RWLock abgelegt wurde. Das gilt auch für Bereiche, die von mehreren Prozessen als Shared Memory genutzt werden.

PTHREAD_PROCESS_PRIVATE

Das RWLock wird nur von Threads des gleichen Prozesses verwendet. Standardmäßig werden alle RWLock-Attribute mit diesem Flag initialisiert.

Nachdem das RWLock konfiguriert wurde, könnten wir es verwenden. Insgesamt definiert POSIX ganze sieben Funktionen für den Umgang mit RWLocks. Tabelle 11.1 listet sie auf.

Natürlich muß ein RWLock wie ein Mutex initialisiert werden, bevor wir mit den anderen Funktionen sperren anfordern können.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
```

Funktion	Beschreibung
<code>pthread_rwlock_init</code>	Initialisierung eines RWLocks.
<code>pthread_rwlock_rdlock</code>	Lesesperre anfordern.
<code>pthread_rwlock_tryrdlock</code>	Lesesperre ohne Blockierung anfordern.
<code>pthread_rwlock_wrlock</code>	Schreibsperre anfordern.
<code>pthread_rwlock_trywrlock</code>	Schreibsperre ohne Blockierung anfordern.
<code>pthread_rwlock_unlock</code>	Schreib-/Lesesperre aufheben.
<code>pthread_rwlock_destroy</code>	Schreib-/Lesesperre freigeben.

Tabelle 11.1: POSIX-Funktionen zur Verwaltung und Anwendungen von RWLocks.

Rückgabewerte: 0 bei Erfolg und `errno` bei Fehler.

rwlockZeiger auf ein RWLock-Objekt vom Typ `pthread_rwlock_t`, das wir initialisieren wollen.**attr**Zeiger auf die mit dem RWLock assoziierten Attribute. Wenn uns die Standardattribute genügen, können wir auch `NULL` übergeben.

Erst einmal initialisiert, kann das RWLock beliebig oft wiederverwendet werden. Auch hier gilt: für statische RWLock-Attribute, die nur mit Standardeigenschaften ausgestattet werden sollen, kann das Makro `PTHREAD_RWLOCK_INITIALIZER` verwendet werden:

```
static pthread_rwlockattr_t lock_attr = PTHREAD_RWLOCK_INITIALIZER;
```

Ein Thread fordert eine Lesesperre mit der Funktion `pthread_rwlock_rdlock` an:

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

Rückgabewert: 0 bei Erfolg und `errno` bei Fehler.

rwlockZeiger auf ein RWLock-Objekt vom Typ `pthread_rwlock_t`, für das wir eine Lesesperre anfordern möchten.

Die Funktion blockiert, bis eine Lesesperre erfolgreich angefordert werden könnte. Das geht nur, wenn keine Schreibsperre durch einen anderen Thread vorliegt und auch keine anderen Threads auf eine Schreibsperre warten. In diesem Zusammenhang steht die Schwesterfunktion `pthread_rwlock_tryrdlock`, mit der wir prüfen können, ob wir in diesem Moment eine Schreibsperre anfordern könnten. Es macht wenig Sinn, sich auf das Ergebnis der Funktion zu verlassen, schließlich sagt es nur aus, daß wir zu diesem Zeitpunkt eine Sperre hätten anfordern können. Das kann sich aber in der Zwischenzeit schon wieder geändert haben. Folgendes Codesegment ist daher nicht korrekt:

```
pthread_rwlock_t      rwlock;

if (pthread_rwlock_init(&rwlock, NULL) != 0)
    err_fatal("pthread_rwlockattr_init() failed");

if (pthread_rwlock_tryrdlock(&rwlock) != EBUSY)
    if (pthread_rwlock_rdlock(&rwlock) != 0)
        err_fatal("pthread_rwlock_rdlock() failed");
else
    /* do some work */
```

Die beiden Vorgänge (Sperre prüfen und Sperre anfordern) können nicht atomar durchgeführt werden, so daß nicht sichergestellt ist, daß inzwischen ein anderer Thread eine Schreibsperre angefordert hat. Aus diesem Grund ist von der Verwendung von `pthread_rwlock_tryrdlock` abzuraten.

Eine Schreibsperre anzufordern funktioniert genau so, nur das wir die Funktion `pthread_rwlock_wrlock` bemühen müssen:

```
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

Rückgabewert: 0 bei Erfolg und errno bei Fehler

rwlock

Zeiger auf ein RWLock-Objekt vom Typ `pthread_rwlock_t`, für das wir eine Schreibsperre anfordern möchten.

Nur wenn kein anderer Thread eine Lese- oder Schreibsperre hält, kehrt die Funktion zurück. Andernfalls blockiert sie bis sie eine Sperre erhalten hat.

Wird dem Thread, der auf eine Schreibsperre eines RWLock-Objekts wartet, ein Signal zugestellt, wartet der Thread auch nach Rückkehr des Signal Handlers weiter auf die Sperre, ganz so als wäre der Vorgang nicht unterbrochen worden.

Ja, auch für Schreibsperren gilt: Abstand von `pthread_rwlock_trywrlock`.

Bevor wir auf ein Beispiel zusprechen kommen, schauen wir uns noch die Funktionen zur Lösung von Sperren und zur Freigabe der RWLock-Objekte an.

```
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Rückgabewerte: 0 bei Erfolg und errno bei Fehler.

rwlock

Zeiger auf ein RWLock-Objekt vom Typ `pthread_rwlock_t`, dessen Sperre wir lösen möchten.

Solange noch weitere Sperren für dieses Objekt vorliegen bleibt es im gesperrten Zustand. Es wird immer nur die zuletzt gesetzte Sperre gelöst. Warten ein oder mehrere Threads auf eine Schreibsperre für dieses Objekt, so entscheidet der Scheduler, welcher zuerst an der Reihe ist.

Wird das RWLock nicht mehr benötigt, kann es mit Hilfe der Funktion `pthread_rwlock_destroy` freigegeben und anschließend wiederverwendet (initialisiert) werden.

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Rückgabewert: 0 bei Erfolg und errno bei Fehler.

rwlock

Zeiger auf ein RWLock-Objekt vom Typ `pthread_rwlock_t`, das wir freigeben möchten.

Der POSIX-Standard schreibt nicht vor, wie Implementierungen vorgehen sollen, falls Threads noch Sperren auf diesem Objekt halten. Es kann davon ausgegangen werden, daß der Aufruf blockiert, bis die letzte Sperre gelöst wurde.

Listing 11.7: Anwendung von Schreib-/Lesesperren

```

1 #include "header.h"
2
3 void *user_thread(void *);
4 void *subtracter_thread(void *);
5 void *changer_thread(void *);
6
7 typedef struct {
8     int a, b, result, result2, count1, count2, max_use, max_use2;
9     pthread_rwlock_t rwl;
10 } app_data;
11
12 int main(void) {
13     pthread_t ct, ut, st;
14     app_data td = {5, 5, 0, 0, 0, 0, 10, 10};
15     void *retval;
16
17     if (pthread_rwlock_init(&td.rwl, NULL) != 0)
18         err_fatal("pthread_rwlock_init() failed");
19
20     if (pthread_create(&ut, NULL, user_thread, &td) != 0)
21         err_fatal("pthread_create() failed");
22
23     if (pthread_create(&ct, NULL, changer_thread, &td) != 0)
24         err_fatal("pthread_create() failed");
25
26     if (pthread_create(&st, NULL, subtracter_thread, &td) != 0)
27         err_fatal("pthread_create() failed");
28
29     if (pthread_join(st, &retval) != 0)
30         err_fatal("pthread_join() failed for st");
31
32     if (pthread_join(ct, &retval) != 0)
33         err_fatal("pthread_join() failed for ct");
34
35     if (pthread_join(ut, &retval) != 0)
36         err_fatal("pthread_join() failed for ut");
37
38     pthread_rwlock_destroy(&td.rwl);
39
40     printf("result should be %d, is %d\n",
41           td.max_use * (5 + 5), td.result);
42     printf("result2 should be %d, is %d\n",
43           -(td.max_use2 * (50 + 50)), td.result2);
44
45     return (0);
46 }
47
48 void *user_thread(void *data) {
49     int use = 0;
50     app_data *td = (app_data *)data;
51
52     while (use < td->max_use) {
53         if (pthread_rwlock_rdlock(&td->rwl) != 0)
54             err_fatal("pthread_rwlock_rdlock failed");
55
56         // Critical section code here
57
58         use++;
59
60         if (use == td->max_use)
61             pthread_rwlock_unlock(&td->rwl);
62
63     }
64 }
```

```

56     printf("[%2ld] user thread acquired read lock (%d)\n",
57            pthread_self(), use);
58
59     if (td->a == 5) {
60         td->result += (td->a + td->b);
61         td->count1++;
62         use++;
63     }
64
65     if (pthread_rwlock_unlock(&td->rwl) != 0)
66         err_fatal("pthread_rwlock_unlock failed()");
67
68     printf("[%2ld] user thread released read lock (%d)\n",
69            pthread_self(), use);
70     usleep(1);
71 }
72
73     return 0;
74 }
75
76 void *changer_thread(void *data) {
77     app_data *td = (app_data*)data;
78
79     while ((td->count1 + td->count2) <
80            (td->max_use + td->max_use2)) {
81         if (pthread_rwlock_wrlock(&td->rwl) != 0)
82             err_fatal("pthread_rwlock_wrlock() failed");
83
84         printf("[%2ld] changer thread acquired write lock\n",
85                pthread_self());
86
87         if (td->a == 5) {
88             td->a = 50;
89             td->b = td->a + usleep(1000);
90         } else {
91             td->a = 5;
92             td->b = td->a + usleep(1000);
93         }
94
95         if (pthread_rwlock_unlock(&td->rwl) != 0)
96             err_fatal("pthread_rwlock_unlock");
97
98         printf("[%2ld] changer thread released write lock\n",
99                pthread_self());
100        usleep(1);
101    }
102
103    return 0;
104 }
105
106 void *subtracter_thread(void *data) {
107     int use = 0;
108     app_data *td = (app_data*)data;
109
110     while(use < td->max_use2) {
111         if (pthread_rwlock_rdlock(&td->rwl) != 0)
112             err_fatal("pthread_rwlock_rdlock() failed");
113
114         printf("[%2ld] subtracter thread acquired read lock (%d)\n",
115                pthread_self(), use);
116
117         if (td->a == 50) {

```

```
118         td->result2 -= (td->a + td->b);
119         use++;
120         td->count2++;
121     }
122
123     if (pthread_rwlock_unlock(&td->rwl) != 0)
124         err_fatal("pthread_rwlock_unlock() failed");
125
126     printf("[%2ld] subtracter thread released write lock\n",
127            pthread_self());
128     usleep(1);
129 }
130 return 0;
131 }
```

Listing 11.7: xcode/thread_rwlock2.c - Anwendung von Schreib-/Lesesperren.

Teil II

Netzwerkprogrammierung

Kapitel 12

Der TCP/IP-Protokollstapel

We know next to nothing about virtually everything. It is not necessary to know the origin of the universe; it is necessary to want to know. Civilization depends not on any particular knowledge, but on the disposition to crave knowledge.

GEORGE F. WILL (1941 - ?)

Programme, die über ein Netzwerk miteinander kommunizieren möchten, müssen sich zuerst auf einen gemeinsamen Satz von Befehlen, ein einheitliches Datenformat und evtl. auch auf eine gemeinsame Schnittstelle einigen. Erst wenn diese Aspekte geklärt wurden, kann eine fehlerfreie Kommunikation stattfinden. Wir können uns diese Vereinbarungen wie die Einigung auf eine neutrale Sprache vorstellen, wenn zwei Menschen mit unterschiedlichen Muttersprachen miteinander kommunizieren. Die neutrale Sprache gibt Grammatik und Vokabular vor und als Schnittstelle könnten wir die Stimme benennen.

Ganz so einfach ist es in der Computertechnik dann doch nicht. Beispielsweise läuft ein Webserver im Hintergrund und wartet auf Anfragen. Sie bestehen aus Befehlen mit Zusatzinformationen, die er beantwortet, in dem er seinerseits einige für die Ausführung des Befehls notwendige Operationen durchführt und das Ergebnis, zum Beispiel in Form einer Website, an den Client (in der Regel ein Browser) übermittelt. Die Befehle (*requests*) und Antworten (*replies*) sind durch ein spezielles *Protokoll* definiert, daß den Rahmen der Kommunikation beschreibt.

Diese Form der Netzwerkkommunikation wird als *Client-Server*-Modell bezeichnet und ist, neben dem *Peer-to-Peer*-Modell, sicherlich das am weitesten verbreitete. Abbildung 12.1 illustriert die Kommunikation zwischen einem Client und einem Server. Der Server ist nicht auf die Bedienung eines Clients beschränkt. Je nach Anforderung kann ein Server bis zu mehrere hundert und gar tausend Clients bedienen. Wie das funktioniert, erfahren Sie in Abschnitt 13.2.2.4.

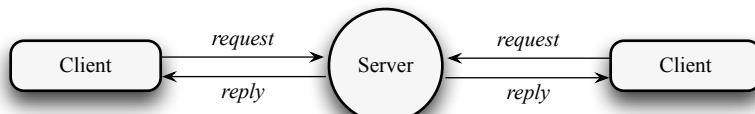


Abbildung 12.1: Client-Server-Kommunikation: zwei Clients, ein Server.

Client- und Server-Anwendungen kommunizieren also über ein vorgeschriebenes Protokoll; in unserem Fall HTTP (*hypertext transfer protocol*). Was wir nicht sehen können, aber in Kürze wissen werden, ist, daß eigentlich mehrere Protokolle auf unterschiedlichen Ebenen während der Kommunikation über

HTTP involviert sind. All diese Protokolle sind in dem TCP/IP-Protokollstapel zusammengefaßt. Dieser Protokollstapel wird Grundlage aller Besprechungen in diesem Teil des Buches sein und in Abschnitt 12.2 eingeführt.

Bleiben wir bei unserem Webserver-Beispiel und schauen wir uns an, wie die Kommunikation zwischen den einzelnen Protokollen des Stapels abläuft. Abbildung 12.2 zeigt einen Client auf der linken und einen Server auf der rechten Seite. Beide senden und empfangen Daten über HTTP.

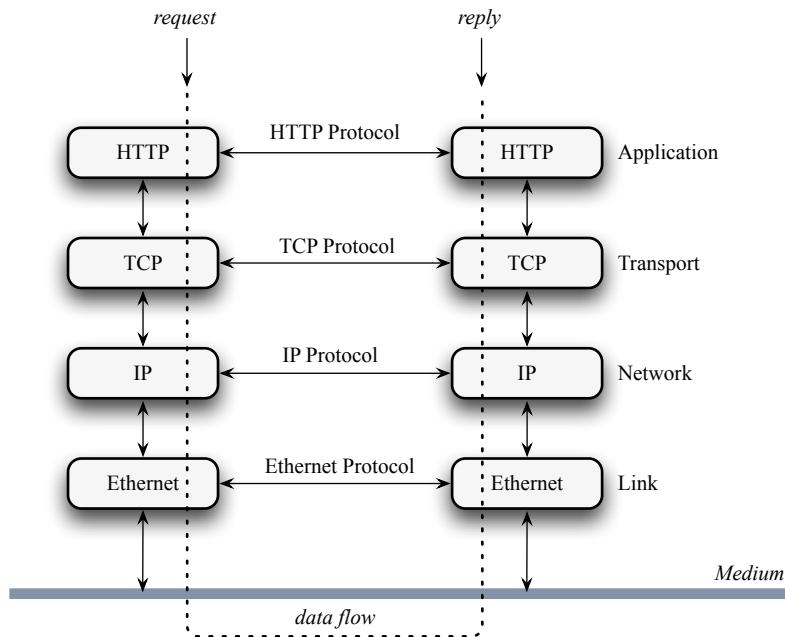


Abbildung 12.2: Client und Server kommunizieren über ein Protokoll, das auf andere Protokolle aufsetzt.

Da es sich bei HTTP um ein Anwendungsprotokoll handelt, befindet es sich an oberster Stelle im Stapel und stellt der Anwendung einen definierten Satz von Befehlen bereit, die ausschließlich für den Betrieb des Webservers und des Browsers notwendig sind. Die HTTP-Informationen werden an die nächste Schicht weitergegeben, die ihrerseits bestimmte Aufgaben hat. Auf diese Weise erreicht die eigentliche Informationen die letzte Schicht, die dafür sorgt, daß sie in Form von 0s und 1s auf das Medium gelangt. Jede Schicht kommuniziert immer nur mit der benachbarten und stellt jeweils ein eigenes Datenformat für die jeweilige Schicht bereit. So entsteht ein virtueller Kommunikationskanal zwischen Client und Server auf HTTP-Ebene, tatsächlich jedoch reicht eine Schicht die Informationen an die nächste weiter, so daß der reelle Kommunikationskanal alle Schichten umfaßt (dargestellt durch die gestrichelte Linie in Abbildung 12.2).

Jede Schicht kommuniziert mit seinem Gegenüber. So spricht das TCP-Protokoll des Clients beispielsweise mit dem TCP-Protokoll des Servers. Zwar haben wir zuvor davon gesprochen, daß die Schichten immer mit ihren Nachbarn sprechen, doch geschieht das eigentlich nur in Form von Daten, die einer bestimmten Art und Weise an die betreffende Schicht übergeben werden. Sie ist immer nur an einem bestimmten Teil der Daten interessiert und kann mit anderen nichts anfangen. Daten, die für das TCP-Protokoll wichtig sind, können von dem IP-Protokoll nicht verarbeitet werden. Aus der Sichtweise kommuniziert eine Schicht also immer nur mit der gleichen auf der anderen Seite der Anwendung.

Aufgaben und Eigenschaften der Schichten im TCP/IP-Protokollstapel:

Link Layer

Der Link Layer, manchmal auch *Data-Link Layer* genannt umfasst in der Regel die Netzwerkschnittstelle (NIC, *network interface card*) des Systems und den passenden Gerätetreiber. Beide zusammen kümmern sie sich um die Hardware-Details, die für das Einbringen der Daten auf das physikalische Übertragungsmedium notwendig sind.

Network Layer

Der Network Layer (auch *Internet Layer* genannt) bewegt die Dateneinheiten im Netzwerk. Hier findet auch das Routing statt. Die Funktionalität des Network Layers wird durch die Protokolle IP (*internet protocol*), ICMP (*internet control message protocol*) und IGMP (*internet group management protocol*) bereitgestellt.

Transport Layer

Der Transport Layer kümmert sich um den Fluss der Daten zwischen den Systemen und spielt dem Application Layer oberhalb des Transport Layers zu. Drei Transportprotokolle sind in der IP Suite enthalten: UDP (*user datagram protocol*), TCP (*transmission control protocol*) und SCTP (*stream control transmission protocol*). Alle drei haben ihre Vor- und Nachteile, die wir in Abschnitt 12.3 genauer kennenlernen werden. Die drei Protokolle sind nicht ohne weiteres austauschbar oder stellen einen Ersatz für das andere dar. Es sind spezialisierte Transportprotokolle, die sich nur für bestimmte Aufgaben eignen:

TCP stellt einen zuverlässigen verbindungsorientierten Transportmechanismus bereit. Es übernimmt die Aufteilung der Daten von der Applikation in passende Größen für den Network Layer, stellt Bestätigungen für empfangene Daten aus, beobachtet Timer, um sicher zu stellen, daß bestimmte Daten auch in einem angemessenen Zeitraum ausgeliefert wurden, und viele andere Dinge. Vorteil dieses Transportmechanismus liegt darin, daß sich die Applikation nicht um solche Dinge kümmern muß und eine zuverlässige Auslieferung garantiert ist. TCP besprechen wir in Abschnitt 12.3.2.

SCTP wurde für den Transport von Signalisierungsnachrichten über IP-Netzwerke konzipiert, kann aber auch für andere Anwendungen eingesetzt werden. SCTP ist ein zuverlässiges Transportprotokoll, daß auf ein unzuverlässiges paketvermittelndes Netzwerk, wie beispielsweise IP, aufsetzt. Obwohl es auf den ersten Blick Ähnlichkeiten mit TCP aufweist, sind die beiden Ansätze grundverschieden. SCTP besprechen wir in Abschnitt 12.3.3.

UDP ist ein deutlich vereinfachter Transportmechanismus für den Application Layer. Es übernimmt die Daten des Layers und sendet sie von einem System zum anderen. Nichts weiter. Es gibt keine Garantie, daß die Daten auch tatsächlich angekommen sind. In diesem Fall ist der Application Layer gefragt und muß diese Fehlerprüfung selbst durchführen. UDP besprechen wir in Abschnitt 12.3.1.

Application Layer

Der Application Layer ist nur für die Details der Anwendung selbst verantwortlich. Eines der vielen Anwendungsprotokolle haben wir bereits kennengelernt: HTTP. Weitere populäre Vertreter sind das Telnet-Protokoll für das (unsichere) Verbinden mit entfernten Systemen und das SMTP-Protokoll für das Senden von Emails.

Das in Abbildung 12.2 beschriebene Szenario ist recht einfach, denn Client und Server befinden sich in dem gleichen lokalen Netzwerk. Etwas anders sieht es aus, wenn wir über ein Weitverkehrsnetzwerk (WAN), beispielsweise dem Internet, kommunizieren müssen. In diesem Fall gelangen die Pakete nur zum Zielsystem, wenn die Daten über einen *Router* in den Teil des WANs geleitet werden, in dem sich das Zielsystem befindet.

12.1 Code-Konventionen

In den vorangegangenen Teilen haben wir gesehen, daß die Fehlerbehandlung stark vereinfacht wurde, indem wir uns nützlicher Hilfsfunktionen bedienten, die den Großteil der Arbeit kapseln. Solche Wrapper-Funktionen sind sehr nützlich und kommen in diesem Teil des Buches des öfteren zum Einsatz, allerdings auch aus anderer Motivation heraus, wie wir gleich sehen werden.

Betrachten wir eine Funktion, die ein binäres Adressformat in ein lesbare Format umwandelt: `inet_ntop(3)`. Sie eignet sich sehr gut für die Erstellung eines Wrappers und hat folgende Synopsis:

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
Rückgabewert: Zeiger auf die Zeichenkette oder NULL bei Fehler.
```

In der Praxis würden wir diese Funktion etwa so einsetzen:

```
char *pres; /* presentation */
struct sockaddr_in sockaddr; /* socket address structure */
...
if ((inet_ntop(AFINET, &sockaddr.sin_addr, pres, sizeof(pres)) == NULL)
    err_fatal("inet_ntop() failed");
...

```

Das Problem an dieser Funktion ist die Tatsache, daß wir ihr einen Zeiger auf eine `sin_addr`-Struktur (`src`) übergeben und, weil es sich um die binäre Darstellung handelt, auch die Adressfamilie (`af`) übergeben müssen. In Zeiten von IPv4 und IPv6 ergibt sich da ein Problem, denn unser Code wäre nicht portabel, in diesem Fall könnten wir nur IPv4 bedienen.

Die Wrapperfunktion mit dem Namen `inet_pton_ex` schaut sich die Struktur an und ruft dann entweder `inet_pton(3)` mit der Adressfamilie `AF_INET` für IPv4 oder `AF_INET6` für IPv6 auf.

Damit vereinfacht sich die Synopsis und lautet folgendermaßen:

```
#include "header.h"

char *inet_pton_ex(const sock_addr sockaddr, size_t socklen);
Rückgabewert: Zeiger auf die Zeichenkette oder null-Zeiger bei Fehler.
```

Auf die Einzelheiten der Funktion und was die einzelnen Datentypen bedeuten, kommen wir in kürze zu sprechen. Momentan interessiert uns vor allem die Notation der Wrapperfunktionen, denn sie erhalten alle ein Suffix `_ex`, das an den eigentlichen Funktionsnamen angehängt wird, um zu verdeutlichen, daß wir nicht die Bibliotheksfunktion, sondern eine unserer Wrapper aufrufen. Die Prototypen sind wie immer im Header `header.h` hinterlegt und in der Bibliothek `nlib.c` implementiert. Zur Übersetzung der Programme benötigen wir also stets den Header und die Datei `nlib.c`:

```
% cc -o myprogram myprogram.c lib/nlib.c
```

Wie ein bestimmtes Programm übersetzt und ausgeführt wird, ist dem jeweiligen Beispiel zu entnehmen.

12.2 Referenzmodelle: OSI und TCP/IP

Derzeit existieren zwei Referenzmodelle, die zur Beschreibung der Protokolle dienen. In vielen Lehrbüchern wird auf das Modell der ISO (International Standards Organization) zurückgegriffen und ist als OSI-Modell (Open Systems Interconnection) bekannt. Das zweite Modell ist die Internet Protocol Suite (IP). Beide sind vom Ansatz her vergleichbar, weisen aber in der Spezifizierung der Protokolle wesentliche Unterschiede auf. Abbildung 12.3 vergleicht beide Modelle.

Anwendungsspezifische Protokolldetails werden vollständig im User Space behandelt, während die kommunikationsspezifischen Details im Kernel implementiert sind. Für die Netzwerkprogrammierung sind die letzten beiden Schichten (*Layer*) des OSI-Modells nicht von Interesse.

Das Internet-Protokoll ist im Network-Layer angesiedelt. Uns stehen für den Datentransport zwei verschiedene Protokolle zur Verfügung: TCP und UDP, die wir beide in Abschnitt 13.2 kennenlernen. Anwendungen ist es durchaus erlaubt auf den Network-Layer zuzugreifen und IPv4 bzw. IPv6 direkt zu verwenden. In diesen Fällen sprechen wir von *Raw Sockets*.

Die Diskussionen dieses Teils befassen sich in der Regel mit der Kommunikation der Anwendungsschicht mit dem Transport-Layer.

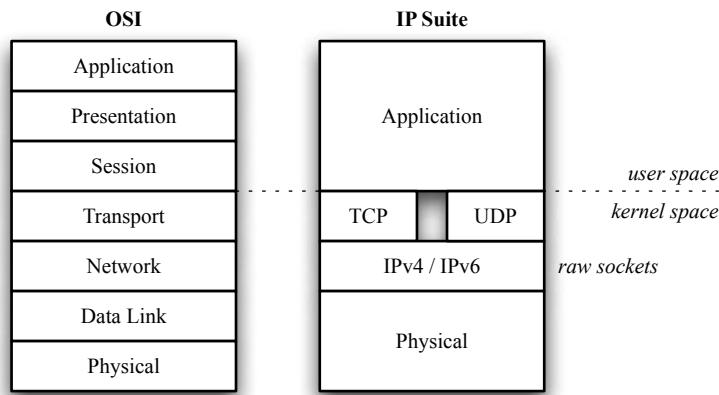


Abbildung 12.3: Das OSI-Modell und die IP Suite im Vergleich.

12.3 TCP, UDP und SCTP - Die Transportschicht

Werfen wir nun einen Blick auf die Protokolle des TCP/IP-Stacks, mit denen wir uns im Rahmen der Netzwerprogrammierung befassen werden. Nach Studium dieses Abschnittes sollten wir in der Lage sein, zu verstehen, wie Protokolle funktionieren und welches das kozessionelle Design dahinter steckt.

Unerwähnt blieb bisher SCTP (*stream control transmission protocol*), das noch relativ jung ist und hauptsächlich für den Einsatz in der IP-Telephonie vorgesehen ist. Da es den Umfang dieses Buches sprengen würde, werden wir auf SCTP nicht näher eingehen. Zentrale Bestandteile unserer Besprechungen sind die beiden traditionellen Transportprotokolle TCP und UDP. Sie setzen beide auf das Internet Protocol der Version 4 und Version 6 auf.

Während TCP ein zuverlässiges Protokoll ist, handelt es sich bei UDP um ein unzuverlässiges Protokoll. Um zu wissen, was wir in unserer Anwendung regeln müssen und was die Transportprotokolle für uns übernehmen bzw. nicht übernehmen, müssen wir uns zunächst mit den Eigenschaften und Fähigkeiten der Protokolle vertraut machen.

12.3.1 User Datagram Protocol (UDP)

UDP ist ein einfaches Transportprotokoll und wurde in RFC 768 definiert. Schreibt eine Anwendung Daten (genauer: eine Nachricht, *message*) in einen Socket, der sich ähnlich wie ein File Descriptor verhält, so werden sie in einem Datagram (Bezeichnung für eine Dateneinheit des UDP-Protokolls) eingeschlossen und an das Internet Protocol übergeben, das es in ein IP Datagram einschließt und an seinen Bestimmungsort befördert. Es gibt keinerlei Garantie, daß die Datagramme an ihren Bestimmungsort gelangen, daß sie in einer vorbestimmten Reihenfolge eintreffen oder daß es keine doppelten Zustellungen des gleichen Datagrams gibt.

Abbildung 12.4 zeigt ein von IP gekapseltes Datagram.

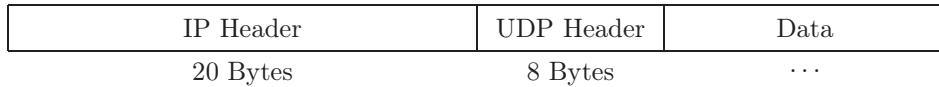


Abbildung 12.4: UDP-Datagram gekapselt in einem IP Datagram.

Der UDP-Header ist eines Datagrams ist recht übersichtlich und in Abbildung 12.5 dargestellt. Er enthält lediglich die Portnummern, eine Längenangabe und die obligatorische Prüfsumme.

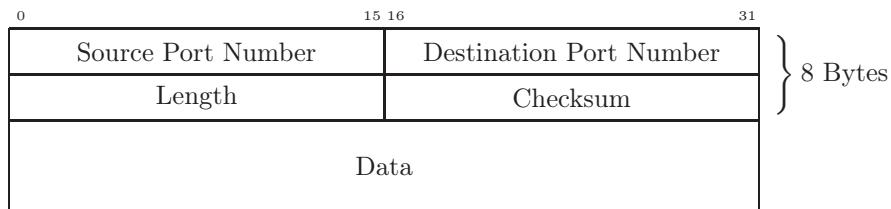


Abbildung 12.5: UDP-Header.

Port Number

Die Portnummern identifizieren den sendenden und empfangenden Prozess. Über sie werden die Daten der Internet-Schicht aufgeschlüsselt (demultiplexed). Da die IP-Schicht die Daten bereits durch einen Blick auf den IP-Header aufgeschlüsselt hat, ist UDP nur an UDP-Ports und TCP nur an TCP-Ports interessiert, so daß sie für beide Protokolle gleich sein können (genauere Informationen erhalten Sie im Abschnitt 12.3.1 *Portnummern und Sockets*).

Length

Die Längenangabe schließt sowohl den UDP-Header als auch den Datenbereich ein und wird immer in Bytes angegeben. Das kleinstmögliche Datagram kann 8 Bytes klein sein; es trägt keine Nutzdaten.

Checksum

Die Prüfsumme (*checksum*) wird auf Basis des UDP-Headers und der Nutzdaten gebildet.

Nachteil von UDP ist die geringe (faktisch nicht vorhandene) Zuverlässigkeit. Tritt beispielsweise ein Fehler bei der Prüfsummenbildung auf oder ist das Datagram im Netzwerk verloren gegangen, so wird das betreffende Datagram nicht automatisch neu übertragen. Wir würden noch nicht einmal mitbekommen, daß etwas schief gegangen ist. Für uns bedeutet es, daß wir solche Sicherheitseinrichtungen selbst in unsere Applikation integrieren müssen. Hilfreich ist dabei nur, daß UDP-Datagramme eine Länge aufweisen, die zusammen mit den Nutzdaten an die Anwendung übermittelt wird. Damit stehen sie im Gegensatz zum TCP-Protokoll, das sich als Byte-Stream darstellt und keine Abgrenzungen der Daten kennt, so genannte *Record Boundaries*.

Durch das Fehlen jeglicher Sicherheitsmechanismen wird das UDP-Protokoll auch als verbindungsloser Dienst (*connectionless service*) bezeichnet. Zwischen den Sockets des Clients und des Servers besteht keine Verbindung, denn der UDP-Header kann keine verbindungspezifischen Daten wie etwa Sequenznummern, speichern. Sie können höchstens in die Nutzlast eingebettet werden.

12.3.2 Transmission Control Protocol (TCP)

Dienste des TCP-Protokolls sind gegenüber dem UDP-Protokoll sehr unterschiedlich. Es wird in RFC 3390 spezifiziert, das die RFCs 793, 1323, 2581 und 2988 aktualisiert.

Abbildung 12.6 zeigt ein von IP gekapseltes TCP-Paket. Der TCP-Header ist 20 Bytes groß solange keine Optionen genutzt werden, die ebenfalls in den Header einfließen.

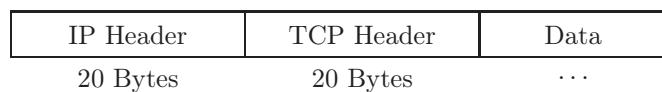


Abbildung 12.6: TCP-Paket gekapselt in einem IP Datagram.

Im Vergleich zu UDP weist der TCP-Header deutlich mehr Felder auf (Abbildung 12.7).

Die Bedeutung der Felder wird nur kurz angesprochen:

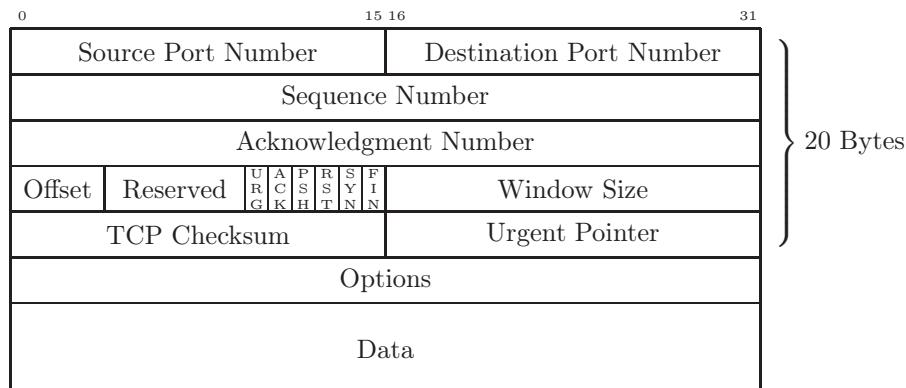


Abbildung 12.7: TCP-Header.

Port Numbers

Haben die gleiche Bedeutung wie für UDP (Abbildung 12.5).

Sequence Number

Die Sequenznummer identifiziert jedes Byte im TCP-Datenstrom.

Acknowledgment

Bestätigt den Eingang der Pakete, indem immer die Sequenznummer + 1 des letzten erhaltenen Bytes gesendet wird. Nur wenn das ACK-Flag gesetzt ist, wird dieses Feld ausgewertet.

Das Senden eines ACK kostet nichts, da es im Header sowieso immer enthalten ist. Aus diesem Grund ist das ACK-Flag immer gesetzt, sobald eine Verbindung erfolgreich hergestellt werden konnte.

Window Size

Die Fenstergröße zeigt an, wie groß der Empfangspuffer der Gegenstelle ist. Bei jedem Eingang von Daten wird das Fenster angepasst. Liegt der Empfänger Daten aus dem Buffer erhöht sich die Fenstergröße wieder.

Offset

Ohne Optionen beträgt die Gesamtlänge des Headers immer 20 Bytes.

Flags

Der TCP-Header kennt sechs Flags, von denen eines oder mehr gleichzeitig aktiv sein dürfen. Für Entwickler sind die meisten uninteressant.

URG (*urgent*) Zeigt an, daß sich Daten mit hoher Wichtigkeit im Datenstrom befinden.

ACK (*acknowledgement*) Gegenstelle validiert das Acknowledgment-Feld.

PSH (*push*) Empfänger soll die Daten so schnell wie möglich an die Anwendung weiterleiten.

RST (*reset*) Setzt die Verbindung zurück.

SYN (*synchronize*) Synchronisiere Sequenznummern zur Herstellung einer Verbindung.

FIN (*finish*) Der Sender ist mit der Datenübertragung fertig.

Checksum

Die Prüfsumme umfaßt sowohl Header als auch Nutzdaten und ist zwingend erforderlich. Der Sender speichert sie und der Empfänger prüft sie beim Empfang jedes Segments nach.

Urgent Pointer

Positiver Offset, der zur Sequenznummer des Segments addiert werden muß. Daraus ergibt sich die Sequenznummer des letzten Bytes der „dringenden Informationen“.

Options

Hier werden zusätzliche Optionen wie MSS (*maximum segment size*), Zeitstempel und andere Angaben hinterlegt.

TCP stellt eine Verbindung zwischen Clients und Servern her, über die Daten ausgetauscht werden können. Erst wenn die Übertragung der Daten abgeschlossen ist, wird die Verbindung beendet. Wenn TCP Daten versendet, wird für jede Dateneinheit (*packet*, dt. Paket) eine Bestätigung (*acknowledgment*) zurückgegeben. Bei Ausbleiben einer Bestätigung versucht TCP das betreffende Paket erneut zu übertragen bis es entweder erfolgreich zugestellt oder die maximale Anzahl von Wiederholungen erreicht wurde.

TCP kann nicht garantieren, daß ein Datenpaket auf der Gegenseite empfangen wird, aber es kann uns darüber informieren, daß es nicht eingetroffen ist. Ist die Gegenstelle nicht erreichbar, kann TCP natürlich nichts dagegen tun. Wenn wir also von *garantierter Zustellung* sprechen, kann nur gemeint sein, daß wir sowohl erfolgreiche als auch nicht erfolgreiche Zustellungen feststellen können. In sofern ist TCP zuverlässig.

Wenn TCP Daten in einen Socket schreibt, so werden sie in Segmente aufgeteilt, die dem IP Layer zugeführt werden. Jedes darin enthaltene Byte erhält eine Sequenznummer (*sequence number*), die es eindeutig identifiziert. Versuchen wir nun 1500 Bytes Daten in den Socket zu schreiben, teilt TCP sie in zwei Segmente auf. Das erste hat eine Größe von 1024 Bytes mit den Sequenznummern 1 bis 1024 und das zweite eine Größe von 476 Bytes mit den Sequenznummern 1025 bis 1500. Treffen nun die Segmente in unterschiedlicher Reihenfolge ein, so reorganisiert sie TCP und führt sie in der richtigen Reihenfolge der Applikation zu. Doppelte Segmente werden einfach verworfen.

Ein wichtiger Dienst des TCP-Protokolls ist die Flußkontrolle. Es kann der Applikation ganz genau mitteilen, wie viele Bytes tatsächlich übertragen wurden. Genauso kann eine Gegenstelle TCP mitteilen, wie viele Daten sie auf einmal empfangen kann. Dieses angezeigte Übertragungsfenster (*advertised window*) soll verhindern, daß der Sender den Empfänger überflutet, indem beispielsweise der Empfangspuffer überläuft. Während der Datenübertragung wird das Fenster angepaßt, denn sobald Daten empfangen wurden wird es kleiner, schließlich hat der Buffer nicht mehr die volle Kapazität. Natürlich liest der Empfänger die Daten nach und nach aus dem Buffer und schafft so wiederum Platz für neue Daten, so daß sich die Fenstergröße erhöht.

Das Empfangsfenster kann auch die Größe 0 erreichen, wenn der Empfangspuffer erschöpft ist. Der Sender wartet dann so lange, bis der Empfänger Daten aus dem Buffer gelesen hat und beginnt dann wieder mit der Übertragung. Im Gegensatz dazu bietet UDP keine Flußkontrolle an. Es ist daher sehr leicht, einen Empfänger mit Daten zu überfluten. Das passiert besonders dann, wenn der Sender schneller Arbeiten kann als der Empfänger.

Verbindungen auf- und abbauen

Als verbindungorientiertes, zuverlässiges Protokoll muß der Verbindungsaufbau zwischen Client und Server ebenso sicher sein, wie die Datenübertragung selbst. Der *Three-Way Handshake* hat sich als probates Mittel erwiesen und wird bei jedem Verbindungsaufbau mit TCP durchgeführt.

Folgende Schritte werden beim Three-Way Handshake durchlaufen:

1. Zuerst muß sich der Server auf eingehende Verbindungen vorbereiten. Meist geschieht das durch den Aufruf der Funktionen `socket`, `bind` und `listen` in dieser Reihenfolge. Wir sprechen hier von einem *Passive Open*.
2. Erst jetzt kann der Client versuchen, sich über die Funktion `connect` mit dem Server zu verbinden. Die Funktion `connect` sendet ein SYN-Segment (*Active Open*) mit der initialen Sequenznummer, ab der die Zählung beginnen soll.
3. Der Server seinerseits bestätigt den Empfang des SYN-Segments (durch ein ACK) und sendet sein eigenes SYN an den Client um seine Sequenznummer für die zu sendenden Daten zu übermitteln. Beides wird mit einem Segment erledigt.

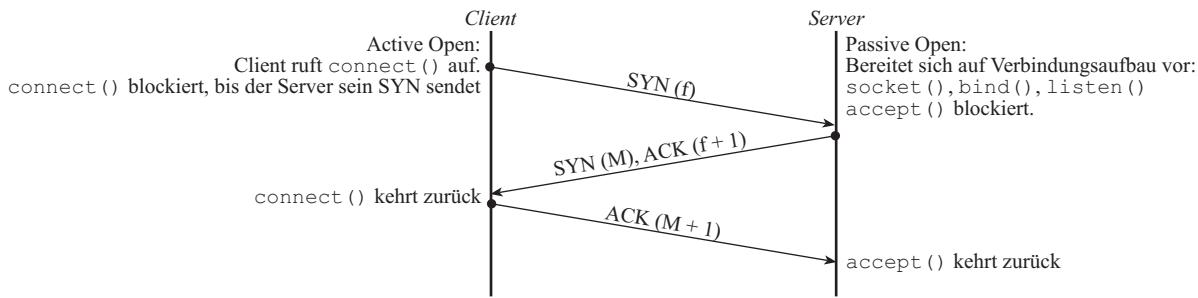


Abbildung 12.8: Three-Way Handshake des TCP-Protokolls

- Der Client bestätigt schließlich das SYN des Servers.

Dieser Vorgang wird in Abbildung 12.8 dargestellt.

Warum wird das FIN mit einer Sequenznummer $FIN + 1$ bestätigt? Da ein FIN genau ein Byte groß ist, erhöht sich die Sequenznummer um genau dieses Byte.

Mit den SYNs von Server und Client werden keine Daten transportiert, wohl aber die eine oder andere TCP-Option, die über das Feld Options des TCP-Headers hinterlegt ist. Zulässige Optionen in einem SYN sind:

Zeitstempel (*timestamp*)

Der Zeitstempel wird für Verbindungen mit hohen Geschwindigkeiten benötigt, um zu verhindern, daß die Daten durch doppelte oder verzögerte Segmente negativ beeinflußt, beispielsweise beschädigt, werden.

Fensterskalierung (*window scale*)

Weiter oben haben wir das Übertragungsfenster kennengelernt, daß dem Sender angezeigt, wie groß der Empfangspuffer des Clients ist. Das Header-Feld für die Fenstergröße ist 16 Bit lang und kann daher nicht größer als 65.535 Bytes sein. Je höher aber die Geschwindigkeit einer Verbindung ist, desto größer sollte auch das Fenster sein, um den Durchsatz zu erhöhen. Die Option *Window Scale* zeigt an, daß das angezeigte Fenster skaliert werden muß. Das empfohlene Verfahren, definiert in RFC 1323, verschiebt den Wert um 0 bis 14 Bits nach links und kann damit Werte von bis zu 14.024.490 Bytes annehmen. Diese Option kann nur in Anspruch genommen werden, wenn sie von beiden Systemen unterstützt wird.

Maximale Segmentgröße (*maximum segment size*)

Mit dem initialen SYN wird die maximale Segmentgröße angezeigt. Es teilt der Gegenstelle mit, wie groß ein TCP-Segment maximal sein darf. Diese Vereinbarung gilt nur für die aktuelle Verbindung.

Der Verbindungsabbau besteht aus insgesamt vier Einzelschritten (illustriert in Abbildung 12.9):

- Eine Seite der Verbindung sendet ein FIN (*finished*), meistens ausgelöst durch den Funktionsaufruf *close*, und vollzieht ein *Active Close*.
- Die andere Seite erhält dieses FIN und vollzieht den *Passive Close*. Wie üblich, wird das FIN bestätigt und der Applikation zugeführt, in der Regel als EOF (*end of file*).
- Die Applikation schließt darauf hin ihren Socket und sorgt dafür, daß es ebenfalls ein FIN sendet.
- Schließlich empfängt die Gegenseite das FIN und bestätigt es.

Zusammenfassend wollen wir uns das TCP-Zustandsdiagramm (*state transition diagram*) in Abbildung 12.10 anschauen, das alle definierten Zustände einer TCP-Verbindung aufzeigt. Es kann sehr nützlich sein, ein solches Diagramm lesen zu können, um TCP besser zu verstehen und um Anwendungen besser Debuggen zu können.

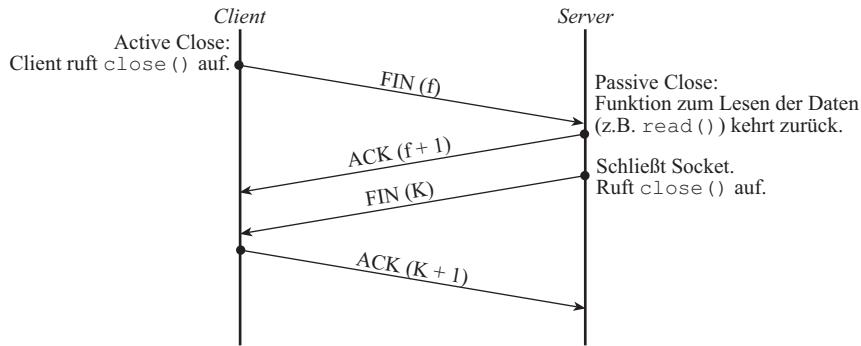


Abbildung 12.9: Verbindungsabbau bei TCP.

Ausgangspunkt ist eine geschlossene Verbindung. Sie kann entweder durch ein Active Open (ausgelöst via `SYN`) oder ein Passive Open (ausgelöst durch seinen serverseitigen Aufruf von `listen`) etabliert werden. Allerdings befindet sich der Server nach Empfang des clientseitigen `SYN` erst im Zustand `SYN_RCVD` (`SYN` empfangen) und geht nach Bestätigung dieses `SYN` in den Zustand `ESTABLISHED` über. Auch der Client muß seine Bestätigung senden, bevor der Datentransfer beginnen kann.

So ähnlich läuft auch die Terminierung einer Verbindung ab. Führt eine Anwendung `close` aus, vollzieht sie damit einen Active Close und sendet ein `FIN` an die Gegenseite. Damit geht die Verbindung auf Clientseite in den Zustand `FIN_WAIT_1` über, wartet auf die Bestätigung des Servers, wechselt in den Zustand `FIN_WAIT_2` über und sendet seinerseits ein `ACK`. Beachten Sie daß eine Anwendung, die `close` aufruft, bevor es ein `FIN` empfangen hat in `FIN_WAIT_1` übergeht, aber andernfalls, nämlich ein `FIN` empfängt, während sie sich im Zustand `ESTABLISHED` befindet, in ein `CLOSE_WAIT` (Passive Close) übergeht.

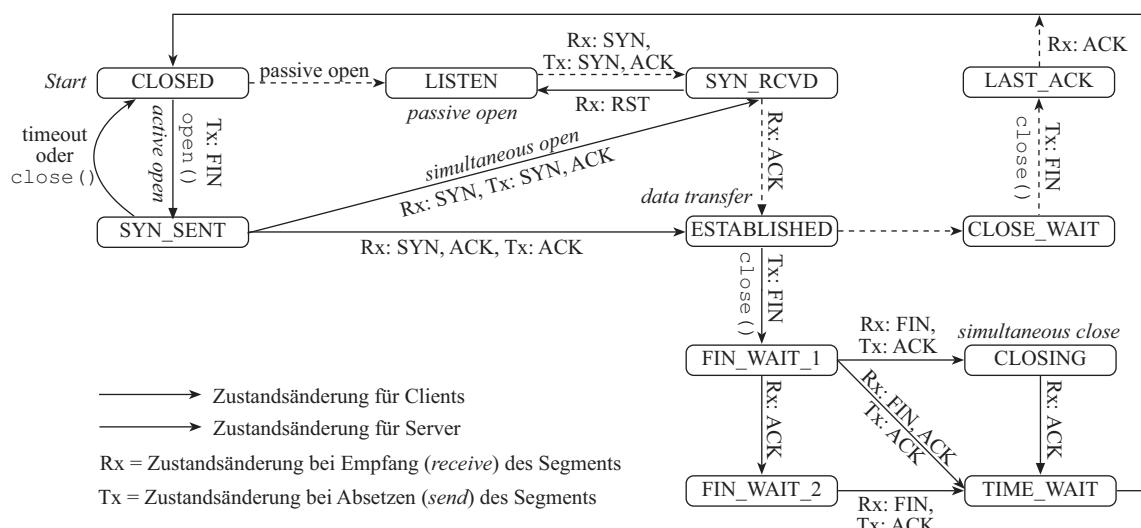


Abbildung 12.10: Zustandsdiagramm von TCP.

Der `TIME_WAIT`-Zustand wird von der Seite durchlaufen, die den Active Close vollzieht. Er dauert genau doppelt so lange wie die maximale Lebenszeit eines TCP-Segments (MSL, *maximum segment lifetime*). Nach RFC 1122 soll die MSL zwei Minuten betragen; bei anderen Systemen auch weniger. Da ein Segment nicht länger als MSL Minuten existieren kann, wird so ausgeschlossen, daß Pakete, die bereits als verloren galten, nicht nachträglich dem Ziel zugestellt werden, beispielsweise wenn ein Router aufgrund eines Fehlers neue Routen berechnen muß und in dieser Zeit Schleifen entstehen, in denen einzelne Segmente festsitzen.

12.3.3 Das Stream Control Transmission Protocol (SCTP)

SCTP, spezifiziert in RFC 2960, ist im Jahr 2000 von der *Transport Area Working Group* der IETF standardisiert worden. Es ist für den Transport von Signalisierungsnachrichten über geswitchte Telekommunikationsnetzwerke vorgesehen und arbeitet auf der Transportschicht oberhalb des IP-Layers.

Als verbindungsorientiertes Protokoll stellt es folgende Dienste bereit:

- Zuverlässiger, fehlerfreier Transport von Daten
- Unterstützung von *MTU Path Discovery* zur Ermittlung der optimalen Fragmentgröße
- Unterstützung der sequenziellen Datenübertragung auch über mehrere Streams hinweg.
- Zusammenfassung von mehreren Nachrichten in ein SCTP-Paket.
- Fehlertolleranz auf Netzwerkebene durch Unterstützung von Multi-Homing an beiden Endpunkten der Verbindung (*Association* genannt).

SCTP ist Byte-orientiert, unterstützt aber, im Gegensatz zu TCP, Record Boundaries. Jedes Paket enthält einen Nutzdatenbereich, der als *Chunk* bezeichnet wird und, sofern möglich, mit anderen Chunks in ein IP-Datagram eingearbeitet werden kann (Multiplexing). Ähnlich wie TCP werden die Daten sequenziert, wobei SCTP mehrere Streams über eine Assoziation mit jeweils eigener Sequenzierung verwalten kann. Die Flusskontrolle wurde so ausgelegt, daß sie der von TCP ähnelt und eine unkomplizierte Integration in IP-Netzwerke erlaubt.

Abbildung Abbildung 12.11 zeigt ein gekapseltes SCTP-Paket in einem IP-Datagram.

IP Header	SCTP Header	Data
20 Bytes	12 Bytes	...

Abbildung 12.11: *SCTP-Datagram gekapselt in einem IP Datagram.*

Eine der wichtigsten Neuerungen verglichen mit TCP ist die Unterstützung von Multi-Homing. Während TCP eine Punkt-zu-Punkt-Verbindung auf Basis eines Socket-Paars aufbaut, versorgt SCTP die Gegenseite mit einer Liste von Schnittstellen und einer SCTP-Port-Nummer, die zusammen eine *Assoziation* formen. Dieses Feature wird von beiden Endpunkten einer Assoziation unterstützt. Auf diese Weise ist ein Knoten über mehrere Pfade und Transportadressen erreichbar.

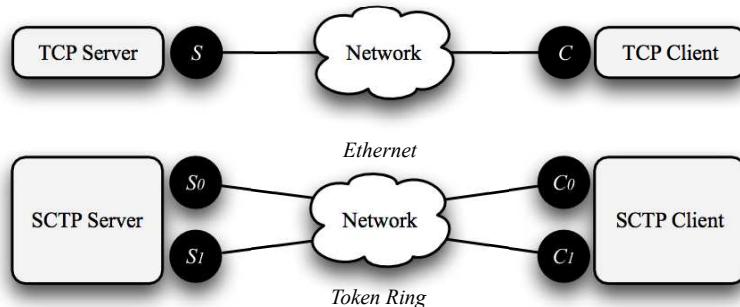


Abbildung 12.12: *Vergleich zwischen TCP und SCTP.*

Diese Eigenschaft wird besonders deutlich, wenn wir uns Abbildung 12.12 anschauen. Während TCP-Verbindungen in der Regel nur über ein Interface gleichzeitig abgewickelt werden können, erlaubt SCTP das Zusammenfassen mehrerer Interfaces zu Assoziationen. Diese Schnittstellen müssen nicht notwendigerweise die gleiche Übertragungstechnologie nutzen, können aber ihre Verbindungen über jeweils andere,

voneinander unabhängige Pfade aufbauen. SCTP überwacht die Verbindungen in einer Assoziation durch einen *Heart Beat*, der bei Versagen eines Pfades das Umschwenken auf den anderen ermöglicht. Die Anwendung kann Ereignisse (auch Benachrichtigungen genannt) abonnieren, so daß sie über bestimmte Vorkommnisse benachrichtigt wird, beispielsweise um eine Meldung auszugeben, wenn eine Verbindung in der Assoziation nicht mehr verfügbar ist. Abgesehen davon, sind Assoziationen gegenüber der Anwendung transparent.

Jede SCTP-Übertragungseinheit (PDU, *protocol data unit*) wird als Paket bezeichnet. Sie weist immer ein allgemeines Header-Format und einen Nutzdatenbereich auf, der aus mehreren Chunks bestehen kann. Das Protokoll sieht vor, entweder Benutzerdaten oder Kontrollinformationen in den Chunks zu platzieren, wobei letztere für Entwickler nicht weiter von Bedeutung sind. Das allgemeine Paketformat ist in Abbildung 12.13 dargestellt.

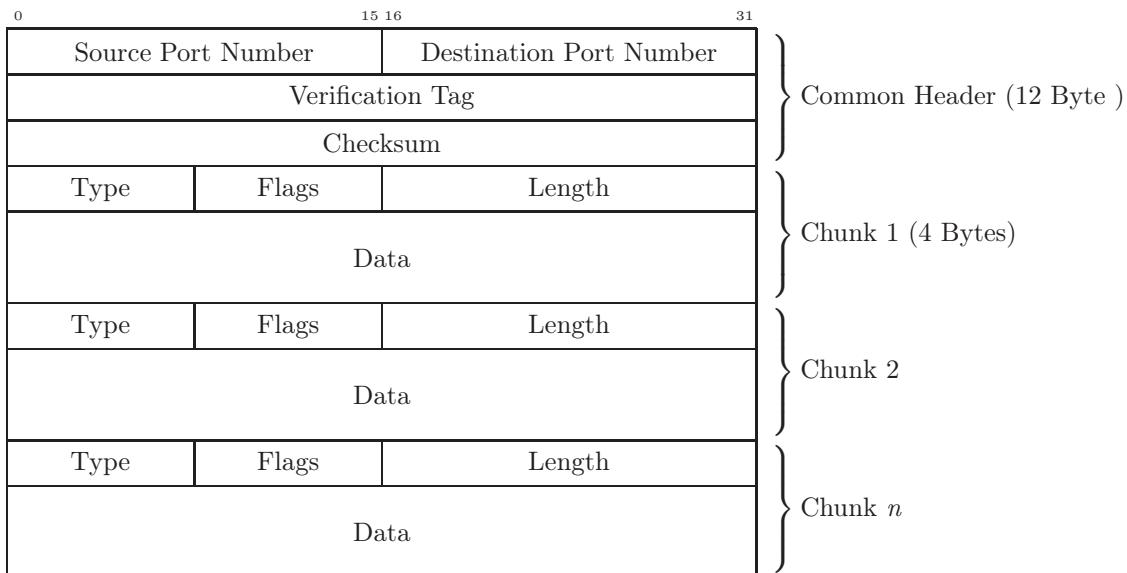


Abbildung 12.13: SCTP-Paketformat.

Zur Identifikation einer Association verwendet SCTP die gleichen Port-Nummern wie TCP und UDP. Die Prüfsumme (*checksum*) wird zur Fehlerprüfung benötigt, um herauszufinden, ob einzelne Bits bei der Übertragung umgekippt sind. Sie wird mit einer Variante des Adler-32-Algorithmus berechnet, der robuster ist als das herkömmliche Verfahren von TCP und UDP. Das Verification Tag wird während des Verbindungsaufbaus ermittelt.

12.3.3.1 Verbindungen auf- und abbauen

Der Verbindungsaufbau (das Herstellen einer Assoziation) findet in vier Schritten statt und wird daher als Four-Way-Handshake bezeichnet. Das Verfahren wird in Abbildung 12.14 illustriert.

Ausgehend vom Zustand CLOSE der beiden Endpunkte wird zunächst das Kontrollpaket INIT vom Client an den Server gesendet. Es enthält einen zufälligen Wert (*Tag*), gekennzeichnet als T_a und eine Sequenznummer j . Das Paket selbst erhält ein Null-Tag (T_0). Auf Serverseite wird ein *Transmission Control Block* erzeugt, der alle relevanten Daten für die Assoziation enthält und dem Client per INIT-ACK als sog. Cookie (C) übermittelt. Der TCB enthält einen geheimen Schlüssel und einen *Message Authentication Code* (MAC), das zusammen das Cookie formen. Das T_a -Tag muß über die gesamte Lebensdauer der Assoziation mit jedem Paket mitgeliefert werden. Das Paket mit dem TCB wird mit T_a markiert und enthält ein zufälliges Tag T_v , welches der Server generiert, bevor es dem Client übermittelt. Noch immer ist die Verbindung auf Seitend es Servers im Zustand CLOSED. Bei Erhalt des INIT-ACKs liefert der Client das Cookie in einem COOKIE-ECHO-Paket zurück. Das Paket ist mit T_v markiert. Diese Markierung wird für die Kommunikation von Client zum Server verwendet. In die Gegenrichtung wird mit T_a gesendet. Im letzten Schritt bestätigt der Server den Erhalt des COOKIE-ECHOs mit

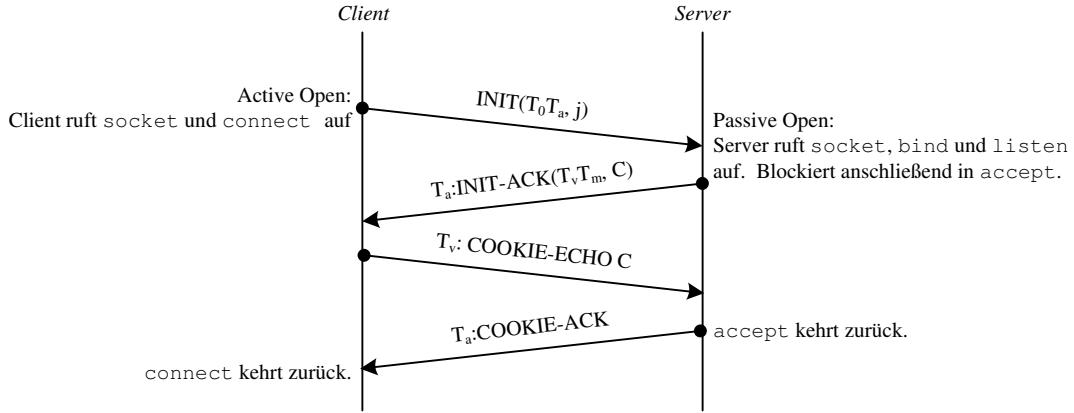


Abbildung 12.14: SCTP Four-Way-Handshake

einem COOKIE-ACK. Erst jetzt befindet sich die Verbindung im Zustand ESTABLISHED und der Datentransfer kann beginnen.

Der Verbindungsabbau ist da deutlich übersichtlicher. Im Gegensatz zu TCP kennt SCTP kein Half Open, bei dem ein Ende noch Daten senden kann, obwohl das andere Ende bereits die Verbindung trennt. Sobald ein Ende die Assoziation auflöst, darf keine Seite mehr Daten übertragen. Alle Daten im Puffer auf Seiten des Passive Close werden übermittelt und die Assoziation aufgelöst. Abbildung X illustriert den Vorgang:

Durch den Einsatz von Verification Tags vermeidet STCP die Aufrechterhaltung von Verbindungen im TIME_WAIT-Zustand. Eintreffende Chunks einer alten Verbindung haben nicht das korrekte Verification Tag und sind damit ungültig.

Das folgende Zustandsdiagramm ist eine graphische Abbildung der Beschreibungen in RFC2960 (Seite 48ff) und zeigt deutlich, daß SCTP wesentliche Vereinfachungen hinsichtlich des Verbindungsmanagements erreicht wurden.

Ist das Cookie im COOKIE-ECHO ungültig (beispielsweise wenn die Integritätsprüfung fehlschlug), wird das Paket verworfen und der Empfänger bleibt im Zustand CLOSED. Sendet eine Applikation ein INIT geht sie in den Zustand COOKIE-WAIT über und bleibt dort solange ein Timer abgelaufen ist, oder bis ein INIT-ACK empfangen wurde, was sie veranlaßt ein COOKIE-ECHO abzusetzen. Erst mit dem Empfang des COOKIE-ACK geht die Applikation in den Zustand ESTABLISHED über.

Führt eine Applikation einen Active Close durch so geht sie in den Zustand SHUTDOWN-PENDING über, um hier ihre Warteschlange zu leeren und sendet ein SHUTDOWN, so daß die Anwendung schließlich im Zustand SHUTDOWN-SENT auf die Bestätigung (SHUTDOWN-ACK) wartet, um in den Zustand CLOSED überzugehe. Empfängt die Anwendung ein SHUTDOWN während sie im Zustand ESTABLISHED verweilt (Passive Close), so geht sie in den Zustand SHUTDOWN-RECEIVED über, um ihre

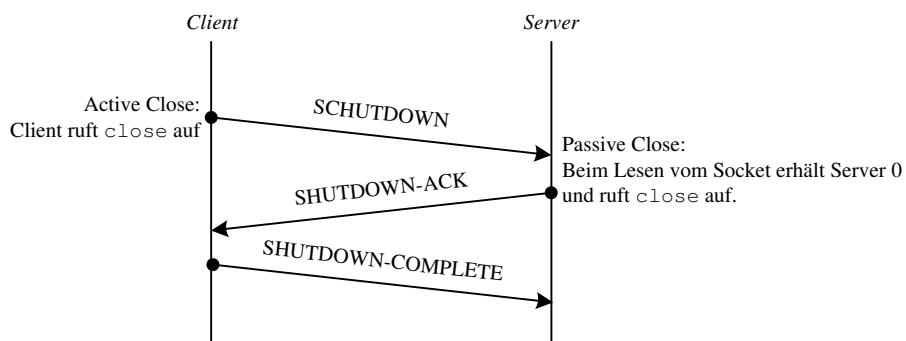


Abbildung 12.15: Abbau von SCTP-Verbindungen

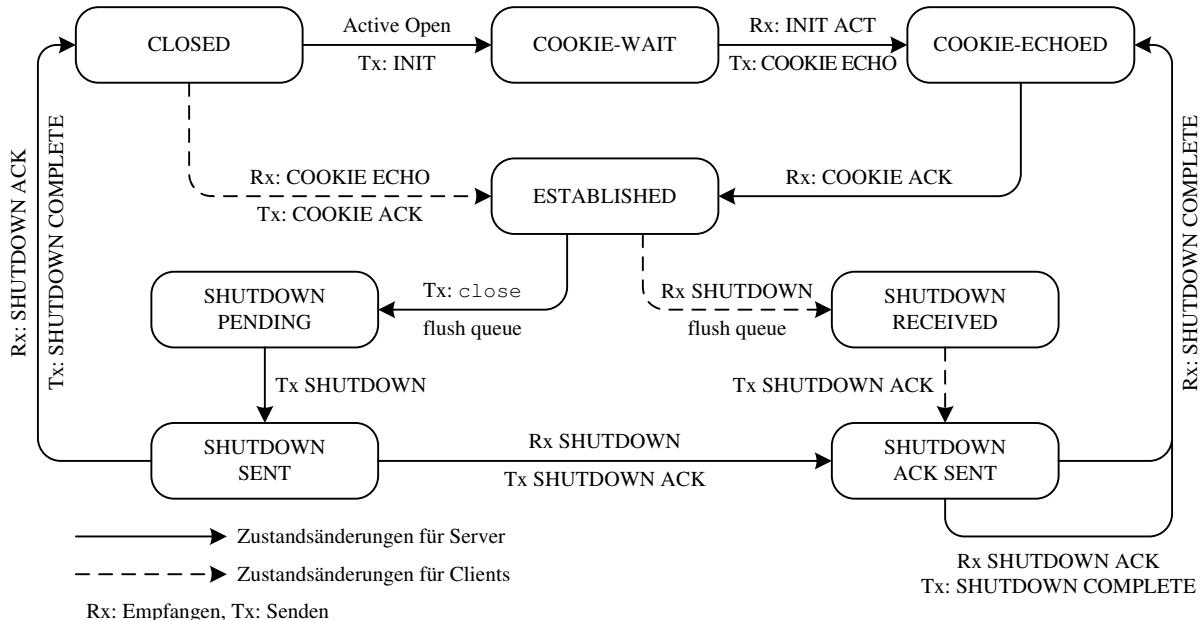


Abbildung 12.16: SCTP-Zustandsdiagramm

Warteschlange zu leeren und ein SHUTDOWN-ACK zu senden. Im Zustand SHUTDOWN-ACK wartet sie auf die Bestätigung der Gegenstelle um den Vorgang abzuschließen.

12.3.4 Portnummern und Sockets

Jedem Prozess, ob Server oder Client, ist eine 16-Bit Portnummer zugewiesen, die für TCP, UDP oder SCTP verwendet werden kann. Möchte ein Client Kontakt zu einem Server herstellen, muß der die Portnummer des Serverprozesses kennen. Während Clients keine festgelegten Portnummern verwenden müssen, sind die Ports der wichtigsten Serverprozesse standardisiert. So ist der FTP-Server in der Regel über Port 21, der Webserver über Port 80, usw. erreichbar. Diese *well known* (genau bekannten) Ports werden von IANA (*Internet Assigned Numbers Authority*) vergeben.

Clients verwenden sogenannte *ephemere* Ports, solche, die nur kurze Zeit benötigt werden. Sie werden durch das Transportprotokoll vergeben, so daß sich Clients in der Regel nicht darum kümmern müssen.

Bereich	Bedeutung
1 - 1023	Well known. Beispielsweise 23 für Telnet oder 110 für News.
1024 - 49151	Registrierte Ports. Meist für herstellerspezifische Anwendungen.
49152 - 65535	Dynamische und private Ports für alle möglichen Anwendungen

Tabelle 12.1: Von der IANA vergebene Portnummern.

Wir haben bisher verschwiegen, was genau ein Socket ist. Die IP-Adresse und die Portnummer eines Prozesses bilden zusammen einen *Socket*, quasi den Endpunkt einer Verbindung. Folglich wird eine Verbindung als Socket-Paar charakterisiert: Client-IP, Client-Port und Server-IP sowie Server-Port. Das gilt nur für TCP, denn UDP arbeitet verbindungslos: zwar gibt es auf beiden Seiten Sockets, aber es existiert zu keinem Zeitpunkt ein Kommunikationskanal zwischen beiden.

Kapitel 13

Die BSD-Socket API

Basic research is what I am doing when I don't know what I am doing.

WERNHER VON BRAUN (1912 - 1977)

Der Begriff der Sockets ist aus der BSD-Entwicklung hervorgegangen und hat seine Bedeutung bis in die moderne Netzwerkprogrammierung halten können. In diesem Kapitel lernen wir die zugrunde liegenden Datenstrukturen kennen und befassen uns mit grundlegenden Details, wie etwa der Ausrichtung der Daten im Netzwerk.

Nach und nach machen wir uns mit wichtigen Funktionen vertraut und versuchen stets protokollunabhängig zu arbeiten. Aus diesem Grund werden wir die eine oder andere Funktion mit einem Wrapper versehen, der uns die Arbeit mit den beiden IP-Generationen IPv4 und IPv6 erleichtern wird.

13.1 Datenstrukturen

Fast alle Operationen im Rahmen der Netzwerkprogrammierung werden direkt auf Sockets angewendet oder stehen in indirektem Zusammenhang zu ihnen. Zentraler Bestandteil ist daher die Socket-Adressstruktur (*socket address structure*) mit dem Namen `sockaddr_in` (IPv4) und `sockaddr_in6` (IPv6). Alle Socket-Adressstrukturen beginnen mit `sockaddr_` und erhalten ein Suffix, das die Protokollfamilie identifiziert. Eine Sonderform bildet die allgemeine Socket-Struktur `sockaddr`, die wir zuerst besprechen wollen und kein Suffix erhält, da sie protokollunabhängig ist.

13.1.1 Standard-Adress-Struktur

Viele Funktionen erwarten eine Adressstruktur, die als Zeiger übergeben wird. Da die Funktionen aber mit vielen verschiedenen Protokollfamilien arbeiten müssen, deren Adressstrukturen sich zum Teil stark voneinander unterscheiden, erwarten sie eine Standardadressstruktur als Argument:

```
struct sockaddr {
    uint8_t      sa_len;
    sa_family_t  sa_family;
    char         sa_data[14];
};
```

`sa_len`

Größe der `sockaddr`-Struktur.

sa_family

Adressfamilie des zugrunde liegenden Protokolls. Beispielsweise AF_INET.

sa_data

Protokollspezifische Adresse. Groß genug, um auch UNIX Domain Sockets zu speichern.

Das ist nicht weiter tragisch, erfordert von uns aber etwas mehr Schreibarbeit, denn wir müssen unsere Zeiger auf protokollspezifische Strukturen via Casting in Zeiger auf `sockaddr`-Strukturen umwandeln.

Praktisch könnte das etwa so aussehen:

```
struct sockaddr_in myaddr;
/* Socket erzeugen */
if (bind(socketfd, (struct sockaddr *)&myaddr, sizeof(myaddr)) < 0)
    err_fatal("bind() failed");
...
```

Wer in Zukunft nicht so viel schreiben möchte, kann das ganze per `#define` abkürzen, etwa so:

```
#define SOCK_PTR (struct sockaddr *) /* in den Header? */
...
if (bind(socketfd, (SOCK_PTR)&myaddr, sizeof(myaddr)) < 0)
    err_fatal("bind() failed");
```

Eine Frage, die unter erfahrenen C-Progroammierern oft gestellt wird, ist, warum nicht einfach ein `void`-Zeiger verwendet wird, um das Casting überflüssig zu machen. Ein richtiger Einwand, aber leider sind die meisten Funktionen aus der Prä-ANSI-Ära. Da gab es nämlich noch keine `void`-Zeiger. Zu Gunsten existierenden Codes wurde das Konzept beibehalten.

13.1.2 IPv4 und IPv6

Der Header `<netinet/in.h>` definiert Datenstrukturen für IPv4 und IPv6. Die Informationen eines IPv4-Sockets sind in `struct sockaddr_in` hinterlegt:

```
struct sockaddr_in {
    uint8_t          sin_len;
    sa_family_t      sin_family;
    in_port_t        sin_port;
    struct in_addr   sin_addr;
    char             sin_zero[8];
};
```

sin_len

Länge der Socket Address Structure. Normalerweise 16 Bytes.

sin_family

Repräsentiert die Adressfamilie der Struktur, immer AF_INET für IPv4.

sin_port

Portnummer für diesen Socket (TCP und UDP).

sin_addr

Struktur zur binären Ablage der IP-Adresse.

```
struct sin_addr {
    in_addr_t s_addr;
}
```

sin_zero

Wird nicht verwendet, muß aber immer 0 gesetzt werden.

Die Datentypen `uint8_t` und `in_port_t` sind POSIX-Datentypen und wurden per `typedef` definiert. Desweiteren können wir auf `int8_t`, `uint16_t`, `int16_t`, `uint32_t` und `int32_t` treffen, die als Beitrag zur Plattformunabhängigkeit definiert wurden (das `u` steht übrigens für `unsigned` und die Ziffer für die Anzahl der Bits). Die beiden Typen `sa_family_t` und `in_port_t` wurden lediglich auf `uint8_t` und `uint16_t` umdefiniert.

Aus historischen Gründen ist die IP-Adresse in einer eigenen Struktur untergebracht. Daher ergeben sich zwei Varianten, um auf die IP-Adresse zuzugreifen: `myaddr.sin_addr` und `myaddr.sin_addr.s_addr`, wobei `myaddr` eine initialisierte Socket Address Structure darstellt. Einige Funktionen erwarten einen Zeiger auf die Adressstruktur, andere wiederum einen Integer. Wir müssen darauf achten, wie wir auf diesen Member zugreifen.

Die Socket-Struktur für IPv6 ist der von IPv4 ähnlich, weist aber im Detail wesentliche Unterschiede auf. POSIX definiert für die IPv6-Adressstruktur folgende Member:

```
struct sockaddr_in6 {
    uint8_t          sin6_len;        /* length of this struct (28) */
    sa_family_t      sin6_family;    /* AF_INET6 */
    in_port_t        sin6_port;      /* transport layer port# */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t         sin6_scope_id;  /* set of interfaces for a scope */
};
```

sin6_len

Länge der Adress-Struktur. Für IPv6 immer 28 Bytes.

sin6_family

Angabe der Adressfamilie. Wird für IPv6 immer auf `AF_INET6` gesetzt.

sin6_port

Angabe des UDP- oder TCP-Ports.

sin6_addr

Struktur vom Typ `in6_addr` zur Aufnahme der IPv6-Adresse; 128 Bits.

```
struct in6_addr {
    uint8_t  s6_addr[16];
};
```

sin6_scope_id

Wird für die Anbindung (*binding*) an Adressen mit Zonen verwendet. Enthält entweder eine Interface-ID oder eine Site-ID.

13.1.3 Host Byte Order und Network Byte Order

Bei der Beschreibung der beiden Adress-Strukturen haben wir eine wichtige Tatsache einfach unter den Tisch fallen lassen: die Felder `sin_port` und `sin_addr` (IPv4) sowie `sin6_port` und `sin6_addr` (IPv6) werden mit einer bestimmten Ausrichtung (*byte order*) gespeichert. Das ist notwendig, damit alle beteiligten Systeme, die naturgemäß auf unterschiedlichsten Plattformen basieren können, die Bits in der richtigen Reihenfolge auslesen.

Holen wir etwas weiter aus. Jeder Prozessor weist eine bestimmte Ausrichtung auf, nach der die Bits im Speicher angeordnet werden: entweder *little endian* oder *big endian*.

Systeme mit Big Endian Byteorder platzieren das erste Byte eines Multibyte-Datentyps auf der Seite des hochwertigsten Bits (MSB, *most significant bit*), während Little Endian genau umgekehrt vorgeht

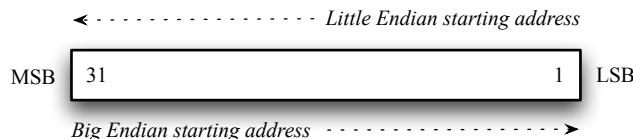


Abbildung 13.1: Unterschied zwischen Little Endian und Big Endian.

und am Ende des niedrigsten Bit (LSB, *least significant bit*) beginnt. Folglich würden beide Systeme den gleichen Wert anders interpretieren und beim Austausch von Daten unbeabsichtigt durcheinander kommen.

Es gibt eine einfache Möglichkeit, herauszufinden, welche Byteorder ein System verwendet. Listing 13.1 führt das passende Programm auf.

Listing 13.1: Byteorder eines Systems herausfinden

```

1 #include <stdio.h>
2
3 static unsigned int is_little_endian(void);
4
5 int main(void) {
6     printf("%s Endian\n", (is_little_endian() == 1 ? "Little" : "Big"));
7     exit(0);
8 }
9
10 unsigned int is_little_endian(void) {
11     union {
12         unsigned int i;
13         unsigned char uc[sizeof(int)];
14     } u;
15
16     u.i = 1;
17     return (unsigned)u.uc[0];
18 }
```

Listing 13.1: `xcode/byteorder.c` - Byteorder eines Systems herausfinden.

Wir verwenden eine `union` mit zwei Membern: `i` und `uc`, weisen `i` den Wert 1 zu und legen einen `char`-Array an, um einen Blick auf das erste Byte zu werfen. Wird eine 1 zurückgegeben, verwendet das System Little Endian, andernfalls Big Endian. Wir verwenden eine `union` keinen `struct`, da eine `union` mit all seinen Membern an einer Speicherposition abgelegt wird und somit alle Bits der Member direkt aufeinander folgen.

Noch schneller geht es mit folgendem Code-Abschnitt:

```

short int word = 0x0001;
char *byte = (char *) &word;
return(byte[0] ? 1 : 0);
```

Wir sparen uns hier ein paar Zeilen, letztendlich kommt es aber auf das gleiche hinaus. □

Grund für die detaillierte Behandlung dieser Thematik ist die Tatsache, daß einige Daten nur in einer bestimmten Byteorder vorliegen. Netzwerkprotokolle arbeiten immer nach der *Network Byte Order* (NBO), die der von Big Endian entspricht und die Systeme arbeiten immer nach der *Host Byte Order* (HBO), die entweder Big Endian oder Little Endian sein kann. Der POSIX-Standard schreibt aber vor, daß bestimmte Felder auch lokal in der NBO abgelegt werden müssen, wie beispielsweise die Portnummer und die Adresse von IPv4 und IPv6.

Glücklicherweise existieren Funktionen für die Umwandlung zwischen Network Byte Order und Host Byte Order.

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
Rückgabewert: hostlong bzw. hostshort in Network Byte Order.

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
Rückgabewert: netlong bzw. netshort in Host Byte Order.
```

Viele Implementierungen legen die vier als Makros andere als Funktionen aus. Auf Systemen deren Byte Order der Network Byte Order entspricht sind die jeweiligen Makros als NULL-Makros definiert.

Das *s* und das *l* in den Funktionsnamen stehen für *short* und *long*, die noch aus der Zeit der legendären VAX-Implementierung früherer BSD-Systeme. Wir müssen uns daran gewöhnen, daß mit *s* ein 16-Bit-Wert und mit *l* ein 32-Bit-Wert gemeint ist. Auch auf 64-Bit-Systemen arbeiten die Funktionen `htonl` und `ntohl` mit 32-Bit-Werten.

13.2 Grundlagen

Primäres Ziel dieses Kapitels ist es, erstmals einen vollständigen Client und Server zu entwickeln. Schritt für Schritt besprechen wir die Einzelheiten und lernen die notwendigen Funktionen kennen. Desweiteren werden wir eine wichtige Eigenschaft der meisten Server kennenlernen, nämlich mehrere Clients zeitgleich bedienen zu können, indem wir einfach einen neuen Serverprozess erzeugen, der die Anfragen erfüllen kann.

13.2.1 Das Transmission Control Protocol

Als wir den Three-Way-Handshake in Abschnitt 12.3.2 besprochen haben, wurde beiläufig erwähnt, daß sich der Server erst auf eingehende Verbindungen vorbereiten muß. Erst wenn er bestimmte Schritte durchgeführt hat, ist er bereit, Anfragen zu verarbeiten.

In der Regel ruft der Server die folgenden Funktion in dieser Reihenfolge auf:

1. `socket` - Erstellung eines Sockets. Hier legen wir fest, welches Protokoll für die Kommunikation verwendet werden soll, welcher Socket-Typ verwendet wird und ob wir UDP, TCP oder SCTP für den Transport nutzen möchten.
2. `bind` - Weist dem Socket eine lokale Adresse zu, die sich stets aus einem Paar von IP-Adresse (IPv4 oder IPv6) und einer Portnummer (UDP oder TCP) zusammensetzt.
3. `listen` - Holt *ausstehende* Verbindungen aus der Warteschlange des Servers, um Client-Anfragen zu beantworten. Erst in diesem Zustand kann der Server Verbindungen annehmen.
4. `accept` - Holt eine *fertige* Verbindung aus der Warteschlange der fertigen Verbindungen.

Clients gehen ähnlich vor, rufen allerdings (fast) niemals `bind`, `listen` oder `accept` auf:

1. `socket` - Erstellt einen Socket. Auch der Client muß Kommunikationsprotokoll (IPv4/v6, etc.), Transportprotokoll und Socket-Typ festlegen.
2. `connect` - verbindet sich über einen Socket mit dem Server, der sich im Zustand LISTEN befinden muß.

Wenn Client und Server diese Schritte erfolgreich durchgeführt haben, besteht eine Verbindung zwischen beiden und die Datenübertragung kann stattfinden. Das geschieht in der Regel über die beiden Funktionen `read` und `write`. Der Client schreibt in den Socket, der Server liest die Anfrage, verarbeitet sie evtl. auch und schreibt seinerseits die Antwort in den Socket, so daß der Client die Antwort erhält.

13.2.1.1 Einen Socket erstellen (socket)

Vor jeder Kommunikation steht die Erstellung eines Sockets.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Rückgabewert: Socket Descriptor oder -1 bei Fehler.

domain

Spezifiziert das Kommunikationsprotokoll dieses Sockets. Das Präfix AF steht für *Address Family* obwohl POSIX oftmals von *Domains* spricht. Folgende Flags sind durch POSIX spezifiziert:

AF_INET Internet Domain Socket für IPv4.

AF_INET6 Internet Domain Socket für IPv6.

AF_LOCAL UNIX Domain Sockets.

type

Typ des Sockets. POSIX stellt uns drei Flags zur Verfügung:

SOCK_STREAM

Stream Socket - Bietet sequenzielle, zuverlässige, bidirektionale, verbindungsorientierte Byte-streams an.

SOCK_DGRAM

Datagram Socket - Bietet verbindungslose, unzuverlässige nachrichtenbasierte Übermittlung von Daten fixer Länge an.

SOCK_SEQPACKET

Bietet sequenzielle, zuverlässige, bidirektionale, verbindungsorientierte Record-basierte Datenübermittlung an.

protocol

Zeigt an, welches Transportprotokoll wir für den Datentransfer verwenden möchten.

IPPROTO_TCP Transmission Control Protocol.

IPPROTO_UDP User Datagram Protocol.

IPPROTO_SCTP Stream Control Transmission Protocol.

Wir können die Konstanten nicht einfach Kombinieren, wie es uns gefällt. Ein Stream-Socket (**SOCK_STREAM**) kann natürlich nur mit einem Stream-Protokoll (**IPPROTO_TCP** oder **IPPROTO_SCTP**) arbeiten. Ebenso kann ein Datagram-Socket (**SOCK_DGRAM**) nur mit einem Datagram-Protokoll (**IPPROTO_UDP**) funktionieren.

Der POSIX-Name **AF_LOCAL** für UNIX Domain Sockets ist in älterem Code noch häufig als **AF_UNIX** anzutreffen. Des Weiteren existieren je nach Bedarf weitere Konstanten für andere Protokollfamilien wie etwa **AF_ISO** (OSI-Protokoll) oder **AF_NS** (für Xerox NS).

Die **socket**-Funktion liefert einen *Socket Descriptor* vom Typ **int**, der ähnlich gehandhabt werden kann wie ein File Descriptor.

13.2.1.2 Den Socket mit einer lokalen Adresse verbinden (bind)

Der erstellte Socket wurde noch nicht mit einer Adresse und einem Port versehen. Die Funktion `socket` hat nur einen Descriptor im Kernel registriert, der für eine bestimmte Protokollfamilie und ein passendes Transportprotokoll vorbereitet wurde. Sonst nichts. Solche Sockets werden von POSIX oftmals als „unbenannte Sockets“ (*unnamed sockets*) bezeichnet.

Die `bind`-Funktion weist diesem Socket nun eine lokale Adresse und eine Portnummer zu.

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

`socket`

Socket Descriptor, der an die Adresse `address` angebunden werden soll.

`address`

Zeiger auf eine `sockaddr`-Struktur mit den Adressinformationen, die an den Socket gebunden werden soll.

`address_len`

Größe der Struktur, auf die `address` zeigt.

Wie wir in Abschnitt 12.3.4 erfahren haben, sind Server in der Regel über ihren Standardport erreichbar. Wenn wir `bind` die Adressstruktur übergeben verfügt sie über die Portinformation, so daß wir nicht dem Kernel die Zuweisung eines freien Ports überlassen (was aber durchaus legitim wäre). Dazu genügt es, `sin_port`, bzw. `sin6_port` eine 0 zuzuweisen; den Rest erledigt der Kernel. Eine IP-Adresse anzugeben kann sinnvoll sein, kann aber auf Systemen mit mehreren Schnittstellen problematisch sein. Wir haben in diesem Fall folgende Optionen:

- Für IPv4 können wir dem Feld `sin_addr` der `sockaddr`-Struktur die Konstante `INADDR_ANY` zuweisen, die den Kernel veranlaßt, den Kernel eine IP-Adresse für uns auszuwählen:

```
struct sockaddr_in sockaddr;
sockaddr.sin_addr.s_addr = INADDR_ANY;
```

- Grundsätzlich ist das mit IPv6 genauso, allerdings weisen wir `sin6_addr` keine Konstante zu, sondern eine Variable, die automatisch initialisiert wird:

```
struct sockaddr_in6 sockaddr;
sockaddr.sin6_addr = in6addr_any;
```

TCP-Clients setzen, wenn überhaupt, nur die IP-Adresse in der Adressstruktur und überlassen dem Kernel die Auswahl eines freien Ports für den Socket. Die meisten Clients rufen nicht einmal `bind` auf.

Wenn der Kernel einen Port für unsere Clients aussucht, können wir ihn hinterher nicht einfach durch einen Blick in die Felder `sin_port` oder `sin6_port` abfragen, denn wir übergeben die Adressstruktur als Konstante (Schlüsselwort `const` in der Synopsis). Die einzige Chance wäre `getsockname` aufzurufen und dann die beiden Felder zu untersuchen.

13.2.1.3 Ausstehende Verbindungen abholen (listen)

Ein Server ruft die Funktionen `socket`, `bind`, `listen` und `accept` in dieser Sequenz auf (Abschnitt 13.2.1). Wir haben erfahren, daß `listen` nur *ausstehende Verbindungen* aus einer Warteschlange des Kernels abholt. `accept` wiederum holt nur *etablierte Verbindungen* aus einer anderen Warteschlange ab.

Das Konzept ist folgendermaßen zu verstehen: Clients, die eine Verbindung mit einem Server herstellen möchten, müssen den Three-Way-Handshake durchführen. Sendet der Client ein SYN erstellt der Kernel des Servers einen neuen Eintrag in der Warteschlange für „noch nicht fertig ausgehandelte Verbindungen“ (ausstehend). Erst nach Abschluß des Three-Way-Handshake wird diese Verbindung in den Status „fertig“ (etabliert) erhoben und in die Warteschlange der fertigen Verbindungen eingereiht. Abbildung 13.2 illustriert diesen Sachverhalt.

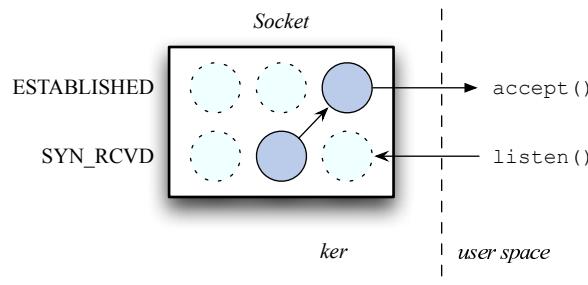


Abbildung 13.2: Zwei Queues eines Sockets

Sobald der Three-Way-Handshake abgeschlossen ist, wandert die Verbindung in die Warteschlange passiver Sockets. Sockets, die mit `listen` bearbeitet worden sind, werden oftmals als *Listening Sockets* bezeichnet.

Aus dieser Perspektive ist der Name `listen` für diese Funktion durchaus passend. Wir veranlassen den Kernel, einen nicht verbundenen Socket in einen passiven Socket zu konvertieren, so daß eingehende Anfragen einer Verbindung an diesen Socket akzeptiert werden können um ihn vollständig zu verbinden. Später können wir mit `accept` solche verbundenen Sockets aus der Warteschlange holen.

Die Synopsis von `listen` lautet folgendermaßen:

```
#include <sys/socket.h>

int listen(int socket, int backlog);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

socket

Socket Descriptor dessen Socket für den Datentransfer verbunden werden soll.

backlog

Anzahl der etablierten Verbindungen, die für diesen Socket durch den Kernel verwaltet werden sollen.

Das Argument `backlog` zeigt an, wie viele Einträge in der Warteschlange für etablierte Verbundungen verwaltet werden sollen. Einige Implementierungen schließen beide Queues in die Anzahl mit ein, andere beziehen sich auf jeweils eine. Alle Implementierungen erstellen beide Queues (SYN_RCVD und ESTABLISHED) mit `backlog` Einträgen und legen gleichzeitig eine Obergrenze fest, die nicht programmatisch umgangen werden kann.

In unseren Erläuterungen verwischt die ursprüngliche Bedeutung von `backlog` mit der tatsächlichen Interpretation dieses Wertes. So ist der Wert eigentlich nur auf die Anzahl der Einträge in der Warteschlange für etablierte Verbindungen (ESTABLISHED) bezogen. So wird ausgeschlossen, daß der Server eine unbegrenzte Anzahl ausstehender Verbindungen verwalten muß, wenn sie sowieso nicht akzeptiert werden können.

Viele Systeme verwenden `backlog` als Faktor zur Berechnung der Einträge beider Warteschlangen. So setzen viele Systeme den BSD-Algorithmus ein, der `backlog` mit 1,5 multipliziert, während andere einfach nur 1 zu `backlog` addieren. Um herauszufinden, wie groß `backlog` maximal sein kann, empfiehlt sich ein Blick in die Manpages.

13.2.1.4 Etablierte Verbindungen akzeptieren (`accept`)

Ruft ein Server `accept` auf, holt er eine etablierte Verbindung aus der Warteschlange, um Client-Anfragen zu verarbeiten.

```
#include <sys/socket.h>

int accept(int socket, struct sockaddr *address, socklen_t *address_len);
```

Rückgabewert: Socket Descriptor oder -1 bei Fehler.

`socket`

Socket Descriptor aus dessen Queue die nächste etablierte Verbindung abgeholt werden soll.

`address`

Adresse und Port des Client-Prozesses.

`address_len`

Zeiger auf eine Variable vom Typ `socklen_t` zur Rückgabe der Größe von `address`.

Der Rückgabewert der Funktion ist ein *neuer* Socket Descriptor, der nur für *diese eine* Verbindung erzeugt wurde. Ist die Datenübertragung beendet, wird er wieder geschlossen. Der Socket des neuen Socket Descriptor bezeichnet man oft als *Connected Socket*. Nach Rückkehr der Funktion ist die tatsächliche Größe der `sockaddr`-Struktur in `address_len` enthalten.

Im Gegensatz zu `socket` und `bind` können wir die Länge der `sockaddr`-Struktur nicht mit einem einfachen `sizeof(address)` übergeben, denn hier wird ein Zeiger erwartet. Grund für diese Besonderheit ist die Tatsache, daß der Kernel nicht im Vorfeld wissen kann, wie groß die Addressstruktur nach dem Abholen einer Verbindung ist. Warum das so ist, wird klar, wenn wir uns überlegen, aus welcher Richtung die Daten kommen. Wenn wir `accept` aufrufen, übergeben wir dem Kernel nur die Adresse im Hauptspeicher, wo er die Informationen über den Client ablegen kann. Um zu verhindern, daß der Kernel über die Struktur hinausschreibt, übergeben wir die Größe der Struktur. Kehrt die Funktion zurück, enthält `address_len` die tatsächliche Anzahl von Bytes, die der Kernel in den Speicher, auf den `sockaddr` zeigt, geschrieben hat. Somit sind die beiden Argumente `sockaddr` und `address_len` bei Aufruf der Funktionen Werte und bei ihrer Rückkehr Ergebnisse. Solche Argumente nennen wir *Value Result Arguments*.

Es gibt genau vier Funktionen, die eine Adress-Struktur als Value Result Arguments verwenden: `accept`, `recvfrom`, `getsockname` und `getpeername`. Die Verwendung folgt damit immer dem gleichen Schema:

```
struct sockaddr_in myaddr;
socklen_t address_len = sizeof(myaddr);

if (getsockname(socketfd, (struct sockaddr *)&myaddr, &address_len) < 0)
    err_fatal("getpeername() failed");
```

Für alle anderen Funktionen wie etwa `bind` oder `connect` bleibt alles beim Alten:

```
struct sockaddr_in myaddr;

if (bind(socketfd, (struct sockaddr *)&myaddr, sizeof(myaddr)) < 0)
    err_fatal("bind() failed");
```

Bei IPv4 und IPv6 ändert sich der Wert von `address_len` nicht, da die Adress-Strukturen immer die gleiche Länge aufweisen (nämlich 16 Bytes für IPv4 und 28 Bytes für IPv6). Eine Ausnahme bilden UNIX Domain Sockets, die bis zu 128 Bytes lang sein können.

UNIX Domain Sockets sind nicht für echte Netzwerkkommunikation vorgesehen, sondern stellen eine einfache Methode zur Kommunikation zweier Prozesse auf dem gleichen System zur Verfügung. Daher auch die traditionelle Bezeichnung der Adressfamilie `AF_LOCAL`.

13.2.1.5 Clients stellen Verbindungen mit connect her

Am Anfang des Abschnitts 13.2.1 haben wir gesehen, daß Clients mindestens `socket` und `connect` aufrufen müssen um eine Verbindung mit einem Server herzustellen. Während `socket` einen Socket Descriptor für die Kommunikation mit einem bestimmten Protokoll vorbereitet, verwenden wir `connect`, um den Three-Way-Handshake mit dem Server einzuleiten.

```
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

Rückgabewert: 0 bei Erfolg und -1 bei Fehler.

`socket`

Socket Descriptor dessen Socket wir verbinden möchten.

`address`

Wird bei Rückkehr der Funktion die Informationen des Servers enthalten.

`address_len`

Größe von `address`.

Clients müssen nicht `bind` aufrufen, um sich einen freien Port und eine gültige IP-Adresse zu besorgen, denn das erledigt standardmäßig der Kernel.

Während der Ausführung von `connect`, sind vielfältige Fehlerscheinungen möglich:

- Wird ein SYN des Clients mit einem RST beantwortet, schlägt die Verbindung fehl, weil kein Prozess am anderen Ende lauscht. Fehlercode `ECONNREFUSED`.
- Wird ein SYN des Clients durch die ICMP-Nachricht `EHOSTUNREACH` oder `ENETUNREACH` beantwortet, so ist es wahrscheinlich, daß ein Fehler auf der Kommunikationsstrecke zwischen Client und Server vorliegt und wird in der Regel als leichter Fehler (*soft error*) eingestuft, weil es nach relativ kurzer Zeit eine neue Route zum Ziel geben wird. Dennoch schlägt die Verbindung fehl, wenn das nicht innerhalb eines festgelegten Zeitraums geschieht. Erst dann wird der Fehler an den Client-Prozess weitergeleitet.
- POSIX sagt, daß ein Aufruf von `connect`, dessen SYN nicht bestätigt wurde nach einer unbestimmten Zeitspanne den Fehler `ETIMEDOUT` liefert. Je nach Plattform beträgt die Spanne zwischen 60 und 120 Sekunden. In vielen Fällen kann der Parameter angepaßt werden.

13.2.2 TCP-Server und Clients entwickeln

Abschließend wollen wir einen kleinen Server samt passendem Client entwickeln. Der einfachste denkbare Server ist ein Zeit-Server, auch *Daytime-Server* genannt. Er beantwortet Client-Anfragen mit der aktuellen Systemzeit. Die Kommunikation zwischen den beiden Prozessen ist in Abbildung 13.3 abgebildet.

Die Abbildung ist möglicherweise etwas irreführend, denn natürlich liest der Client nicht direkt vom Socket-Endpunkt des Servers und der Server schreibt nicht in den Endpunkt des Clients. Beide schreiben und lesen in ihre eigenen Sockets.

Später entwickeln wir einen Echo-Server, der alle Eingaben des Clients wieder an ihn zurückgibt. Auch wenn das gar nicht so schwer klingt, gibt es in diesem Zusammenhang doch einige Punkte, die es zuvor zu klären gilt.

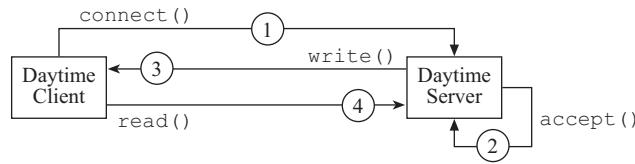


Abbildung 13.3: Kommunikation zwischen Daytime-Client und -Server.

13.2.2.1 Ein Daytime-Server

Im vorangegangenen Abschnitt haben wir uns mit den wichtigsten Funktionen zur Entwicklung von TCP-Clients und -Servern vertraut gemacht, so daß wir uns gleich auf echten Code stürzen wollen.

Listing 13.2: Ein einfacher Daytime-Server

```

1 #include <time.h>
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     int             socketfd, acceptfd;
6     struct sockaddr_in srvaddr;
7     char            buf[BUFSIZ];
8
9     socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
10
11    /* clear socket address structure */
12    memset((char *)&srvaddr, 0, sizeof(struct sockaddr_in));
13
14    /* fill in socket address structure */
15    srvaddr.sin_family = AF_INET;
16    srvaddr.sin_addr.s_addr = INADDR_ANY;
17    srvaddr.sin_port = htons(13); /* network byte order! */
18
19    bind_ex(socketfd, (struct sockaddr *)&srvaddr, sizeof(srvaddr));
20    listen_ex(socketfd, -1); /* force implementation default */
21
22    while (TRUE) {
23        acceptfd = accept_ex(socketfd, NULL, NULL);
24        time_t t = time(NULL);
25        snprintf(buf, sizeof(buf), "%.24s\r\n", ctime(&t));
26        write_ex(acceptfd, buf, strlen(buf));
27        close_ex(acceptfd);
28    }
29}

```

Listing 13.2: xcode/tcpdaytimesrv1.c - Ein einfacher Daytime-Server.

Gehen wir den Code Zeile für Zeile durch:

8-10

Deklaration der benötigten Datenstrukturen. Wir brauchen zwei Socket Descriptoren: einen für den Zustand LISTEN und einen anderen für ESTABLISHED. Natürlich darf die Adress-Struktur nicht fehlen und schließlich brauchen wir auch einen Buffer, den wir mit der aktuellen Zeit an den Client als Antwort senden.

14

Immer der erste Schritt: einen Socket erstellen. Den benötigen wir ebenso wie die Adressstruktur für fast alle Funktionen. Als Protokollfamilie haben wir natürlich IPv4 ausgewählt (IPv6 (`AF_INET`

wäre auch in Ordnung) und als Transportprotokoll TCP. `SOCK_STREAM` tut Not, schließlich wollen TCP verwenden. `AF_INET6` setzt allerdings ein funktionierendes IPv6-Netzwerk voraus, dessen Existenz weniger wahrscheinlich ist als die eines einsatzfähigen IPv4-Netzwerkes).

17

Vor Verwendung der Addressstruktur müssen wir sie vollständig mit 0 initialisieren. Dazu verwenden wir die Funktion `memset`, die als ersten Parameter das zu manipulierende Objekt, als zweiten Parameter den initialen Wert und als letzten Parameter die Anzahl der zu schreibenden Bytes erfordert.

20-22

Einrichtung der Adressstruktur unseres Servers. Neben der Adressfamilie, müssen wir auch die IP-Adresse und den Port hinterlegen. Da wir es dem Kernel überlassen wollen, welche IP-Adresse für den Server genutzt wird, hinterlegen wir für den Member `sin_addr` einfach `AF_INET_ANY`. Da `AF_INET_ANY` immer in Network Byte Order vorliegt, brauchen wir nur den Port nach NBO konvertieren.

24

Durch den Aufruf von `bind` fordern wir den Kernel auf, uns eine IP-Adresse zuzuweisen, die wir zur Kommunikation einsetzen werden.

25

`listen` erhält den Wert 5 als `backlog`. Das ist für unsere Anwendung durchaus genug, obwohl das zugrundeliegende System möglicherweise viel mehr unterstützt (Abschnitt 13.2.1.4 enthält mehr Informationen zu diesem Thema).

27-31

Kernstück eines jeden Servers ist die Schleife, in der die eingehenden Verbindungen aus der Warteschlange geholt werden. Das erledigt `accept` für uns. Damit erhalten wir einen neuen Socket Descriptor, der nur für die Dauer der Verbindung existiert und anschließend verworfen wird. Da wir keine Informationen über den Client abfragen möchten sind die beiden Value Result Arguments `NULL` (Abschnitt 13.2.1.4). Bevor wir letztendlich dem Client antworten, bereiten wir noch die Zeitangabe in einen für Menschen lesbaren String auf und speichern ihn mit `snprintf` im Buffer `buf`. Mit einem einfachen `write`-Kommando beantworten wie die Client-Anfrage und schließen die Kommunikation durch ein `close` des verbundenen File Descriptors ab.

Der Verbindungsaufbau wird vom Client initiiert. Der Server befindet sich nach dem Start im Zustand `LISTEN` und blockiert mit dem Aufruf von `accept`, bis ein Client eine Verbindung wünscht. Sendet ein Client sein `SYN`, sorgt der Kernel dafür, daß eine Verbindung etabliert und der Funktion als neuer Socket Descriptor aus der Warteschlange zugeführt wird. Sobald wir dem Client geantwortet haben, wird die Verbindung getrennt, indem der Server sein `FIN` sendet, das vom Client bestätigt wird, der ebenfalls sein `FIN` sendet.

Wie wir sehen können, machen wir intensiven Gebrauch von unseren Wrapper-Funktionen. □

13.2.2.2 Ein Daytime-Client

Nun, da wir einen Server erstellt haben, benötigen wir noch einen passenden Client. Prinzipiell gehen wir ähnlich wie bei der Entwicklung des Servers vor:

1. Wir konfigurieren eine Socket Address-Struktur, die wir für die Verbindung mit dem Server konfigurieren, nicht für uns selbst, denn das überlassen wir unserem System.
2. Wir erstellen einen Socket, mit den gewünschten Optionen. Wir bleiben noch bei `IPPROTO_TCP`.
3. Im letzten Schritt verbinden wir uns mit dem Server, indem wir `connect` Aufrufen.

Listing 13.3: Ein Daytime-Client

```

1 #include <unistd.h>
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     int             socketfd, bytes_read;
6     struct sockaddr_in srvaddr;
7     char            buf [BUFSIZ];
8     char            *basename = basename_ex(argv[0]);
9
10    if (argc != 2)
11        err_fatal("Usage: %s <ipaddress>\n", basename);
12
13    memset(&srvaddr, 0, sizeof(struct sockaddr_in));
14    srvaddr.sin_family = AF_INET;
15    srvaddr.sin_port = htons(13);
16
17    //inet_pton_ex(AF_INET, argv[1], &srvaddr.sin_addr);
18    set_sinaddr(srvaddr.sin_family, argv[1], &srvaddr.sin_addr);
19
20    socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
21    connect_ex(socketfd, (SA_PTR)&srvaddr, sizeof(srvaddr));
22
23    while (read_ex(socketfd, buf, sizeof(buf)) > 0)
24        fputs_ex(buf, stderr);
25
26    close_ex(socketfd);
27 }

```

Listing 13.3: xcode/tcpdaytimecli1.c - Ein Daytime-Client.

Durch den Aufruf von `connect` veranlassen wir den Server eine ausstehende Verbindung aus der Warteschlange zu holen und den eigenen noch unvollständigen Socket in einen fertigen zu verwandeln. Sobald `accept` zurückkehrt schreibt der Server die aktuelle Systemzeit als lesbaren String in seinen Socket. Auf Client-Seite blockiert der Aufruf von `read_ex` bis Daten am Socket eintreffen. Um sicherzustellen, daß wir auch alle Daten erhalten, verschachteln wir `read_ex` in einer `while`-Schleife:

```

while (read_ex(socketfd, buf, sizeof(buf)) > 0)
    fputs_ex(buf, stderr);

```

Die Funktion `read_ex` ist durchaus einen genaueren Blick wert:

```

1 ssize_t read_ex(int fd, void *buf, size_t size) {
2     size_t bytesleft = size, bytesread;
3     char  *buffer = buf;
4
5     while (bytesleft > 0) {
6         if ((bytesread = read(fd, buffer, bytesleft)) < 0)
7             /* check if we have been interrupted and force retry */
8             if (errno == EINTR)
9                 bytesread = 0;
10            else /* can't recover from error. */
11                err_fatal("read_ex() failed.");
12            else if (bytesread == 0)
13                break; /* all bytes read */
14
15            bytesleft -= bytesread;
16            buffer += bytesread;
17        }
18
19    return (size - bytesleft);
20 }

```

Da es immer mal wieder vorkommen kann, daß ein Lesevorgang unterbrochen wird, müssen wir diesen Fall abfangen. Das erledigen wir, indem `errno` nach `EINTR` befragt wird. Nur wenn wir die Anzahl der gelesenen Bytes wieder auf 0 zurücksetzen, können wir garantieren, daß wir alle Daten erhalten und nicht das eine oder andere Byte verloren geht. Das gleiche Prinzip haben wir kennengelernt, als wir eine einfache Implementierung des `cp`-Befehls in Listing 3.4 aufgewertet haben. Selbstverständlich ist der Einsatz der Funktion nicht nur auf Sockets beschränkt. □

13.2.2.3 Ein Echo-Server

Zugegeben, so ein Daytime-Server und ein passender Client sind sehr einfache Beispiele für ein komplexes Thema wie die Netzwerkprogrammierung. Sie eignen sich aber sehr gut, um einen leichten Zugang zur Materie zu erlangen. In diesem und dem nächsten Abschnitt schauen wir uns an, wie wir eine Zweierkommunikation herstellen (nämlich einen Echo-Server) und einen Server entwickeln, der mehrere Anfragen gleichzeitig verarbeiten kann.

Die Kommunikation zwischen einem Echo-Client und -Server läuft wie ab wie in Abbildung 13.4 dargestellt wird.

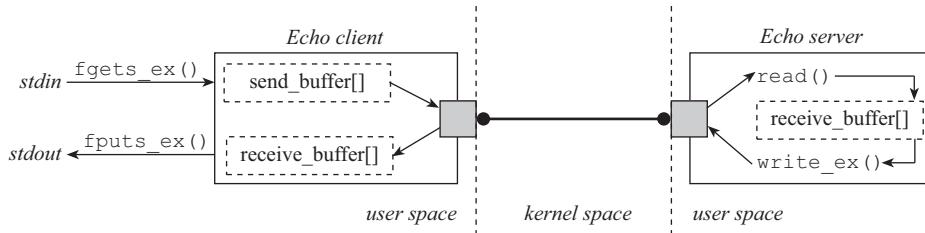


Abbildung 13.4: Kommunikation zwischen Echo-Client und Echo-Server.

Praktisch sieht die Anwendung des Echo-Clients (Listing 13.5) etwa so aus; zuerst starten wir den Server auf Host `obsd`:

```
obsd % ./tcpechosrv1 & // Server starten
```

Anschließend verbinden wir uns mit dem Echo-Server (Listing 13.4), indem dem Client die IP-Adresse des Servers mitgeteilt wird:

```
xp101 % ./tcpechocli1 192.168.1.2 // obsd hat diese IP-Adresse
Ein Echo kehrt immer zurueck.
Ein Echo kehrt immer zurueck.
QUIT
%
```

Der Server lauscht auf Port 9191 auf Verbindungen. Sobald der Client gestartet wird, holt `accept` eine Verbindung aus der Warteschlange und der Server blockiert erneut mit dem Aufruf von `read`. Starten wir nun den Client, blockiert er in `fgets_ex` und wartet auf Eingaben. Jede Zeile wird mit `write_ex` in den aktuellen Socket Descriptor geschrieben. Auf Server-Seite wacht `read` wieder auf, speichert die Eingaben in einem Buffer um sie gleich wieder mit `write_ex` zurück schreiben zu lassen, sobald sie vollständig eingetroffen sind. Im letzten Schritt schreibt der Client die Rückgabe nach `stdout`.

Listing 13.4: Der TCP Echo Server

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     int                 sockfd, acceptfd;
5     struct sockaddr_in server, client;
6     ssize_t             bytesread;

```

```

7      char          receive_buf [LINESIZ], *str_quit = "QUIT";
8
9      memset((char *)&server, 0, sizeof(struct sockaddr_in));
10
11     server.sin_family = AF_INET;
12     server.sin_port = htons(9191);
13     server.sin_addr.s_addr = INADDR_ANY;
14
15     socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
16     bind_ex(socketfd, (SA_PTR)&server, sizeof(server));
17     listen_ex(socketfd, -1);
18
19     while (TRUE) {
20         socklen_t clilen = sizeof(client);
21         acceptfd = accept_ex(socketfd, (SA_PTR)&client, &clilen);
22
23         /* read client input and write back to client */
24         while (TRUE) {
25             bytesread = read(acceptfd, receive_buf, LINESIZ);
26
27             if (strstr(receive_buf, str_quit) != NULL) {
28                 close(acceptfd);
29                 close(socketfd);
30                 exit(0);
31             }
32
33             if (bytesread < 0 && !(errno == EINTR))
34                 err_fatal("ERROR: could not read input");
35
36             write_ex(acceptfd, receive_buf, bytesread);
37         }
38
39         close(acceptfd);
40     }
41
42     close(socketfd);
43 }
```

Listing 13.4: xcode/tcpecho/tcpechosrv1.c - Der TCP Echo Server.

Hauptaugenmerk legen wir natürlich wieder auf die `while`-Schleife, in der wir Verbindungen abholen und die ganze Arbeit erledigen. Wir sind zwar nicht an der Client-Information interessiert, wollen hier aber zeigen, daß `accept` eine der wenigen Funktionen ist, die die Länge der Socket Address-Struktur als Value-Result-Argument erwartet. Ganz klar, die letzten beiden Argumente hätten auch `NULL` sein können. Auf die Arbeitsweise des Servers und des Clients sind wir schon eingegangen. Erwähnenswert ist nur, daß wir die Verbindung mit dem Server von der Client-Seite aus durch das `QUIT`-Kommando beenden können.

Die ganze Sache hat einen Schönheitsfehler, der auch schon im Daytime-Server vorhanden war. Wird der Server beendet, aber sofort wieder neu gestartet, so wird uns die unliebsame Nachricht *Connection refused: address already in use* begegnen. Dem können wir entgegenwirken, indem wir die Socket-Option `SO_REUSEADDR` verwenden, die es uns erlaubt, eine lokale Adresse erneut zu verwenden. Sie sollte vor dem Aufruf von `bind(2)` angewendet werden, da die Auswertung des Adress-/Portpaars hier erfolgt. Nach dem Aufruf von `bind(2)` können wir nur noch `ioctl(2)` verwenden, um den Socket mit dieser Option zu versehen.

Praktisch sähe das dann etwa so aus:

```

int s, opt = 1;
struct sockaddr_in sa;

s = socket_ex(AF_INET, SOCK_STREAM, 0);
```

```

if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt)) < 0)
    err_fatal("setsockopt() failed");

/* set up socket address structure here */
bind_ex(s, (SA_PTR)&sa, sizeof(sa));

```

Listing 13.5: Der TCP Echo Client

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     int                 socketfd;
5     struct sockaddr_in server;
6     char                *basename = basename_ex(argv[0]);
7     char                *str_quit = "QUIT";
8     char                send_buffer[LINESIZ], receive_buffer[LINESIZ];
9
10    if (argc != 2)
11        err_fatal("Usage: %s <server>\n", basename);
12
13    memset((char *)&server, 0, sizeof(struct sockaddr_in));
14
15    server.sin_family = AF_INET;
16    server.sin_port = htons(9191);
17
18    socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
19    inet_pton_ex(AF_INET, argv[1], &server.sin_addr);
20    connect_ex(socketfd, (struct sockaddr *)&server, sizeof(server));
21    signal_ex(SIGPIPE, SIG_IGN);
22
23    /* process input and server response */
24    while (fgets_ex(send_buffer, LINESIZ, stdin) != NULL) {
25        if (strstr(send_buffer, str_quit) != NULL) { /* got QUIT? */
26            close(socketfd);
27            exit(0);
28        }
29
30        write_ex(socketfd, send_buffer, strlen(send_buffer));
31
32        if (readline_ex(socketfd, receive_buffer, LINESIZ) == 0)
33            err_fatal("readline_ex() failed.\n");
34
35        fputs_ex(receive_buffer, stdout);
36    }
37
38    close(socketfd);
39}

```

Listing 13.5: *xcode/tcpecho/tcpechocli1.c - Der TCP Echo Client.*

Der Client wird mit dem QUIT-Kommando beendet. □

13.2.2.4 Mehrere Clients gleichzeitig verarbeiten

Die bisher besprochenen Server haben einen entscheidenden Nachteil: sie können immer nur einen Client zu einem bestimmten Zeitpunkt bedienen. Jeder Client, der eine Verbindung anfordert, landet zwangsläufig in der Warteschlange ausstehender Verbindungen und erst wenn ein Client die Verbindung schließt, kommt der nächste dran. Der Daytime-Client ist mit dem Server nur sehr kurz (meist nicht mehr als ein

paar Millisekunden) verbunden, aber im Fall eines Echo-Servers kann die Verbindungszeit unbegrenzt sein. Für eine professionelle Anwendung mehr als unbefriedigend. Schauen wir uns nun an, wie wir Server entwickeln können, die mit diesem Nachteil aufräumen. Eine Möglichkeit sind Threads, eine andere sieht die Verwendung von fork vor. Da letztere Variante momentan die einfachere ist, wollen wir damit beginnen. Listing 13.7 zeigt den Code für einen TCP Echo Server mit Threads.

Um es gleich vorweg zu nehmen: der Clou liegt darin, daß wir in der `while`-Schleife des Servers einen Child-Prozess erzeugen, der die nächste Verbindung abarbeitet. Code-technisch sieht das folgendermaßen aus:

```

while (TRUE) {
    socklen_t clilen = sizeof(client);
    acceptfd = accept_ex(socketfd, (struct sockaddr *)&client, &clilen);

    if ((pid = fork()) == 0) {
        close_ex(socketfd);
        process_request(acceptfd, receive_buf, LINESIZ);
        close_ex(acceptfd);
        exit(0);
    } else if (pid < 0) {
        err_fatal("fork() failed in server");
    }

    close_ex(socketfd);
}

```

Wir holen eine Verbindung aus der Warteschlange, erzeugen einen Child, schließen im Child den alten, geerbten Socket Descriptor, verarbeiten den Client (wir haben diese Funktionalität in die Funktion `process_request` verlagert, um den Code übersichtlicher zu gestalten), schließen den verbundenen Socket Descriptor und beenden den Child-Prozess anschließend.

Der Code für den neuen Server ist in Listing 13.6 dargestellt.

Listing 13.6: Verbesserter TCP Echo Server

```

1 #include "header.h"
2
3 void process_request(int socketfd, char *buf, size_t size);
4 void handler(int signum);
5
6 int main(int argc, char **argv) {
7     int                 socketfd, acceptfd, opt = 1;
8     struct sockaddr_in server, client;
9     char                receive_buf[LINESIZ];
10    pid_t               pid;
11
12    memset((char *)&server, 0, sizeof(struct sockaddr_in));
13
14    server.sin_family = AF_INET;
15    server.sin_port = htons(9191);
16    server.sin_addr.s_addr = INADDR_ANY;
17
18    socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
19
20    if (setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR,
21                   (char *)&opt, sizeof(opt)) < 0)
22        err_fatal("setsockopt() failed");
23
24    bind_ex(socketfd, (SA_PTR)&server, sizeof(server));
25    listen_ex(socketfd, -1);
26    signal_ex(SIGCHLD, handler);

```

```

27     while (TRUE) {
28         printf("Server [%d] up and running...\n", getpid());
29         socklen_t clilen = sizeof(client); /* value-result argument */
30         acceptfd = accept_ex(socketfd, (SA_PTR)&client, &clilen);
31
32         if ((pid = fork()) == 0) {
33             printf("\t*** now serving...\n");
34             close_ex(socketfd);
35             process_request(acceptfd, receive_buf, LINESIZ);
36             exit(0); /* terminate child process */
37         } else if (pid < 0) {
38             err_fatal("fork() failed in server");
39         }
40
41         close_ex(acceptfd);
42     }
43 }
44
45 void process_request(int socketfd, char *buf, size_t size) {
46     ssize_t bytesread;
47
48     while ((bytesread = read_ex(socketfd, buf, LINESIZ)) > 0) {
49         if (strstr(buf, "QUIT") != NULL)
50             return;
51
52         write_ex(socketfd, buf, bytesread);
53     }
54 }
55
56 void handler(int signum) {
57     pid_t pid;
58     int status;
59
60     while ((pid = waitpid(WANYCHILD, &status, WNOHANG)) > 0)
61         ;
62
63     return; /* wake up */
64 }

```

Listing 13.6: `xcode/tcpecho/tcpechosrv2.c` - Mehrere Clients mit Hilfe des `fork`-Systemaufrufs bedienen.

Dem aufmerksamen Leser stellt sich sofort die Frage, warum wir erst den „alten“ Socket Descriptor und dann den verbundenen Socket Descriptor schließen, schließlich würden wir damit die Verbindung zum Client beenden (denn `close` setzt ein FIN ab, wenn sie mit einem Socket aufgerufen wird). Wir wollen nicht vergessen, daß alle File Descriptoren des Parents mit `fork` vererbt werden. Indem wir also vor dem Aufruf von `fork` den verbundenen File Descriptor anfordern steht er uns auch im Child zur Verfügung; das gleiche trifft auch für den unverbundenen Descriptor zu. Des Weiteren führt der Kernel Buch über die Anzahl der offenen File Descriptoren eines Sockets, den sogenannten *Reference Count*. Aus diesem Grund müssen wir dafür sorgen, daß die Descriptoren geschlossen werden, sonst würde der Parent nur den Reference Count dekrementieren, weiter nichts.

Immer wenn wir `forken`, müssen wir auch einen Signal Handler registrieren, der `SIGCHLD` behandelt, damit wir gezielt auf unsere Childs warten können. Beachten Sie aber, daß `wait(2)` allein nicht genügt, denn Signale werden nicht aufgezeichnet, so daß wir immer nur ein Child Prozess beenden würden, auch wenn ein Client mehrere Verbindungen hergestellt hat und wir mehrmals im Server geforkt haben. Der Client wird beendet und sendet über alle Verbindungen ein FIN, so daß die Childs im Server beendet werden und ein `SIGCHLD` generiert wird, von dem aber nur eines von `wait` aber alle von `waitpid` entdeckt werden. Mehr Informationen finden Sie im Abschnitt 7.3.2.

Prozesse sind verglichen mit Threads relativ schwerfällig. Ihre Erzeugung und Zerstörung benötigt relativ viel Zeit. Des Weiteren kommt noch der Overhead der Speicherverwaltung hinzu. Für kleinere Serveran-

wendungen eignen sich Threads besser. Listing 13.7 zeigt unseren TCP Echo Server in einer Version mit mehreren Threads.

Listing 13.7: Multithreaded TCP Echo Server

```

1 #include "header.h"
2
3 void *process_request(void *arg);
4
5 int main(void) {
6     int             socketfd, opt = 1;
7     int             server_len;
8     struct sockaddr_in server;
9
10    memset((char *)&server, 0, sizeof(struct sockaddr_in));
11
12    server.sin_family = AF_INET;
13    server.sin_port = htons(9191);
14    server.sin_addr.s_addr = INADDR_ANY;
15
16    socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
17
18    setsockopt_ex(socketfd, SOL_SOCKET, SO_REUSEADDR,
19                  (char *)&opt, sizeof(opt));
20
21    bind_ex(socketfd, (SA_PTR)&server, sizeof(server));
22    listen_ex(socketfd, -1);
23
24    while (TRUE) {
25        struct sockaddr_in client;
26        int             acceptfd;
27        int             client_len = sizeof(client);
28        pthread_t       worker;
29
30        acceptfd = accept_ex(socketfd, (SA_PTR)&client, &client_len);
31        printf("Peer: %s\n", inet_ntop_ex((SA_PTR)&client, client_len));
32        pthread_create_ex(&worker, NULL, &process_request, (void *)acceptfd);
33        pthread_detach(worker);
34    }
35
36    return 0;
37 }
38
39 void *process_request(void *arg) {
40     char            buf[MAX_BUF];
41     int             bytes_read;
42     int             socketfd = *((int *)arg);
43
44     while ((bytes_read = read_ex(socketfd, buf, LINESIZ)) > 0 ) {
45         if (strncmp(buf, "QUIT", 4) == 0) {
46             close(socketfd);
47             pthread_exit(NULL);
48         }
49
50         write_ex(socketfd, buf, bytes_read);
51     }
52
53     close(socketfd);
54     return (NULL);
55 }
```

Listing 13.7: xcode/tcpecho/tcpechosrv3.c - Mehrere Clients mit Hilfe von Threads bedienen.

Statt `fork(2)` erzeugen wir nun einen Thread für jeden verbundenen Socket auf. Als Argumente übergeben wir einen Zeiger auf eine Variable vom Typ `pthread_t`, eine Zeiger auf eine Funktion, die den Thread antreibt (in unserem Fall natürlich `process_request`), und den Socket Descriptor der Verbindung. Auf Thread-Attribute können wir verzichten, so daß `NULL` als Argument ausreicht. Mehr Informationen zum Thema Threads erhalten Sie in Kapitel 11 *POSIX-Threads*.

Der Funktion `process_request` übergeben wir den verbundenen Socket Descriptor, der als `void`-Zeiger kodiert ist, so daß wir ihn in einen Integer casten müssen. Der Rest der Funktion ist wie gehabt. □

13.2.3 Das User Datagram Protocol

Im Gegensatz zu TCP ist UDP verbindungslos, so daß kein Three-Way-Handshake durchgeführt werden muß, bevor Datagramme abgesetzt werden können. Die wichtigsten Eigenschaften von UDP haben wir bereits in Abschnitt 12.3.1 kennengelernt.

Der verbindungslose Charakter bedeutet für uns, daß unsere Aufrufsequenz für das Übermitteln von Daten im Server und zum Empfang von Daten im Client anders ist als mit TCP:

1. `socket` - Socket erstellen, diesmal mit der Option `SOCK_DGRAM` für UDP.
2. `bind` - Weist dem Socket eine lokale Adresse zu, die sich stets aus einem Paar von IP-Adresse (IPv4 oder IPv6) und einer Portnummer (je nach Anwendung *well-known*) zusammensetzt.

Das war es schon. Anschließend blockiert der Server in einem Aufruf beispielsweise von `recvfrom(3)` bis ein Client Daten übermittelt.

Für Clients funktioniert das ganz ähnlich:

1. `socket` - Socket erstellen, natürlich mit der Option `SOCK_DGRAM` für UDP.
2. `connect` - Optional kann `connect(2)` aufgerufen werden, doch anders als der Name vermuten läßt wird hier keine Verbindung zur Gegenstelle aufgebaut, sondern lediglich eine Assoziation zwischen dem Socket und der Gegenstelle hergestellt.

Der Client seinerseits kann Datagramme beispielsweise mit `sendto(3)` absetzen und anschließend in `recvfrom(3)` blockieren, bis er eine Antwort vom Server erhält.

13.2.4 UDP-Server und Clients entwickeln

Die Entscheidung wann TCP oder UDP zu bevorzugen ist von hauptsächlich von zwei Faktoren abhängig: Anzahl der Pakete, die notwendig sind, um Daten auszutauschen und die Art des Datenaustauschs. Anwendungen, die nur einfach nur Anfragen beantworten, wie beispielsweise DNS-Server, fahren mit UDP besser, da nur zwei Pakete notwendig sind, um eine Anfrage zu senden und eine Antwort zu empfangen. TCP wäre mit seinem Three-Way-Handshake und dem koordinierten Verbindungsabbau überdimensioniert. FTP beispielsweise fährt mit TCP aus verschiedenen Gründen besser, denn meist dauert die Datenübertragung länger, es sind nur wenige Verbindungen nötig und die Kontrollmechanismen von TCP, wie Congestion Avoidance, fenster-basierte Flußkontrolle und die erneute Übertragung verlorengegangener Pakete sind ein großer Vorteil. Diese Mechanismen in die Applikation zu integrieren nur um ein paar Bytes beim Verbindungsauflauf und Abbau zu sparen macht keinen Sinn.

In diesem Abschnitt entwickeln wir einen UDP-basierten Server und Client. Dazu führen wir die drei neuen Funktionen `recvfrom(3)`, `sendto(3)` und `connect(2)` ein, von denen die ersten beiden auch mit TCP verwendet werden können, was aber keinen Sinn ergibt. Wie die Namen schon sagen, lesen wir mit `recvfrom` Daten und schreiben mit `sendto(3)`:

```
#include <sys/socket.h>

ssize_t recvfrom(int socket, void *buffer, size_t length, int flags,
                 struct sockaddr *address, socklen_t *address_len);
```

Rückgabewert: Anzahl der gelesenen Bytes oder -1 bei Fehler

socket

Socket Descriptor von dem wir die empfangenen Daten lesen möchten.

buffer

Zeiger auf einen Puffer in dem die gelesenen Daten gespeichert werden.

length

Größe des Puffers auf den **buffer** zeigt.

flags

Zeigt an, wie und welche Art von Daten wir empfangen möchten (siehe weiter unten). Der Wert kann 0 sein.

address

Zeiger auf eine **sockaddr_in**-Struktur, in die Informationen über die Quelle des Datenempfangs geschrieben werden. Darf NULL sein.

address_len

Zeiger auf eine Variable vom Typ **socklen_t** in der die Länge der befüllten **sockaddr_in**-Struktur abgelegt wird. Muß NULL sein, wenn **address** auf NULL gesetzt wird.

Ähnlich wie **read(2)** lesen wird Daten von einem Descriptor in einen Buffer, dessen Länge an die Funktion übergeben wird. Die Flags sind für uns momentan nicht interessant, haben aber folgende Bedeutung:

MSG_PEEK

Erlaubt eine Vorschau auf die Daten, aber Empfangspuffer wird nicht beeinflußt. Ein zweiter Aufruf von **recvfrom** erlaubt die Abfrage aller Daten aus dem Buffer.

MSG_OOB

Fragt spezifische Out-Of-Band-Daten ab (OOB). Die Bedeutung der OOB-Daten ist protokollabhängig.

MSG_WAITALL

Veranlaßt **recvfrom** solange zu blockieren, bis alle Daten empfangen wurden. Nur wenn ein Signal, **MSG_PEEK** angegeben wurde oder ein Fehler auftrat werden weniger Daten geliefert.

Solange keine Daten empfangen werden und **O_NONBLOCK** nicht für den Socket Descriptor gesetzt ist, blockiert **recvfrom**. Das Verhalten kennen wir von **read(2)**.

```
#include <sys/socket.h>

ssize_t sendto(int socket, const void *message, size_t length, int flags,
               const struct sockaddr *dest_addr, socklen_t dest_len);
```

Rückgabewert: Anzahl der gesendeten Bytes, -1 bei Fehler

socket

Socket Descriptor in den wir Daten schreiben möchten.

message

Zeiger auf einen Puffer in dem die zu sendenden Daten enthalten sind.

length

Größe des Puffers auf den **message** zeigt.

flags

Zeigt an, wie und welche Art von Daten wir senden möchten (siehe weiter oben). Der Wert kann 0 sein.

address

Zeiger auf eine `sockaddr_in`-Struktur der Gegenstelle.

address_len

Länge der Adressstruktur auf die `address` zeigt.

Liefert `sendto(3)` einen Fehler (-1) zurück, so wird uns nur angezeigt, daß ein lokales Problem aufgetreten ist, aber nicht, daß ein Problem mit der Zustellung des Datagrams festgestellt wurde. Diese Funktionalität liefert ein verbindungsloses Protokoll nicht.

Die Funktion `connect(2)` ist in der Lage einen UDP-Socket zu verbinden, wobei das Ergebnis nicht wie eine TCP-Verbindung zu interpretieren ist. Wir verwandeln damit lediglich einen nicht verbundenen Socket in einen verbundenen Socket. Unter *verbunden* verstehen wir nur die Tatsache, daß der Funktion eine Adressstruktur übergeben wird, die unsere Gegenstelle genau identifiziert, so daß wir quasi an sie gebunden sind.

Mit `connect(2)` sind wir in der Lage zu erkennen, daß die Gegenseite nicht erreichbar ist, beispielsweise dann, wenn gar kein Echo-Server läuft (ein asynchroner Fehler). Das wird möglich, weil der Kernel sofort prüft, ob ein Fehler aufgetreten ist, da er die an `connect(2)` übergebene Adressstruktur verwendet. Allerdings tritt dieser Fehler erst auf, wenn wir versuchen in einen verbundenen Socket zu schreiben.

Wie wir in unseren TCP-Clients aus vorangegangenen Beispielen gesehen haben, verwenden wir `connect(2)` auch mit verbindungsorientierten Transportprotokollen. In diesem Fall richtet die Socket-Schicht¹ eine Verbindungsanfrage an die Protokollsicht, die einen Three-Way-Handshake versucht und anschließend die Socket-Schicht über den Erfolg informiert und gleichzeitig einen evtl. schlafenden Prozess aufweckt, damit der `connect(2)`-Aufruf abgeschlossen werden kann. Bei UDP kehrt `connect(2)` sofort wieder zurück, da keine Verbindung aufgebaut werden muß.

Im Zusammenhang mit verbundenen Sockets verwenden wir die Funktionen `read(2)`, `recv(3)`, `recvmsg(3)`, `send(3)` und `write(2)`, aber nicht `recvfrom(3)` oder `sendto(3)`.

13.2.4.1 Der UDP Echo-Server

Die Funktionsweise des UDP-Servers ist identisch mit der TCP-Version: Wir lesen die Daten des Clients ein und schreiben sie wieder zurück. Abbildung 13.5 illustriert, welche Funktionen wir an welchem Endpunkt aufrufen.

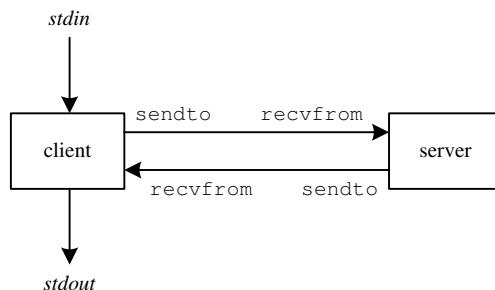


Abbildung 13.5: *udpclientserver*

Auffällig ist die Kürze von Listing 13.8. Der im Protokoll quasi nicht vorhandene Overhead von UDP schlägt sich unmittelbar auch auf den Quelltext nieder. Listing 13.8 zeigt den vollständigen Servercode.

Listing 13.8: UDP-Version des Echo Servers

¹Die Socket-Schicht (*socket layer*) ist keine Schicht im TCP/IP-Stack. Vielmehr handelt es sich hier um eine softwaretechnische Abstraktion zur flexibleren Behandlung von Systemaufrufen.

```

1 #include "header.h"
2
3 void process_request(int, SA_PTR, socklen_t);
4
5 int main(int argc, char **argv) {
6     int             socketfd;
7     struct sockaddr_in server, client;
8
9     memset((char *)&server, 0, sizeof(struct sockaddr_in));
10
11    socketfd = socket_ex(AF_INET, SOCK_DGRAM, 0);
12    server.sin_family      = AF_INET;
13    server.sin_addr.s_addr = htonl(INADDR_ANY);
14    server.sin_port        = htons(9191);
15
16    bind_ex(socketfd, (SA_PTR)&server, sizeof(server));
17    process_request(socketfd, (SA_PTR)&client, sizeof(client));
18 }
19
20 void process_request(int socketfd, SA_PTR client, socklen_t socklen) {
21     int          bytes_read;
22     socklen_t    len = socklen;
23     char         buf[LINESIZ];
24
25     while (TRUE) {
26         bytes_read = recvfrom_ex(socketfd, buf, LINESIZ, 0, client, &len);
27         sendto_ex(socketfd, buf, bytes_read, 0, client, len);
28     }
29 }
```

Listing 13.8: xcode/udpechosrv1.c - UDP-Version des Echo Servers.

Der Server muß nichts weiter machen, als einen Socket Descriptor mit `socket(2)` anfordern und `bind(2)` aufzurufen. Anschließend blockiert der Server in `recvfrom(3)`, wo er auf Daten wartet. Sobald Daten verfügbar sind, kommt `sendto(3)` zum Einsatz.

Bemerkenswert ist, daß die Funktion `process_client` nicht zurückkehrt, wie wir es vom TCP Echo Server gewohnt sind. Tatsächlich sind die meisten UDP-Server iterativ, was bedeutet, daß nur ein Prozess für die Behandlung von Clients zuständig ist, während die meisten TCP-Server mehrere Prozesse oder Threads verwenden.

13.2.4.2 Der UDP Echo-Client

Im Gegensatz zum UDP-Server gibt es im Client nur wenige Änderungen. Statt `connect(2)` rufen wir nun `sendto(3)` auf. Der Client blockiert in `fgets_ex`, um Eingaben von `stdin` zu lesen. Sobald Daten verfügbar sind, schreiben wir in den Socket und blockieren nun wieder in `recvfrom(3)`.

Listing 13.9: UDP-Version des Echo Clients

```

1 #include "header.h"
2
3 void process_reply(FILE *, int, const SA_PTR, socklen_t);
4
5 int main(int argc, char **argv) {
6     char             *basename = basename_ex(argv[0]);
7     int              socket;
8     struct sockaddr_in server;
9
10    if (argc != 2)
```

```

11     err_fatal("Usage: %s <server-ip-address>\n", basename);
12
13     memset((char *)&server, 0, sizeof(struct sockaddr_in));
14
15     server.sin_family = AF_INET;
16     server.sin_port = htons(9191);
17     inet_pton_ex(AF_INET, argv[1], &server.sin_addr);
18     socket = socket_ex(AF_INET, SOCK_DGRAM, 0);
19
20     process_reply(stdin, socket, (SA_PTR)&server, sizeof(server));
21
22     return (0);
23 }
24
25 void process_reply(FILE *str, int socket, const SA_PTR srv, socklen_t socklen) {
26     int bytes_read;
27     char in_buffer[MAX_DGRAM_SIZ], out_buffer[MAX_DGRAM_SIZ];
28     socklen_t peer_len;
29     struct sockaddr *peer;
30
31     peer = (SA_PTR)malloc(sizeof(socklen));
32
33     while (fgets_ex(in_buffer, MAX_DGRAM_SIZ, str) != NULL) {
34         sendto_ex(socket, in_buffer, MAX_DGRAM_SIZ, 0, srv, socklen);
35         peer_len = socklen;
36         bytes_read = recvfrom_ex(socket, out_buffer,
37                               MAX_DGRAM_SIZ, 0, peer, &peer_len);
38         out_buffer[bytes_read] = 0;
39
40         if (verify_peers(srv, peer, socklen, peer_len) < 0)
41             continue; /* ignore */
42
43         if (fprintf(stdout, out_buffer, MAX_DGRAM_SIZ) < 0)
44             err_normal("Error while writing to stdout\n");
45     }
46 }
47
48 int verify_peers(SA_PTR peer1, SA_PTR peer2, socklen_t len1, socklen_t len2) {
49     if ((len1 != len2) || memcmp(peer1, peer2, len2) != 0) {
50         fprintf(stderr, "Unknown peer: %s\n", inet_ntop_ex(peer2, len2));
51         return -1;
52     } else {
53         return 0;
54     }
55 }
```

Listing 13.9: xcode/udpechocli1.c - UDP-Version des Echo Clients.

Dem Client wurde noch eine kleine Zusatzfunktion spendiert: `verify_peers`. Sie gibt das Ergebnis eines Vergleichs zweier Adressstrukturen zurück. Die eine Struktur ist die unseres Servers, an den wir mit `sendto(3)` Daten gesendet haben. Die zweite Struktur wurde nur angelegt, um sie `recvfrom(3)` als Zeiger zuzuführen, damit wir feststellen können, ob das empfangene Datagram auch tatsächlich für uns bestimmt ist, denn durch den verbindungslosen Charakter UDPs ist es legitim Daten an Clients zu senden, die wir eigentlich nicht angefordert haben. Nachteilig an dieser Methode ist, daß sie bei Host mit mehreren Netzwerkschnittstellen (*multi-homed host*) versagen kann, da die Prüfung mit `memcmp(3)` fehlschlägt, wenn eine andere Schnittstelle als die von uns ausgewählte für die Übertragung verwendet wird. □

Wie eingangs erwähnt, können wir auch alternativ `connect(2)` mit UDP-Sockets verwenden. Dazu müssen wir die `process_reply` so verändern, daß wir nicht mehr `recvfrom(3)` oder `sendto(3)` zum Senden und Empfangen nutzen. Die `main`-Funktion bleibt hingegen unverändert, wie Listing 13.10 zeigt.

Da wir es dank `connect(2)` mit einem verbundenen Socket zu tun haben, sind wir in der Lage mit der Funktion `getsockname(3)` die mit einem Socket assoziierten Adressinformationen abfragen.

```
#include <sys/socket.h>

int getsockname(int socket, struct sockaddr *addr, socklen_t *addrlen);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler

socket

Socket, dessen Addressinformationen abgefragt werden sollen.

addr

Zeiger auf eine Adressstruktur in der die Informationen aus dem Kernel kopiert werden sollen.

addrlen

Größe der Adressstruktur auf die `addr` zeigt.

Zur Abfrage der lokalen Adressinformationen würden wir nur ein paar Zeilen Code benötigen:

```
char          *address
struct sockaddr local;
socklen_t      len = sizeof(local);

if (getsockname(socket, local, len) < 0)
    err_fatal("getsockname() failed");

address = sock_ntop_ex((SA_PTR)&local, len);
```

Selbstverständlich gibt es für `getsockname(3)` auch einen Wrapper: `getsockname_ex`. Listing 13.10 zeigt den neuen Code unseres UDP Clients.

Listing 13.10: Ein UDP Echo Client mit `connect(2)`

```
1 #include "header.h"
2
3 void      process_reply(FILE *, int, const SA_PTR, socklen_t);
4 char      *get_local_address(int);
5
6 int main(int argc, char **argv) {
7     char                  *basename = basename_ex(argv[0]);
8     int                   socket;
9     struct sockaddr_in   server;
10
11     if (argc != 2)
12         err_fatal("Usage: %s <server-ip-address>\n", basename);
13
14     memset((char *)&server, 0, sizeof(struct sockaddr_in));
15
16     server.sin_family = AF_INET;
17     server.sin_port = htons(9191);
18     inet_pton_ex(AF_INET, argv[1], &server.sin_addr);
19     socket = socket_ex(AF_INET, SOCK_DGRAM, 0);
20
21     process_reply(stdin, socket, (SA_PTR)&server, sizeof(server));
22
23     return (0);
24 }
```

```

26 void process_reply(FILE *str, int socket, const SA_PTR srv, socklen_t len) {
27     int bytes_read;
28     char in_buffer[MAX_BUF + 1], out_buffer[MAX_BUF];
29
30     connect_ex(socket, (SA_PTR)srv, len);
31     printf("Local socket: %s\n", get_local_address(socket));
32
33     while (fgets_ex(out_buffer, MAX_BUF, str) != NULL) {
34         write_ex(socket, out_buffer, strlen(out_buffer));
35         bytes_read = read_ex(socket, in_buffer, MAX_BUF);
36         in_buffer[bytes_read] = 0;
37         fputs_ex(in_buffer, stdout);
38     }
39 }
40
41 char *get_local_address(int socket) {
42     char *address;
43     struct sockaddr local;
44     socklen_t len = sizeof(local);
45
46     getsockname_ex(socket, (SA_PTR)&local, &len);
47     address = sock_ntop_ex((SA_PTR)&local, len);
48
49     return address;
50 }
```

Listing 13.10: xcode/udpechocli2.c - Ein UDP Echo Client mit connect(2).

Ein weiterer Vorteil von `connect(2)` ist, daß wir nicht mehr überprüfen müssen, wer uns Datagramme schickt, da verbundene Sockets nur Daten von Peers annehmen, die beim Aufruf von `connect(2)` mit dem Socket registriert wurden. `verify_peers` ist hier also überflüssig.

Starten wir Server und Client, sehen wir folgende Ausgabe:

```
% bin/udpechosrv1 &
% bin/udpechocli2 192.168.0.9
Local socket: 192.168.0.8:23713
Ein Echo, bitte!
Ein Echo, bitte!
^C
```

□

13.3 Asynchrone Sockets

Das Problem blockierende Systemaufrufe ist immer, daß wir Gefahr laufen, nicht zeitnah zu reagieren, weil wir nicht in der Lage sind, auf bestimmte Ereignisse zu reagieren, solange wir in einem Systemaufruf blockieren. Das ist besonders in vernetzten Umgebungen der Fall, wenn wir in einem lokalen Systemaufruf (beispielsweise `read(2)`) blockieren, während die Gegenstelle Daten sendet. Da kann es schon mal passieren daß wir gar nicht merken, daß uns die Gegenstelle ein FIN gesendet hat. Erst beim nächsten Zugriff auf den Socket lesen wir das EOF-Zeichen.

In den gleichen Problembereich läßt sich auch die Verwaltung mehrerer Verbindungen einordnen. Zwar können wir Threads verwenden, um die Flexibilität unserer Anwendung zu steigern, allerdings erhöht sich damit auch der Entwicklungsaufwand erheblich. In solchen Fällen ist es sinnvoll eine Technik zu verwenden, die als *I/O Multiplexing* bezeichnet wird. Diese Technik erlaubt es uns, dem Kernel mitzuteilen, daß wir über bestimmte Ereignisse benachrichtigt werden wollen. Im Speziellen heißt das einfach, daß uns der Kernel Bescheid sagt, wenn an einem Socket oder File Descriptoren Daten zum Lesen oder Schreiben anstehen. Auf diese Weise blockiert unsere Anwendung nicht beim Lesen von einem Descriptor, beispielsweise in der Funktion `process_reply`, die in `fgets(3)` blockiert.

In diesem Abschnitt beschäftigen wir uns mit den Systemaufrufen `select(2)`, `pselect(2)` und `poll(2)`.

Zwar sprechen wir immer von asynchronen Sockets (nicht zu verwechseln mit asynchronem I/O der POSIX Realtime Definition), obwohl in der Fachwelt ausschließlich von I/O Multiplexing gesprochen wird, wenn (`p`)`select(2)` und `poll(2)` gemeint sind. Ich beziehe mich hier aber auf den Faktor Zeit, denn asynchron bedeutet nichts weiter, als daß wir nicht vorhersagen können, wann ein bestimmtes Ereignis eintrifft. Genau das ist auch mit den genannten Funktionen der Fall: irgendwann wird uns der Kernel benachrichtigen, daß Daten an bestimmten Descriptoren zur Verarbeitung anstehen; wann das ist wissen wir aber nicht.

Das mit I/O Multiplexing bezeichnete Verfahren ist folgendermaßen zu verstehen:

1. Zunächst blockiert `select(2)` im Kernel, während auf lesbare oder schreibbare Descriptoren gewartet wird.
2. Anschließend kehrt `select(2)` zurück so daß wir nun
3. eine Lese- oder Schreiboperation (beispielsweise mit `read(2)`/`write(2)`) ausführen können, die
4. ihrerseits blockiert, bis alle Daten gelesen oder geschrieben wurden.

Der Ablauf wird in Abbildung 13.6 illustriert.

Unsere Server und Clients blockieren immer bei mindestens zwei Systemaufrufe. Jeweils beim Lesen und Schreiben: der Client blockiert zunächst in `fgets(3)` auf Benutzereingaben, um sie an den Server weiterzuleiten und blockiert ein zweites mal beim Lesen vom Socket Descriptor, während der Server die Daten überträgt. Je nachdem an welchem Descriptor der Client blockiert, bekommen wir die Ereignisse, wie etwa das Eintreffen von Daten auf dem anderen Descriptor, nicht mit.

13.3.1 select verwenden

`select(2)` löst die zuvor beschriebene Problematik, indem wir dem Funktionsaufruf ein *File Descriptor Set* übergeben, das alle Descriptors enthält, von denen wir interessiert sind. In unserem Fall sind der Socket Descriptor und `stdin`.

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
```

Rückgabewert: Liefert Anzahl der überwachten File Descriptors zurück, oder -1 bei Fehler.

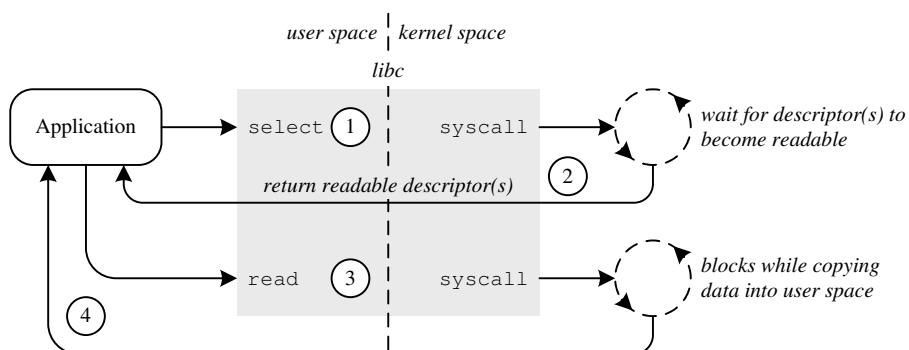


Abbildung 13.6: I/O Multiplexing mit `select(2)`

nfds

Gesamtzahl der zu überwachenden File Descriptoren.

readfds

File Descriptor Set für den lesenden Zugriff.

writelfds

File Descriptor Set für den schreibenden Zugriff.

errorfds

File Descriptor Set die wir auf Fehlerbedingungen prüfen möchten.

timeout

Zeiger auf eine konfigurierte **timeval**-Struktur, die Anzeigt, wie lange wir **select(2)** blockieren lassen möchten.

Der Datentyp **fd_set** ist ein Array von **ints** oder **longs**, abhängig von der Systemarchitektur. Jedes Bit in diesem Array repräsentiert einen File Descriptor, den wir überwachen möchten. Descriptor 0 ist somit Bit 1 im ersten Element, Descriptor 1 demnach Bit 2 im ersten Element und Descriptor 32 ist Bit 1 im zweiten Element, etc. Abbildung 13.7 illustriert den Aufbau von File Descriptor Sets.

Damit wir in unserer Applikation nicht jedesmal Bitoperationen durchführen müssen, um die Arrays zu lesen und zu schreiben, definieren System mit Unterstützung für I/O Multiplexing mehrere Makros. OpenSolaris 10 setzt sie folgendermaßen um:

```
#ifdef _LP64
    #define FD_SETSIZE 65536
#else
    #define FD_SETSIZE 1024
#endif

#define NBBY 8          /* bits per byte */
#define FD_NFDBITS     (sizeof (fds_mask) * NBBY) /* bits per mask */
#define howmany(x, y)   (((x)+(y)-1))/(y)

typedef struct __fd_set {
    long    fds_bits[howmany(FD_SETSIZE, FD_NFDBITS)];
} fd_set;

#define FD_ZERO(__p)      (void) memset((__p), 0, sizeof (*(__p)))

#define FD_SET(__n, __p)  ((__p)->fds_bits[(__n)/FD_NFDBITS] |= \
                        (1ul << ((__n) % FD_NFDBITS)))

#define FD_CLR(__n, __p)  ((__p)->fds_bits[(__n)/FD_NFDBITS] &= \
                        ~ (1ul << ((__n) % FD_NFDBITS)))
```

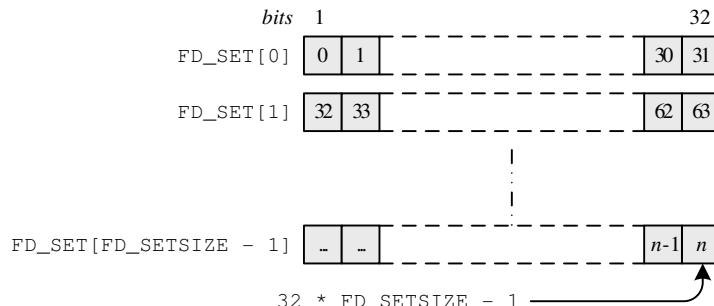


Abbildung 13.7: Organisation von File Descriptor Sets

```
#define FD_ISSET(_n, __p) (((__p)->fds_bits[(__n)/FD_NFDBITS] & \  
 (1ul << ((__n) % FD_NFDBITS))) != 0)
```

Das Makro FD_SETSIZE bestimmt die Obergrenze der maximalen Anzahl zu überwachender File Descriptors. Auf LP64-Systemen sind das bei OpenSolaris $64 * 2^{16}$ und auf LP32-Systemen $32 * 2^{10}$ Descriptors. Zwar kapselt Solaris das Descriptor Set in einer Struktur, doch durch verschiedene Zugriffsmakros ist das für unsere Diskussion nicht relevant.

Die vier Makros FD_ZERO, FD_SET, FD_CLR und FD_ISSET verwenden wir bei allen Operationen, die lesenden oder schreibenden Zugriff auf File Descriptor Sets benötigen. Sie weisen folgende Synopsis auf:

```
#include <sys/select.h>

void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
```

Rückgabewert: FD_ISSET liefert 1 zurück, oder 0 bei Fehler.

FD_ZERO

Initialisiert das mit `fd_set` referenzierte Descriptor Set mit 0.

FD_SET

Setzt das entsprechende Bit für File Descriptor `fd` in `fd_set`.

FD_CLR

Löscht das entsprechende Bit für File Descriptor `fd` in `fd_set`.

FD_ISSET

Gibt 1 zurück, wenn das entsprechende Bit für `fd` in `fd_set` gesetzt ist, andernfalls 0.

Verwenden wir beispielsweise `stdin`, um Benutzereingaben zu lesen und den Socket Descriptor `socketfd`, der mit unserem Server verbunden ist, würden wir etwa so vorgehen:

```
int      socketfd;    /* Server-Socket                      */
fd_set  readset;     /* Descriptor Set zum Lesen          */
...
FD_ZERO(&readset); /* Descriptor Set initialisieren */
FD_SET(socketfd, &readset);
FD_SET(fileno(stdin), &readset);
...
if (FD_ISSET(fileno(stdin), &readset)) {
    acceptfd = accept_ex(socketfd, NULL, NULL);
    process(acceptfd);
}

if (FD_ISSET(socketfd, &readset)) {
    /* lesen socketfd und schreiben in socketfd */
}
...
```

`fileno(3)` ist eine Bibliotheksfunktion, die den mit dem Stream verbundenen File Descriptor zurückgibt. Einzelne Descriptoren entfernen wir aus dem Set, indem wir `FD_CLR` aufrufen:

```
FD_CLR(socketfd, &readset);
```

Wenn wir verschiedene Descriptors mit unterschiedlichen Aufgaben überwachen, müssen wir für jeden dieser Descriptors eine FD_SET-Abfrage formulieren (wie oben für `stdin` und `socketfd` gesehen), denn nur wenn ein bestimmtes Bit im Set gesetzt ist, können wir den entsprechenden Descriptor verwenden.

Die `timeval`-Struktur, welche als letztes Argument an `select(2)` übergeben wird, hat je nach Konfiguration folgende Bedeutung: entweder ist das Argument NULL, so daß `select(2)` solange blockiert, bis einer des Descriptors im Set lesbar oder beschreibbar ist, oder wir konfigurieren es mit einer bestimmten Zeitspanne, die angibt, wie lange `select(2)` blockieren soll und nach Ablauf dieser Zeit auf jeden Fall zurückkehrt. Ist einer der Descriptors im Set früher lesbar (oder beschreibbar) kehrt `select(2)` auch früher zurück. Letztendlich können beide Member der Struktur mit 0 besetzt werden, so daß `select(2)` gar nicht wartet und sofort zurückkehrt unabhängig von dem Zustand der Descriptors.

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;   /* and microseconds */
};
```

Zwar können wir in `timeval` Millisekunden einstellen, jedoch sind die meisten Kernel nicht in der Lage Auflösungen von weniger als 10ms zu verarbeiten, so daß der Member `tv_usec` nur von begrenztem Nutzen ist.

Außerdem müssen wir beachten, daß `select(2)` bereits früher als vereinbart zurückkehrt, weil der Aufruf beispielsweise von einem Signal, daß wir mit einem Signal Handler behandeln, unterbrochen wurde. Demnach müssen wir EINTR abfangen und den Vorgang neu starten, denn wir können uns nicht darauf verlassen, daß `select(2)` auf allen Systemen automatisch neu gestartet wird. Die Abschnitte 8.2 und 8.7.1 besprechen diese Problematik. Obwohl POSIX den `const`-Qualifizierer für `timeout` in `select(2)` definiert, sollten wir beim erneuten Aufruf die `timeval`-Struktur immer neu initialisieren, denn einige Systeme modifizieren diese Struktur, andere nicht.

Eine weitere Frage, die sich stellt ist: Was machen wir mit den beiden Parametern `writfds` und `errorfds`? Während ersterer recht einleuchtend ist, läßt sich letzterer relativ schwer beschreiben. Eine Fehlerquelle, die über Descriptors in diesem Set abgewickelt wird ist beispielsweise das Eintreffen von Out-of-Band-Daten (OOB). Dabei handelt es sich um Nachrichten eines Peers, daß uns über besonders wichtige Ereignisse, wie etwa Fehler informieren möchte. In diesem Buch besprechen wir OOB nicht. Das Descriptor Set `writfds` nutzen wir für den schreibenden Zugriff auf Ressourcen. Werden die beiden oder nur eines der Sets nicht benötigt, übergeben wir einfach NULL.

Die meisten verbindungsorientierten Transportprotokolle unterstützen Out-of-Band-Daten. OOB heißt, daß bestimmte Daten außerhalb der normalen Reihenfolge (Sequenzierung) verarbeitet werden sollen. FTP und Telnet sind bekannterweise Nutzer von OOB-Transmissionen. Telnet beispielsweise verarbeitet darüber Signale, die durch STRG-C abgesetzt werden. Meist sind es interaktive Protokolle und Anwendungen, die OOB-Daten nutzen.

Die folgenden Beispiele fassen die bisher gewonnenen Kenntnisse zusammen.

Listing 13.11: TCP Echo Server mit select(2)

Der Server blockiert beim Aufruf von `select(2)` und wartet bis entweder ein Client eine Verbindung aufbaut, der auf der Kommandozeile spezifizierte Timeout abläuft, um anschließend erneut auf verfügbare Daten zu warten.

```
1 #include <sys/select.h>
2 #include "header.h"
3
4 void process_request(int);
5
6 /*
7  * tcpechosrv4 -- waits for incoming connections with a
8  *                 default timeout of 5 seconds.
```

```

9   *
10  *      Usage: tcpechosrv4 [<timeout-in-secs>]
11  */
12 int main(int argc, char **argv) {
13     char             *basename = basename_ex(argv[0]);
14     char             *usage = "Usage: %s [<timeout-in-secs>] ";
15     int              socketfd, acceptfd, timeout = 5;
16     struct sockaddr_in server;
17     socklen_t         server_len;
18     fd_set            readset;
19     struct timeval    to;
20
21     if (argc == 2)
22         if ((timeout = atoi(argv[1])) <= 0)
23             err_fatal("Timeout must be > 0\n%s%s\n", usage, basename);
24
25     memset((char *)&server, 0, sizeof(server));
26     socketfd = make_tcp_socket(AF_INET, 9191, &server);
27     listen(socketfd, 5);
28
29     while (TRUE) {
30         struct sockaddr_in client;
31         socklen_t         client_len;
32
33         FD_ZERO(&readset);
34         FD_SET(socketfd, &readset);
35
36         to.tv_sec = timeout;
37         to.tv_usec = 0;
38
39         if (select_ex(socketfd + 1, &readset, NULL, NULL, &to) == 0) {
40             printf("select() timed out.\n");
41             continue; /* skip FD_ISSET */
42         }
43
44         if (FD_ISSET(socketfd, &readset)) {
45             acceptfd = accept_ex(socketfd, (SA_PTR)&client, &client_len);
46             printf("Peer: %s\n", inet_ntop_ex((SA_PTR)&client, &client_len));
47             process_request(acceptfd);
48         }
49     }
50 }
51
52 void process_request(int socketfd) {
53     char   buf[MAX_BUF];
54     int    bytes_read;
55
56     memset((char *)buf, 0, sizeof(buf));
57
58     while ((bytes_read = read_ex(socketfd, buf, MAX_BUF)) > 0) {
59         if (strstr(buf, "QUIT") != NULL)
60             return;
61
62         write_ex(socketfd, buf, bytes_read);
63     }
64
65     printf("Closing connection\n");
66     close(socketfd);
67 }

```

Listing 13.11: xcode/tcpecho/tcpechosrv4.c - TCP Echo Server mit select(2).

Da wir im Servercode ja nur den verbundenen Socket überwachen müssen, genügt hier eine einfache FD_ISSET-Abfrage. Natürlich müssen wir den Timeout innerhalb der `while`-Schleife konfigurieren, damit wir ihn erneut für `select(2)` setzen können, falls er abgelaufen ist. Das erste Argument von `select(2)` ist immer die Anzahl der zu überwachenden File Descriptors. Im Servercode machen wir es uns einfach, denn da ja nur ein Descriptor überwacht wird geben wir einfach `socketfd + 1` an. Damit überwacht `select(2)` alle Descriptors von 0 bis `socketfd + 1`, die im Set gesetzt sind. Der Code zur Behandlung von Client-Anfragen ist im Wesentlichen unverändert.

Listing 13.12: TCP Echo Client mit `select(2)`

Der Echo-Client weist zwei Aufrufe von FD_ISSET auf: einen für `stdin` und einen anderen für `socketfd`. Kehrt `select(2)` zurück, ist im Descriptor Set das Bit für einen der beiden Descriptors gesetzt, so daß einer FD_ISSET-Aufrufe die Daten verarbeiten kann.

```

1 #include "header.h"
2
3 void process_request(int);
4
5 int main(int argc, char **argv) {
6     int             socketfd;
7     struct sockaddr_in server;
8     char            *basename = basename_ex(argv[0]);
9
10    if (argc != 2)
11        err_fatal("Usage: %s <server-ip>\n", basename);
12
13    memset((char *)&server, 0, sizeof(struct sockaddr_in));
14
15    server.sin_family = AF_INET;
16    server.sin_port = htons(9191);
17
18    socketfd = socket_ex(AF_INET, SOCK_STREAM, IPPROTO_TCP);
19    inet_pton_ex(AF_INET, argv[1], &server.sin_addr);
20    connect_ex(socketfd, (struct sockaddr *)&server, sizeof(server));
21    signal_ex(SIGPIPE, SIG_IGN);
22    process_request(socketfd);
23
24    close(socketfd);
25    return (0);
26 }
27
28 void process_request(int socketfd) {
29     int      maxfd, fin = 0;
30     fd_set  readset;
31     char    out_buffer[MAX_BUF], in_buffer[MAX_BUF];
32
33     FD_ZERO(&readset);
34
35     while (TRUE) {
36         memset((char *)out_buffer, 0, sizeof(strlen(out_buffer)));
37         memset((char *)in_buffer, 0, sizeof(strlen(in_buffer)));
38
39         if (fin == 0)
40             FD_SET(fileno(stdin), &readset);
41
42         FD_SET(socketfd, &readset);
43         maxfd = MAX(fileno(stdin), socketfd);
44         select_ex(maxfd + 1, &readset, NULL, NULL, NULL);
45
46         if (FD_ISSET(socketfd, &readset)) { /* socket is readable */
47             if (read_ex(socketfd, in_buffer, MAX_BUF) == 0) {

```

```

48         if (fin == 1) /* perform passive close */
49             return;
50         else
51             err_fatal("Server error\n");
52     }
53
54     fputs_ex(in_buffer, stdout);
55 }
56
57 if (FD_ISSET(fileno(stdin), &readset)) { /* input is readable */
58     if (fgets_ex(out_buffer, MAX_BUF, stdin) == NULL) {
59         fin = 1; /* indicate active close */
60         shutdown_ex(socketfd, SHUT_WR); /* send FIN */
61         FD_CLR(fileno(stdin), &readset);
62
63         continue;
64     }
65
66     write_ex(socketfd, out_buffer, strlen(out_buffer));
67 }
68 }
69 }
```

Listing 13.12: xcode/tcpecho/tcpechocli2.c - TCP Echo Client mit select(2).

Im Client verwalten wir ein Flag, `fin`, daß gesetzt wird, wenn wir beispielsweise STRG-C drücken, um den Client zu beenden. Wenn das passiert, liefert `fgets(3)` ein NUL-Byte und gibt `NULL` zurück. Das ist unser Zeichen, die Verbindung zur schließen (Stichwort: FIN). Dazu rufen wir in diesem Beispiel die Funktion `shutdown(2)` auf, die wir in kürze besprechen. Der Server empfängt das FIN, angezeigt durch den Rückgabewert 0 von `read(2)`, und schließt seinerseits die Verbindung mit `close(2)`. Im nächsten Durchgang liest der Client das FIN des Servers vom Socket und erkennt am Flag `fin`, daß ein Active Close durchgeführt wurde und nicht der Server die Verbindung vorzeitig beendet hat. □

13.3.2 pselect verwenden

POSIX.1 definiert eine eigene Variante von `select(2)`, die unter dem Namen `pselect(2)` existiert und damit die unmittelbare Verwandschaft zu ihrem Vorgänge verdeutlicht. Sie unterstützt nun die Angabe einer Signalmaske, um die Zustellung bestimmter Signale während der Ausführung, genauer während des Blockierens, von `pselect(2)` zu verhindern.

Bis auf die letzten beiden Parameter gleicht die Synopsis von `pselect(2)` der von `select(2)`:

```
#include <sys/select.h>
#include <signal.h>
#include <time.h>

int pselect(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
           const struct timespec *timeout, const sigset_t *sigmask);
```

Rückgabewert: ...

nfds

Gesamtzahl der zu überwachenden File Descriptoren.

readfds

File Descriptor Set für den lesenden Zugriff.

writefds

File Descriptor Set für den schreibenden Zugriff.

errorfds

File Descriptor Set die wir auf Fehlerbedingungen prüfen möchten.

timeout

Zeiger auf eine konfigurierte **timespec**-Struktur, die Anzeigt, wie lange wir **pselect(2)** blockieren lassen möchten.

timeout

Zeichert auf einen Signalsatz, der anzeigt, welche Signale dem Prozess nicht zugestellt werden sollen, wenn **select(2)** gerade blockiert.

Die **timespec**-Struktur ist ein POSIX-Kind und bietet Auflösungen im Nanosekundenbereich:

```
struct timespec {
    time_t tv_sec; /* seconds */
    long   tv_nsec; /* nanoseconds */
};
```

(p)**select(2)** und Signale sind eine interessante Kombination. Wenn Prozesse Signale behandeln, setzen sie meist nur ein globales Flag, um einen bestimmten Zustand anzuzeigen. Oftmals wird diese Variable an einer Stelle im Programm (beispielsweise einer Schleife) ausgelesen und eine spezifische Aktion ausgeführt. **select(2)** und **pselect(2)** setzen **errno** auf **EINTR** zurück, wenn sie durch ein Signal unterbrochen wurden. Auf diese Weise wird es uns ermöglicht auf Ereignisse zu reagieren und einen Systemaufruf gegebenenfalls wieder neu zu starten.

Mit **select(2)** könnten wir folgenden Code schreiben:

```
int child_events = 0;

void child_sig_handler (int x) {
    child_events++;
    signal(SIGCHLD, child_sig_handler);
}

int main (int argc, char **argv) {
    sigset(SIGCHLD, child_sig_handler);

    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGCHLD); /* block SIGCHLD */
    sigprocmask(SIG_BLOCK, &sigmask, &orig_sigmask);

    signal(SIGCHLD, child_sig_handler);

    for (;;) { /* main loop */
        for (; child_events > 0; child_events--) {
            /* do work here */
        }
        ...
        r = pselect (nfds, &rd, &wd, &ed, 0, &orig_sigmask);
        ...
    }
}
```

Doch die Sache hat einen Haken: wenn das Signal nach dem ersten Aufruf von **sigprocmask(2)** und vor dem Aufruf von **select(2)** eintrifft kann es zu einer Race Condition kommen. Nun haben wir zwei Möglichkeiten: (a) wir warten eine bestimmte Zeit bis die Race Condition quasi selbst aufgelöst hat (einige Glibc-Implementierungen arbeiten noch immer nach diesem Prinzip) oder (b) es wird ein atomarer Test genutzt um die Race Condition zu vermeiden. Das ganze sieht dann etwa so aus: der Signal Handler setzt ein globales Flag und kehrt zurück. Anschließend wird das Flag getestet und **select(2)** aufgerufen, was dazu führen kann, daß der Funktionsaufruf für immer blockiert, wenn das Signal nach dem Test, aber vor dem Aufruf von **select(2)** eintrifft.

Im Gegensatz dazu erlaubt `pselect(2)`, zunächst bestimmte Signale zu blockieren, dann die eintreffenden Signale zu behandeln und anschließend `pselect(2)` mit der gewünschten Signalmaske aufzurufen.

Der Systemaufruf könnte dann den Anforderungen entsprechend so codiert werden:

Listing 13.13: Einfache Implementierung von `pselect(2)`

```

1 int pselect(int n, fd_set inp, fd_set outp, fd_set exp,
2             struct timeval tvp, const sigset_t sigmask, size_t sigsetsz) {
3     sigset_t ksigmask, sigsaved;
4     struct timeval tv;
5     long ret;
6
7     if (sigmask) {
8         if (sigsetsz != sizeof(sigset_t))
9             return -EINVAL;
10
11     if (tvp != NULL) {
12         tv.tv_sec = tvp->tv_sec;
13         tv.tv_usec = tvp->tv_nsec / 1000;
14     }
15
16     sigdelsetmask(&ksigmask, sigmask(SIGKILL)|sigmask(SIGSTOP));
17     sigprocmask(SIG_SETMASK, &ksigmask, &sigsaved);
18 }
19
20 /* issue system call */
21 ret = select(n, inp, outp, exp, (tvp == NULL ? NULL : &tv));
22
23 if (sigmask)
24     sigprocmask(SIG_SETMASK, &sigsaved, NULL);
25
26 return ret;
27 }
```

Listing 13.13: `xcode/pselect_impl.c` - Einfache Implementierung von `pselect(2)`

Um unsere Anwendungen auf `pselect(2)` umzustellen, ist nicht viel notwendig, wie wir anhand der `process_request`-Funktion aus Beispiel 13.12 sehen:

```

void process_request(int socketfd) {
    ...
    sigset_t sigset_usr1, sigset_empty;

    Sigemptyset(&sigset_empty);
    Sigemptyset(&sigset_usr1);
    Sigaddset(&sigset_usr1, SIGUSR1);
    Signal(SIGUSR1, signal_handler);

    FD_ZERO(&readset);

    while (TRUE) {
        memset((char *)out_buffer, 0, sizeof(strlen(out_buffer)));
        memset((char *)in_buffer, 0, sizeof(strlen(in_buffer)));

        if (fin == 0)
            FD_SET(fileno(stdin), &readset);

        Sigprocmask(SIG_BLOCK, &sigset_usr1, NULL);
        FD_SET(socketfd, &readset);
        maxfd = MAX(fileno(stdin), socketfd);
```

```

n = pselect(maxfd + 1, &readset, NULL, NULL, NULL, &sigset_empty);

if (n >= 0) {
    if (FD_ISSET(socketfd, &readset)) {
        ... /* socket is readable */
    }

    if (FD_ISSET(fileno(stdin), &readset)) {
        ... /* input is readable */
    }
} else {
    err_fatal("pselect() failed");
}

void signal_handler(int signum) {
    return; /* wake up */
}

```

Alles was wir tun müssen ist, die benötigten Signalsätze zu definieren, die in dem Funktionsaufruf zu blockierenden Signale hinzu zufügen (hier: SIGUSR1) und anschließend die Signalmaske mit `sigprocmask(2)` setzen. Beim Aufruf von `pselect(2)` übergeben wir eine leere Signalmaske, die es `pselect(2)` erlaubt selbst alle Signale zu empfangen. Möchten wir nicht, daß alle oder nur bestimmte Signale auch in `pselect(2)` blockiert werden, können wir die übergebene Sinalmaske entsprechend konfigurieren. □

13.3.3 poll(2) verwenden

Während `select(2)` einzig auf E/A reagiert, können wir mit dem `poll(2)`-Systemaufruf bestimmte Ereignisse abonnieren, an denen wir interessiert sind. Auf diese Weise reagieren wir nur auf vorher festgelegte Ereignisse. Ähnlich wie `select(2)` spezifizieren wir einen Satz von File Descriptors, die wir überwachen möchten. Dazu legen wir ein Array von `pollfd`-Strukturen an, in dem jedes Element einen Descriptor aufnimmt. Anschließend übergeben wir das Array an die `poll(2)`-Funktion.

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
Rückgabewert: Anzahl der E/A-bereiten File Descriptors, 0 für Timeout oder -1 bei Fehler.
```

fds

Ein Array von `pollfd`-Strukturen, die jeweils einen zu überwachenden Descriptor beschreiben.

nfds

Anzahl der Elemente im `fds`-Array.

timeout

Zeit, die `poll(2)` warten soll, bis die Funktion zurückkehrt, falls kein spezifiziertes Ereignis an einem der Descriptoren eingetroffen ist (siehe Text).

Das Array von `pollfd`-Strukturen beschreibt die abonnierten Ereignisse für jeden File Descriptor:

```

struct pollfd {
    int      fd;          /* descriptor to check */
    short    events;       /* events subscribed to on fd */
    short    revents;      /* events that occurred on fd */
};

```

Die Ereignisse des Descriptors `fd` werden im `events`-Member durch logische OR-Operationen definiert. Ereignisse, die tatsächlich aufgetreten sind werden auf die gleiche Weise im `revents`-Member kodiert. Zulässig sind:

POLLIN

Nichtpriorisierte Daten können gelesen werden ohne zu blockieren. Abkürzung für `POLLRDNORM` | `POLLRDBAND`.

POLLRDNORM

Nichtpriorisierte Daten können gelesen werden ohne zu blockieren.

POLLRDBAND

Priorisierte Daten können gelesen werden ohne zu blockieren.

POLLPRI

Priorisierte Daten, z.B. OOB-Daten, können gelesen werden ohne zu blockieren.

POLLOUT

Nichtpriorisierte Daten können geschrieben werden ohne zu blockieren. Abkürzung für `POLLWRNORM` | `POLLWRBAND`.

POLLWRNORM

Nichtpriorisierte Daten können geschrieben werden ohne zu blockieren.

POLLWRBAND

Priorisierte Daten können geschrieben werden ohne zu blockieren.

POLLERR

Ein Fehler ist auf dem Gerät oder STREAM aufgetreten. Das Flag ist nur in `pollfd.revents` gültig und wird im `events`-Member ignoriert.

POLLINVAL

Der File Descriptor ist ungültig. Das Flag ist nur in `pollfd.revents` gültig und wird im `events`-Member ignoriert.

POLLHUP

Die Verbindung zum Gerät wurde von der Gegenseite unterbrochen. Dieses Ereignis kann nur im Zusammenhang mit `POLLIN`, `POLLRDNORM`, `POLLRDBAND` oder `POLLPRI` gesetzt sein, da beispielsweise ein Stream niemals schreibbar ist, nachdem ein `POLLHUP` aufgetreten ist. Das Flag ist nur in `pollfd.revents` gültig und wird im `events`-Member ignoriert.

Während die der Einsatz der meisten Flags selbsterklärend ist, dürfte die wichtigste Frage nun sein: Was ist der Unterschied zwischen `POLLPRI` und `POLLRDBAND`? Ersteres wird für das Lesen von OOB-Daten über TCP-Sockets genutzt oder für Pseudo-Terminals (Master) im Paketmodus zur Erkennung von Zustandsänderungen in Slaves (mehr dazu in den Manpages zu `tty_ioctl(4)`) und letzteres für STREAMS. Auf Systemen, die keine STREAMS unterstützen wird das Flag meist auf `POLLPRI` umdefiniert.

Socket Descriptors abonnieren Events durch Verknüpfung der Flags, beispielsweise so:

```
struct pollfd pollfds[2];
... /* set up socket(s) and connect them */
pollfds[0].events = POLLRDBAND | POLLRDNORM;
```

Schlägt der Aufruf von `poll(2)` fehl können wir durch das Abfragen der Flags die Ursache ermitteln:

```
if ((ret = poll(pollfds, 2, 5000)) < 0)
    err_fatal("poll() failed");
else
    if (ret == 0) {
        printf("timed out after 5 seconds");
        return;
```

```
    }

    if (pollfds[0].revents & POLLHUP)
        err_normal("peer hung up");
    else if (pollfds[0].revents & POLLERR)
        err_normal("poll error");
}
```

Fehlschlagen heißt in diesem Zusammenhang, daß weder -1 oder 0 zurückgegeben wurde, aber auch keine Daten anliegen. Wenn der `revents`-Member gesetzt werden konnte, war der Funktionsaufruf zwar erfolgreich, es müssen aber nicht unbedingt Daten übermittelt worden sein, beispielsweise beim Empfang einer Nachricht mit Länge 0.

Listing 13.14: Einfaches Beispiel eines Clients mit `poll(2)`

Was wir hier sehen, ist die `poll(2)`-Variante aus Listing 13.12.

```

45     /* wait for events on the sockets, 5 second timeout */
46     switch (poll_ex(pollfds, 2, 5000)) {
47         case 0:
48             printf("Timeout occurred!  No data after 5 seconds.\n");
49             break;
50
51         default:
52             if (pollfds[0].revents & (POLLIN | POLLPRI)) {
53                 if (read_ex(pollfds[1].fd, in_buffer, MAX_BUF) == 0)
54                     if (fin)
55                         return;
56                 else
57                     err_fatal("Server error\n");
58
59                 fputs_ex(in_buffer, stdout);
60             }
61
62             if (pollfds[1].revents & POLLIN) {
63                 if (fgets_ex(out_buffer, MAX_BUF, stdin) == NULL) {
64                     fin = 1; /* indicate active close */
65                     shutdown_ex(pollfds[0].fd, SHUT_WR); /* send FIN */
66
67                     continue;
68                 }
69
70                 write_ex(pollfds[0].fd, out_buffer, strlen(out_buffer));
71             }
72         }
73     }
74 }
75 }
76 }
```

Listing 13.14: xcode/poll.c - Einfaches Beispiel eines Clients mit poll(2).



Kapitel 14

Fortgeschrittene Socket-Programmierung

Für die Entwicklung professioneller netzwerkfähiger Programme reichen die bisher gewonnenen Kenntnisse nicht unbedingt aus. In diesem Kapitel beschäftigen wir uns daher mit folgenden Themen:

Alternative und spezielle E/A-Funktionen

Wir schauen uns zunächst die beiden Funktionen `recv(2)` und `send(2)` an, die sich ganz ähnlich wie `read(2)` und `write(2)` verhalten, deren Arbeitsweise aber mit einigen Flags verändert werden kann. Auf diese Weise sind wie beispielsweise in der Lage, E/A mal blockierend und mal nicht-blockierend durchzuführen.

Eine weitere völlig andere Form Daten zu lesen und zu schreiben, bieten `readv(2)` und `writev(2)`. Mit ihrer Hilfe können wir Daten unterschiedlicher Puffer schreiben oder Lesen, ohne mehrere Systemaufrufe verwenden zu müssen.

UNIX Domain Sockets

Das UNIX Domain Protocol ist zwar kein echtes Netzwerkprotokoll, aber dennoch hier Teil unserer Besprechungen, da es fast die gleichen Funktionen und Datenstrukturen verwendet, die wir bisher kennengelernt haben. Hierbei handelt es sich um eine Form der Interprozesskommunikation (IPC), die lokal zum Einsatz kommt, um die Zusammenarbeit zwischen Client und Server effektiv zu gestalten.

Broadcasting und Multicasting

Broadcasting beschreibt die Fähigkeit, alle Hosts in einem Subnetz mit bestimmten Daten zu versorgen, ohne eine Anfrage für jeden Host einzeln zu formulieren. Multicasting wird nur für Hosts verwendet, die sich für einen Dienst registriert haben. Auch hier werden die Daten nur einmal vom Server abgesetzt, aber allen registrierten Clients zugestellt. Wie wir das programmatisch umsetzen beschreibe ich in diesem Abschnitt.

Die Liste mit möglichen Themen lässt sich noch erweitern, allerdings würden wir damit den Rahmen des Buches sprengen.

14.1 Alternative und spezielle E/A-Funktionen

In diesem Abschnitt lernen wir drei Techniken für das Lesen und Schreiben von Daten kennen, die sich jeweils fundamental voneinander unterscheiden. Zwar ist das Ergebnis das gleiche, jedoch sind die Ansätze und Anwendungen verschieden. Die Rede ist von `recv(2)/send(2)`, `readv(2)/writev(2)` und `sendmsg(2)` und `recvmsg(2)`.

14.1.1 recv und send verwenden

Die beiden Systemaufrufe `recv(2)` und `send(2)` haben folgende Synopsis:

```
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t len, int flags);
ssize_t send(int socket, const void *buffer, size_t len, int flags);
```

Rückgabewert: Anzahl der Bytes oder -1 bei Fehler.

socket

Socket Descriptor von dem wir lesen oder in den wir schreiben möchten.

buffer

Puffer in den die Daten geschrieben, bzw. aus dem die Daten gelesen werden.

len

Größe des Lese-/Schreibpuffers in Bytes.

flags

Zeigen an, wie die Lese- oder Schreiboperation durchgeführt werden soll.

Der POSIX.1-Standard schreibt nur die beiden Flags `MSG_EOR` und `MSG_OOB` vor. Die meisten Systeme unterstützen aber auch weitere Flags:

MSG_EOR (*end-of-record*)

Zeigt die Terminierung einer Dateneinheit (*record*) an und kann nur mit `send(2)` genutzt werden.
Wird nicht mit TCP/IP, aber beispielsweise OSI verwendet.

MSG_OOB (*out-of-band*)

Die Daten sollen als Out-of-Band-Daten über den Socket gelesen oder geschrieben werden. Wie das geschieht ist protokollabhängig.

MSG_DONTWAIT

Erlaubt das Übertragen von Daten im blockierenden und nicht-blockierenden Modus. Bei jedem Funktionsaufruf kann somit entschieden werden, ob der Aufruf blockiert. Vorteil ist, daß wir uns nicht bereits bei der Erstellung des Sockets auf einen der beiden Modi festlegen müssen. Ob das Flag von dem jeweiligen System unterstützt wird, sollte zuvor geprüft werden.

MSG_DONTROUTE

Weist den Kernel an, nicht die Routing-Tabelle zu befragen, um herauszufinden, wo sich das Ziel befindet. Auch hier gilt, daß wir nicht die Socket-Option `SO_DONTROUTE` setzen müssen, wenn der Socket erstellt wird. Ist das Flag gesetzt, aber das Ziel ist nicht im lokalen Subnetz, wird ein Fehler zurückgeliefert.

MSG_PEEK

Das Flag erlaubt uns eine Vorschau auf die verfügbaren Daten, ohne sie jedoch aus dem Buffer zu holen. Beim nächsten Zugriff auf den Buffer sind noch alle Daten vorhanden.

MSG_WAITALL

Teilt dem Kernel mit, daß die Funktion nur dann zurückkehren soll, wenn die gewünschte Anzahl von Bytes gelesen wurde. Allerdings kehrt sie auch zurück, wenn ein Fehler aufgetreten ist, die Verbindung beendet oder ein Signal empfangen wurde.

Benötigen wir keine der Optionen, können wir einfach 0 übergeben.

Im Grunde könnten wir jeden Aufruf von `read(2)` und `write(2)` mit `recv(2)` und `send(2)` ersetzen ohne die Funktionalität zu beeinträchtigen. Bisher haben wir Dateien immer mit `open(2)` geöffnet und mußten uns bereits jetzt entscheiden, welchen E/A-Modus wir verwenden möchten (`O_NONBLOCK`). Um später zwischen den Modi umzuschalten, mußten wir folgenden Code nutzen:

```

int fd, bytes, flag;
...
fd = open("/path/to/file", O_CREAT | O_EXCL);
bytes = read(fd, buffer, buflen);
/* change to non-blocking mode */
flag = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flag | O_NONBLOCK);
bytes = read(fd, buffer, buflen);

```

Das geht mit `recv(2)` viel einfacher und sauberer:

```

int fd, bytes;
...
fd = open("/path/to/file", O_CREAT | O_EXCL);
bytes = recv(fd, buffer, buflen, 0);
/* read in non-blocking manner */
bytes = read(fd, buffer, buflen, MSG_DONTWAIT);

```

14.1.2 `readv` und `writev` verwenden

Während die bisher besprochenen Funktionen zum Lesen und Schreiben von Daten jeweils nur auf einen Buffer angewendet wurden, erlauben `readv(2)` und `writev(2)` das Lesen und Schreiben mehrerer Buffer mit nur einem Funktionsaufruf. Die Synopsis der beiden Funktionen lautet:

```
#include <sys/uio.h>

ssize_t readv(int fd, const struct iovec *iov, int count);
ssize_t writev(int fd, const struct iovec *iov, int count);

Rückgabewerte: Anzahl der gelesenen oder geschriebenen Bytes oder -1 bei Fehler.
```

fd

Ist ein File Descriptor, der einem verbundenen Socket zugeordnet ist, oder einer geöffneten Datei.

iov

Ein Zeiger auf ein Array von `iovec`-Strukturen, die jeweils auf einen Buffer zeigen, in den die Daten gelesen oder aus dem die Daten geschrieben werden sollen.

count

Anzahl der Elemente im `iovec`-Array.

Die `iovec`-Struktur weist nur zwei Felder auf:

```

struct iovec {
    void *iov_base; /* starting address of buffer */
    size_t iov_len; /* size of buffer */
};

```

Das erste Feld zeigt auf die Anfangsadresse des Buffers und `iov_len` gibt an, wie groß der Buffer ist. Das Prinzip ist denkbar einfach: jedes Element im `iovec`-Array enthält eine `iovec`-Struktur, dessen `iov_base`-Feld auf die Anfangsadresse des betreffenden Buffers zeigt. Abbildung X illustriert diesen Zusammenhang.

Das Lesen und Schreiben mehrerer, evtl. unterschiedlicher, Buffer in nur einem Aufruf wird als *Scattered Read* bzw. *Gathered Write* bezeichnet.

Listing 14.1 zeigt ein Beispielprogramm für die Anwendung von `readv(2)`. Es liest Daten aus einer Datei in zwei Buffer, von dem einer 5 und der andere 20 Bytes groß ist.

Listing 14.1: Anwendung von `readv(2)`.

```

1 #include <stdio.h>
2 #include <sys/uio.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 int main(int argc, char **argv) {
7     int             fd, bytes_read, i;
8     struct iovec   iov[2];
9     char           buf1[5];
10    char           buf2[20];
11
12    for (i = 0; i < 20; i++)
13        buf2[i] = 'X';
14
15    for (i = 0; i < 5; i++)
16        buf1[i] = 'X';
17
18    iov[0].iov_len = 5;
19    iov[0].iov_base = buf1;
20    iov[1].iov_len = 20;
21    iov[1].iov_base = buf2;
22
23    fd = open_ex("file.dat", O_RDWR);
24    bytes_read = readv_ex(fd, iov, 2);
25
26    printf("Total Bytes: %d\n", bytes_read);
27    printf("Vectors:\n\t");
28
29    for (i = 0; i < (bytes_read < 5 ? bytes_read : 5); i++)
30        printf("%c", buf1[i]);
31
32    printf("\n\t");
33    bytes_read -= 5;
34
35    for(i = 0;i < bytes_read; i++)
36        printf("%c", buf2[i]);
37
38    printf("\n");
39    close(fd);
40    return (0);
41 }

```

Listing 14.1: xcode/readv.c - Anwendung von readv(2).



Die Anzahl der in `count` angegebenen Elemente darf nicht größer sein als `IOV_MAX`. Die meisten Implementierungen geben `EINVAL` zurück, wenn diese Grenze überschritten wird. Da POSIX.1 einen Wert von mindestens 16 für `IOV_MAX` vorschreibt, einige Implementierungen aber weit mehr als 1024 erlauben, sollten wir stets prüfen, ob wir die Grenze nicht überschreiten.

14.1.3 `recvmsg` und `sendmsg` verwenden

Die beiden Funktionen `recvmsg(2)` und `sendmsg(2)` haben ihren Ursprung in 4.2BSD. Grundsätzlich können sie alle möglichen Daten von vielen unterschiedlichen Quellen lesen (`recvmsg(2)`) oder schreiben (`sendmsg(2)`).

Zur Datenhaltung greifen beide Funktionen auf eine `msghdr`-Struktur zurück, die ihrerseits mindestens einen Zeiger auf eine `iovec`-Struktur und unter Umständen einen Zeiger auf eine `cmsghdr`-Struktur aufweist und eine Socketadressstruktur. Dazu gleich mehr.

Die Synopsis für `recvmsg(2)` und `sendmsg(2)` lautet folgendermaßen:

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t recvmsg(int fd, struct msghdr *msg, int flags);
ssize_t sendmsg(int fd, const struct msghdr *msg, int flags);
```

Rückgabewerte: Anzahl der gelesenen oder geschriebenen Bytes oder -1 bei Fehler.

fd

File Descriptor von dem gelesen oder in den geschrieben werden soll.

msg

Zeiger auf eine `msghdr`-Struktur, die bestimmt was und wie es gesendet werden soll.

flags

Kein, ein oder mehrere Flags, die bestimmen oder anzeigen, wie `sendmsg(2)` oder `recvmsg(2)` die Daten gesendet oder gelesen hat.

Die `_XOPEN_SOURCE`-Direktive stellt sicher, daß zugrundeliegende Systeme die 4.4BSD-Variante des Systemaufrufs richtig interpretieren. Fehlt sie, so kann es passieren, daß, besonders ältere Implementierungen, standardmäßig von der alten 4.3-Semantik ausgehen, die `ssize_t` und `const` nicht kennen.

Zum Lesen und Schreiben greifen die Funktionen auch hier auf die `iovec`-Struktur zurück, die wir gerade kennengelernt haben:

```
struct msghdr {
    void        *msg_name;           /* Set to NULL if not needed */
    socklen_t   msg_namelen;         /* Set to 0 if not needed */
    struct iovec *msg_iov;
    int         msg_iovlen;
    void        *msg_control;        /* Set to NULL if not needed */
    socklen_t   msg_controllen;      /* Set to 0 if not needed */
    int         msg_flags;
};
```

Bis auf das `msg_flags`-Feld treten die übrigen immer Paarweise auf: so spezifizieren `msg_name` und `msg_namelen` eine Socketadressstruktur sowie deren Größe, `msg_iov` und `msg_iovlen` einen Zeiger auf die Buffer und deren Längen und schließlich `msg_control` und `msg_controllen` einen Zeiger auf einen Control Message Buffer und dessen Größe. Flags werden nur von `recvmsg(2)` gesetzt und haben beim Senden mit `sendmsg(2)` keine Bedeutung.

Wenn wir den `msg_name`-Zeiger auf `NULL` setzen, zeigen wir bei einer Leseoperation an, daß wir nicht an der Hostadresse interessiert sind, was zum Beispiel auf lokaler Ebene Sinn macht, oder wenn wir ohnehin eine Peer-to-Peer-Verbindung hergestellt haben. Auf die gleiche Weise verfahren wir mit `msg_control`, wenn wir weder die zusätzlichen Daten (*ancillary data*) lesen oder welche senden möchten. Mehr über Ancillary Data Objects erfahren Sie weiter hinten in diesem Abschnitt.

Das `msg_flags`-Feld kann nach einem Aufruf von `recvmsg(2)` folgende Flags aufweisen:

MSG_BCAST

Zeigt an, daß die Zieladresse eines empfangenen Datagrams eine Broadcast-Adresse ist. Alternativ kann diese Information für UDP-Datagramme mit Hilfe der Socketoption `IP_RECVSTADDR` abgefragt werden.

MSG_MCAST

Zeigt an, daß die Zieladresse eines empfangenen Datagrams eine Multicast-Adresse ist.

MSG_TRUNC

Zeigt an, daß die nicht alle Daten vom Kernel in den User Space durchgereicht wurden, der Kernel also mehr Daten hat, als mit `recvmsg(2)` abgerufen wurden. Das deutet darauf hin, daß der Prozess nicht genug Elemente im `iovec`-Array allokiert hat.

MSG_CTRUNC

Zeigt an, daß die Zusatzdaten (`cmsg`) nicht vollständig vom Prozess gelesen wurden, und dem Kernel noch Daten vorliegen.

Wenn `MSG_TRUNC` oder `MSG_CTRUNC` gesetzt sind, können wir nicht einfach ein zweites mal `recvmsg(2)` aufrufen, um die übrigen Daten abzurufen, weil es vorkommen kann, daß die Daten nach dem ersten Aufruf nicht mehr vorliegen und vom Kernel bereits verworfen wurden, was POSIX.1 empfiehlt, es aber Implementierungen gibt, die sie dennoch vorhalten. Einige Implementierungen verwerfen die Daten und benachrichtigen die Applikation gar nicht. Da wir es aber im Vorfeld nicht wissen können, gibt es nur eine Strategie: wir allokierten Buffer, die immer mindestens 1 Byte größer sind, als das größtmögliche Datagramm, daß wir überhaupt empfangen können. Auf diese Weise ist jedes Datagramm, welches größer ist, als Fehler zu betrachten.

MSG_EOR

Ist das Datagramm das Ende eines Records, ist das Flag gesetzt, andernfalls nicht. Naturgemäß trifft das nur auf UDP zu.

MSG_OOB

Das Flag wird momentan nur von OSI-Protokollen genutzt, obwohl andere Implementierungen das Senden von Out-of-Band-Daten unterstützen.

MSG_NOTIFICATION

Zeigt bei Verwendung von SCTP an, daß ein Ereignis aufgetreten ist und keine Datennachricht gesendet wurde.

MSG_NOTIFICATION

Für das Senden von Nachrichten mit `sendmsg(2)` stehen nur drei Flags zur Verfügung: `MSG_EOR` um das Ende eines logischen Records anzugeben, `MSG_OOB` um den Transport von Out-of-Band-Daten anzugeben und `MSG_DONTROUTE` um das automatische Routing zu umgehen.

Eine Besonderheit von `sendmsg(2)` ist die Tatsache, daß der Funktionsaufruf eine atomare Operation zur Folge hat. Damit versucht der Kernel alle Daten der `iovec`-Buffer in nur einem Paket unterzubringen.

14.1.3.1 Kontrollinformationen verarbeiten

Die `msghdr`-Struktur kann einen Buffer referenzieren (`msghdr.msg_control`), der Kontrollinformationen (auch Zusatzinformationen, *ancillary data objects*) beinhaltet. Dieser Buffer wird durch eine oder mehrere `cmsg`-Strukturen beschrieben, die beispielsweise angeben, welche Kontrollinformationen wir empfangen oder senden möchten. Je Informationsobjekt ist ein `cmsg` vorangestellt. Damit wird klar, daß die `cmsg`-Struktur nicht beliebige Daten enthalten kann. Während der Header für alle Kontrollinformationen gleich ist, unterscheidet sich der Nutzdatenbereich je nach Inhalt in der Länge. Jedem Nutzdatenbereich wird eine `cmsg`-Struktur vorangestellt, der anzeigt, welche Daten transportiert werden. Dennoch sind alle `cmsg` und die Nutzdaten Bestandteil des gleichen Buffers wie Abbildung 14.1 zeigt.

Die `cmsg`-Struktur ist folgendermaßen in `<sys/socket.h>` definiert:

```
struct cmsghdr {
    socklen_t cmsg_len; /* length in bytes, including this structure */
    int cmsg_level; /* originating protocol */
    int cmsg_type; /* protocol-specific type */
    /* followed by unsigned char cmsg_data[] */
};
```

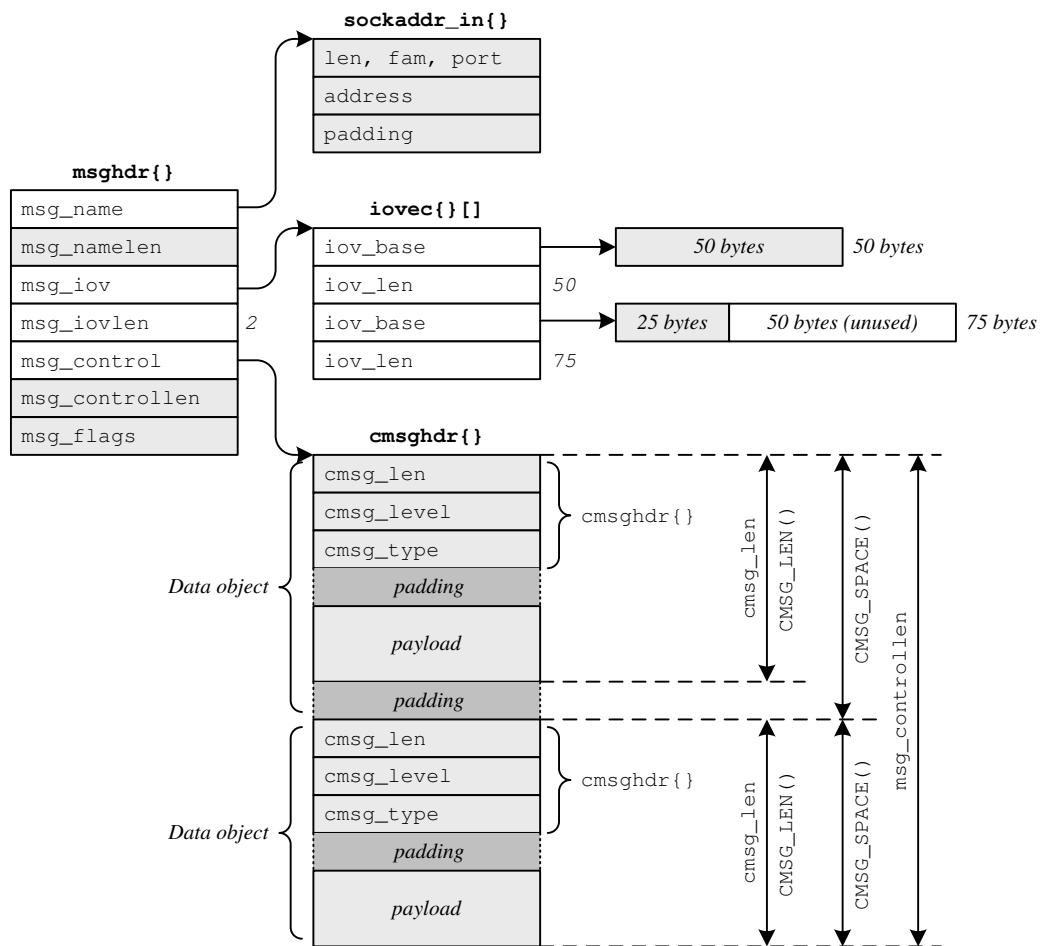


Abbildung 14.1: Kapselung der cmsghdr-Struktur.

Die Felder der Struktur haben folgende Bedeutungen:

cmsg_len

Größe der Struktur: `cmsghdr` plus Nutzdaten. Das ist genau die Größe, die das `CMSG_LEN`-Makro zurückliefert, das weiter unten besprochen wird.

cmsg_level

Bennent eine Schicht in der Socket-API bzw. im Protokollstabel in der bestimmte Kontrollinformationen ermittelt oder übermittelt werden sollen.

cmsg_type

Typ der Kontrollnachricht, abhängig von `cmsg_level`.

Das letzte kommentierte Feld in der `cmsghdr`-Struktur, `cmsg_data`, ist nicht Bestandteil derselben, zeigt aber an, wo das betreffende Datenobjekt im Speicher platziert wird. Die beiden Felder `cmsg_level` und `cmsg_type` sind direkt voneinander abhängig. Tabelle 14.1 zeigt welche Anwendungen von Kontrollinformationen möglich sind.

<code>cmsg_level</code>	<code>cmsg_type</code>	Beschreibung
<code>IPPROTO_IP</code>	<code>IP_RECVDSTADDR</code>	Auslesen der Zieladresse eines UDP-Datagrams
	<code>IP_RECVIF</code>	Auslesen des eingehenden Interfaces eines UDP-Datagrams
<code>IPPROTO_IPV6</code>	<code>IPV6_DSTOPTS</code>	Angabe von Optionen für die Gegenstelle
	<code>IPV6_HOPLIMIT</code>	Angabe von Hop Limits
	<code>IPV6_HOPOPTS</code>	Setzen von Optionen für Hops
	<code>IPV6_NEXTHOP</code>	Spezifiziert die Adresse des nächsten Hops
	<code>IPV6_PKTINFO</code>	Spezifizieren von Paketinformationen (<code>in6_pktinfo</code>)
	<code>IPV6_RTHDR</code>	Übermittlung eines Routing-Headers
	<code>IPV6_TCLASS</code>	Angabe einer Traffic Class
<code>SOL_SOCKET</code>	<code>SCM_RIGHTS</code>	Senden und Empfangen von Descriptoren
	<code>SCM_CREDS</code>	Senden und Empfangen von Benutzerinformationen (<code>credentials</code>)

Tabelle 14.1: Mögliche Anwendungen von Kontrollinformationen

Eine vollständige Beschreibung der meisten verfügbaren Socketoptionen für jede Schicht finden Sie in Anhang B ab Seite 449.

Die Frage ist nun, was wir in die Felder `msghdr.msg_controllen` und `cmsghdr->cmsg_len` eintragen müssen. Das Feld beschreibt die Größe aller enthaltenen `cmsghdr`-Strukturen inklusive ihrer angegliederten Nutzdatenbereiche. Allerdings können wir nicht einfach

```
struct msghdr    m;
struct cmsghdr *c1, *c2;

m.msg_controllen = (sizeof(struct cmsghdr) + sizeof(int)) * 2;
```

schreiben, da es gewissen Anforderungen an das Alignment gibt, die berücksichtigt werden müssen. Aus diesem Grund verwenden wir vordefinierte Makros, die uns bei der Arbeit mit Ancillary Data Objects unterstützen. Dazu gleich mehr.

Abbildung X zeigt die beteiligten Datenstrukturen nach einem Aufruf von `recvmsg(2)`. Während wir `iovec` bereits kennen, sind `msghdr` und `cmsghdr` neu hinzugekommen. Die hellgrauen Felder wurden vom Kernel modifiziert, während die weißen Felder von der Applikation gesetzt werden.

Wir sehen können schließen, dass der Nutzdatenbereich nicht immer direkt an den Header an, sondern wird optional durch ein Padding an Byte-Grenzen ausgerichtet (*alignment*). Wenn wir uns die `cmsghdr`-Struktur ansehen, bemerken wir das Fehlen eines Feldes für die Daten, beispielsweise in Form eines Zeigers auf eine `char`-Array oder ähnliches. Stattdessen ist der Header Bestandteil des Buffers (der groß genug sein muss, um den Header und die Nutzdaten zu beinhalten) und wir müssen Makros verwenden, um auf die Datenstrukturen, die in der Regel tatsächlich `char`-Arrays sind, zuzugreifen. POSIX.1 definiert zu diesem Zweck die folgenden drei:

```
#include <sys/socket.h>
#include <sys/param.h>

struct cmsghdr *CMSP_FIRSTHDR(struct msghdr *mhdrptr);

Rückgabewert: Zeiger auf die erste cmsghdr-Struktur oder NULL wenn keine Kontrolldaten vorhanden sind.

struct cmsghdr *CMSP_NXTHDR(struct msghdr *mhdrptr, struct cmsghdr *cmsgptr);

Rückgabewert: Zeiger auf die nächste cmsghdr-Struktur oder NULL, wenn das Ende erreicht ist.

unsigned char *CMSP_DATA(struct cmsghdr *cmsgptr);

Rückgabewert: Zeiger auf erste Byte im Nutzdatenbereich der jeweiligen cmsghdr-Struktur.
```

CMSP_FIRSTHDR liefert immer einen Zeiger auf die erste cmsghdr-Struktur zurück. Dieses Makro benötigen wir immer, wenn wir Ancillary Data Objects erstellen möchten. Im Gegensatz dazu liefert CMSP_NXTHDR immer das cmsgptr nachfolgende cmsghdr-Objekt zurück oder NULL, falls es nur eines gibt. Ist cmsgptr hier NULL so wird einfach die erste cmsghdr-Struktur zurückgegeben, sofern vorhanden. Die beiden Aufrufe CMSP_FIRSTHDR(mhdr) und CMSP_NXTHDR(mhdr, NULL) sind damit gleichbedeutend. Mit CMSP_DATA erhalten wir ganz bequem Zugriff auf den Nutzdatenbereich der Objekte.

RFC3542 erweitert die Schnittstelle zur Manipulation von Ancillary Data Objects um zwei weitere, äußerst nützliche Makros:

```
#include <sys/socket.h>
#include <sys/param.h>

unsigned int CMSP_LEN(unsigned int length);

Rückgabewert: In cmsg_len einzutragende Länge der Daten ohne Padding.

unsigned int CMSP_SPACE(unsigned int length);

Rückgabewert: Gesamtgröße der Kontrolldaten mit Padding.
```

Wir können davon ausgehen, daß quasi alle Implementierungen alle fünf Makros implementieren. Das könnte etwa so aussehen:

```
#define _CMSP_DATA_ALIGNMENT (sizeof (int))
#define _CMSP_HDR_ALIGN(x) (((uintptr_t)(x) + _CMSP_HDR_ALIGNMENT - 1) & \
~(_CMSP_HDR_ALIGNMENT - 1))

#define _CMSP_DATA_ALIGN(x) (((uintptr_t)(x) + _CMSP_DATA_ALIGNMENT - 1) & \
~(_CMSP_DATA_ALIGNMENT - 1))

#define CMSP_DATA(c) \
((unsigned char *)_CMSP_DATA_ALIGN((struct cmsghdr *)(c) + 1))

#define CMSP_FIRSTHDR(m) \
(((m)->msg_controllen < sizeof (struct cmsghdr)) ? \
(struct cmsghdr *)0 : (struct cmsghdr *)((m)->msg_control))

#define CMSP_NXTHDR(m, c) \
((c) == 0) ? CMSP_FIRSTHDR(m) : \
((((uintptr_t)_CMSP_HDR_ALIGN((char *)(c) + \
((struct cmsghdr *)((c))->cmsg_len) + sizeof (struct cmsghdr)) > \
(((uintptr_t)((struct msghdr *)(m))->msg_control) + \
((uintptr_t)((struct msghdr *)(m))->msg_controllen))) ? \
((struct cmsghdr *)0) : \
((struct cmsghdr *)_CMSP_HDR_ALIGN((char *)(c) + \
((struct cmsghdr *)((c))->cmsg_len) + sizeof (struct cmsghdr))))
```

```
((struct cmsghdr *)(c))->cmsg_len)))))

#define CMSG_SPACE(l)
    ((unsigned int)_CMSG_HDR_ALIGN(sizeof (struct cmsghdr) + (l))) \
    \
#define CMSG_LEN(l)
    ((unsigned int)_CMSG_DATA_ALIGN(sizeof (struct cmsghdr)) + (l)) \
```

14.1.3.2 Das CMSG_LEN-Makro

Der an das Makro übergebene Parameter ist die Größe des Objekts, das im Message Control Buffer transportiert werden soll. Es errechnet die Gesamtgröße des betreffenden Datenobjekts inklusive der `cmsghdr`-Struktur und aller notwendigen Pad-Bytes, die *nach dem Header* aber *vor den Daten* eingefügt werden müssen. Das Ergebnis wird dem `cmsg_len`-Feld zugewiesen. Möchten wir beispielsweise nur einen Filedescriptor transportieren, genügt es zu schreiben:

```
struct cmsghdr *cmsgptr;
...
cmsgptr->cmsg_len = CMSG_LEN(sizeof(int));
```

14.1.3.3 Das CMSG_SPACE-Makro

`CMSG_SPACE` berechnet die Gesamtgröße des Buffers (Daten plus `cmsghdr`-Struktur), berücksichtigt aber dabei aber auch das Padding nach den Daten. Daher ist das Makro besonders nützlich, um herauszufinden, wie groß der Buffer für das Datenobjekt insgesamt gewählt werden muß, damit `cmsghdr` und Daten Platz finden. Um auf das Beispiel mit dem File Descriptor zurückzukommen, heißt das, wir müssen einen Buffer erstellen, der neben dem Header auch noch die Daten hineinpassen:

```
int filedes;
char msg_buf [CMSG_SPACE(sizeof(filedes))];
```

Der wesentliche Unterschied zwischen `CMSG_LEN` und `CMSG_SPACE`, daß ersteres nur das Padding vor den Daten berücksichtigt, während letzteres auch nachfolgende Pad-Bytes in die Berechnung einbezieht.

14.1.3.4 Das CMSG_DATA-Makro

Aufgrund des Paddings, ist es nicht mehr so einfach, den eigentlichen Anfang des Datenbereichs im Buffer zu lokalisieren. Deshalb wurde uns das `CMSG_DATA`-Makro spendiert. Als Parameter übergeben wir den Zeiger auf eine `cmsghdr`-Struktur und erhalten dafür einen Zeiger auf den Anfang des Datenbereichs, natürlich ohne die Pad-Bytes. Auf diese Weise können wir nun einen File Descriptor im Buffer platzieren:

```
struct cmsghdr *cmsgptr;
int filedes, *fdptr;
char cmsg_buf [CMSG_SPACE(sizeof(filedes))];
...
/* place fd in ancillary data object */
fdptr = (int *)CMSG_DATA(cmsgptr);
*fdptr = filedes;
```

Später, auf der Gegenseite, können wir umgekehrt vorgehen, um den File Descriptor zu extrahieren:

```
struct cmsghdr *cmsgptr;
int filedes;
...
fd = *(int *)CMSG_DATA(cmsgptr);
```

14.1.3.5 Das CMSG_ALIGN-Makro

Zwar spezifiziert POSIX1.g `CMSG_ALIGN` nicht, dennoch verfügen quasi alle aktuellen Unices über ein solches Makro. Da es nur intern verwendet wird, kann der Name abweichen. Als Parameter übergeben wir eine Länge in Bytes und erhalten eine neue Länge, die sich am systemspezifischen Alignment orientiert.

14.1.3.6 Das CMSG_FIRSTHDR-Makro

Dieses Makro liefert einen Zeiger auf das erste Objekt im Buffer des Datenobjekts zurück. Wir übergeben einen Zeiger der betreffenden `msghdr`-Struktur und erhalten einen Zeiger auf die erste `cmsghdr`-Struktur. Existiert kein Objekt im Buffer liefert das Makro `NULL`, andernfalls verwendet es die Member `msg_control` und `msg_controllen` um die Zeiger auf das Objekt zu berechnen.

14.1.3.7 Das CMSG_NXTHDR-Makro

Während `CMSG_FIRSTHDR` das erste Objekt im Buffer findet, brauchen wir natürlich auch ein weiteres, um einen Zeiger auf das nächste Objekt zu erhalten. Dazu übergeben wir dem Makro einen Zeiger auf die zu durchsuchende `msghdr`-Struktur und einen Zeiger auf das aktuelle `cmsghdr`-Objekt, beispielsweise das erste in der Kette. Gibt es kein weiteres Objekt im Buffer liefert das Makro `NULL`.

Zusammen ergeben die beiden Makros das Handwerkszeug, welches wir benötigen, um alle Objekte in einem Datenobjekt mit einer `for`-Schleife zu untersuchen. Der folgende Codeausschnitt zeigt, wie wir alle vorhandenen Objekte aus dem Buffer extrahieren:

```

struct iovec      iov[2];
u_char           buf[BUFSIZ], data[2048];
struct cmsghdr  *cm;
struct msghdr   m;
int              found = 0, optval = 1;

/* Create socket here ... */

memset((char *)&m, 0, sizeof(m));
memset((char *)&iov, 0, sizeof(iov));

iov[0].iov_base = data;          /* buffer for packet payload */
iov[0].iov_len  = sizeof(data);  /* expected packet length */

m.msg_name = &from;             /* sockaddr_in6 of peer */
m.msg_namelen = sizeof(from);
m.msg_iov = iov;
m.msg_iovlen = 1;
m.msg_control = (caddr_t)buf;   /* buffer for control messages */
m.msg_controllen = sizeof(buf);

/*
 * Enable the hop limit value from received packets to be
 * returned along with the payload.
 */
setsockopt_ex(s, IPPROTO_IPV6, IPV6_HOPLIMIT, &optval, sizeof(optval));

while (!found) {
    recvmsg_ex(s, &m, 0);

    for (cm = CMSG_FIRSTHDR(&m); cm != NULL; cm = CMSG_NXTHDR(&m, cm)) {
        if (cm->cmsg_level == IPPROTO_IPV6 &&
            cm->cmsg_type == IPV6_HOPLIMIT &&
            cm->cmsg_len == CMSG_LEN(sizeof(int))) {
            found = 1;
    }
}

```

```
        printf("hop limit: %d\n", *(int *) CMSG_DATA(cm));

        break;
    }
}
```

In diesem Beispiel möchten wir das Hop-Limit der empfangenen Pakete auslesen. Dazu müssen wir zunächst die Socketoption `IPV6_HOPLIMIT` für die Schicht `IPPROTO_IPV6` setzen. Anschließend bemühen wir uns einer `for`-Schleife, um das Objekt im Buffer zu finden, daß die gewünschte Information enthält.

Möchten wir hingegen Daten aus dem Datenobjekt extrahieren, müssen wir `CMSG_DATA` aufrufen:

```
struct msghdr      m;
struct cmsghdr   *cm;
int             *fdptr;
int              fd;

for (cm = CMSG_FIRSTHDR(&m); cm != NULL; cm = CMSG_NXTHDR(&m, cm)) {
    if (cm->cmsg_level == SOL_SOCKET && cm->cmsg_type == SCM_RIGHTS) {
        fdptr = (int *)CMSG_DATA(cm);
        received_fd = *fdptr;
        break;
    }
}

if (cm == NULL) {
    /* handle error here */
}
```

14.2 Signalgesteuertes I/O

Die signalgesteuerte Ein- und Ausgabe ist vergleichbar mit `select(2)` und `poll(2)` mit dem Unterschied, daß wir das Signal **SIGIO** empfangen, wenn Daten am Descriptor anliegen. Dazu müssen wir den Socket zuerst für den Betrieb im asynchronen Modus konfigurieren und anschließend einen Signal Handler für **SIGIO** einrichten. Trifft ein Datagramm ein, können wir es beispielsweise direkt im Signal Handler empfangen oder die Hauptschleife alles erledigen lassen.

Unabhängig von der Herangehensweise, unser Prozess wird nicht blockiert, solange wir auf Datagramme warten. Trotzdem ist das signalgesteuerte I/O nicht wirklich asynchron, denn der Kernel sagt uns in diesem Modus lediglich, daß wir nun mit einer I/O-Operation *beginnen* können. Im Gegensatz dazu steht das „echte“ asynchrone I/O der `aio_*`-Funktionsfamilie: hier teilt uns der Kernel mit, daß eine I/O-Operation *abgeschlossen* ist, inklusive des Kopierens der Daten vom User Space in den Kernel Space.

Signalgesteuertes I/O ist nur im Zusammenhang mit Sockets, Device Special Files und Pipes sinnvoll, aber nicht mit ordinären Dateien, die immer bereit sind zum Lesen und Schreiben. Dennoch macht es wiederum mit TCP-Sockets nur wenig bis gar keinen Sinn, denn das Signal wird durch den zuverlässigen Charakter des Transportprotokolls viel zu oft erzeugt und führt schließlich zu schweren Leistungseinbußen. Folglich ist signalgesteuertes I/O ausgesprochen gut für UDP-Sockets geeignet.

SIGIO wird mit UDP nur für zwei Ereignisse generiert: wenn ein Datagram eintrifft und ein asynchroner Fehler auftritt. SIGIO wird für TCP-Sockets jedesmal erzeugt, wenn: ein lauschender Socket verbunden wurde, ein Verbindungsabbau eingeleitet und der Verbindungsabbau abgeschlossen wurde, der Half Close abgeschlossen wurde, Daten empfangen und in den Socket geschrieben wurden, ein asynchroner Fehler aufgetreten ist. Verwenden wir dazu noch ein zustandsloses Anwendungsprotokoll wie HTTP, gleicht das einer selbst verordneten DoS-Attacke. Deshalb heißt die Faustregel: Finger weg von asynchronen TCP-Sockets mit SIGIO.

14.3 UNIX Domain Sockets

Seit 4.2BSD ist die Interprozesskommunikation fester Bestandteil aller nachfolgenden Implementierungen. Zu Beginn waren nur Pipes, die der Feder AT&T entsprungen sind, verfügbar. Die Nachteile dieser IPC-Variante zeichneten sich schnell ab, beispielsweise mußten die beiden Kommunikationspartner (ja, nur zwei Prozesse sind erlaubt) einen gemeinsamen Parent haben um ein Rendezvous arrangieren zu können. Darüber hinaus eignete sich die Semantik ganz und gar nicht für verteilte Anwendungen. Nach und nach wurde das Repertoire erweitert, so daß eine einheitliche Schnittstelle zur Kommunikation zwischen Prozessen geschaffen wurde, die von der Einfachheit der Sockets API profitiert und die Nutzung bereits vorhandener Systemaufrufe erlaubt: UNIX Domain Sockets.

Um es gleich vorweg zu nehmen: UNIX Domain Sockets sind im Grunde nicht der Netzwerkprogrammierung im klassischen Sinn zuzuordnen, denn sie stellen lediglich eine Schnittstelle für die lokale Kommunikation zwischen zwei oder mehreren Prozessen dar. Allerdings setzen UNIX Domain Sockets auf die BSD Socket API, was eine Besprechung dieser Technik im Rahmen der Netzwerkprogrammierung durchaus rechtfertigt.

14.3.1 Grundlagen

Bei UNIX Domain Sockets bildet der Socket den Endpunkt einer Kommunikation. Für die Anwender sind UNIX Domain Sockets damit ähnlich zu behandeln wie TCP- oder UDP-Sockets, allerdings sind sie nur innerhalb ihrer sog. *Kommunikationsdomäne* gültig, die sich in der Regel auf das lokale System beschränkt. Der Begriff der *Kommunikationsdomäne* ist eine Abstraktion zur Abbildung eines Sockets. Während alle in den vorangegangenen Sockets mit Hilfe von `bind(2)` an einen Namen gebunden wurden, binden wir UNIX Domain Sockets an einen Pfad im Dateisystem, beispielsweise `/var/socket`. Diese Eigenschaft unterstreicht den lokalem Charakter dieser Technik.

Die Anwendungsmöglichkeiten von UNIX Domain Sockets sind vielfältig. Viele Datenbanksysteme verwenden lokale Sockets um den Zugriff mehrerer Client-Prozesse zu kanalysieren. Ein weiterer prominenter Vertreter ist der X-Server des MIT X11 Systems, der, wenn sich das zu öffnende Display auf dem lokalen System befindet statt eines TCP Sockets einen UNIX Domain Socket öffnet.

UNIX Domain Sockets sind als alternative IPC-Technik eingeführt worden, da sie gegenüber herkömmlichen Varianten einige Fähigkeiten von Haus aus mitbringen, die andere nur durch weiteren Codierungsaufwand zu erreichen sind. So ist es beispielsweise möglich File Descriptoren zwischen Prozessen auszutauschen oder sog. *Credentials* auszutauschen. Sie enthalten Benutzernamen und Passwort und bieten daher ein höheres Maß an Sicherheit und Flexibilität gegenüber anderen Techniken. In diesem Abschnitt besprechen wir einige Beispiele, die das Verfahren genauer beleuchten.

14.3.1.1 UNIX Domain Sockets erstellen

Im Abschnitt 13.1 haben wir die drei Datenstrukturen für die Anwendung von `SOCK_DGRAM` (UDP) und `SOCK_STREAM` (TCP) besprochen. Für UNIX Domain Sockets, die ebenfalls als `SOCK_DGRAM` und `SOCK_STREAM` konfiguriert werden können, existiert die Addressstruktur `sockaddr_un`, welche in `<sys/un.h>` definiert ist:

```
struct sockaddr_un {
    sa_family_t sun_family; /* AF_LOCAL / AF_UNIX */
    char        sun_path[]; /* null-terminated pathname */
};
```

Während `sun_family` die Protokollfamilie identifiziert legen wir mit `sun_path` den Pfadnamen fest, an den der Socket gebunden werden soll. Dieser muß NUL-terminiert sein. `sun_family` kann entweder `AF_UNIX` oder `AF_LOCAL` sein. Das macht eigentlich keinen Unterschied, allerdings spricht der Standard nur von `AF_LOCAL`, wobei `AF_UNIX` auch noch in Zukunft gültig sein dürfte. Die Länge von `sun_path` wurde nicht von POSIX festgelegt, weil einige Systeme oft recht unterschiedliche Kapazitäten für den Pfadnamen vorgeben. Der Standard drückt sich hier folgendermaßen aus:

The size of `sun_path` has intentionally been left undefined. This is because different implementations use different sizes. For example, 4.3 BSD uses a size of 108, and 4.4 BSD uses a size of 104. Since most implementations originate from BSD versions, the size is typically in the range 92 to 108.

Applications should not assume a particular length for `sun_path` or assume that it can hold `_POSIX_PATH_MAX` characters (255).

Das macht die Sache für uns natürlich nicht einfach: weder können wir von einer maximalen Länge ausgehen noch können wir uns auf `_POSIX_PATH_MAX` verlassen. Uns bleibt also nichts anderes übrig als die Größe von `sun_path` mit `sizeof` zu bestimmen und den Pfad, sollte er länger sein als erlaubt, zu kürzen. Über mögliche Folgen sollten wir uns selbstverständlich im Klaren sein, den schließlich kann es dazu führen, daß andere Prozesse den Ort des Rendevouz nicht erkennen. Darüber hinaus, darf der Pfad, auf den wir verweisen, nicht schon im Dateisystem vorhanden sein, so daß wir zuvor `unlink(2)` aufrufen müssen, bevor wir `bind(2)` ausführen.

Der entsprechende Server-Code sieht dann etwa so aus:

```
#include <sys/un.h>
...
struct sockaddr_un addr;
int socketfd;
...
socketfd = socket_ex(AF_LOCAL, SOCK_STREAM, 0);
strncpy(addr.sun_path, "/var/mysocket", sizeof(addr.sun_path) - 1);
addr.sun_family = AF_LOCAL;
unlink("/var/mysocket");
bind_ex(socketfd, (SA_PTR)&addr, SUN_LEN(&addr));
```

Das `SUN_LEN`-Makro bestimmt die Länge der gesamten Socket-Adresstruktur inklusive der NUL-Terminierung des Pfades. Es ist zwar nicht in POSIX.1g definiert, allerdings liefern es viele Implementierungen aus. Sollte es auf Ihrem System nicht vorhanden sein, so habe ich eine mögliche Variante in `<header.h>` definiert:

```
#ifndef SUN_LEN
#define SUN_LEN(ptr) (((struct sockaddr_un *) 0)->sun_path) \
    + strlen((ptr)->sun_path))
#endif
```

Ohne das Makro müßten wir schreiben:

```
bind_ex(socketfd, (SA_PTR)&addr, strlen(addr.sun_path) +
    sizeof(addr.sun_len) + sizeof(addr.sun_family));
```

Wie wir bereits wissen, ruft in der Regel nur der Server `bind(2)` auf. Clients verbinden sich mit einem Server über `connect(2)`. Das funktioniert auch mit UNIX Domain Sockets:

```
#include <sys/un.h>
...
struct sockaddr_un server;
int socketfd;
...
socketfd = socket_ex(AF_LOCAL, SOCK_STREAM, 0);
strncpy(server.sun_path, "/var/mysocket", sizeof(addr.sun_path) - 1);
addr.sun_family = AF_LOCAL;
unlink("/var/mysocket");
connect_ex(socketfd, (SA_PTR)&server, sizeof(server));
```

Wir sehen können, unterscheidet sich die Handhabung von UNIX Domain Sockets nicht wesentlich von den bisher besprochenen. Nachdem so ein Socket erstellt wurde, können wir mit den bekannten Systemaufrufen lesen und schreiben.

14.3.1.2 Stream-Sockets in der UNIX Domain

Das folgende Beispiel zeigt, wie wir einen UNIX Domain Server und einen Client für SOCK_STREAM entwickeln. Im Prinzip gehen wir vor, wie wir es von der Netzwerkprogrammierung her kennen. Für den Server heißt das: 1. Socket erstellen und Socket-Adressstruktur konfigurieren, 2. bind(2) aufrufen, 3. mit listen(2) auf eingehende Verbindungen warten und 4. mit accept(2) Verbindungen aus der Warteschlange abholen.

Listing 14.2: Ein einfacher Echo-Server mit UNIX Domain Sockets

Ganz nach Art eines Echo-Servers lauscht er auf eingehende Verbindungen und sendet alles zurück, was er empfängt. Die Funktion `process_request()` ist identisch mit der aus Listing 13.6 auf Seite 379.

```

1 #include "header.h"
2
3 void process_request(int, char *, size_t);
4 void handler(int);
5
6 int main(int argc, char **argv) {
7     struct sockaddr_un addr;
8     struct sockaddr_un client;
9     char receive_buf[MAX_BUF];
10    char *basename = basename_ex(argv[0]);
11    int socketfd, acceptfd;
12    pid_t pid;
13
14    if (argc != 2)
15        err_fatal("Usage: %s <pathname>\n", basename);
16
17    socketfd = socket_ex(AF_UNIX, SOCK_STREAM, 0);
18    memset((char *)&addr, 0, sizeof(addr));
19    addr.sun_family = AF_UNIX;
20    strncpy(addr.sun_path, argv[1], sizeof(addr.sun_path) - 1);
21    unlink(addr.sun_path);
22
23    bind_ex(socketfd, (SA_PTR)&addr, SUN_LEN(&addr));
24    listen_ex(socketfd, -1);
25    signal_ex(SIGCHLD, handler);
26
27    while (TRUE) {
28        printf("Server [%d] up and running...\n", getpid());
29        socklen_t clilen = sizeof(client); /* value-result argument */
30
31        if ((acceptfd = accept(socketfd, (SA_PTR)&client, &clilen)) < 0)
32            if (errno == EINTR)
33                continue;
34            else
35                err_fatal("accept() failed");
36
37        if ((pid = fork()) == 0) {
38            printf("\t*** now serving...\n");
39            close_ex(socketfd);
40            process_request(acceptfd, receive_buf, LINESIZ);
41            exit(0); /* terminate child process */
42        } else if (pid < 0) {
43            err_fatal("fork() failed in server");
44        }
45
46        close_ex(acceptfd);
47    }
48 }
```

Listing 14.2: xcode/unix_domain/ud_server1.c - Ein einfacher Echo-Server mit UNIX Domain Sockets.

Wir starten den Server am besten auf einer zweiten Konsole, so daß wir die Ausgaben sehen können. Als Parameter übergeben wir den Pfad für das Rendevouz von Server und Client(s):

```
% ./bin/ud_server1 /tmp/socket1
Server [9792] up and running...
```

Der passende Client ist im nächsten Listing (14.3) zu finden.

Listing 14.3: Ein einfacher Echo-Client mit UNIX Domain Sockets

Der Client wartet auf Benutzereingaben und sendet sie an den Server. Der wiederum sendet die Daten sofort wieder zurück, die wir anschließend auf `stdout` ausgeben.

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     char             *basename = basename_ex(argv[0]);
5     int              socketfd;
6     struct sockaddr_un server;
7
8     if (argc != 2)
9         err_fatal("Usage: %s <path>\n", basename);
10
11    memset((char *)&server, 0, sizeof(server));
12    strncpy(server.sun_path, argv[1], sizeof(server.sun_path) - 1);
13    server.sun_family = AF_UNIX;
14
15    socketfd = socket_ex(AF_UNIX, SOCK_STREAM, 0);
16    connect_ex(socketfd, (SA_PTR)&server, sizeof(struct sockaddr_un));
17
18    process_reply(socketfd);
19
20    return (0);
21 }
```

Listing 14.3: xcode/unix_domain/ud_client1.c - Ein einfacher Echo-Client mit UNIX Domain Sockets.

`process_request()` ist identisch mit der aus Listing 13.12 auf Seite 394. □

14.3.1.3 Datagram-Sockets in der UNIX Domain

Datagram-Sockets verwenden meist UDP. In der UNIX Domain ist das auch so. Trotz des lokalen Charakters dieser Technik verwenden wir im Wesentlichen die gleichen Protokolle und die gleiche Semantik. In den folgenden beiden Beispielen erstellen wir einen Datagram-Server und -Client in der UNIX Domain.

Listing 14.4: SOCK_DGRAM Echo-Server mit UNIX Domain Sockets

Die zweite Variante unseres Echo-Servers auf Basis von UNIX Domain Sockets unterscheidet sich bis auf die spezifischen Eigenschaften nicht vom UDP-Echo-Server aus Listing 13.8 auf Seite 385. Selbst die Funktion `process_request()` ist im Wesentlichen gleich.

```

1 #include "header.h"
2
3 void process_request(int, SA_PTR, size_t);
4
```

```

5  int main(int argc, char **argv) {
6      struct sockaddr_un addr;
7      struct sockaddr_un client;
8      char receive_buf[MAX_BUF];
9      char *basename = basename_ex(argv[0]);
10     int socketfd;
11     pid_t pid;
12
13     if (argc != 2)
14         err_fatal("Usage: %s <pathname>\n", basename);
15
16     socketfd = socket_ex(AF_UNIX, SOCK_DGRAM, 0);
17
18     memset((char *)&addr, 0, sizeof(addr));
19     addr.sun_family = AF_UNIX;
20     strncpy(addr.sun_path, argv[1], sizeof(addr.sun_path) - 1);
21     unlink(addr.sun_path);
22     bind_ex(socketfd, (SA_PTR)&addr, SUN_LEN(&addr));
23
24     process_request(socketfd, (SA_PTR)&client, sizeof(client));
25 }
```

Listing 14.4: xcode/unix_domain/ud_server2.c - SOCK_DGRAM Echo-Server mit UNIX Domain Sockets.

Listing 14.5: SOCK_DGRAM Echo-Client mit UNIX Domain Sockets

Der Client für einen SOCK_DGRAM Server weist eine Besonderheit auf, die nach dem Code erläutert wird.

```

1 #include "header.h"
2
3 int main(int argc, char **argv) {
4     char *basename = basename_ex(argv[0]);
5     int socketfd;
6     struct sockaddr_un server, client;
7
8     if (argc != 2)
9         err_fatal("Usage: %s <path>\n", basename);
10
11     socketfd = socket_ex(AF_UNIX, SOCK_DGRAM, 0);
12
13     memset((char *)&client, 0, sizeof(client));
14     strcpy(client.sun_path, tmpnam(NULL));
15     client.sun_family = AF_UNIX;
16     bind_ex(socketfd, (SA_PTR)&client, sizeof(client));
17
18     memset((char *)&server, 0, sizeof(server));
19     strncpy(server.sun_path, argv[1], sizeof(server.sun_path) - 1);
20     server.sun_family = AF_UNIX;
21
22     process_reply(stdin, socketfd, (SA_PTR)&server, sizeof(server));
23
24     return (0);
25 }
```

Listing 14.5: xcode/unix_domain/ud_client2.c - SOCK_DGRAM Echo-Client mit UNIX Domain Sockets.

Als wir UDP-Clients besprachen, erfuhren wir, daß auch Clients `bind(2)` aufrufen können, es in den meisten Fällen jedoch nicht notwendig ist, da ein passender Port automatisch ausgewählt wird. Mit UNIX Domain Sockets im SOCK_DGRAM-Modus müssen wir `bind(2)` aufrufen, damit der Server weiß über welchen Pfadnamen er die Antwort senden kann. Würden wir das nicht tun, gibt der Server einen

Transport endpoint is not connected Fehler beim Aufruf von `sendto(2)` zurück. Da der Pfadnamen an den der Client gebunden wird nur für den Client von Bedeutung ist, können wir `tmpnam(3)` verwenden, um einen Pfadnamen automatisch zu erzeugen. □

14.3.2 Die Funktion `socketpair`

An dieser Stelle wollen wir einen kurzen Ausflug zu den Pipes wagen, denn der `socketpair(2)`-Systemaufruf erlaubt die Erstellung eines anonymen Paars verbundener Sockets, die eine Zwei-Wege-Kommunikation erlauben und damit dem Pipe-Konzept in dieser Hinsicht überlegen sind. Die Synopsis des Systemaufrufs ist der von `socket(2)` sehr ähnlich:

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sv[2]);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler

domain

Spezifiziert das Kommunikationsprotokoll dieses Sockets. `AF_LOCAL` (sprich `AF_UNIX`) ist hier das einzige gültige.

type

Typ des Sockets. Nur `SOCK_STREAM` ist erlaubt.

protocol

Zeigt an, welches Transportprotokoll wir für den Datentransfer verwenden möchten. Nur `IPPROTO_TCP` oder 0.

sv

Ein Array von Socket Descriptors, die für die Kommunikation verwendet werden.

Die beiden Sockets sind nach dem Aufruf von `socketpair(2)` nicht voneinander zu unterscheiden. Es ist damit der Anwendung überlassen, jeweils einen für Client und Server bzw. Parent und Child auszuwählen.

Listing 14.6: Einfaches Beispiel für die Anwendung von `socketpair(2)`

```
1 #include "header.h"
2
3 #define DATA1 "The child is writing."
4 #define DATA2 "Thanks for writing."
5
6 int main(void) {
7     int      sockets[2];
8     pid_t   pid;
9     char    buf[1024];
10
11     socketpair_ex(sockets);
12     pid = fork_ex();
13
14     if (pid > 0) { /* parent code */
15         close(sockets[0]);
16         read_ex(sockets[1], buf, 1024, 0);
17         fprintf(stderr, "Parent read: %s\n", buf);
18         write_ex(sockets[1], DATA2, sizeof(DATA2));
19         close_ex(sockets[1]);
20     } else { /* child code */
21         close(sockets[1]);
```

```

22     write_ex(sockets[0], DATA1, sizeof(DATA1));
23     read_ex(sockets[0], buf, 1024, 0);
24     fprintf(stderr, "Child read: %s\n", buf);
25     close_ex(sockets[0]);
26 }
27
28 exit(0);
29 }
```

Listing 14.6: `xcode/unix_domain/ud_socketpair.c` - Einfaches Beispiel für die Anwendung von `socketpair(2)`.

Starten wir das Programm, sehen wir folgende Ausgaben:

```
% bin/ud_socketpair
Parent read: The child is writing.
Child read: Thanks for writing.
```

Zunächst legen wir fest, daß der Parent File Descriptor 1 zum Lesen und Schreiben verwendet und unser Child File Descriptor 0. Das erledigen wir, indem gleich zu Beginn einer der beiden Descriptoren im `sockets`-Array geschlossen wird. Anschließend können wir immer von dem einen verbliebenen Descriptor lesen oder Daten an ihn übermitteln. □

14.4 Broadcasting und Multicasting

In diesem Abschnitt sehen wir uns die beiden Addressierungsmechanismen Broadcasting und Multicasting an. Bisher haben wir es ausschließlich mit Unicasting zu tun gehabt. Bei diesem Adressierungsmodus spricht ein Prozess immer nur mit einem anderen, der an einem entfernten Endpunkt lauscht. Jedes Protokoll unterstützt unterschiedliche Adressierungsmodi, die in Tabelle 14.2 aufgelistet sind.

Modus	IPv4	IPv6	TCP	UDP
Unicast	•	•	•	•
Multicast	optional	•		•
Anycast	•	•		•
Broadcast	•			•

Tabelle 14.2: Mögliche Anwendungen von Kontrollinformationen

Während Anycasting in die Adressierungsarchitektur von IPv6 integriert ist, fand Anycasting in IPv4 keine weite Verbreitung.

Bei Anycasting spricht ein Prozess auch nur mit einem anderen Prozess wie beim Unicasting. Jedoch wird ein System aus einer Menge von Systemen, die in der Regel die gleichen Dienste bereitstellen, auf Basis einer Metrik ausgewählt (meist das nächstgelegene System). Mit der richtigen Routing-Konfiguration können Hosts ihre Anycast-Dienste für IPv4 oder IPv6 anbieten, indem die gleiche Adresse in die Routing-Protokolle an mehreren Orten eingepflegt wird. Je nach Metrik wird dann automatisch das „beste“ System für die Kommunikation ausgewählt.

Aus Tabelle 14.2 können wir außerdem folgende Tatsachen ableiten:

- Multicasting ist für IPv4 optional, jedoch Pflicht für IPv6.
- IPv6 unterstützt kein Broadcasting, so daß ältere IPv4-Applikationen, die Broadcasting verwenden, umgeschrieben werden müssen, denn nun muß Multicasting die Aufgabe übernehmen.
- Broadcasting und Multicasting können nur mit UDP oder Raw IP, aber nicht mit TCP abgewickelt werden.

Das Broadcasting wird in der Regel verwendet, um einen Dienst im lokalen Subnetz zu lokalisieren, dessen Unicast-Adresse nicht bekannt ist. Diese Form der *Resource Discovery* ist sehr weit verbreitet. Broadcasting ist ebenso einsetzbar, wenn mehrere Clients die gleichen Daten von einem Server benötigen, wie beispiel beim Video-Broadcasting. Oftmals kann aber auch Multicasting zu diesem Zweck zum Einsatz kommen.

Resource Discovery im Internet

ARP und DHCP sind wohl die bekanntesten Vertreter dieser Disziplin. Das Address Resolution Protocol (unterhalb des IP-Layers) sendet eine Anfrage an alle Rechner im lokalen Subnetz, indem es einen MAC-Frame mit der Broadcastadresse als Empfänger absetzt. Es fragt „welche Hardware-Adresse hat das System mit der IP-Adresse w.x.y.z?“. Das ganze funktioniert zwar ohne Einsatz des IP-Layers, ist jedoch ein Beispiel für Broadcasting.

DHCP-Clients senden ebenfalls Broadcasts (meist an die Adresse 255.255.255.255) in das lokale Subnetz in der Annahme, daß sich ein Server im Netzwerk befindet, der die Einstellungen für die Hostkonfiguration übermittelt. Der Server hört den Broadcast und antwortet schließlich, während andere System im lokalen Subnetz die Anfragen einfach ignorieren.

Netzwerkverkehr reduzieren

Broadcasting kann auch zur Reduktion des Netzwerkverkehrs verwendet werden, beispielsweise im Zusammenhang mit Time Servern. Es macht mehr Sinn, die aktuelle Uhrzeit in einem bestimmten Intervall via Broadcasting im Subnetz bekannt zugeben, als daß jeder Client den Server nach der Uhrzeit fragt, was sich bei mehreren hundert Hosts sehr stark bemerkbar macht.

14.4.1 Wie funktioniert Broadcasting

Broadcasting kann in zwei unterschiedlichen Ausprägungen auftreten: Subnetz-Broadcasting und Beschränktes Broadcasting. In beiden Fällen bestimmt die Adresse, welche Ausprägung zum Einsatz kommt.

Subnetz-Broadcasting

Beim Subnetz-Broadcasting wird beispielsweise ein UDP-Datagramme an alle Schnittstellen im lokalen Subnetz gesendet. Das Netzwerk 192.168.1/24 beispielsweise hat die Subnetz-Broadcastadresse 192.168.1.255. Datagramme, die an diese Adresse gesendet werden, sind für alle Netzwerkschnittstellen in diesem /24er Subnetz bestimmt.

Broadcasts dieser Art werden in der Regel nicht von Routern weitergeleitet. Wenn ein Router ein Unicast-IP-Datagramm mit der Empfängeradresse 192.168.1.255 erhält, so wird das Datagramm nicht automatisch weitergeleitet, sofern der Router nicht so konfiguriert wurde. Einige Router verfügen über Konfigurationseinstellungen, die das Routing von Subnetz-Broadcasting aktivieren können.

Beschränktes Broadcasting

Wird die Broadcast-Adresse 255.255.255.255 verwendet so werden IP-Datagramme an alle Schnittstellen gesendet, die über ein Link-Layer-Broadcast erreichbar sind. Das ist beispielsweise während des Bootstrappings über BOOTP oder beim Anfordern einer Hostkonfiguration via DHCP notwendig, wenn die IP-Adresse des Knotens noch nicht bekannt ist. Diese Broadcasts werden niemals weitergeleitet und es gibt keine Routerkonfiguration, die das erlaubt.

Setzt eine Applikation ein UDP-Datagramm mit der Empfängeradresse 255.255.255.255 ab, wandelt der TCP/IP-Stack die Adresse in eine passende Subnetz-Broadcastingadresse der ausgehenden Schnittstelle um. Oftmals ist es notwendig, den Link-Layer direkt anzusprechen, um solche Broadcasts umzusetzen.

Manche Multihomed Hosts senden nur ein Broadcast über das primäre Interface aus. Andere wiederum senden jeweils eine Kopie des Datagramms über jedes Interface, das Broadcasting unterstützt, aus. Für maximale Portabilität sollte die Schnittstellen abgefragt werden und für jede Schnittstelle einmal `sendto(2)` mit der Broadcast-Adresse als Zieladresse der jeweiligen Schnittstelle aufgerufen werden.

Wie Broadcasting funktioniert verstehen wir am besten, wenn wir uns zunächst anschauen, was passiert, wenn wir ein UDP-Datagramm via Unicasting absenden. Abbildung 14.2 zeigt, wie ein UDP-Datagramm mittels Unicasting (links) und Broadcasting (rechts) übertragen wird.

Abbildung 14.2: Unicasting und Broadcasting im Subnetz 192.168.1/24

Das Beispelsubnetz hat die Adresse 192.168.1/24 mit einer 24-Bit Subnetzmaske und 8 Bits für die Hosts. Die sendende Applikation auf der linken Seite der Abbildung ruft `sendto(2)` für einen UDP-Socket auf und sendet ein UDP-Datagramm an den Host mit der Adresse 192.168.1.3 dessen Prozess an Port 1234 lauscht. UDP stellt seinen Header voran, übergibt das Datagramm an IP, das wiederum seinen Header voranstellt, das ausgehende Interface bestimmt und für Ethernet ARP einschalten muß, um die Ziel-IP-Adresse der Hardwareadresse 00:03:ef:56:ab:78 zuzuordnen. Schließlich wird es an den Link Layer übergeben und als Frame mit der 48-Bit-Zieladresse an den Host geschickt. Das Ethernet-Feld *Frame Type* wird auf 0x0800 gesetzt und zeigt an, daß ein IPv4-Paket transportiert wird.

In der Mitte der linken und rechten Abbildung ist ein Host, dessen Ethernet-Interface den Rahmen sehen kann. Er vergleicht die Hardwareadresse des Ziels mit seiner eigenen Adresse und wenn sie nicht übereinstimmen, wird das Paket verworfen. Bei Unicasting entsteht für den Host kein Overhead.

Das Interface des Zielhosts erkennt seine Hardwareadresse als Ziel im Frame Header und liest den gesamten Frame ein. Durch das Auslösen eines Hardware Interrupts wird der Treiber veranlaßt, daß Paket aus dem Speicher des Adapters zu lesen, und da es sich um einen IPv4-Rahmentyp (0x0800) handelt, wird es in die IP-Eingabewarteschlange eingeordnet.

Zunächst vergleicht IP die Zieladresse mit den eigenen bekannten IP-Adressen. Da die Zieladresse in der Liste der eigenen Adressen enthalten ist, wird das Paket akzeptiert und dem UDP-Layer zugeführt. Das IP-Headerfeld *Protocol* enthält den Wert 17, was es als UDP-Datagramm identifiziert.

Der UDP-Layer sieht nun den Zielport an und platziert das Datagramm in der entsprechenden Warteschlange des Sockets. Falls notwendig wird der Prozess aufgeweckt, um das Datagramm zu lesen. Unicast-IP-Datagramme werden nur von dem Host empfangen für den es bestimmt ist; alle anderen Hosts im Subnetz sehen zwar die Frames, aber nicht das IP-Paket.

Anders ist es mit Subnetz-Broadcasts wie es die Abbildung auf der rechten Seite zeigt. Prinzipiell unterscheidet sich die Verarbeitung der Daten in den jeweiligen Schichten nicht grundlegend. Diesmal aber, sendet die Applikation ein UDP-Datagramm an die Broadcastadresse 192.168.1.255, was dazu führt, daß der IP-Layer des Senders, genau wie alle anderen Hosts, die Zieladresse in der Liste der eigenen IP-Adressen vorfindet und somit akzeptiert. Der Link Layer wandelt die Broadcastadresse in eine Link-Layer-Broadcastadresse (ff:ff:ff:ff:ff:ff) um, damit die Ethernet-Adapter die Rahmen nicht verwerfen, sondern dem IP-Layer zuführen. Da der Host in der Mitte nicht auf Port 1234 lauscht, verwirft UDP das Datagramm. Im Gegensatz dazu hat der Sender nicht den Port 1234 gebunden (`bind(2)`), was aber durchaus möglich wäre so daß dessen UDP das Datagramm ebenfalls verwirft.

Das eigentliche Problem mit Broadcasting ist, daß jeder IPv4-Host im Subnetz, alle Pakete vollständig bearbeiten muß, auch wenn er gar nicht von der jeweiligen Applikation betroffen ist. Hosts, die nicht IP-fähig sind, sehen die Pakete nur Ebene des Link Layers und verwerfen die Frames, da sie als 0x0800 (IPv4) kodiert sind.

14.4.2 Broadcasting Clients entwickeln

Wir wollen nun einen Client entwickeln, der via Broadcasting mehrere UDP-Daytime-Server (Standardport 13) im Netzwerk anspricht und die zurückgelieferten Werte ausgibt. Dazu müssen wir lediglich einen UDP-Client entwickeln, der alle Daytime-Server im Netzwerk anspricht. Unser Vorhaben läuft prinzipiell auf zwei Tatsachen hinaus: zum einen müssen wir die Socket-Option `SO_BROADCAST` für den UDP-Socket des Clients aktivieren, und zum anderen müssen wir die Daten der Server in einer Schleife auswerten, da wir ja nun mehrere Datenquellen angesprochen haben.

Zunächst müssen wir die `main`-Funktion aus Listing 13.9 so ändern, daß nun statt Port 9191 Port 13 gebunden wird:

```
server.sin_port = htons(13);
```

Anschließend modifizieren wir den Client wie in Beispiel 14.7 beschrieben.

Listing 14.7: Die process_request-Funktion arbeitet nun mit Broadcasting

```

1 void process_reply(FILE *str, int socket, const SA_PTR srv, socklen_t socklen) {
2     int bytes_read, on = 1;
3     char in_buffer[MAX_DGRAM_SIZ], out_buffer[MAX_DGRAM_SIZ];
4     socklen_t peer_len;
5     struct sockaddr *peer;
6
7     peer = malloc(sizeof(socklen));
8     setsockopt_ex(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
9
10    while (fgets_ex(in_buffer, MAX_DGRAM_SIZ, str) != NULL) {
11        sendto_ex(socket, in_buffer, strlen(in_buffer), 0, srv, socklen);
12
13        while (TRUE) {
14            peer_len = socklen;
15            bytes_read = recvfrom_ex(socket, out_buffer,
16                                      MAX_DGRAM_SIZ, 0, peer, &peer_len);
17
18            if (bytes_read < 0)
19                if (errno == EINTR)
20                    break;
21                else
22                    err_fatal("recv_from() failed in process_reply");
23            else {
24                out_buffer[bytes_read] = 0;
25                printf("from %s: %s",
26                      inet_ntop_ex(peer, peer_len), recvline);
27            }
28        }
29    }
30
31    free(peer);
32 }
```

Listing 14.7: xcode/bcast/bcastclifun1.c - Die process_request-Funktion arbeitet nun mit Broadcasting

Zuerst müssen wir `malloc(2)` zur Anforderung von ausreichend Speicher für die Serveradresse bemühen und `setsockopt(3)` zum Setzen der Socket-Option `SO_BROADCAST` für den UDP-Socket.

Die wichtigste Neuerung gegenüber Listing 13.9 ist die Einführung einer zweiten `while`-Schleife zur Behandlung der Rückgaben unserer Server. □

Soweit so gut. Die Sache hat nur einen Haken: was passiert, wenn der Client in `recvfrom(2)` blockiert und nicht innerhalb einer bestimmten Zeit zurückkehrt? Im einfachsten Fall sehen wir keine Serverantworten in diesem Zeitraum. Das können wir mit einer kleinen Modifikation ändern.

Sobald wir eine Zeile von `stdin` mittels `fgets(3)` gelesen und mit `sendto(2)` an die Server gesendet haben, setzen wir einen Timer von genau 5 Sekunden ein. Läuft er aus, wird das Signal `SIGALARM` abgesetzt und `recvfrom(2)` liefert `EINTR` zurück. Die geänderte Funktion ist in Listing 14.8 abgebildet.

Listing 14.8: Die process_request-Funktion mit einem Timer

```

1 static void alarm_func(int);
2
3 void process_reply(FILE *str, int socket, const SA_PTR srv, socklen_t socklen) {
4     int bytes_read, on = 1;
5     char in_buffer[MAX_DGRAM_SIZ], out_buffer[MAX_DGRAM_SIZ];
```

```

6      socklen_t          peer_len;
7      struct sockaddr *peer;
8
9      peer = malloc_ex(sizeof(socklen));
10     setsockopt_ex(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
11     signal_ex(SIGALRM, alarm_func);
12
13     while (fgets_ex(in_buffer, MAX_DGRAM_SIZ, str) != NULL) {
14         sendto_ex(socket, in_buffer, strlen(in_buffer), 0, srv, socklen);
15         alarm(5);
16
17         while (TRUE) {
18             peer_len = socklen;
19             bytes_read = recvfrom_ex(socket, out_buffer,
20                                     MAX_DGRAM_SIZ, 0, peer, &peer_len);
21
22             if (bytes_read < 0)
23                 if (errno == EINTR)
24                     break;
25                 else
26                     err_fatal("recv_from() failed in process_reply");
27             else {
28                 out_buffer[bytes_read] = 0;
29                 printf("from %s: %s",
30                         inet_ntop_ex(peer, peer_len), recvline);
31             }
32         }
33     }
34
35     free(peer);
36 }
37
38 static void alarm_func(int signum) {
39     return; /* wake up */
40 }
```

Listing 14.8: xcode/bcast/bcastclifun2.c - Die process_request-Funktion mit einem Timer

Auf diese einfache Weise können wir verhindern, daß `recvfrom(2)` dauerhaft blockiert, oder doch nicht?

14.4.2.1 Probleme mit `alarm` und `recvfrom`

Wäre es nicht schön, wenn wir `alarm(3)` so einfach verwenden könnten, wie wir es im letzten Beispiel getan haben? Sicher, das wäre es. Doch wie so oft, müssen wir uns erneut mit der asynchronen Natur der Signale auseinandersetzen.

Nehmen wir an, wir lesen die Rückgabe von `recvfrom(2)` und der Prozess wird schlafen geschickt (können wir mittels `sleep(3)` simulieren). Alle ausstehenden Replies werden weiterhin empfangen und in der Warteschlange des Sockets zwischengespeichert. Während wir schlafen, kann es passieren, daß unser Timer ausläuft, das Signal `SIGALARM` abgesetzt und der Signal Handler aufgerufen wird und den Prozess aufweckt. Anschließend durchlaufen wir die innere while-Schleife solange bis alle Antworten aus der Warteschlange des Sockets abgearbeitet wurden. Sind wir damit fertig blockieren wir erneut in `recvfrom(2)`, diesmal allerdings ohne Timer. Das führt dazu, daß wir schließlich für immer in dem Systemaufruf blockieren und nicht mehr zurückkehren.

Das grundlegende Problem liegt darin, daß Signale zu jedem Zeitpunkt zugestellt werden können und damit unsere Absicht, `recvfrom(2)` zu unterbrechen dahin ist, denn wenn das Signal auftritt, können wir uns gerade irgendwo in der Schleife befinden.

Um dieser Problematik (hierbei handelt es sich um eine klassische Race Condition) zu begegnen, zeige ich zunächst, wie wir es besser nicht machen sollten und stelle dann drei Varianten vor, die das Problem aus der Welt schaffen.

Listing 14.9: In der Schleife Signale blockieren (falsch)

```

1  static void alarm_func(int);
2
3  void process_reply(FILE *str, int socket, const SA_PTR srv, socklen_t socklen) {
4      int             bytes_read, on = 1;
5      char            in_buffer[MAX_DGRAM_SIZ], out_buffer[MAX_DGRAM_SIZ];
6      socklen_t       peer_len;
7      sigset_t        sigset_alarm;
8      struct sockaddr *peer;
9
10     peer = malloc_ex(sizeof(socklen));
11     setsockopt_ex(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
12
13     sigemptyset_ex(&sigset_alarm);
14     sigaddset_ex(&sigset_alarm, SIGALRM);
15     signal_ex(SIGALRM, alarm_func);
16
17     while (fgets_ex(in_buffer, MAX_DGRAM_SIZ, str) != NULL) {
18         sendto_ex(socket, in_buffer, strlen(in_buffer), 0, srv, socklen);
19         alarm(5);
20
21         while (TRUE) {
22             peer_len = socklen;
23             sigprocmask_ex(SIG_UNBLOCK, &sigset_alarm, NULL);
24             bytes_read = recvfrom_ex(socket, out_buffer,
25                                     MAX_DGRAM_SIZ, 0, peer, &peer_len);
26             sigprocmask_ex(SIG_BLOCK, &sigset_alarm, NULL);
27
28             if (bytes_read < 0)
29                 if (errno == EINTR)
30                     break;
31             else
32                 err_fatal("recv_from() failed in process_reply");
33             else {
34                 out_buffer[bytes_read] = 0;
35                 printf("from %s: %s",
36                         inet_ntop_ex(peer, peer_len), recvline);
37             }
38         }
39     }
40
41     free(peer);
42 }
43
44 static void alarm_func(int signum) {
45     return; /* wake up */
46 }
```

Listing 14.9: xcode/bcast/bcastclifun3.c - In der Schleife Signale blockieren (falsch)

Diese falsche reduziert das Fenster durch das Blockieren des Signals während wir in der Schleife sind, aber nicht `recvfrom(2)` aufrufen. Bevor wir `recvfrom(2)` aufrufen, heben wir die Signalblockierung auf und blockieren sobald `recvfrom(2)` zurückkehrt. Wenn das Signal während es blockiert ist generiert wird, stellt es der Kernel zum sobald es nicht mehr blockiert ist. Kapitel 8 *Signalbehandlung* befasst sich ausführlich mit diesem Thema.

Das Programm arbeitet meistens fehlerfrei, so wie immer wenn wir es mit Race Conditions nun mal so ist. Das Problem ist zwar leicht verlagert worden, aber immer noch vorhanden: das Blockieren des Signals und die Aufhebung derselben geschieht in zwei unterschiedlichen Systemaufrufen. Kehrt `recvfrom(2)` mit dem letzten Reply des Servers zurück und das Signal wird zwischen dem Aufruf von `recvfrom(2)` und `sigprocmask(2)` mit `SIG_BLOCK` zugestellt, wird der nächste Aufruf von `recvfrom(2)` für immer blockieren.

Die einfachste Variante zur Lösung der Problematik liegt in der Verwendung der Funktionen `sigsetjmp(2)` und `siglongjmp(2)` wie Listing 14.10 zeigt.

Listing 14.10: `sigsetjmp(2)` und `siglongjmp(2)` verwenden

```

1  static void alarm_func(int);
2  static sigjmp_buf jmp_buf;
3
4  void process_reply(FILE *str, int socket, const SA_PTR srv, socklen_t socklen) {
5      int             bytes_read, on = 1;
6      char            in_buffer[MAX_DGRAM_SIZ], out_buffer[MAX_DGRAM_SIZ];
7      socklen_t       peer_len;
8      struct sockaddr *peer;
9
10     peer = malloc_ex(sizeof(socklen));
11     setsockopt_ex(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
12     signal_ex(SIGALRM, alarm_func);
13
14     while (fgets_ex(in_buffer, MAX_DGRAM_SIZ, str) != NULL) {
15         sendto_ex(socket, in_buffer, strlen(in_buffer), 0, srv, socklen);
16         alarm(5);
17
18         while (TRUE) {
19             if (sigsetjmp(jmp_buf, 1) != 0)
20                 break;
21
22             peer_len = socklen;
23             sigprocmask_ex(SIG_UNBLOCK, &sigset_alarm, NULL);
24             bytes_read = recvfrom_ex(socket, out_buffer,
25                                     MAX_DGRAM_SIZ, 0, peer, &peer_len);
26             out_buffer[bytes_read] = 0;
27             printf("from %s: %s",
28                   inet_ntop_ex(peer, peer_len), recvline);
29         }
30     }
31
32     free(peer);
33 }
34
35 static void alarm_func(int signum) {
36     siglongjmp(jmp_buf, 1); /* wake up */
37 }
```

Listing 14.10: `xcode/bcast/bcastclifun4.c` - `sigsetjmp(2)` und `siglongjmp(2)` verwenden

Wenn das Signal ausgeliefert wurde, rufen wir im Signal Handler `siglongjmp(2)` auf. Damit kehrt `sigsetjmp(2)` in der `process_reply`-Funktion mit 1 zurück (das ist das zweite Argument von `siglongjmp(2)`) und die Ausführung der `while`-Schleife wird unterbrochen.

Anhang A

POSIX-Konstanten für Optionen und Limits

Der folgende Abschnitt listet die POSIX-Konstanten für die Ermittlung von Konfigurationen und Limits auf. Wie Sie diese Konstanten verwenden, erfahren Sie in Kapitel 2.6 *Implementierungsdetails*.

A.1 Laufzeit-Limits und -Werte (sysconf)

Einige dieser Werte, die wir mit `sysconf` abfragen, könnten auf einigen Systemen nicht zur Verfügung stehen, abhängig von der Art des zugrundeliegenden Systems und von anderen Faktoren, wie etwa verfügbare Speicher, etc. Des Weiteren hat POSIX auch eine Untergrenze festgelegt, die für die Ausnutzung einer Option oder eines Limits mindestens notwendig ist.

`AIO_LISTIO_MAX`

Anzahl der E/A-Operationen, die für einen einzigen Aufruf von `listio` von der Implementierung unterstützt werden. Der kleinste Wert muß mindestens `_POSIX_AIO_LISTIO_MAX` betragen.

`AIO_MAX`

Maximale Anzahl ausstehender, asynchroner E/A-Operationen, die von der Implementierung unterstützt werden. Der kleinste Wert muß mindestens `_POSIX_AIO_MAX` betragen.

`AIO_PRIO_DELTA_MAX`

Der Maximale Wert um den ein Prozess seine asynchrone E/A-Priorität, ausgehend von der Scheduling Priority, verringern darf. Der kleinste Wert muß mindestens 0 betragen.

`ARG_MAX`

Maximale Länge des Arguments für den Aufruf der `exec`-Funktionen, einschließlich der Umgebungsinformationen. Der kleinste Wert muß mindestens `_POSIX_ARG_MAX` betragen.

`ATEXIT_MAX`

Maximale Anzahl von Funktionen, die mit der Funktion `atexit` registriert werden können. Der kleinste Wert muß mindestens 32 betragen.

`CHILD_MAX`

Maximale Anzahl der Prozesse, die einer reellen Benutzerkennung (*real user ID*) zugeordnet werden können. Der kleinste Wert muß mindestens `_POSIX_CHILD_MAX` betragen.

`DELAYTIMER_MAX`

Zeigt an wie oft ein Timer abgelaufen ist, bis das Signal tatsächlich ausgeliefert wurde. Dieser so genannte Timer Overrun kann mit `timer_getoverrun` abgefragt werden. Der kleinste Wert muß mindestens `_POSIX_DELAYTIMER_MAX` betragen.

HOST_NAME_MAX

Maximale Länge des Hostnamen (ohne abschließende Null-Terminierung), die von `gethostname` zurückgegeben werden kann. Der kleinste Wert muß mindestens `_POSIX_HOST_NAME_MAX` betragen.

IOV_MAX

Maximale Anzahl von iovec-Strukturen, die für Aufrufe von `readv` und `writev` zur Verfügung stehen. Der kleinste Wert muß mindestens `_XOPEN_IOV_MAX` betragen.

LOGIN_NAME_MAX

Maximale Länge des Login-Namens. Der kleinste Wert muß mindestens `_POSIX_LOGIN_NAME_MAX` betragen.

MQ_OPEN_MAX

Maximale Anzahl von offenen Message Queue Descriptors, die ein Prozess halten darf. Der kleinste Wert muß mindestens `_POSIX_MQ_OPEN_MAX` betragen.

MQ_PRIO_MAX

Maximale Anzahl der Nachrichtenprioritäten, die von der Implementierung unterstützt werden. Der kleinste Wert muß mindestens `_POSIX_MQ_PRIO_MAX` betragen.

OPEN_MAX

Maximale Anzahl an Dateien, die ein Prozess zu einem Zeitpunkt öffnen kann. Der kleinste Wert muß mindestens `_POSIX_OPEN_MAX` betragen.

PAGESIZE

Größe einer Page in Bytes. Der kleinste Wert muß mindestens 1 betragen.

PAGE_SIZE

Entspricht `PAGESIZE`.

PTHREAD_DESTRUCTOR_ITERATIONS

Maximale Anzahl von Versuchen, die unternommen werden können, um einen Thread-spezifischen Datenwert bei Terminierung des Threads zu zerstören. Der kleinste Wert muß mindestens `_POSIX_THREAD_DESTRUCTOR_ITERATIONS` betragen.

PTHREAD_KEYS_MAX

Maximale Anzahl von Schlüsseln, die von einem Prozess erzeugt werden dürfen. Der kleinste Wert muß mindestens `_POSIX_THREAD_KEYS_MAX` betragen.

PTHREAD_STACK_MIN

Minimale Größe des Stacks eines Threads in Bytes. Der kleinste Wert muß mindestens 0 betragen.

PTHREAD_THREADS_MAX

Maximale Anzahl von Threads, die ein Prozess erzeugen darf. Der kleinste Wert muß mindestens `_POSIX_THREAD_THREADS_MAX` betragen.

RE_DUP_MAX

Maximale Anzahl einer hintereinander auftretender BRE (*basic regular expression*), die von den Funktionen `regexec` und `regcomp` erlaubt werden, wenn die Intervallnotation `\(m,n\)` verwendet wird. Der kleinste Wert muß mindestens `_POSIX2_RE_DUP_MAX` betragen.

RTSIG_MAX

Maximale Anzahl von Echtzeitsignalen, die für Anwendungen der Implementierung reserviert sind. Der kleinste Wert muß mindestens `_POSIX_RTSIG_MAX` betragen.

SEM_NSEMS_MAX

Maximale Anzahl von Semaphores, die ein Prozess halten darf. Der kleinste Wert muß mindestens `_POSIX_SEM_NSEMS_MAX` betragen.

SEM_VALUE_MAX

Maximaler Wert, den ein Semaphore aufweisen darf. Der kleinste Wert muß mindestens `_POSIX_SEM_VALUE_MAX` betragen.

SIGQUEUE_MAX

Maximale Anzahl von zwischengespeicherten Signalen, die ein Prozess senden und halten darf. Der kleinste Wert muß mindestens `_POSIX_SIGQUEUE_MAX` betragen.

SS_REPL_MAX

Maximale Anzahl von Anreicherungsoperationen (*replenishment operations*), die gleichzeitig für einen Server Scheduler ausstehen dürfen. Der kleinste Wert muß mindestens `_POSIX_SS_REPL_MAX` betragen.

STREAM_MAX

Maximale Anzahl von Streams, die ein Prozess geöffnet halten kann. Der POSIX-Standard legt fest, daß `STREAM_MAX` den gleichen Wert wie `FOPEN_MAX` aufweisen muß. Der kleinste Wert muß mindestens `_POSIX_STREAM_MAX` betragen.

SYMLOOP_MAX

Maximale Anzahl symbolischer Links, die verlässlich zur Auflösung eines Pfadnamens durchlaufen werden können, sofern kein Loop zwischen den Links vorliegt. Der kleinste Wert muß mindestens `_POSIX_SYMLOOP_MAX` betragen.

TIMER_MAX

Maximale Anzahl von Zeitgebern (*timer*), die die Implementierung für einen Prozess unterstützt. Der kleinste Wert muß mindestens `_POSIX_TIMER_MAX` betragen.

TRACE_EVENT_NAME_MAX

Maximale Länge eines Trace Event Name. Der kleinste Wert muß mindestens `_POSIX_TRACE_EVENT_NAME_MAX` betragen.

TRACE_NAME_MAX

Maximale Länge der Versionsangabe der trace-Generation oder des Stream-Namens. Der kleinste Wert muß mindestens `_POSIX_TRACE_NAME_MAX` betragen.

TRACE_SYS_MAX

Maximale Anzahl von Trace-Streams, die gleichzeitig in einem System vorhanden sein dürfen. Der kleinste Wert muß mindestens `_POSIX_TRACE_SYS_MAX` betragen.

TRACE_USER_EVENT_MAX

Maximale Anzahl von user Trace Event Type IDs, die gleichzeitig in einem „getraceten“ Prozess auftreten dürfen, inklusive dem vordefinierten User Trace Event `POSIX_TRACE_UNNAMED_USER_EVENT`. Der kleinste Wert muß mindestens `_POSIX_TRACE_SYS_MAX` betragen.

TTY_NAME_MAX

Maximale Länge des Namens eines Terminals. Der kleinste Wert muß mindestens `_POSIX_TTY_NAME_MAX` betragen.

TZNAME_MAX

Maximale Anzahl von Bytes, die für den Namen einer Zeitzone von der Implementierung unterstützt werden. Der kleinste Wert muß mindestens `_POSIX_TZNAME_MAX` betragen.

A.2 Laufzeit-Limits und -Werte (pathconf)

Die Mehrzahl der folgenden Limits einer spezifischen Implementierung weisen andere, höhrere Werte auf und sind in `<limits.h>` dokumentiert (Beachten sie, daß die entsprechenden `_PC_`-Werte in `<unistd.h>` zu finden sind). Wir rufen sie mit Hilfe der Funktion `pathconf` ab.

FILESIZEBITS

Minimale Anzahl von Bits, die benötigt werden um die maximale Größe einer regulären Datei als vorzeichenbehafteten Integer-Wert anzuzeigen. Der kleinste Wert muß mindestens 32 betragen.

LINK_MAX

Maximale Anzahl von Links zu einer Datei. Der kleinste Wert muß mindestens `_POSIX_LINK_MAX` betragen.

MAX_CANON

Maximale Anzahl von Bytes einer Zeile eines Terminals im kanonischen Modus. Der kleinste Wert muß mindestens `_POSIX_MAX_CANON` betragen.

MAX_INPUT

Minimale Anzahl von Bytes, die in der Eingabewarteschlange eines Terminals zur Verfügung stehen. Der kleinste Wert muß mindestens `_POSIX_MAX_INPUT` betragen.

NAME_MAX

Maximale Anzahl von Bytes eines Dateinamens (ohne Null-Terminierung). Der kleinste Wert muß mindestens `_POSIX_NAME_MAX` betragen.

PATH_MAX

Maximale Anzahl von Bytes in einem Pfadnamen (ohne Null-Terminierung). Der kleinste Wert muß mindestens `_POSIX_PATH_MAX` betragen.

PIPE_BUF

Maximale Anzahl von Bytes, die beim Schreiben in eine Pipe als atomar behandelt werden können. Der kleinste Wert muß mindestens `_POSIX_PIPE_BUF` betragen.

POSIX_ALLOC_SIZE_MIN

Minimale Anzahl von Bytes, die für den Speicherbereich einer Datei tatsächlich allokiert werden. Hier wurde kein Minimum festgelegt.

POSIX_REC_INCR_XFER_SIZE

Empfohlene Schrittweite zur Erhöhung der Transfergröße von Dateien zwischen `POSIX_REC_MIN_XFER_SIZE` und `POSIX_REC_MAX_XFER_SIZE`. Hier wurde kein Minimum festgelegt.

POSIX_REC_MAX_XFER_SIZE

Maximale empfohlene Größe für den Dateitransfer. Hier wurde kein Minimum festgelegt.

POSIX_REC_MIN_XFER_SIZE

Minimale empfohlene Größe für den Dateitransfer. Hier wurde kein Minimum festgelegt.

POSIX_REC_XFER_ALIGN

Empfohlene Ausrichtung des Buffers für den Dateitransfer. Hier wurde kein Minimum festgelegt.

SYMLINK_MAX

Maximale Anzahl von Bytes eines symbolischen Link. Der kleinste Wert muß mindestens `_POSIX_SYMLINK_MAX` betragen.

Anhang B

Socketoptionen

Es gibt eine ganze Reihe von Socketoptionen. Jeweils anwendbar auf eine bestimmte Schicht in der API, entweder auf Protokoll- oder Socketschicht. Als wichtigste Schnittstellen stehen uns die beiden Funktionen `setsockopt(2)` und `getsockopt(2)` für das Setzen und Abfragen von Socketoptionen zur Verfügung. Alternativ können auch die beiden Universalfunktionen `fcntl(2)` und `ioctl(2)` verwendet werden.

In den weiteren Abschnitten besprechen wir die verfügbaren Socketoptionen für die Schichten `SOL_SOCKET`, `IPPROTO_IP`, `IPPROTO_ICMPV6` und `IPPROTO_IPV6`.

In den folgenden Abschnitten werden die einzelnen Optionen erläutert. Für den Zugriff auf die Socketoptionen verwenden wir `getsockopt(2)` und `setsockopt(2)`:

```
#include <sys/socket.h>

int getsockopt(int fd, int level, int name, void *val, socklen_t *optlen);
int setsockopt(int fd, int level, int name, const void *val, socklen_t optlen);
```

Rückgabewert: 0 bei Erfolg, -1 bei Fehler.

fd

Zeigt auf einen geöffneten Socket Descriptor.

level

Zeigt die Schicht an, der **name** zugeordnet ist.

val

Zeiger auf eine Variable, die entweder den zu setzen Wert enthält oder den abzufragenden Wert aufnimmt.

len

Zeiger, der die Länge des Optionswertes, aufnimmt oder eine Variable die die Länge angeibt.

Wird eine Option mit `setsockopt(2)` gesetzt, so ist `optlen` eine Variable, der die Länge des Wertes von `val` angibt. Für `getsockopt(2)` ist `optlen` ein Value Result Argument.

Wichtig beim Umgang mit Socketoptionen ist die Tatsache, daß einige Optionen von einem nicht verbundenen Socket (siehe `listen(2)`, Abschnitt 13.2.1.3) an den verbundenen Socket (nach `accept(2)`) vererbt werden. Dazu Zählen: `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_LINGER`, `SO_OOBINLINE`, `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF`, `SO SNDLOWAT`, `TCP_MAXSEG`, and `TCP_NODELAY`. Das bedeutet, daß wenn eine dieser Optionen nicht gesetzt ist, bevor `accept(2)` aufgerufen wird, hat der neue Socket ebenfalls keine diese Optionen.

Level	Option	L	S	Beschreibung	F	Datentyp
SOL_SOCKET	SO_BROADCAST	•	•	Broadcasting erlauben	•	int
	SO_DEBUG	•	•	Debugging einschalten	•	int
	SO_DONTROUTE	•	•	Routingtabelle nicht befragen	•	int
	SO_ERROR	•	•	Hole Fehler ab und setze zurück	int	
	SO_KEEPALIVE	•	•	Prüfe regelmäßig ob Verbindung steht	•	int
	SO_LINGER	•	•	Bei close warten bis Daten gesendet wurden	linger{}	
	SO_OOBINLINE	•	•	OOB-Daten inline belassen	•	int
	SO_RCVBUF	•	•	Größe des Empfangspuffers verwalten	int	
	SO_SNDBUF	•	•	Größe des Sendepuffers verwalten	int	
	SO_RCVLOWAT	•	•	Untergrenze des Empfangspuffers verwalten	int	
	SO SNDLOWAT	•	•	Untergrenze des Sendepuffers verwalten	int	
	SO_RCVTIMEO	•	•	Empfangstimeout verwalten	int	
	SO_SNDDTIMEO	•	•	Sendetimeout verwalten	int	
	SO_REUSEADDR	•	•	Wiederverwendung der lokalen erlauben	•	int
	SO_REUSEPORT	•	•	Wiederverwendung des lokalen Ports	•	int
	SO_TYPE	•	•	Sockettyp abfragen	int	
	SO_USELOOPBACK	•	•	Routing-Socket erhält Kopien der Pakete	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP-Header ist im Datenbereich enthalten	•	int
	IP_OPTIONS	•	•	IP-Header Optionen verwalten	ipoption{}	
	IP_RECVSTADDR	•	•	IP-Zieladresse zurückgeben	•	int
	IP_RECVIF	•	•	Gib Index der Empfangsschnittstelle zurück	•	int
	IP_TOS	•	•	Type of Service	int	
	IP_TTL	•	•	Time to Live	int	
	IP_MULTICAST_IF	•	•	Gibt ausgehendes Interface an	in_addr{}	
	IP_MULTICAST_TTL	•	•	Gibt ausgehenden TTL Wert an	u_char	
	IP_MULTICAST_LOOP	•	•	Gibt Loopback-Gerät an	u_char	
	IP_ADD_MEMBERSHIP	•	•	Multicast-Gruppe beitreten	ip_mreq{}	
	IP_DROP_MEMBERSHIP	•	•	Multicast-Gruppe verlassen	ip_mreq{}	
	IP_UNBLOCK_SOURCE	•	•	Multicast-Gruppe: Sperre setzen/aufheben	ip_mreq_source{}	
	IP_ADD_SOURCE_MEMBERSHIP	•	•	Quellspezifischer Multicast-Gruppe beitreten	ip_mreq_source{}	
	IP_DROP_SOURCE_MEMBERSHIP	•	•	Quellspezifische Multicast-Gruppe verlassen	ip_mreq_source{}	
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Welche ICMP6-Nachrichten zulassen		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset des CS-Feldes für Raw-Sockets	int	
	IPV6_DONTFRAG	•	•	Große Pakete verwerfen, nicht fragmentieren	int	
	IPV6_NEXTHOP	•	•	Spezifiziert Adresse des nächsten Hops	sockaddr_in6{}	
	IPV6_PATHMTU	•	•	Frage aktuelle MTU des Pfades ab	ip6_mtuinfo{}	
	IPV6_RECVDSTOPTS	•	•	Empfange Optionen der Gegenstelle	int	
	IPV6_RECVHOPLIMIT	•	•	Empfange Unicast Hop Limit	int	
	IPV6_RECVHOPOPTS	•	•	Empfange Hop-by-Hop Optionen	int	
	IPV6_RECVPATHMTU	•	•	Empfange Pfad-MTU	int	
	IPV6_RECVPKTINFO	•	•	Empfange Paketinformationen	int	
	IPV6_RECVRTHDR	•	•	Empfange Source Route	int	
	IPV6_RECVTCLASS	•	•	Empfange Traffic Class	int	
	IPV6_UNICAST_HOPS	•	•	Standard Unicast Hoplimit	int	
	IPV6_USE_MIN_MTU	•	•	Verwende minimale MTU	int	
	IPV6_V6ONLY	•	•	Schalte IPv4-Kompatibilität ab	int	
	IPV6_MULTICAST_IP	•	•	Spezifiziert ausgehende Schnittstelle	u_int	
	IPV6_MULTICAST_HOPS	•	•	Spezifiziert ausgehendes Hoplimit	int	
	IPV6_MULTICAST_LOOP	•	•	Gibt Loopback-Gerät an	•	u_int
	IPV6_JOIN_GROUP	•	•	Trete einer Multicast-Gruppe bei	ipv6_mreq{}	
	IPV6_LEAVE_GROUP	•	•	Verlasse Multicast-Gruppe	ipv6_mreq{}	
IPPROTO_IP IPPROTO_IPV6	MCAST_JOIN_GROUP	•	•	Trete Multicast-Gruppe bei		group_req{}
	MCAST_LEAVE_GROUP	•	•	Verlasse Multicast-Gruppe		group_source_req{}
	MCAST_UNBLOCK_SOURCE	•	•	Multicast-Gruppe: setzen/aufheben		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP	•	•	Trete quellspezifischer Gruppe bei		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP	•	•	Verlasse quellspezifische Gruppe		group_source_req{}
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP Maximale Segmentgröße	int	
	TCP_NODELAY	•	•	Deaktiviert Dagle-Algorithmus	int	
IPPROTO_SCTP	SCTP_ADAPTATION_LAYER	•	•	Zeigt Adaption Layer an		sctp_setadaption{}
	SCTP_ASSOCINFO	•	•	Assoziation verwalten		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Führt Auto-Close durch	int	sctp_sndrcvinfo{}
	SCTP_DEFAULT_SEND_PARAM	•	•	Standardsendparameter	int	sctp_event_subscribe{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP-Fragmentation	int	sctp_paddrinfo{}
	SCTP_EVENTS	•	•	Ereignisse abonnieren	int	sctp_initmsg{}
	SCTP_GET_PEER_ADDR_INFO	•	•	Status der Gegenstelle abfragen	int	sctp_paddrparams{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	IPv4-Adressen mappen	int	sctp_setprim{}
	SCTP_INIT_MSG	•	•	Standard Initialisierungsparameter	int	sctp_rtoinfo{}
	SCTP_MAX_BURST	•	•	Maximale Burst-Größe	int	sctp_setpeerprimary{}
	SCTP_MAXSEG	•	•	Maximale Fragment-Größe	int	sctp_status{}
	SCTP_NODELAY	•	•	Nagle-Algorithmus deaktivieren	int	
	SCTP_PEER_ADDR_PARAMS	•	•	Adressparameter der Gegenstelle	int	
	SCTP_PRIMARY_ADDR	•	•	Primäre Zieladresse	int	
	SCTP_RTOINFO	•	•	RTO-Informationen	int	
	SCTP_SET_PEER_PRIMARY_ADDR	•	•	Primäre Zieladresse der Gegenstelle	int	
	SCTP_STATUS	•	•	Status der Assoziation abfragen	int	

Tabelle B.1: Übersicht über die Socketoptionen und -Schichten. Erläuterung: S = Wert schreiben, L = Wert lesen, F = ist die Option ein Schalter?

B.1 Socketoptionen für SOL_SOCKET

Es folgt eine Erläuterung der Socketoptionen für die Schicht SOL_SOCKET.

SO_BROADCAST

Diese Option bestimmt, ob ein Prozess Broadcast-Nachrichten senden kann. Naturgemäß kann Broadcasting nur mit Datagramm-Sockets verwendet werden, da Broadcasting mit Punkt-zu-Punkt-Verbindungen keine Bedeutung hat. Broadcasting wird tatsächlich nur dann durchgeführt, wenn die Option ausdrücklich gesetzt wurde, aber nicht wenn beispielsweise eine Socketadressstruktur mit einer Broadcastadresse initialisiert wurde und anschließend `send(2)` aufgerufen wird. In diesem Fall wird `EACCES` zurückgegeben.

SO_DONTROUTE

Diese Option umgeht die Befragung der Routing-Tabelle des Protokolls. In IP-Netzwerken bestimmen die Netzwerkadresse und die Subnetzmaske, welche Schnittstelle für die Weiterleitung der Pakete verwendet wird. Ist diese Option aktiviert, wird die entsprechende Schnittstelle von der Zieladresse abgeleitet und `ENETUNREACH` zurückgegeben, wenn der Versuch fehlschlägt (beispielsweise wenn das Ziel nicht im lokalen Netzwerk vorhanden ist). Normalerweise würde die Routing-Tabelle letztendlich das Default-Gateway als ausliefern, dessen Schnittstelle bekannt ist. Diese Option haben wir im Zusammenhang mit `send(2)`, `sendto(2)` und `sendmsg(2)` kennengelernt: `MSG_DONTROUTE`.

SO_ERROR

Tritt ein Fehler in der Sockeschicht auf, setzt der Kernel eine Variable, die den Fehler anzeigt (einer der typischen EXXX-Werte). Da der Wert nicht sofort ausgeliefert wird, beispielsweise als Rückgabewert, ist der *ausstehend*. Nun kann der Prozess von diesem Fehler unterrichtet werden, indem er entweder in `select(2)` blockiert und durch das Anstehen von Daten zurückkehrt oder das Signal `SIGIO` erhält, wenn diese Form der E/A-Behandlung genutzt wird. Nun kann der Prozess den Wert des Fehlers abfragen, indem er die Option `SO_ERROR` verwendet und den Fehlercode in `val` untersucht. Anschließend wird der Fehlercode zurückgesetzt.

SO_KEEPALIVE

Diese Option hat nur Einfluß auf verbindungsorientierte Protokolle. Besteht eine Verbindung, jedoch wurden für einen gewissen Zeitraum keine Daten übertragen, so sendet das Transportprotokoll automatisch ein Paket (*probe*), das die Gegenstelle beantworten muß. Entweder antwortet die Gegenstelle mit einem ACK und die Applikation wird nicht benachrichtigt, oder die Gegenstelle antwortet mit einem RST, was bedeutet, daß sie nicht mehr verfügbar ist (der Fehlercode für `SO_ERROR` wird auf `ECONNRESET` gesetzt) oder die Gegenstelle antwortet gar nicht. Nach einiger Zeit (die leider abhängig von der Implementierung ist), sendet das Transportprotokoll einen weiteren Probe und gibt schließlich auf.

SO_LINGER

Diese Option bestimmt, wie die Funktion `close(2)` ein verbindungsorientiertes Protokoll behandelt. Wird `close(2)` aufgerufen, so werden alle Daten, die sich noch im Sendepuffer befinden an die Gegenstelle ausgeliefert, aber keine neuen Daten in diesen Speicher geschrieben. Die `SO_LINGER`-Option erlaubt uns, das Verhalten zu beeinflussen, erfordert aber die Übergabe eines Zeiger auf eine `linger`-Struktur, die wir durch Einfügen von `<sys/socket.h>` erhalten:

```
struct linger {
    int l_onoff; /* 0 = off, nonzero = on */
    int l_linger; /* linger time in seconds */
};
```

Setzen wird `l_onoff` auf 0, kehrt `close(2)` sofort zurück und das Transportprotokoll verhält sich wie vereinbart (`l_linger` wird ignoriert). Schalten wir das Lingering ein, setzen aber `l_linger` auf 0, kehrt `close(2)` auch hier sofort zurück, jedoch werden alle Daten im Sendepuffer verworfen und ein RST wird abgesetzt, die Verbindung also abgebrochen. Damit durchläuft die Verbindung nicht den `TIME_WAIT`-Zustand. Ist `l_linger` nicht 0 so hängt der Socket noch die angegebene Zeitspanne herum und der Prozess wird solange schlafen geschickt, bis alle Daten übertragen wurden oder der Timer abgelaufen ist. Wird diese Option mit einem nicht blockierendem Socket Descriptor

verwendet, so kehrt `close(2)` sofort zurück und wir müssen den Rückgabewert auf `EWOULDBLOCK` prüfen, der anzeigen, daß noch immer Daten im Sendepuffer vorhanden sind.

`SO_OOBINLINE`

Diese Option sorgt dafür, daß OOB-Daten in die normale Eingangswarteschlange eingeordnet werden. Die Daten können jetzt nicht mehr mit `MSG_OOB`, beispielsweise mit `recv(2)`, gelesen werden.

`SO_RCVBUF` und `SO_SNDBUF`

Solange die Anwendung eintreffende Daten nicht mit einem der entsprechenden Systemaufrufe liest, verbleiben sie im Empfangspuffer. Die Größe des Empfangspuffers wird der Gegenstelle durch die Fenstergröße mitgeteilt. Auf diese Weise kann der Puffer nicht überlaufen, da die Gegenstelle nicht mehr Daten senden darf, und wenn doch, werden sie einfach verworfen. Das funktioniert nur so mit TCP, UDP verhält sich anders, denn wenn die Daten nicht in den Puffer passen werden sie verworfen ohne daß die Gegenstelle das bemerkt, denn sie kennt die Größe des Puffers nicht. Die Standardgrößen der Empfangs- und Sendepuffer, welche abhängig von der Implementierung sind, lassen sich mit diesen Optionen einstellen. Da bei TCP bereits während des Handshakes die Fenstergröße verändert werden kann, muß der Client diese Optionen vor dem Aufruf von `connect(2)` setzen, da sonst kein Einfluß auf das sog. *Window Scaling* genommen werden kann. Der Server wiederum muß diese Optionen vor dem Aufruf von `listen(2)` setzen, da ein verbundener Socket nicht zurückkehrt bevor der Handshake abgeschlossen ist, dann aber bereits das Window Scaling ausgehandelt wurde. Die Einstellung wird auf den verbundenen Socket übertragen. Als Faustregel gilt, daß die Puffer mindestens vier mal so groß sein sollten, wie die maximale Segmentgröße (MSS).

`SO_RCVLOWAT` und `SO_SNDDLOWAT`

Die beiden sog. *Niedrigwasserstandsmarkierungen* (*low-water marks*) für den Sende- und Empfangspuffer geben Aufschluß darüber, wie viele Daten ein Puffer enthalten muß, damit er lesbar bzw. schreibbar ist. Für den Lesepuffer genügt standardmäßig bereits ein Byte. Die Wasserstandsmarkierung für Schreibpuffer zeigt an, wie viel Platz er mindestens aufweisen muß, damit er als schreibbar akzeptiert wird. Wo die Untergrenze liegt, hängt von der Implementierung ab. Für verbindungslose Protokolle ist der Sendepuffer immer schreibbar, da keine Kopie der Daten vorgehalten wird.

`SO_RCVTIMEO` und `SO_SNDDTIMEO`

Diese beiden Optionen unterstellen die Lese- und Schreiboperationen mit dem betreffenden einem Timeout. Als Wert für den Parameter `val` von `setsockopt(2)` wird eine `timeval`-Struktur übergeben, wie wir sie bereits in `select(2)` kennengelernt haben. Wie in diesem Systemaufruf schalten wir den Timeout durch das Setzen beider Member auf 0 aus. Alle lesenden und schreibenden Systemaufrufe sind von diesem Timeout betroffen.

`SO_REUSEADDR` und `SO_REUSEPORT`

Die Option `SO_REUSEADDR` hat Einfluß auf folgende Szenarien:

- Ein Server kann seinen Standardport binden (`bind(2)`), obwohl dieser bereits von einer anderen Instanz verwendet wird und bereits eine Verbindung aufgebaut wurde.
- Sie erlaubt einem Server den gleichen Port zu binden wie eine andere Instanz, die mit `INADDR_ANY` gestartet wurde, solange eine andere lokale IP-Adresse verwendet wird.
- Ein Prozess kann den gleichen Port an mehrere Sockets binden, so langen bei jedem Aufruf von `bind(2)` eine andere IP-Adresse angegeben wird.
- Die Option erlaubt doppelte Verwendung von IP-Adresse und Port, was normalerweise nur mit UDP im Zusammenhang mit Multicasting verwendet wird.

`SO_REUSEADDR` wird als equivalent zu `SO_REUSEPORT`, wenn die gebundene IP-Adresse eine Multicast-Adresse ist.

`SO_TYPE`

Liefert den Socket-Typ zurück, der beim Aufruf von `socket(2)` übergeben wurde. Dabei handelt es sich um eine der `SOCK_XXX`-Konstanten, wie beispielsweise `SOCK_STREAM`.

`SO_USELOOPBACK`

Sockets in einer sog. *Routing Domain*, gekennzeichnet durch `AF_ROUTE`, können mit dieser Option festlegen, ob sie Kopien aller Daten erhalten möchten, die an den Socketn gehen.

B.2 Socketoptionen für IPPROTO_IP

Es folgt eine Erläuterung der Socketoptionen für die Schicht IPPROTO_IP.

IP_HDRINCL

Normalerweise überlassen wir das Erstellen des IP-Headers dem Kernel. Wenn wir jedoch mit Raw-Sockets arbeiten, müssen wir das selbst übernehmen und zeigen das durch setzen dieses Flags an. Es gibt Felder im Header, die wir nicht selbst setzen dürfen oder setzen müssen, darunter die Prüfsumme, welche immer von IP errechnet wird, das IP-ID-Feld, welches mit 0 belegt wird, wenn wir wollten, daß der Kernel das übernimmt. Setzen wir die IP-Adresse auf INADDR_ANY, setzt der Kernel die primäre IP-Adresse des ausgehenden Interfaces ein. Das Setzen der Optionen kann entweder manuell geschehen oder durch Verwendung der Socketoption IP_OPTIONS.

IP_OPTIONS

Erlaubt das Setzen der IP-Optionen (40 Bytes) im IP-Header. Das Format ist in RFC791 definiert. Optionen die gesetzt werden können sind:

- **End of option list** – wird benötigt, wenn die Optionen nicht am Ende des Headers aufhören.
- **No operation (NOP)** – wird für die Ausrichtung von Oktets in der Liste von Optionen benötigt.
- **Sicherheitsoptionen** (veraltet).
- **Loose source routing (LSR)** – Legt die Route fest, die ein Datagramm auf Pfad bis zum Ziel passieren muß. Diese Option erlaubt mehrere Hops zwischen zwei beliebigen Knoten auf dem Pfad.
- **Record route** – Wird für das Aufzeichnen von Routen benötigt.
- **Strict source routing** – Wie LSR, allerdings ist keine Abweichung von der vorgeschriebenen Route erlaubt.
- **Internet timestamp** – Zeitstempel des Datagrams. Wird während des Transport verändert.
- Basic security, Extended security und Stream identifier sind veraltet und sollten nicht mehr gesetzt werden, da sie EINVAL.

Grundsätzlich ist das Setzen von Optionen im IP-Header eine defizile Angelegenheit. Neben den Optionen sind einige Felder im IP-Header in *Host Byte Order* andere in *Network Byte Order* zu setzen.

IP_RECVDSTADDR

Sorgt dafür, daß die Ziel-IP-Adresse eines empfangenen als Ergänzungsinformation von `recvmsg(2)` zurückgegeben wird.

IP_RECVIF

Diese Option sorgt dafür, daß der Index der Eingangsschnittstelle als Zusatzinformation von `recvmsg(2)` zurückgegeben wird.

IP_TOS

Diese Option erlaubt das Setzen des TOS-Feldes des IP-Headers für TCP, UDP und SCTP. Wenn wir `getsockopt(2)` aufrufen, erhalten wir den Wert, den IP für ausgehende Datagramme setzen würde, aber nicht den TOS-Wert des gerade empfangenen Datagrams.

IP_TTL

Mit dieser Option können wir die TTL abrufen oder setzen. Für Multicast-Pakete verwenden wir IP_MULTICAST_IP. Ebenso wie mit IP_TOS gibt `getsockopt(2)` immer nur den Wert zurück, den das System setzen würde, aber nicht den des zuletzt empfangenen Datagrams.

IP_ADD_MEMBERSHIP

Sorgt dafür, daß wir eine Multicast-Gruppe beitreten und den Empfang die über eine lokale Schnittstelle spezifizieren. Dazu ist es notwendig eine `ip_mreq`-Struktur zu definieren und sie `setsockopt(2)` zu übergeben. Die Struktur hat folgendes Format:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* IPv4 class D multicast addr */
    struct in_addr imr_interface; /* IPv4 addr of local interface */
};
```

Praktisch sieht das folgendermaßen aus. Um der Multicast-Gruppe 225.1.1.1 beizutreten und über die lokale Schnittstelle 12.10.13.11 zu empfangen schreiben wir:

```
struct ip_mreq mcast;

mcast.imr_multiaddr.s_addr = inet_addr("225.1.1.1");
mcast.imr_interface.s_addr = inet_addr("12.10.13.11");
setsockopt_ex(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    (char *)&mcast, sizeof(mcast))
```

Geben wir für dem Member `imr_interface.s_addr` die Konstante `INADDR_ANY` an wählt der Kernel ein passendes Interface für den empfang der Multicast-Daten aus.

IP_DROP_MEMBERSHIP

Sobald der Socket geschlossen wird, verlassen wir automatisch auch alle Multicast-Gruppen, die über diese Socket abgewickelt wurden. Mit der Option `IP_DROP_MEMBERSHIP` können wir das auch programmatisch erledigen:

```
struct ip_mreq mcast;

mcast.imr_multiaddr.s_addr = inet_addr("225.1.1.1");
mcast.imr_interface.s_addr = inet_addr("12.10.13.11");
setsockopt_ex(fd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
    (char *)&mcast, sizeof(mcast));
```

In diesem Beispiel verlassen wir die Gruppe 225.1.1.1, welche über das Interface 12.10.13.11 abgewickelt wird. Geben wir hier `INADDR_ANY` an, wird die erst beste Multicast-Gruppe verlassen, die zu dem Interface paßt.

IP_ADD_SOURCE_MEMBERSHIP

Sorgt dafür, daß wir einer Multicast-Gruppe einer bestimmten Quelle betreten. Dazu übergeben wir `setsockopt(2)` eine definierte `ip_mreq_source`-Struktur:

```
struct ip_mreq_source {
    struct in_addr imr_multiaddr; /* IPv4 class D multicast addr */
    struct in_addr imr_sourceaddr; /* IPv4 source addr */
    struct in_addr imr_interface; /* IPv4 addr of local interface */
};
```

Um beispielsweise der Gruppe 225.1.1.1 beizutreten, die vom Server 80.12.81.13 gespeist wird, geben wir

```
struct ip_mreq_source smcast;

smcast->imr_multiaddr.s_addr = inet_addr("225.1.1.1");
smcast->imr_sourceaddr.s_addr = inet_addr("80.12.81.13");
smcast->imr_interface.s_addr = INADDR_ANY;
setsockopt_ex(fd, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP,
    (char *)&smcast, sizeof(smcast));
```

Statt `INADDR_ANY` kann natürlich auch ein spezifisches Interface ausgewählt werden über das die Multicast-Anwendung abgewickelt werden soll.

IP_DROP_SOURCE_MEMBERSHIP

Diese Option sorgt dafür, daß wir eine quellspezifische Multicast-Mitgliedschaft beenden. Geben wir für das lokale Interface `INADDR_ANY` an, so wird das erste Interface auf das die Multicast-Mitgliedschaft zutrifft ausgewählt und abgemeldet. Das Verfahren ist das gleiche wie mit `IP_DROP_MEMBERSHIP`.

IP_MULTICAST_IF

Normalweise werden Multicast-Datagramme über das mit der Default Route assoziierte Interface gesendet, sofern sie eingerichtet ist. Alternativ kann auch ein anderes Interface als Default Route für Multicast-Anwendungen spezifiziert werden, was wir mit dieser Option erledigen:

```
struct in_addr addr;

addr.s_addr = inet_addr("192.168.1.2");
setsockopt_ex(fd, IPPROTO_IP, IP_MULTICAST_IF,
              (char *)&addr, sizeof(addr));
```

In diesem Fall wird das Interface mit der IP-Adresse 192.168.1.2 als Default für Multicasting ausgewählt.

IP_MULTICAST_TTL

In der Regel sind Multicast-Datagramme für Systeme des lokalen Netzwerks bestimmt. Müssen sie jedoch mehrere Stationen durchlaufen, um weiter entfernte Systeme zu erreichen, muß unter Umständen die TTL angepaßt werden. Dazu dient uns die Socketoption IP_MULTICAST_TTL:

```
unsigned char ttl = 64;
setsockopt_ex(s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Ist der TTL-Wert 0, so ist die Zustellung auf das lokale System beschränkt. Je höher die TTL-Werte (maximal 255) um so mehr Knoten können erreicht werden. Ein Wert von 1 beschränkt die Zustellung auf das lokale Netzwerk.

IP_MULTICAST_LOOP

Wird ein Multicast-Datagram nur lokal zugestellt (die Applikation schreibt in einen lokalen Port und ein Client liest von diesem) so wird eine Kopie des Datagrams an die Applikation ausgeliefert. Mit der Option IP_MULTICAST_LOOP kann dieses Verhalten deaktiviert werden:

```
unsigned char loop = 0;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

Nur wenige Applikationen deaktivieren das Loopbacking, da nur ein geringer Leistungszuwachs verzeichnet werden kann, aber gleichzeitig bestimmte Diagnosemöglichkeiten dann nicht mehr zur Verfügung stehen.

IP_BLOCK_SOURCE

Erlaubt das Sperren spezifischer Quellen in einer Multicast-Mitgliedschaft, die nicht notwendigerweise mit IP_ADD_SOURCE_MEMBERSHIP definiert werden mußte, sondern für jede beliebige Quelle der Mitgliedschaft funktioniert:

```
struct ip_mreq_source smcast;

smcast.imr_multiaddr = inet_addr("224.0.0.199");
smcast.imr_sourceaddr = inet_addr("80.12.81.13");
setsockopt_ex(fd, IPPROTO_IP, IP_UNBLOCK_SOURCE,
              (char *)&smcast, sizeof(smcast));
```

In diesem Beispiel deaktivieren wir Multicast-Datagramme, die von dem Server mit der Quell-IP-Adresse 80.12.81.13 stammen.

IP_UNBLOCK_SOURCE

Ebenso wie das Sperren einer Multicast-Quelle erlaubt die Option IP_UNBLOCK_SOURCE die Aufhebung derselben nach dem gleichen Prinzip wie wir zuvor gesehen haben.

B.3 Socketoptionen für IPPROTO_ICMPV6

Es folgt eine Erläuterung der Socketoptionen für die Schicht IPPROTO_ICMPV6.

ICMP6_FILTER

Diese Option erlaubt das Konfigurieren eines Filters für die insg. 256 ICMPv6-Nachrichten. Nur die nicht im Filter spezifizierten Nachrichten, werden dem Prozess, der mit einem Raw-Socket verbunden ist, zugestellt.

Für die Verwaltung des Filters sind sechs Makros vorgesehen:

```
#include <netinet/icmp6.h>

void ICMP6_FILTER_SETPASSALL(struct icmp6_filter *filt);
void ICMP6_FILTER_SETBLOCKALL(struct icmp6_filter *filt);
void ICMP6_FILTER_SETPASS(int msgtype, struct icmp6_filter *filt);
void ICMP6_FILTER_SETBLOCK(int msgtype, struct icmp6_filter *filt);

int ICMP6_FILTER_WILLPASS(int msgtype, const struct icmp6_filter *filt);
int ICMP6_FILTER_WILLBLOCK(int msgtype, const struct icmp6_filter *filt);

Rückgabewert: 1 wenn Filter die Nachricht blockiert/zuläßt oder 0 bei Fehler.
```

ICMP6_FILTER_SETPASSALL

Reiche alle ICMPv6-Nachrichten an die Applikation durch.

ICMP6_FILTER_SETBLOCKALL

Blockiere alle ICMPv6-Nachrichten und reiche keine der Applikation durch.

ICMP6_FILTER_SETPASS

Reiche Nachrichten bestimmten ICMPv6-Typs an die Applikation durch.

ICMP6_FILTER_SETBLOCK

Blockiere Nachrichten bestimmten ICMPv6-Typs und reiche sie nicht an die Applikation durch.

ICMP6_FILTER_WILLPASS

Gibt `true` oder `false` zurück, abhängig davon, ob der angegebene Nachrichtentyp an die Applikation durchgereicht wird.

ICMP6_FILTER_WILLBLOCK

Gibt `true` oder `false` zurück, abhängig davon, ob der angegebene Nachrichtentyp blockiert und nicht an die Applikation durchgereicht wird.

Alle sechs Makros erwarten eine `icmp6_filter`-Struktur als Parameter und zwei einen Integerwert, der ein spezifische Nachricht identifiziert. Die Struktur ist folgendermaßen in `<netinet/icmp6.h>` definiert:

```
struct icmp6_filter {
    u_int32_t icmp6_filt[8];
};
```

Die Makros modifizieren das Integer-Array beispielsweise so:

```
#define ICMP6_FILTER_SETPASSALL(filterp) \
    memset(filterp, 0xff, sizeof(struct icmp6_filter))
#define ICMP6_FILTER_SETBLOCKALL(filterp) \
    memset(filterp, 0x00, sizeof(struct icmp6_filter))

#define ICMP6_FILTER_SETPASS(type, filterp) \
    (((filterp)->icmp6_filt[(type) >> 5]) |= (1 << ((type) & 31)))
#define ICMP6_FILTER_SETBLOCK(type, filterp) \
    (((filterp)->icmp6_filt[(type) >> 5]) &= ~(1 << ((type) & 31)))
```

```
#define ICMP6_FILTER_WILPPASS(type, filterp) \
    (((filterp)->icmp6_filt[(type) >> 5]) & (1 << ((type) & 31))) != 0
#define ICMP6_FILTER_WILLBLOCK(type, filterp) \
    (((filterp)->icmp6_filt[(type) >> 5]) & (1 << ((type) & 31))) == 0
```

Um nun beispielsweise die Nachricht ICMP6_ECHO_REQUEST durchzulassen, aber alle anderen nicht, gehen wir folgendermaßen vor:

```
struct icmp6_filter filter;
fd = socket_ex(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6);

ICMP6_FILTER_SETBLOCKALL(&filter);
ICMP6_FILTER_SETPASS(ICMP6_ECHO_REQUEST, &filter);
Setsockopt(fd, IPPROTO_ICMPV6, ICMP6_FILTER, &filter, sizeof(filter));
```

sowohl ICMPv4- als auch ICMPv6-Nachrichten werden durch ein Type- und Code-Feld eindeutig identifiziert. Die ICMPv6-Typen und ihre Codes sind in Tabelle B.2 zusammengefaßt.

Type	Code	Konstante	Beschreibung
1	0	ICMP6_DST_UNREACH_NOROUTE	Destination unreachable
	1	ICMP6_DST_UNREACH_ADMIN	No route to destination
	2	ICMP6_DST_UNREACH_BEYONDSCOPE	Administratively prohibited
	3	ICMP6_DST_UNREACH_ADDR	Beyond scope of source address
	4	ICMP6_DST_UNREACH_NOPORT	Address unreachable
2		ICMP6_PACKET_TOO_BIG	Port unreachable
3	0	ICMP6_TIME_EXCEED_TRANSIT	Packet too big
	1	ICMP6_TIME_EXCEED_REASSEMBLY	Time exceeded
4	0	ICMP6_PARAMPROB_HEADER	TTL == 0 in transit
	1	ICMP6_PARAMPROB_NEXTHEADER	TTL == 0 in reassembly
	2	ICMP6_PARAMPROB_OPTION	Invalid IPv6 header
128	0	ICMP6_ECHO_REQUEST	Erroneous header field
129	0	ICMP6_ECHO_REPLY	Unrecognized next header
130	0	MLD_LISTENER_QUERY	Unrecognized option
131	0	MLD_LISTENER_REPORT	Echo service
132	0	MLD_LISTENER_DONE	Echo reply
133	0	ND_ROUTER_SOLICIT	Multicast listener query
134	0	ND_ROUTER_ADVERT	Multicast listener report
135	0	ND_NEIGHBOR_SOLICIT	Multicast listener done
136	0	ND_NEIGHBOR_ADVERT	Router solicitation
137	0	ND_REDIRECT	Neighbor advertisement
138	0	ICMP6_ROUTER_RENUMBERING	Redirect
139	0	ICMP6_WRUREQUEST	Router renumbering
140	0	ICMP6_WRUREQUEST	Who are you request
139	0	ICMP6_FQDN_QUERY	Who are you reply
140	0	ICMP6_FQDN_REPLY	FQDN query
139	0	ICMP6_NI_QUERY	FQDN reply
140	0	ICMP6_NI_REPLY	Node information request
			Node information reply

Tabelle B.2: ICMPv6 Type- und Code-Fields.

B.4 Socketoptionen für IPPROTO_IPV6

Es folgt eine Erläuterung der Socketoptionen für die Schicht IPPROTO_IPV6.

IPV6_CHECKSUM

Legt den Byte-Offset fest, der auf den Speicherbereich in den Nutzdaten zeigt, der die Prüfsumme enthält. Ist der Wert nicht negativ, so wird für alle ausgehenden Pakete eine Prüfsumme berechnet und gespeichert. Zusätzlich werden die Prüfsummen aller eingehenden Pakete geprüft und bei nicht Übereinstimmung verworfen. Diese Option muß von allen Anwendungen, die IPv6 Raw-Sockets verwenden, gesetzt werden. Ausgenommen sind ICMPv6 Raw-Sockets.

Ein Beispiel:

```
int offset = 0x02;
setsockopt_ex(fd, IPPROTO_IPV6, IPV6_CHECKSUM, &offset, sizeof(offset))
```

Damit wird der Kernel angewiesen, eine Prüfsumme für ausgehenden Pakete zu ermitteln und zu speichern und eingehende Pakete zu prüfen.

IPV6_DONTFRAG

Diese Option schaltet die Fragmentierung großer Pakete ab, so daß alle Pakete, die größer als die MTU sind, verworfen werden. Da dieser Fehler auf dem Pfad zwischen Quelle und Ziel auftreten soll, wird kein Fehler zurückgegeben, sondern die Option IPV6_RECVPATHMTU setzen, damit die MTU erkannt werden kann.

IPV6_NEXTHOP

Erlaubt es uns, eine IPv6-Adresse des ersten Hops anzugeben, der allerdings ein Nachbar des Absenders sein muß. Der Parameter für `setsockopt(2)` ist ein Zeiger auf eine `sockaddr_in6`-Struktur. Stimmt die Adresse des Hops mit der Zieladresse überein, entspricht diese Option dem Setzen von `SO_DONTROUTE` (Abschnitt B.1).

IPV6_PATHMTU

Diese Option kann nicht gesetzt werden und liefert die aktuelle MTU zurück, die der Kernel mit Hilfe von *Path MTU Discovery* ermittelt hat.

IPV6_RECVDSTOPTS

Diese Option sorgt dafür, daß alle Optionen eines IPv6-Ziels als Zusatzinformationen mit `recvmsg(2)` zurückgeliefert werden. Standardmäßig ist die Option deaktiviert.

IPV6_RECVHOPLIMIT

Gibt an, daß das Hop-Limit-Feld des empfangenen Pakets als Zusatzinformationen mit `recvmsg(2)` zurückgegeben werden sollen. Die Option ist standardmäßig deaktiviert.

IPV6_RECVPATHMTU

Ist dieses Flag gesetzt, zeigt es an, daß die MTU des Pfades als Zusatzinformation mit `recvmsg(2)` zurückgegeben werden soll, sobald sie sich ändert. Dazu wird ein leeres Datagramm erstellt (0 Byte Länge), das lediglich die gewünschte Information als *Ancillary Data* mit zieht.

IPV6_RECVPKTINFO

Mit Hilfe dieser Option können die IP-Adresse der Gegenstelle und das eingehende Interface bestimmt werden. Diese Information wird als Zusatzinformation mit `recvmsg(2)` geliefert.

IPV6_RECVRTHDR

Diese Option sorgt dafür, daß ein empfangener IPv6 Routing Header als Zusatzinformation mit `recvmsg(2)` zurückgegeben wird. Standardmäßig ist die Option nicht aktiv.

IPV6_RECVTCLASS

Liefert die Traffic Class (die Felder DSCP und ECN) des empfangenen Pakets als Zusatzinformation mit `recvmsg(2)` zurück. Standardmäßig ist die Option nicht aktiv.

IPV6_UNICAST_HOPS

Setzt das standardmäßige Hop-Limit für ausgehende Datagramme oder, wenn die Option mit `getsockopt(2)` genutzt wird liefert sie das Standard-Hop-Limit zurück, das der Kernel mit diesem Socket verwenden würde. Der tatsächliche Hop-Count wird mit `IPV6_RECVHOPLIMIT` zurückgegeben.

Um die Anzahl der Unicast-Hops auf 10 zu begrenzen gehen wir folgendermaßen vor:

```
int hoplimit = 10;
setsockopt_ex(fd, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
              (char *) &hoplimit, sizeof(hoplimit));
```

Möchten wir die aktuelle Einstellung abfragen, schreiben wir:

```
int hoplimit;
size_t len = sizeof(hoplimit);

getsockopt_ex(fd, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
              (char *) &hoplimit, &len);
printf("Using %d for hop limit.\n", hoplimit);
```

IPV6_USE_MIN_MTU

Wird diese Option eingeschaltet, weisen wir den Kernel an, keine Path MTU Discovery durchzuführen, was eigentlich Standard ist, sondern stets die kleinste IPv6-MTU zu verwenden, um Fragmentierung zu vermeiden. Für Multicast-Anwendungen wird immer die kleinste MTU verwendet, wenn wir -1 angeben. Unicast-Anwendungen profitieren dann von Path MTU Discovery.

IPV6_V6ONLY

Setzen wir diese Option, so ist nur noch IPv6-Kommunikation über diesen Socket möglich. Standardmäßig kann auch IPv4 über einen AF_INET6-Socket abgewickelt werden.

IPV6_MULTICAST_IP

Gibt das ausgehende Interface für Multicast-Datagramme an. Als Parameter übergeben wir setsockopt(2) den Interface-Index (nur IPv6, IPv4 erfordert die Angabe einer `sockaddr_in`-Struktur). Übergeben wir einen Index von 0 so wählt der Kernel eine passende Schnittstelle aus, entfernt aber alle zuvor durch die Option spezifizierten Schnittstellen.

IPV6_MULTICAST_HOPS

Legt die Anzahl der Hops fest. Wird der Wert auf 0 gesetzt, bleiben Multicast-Datagramme auf den Host beschränkt. Ein Wert von 1 erlaubt die Übertragung ins lokale Subnetz. Standardmäßig beträgt der Wert 1.

```
int hoplimit = 255; /* hop limit = -1 sets to default of 1 */
setsockopt_ex(fd, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
              (char *)&hoplimit, sizeof(hoplimit));
```

IPV6_MULTICAST_LOOP

Aktiviert oder deaktiviert die Zustellung von Kopien der Multicast-Datagramme, so daß ein Prozess der Multicast-Datagramme absetzt selbst auch Kopien dieser Datagramme erhält. Diese Option ist standardmäßig aktiviert.

IPV6_JOIN_GROUP

Erlaubt einem Host einer Multicast-Gruppe beizutreten. Als Parameter für setsockopt(2) übergeben wir eine `ipv6_mreq`-Struktur:

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; /* IPv6 multicast address */
    unsigned int     ipv6mr_interface; /* interface index */
};
```

Praktisch könnte das etwa so aussehen (`if_index` enthält den Index der gewünschten Schnittstelle):

```
struct ipv6_mreq imr6;

imr6.ipv6mr_interface = if_index;
setsockopt_ex(fd, IPPROTO_IPV6, IPV6_JOIN_GROUP,
              (char *)&imr6, sizeof(imr6));
```

IPV6_LEAVE_GROUP

Kündigt die Mitgliedschaft in einer IPv6 Multicast-Gruppe auf. Dazu übergeben wir eine `ipv6_mreq`-Struktur, die die gleichen Angaben enthält, mit der wir einer Mitgliedschaft beigetreten sind:

```
struct ipv6_mreq imr6 = NULL;
/* have joined multicast group, now leaving */
if (setsockopt(s, IPPROTO_IPV6, IPV6_LEAVE_GROUP,
    (char *)&imr6, sizeof(imr6))
```

Spezifizieren wir in der `ipv6_mreq`-Struktur einen Interface-Index von 0 wird das erste passende Interface vom Kernel ausgewählt und die Mitgliedschaft beendet, sofern vorhanden.

B.5 Socketoptionen für IPPROTO_IPV6 und IPPROTO_IP

Es folgt eine Erläuterung der Multicast-Socketoptionen für die Schichten IPPROTO_IPV6 und IPPROTO_IP.

MCAST_JOIN_GROUP

Sorgt dafür, daß der Host einer Multicast-Gruppe beitritt. Dazu übergeben wir `setsockopt(2)` eine `group_req`-Struktur:

```
struct group_req {
    uint32_t gr_interface; /* interface index */
    struct sockaddr_storage gr_group; /* group address */
};
```

Um beispielsweise der Gruppe 224.0.0.2 beizutreten, schreiben wir:

```
struct addrinfo hints, *res;
struct group_req greq;
char *source, *group;

greq.grs_interface = if_nametoindex("eth0"); /* Linux */
getaddrinfo_ex(group, NULL, &hints, &res);
memcpy(&greq.grs_group, res->ai_addr, res->ai_addrlen);
freeaddrinfo(res);

/* Assume IPv6 socket here, for IPv4 use IPPROTO_IP */
setsockopt_ex(s, IPPROTO_IPV6, MCAST_JOIN_GROUP,
    (char *)&greq, sizeof(greq));
```

Beachten Sie, daß der Member `gr_group` eine `sockaddr_storage`-Struktur ist. Sie ist groß genug, um alle vom System unterstützten Protokolle zu verarbeiten.

MCAST_LEAVE_GROUP

Kündigt die Mitgliedschaft in einer Multicast-Gruppe auf. Ihr übergeben wir eine `group_req`-Struktur mit den gleichen Daten, mit denen wir der Gruppe beigetreten sind.

```
setsockopt_ex(fd, slevel, MCAST_LEAVE_GROUP,
    (char *)&greq, sizeof(greq))
```

MCAST_BLOCK_SOURCE und MCAST_BLOCK_SOURCE

Sperrt den Empfang von Multicast-Datagrammen oder hebt die Sperre auf. Wir übergeben `setsockopt(2)` eine `group_source_req`-Struktur, dessen Member `gsr_source` die betreffende Quelle definiert.

Folgende Funktion sperrt die als Parameter übergebene Multicast-Quelle:

```

int mcast_block_source(int sockfd, const SA_PTR src, socklen_t srclen,
                      const SA_PTR grp, socklen_t grplen) {
    struct group_source_req req;
    req.gsr_interface = 0;

    if (grplen > sizeof(req.gsr_group)
        srclen > sizeof(req.gsr_source)) {
        errno = EINVAL;
        return -1;
    }

    memcpy(&req.gsr_group, grp, grplen);
    memcpy(&req.gsr_source, src, srclen);
    return (setsockopt(sockfd, family_to_level(grp->sa_family),
                      MCAST_BLOCK_SOURCE, &req, sizeof(req)));
}

```

MCAST_JOIN_SOURCE_GROUP

Erlaubt das Beitreten einer Multicast-Gruppe auf Basis einer bestimmten Quelle. Wir übergeben `setsockopt(2)` eine `group_source_req`-Struktur, dessen Member `gsr_source` die betreffende Quelle definiert:

```

struct group_source_req {
    uint32_t gsr_interface;           /* interface index */
    struct sockaddr_storage gsr_group; /* group address */
    struct sockaddr_storage gsr_source; /* source address */
};

```

Das folgende Codebeispiel schreibt einen Host mit dem ersten Interface (`eth0`) einer Multicast-Gruppe mit lokaler Quelle ein:

```

struct addrinfo hints, *res;
struct group_source_req gsreq;
char *source, *group;

gsreq.gsr_interface = if_nametoindex("eth0"); /* Linux */
getaddrinfo_ex(source, NULL, &hints, &res)

memcpy(&gsreq.gsr_source, res->ai_addr, res->ai_addrlen);
freeaddrinfo(res);

getaddrinfo_ex(group, NULL, &hints, &res);

memcpy(&gsreq.gsr_group, res->ai_addr, res->ai_addrlen);
freeaddrinfo(res);

/* Assume IPv6 socket here, for IPv4 use IPPROTO_IP */
setsockopt_ex(s, IPPROTO_IPV6, MCAST_JOIN_SOURCE_GROUP,
              (char *)&gsreq, sizeof(gsreq));

```

MCAST_LEAVE_SOURCE_GROUP

Beendet die Gruppenmitgliedschaft eines Hosts. Wir übergeben `setsockopt(2)` eine `group_source_req`-Struktur, die die gleichen Informationen enthält mit denen wir der Gruppe beigetreten sind.

```

/* Assume IPv6 socket here, for IPv4 use IPPROTO_IP */
setsockopt_ex(s, IPPROTO_IPV6, MCAST_LEAVE_SOURCE_GROUP,
              (char *)&gsreq, sizeof(gsreq));

```

B.6 Socketoptionen für IPPROTO_TCP

Es folgt eine Erläuterung der Socketoptionen für die Schicht IPPROTO_TCP.

TCP_MAXSEG

Mit dieser Socket-Option sind wir in der Lage, die maximale Segmentgröße (MSS, *maximum segment size*). Normalerweise wird sie von der Gegenstelle während des Handshakes übermittelt. Wenn wir die Segmentgröße abfragen, bevor der Socket verbunden wurde, so erhalten wir die Standard-MSS, quasi als ob uns die Gegenstelle keine Größe übermittelt hätte.

Das folgende Beispielprogramm zeigt, wie die MSS abgefragt werden kann.

Listing B.1: MSS programmatisch abfragen.

```

1 #include <netinet/tcp.h>
2 #include "header.h"
3
4 int main(int argc, char **argv) {
5     struct sockaddr_in dest;
6     unsigned int mss;
7     socklen_t sl;
8     int fd;
9
10    if (argc != 3)
11        err_fatal("USAGE: showmss <host_ip> <port>\n");
12
13    fd = socket_ex(AF_INET, SOCK_STREAM, 0);
14    memset(&dest, 0, sizeof(dest));
15    dest.sin_family = AF_INET;
16    dest.sin_addr.s_addr = inet_addr(argv[1]);
17    dest.sin_port = htons(atoi(argv[2]));
18
19    connect_ex(fd, &dest, sizeof(dest));
20    fprintf(stderr, "Connected to %s on port %s !\n",
21            argv[1], argv[2]);
22
23    mss = 0;
24    sl = sizeof(mss);
25
26    getsockopt_ex(s, IPPROTO_TCP, TCP_MAXSEG, &mss, &sl);
27
28    printf(stdout, "MSS = %u (sizeof(MSS) = %d)\n", mss, sl);
29    fflush(stdout);
30    close(fd);
31
32    return 0;
33 }
```

Listing B.1: src/showmss.c - MSS programmatisch abfragen.

TCP_NODELAY

Diese Socketoption deaktiviert den Nagle-Algorithmus¹, was sich durch erhöhte Netzlast bemerkbar macht, da die Segmente kleiner sind als notwendig. Die Option ist nur für Anwendungen

¹Der Nagle-Algorithmus ist benannt nach John Nagle. Er wird in der Regel im TCP-Teil eines Netzwerkprotokolls eingesetzt. Der Nagle-Algorithmus soll zu kleine Pakete verhindern, bei denen die zusätzliche Last durch Header, etc. wesentlich größer als die tatsächlichen Nutzdaten sind:

- Ist ein Paket voll, schicke es
- Ist ein Paket nicht voll, dann schicke es erst, wenn du genug Daten hast, oder keine unbestätigten Pakete mehr unterwegs sind

Die genaue Definition findet sich in RFC 896 und RFC 1122.

interessant, die hin und wieder nur wenige Informationen übertragen müssen und keine unmittelbare Antwort (ACK) erwarten, aber die Daten so schnell wie möglich übermitteln müssen. Auf diese Weise soll der Durchsatz kurzfristig erhöht werden, da die ACKs des Servers unmittelbar nach Eingang der Segmente abgesetzt werden und nicht erst mit zurückgesendeten Nutzdaten. Diese Option ist standardmäßig aktiviert.

Wenn möglich, sendet TCP die ACKs eingegangener Segmente Huckepack (*piggyback*), das heißt mit zurückgesendeten Nutzdaten. Server, die kaum Daten an die Clients senden, verursachen auf Client-Seite erhebliche Verzögerungen, da keine weiteren Segmente vom Client gesendet werden können, solange der serverseitige ACK-Timer nicht abgelaufen ist. Für solche Client ist diese Option gedacht.

B.7 Socketoptionen für IPPROTO_SCTP

Es folgt eine Erläuterung der Socketoptionen für die Schicht IPPROTO_SCTP. Da SCTP ein noch relativ junges Protokoll ist, kann es vorkommen, daß es selbst auf recht aktuellen Systemen noch nicht vollständig verfügbar ist, bzw. sich die API noch nicht stabilisiert hat. Die Systeme AIX5r3, Linux 2.6, Solaris 10 und HP-UX 11.0 unterstützen SCTP vollständig.

Mit dem neuen Protokoll hat auch die Funktion `sctp_opt_info(3)` Einzug gehalten. Sie ist besser für die Interaktion mit SCTP geeignet als `getsockopt(2)`, da es nicht immer den Datenaustausch zwischen Kernel und User Mode ermöglicht. Um größtmögliche Portabilität zu gewährleisten, sollten wir zur Abfrage von SCTP-Details immer `sctp_opt_info(3)` verwenden:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sctpapi.h>

int sctp_opt_info(int sd, sctp_assoc_t id, int opt,
    void *arg_size, size_t *size);
```

Rückgabewert: 0 bei Erfolg oder -1 bei Fehler.

sd

Socket Descriptor, den wir von `socket(2)` erhalten haben.

id

Spezifiziert die ID der Assoziation über die wir Informationen erhalten möchten.

opt

Zeigt an, welche Socketoption wir abrufen möchten.

arg_size

Zeiger auf eine optionsspezifische Datenstruktur, die der Aufrufer bereitstellen muß.

arg_size

Gibt die Größe der zurückzugebenden Option an.

Für Sockets mit mehreren Assoziationen, kann zur Abfrage von Informationen über eine bestimmte Assoziation eine ID angegeben werden. Das ist nützlich, wenn eine bestimmte Adresse der Gegenstelle (definiert durch eine `sockaddr_storage`-Struktur) spezifiziert wird und der Host Multihoming unterstützt. In diesem Fall wird die Operation nur auf diese Assoziation angewendet.

Das folgende Beispiel setzt einige Parameter (`struct sctp_paddrparams`) für alle Adressen einer Assoziation.

```

struct sockaddr_storage *addrs;
struct sctp_paddrparams peer_params;
int n, j, size;

if (n = sctp_getpaddrs(fd, 0, &addrs) < 0)
    perror("sctp_getpaddrs");

for (j = 0; j < n; j++) {
    peer_params.spp_assoc_id = 0;
    memcpy((void *)&(peer_params.spp_address), (const void *)&addrs[j],
           sizeof(struct sockaddr_storage));

    peer_params.spp_hbinterval = 100;
    peer_params.spp_pathmaxrxt = 2;

    size = sizeof(peer_params);
    if (sctp_opt_info(fd, 0, SCTP_SET_PEER_ADDR_PARAMS,
                      (void *)&peer_params, &size) < 0)
        perror("sctp_opt_info");
}

```

Auf die Datenstrukturen, die für die jeweiligen Optionen notwendig sind, werden in den jeweiligen Optionsbeschreibungen erläutert.

SCTP_ADAPTION_LAYER

Sorgt dafür, daß der *Adaption Layer Indicator Parameter* des lokalen Endpunktes für alle zukünftigen INIT- und INIT-ACK-Segmente gesetzt wird.

```

struct sctp_setadaption {
    uint32_t ssb_adaption_ind;
};

```

Der angegebene Indicator wird in jedem ausgehenden Adaption Layer Indication Parameter enthalten sein. Während der Initialisierung der Assoziation darf ein Endpunkt eine sog. Adaption Layer Indication spezifizieren, der es beiden Anwendungen erlaubt den lokalen Adaption Layer zu koordinieren. Dafür muß die Applikation Adaption Layer Events abonnieren.

SCTP_ASSOCINFO

Liefert umfangreiche Informationen über eine Assoziation. Dazu wird eine *sctp_assocparams*-Struktur verwendet.

```

struct sctp_assocparams {
    sctp_assoc_t sasoc_assoc_id;
    uint16_t sasoc_asocmaxrxt;
    uint16_t sasoc_number_peer_destinations;
    uint32_t sasoc_peer_rwnd;
    uint32_t sasoc_local_rwnd;
    uint32_t sasoc_cookie_life;
};

```

Die Member haben folgende Bedeutung:

srto_assoc_id

ID der Assoziation. Muß vom Aufrufer von *getsockopt(2)* spezifiziert werden.

sasoc_asocmaxrxt

Maximale Anzahl von Retransmissionen für diese Assoziation.

sasoc_number_peer_destinations

Anzahl der Adressen des Peers.

sasoc_peer_rwnd

Aktuelle Einstellung des Empfangsfenster des Peers. Zeigt an, wie viele Bytes an das Peer gesendet werden können und verändert sich während der Transmission.

sasoc_local_rwnd

Das letzte lokale dem Peer übermittelte Empfangsfenster.

sasoc_cookie_life

Wenn Cookies an Peers ausgegeben werden, zeigt dieses Member Cookie-Lebenszeit an.

SCTP_AUTOCLOSE

Erlaubt das Abrufen der Zeit, bis ein Auto Close des Endpunktes durchgeführt wird. Sie Dauer wird in Sekunden angegeben und zeigt an, wenn die Verbindung geschlossen wird, wenn keines der beiden Peers Daten übermittelt. Standardmäßig ist diese Option deaktiviert.

SCTP_DEFAULT_SEND_PARAM

Gibt die Standardeinstellungen der Assoziation zurück, die `sendto(2)` verwendet. Die Informationen werden in einer `sctp_sndrcvinfo`-Struktur gespeichert:

```
struct sctp_sndrcvinfo {
    uint16_t      sinfo_stream;
    uint16_t      sinfo_ssn;
    uint16_t      sinfo_flags;
    uint32_t      sinfo_ppid;
    uint32_t      sinfo_context;
    uint32_t      sinfo_timetolive;
    uint32_t      sinfo_tsn;
    uint32_t      sinfo_cumtsn;
    sctp_assoc_t  sinfo_assoc_id;
};
```

Die Member der Struktur haben folgende Bedeutung:

sinfo_stream

Angabe des Standardstreams für `sendmsg(2)`.

sinfo_ssn

Ist immer 0, wenn die Struktur gesetzt wird. Wenn die Struktur zur Abfrage verwendet wird, enthält dieses Member die Stream Sequence Number (SSN) des SCTP DATA Chunks. Steht zur Verfügung wenn `recvmsg(2)` oder `sctp_recvmsg(2)` aufgerufen wird.

sinfo_flags

Standard-Flags für `sendmsg(2)`. Kann eine der folgenden Konstanten sein: `MSG_UNORDERED`, `MSG_ADDR_OVER`, `MSG_ABORT`, `MSG_EOF` oder `MSG_PR_SCTP`.

sinfo_ppid

Bezeichner, der das transportierte Protokoll im Nutzdatenbereich für `sendmsg(2)` identifiziert.

sinfo_context

Standardkontext für `sendmsg(2)`.

sinfo_timetolive

TTL einer Nachricht in Millisekunden. Die zu sendende Nachricht ist abgelaufen, wenn der Absender mit der Transmission nicht innerhalb einer bestimmten Zeit beginnt. Ein Wert von 0 zeigt an, daß die Nachricht nie abläuft. Wenn `MSG_PR_SCTP` in `sinfo_flags` gesetzt ist, läuft die Nachricht auch ab, wenn sie nicht innerhalb des selben Zeitraums bestätigt wurde.

sinfo_tsn

Wenn die Option abgerufen wird, enthält dieses Member die Transport Sequence Number (TSN) des Peers. Verfügbar nach dem Aufruf von `recvmsg(2)` oder `sctp_recvmsg(2)`.

sinfo_cumtsn

Wird beim Setzen ignoriert, sonst enthält das Member die kummulative TSN des lokale SCTP-Stacks, der mit dem entfernten Peer.

sinfo_assoc_id

ID der Assoziation, dessen Standardoptionen der Aufrufer setzen lassen möchte.

SCTP_DISABLE_FRAGMENTS

Paßt eine Nachricht nicht in ein SCTP-Paket, so wird es in mehrer DATA Chunks aufgeteilt, fragmentiert. Das Verhalten läßt sich auf Senderseite deaktivieren, führt aber dazu, daß EMSGSIZE zurückgeliefert wird, wenn die Nachricht nicht gesendet werden konnte. Standardmäßig ist diese Option nicht aktiviert.

SCTP_EVENTS

Durch das Setzen dieser Option kann der Aufrufer diverse SCTP-Benachrichtigungen abfragen und verwalten. Eine Benachrichtigung wird vom SCTP-Stack an die Anwendung durchgereicht und als ganz normales Datenpaket behandelt, wobei das Flag MSG_NOTIFICATION für msg_flags in recvmsg(2) gesetzt ist. Eine Applikation kann bis zu 8 unterschiedliche Ereignisse abonnieren, die in der sctp_event_subscribe-Struktur zusammengefaßt sind:

```
struct sctp_event_subscribe {
    /* sctp_sendrcvinfo with each request */
    uint8_t sctp_data_io_event;
    /* association notifications */
    uint8_t sctp_association_event;
    /* address notifications */
    uint8_t sctp_address_event;
    /* message send failure notifications */
    uint8_t sctp_send_failure_event;
    /* peer protocol error notifications */
    uint8_t sctp_peer_error_event;
    /* shutdown notifications */
    uint8_t sctp_shutdown_event;
    /* partial delivery API notifications */
    uint8_t sctp_partial_delivery_event;
    /* adaption layer notifications */
    uint8_t sctp_adaption_layer_event;
};
```

Jeder Member ist als Schalter zu verstehen. Wird er auf 1 gesetzt so erhält die Applikation Benachrichtigungen (*notifications*) von genau diesem Ereignis. Ein Wert von 0 schaltet die Benachrichtigung ab.

Praktisch sieht das Setzen der Socketoption folgendermaßen aus:

```
struct sctp_event_subscribe events;

/* Create an SCTP socket here... */

events.sctp_association_event = 1;
events.sctp_data_io_event = 1;
setsockopt_ex(fd, IPPROTO_SCTP, SCTP_EVENTS, &events, sizeof(events));
```

In diesem Beispiel abonnieren wir *Association Notification Events* und möchten, die Struktur sctp_sendrcvinfo bei jedem Aufruf von recvmsg(2) erhalten.

Um einen Event auf der Gegenseite auszulesen könnten wir eine kleine Funktion schreiben:

```
static void handle_event(void *event) {
    union sctp_notification *snp;

    struct sctp_assoc_change *sac;
    struct sctp_send_failed *ssf;
    struct sctp_paddr_change *spc;
    struct sctp_remote_error *sre;
    struct sockaddr_in *sin;
```

```

    struct sockaddr_in6      *sin6;

    char addrbuf [INET6_ADDRSTRLEN];
    const char *ap;

    snp = event;

    switch (snp->sn_header.sn_type) {
        case SCTP_ASSOC_CHANGE:
            sac = &snp->sn_assoc_change;
            printf("assoc_change: state=%hu, err=%hu, instr=%hu "
                   "outstr=%hu\n", sac->sac_state, sac->sac_error,
                   sac->sac_inbound_streams, sac->sac_outbound_streams);
            break;
        case SCTP_SEND_FAILED:
            ssf = &snp->sn_send_failed;
            printf("sendfailed: len=%hu err=%d\n", ssf->ssf_length,
                   ssf->ssf_error);
            break;
        case SCTP_PEER_ADDR_CHANGE:
            spc = &snp->sn_paddr_change;
            if (spc->spc_aaddr.ss_family == AF_INET) {
                sin = (SIN_PTR)&spc->spc_aaddr;
                ap = inet_ntop(AF_INET, &sin->sin_addr, addrbuf,
                               INET6_ADDRSTRLEN);
            } else {
                sin6 = (SIN6_PTR)&spc->spc_aaddr;
                ap = inet_ntop(AF_INET6, &sin6->sin6_addr, addrbuf,
                               INET6_ADDRSTRLEN);
            }
            printf("intf_change: %s state=%d, err=%d\n", ap,
                   spc->spc_state, spc->spc_error);
            break;
        case SCTP_REMOTE_ERROR:
            sre = &snp->sn_remote_error;
            printf("remote_error: err=%hu len=%hu\n",
                   ntohs(sre->sre_error), ntohs(sre->sre_length));
            break;
        case SCTP_SHUTDOWN_EVENT:
            printf("shutdown event\n");
            break;
        default:
            printf("unknown type: %hu\n", snp->sn_header.sn_type);
            break;
    }
}

```

SCTP_GET_PEER_ADDR_INFO

Mit dieser Option können Anwendungen Informationen über eine spezifische Verbindung einer Assoziation abfragen, beispielsweise über die Erreichbarkeit, Fenstergöße, usw. Das INFO-Suffix zeigt an, daß diese Option lediglich lesend auf die Informationen im Stack zugreift. Verwenden Sie zur Abfrage der Informationen nur `sctp_opt_info(2)` und nicht `getsockopt(2)`. Auf diese Weise maximieren Sie die Portabilität Ihrer Software.

Die verwendete Datenstruktur `sctp_paddr_info` hat folgendes Format:

```

struct sctp_paddrinfo {
    sctp_assoc_t          spinfo_assoc_id;
    struct sockaddr_storage spinfo_address;
    int32_t               spinfo_state;
    uint32_t              spinfo_cwnd;
    uint32_t              spinfo_srtt;

```

```

    uint32_t          spinfo_rto;
    uint32_t          spinfo_mtu;
};


```

Bis auf `spinfo_address` und `spinfo_assoc_id` werden alle Member vom Kernel gesetzt.

`spinfo_assoc_id`

Identifiziert die zu befragende Assoziation. `spinfo_address` muß Teil dieser Assoziation sein.

`spinfo_address`

Wird vor der Abfrage der Peer-Informationen von der Applikation gesetzt und spezifiziert die Adresse der Verbindung, die befragt werden soll.

`spinfo_state`

Enthält den Status der jeweiligen Peer-Adresse. Kann entweder `SCTP_ACTIVE`, `SCTP_INACTIVE` oder `SCTP_UNCONFIRMED` sein.

`spinfo_cwnd`

Gibt die aktuelle Fenstergröße zum Zeitpunkt der Abfrage an. Diese Information kann beim Auslesen der `sctp_paddrinfo`-Struktur bereits wieder veraltet sein.

`spinfo_srtt`

Enthält die sRTT (*smoothed round-trip time*) der Adresse des befragten Peers in Milliseunden.

`spinfo_rto`

Enthält den RTO-Wert (*retransmission timeout*) der Adresse des befragten Peers in Milliseunden.

`spinfo_mtu`

Enthält die MTU (*maximum transmission unit*) der Adresse des befragten Peers. Der Wert wird mittels Path MTU Discovery für den Pfad der Verbindung innerhalb der Assoziation ermittelt.

Um die Adressinformationen in einer Assoziation abzufragen, könnten wir beispielsweise schreiben:

```

struct sctp_paddrinfo *ai;
socklen_t size = sizeof(struct sctp_paddrinfo);
sctp_opt_info_ex(socket, assoc_id, SCTP_GET_PEER_ADDR_INFO, ai, &size);

```

`SCTP_INITMSG`

Verwenden Sie diese Optionen um die Einstellungen eines Sockets vor dem Absetzen einer INIT-Nachricht zu setzen oder abzufragen. Die Informationen werden in einer `sctp_initmsg`-Struktur übermittelt oder gespeichert.

```

struct sctp_initmsg {
    uint16_t sinit_num_ostreams;
    uint16_t sinit_max_instreams;
    uint16_t sinit_max_attempts;
    uint16_t sinit_max_init_timeo;
};

```

Die Member der Struktur haben folgende Bedeutungen:

`sinit_num_ostreams`

Ein 16-bit-Wert, der die Anzahl der ausgehenden Streams, die eine Anwendung nutzen möchte, spezifiziert. Da es sich um eine ausgehandelte Größe handelt, muß sie vom Peer mit einer `SCTP_COMM_UP`-Nachricht bestätigt werden. Ein Wert von 0 zeigt die implementierungsabhängige Standardanzahl an.

`sinit_max_instreams`

Gibt oder zeigt an, wie viele eingehende Verbindungen eine Applikation unterstützen möchte. Normalerweise ist der Maximalwert durch das Betriebssystem vorgegeben, aber unter Umständen kann der Wert auch größer sein. Ein Wert von 0 zeigt die Standardeinstellungen an.

sinit_max_attempts

Gibt an, wie oft der SCTP-Endpunkt versuchen soll eine INIT-Nachricht abzusetzen. Wird die INIT-Nachricht nicht bestätigt stuft SCTP den Endpunkt als nicht erreichbar ein. Ein Wert von 0 zeigt die Standardeinstellungen an.

sinit_max_init_timeo

Gibt an, wann ein INIT-Request abläuft. Dieser RTO in Millisekunden angegebene Wert liegt standardmäßig bei 60 Sekunden.

SCTP_MAXBURST

Gibt an, wie viele Segmente in einem Schub übertragen werden dürfen. Dieser Wert ist abhängig von der Implementierung und liegt bei Solaris 10 beispielsweise bei 4 Segmenten und darf maximal 8 betragen. Der Wert wird normalerweise nicht manuell gesetzt, sondern nur zu Testzwecken verwendet. Er soll verhindern, daß das Netzwerk mit Paketen überflutet wird.

SCTP_I_WANT_MAPPED_V4_ADDR

Diese Socketoption ist ein Flag, das anzeigt, ob gemappte IPv4-Adressen verwendet werden sollen oder nicht. Wird IPv6 über SCTP abgewickelt und die Option angeschaltet, werden IPv4-Adressen auf das IPv6-Adressen umgeschrieben. Andernfalls werden sowohl AF_INET6- und AF_INET-Adressformate empfangen. Standardmäßig ist diese Option aktiviert.

SCTP_MAXSEG

Gibt die maximale Größe einer Nachricht an, die in einem SCTP DATA Chunk enthalten sein darf. Ist eine Nachricht größer als dieser Wert, so wird sie fragmentiert. Das passiert auch unabhängig von diesem Wert, wenn beispielsweise die Path MTU kleiner als die spezifizierte Größe ist. Beträgt der Wert 0, zeigen wir an, daß wir die Fragmentierung nicht beeinflussen, sondern lediglich die Path MTU für die Ermittlung der Größe für die DATA Chunks zulassen.

SCTP_NODELAY

Schaltet den Nagle-Algorithmus ab (siehe B.6 auf Seite 462). Der Algorithmus funktioniert mit SCTP genauso wie mit TCP, allerdings versucht SCTP immer nur DATA Chunks und nicht Bytes zusammenzufassen.

SCTP_PEER_ADDR_PARAMS

Um die Parameter einer Adresse innerhalb der Assoziation zu verändern (beispielsweise den Heartbeat-Intervall), verwenden wir die Option **SCTP_PEER_ADDR_PARAMS**, die eine **sctp_paddrparams** erfordert:

```
struct sctp_paddrparams {
    sctp_assoc_t          spp_assoc_id;
    struct sockaddr_storage spp_address;
    uint32_t               spp_hbinterval;
    uint16_t               spp_pathmaxrxt;
    uint32_t               spp_pathmtu;
    uint32_t               spp_sackdelay;
    uint32_t               spp_flags;
    uint32_t               spp_ipv6_flowlabel;
    uint8_t                spp_ipv4_tos;
};
```

Die Member der Struktur haben folgende Bedeutung:

spp_assoc_id

Muß von der Applikation gesetzt werden, um die betreffende Assoziation für diese Operation zu spezifizieren.

spp_address

Gibt die betreffende Adresse für diese Operation an. Sie muß von der Anwendung gesetzt werden.

spp_hbinterval

Gibt den Heartbeat-Intervall in Millisekunden an. Zusätzlich muß auch **spp_flags** auf SPP_HB_ENABLE gesetzt werden, ansonsten wird das Feld ignoriert. Wenn es mit 0 initialisiert wurde, zeigen wir an, daß die Einstellung nicht verändert werden soll. Um nun dennoch einen echten Wert von 0 anzugeben, müssen wir zusätzlich **spp_flags** auf SPP_HB_TIME_IS_ZERO setzen.

spp_pathmaxrxt

Zeigt die maximale Anzahl von Retransmissionen an, bevor diese Adresse innerhalb der Assoziation als nicht erreichbar eingestuft wird. Dieses Feld ist nur wirksam, wenn **spp_flags** auf SPP_PMTUD_ENABLE gesetzt wird. Wenn es mit 0 initialisiert wurde, zeigen wir an, daß die Einstellung nicht verändert werden soll.

spp_pathmtu

Wenn PMTU (*path MTU discovery*) deaktiviert ist, gibt dieser Wert eine feste MTU für den Pfad dieser Adresse an. Diese MTU wird aber ignoriert, solange nicht auch SPP_PMTUD_DISABLE in **spp_flags** gesetzt ist. Beachten Sie, daß wenn **spp_address** nicht gesetzt ist, alle Adressen dieser Assoziation genau diese MTU erhalten.

spp_sackdelay

Wenn die Verzögerung selektiver Bestätigungen (*selective acknowledgements*, SACKs) aktiviert ist, **spp_flags** auf SPP_SACKDELAY_ENABLE gesetzt wurde, gibt dieses Feld die Verzögerung in Millisekunden an. Wenn es mit 0 initialisiert wurde, zeigen wir an, daß die Einstellung nicht verändert werden soll.

spp_ipv6_flowlabel

Dieses Feld wird im Zusammenhang mit dem SPP_IPV4_FLOWLABEL-Flag verwendet.

spp_ipv4_tos

Dieses Feld wird im Zusammenhang mit dem SPP_IPV4_TOS-Flag verwendet.

spp_flags

Bestimmt, wie einige Felder der Struktur interpretiert werden. Folgende Optionen sind zulässig, von denen keines oder mehrere durch logische ODER-Operationen verknüpft werden dürfen:

SPP_HB_DIS|ENABLE

Schaltet den Heartbeat für die spezifizierte Adresse ein oder aus. Beide Flags schließen sich gegenseitig aus.

SPP_HB_DEMAND

Sorgt dafür, daß der angeforderte Heartbeat-Wert sofort aktiv wird.

SPP_HB_TIME_IS_ZERO

Schaltet die Verzögerung für den Heartbeat ab.

SPP_PMTUD_DIS|ENABLE

Schaltet die PMTU für die spezifizierte Adresse ein oder aus. Ist das Feld **spp_address** leer, sind alle Adressen derselben Assoziation betroffen.

SPP_SACKDELAY_DIS|ENABLE

Schaltet die Verzögerung von selektiven Bestätigungen (SACKs) an oder aus. Der Wert des Feldes **spp_sackdelay** wird nur mittels dieses Flags aktiviert oder deaktiviert. Ist das Feld **spp_address** leer, sind alle Adressen derselben Assoziation betroffen.

SPP_IPV6_FLOWLABEL

Wenn das Flag gesetzt ist, wird das Flowlabel für die betreffende Adresse in der Assoziation gesetzt. Ist das Feld **spp_address** leer, sind alle IPv6-Adressen derselben Assoziation betroffen.

SPP_IPV4_TOS

Wenn das Flag gesetzt ist, wird der TOS-Wert in **spp_ipv4_tos** für die betreffende Adresse in der Assoziation gesetzt. Ist das Feld **spp_address** leer, sind alle IPv4-Adressen derselben Assoziation betroffen.

SCTP_PRIMARY_ADDR

Wird diese Option verwendet, so zeigt es an, daß die angegebene *lokale* Adresse die Primäradresse

der Assoziation ist. Die Primäradresse ist die Quelladresse für alle Nachrichten des lokalen Peers; es wird also keine Nachricht an die Gegenstelle abgesetzt. Zur Ausführung dieser Anfrage wird eine `sctp_setprim`-Struktur verwendet:

```
struct sctp_setprim {
    sctp_assoc_t           ssp_assoc_id;
    struct sockaddr_storage ssp_addr;
};
```

Das Feld `ssp_assoc_id` zeigt die betreffende Assoziation an und `ssp_addr` die Adresse, welche die Primäradresse des Peers sein soll. Beide Felder müssen von der Applikation gesetzt werden.

SCTP_RTOINFO

Diese Socketoption erlaubt das Tuning unterschiedlicher RTOs. Dazu wird folgende `sctp_rtoinfo`-Struktur verwendet:

```
struct sctp_rtoinfo {
    sctp_assoc_t           srto_assoc_id;
    uint32_t                srto_initial;
    uint32_t                srto_max;
    uint32_t                srto_min;
};
```

Die Felder der Struktur haben folgende Bedeutungen:

`srto_initial`
Gibt den initialen RTO-Wert an.

`srto_min|max`
Gibt die minimalen und maximalen RTO-Werte an.

`srto_assoc_id`
Wird von der Applikation gesetzt und bestimmt, welche Assoziation von diesen Einstellungen betroffen sein soll. Ist dieser Wert 0, sind alle Assoziationen betroffen.

Alle Felder erwarten Angaben in Millisekunden. Bis auf `srto_assoc_id` zeigen Werte von 0 an, daß die Einstellungen nicht verändert werden sollen.

SCTP_SET_PEER_PRIMARY_ADDR

Sorgt dafür, daß die angegebene *lokale* Adresse der Assoziation die standardmäßige Zieladresse für alle Nachrichten der Gegenseite ist. Dazu wird eine Nachricht abgesetzt, die eine `sctp_setpeerprim`-Struktur mit den notwendigen Angaben enthält:

```
struct sctp_setpeerprim {
    sctp_assoc_t           sppp_assoc_id;
    struct sockaddr_storage sppp_addr;
};
```

Die Felder `sppp_assoc_id` und `sppp_addr` geben die jeweiligen Assoziation und die zu konfigurierende Adresse an. Beide Felder müssen von der Applikation gesetzt werden. Der Standard definiert diese Funktionalität als Optional, so daß mangels Unterstützung `EOPNOTSUPP` zurückgegeben werden kann.

SCTP_STATUS

Diese Socketoption erlaubt Anwendungen den Zustand einer Assoziation abzufragen, darunter Angaben wie die Fenstergröße oder die Anzahl der unbestätigten Chunks. Diese Informationen können nur gelesen werden und sind in einer `sctp_status`-Struktur zusammengefaßt.

```
struct sctp_status {
    sctp_assoc_t           sstat_assoc_id;
    int32_t                  sstat_state;
    uint32_t                  sstat_rwnd;
```

```

    uint16_t          sstat_unackdata ;
    uint16_t          sstat_penddata ;
    uint16_t          sstat_instrms ;
    uint16_t          sstat_outstrms ;
    uint32_t          sstat_fragmentation_point;
    struct sctp_paddrinfo sstat_primary ;
};


```

Die Felder der Struktur haben folgende Bedeutungen:

sstat_state

Enthält den aktuellen Zustand der Assoziation. Er wird durch eines der folgenden Flags angezeigt:

- SCTP_CLOSED – Die Verbindung ist geschlossen.
- SCTP_BOUND – Bereit zu verbinden oder Verbindungen zu akzeptieren.
- SCTP_LISTEN – Verbunden, warte auf Verbindungsanfragen.
- SCTP_COOKIE_WAIT – Applikation hat INIT gesendet.
- SCTP_COOKIE_ECHOED – Cookie zurückgegeben.
- SCTP_ESTABLISHED – Verbindung erfolgreiche hergestellt.
- SCTP_SHUTDOWN_PENDING – Applikation führt einen Active Close aus.
- SCTP_SHUTDOWN_SENT – Applikation wartet auf SHUTDOWN-ACK.
- SCTP_SHUTDOWN_RECEIVED – Applikation führt Passive Close aus.
- SCTP_SHUTDOWN_ACK_SENT – Applikation hat SHUTDOWN-ACK gesendet.

sstat_rwnd

Enthält die Größe des Empfangsfensters des Peers.

sstat_unackdata

Zeigt die Anzahl der ausstehenden Bestätigungen für DATA Chunks an.

sstat_penddata

Zeigt die Anzahl der ausstehenden DATA Chunks an.

sstat_primary

Enthält die primäre Adresse des Peers.

sstat_primary

Enthält die ID der aktuellen Assoziation.

sstat_instrms

Anzahl der eingehenden Input Streams des Peers.

sstat_outstrms

Anzahl der erlaubten ausgehenden Input Streams des Peers.

sstat_fragmentation_point

Zeigt an, ab welcher Größe SCTP Fragmentierung anwendet.

Anhang C

Fehlerbehandlung

*Part of the inhumanity of the computer is that,
once it is competently programmed and working
smoothly, it is completely honest.*

ISAAC ASIMOV (1920 - 1992)

Die meisten robusten Programme enthalten viel Code zur Fehlerbehandlung und für das Logging. Um nicht unnötig viel Code wieder und wieder tippen zu müssen und um den Programmierprozess zu vereinfachen, habe ich einfache Fehlerbehandlungsroutine geschrieben, die ich in den Quelltexten immer wieder einsetze.

Zwar bietet die C-Bibliothek die Funktion `perror(3)`, die einen benutzerdefinierten String auf der Standardausgabe ausgibt, gefolgt von einer Fehlerkennzahl basierend auf `errno` (siehe Abschnitt 2.5.3 Fehlerkennziffern). Leider sind mit `perror` auch wichtige Nachteile verbunden. Zum einen kann `perror` nur ein Argument aufnehmen. Soll die Fehlermeldung aus mehreren Teilen bestehen, ist es notwendig zuerst einen String zusammenzusetzen, um ihn anschließend an `perror` zu übergeben:

```
char buf50;
...
sprintf(buf, "%s: Cannot open file %s.\n", argv0, argv1);
perror(cStringBuf);
```

Zugegeben, das ist zwar praktikabel, aber wenn wir diese Konstruktion öfter benötigen wird es lästig. Zum anderen ist es irgendwie störend, dass wenn `errno` den Wert 0 hat, die Meldung

```
: Error 0
```

auf der Konsole erscheint. Nicht schön und dazu noch überflüssig.

Im folgenden zeige ich eine Möglichkeit auf, die Fehlerbehandlung elegant auf ein Minimum zu beschränken. Dazu implementiere ich zwei verschiedene Funktionen: eine für nicht besonders schwerwiegende Fehler (*non-fatal errors*) und eine für kritische Fehler (*fatal errors*).

Kernbestandteil der beiden Funktionen ist die Verwendung der variablen Argumentliste (*variadic argument list*), die durch den Header `<stdarg.h>` bereitgestellt wird (Manpage `stdarg(5)`). Die Liste der Argumente wird durch den Typ `va_list` repräsentiert und mit Hilfe des Makros `va_start` initialisiert. Um auf einzelne Argumente aus der Liste zuzugreifen, steht uns das Makro `va_arg` zur Verfügung. Wenn wir mit der Bearbeitung der Argumentliste fertig sind, rufen wir `va_end` auf, um alles aufzuräumen, was angefallen ist.

```
void err_normal(const char *fmt, ...) {
    va_list args;
```

```

va_start(args, fmt);           /* initialize */
__process_error(fmt, args);   /* process this error */
va_end(args);
}

void err_fatal(const char *fmt, ...) {
    va_list args;

    va_start(args, fmt);           /* initialize */
    __process_error(fmt, args);   /* process this error */
    va_end(args);
}

```

Die eigentliche Arbeit übernimmt die Funktion `__process_error`. Die Funktionen `err_normal` und `err_fatal` sind nur Wrapper-Funktionen, die den Aufruf von `__process_error` durchführen.

Die Funktion `__process_error` könnte wie folgt implementiert werden.

```

void __process_error(const char *fmt, va_list args) {
    int error;

    /*
     * Save errno, to make sure we can restore it if one of the following
     * function calls changes it for some reason.
     */
    error = errno;

    /*
     * Print: 1. caller's error message,
     *        2. system error message if applicable,
     *        3. add newline character.
     */
    vfprintf(stderr, fmt, args);
    if (error != 0) { /* a system error has occurred */
        fprintf(stderr, ": %s", strerror(error));
        putc('\n', stderr);
    }
}

```

Praktisch bedeutet dies für uns, daß wir `err_normal` und `err_fatal` wie `printf` benutzen können. Das illustriert dieser Code-Abschnitt:

```

if ((fd = open(argv[1], O_RDONLY)) < 0) {
    err_fatal("open error for %s", argv[1]);
}

```

Auf der Konsole unseres Terminals sehen wir dann dies (oder so ähnlich):

```
% myapp /tmp/file.sh
open error for /tmp/file.sh: No such file or directory (ENOENT)
```

Damit der Compiler (besser: der Linker) die Funktionsaufrufe richtig einordnen kann, macht es Sinn, sie in einer eigenen Datei zu speichern, z.B. `errors.c`, und bei der Übersetzung eines Programms, das die Funktionen benötigt, statisch dagegen zu linken, beispielsweise so:

```
% cc -o myapp myapp.c errors.c
```

Anhang D

Beschreibung der Hilfsfunktionen

Home computers are being called upon to perform many new functions, including the consumption of homework formerly eaten by the dog.

DOUG LARSON

Im Rahmen der Arbeit an diesem Buch sind einige Funktionen entstanden, die mir die Arbeit erleichtert haben, beispielsweise bei der Fehlerbehandlung. Andere Funktionen wie etwa Wrapper von Systemaufrufen erhöhen die Lesbarkeit der Codes. In diesem Teil des Anhangs finden Sie die Quellen der verwendeten Funktionen.

D.1 Der Header <header.h>

In fast allen Beispielen importieren wir den Header <header.h>, in dem wichtige Konstanten, Prototypen und Typdefinitionen enthalten sind.

Listing D.1: Der Header <header.h>

```
1 #ifndef __HEADER_H
2 #define __HEADER_H
3
4 #define _XOPEN_SOURCE 500
5
6 #ifndef _XPG4_2 /* for Solaris */
7 # define _XPG4_2
8 #endif
9
10#define DEBUG 1
11
12#include <sys/stat.h>
13#include <sys/types.h>
14#include <sys/socket.h>
15#include <sys/select.h>
16#include <sys/un.h>
17
18#ifndef SYSV
19    #include <sys/wait.h>
20#else
21    #include <wait.h>
22#endif
```

```

23
24 #ifdef X_POSIX_C_SOURCE
25     #define _POSIX_C_SOURCE X_POSIX_C_SOURCE
26     #include <signal.h>
27     #undef _POSIX_C_SOURCE
28 #else
29     #if defined(X_NOT_POSIX) || defined(_POSIX_SOURCE)
30         #include <signal.h>
31     #else
32         #define _POSIX_SOURCE
33         #include <signal.h>
34         #undef _POSIX_SOURCE
35     #endif
36 #endif
37
38 #if defined(__FreeBSD__)
39 # include <sys/param.h> /* for FreeBSD */
40 #endif
41
42 #include <arpa/inet.h>
43 #include <netinet/in.h>
44 #include <netdb.h>
45
46 #include <fcntl.h>
47 #include <stdio.h>
48 #include <string.h>
49
50 #include <stdlib.h>
51 #include <errno.h>
52 #include <stdarg.h>
53 #include <limits.h>
54 #include <unistd.h>
55
56 #include <semaphore.h>
57 #include <pthread.h>
58
59
60
61
62 #if defined LINE_MAX
63     #define LINESIZ LINE_MAX
64 #elif defined _SC_LINE_MAX
65     #define LINESIZ      (sysconf(_SC_LINE_MAX));
66 #else
67     #define LINESIZ      512
68 #endif
69
70 #define MAX_BUF          512
71 #define MAX_DGRAM_SIZ    8192 /* fallback */
72
73 #define TRUE           1
74 #define FALSE          0
75
76 #define MIN(a,b)        ((a) < (b) ? (a) : (b))
77 #define MAX(a,b)        ((a) > (b) ? (a) : (b))
78
79 #define MODE_FLAGS      (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
80 #define PIPE_MODE       (S_IRUSR | S_IWUSR | S_IRGRP | \
81                         S_IWGRP | S_IROTH | S_IWOTH)
82
83 #define SIN_PTR         struct sockaddr_in *
84 #define SIN6_PTR        struct sockaddr_in6 *

```

```

85 #define SA_PTR      struct sockaddr *
86
87 #ifndef SUN_LEN
88     #define SUN_LEN(ptr) (((struct sockaddr_un *) 0)->sun_path) \
89             + strlen ((ptr)->sun_path))
90 #endif
91
92 /* used with waitpid's first argument */
93 #define WANYCHILD -1
94 #define WGROUCHILD 0
95
96 /* dummy buffer for passing n FDs via recvmsg/sendmsg */
97 #define FD_BUFFER(n) \
98     struct { \
99         struct cmsghdr h; \
100        int     fds[n]; \
101    }
102
103 typedef int64_t longlong_t;
104 typedef uint64_t u_longlong_t;
105 typedef uint32_t ulong_t;
106
107 typedef void sigfunc_t(int);
108
109 /*
110  * Threading and related stuff
111  */
112 #define NULL_TID (pthread_t) 0L
113
114 void *thread_func1(void *arg);
115 void *thread_func2(void *arg);
116 void *thread_func3(void *arg);
117 void cleanup(void *arg);
118
119 void pthread_create_ex(pthread_t *thread, const pthread_attr_t *attr,
120                         void *(*start_routine)(void*), void *arg);
121 char *thread_name(pthread_t tid);
122
123 typedef struct thread {
124     pthread_t      tid;
125     char          *name;
126     struct thread *next;
127 } thread_name_t;
128
129 /*
130  * Prototypes
131  */
132 void           err_fatal(const char *, ...);
133 void           err_normal(const char *, ...);
134
135 /* libsock.c */
136 int            accept_ex(int socket, SA_PTR address, socklen_t *len);
137 void           bind_ex(int socket, const struct sockaddr *address, socklen_t len);
138 void           connect_ex(int socket, const SA_PTR address, socklen_t len);
139 struct hostent *gethost(char *host);
140 void           getpeername_ex(int fd, SA_PTR addr, socklen_t *addrlen);
141 void           getsockname_ex(int fd, SA_PTR addr, socklen_t *addrlen);
142 void           getsockopt_ex(int fd, int level, int name, void *optval, socklen_t *optle);
143 void           inet_pton_ex(int family, const char *address, void *dst);
144 char           *inet_ntop_ex(const SA_PTR saddr, socklen_t len);
145 void           listen_ex(int socket, int backlog);
146 int            make_socket(uint16_t port);

```

```

147 ssize_t           recv_ex(int fd, void *ptr, size_t nbytes, int flags);
148 ssize_t           recvfrom_ex(int fd, void *buf, size_t bufsiz, int flags,
149                                SA_PTR from, socklen_t *len);
150 ssize_t           recvmsg_ex(int fd, struct msghdr *msg, int flags);
151 int               select_ex(int nfds, fd_set *rfds, fd_set *wfds, fd_set *efds,
152                             struct timeval *t);
153 void              send_ex(int fd, const void *ptr, size_t nbytes, int flags);
154 void              sendmsg_ex(int fd, const struct msghdr *msg, int flags);
155 void              sendto_ex(int fd, const void *buf, size_t bufsiz, int flags,
156                            SA_PTR to, socklen_t len);
157 void              setsockopt_ex(int, int, int, const void *, socklen_t);
158 void              shutdown_ex(int socketfd, int how);
159 int               socket_ex(int domain, int type, int protocol);
160 void              socketpair_ex(int *sockets);

161 /* libsys.c */
162 void              close_ex(int fd);
163 int               create_argv(char *, char *, char **);
164 void              echo_line(int fd);
165 char              *fgets_ex(char *buf, int size, FILE *stream);
166 pid_t             fork_ex(void);
167 void              fputs_ex(const char *s, FILE *stream);
168 int               get_block_size(int *size, char *path);
169 char              *getcwd_ex(void);
170 void              *malloc_ex(size_t size);
171 void              open_ex(char *path, int oflag, mode_t mode);
172 int               open_paranoid(char *path, int oflag, mode_t mode);
173 ssize_t            read_ex(int fd, char *buf, size_t bufsiz);
174 void              sem_init_ex(sem_t *sem, int pshared, unsigned int value);
175 void              sem_wait_ex(sem_t *sem);
176 void              sem_post_ex(sem_t *sem);
177 void              sem_getvalue_ex(sem_t *sem, int *sval);
178 sigfunc_t          *signal_ex(int signum, sigfunc_t *f);
179 void              waitpid_ex(pid_t pid, int *status, int options);
180 void              write_ex(int fd, void *buf, size_t size);

181 /* lltostr.c */
182 char              *lltostr(longlong_t value, char *ptr);
183 char              *ulltostr(u_longlong_t value, char *ptr);

184 /* make_tcp_socket.c */
185 int               make_tcp_socket(int family, uint16_t port, SA_PTR addr);

186 /* print_exit_status.c */
187 void              print_exit_status(int status);

188 /* readbytes_ex.c */
189 ssize_t            readbytes_ex(int fd, void *buf, size_t size);

190 /* readline.c */
191 ssize_t            readline(int fd, char *buf, size_t size);

192 /* set_sinaddr.c */
193 void              set_sinaddr(sa_family_t family,
194                               const char *strptr, void *addrptr);

195 /* share_fd.c */
196 int               send_fd(int socket, int fd);
197 int               recv_fd(int socket);

198 /* utils.c */
199 char              *basename_ex(char *);

200
201
202
203
204
205
206
207
208

```

```

209  /* writen.c */
210  ssize_t      writen(int fd, const void *buf, size_t size);
211
212
213 #endif

```

Listing D.1: xcode/lib/header.h - Der Header <header.h>.

D.2 Die Wrapper-Funktionen in libsys.c und libsock.c

Während der Ausarbeitung des Buches habe ich eine Technik angewandt, die mir das erste mal beim Studium von [Stevens91] begegnete: Wrapper-Funktionen für Systemaufrufe. Der Vorteil liegt darin, daß wir uns besser auf bestimmte Teile des Quelltextes konzentrieren können, weil unsere Aufmerksamkeit nicht ständig von der Fehlerbehandlung der Systemaufrufe abgelenkt wird. Die gesamte Fehlerbehandlung wird im Wrapper erledigt, so daß wir uns die Abfrage der Rückgabewerte sparen können und der Code dadurch viel übersichtlicher wird.

D.2.1 Wrapper für UNIX-Systemaufrufe

Die Datei lib/libsys.c enthält alle verwendeten Wrapper-Funktionen für Standardsystemaufrufe unter Unix. Die Socket-Funktionen sind im nächsten Listing aufgeführt.

Listing D.2: Die Bibliothek lib/libsys.c

```

1  #include "header.h"
2
3  void close_ex(int fd) {
4      if (close(fd) < 0)
5          err_fatal("close() failed");
6  }
7
8
9  char *fgets_ex(char *buf, int size, FILE *stream) {
10     char *buffer;
11
12     if ((buffer = fgets(buf, size, stream)) == NULL) {
13 #ifdef DEBUG
14         printf("fgets_ex(): got NUL character");
15 #endif
16     if (ferror(stream))
17         err_fatal("fgets() failed");
18 }
19
20     return (buffer);
21 }
22
23
24 pid_t fork_ex(void) {
25     pid_t pid;
26
27     if ((pid = fork()) == -1)
28         err_fatal("fork() failed");
29
30     return (pid);
31 }
32
33

```

```

34 void fputs_ex(const char *s, FILE *stream) {
35     if (fputs(s, stream) == EOF)
36         err_fatal("fputs_ex() failed.");
37 }
38
39
40 char *getcwd_ex(void) {
41     char *path_buf = NULL, *ptr = NULL;
42
43     if ((path_buf = (char *)malloc((size_t)PATH_MAX)) != NULL) {
44         if ((ptr = getcwd(path_buf, (size_t)PATH_MAX)) == NULL) {
45             err_fatal("getcwd failed in getcwd_ex");
46         }
47     } else {
48         err_fatal("malloc failed in getcwd_ex");
49     }
50
51     return ptr;
52 }
53
54
55 void *malloc_ex(size_t size) {
56     void *ptr;
57
58     if ((ptr = malloc(size)) == NULL)
59         err_fatal("malloc() failed");
60
61     return (ptr);
62 }
63
64
65 int open_ex(char *path, int oflag, mode_t mode) {
66     int fd;
67
68     if ((fd = open(path, oflag, mode)) < 0)
69         err_fatal("open() failed");
70
71     return fd;
72 }
73
74
75 int open_paranoid(char *path, int oflag, mode_t mode) {
76     char *msg = "Will not proceed";
77     char *func = "open_paranoid()";
78     int fd;
79     struct stat buf1, buf2;
80
81     /* step 1: lstat the path, check that lstat succeeds */
82     if (lstat(path, &buf1) < 0)
83         err_fatal("%s: lstat() failed", func);
84
85     /* step 2: is path a symlink? */
86     if (S_ISLNK(buf1.st_mode))
87         err_fatal("%s: path is a symbolic link. %s.\n", func, msg);
88
89     /* step 3: try to open path for reading, omitting O_RDONLY */
90     if ((fd = open(path, oflag, O_RDONLY)) < 0)
91         err_fatal("%s: open() failed for O_RDONLY", func);
92
93     /* step 4: fstat the fd returned by open */
94     if (fstat(fd, &buf2) < 0)
95         err_fatal("%s: fstat() failed", func);

```

```

96     /* step 5: compare st_ino and st_dev fields if they match */
97     if (buf1.st_ino != buf2.st_ino || buf1.st_dev != buf2.st_dev)
98         err_fatal("%s: inode and device do not match. %s", func, msg);
99     else
100         if ((fd = open(path, oflag, mode)) < 0)
101             err_fatal("%s: open() failed", func);
102
103     return fd;
104 }
105
106
107
108 ssize_t read_ex(int fd, char *buf, size_t bufsiz) {
109     int bytes_read;
110
111     eintr:
112     if ((bytes_read = read(fd, buf, bufsiz)) < 0) {
113         if (errno == EINTR) {
114 #ifdef DEBUG
115             printf("read_ex(): interrupted...\n");
116 #endif
117             goto eintr;
118         }
119
120         return (-1);
121     }
122 #ifdef DEBUG
123     printf("read_ex(): read %d bytes\n", bytes_read);
124 #endif
125     return (bytes_read);
126 }
127
128
129 void sem_init_ex(sem_t *sem, int pshared, unsigned int value) {
130     int ret;
131
132 #if defined(SEM_DEBUG)
133     printf("calling sem_init(): thread [%d]\n", pthread_self());
134 #endif
135     if ((ret = sem_init(sem, pshared, value)) < 0)
136         err_fatal("sem_init() failed in thread [%d]", pthread_self());
137 }
138
139
140 void sem_wait_ex(sem_t *sem) {
141     int ret;
142
143 #if defined(SEM_DEBUG)
144     printf("calling sem_wait(): thread [%d]\n", pthread_self());
145 #endif
146     if ((ret = sem_wait(sem)) < 0)
147         err_fatal("sem_wait() failed in thread [%d]", pthread_self());
148 }
149
150
151 void sem_post_ex(sem_t *sem) {
152     int ret;
153
154 #if defined(SEM_DEBUG)
155     printf("calling sem_post(): thread [%d]\n", pthread_self());
156 #endif
157     if ((ret = sem_post(sem)) < 0)

```

```

158         err_fatal("sem_post() failed in thread [%d]", pthread_self());
159     }
160
161
162 void sem_getvalue_ex(sem_t *sem, int *sval) {
163     int ret;
164
165 #if defined(SEM_DEBUG)
166     printf("calling sem_getvalue(): thread [%d]\n", pthread_self());
167 #endif
168     if ((ret = sem_getvalue(sem, sval)) < 0)
169         err_fatal("sem_getvalue() failed in thread [%d]", pthread_self());
170 }
171
172 /*
173  * An implementation by W. R. Stevens
174 */
175 sigfunc_t *signal_ex(int signum, sigfunc_t *f) {
176     struct sigaction act, oact;
177
178     act.sa_handler = f;
179     sigemptyset(&act.sa_mask);
180     act.sa_flags = 0;
181
182     if (signum == SIGALRM) {
183 #ifdef SA_INTERRUPT
184         act.sa_flags |= SA_INTERRUPT; /* SunOS 4.x */
185 #endif
186     } else {
187 #ifdef SA_RESTART
188         act.sa_flags |= SA_RESTART; /* SVR4, 4.4BSD */
189 #endif
190     }
191
192     if (sigaction(signum, &act, &oact) < 0)
193         err_fatal("signal_ex(): sigaction() failed");
194
195     return (oact.sa_handler);
196 }
197
198
199 void waitpid_ex(pid_t pid, int *status, int options) {
200     if (waitpid(pid, status, options) < 0)
201         err_fatal("waitpid() failed");
202 }
203
204
205 void write1_ex(int fd, void *buf, size_t size) {
206     char    *bufptr = buf;
207     size_t  bytestowrite = size;
208     ssize_t byteswritten;
209
210     while (bytestowrite > 0) {
211         byteswritten = write(fd, bufptr, bytestowrite);
212
213         bufptr += byteswritten;
214         bytestowrite -= byteswritten;
215
216         if (errno == EPIPE)
217             err_fatal("write() error: got EPIPE");
218
219         if (byteswritten == -1 && (errno != EINTR))

```

```

220             err_fatal("write() failed.");
221
222         if (byteswritten == -1)
223             byteswritten = 0;
224     }
225 }
226
227
228 void write_ex(int fd, void *buf, size_t size) {
229     if (write(fd, buf, size) != size)
230         err_fatal("write() failed");
231 }
```

Listing D.2: xcode/lib/libsys.c - Die Bibliothek lib/libsys.c.

D.2.2 Wrapper für Socketfunktionen

Die Datei lib/libsock.c enthält alle verwendeten Wrapper-Funktionen für Socketfunktionen der BSD Socket API.

Listing D.3: Die Bibliothek lib/libsock.c

```

1 #include "header.h"
2
3 int accept1_ex(int socket, SA_PTR address, socklen_t *len) {
4     int fd;
5
6     if ((fd = accept(socket, address, len)) == -1)
7         err_fatal("accept() failed");
8
9     return (fd);
10 }
11
12 int accept_ex(int socket, SA_PTR address, socklen_t *len) {
13     int fd;
14
15     label:
16         while (((fd = accept(socket, address, len)) == -1) &&
17             (errno != EINTR)) ;
18         /* accept has not been EINTR'd; something else went wrong */
19 #ifdef EPROTO
20         if (errno == EPROTO || errno == ECONNABORTED) {
21 #else
22         if (errno == ECONNABORTED) {
23 #endif
24             goto label;
25         } else if (fd == -1) {
26             err_fatal("accept() failed");
27         }
28
29     return (fd);
30 }
31
32 void bind_ex(int socket, const struct sockaddr *address, socklen_t len) {
33     if (bind(socket, address, len) < 0) {
34         err_normal("bind() failed for socket [%d]", socket);
35         close_ex(socket); /* exit gracefully */
36         exit(1);
37     }
}
```

```

38 }
39
40 void connect_ex(int socket, const SA_PTR address, socklen_t len) {
41     if (connect(socket, address, len) < 0)
42         err_fatal("connect() failed");
43 }
44
45 void getpeername_ex(int fd, SA_PTR addr, socklen_t *addrlen) {
46     if (getpeername(fd, addr, addrlen) < 0)
47         err_fatal("getpeername() failed");
48 }
49
50 void getsockname_ex(int fd, SA_PTR addr, socklen_t *addrlen) {
51     if (getsockname(fd, addr, addrlen) < 0)
52         err_fatal("getsockname() failed");
53 }
54
55 void getsockopt_ex(int fd, int level, int name, void *optval, socklen_t *optlen) {
56     if (getsockopt(fd, level, name, optval, optlen) < 0)
57         err_fatal("getsockopt() failed");
58 }
59
60 /*
61  * inet_ntop_ex -- Wrapper for inet_ntop.
62  *
63  * Converts a binary IPv4 or IPv6 address into a human readable
64  * format. addr is a generic sockaddr structure which is to
65  * be converted and len specifies the size of the particular
66  * structure .
67  *
68  * Returns a buffer containing the human readable presentation
69  * of the specified address.
70 */
71 char *inet_ntop_ex(const SA_PTR saddr, socklen_t len) {
72     char port[7];
73     static char address[128];
74
75     switch (saddr->sa_family) {
76     case AF_INET: {
77         struct sockaddr_in *sin = (struct sockaddr_in *)saddr;
78
79         if (inet_ntop(AF_INET, &sin->sin_addr, address, sizeof(address)) == NULL)
80             return(NULL);
81
82         if (ntohs(sin->sin_port) != 0) {
83             snprintf(port, sizeof(port), ".%d", ntohs(sin->sin_port));
84             strcat(address, port);
85         }
86
87         return (address);
88     }
89
90     case AF_INET6: {
91         struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *)saddr;
92
93         if (inet_ntop(AF_INET6, &sin6->sin6_addr, address,
94             sizeof(address)) == NULL)
95             return(NULL);
96
97         if (ntohs(sin6->sin6_port) != 0) {
98             snprintf(port, sizeof(port), ".%d", ntohs(sin6->sin6_port));
99             strcat(address, port);

```

```

100         }
101
102         return (address);
103     }
104 #ifdef __UN_H
105     case AF_UNIX: {
106         struct sockaddr_un *unp = (struct sockaddr_un *)saddr;
107
108         if (unp->sun_path[0] == 0)
109             strcpy(address, "(not bound)");
110         else
111             snprintf(address, sizeof(address), "%s", unp->sun_path);
112
113         return (address);
114     }
115 #endif
116     default: {
117         snprintf(address, sizeof(address),
118                 "sock_ntop: unknown AF_XXX: %d, len %d",
119                 saddr->sa_family, len);
120
121         return (address);
122     }
123 }
124
125     return (NULL);
126 }
127
128 void inet_pton_ex(int family, const char *address, void *dst) {
129     if (!inet_pton(family, address, dst))
130         err_fatal("inet_pton() failed");
131 }
132
133 void listen1_ex(int socket, int backlog) {
134     if (listen(socket, backlog) < 0)
135         err_fatal("listen() failed for socket [%d] \
136                 with backlog [%d].\n", socket, backlog);
137 }
138
139 void listen_ex(int socket, int backlog) {
140     int bl = 5;
141 #if defined SOMAXCONN
142     bl = SOMAXCONN;
143 #endif
144     if (backlog == -1) {
145         if (listen(socket, bl) < 0) {
146             err_fatal("listen() failed for socket [%d] \
147                     with backlog [%d].\n", socket, backlog);
148         }
149     } else {
150         if (listen(socket, backlog) < 0) {
151             err_fatal("listen() failed for socket [%d] \
152                     with backlog [%d].\n", socket, backlog);
153         }
154     }
155 }
156
157 ssize_t recv_ex(int fd, void *ptr, size_t nbytes, int flags) {
158     ssize_t bytes_read;
159
160     if ((bytes_read = recv(fd, ptr, nbytes, flags)) < 0)
161         err_fatal("recv() failed");

```

```

162     return (bytes_read);
163 }
165
166 ssize_t recvfrom_ex(int fd, void *buf, size_t bufsiz, int flags,
167                      SA_PTR from, socklen_t *len) {
168     ssize_t bytes_read;
169
170     if ((bytes_read = recvfrom(fd, buf, bufsiz, flags, from, len)) < 0)
171         err_fatal("recvfrom() failed");
172
173     return (bytes_read);
174 }
175
176 ssize_t recvmsg_ex(int fd, struct msghdr *msg, int flags) {
177     ssize_t bytes_read;
178
179     if ((bytes_read = recvmsg(fd, msg, flags)) < 0)
180         err_fatal("recvmsg error");
181
182     return (bytes_read);
183 }
184
185
186 /**
187  * select_ex -- Wrapper for select system call.
188  *
189  *      Returns 0 on timeout and -1 on error.
190  */
191 int select_ex(int nfds, fd_set *rfds, fd_set *wfds, fd_set *efds,
192               struct timeval *t) {
193     int rc;
194
195     if ((rc = select(nfds, rfds, wfds, efds, t)) < 0)
196         err_fatal("select() failed");
197
198     return (rc);
199 }
200
201 void send_ex(int fd, const void *ptr, size_t nbytes, int flags) {
202     if (send(fd, ptr, nbytes, flags) != nbytes)
203         err_fatal("send() failed");
204 }
205
206 void sendmsg_ex(int fd, const struct msghdr *msg, int flags) {
207     int i;
208     ssize_t bytes = 0;
209
210     for (i = 0; i < msg->msg_iovlen; i++)
211         bytes += msg->msg_iov[i].iov_len;
212
213     if (sendmsg(fd, msg, flags) != bytes)
214         err_fatal("sendmsg() failed");
215 }
216
217 void sendto_ex(int fd, const void *buf, size_t bufsiz, int flags,
218                 SA_PTR to, socklen_t len) {
219     if (sendto(fd, buf, bufsiz, flags, to, len) != bufsiz)
220         err_fatal("sendto() failed");
221 }
222
223 void setsockopt_ex(int fd, int level, int optname,

```

```

224     const void *optval, socklen_t optlen) {
225     if (setsockopt(fd, level, optname, optval, optlen) < 0)
226         err_fatal("setsockopt() failed");
227 }
228
229 void shutdown_ex(int socketfd, int how) {
230     if (shutdown(socketfd, how) < 0)
231         err_fatal("shutdown() failed");
232 }
233
234 int socket_ex(int domain, int type, int protocol) {
235     int socket_fd;
236
237     if ((socket_fd = socket(domain, type, protocol)) < 0)
238         err_fatal("socket() failed");
239
240     return (socket_fd);
241 }
242
243 void socketpair_ex(int *sockets) {
244     if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0)
245         err_fatal("socketpair() failed");
246 }
```

Listing D.3: xcode/lib/libsock.c - Die Bibliothek lib/libsock.c.

D.2.3 Wrapper für pthread-Funktionen

Die Datei lib/libthread.c enthält alle verwendeten Wrapper-Funktionen POSIX Threads API.

Listing D.4: Die Bibliothek lib/libthread.c

```

1 #include "header.h"
2
3 void pthread_create_ex(pthread_t *thread, const pthread_attr_t *attr,
4                         void *(*start_routine)(void*), void *arg) {
5     if (pthread_create(thread, attr, start_routine, &arg) != 0 )
6         err_fatal("pthread_create() failed");
7 }
8
9 char *pthread_name(pthread_t tid) {
10    char s[100];
11    thread_name_t *n;
12    static int tid_count = 1000;
13    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
14    static thread_name_t *thread_names = NULL;
15
16    if (pthread_equal(NULL_TID, tid)) tid = pthread_self();
17    pthread_mutex_lock(&lock);
18
19    for (n = thread_names; n != NULL; n = n->next) {
20        if (pthread_equal(n->tid, tid))
21            goto L;
22    }
23
24    n = (thread_name_t *) malloc(sizeof(thread_name_t));
25    n->tid = tid;
26    sprintf(s, "T@%d", tid_count);
27    tid_count++;
```

```

28     n->name = (char *)malloc(strlen(s) + 1);
29     strcpy(n->name, s);
30     n->next = thread_names;
31     thread_names = n;
32 L:
33     pthread_mutex_unlock(&lock);
34     return (n->name);
35 }
```

Listing D.4: xcode/lib/libthread.c - Die Bibliothek lib/libthread.c.

D.3 Verschiedene Source Codes

Im Laufe des Textes haben wir einige Funktionen definiert, die uns die Arbeit erleichtern sollten, beispielsweise `get_block_size` oder `create_argv`. In diesem Abschnitt finden Sie die Quellcodes, zu denen ich hin und wieder ein paar Worte verliere. Die Reihenfolge der Liste ist alphabetisch.

Listing D.5: Die Funktion `create_argv`

```

1 #include "header.h"
2
3 int create_argv(char *s, char *delims, char **argvp[]) {
4     char *temp, *command_string;
5     int num_tokens, i = 1;
6
7     /* get start of string; strip off leading delims */
8     command_string = s + strspn(s, delims);
9
10    /* allocate storage for copying command_string to t */
11    if ((temp = calloc(strlen(command_string) + 1, sizeof(char))) == NULL) {
12        *argvp = NULL;
13        num_tokens = -1; /* calloc() failed */
14    } else {
15        strcpy(temp, command_string);
16
17        if (strtok(temp, delims) == NULL) {
18            num_tokens = 0;
19        } else {
20            num_tokens = 1;
21
22            while (strtok(NULL, delims) != NULL)
23                num_tokens++;
24        }
25
26        /* create an argument array to contain pointers to tokens */
27        if ((*argvp = calloc(num_tokens + 1, sizeof(char *))) == NULL) {
28            free(temp);
29            num_tokens = -1; /* calloc() failed */
30        } else { /* place pointers to tokens into slots of argvp */
31            if (num_tokens > 0) {
32                strcpy(temp, command_string);
33                **argvp = strtok(temp, delims);
34
35                for (i = 1; i < num_tokens; i++)
36                    *(argvp[0] + i) = strtok(NULL, delims);
37            } else {
38                **argvp = NULL;
39                free(temp);

```

```

40         }
41     }
42 }
43
44     return (num_tokens);
45 }
```

Listing D.5: xcode/lib/create_argv.c - Die Funktion lib/create_argv.c.

Listing D.6: Die Funktion get_block_size

```

1 #include "header.h"
2
3 /*
4  *     get_block_size(int *size, char *path)
5  *
6  *         Returns the blocksize of the filesystem on which path
7  *         is located. If path is a NULL pointer the current
8  *         working directory is used by default.
9 */
10 int get_block_size(int *size, char *path) {
11     int error = -1;
12     char *path_buf;
13     struct stat stat_buf;
14
15     if (size == NULL)
16         return error;
17
18     if (stat((path_buf == NULL ? getcwd_ex() : path), &stat_buf) < 0) {
19         error = errno;
20         return (error);
21     } else {
22         *size = stat_buf.st_blksize;
23         return (0);
24     }
25 }
```

Listing D.6: xcode/lib/get_block_size.c - Die Funktion lib/get_block_size.c.

Listing D.7: Die Funktionen lltosstr und ulltosstr

Viele Systeme unterstützen die beiden Funktionen `lltosstr` und `ulltosstr` nicht, so daß wir uns einer eigenen Implementierung bemühen müssen.

```

1 #include "header.h"
2
3 char *lltosstr(longlong_t value, char *ptr) {
4     longlong_t t;
5
6 #ifdef _ILP32
7     if (!(0xffffffff0000000ULL & value)) {
8         ulong_t t, val = (ulong_t)value;
9
10        do {
11            ---ptr = (char)('0' + val - 10 * (t = val / 10));
12        } while ((val = t) != 0);
13
14        return (ptr);
15    }
```

```

16  #endif
17
18  do {
19      --ptr = (char)('0' + value - 10 * (t = value / 10));
20  } while ((value = t) != 0);
21
22  return (ptr);
23 }
24
25 char *ulltostr(u_longlong_t value, char *ptr) {
26     u_longlong_t t;
27
28 #ifdef _ILP32
29     if (!(0xffffffff0000000ULL & value)) {
30         ulong_t t, val = (ulong_t)value;
31
32         do {
33             --ptr = (char)('0' + val - 10 * (t = val / 10));
34         } while ((val = t) != 0);
35
36         return (ptr);
37     }
38 #endif
39
40     do {
41         --ptr = (char)('0' + value - 10 * (t = value / 10));
42     } while ((value = t) != 0);
43
44     return (ptr);
45 }
```

Listing D.7: xcode/lib/lltostr.c - Die Funktionen lltostr und ulltostr.

Listing D.8: Die Funktion make_tcp_socket

Immer wieder müssen wir TCP-Sockets für unsere Server-Anwendungen erstellen. Im Wesentlichen beinhaltet es immer die gleichen Schritte, so daß wir diese Aufgabe eigentlich auch mittels einer Funktion lösen können.

```

1 #include "header.h"
2
3 /*
4  * make_tcp_socket -- convenience function to create TCP sockets.
5  *
6  * If family is not 0 its value will be passed to the call
7  * to socket, otherwise AF_INET will be used. If addr
8  * is not a NULL pointer the caller can pass this function
9  * a pointer to a variable where the address structure of the
10 * connected socket can be found. This is usefull if you
11 * are creating a server socket and need to call bind.
12 *
13 * Returns the socket descriptor of the connected socket.
14 */
15 int make_tcp_socket(int family, uint16_t port, SA_PTR addr) {
16     int opt = 1;
17     int af;
18     int socket;
19     struct sockaddr_in name;
20
21     if (family == -1)
22         af = AF_INET; /* fallback */
```

```

23     socket = socket_ex(AF_INET, SOCK_STREAM, 0);
24     setsockopt_ex(socket, SOL_SOCKET, SO_REUSEADDR,
25                     (char *)&opt, sizeof(opt));
26
27     memset((char *)&name, 0, sizeof(struct sockaddr_in));
28     name.sin_family = (sa_family_t)family;
29     name.sin_port = port;
30
31 #ifdef DEBUG
32     printf("Port %d [%d]\n", port, name.sin_port);
33 #endif
34     name.sin_addr.s_addr = htonl(INADDR_ANY);
35
36     bind_ex(socket, (SA_PTR)&name, sizeof(name));
37     listen_ex(socket, 5);
38
39 #ifdef DEBUG
40     printf("Socket listening at port %d\n", (int)name.sin_port);
41 #endif
42
43     if (addr != NULL)
44         addr = (SA_PTR)&name;
45
46     return socket;
47 }

```

Listing D.8: xcode/lib/make_tcp_socket.c - Die Funktion lib/make_tcp_socket.c.

Listing D.9: Die Funktion print_exit_status

Wenn wir mit `waitpid(2)` auf Child-Prozesse warten kann es für das Debugging sehr hilfreich sein, den Termination-Status mittels einiger Standardmakros abzufragen. Die Funktion `print_exit_status` erledigt das für uns.

```

1 #include "header.h"
2
3 void print_exit_status(int status) {
4     if (WIFEXITED(status))
5         printf("normal termination, exit status = %d\n",
6                WEXITSTATUS(status));
7     else if (WIFSIGNALED(status))
8         printf("abnormal termination, signal number = %d%s\n",
9                WTERMSIG(status),
10 #ifdef WCOREDUMP
11         WCOREDUMP(status) ? "(core file generated)" : "");
12 #else
13         "";
14 #endif
15     else if (WIFSTOPPED(status))
16         printf("child stopped, signal number = %d\n",
17                WSTOPSIG(status));
18 }

```

Listing D.9: xcode/lib/print_exit_status.c - Die Funktion lib/print_exit_status.c.

Listing D.10: Die Funktion readline

Die Funktion `readline` liest eine ganze Zeile in den spezifizierten Buffer ein. Sie kehrt zurück, wenn entweder der Buffer voll ist und noch Bytes zu lesen sind, der Buffer nicht voll ist, aber alle Bytes gelesen wurden, oder wenn ein Fehler aufgetreten ist.

```

1 #include "header.h"
2
3 ssize_t readline(int fd, char *buf, size_t size) {
4     size_t bytestoread = size, bytesread = 0;
5
6     while (bytestoread > 0) {
7         bytesread = read(fd, buf, bytestoread);
8
9         if (bytesread == -1 && errno == EINTR)
10             continue;
11
12         if (bytesread == 0 && errno != EINTR)
13             return (0);
14
15         if (bytesread == -1 && errno != EINTR)
16             return (-1);
17
18         bytestoread -= bytesread;
19
20         if (buf[bytesread - 1] == '\n') {
21             buf[bytesread] = '\0';
22             return (bytesread);
23         }
24     }
25
26     return -1;
27 }
```

Listing D.10: `xcode/lib/readline.c` - Die Funktion `lib/readline.c`.

Listing D.11: Die Funktionen `recv_fd` und `send_fd`

Das Senden und Empfangen von File Descriptors ist mit viel schwarzer Magie rund um die Ancillary Data Buffers (siehe Abschnitt 14.1.3.1) verbunden. Die beiden Funktionen `recv_fd` und `send_fd` erleichtern uns die Arbeit erheblich.

```

1 #include "header.h"
2
3 int send_fd(int socket, int fd) {
4 #if defined(HAVE_SENDMSG) &&
5     (defined(HAVE_ACCRIGHTS_IN_MSGHDR) || \
6     defined(HAVE_CONTROL_IN_MSGHDR))
7     struct msghdr msg;
8     struct iovec vec;
9     char ch = '\0';
10    int n;
11 #ifndef HAVE_ACCRIGHTS_IN_MSGHDR
12     char tmp[CMSG_SPACE(sizeof(int))];
13     struct cmsghdr *cmsg;
14 #endif
15
16     memset(&msg, 0, sizeof(msg));
17 #ifdef HAVE_ACCRIGHTS_IN_MSGHDR
18     msg.msg_accrights = (caddr_t)&fd;
19     msg.msg_accrightslen = sizeof(fd);
20 #else
21     msg.msg_control = (caddr_t)tmp;
22     msg.msg_controllen = CMSG_LEN(sizeof(int));
23
24     /*
```

```

25     * Stevens does 'nt pre-initialize cmsg, but RH 5.2,
26     * probably among others, acts flaky without it.
27     */
28     cmsg = CMSG_FIRSTHDR (&msg);
29     cmsg->cmsg_len = CMSG_LEN (sizeof (int));
30     cmsg->cmsg_level = SOL_SOCKET;
31     cmsg->cmsg_type = SCM_RIGHTS;
32     *(int *)CMSG_DATA (cmsg) = fd;
33 #endif
34
35     vec.iov_base = &ch;
36     vec.iov_len = 1;
37     msg.msg iov = &vec;
38     msg.msg iovlen = 1;
39
40     if ((n = sendmsg (socket, &msg, 0)) == -1)
41         fprintf (stderr, "send_fd(): sendmsg(%d): %s", fd,
42                  strerror (errno));
43     if (n != 1)
44         fprintf (stderr, "send_fd(): sendmsg: expected sent 1 got %d", n);
45
46     return 0;
47 #else
48     return -1;
49 #endif
50 }
51
52 int recv_fd (int socket) {
53 #if defined (HAVE_RECVMSG) &&
54     (defined (HAVE_ACCRIGHTS_IN_MSGHDR) || \
55      defined (HAVE_CONTROL_IN_MSGHDR))
56     struct msghdr msg;
57     struct iovec vec;
58     char ch;
59     int fd, n;
60 #ifndef HAVE_ACCRIGHTS_IN_MSGHDR
61     char tmp [CMSG_SPACE (sizeof (int))];
62     struct cmsghdr *cmsg;
63 #endif
64
65     memset (&msg, 0, sizeof (msg));
66     vec.iov_base = &ch;
67     vec.iov_len = 1;
68     msg.msg iov = &vec;
69     msg.msg iovlen = 1;
70 #ifdef HAVE_ACCRIGHTS_IN_MSGHDR
71     msg.msg_accrights = (caddr_t) &fd;
72     msg.msg_accrightslen = sizeof (fd);
73 #else
74     msg.msg_control = tmp;
75     msg.msg_controllen = sizeof (tmp);
76 #endif
77
78     if ((n = recvmsg (socket, &msg, 0)) == -1)
79         printf ("%s: recvmsg: %s", __func__, strerror (errno));
80
81     if (n != 1)
82         printf ("%s: recvmsg: expected 1 but received %d instead",
83                 __func__, n);
84
85 #ifdef HAVE_ACCRIGHTS_IN_MSGHDR
86     if (msg.msg_accrightslen != sizeof (fd))

```

```
87         printf("recv_fd(): no fd\n");
88 #else
89     cmsg = CMSG_FIRSTHDR(&msg);
90     if (cmsg->cmsg_type != SCM_RIGHTS)
91         printf("recv_fd(): expected type %d got %d",
92                SCM_RIGHTS, cmsg->cmsg_type);
93     fd = (*(int *)CMSG_DATA(cmsg));
94 #endif
95     return fd;
96 #else
97     return -1;
98 #endif
99 }
```

Listing D.11: xcode/lib/share_fd.c - Die Funktionen recv_fd und send_fd.

Literaturverzeichnis

- [XPG] www.opengroup.com
- [Korn90] Eine Beschreibung der Safe/Fast String/File IO Library erhalten Sie durch Senden einer Email:
`echo 'send attgifts/sfio.shar' | mail netlib@research.att.com`
- [Stevens91] Stevens, W. R., Advanced Programming in the UNIX Environment, Addison Wesley 1991
- [Tho74] Ken Thompson, 1974, *The UNIX Time-Sharing System*,
<http://cm.bell-labs.com/cm/cs/who/dmr/cacm.pdf>
- [Rit79] Dennis M. Ritchie, 1979, *The Evolution of the Unix Time-sharing System*,
<http://cm.bell-labs.com/cm/cs/who/dmr/hist.pdf>
- [Chou01] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, *An empirical study of operating system errors*, Proc. 18th ACM Symposium on Operating Systems Principles, 2001, pp. 73f
- [FSF] The Free Software Foundation

Index

_POSIX_JOB_CONTROL, 177
_exit, 88, 154, 155, 168, 211
abort, 88, 168, 211, 212
access, 62
alarm, 209
asctime, 143
atexit, 168
brk, 157
cfgetispeed, 243
cfgetospeed, 243
cfsetispeed, 243
cfsetospeed, 243, 252
changedir, 80
chdir, 79
chgrp, 128
chmod, 62, 65
chown, 62, 67
clearerr, 97
clone, 198
close, 40, 42, 88
closedir, 76
connect, 26
creat, 40, 41
ctime, 143
dirent, 76
dup, 50, 88, 118
dup2, 50
endgrent, 123, 128
endhostent, 130
endnetent, 133
endprotoend, 136
endpwent, 123, 125
endutent, 139
endutxent, 141
exec, 4, 20, 88, 150–156, 160, 163, 164, 183, 209,
 230
execl, 161, 164
execle, 161, 162, 183
execlp, 161, 162
execve, 162, 183
execvp, 151, 180
exit, 88, 155, 168, 211
export, 157
F_GETLK, 115
F_SETLK, 115, 116
F_SETLKW, 116
F_UNLCK, 115
fchdir, 79
fchmod, 65
fchown, 67
fclose, 88, 96, 211
fcntl, 50, 52, 88, 110, 112, 115, 118, 236
fdopen, 84, 88, 93–96
fflush, 94
fgetc, 88, 97, 98
fgetpos, 43, 89, 103, 104
fgets, 16, 18, 96, 98, 230
FILE, 87–89
flock, 112
flush, 96
fopen, 84, 87, 88, 93, 94
fork, 4, 20, 88, 92, 150–154, 156, 163, 164, 170, 198,
 209
fpathconf, 34
fpipe, 88
fprintf, 105, 106
fputc, 88, 98
fputs, 96, 98, 99
fread, 96, 99–101
free, 148
freopen, 88, 89, 93, 94
fscanf, 107
fseek, 94, 103, 104
fsetpos, 94, 103, 104
fstat, 18, 56
ftell, 103, 104
ftello, 104
fully buffered, 88, 90
fwrite, 96, 99–101
getaddrinfo, 133
getc, 97–100
getchar, 97
getcwd, 79, 81
getegid, 150
getenv, 183
geteuid, 150
getgid, 150
getgrent, 123, 128
getgrgid, 123, 128
getgrnam, 127, 128
getgroups, 123, 128

gethostbyaddr, 130
gethostbyname, 130
getlogin, 177
getmsg, 26
getnetbyaddr, 133, 134
getnetbyname, 133, 134
getnetent, 133
 getopt, 184, 186–188, 190
 getopt_long, 196
 getoptreset, 186
 getpid, 150
 getpmsg, 26
 getppid, 151
 getprotobynumber, 135
 getprotoent, 136
 getpwent, 123, 125, 126
 getpwnam, 123–125
 getpwuid, 123–125, 177
 getrnam, 123
 gets, 16, 18, 98
 getservbyname, 137
 getservbyport, 137
 getsockopt, 190, 192, 193
 gettimeofday, 143
 getuid, 140, 150, 177
 getutent, 139
 getutid, 140
 getutline, 140
 getutxent, 141
 getutxid, 141
 gmtime, 143
 grep, 182

i-node, 68
i-nodes, 68
init, 5, 149, 169
initgroups, 123
ioctl, 26, 31

kill, 28, 198, 204, 207–209

lchown, 67
line buffered, 88, 90
link, 27, 69
localtime, 143
lockf, 112
login, 124
longjmp, 211
lseek, 42, 44, 88, 89, 103
lstat, 18, 56

malloc, 81, 149
mkdir, 75
mkstemp, 83
mktemp, 83
mktime, 143
mq_open, 22

NGROUPS_MAX, 128
open, 14, 18, 40, 62, 74, 84, 87, 88, 110, 182, 235–
 237
opendir, 76, 77

pathconf, 34, 237
pause, 204, 205, 212–214, 228, 229
pipe, 88
printenv, 182, 183
printf, 104, 105, 145
pthread_self, 207
putc, 98–100
putchar, 98
puts, 98, 99
pututline, 140
pututxline, 141

raise, 206, 208
read, 26, 42, 44, 88, 100, 110, 118, 236–238
readdir, 76
readlink, 73
rename, 73
rewind, 94
rewinddir, 76
rmdir, 75

SA_SIGINFO, 225
scanf, 15, 16, 107, 108
seekdir, 76
sem_open, 22
sentenv, 157
setbuf, 88, 90, 91
setgrent, 123, 128
setgroups, 123, 129
setguid, 5
sethostent, 130
setnetent, 133
setprotoent, 136
setpwent, 123, 125
setregid, 174
setreuid, 174
setservent, 137
setsid, 175, 180
setsockopt, 29
settimeofday, 143
setuid, 5, 62, 141
setutent, 139
setutxent, 141
setvbuf, 88, 90, 92, 93
shm_open, 22
SIG_BLOCK, 224
SIG_DFL, 164, 203–205, 212, 215
SIG_ERR, 204
SIG_HOLD, 215
SIG_IGN, 203, 215, 216, 219, 225
SIG_SETMASK, 224
SIG_UNBLOCK, 224

SIGABRT, 168, 211, 212
 sigaction, 200, 205, 215, 223–226, 228
 sigaddset, 214, 220, 224
 SIGALRM, 206, 209, 210
 SIGCHLD, 164, 169, 198
 SIGCONT, 175, 208
 sigdelset, 215, 220
 sigemptyset, 214, 220, 224
 sigfillset, 214, 220
 SIGFPE, 207
 sighold, 215
 SIGHUP, 175, 176, 204
 sigignore, 215, 216
 SIGINT, 164, 197, 205, 218, 219, 223, 224, 228, 230
 sigismember, 220, 221, 224
 sigispending, 224
 SIGKILL, 202, 208, 214, 215
 siglongjmp, 211
 siglongjump, 230
 signal, 28, 202–205, 215, 223–226
 sigpause, 215, 216
 sigpending, 221, 223, 224
 sigprocmask, 213, 214, 221, 222, 224, 226, 229
 SIGQUIT, 164, 230
 sigrelease, 216
 SIGSEGV, 198
 sigset, 215, 224
 sigsetjump, 230
 SIGSTOP, 202, 215
 sigsuspend, 213, 214, 226, 228, 229
 SIGTERM, 207–209
 SIGTSTP, 240
 SIGUSR1, 203, 204
 sigwait, 198, 199
 Single UNIX Specification, 21
 sleep, 205
 snprintf, 105
 sprintf, 105
 sscanf, 107, 108
 stat, 55, 61, 66, 70, 84
 stderr, 89
 STDERR_FILENO, 89, 90, 93, 151
 stdin, 89, 98
 STDIN_FILENO, 89, 90, 93, 151
 stdout, 89, 92
 STDOUT_FILENO, 89, 90, 93, 151, 152
 stime, 143
 strcat, 18
 strchr, 189
 strcpy, 18
 Stream, 87
 strftime, 143
 strlen, 18
 strncat, 18
 strncpy, 18
 strrchr, 14
 subgetopt, 190
 symlink, 73
 sysconf, 34, 236
 system, 150, 164, 167, 224
 Systemaufrufe, 4
 tcgetpgrp, 176
 tcsetpgrp, 176
 termios, 236, 239, 242
 times, 178
 TIOCGPGRP, 175
 TIOCSPGRP, 175
 tmpnam, 82
 umask, 62, 63
 uname, 22, 142
 unbuffered, 88, 90
 ungetc, 97, 104
 ungetwc, 104
 unlink, 69, 72, 84
 utime, 55, 59
 utmp, 138
 utmpname, 140, 142
 utmpx, 138
 utmpxname, 142
 utsname, 22
 vfork, 150, 153, 155, 156, 198
 wait, 27, 163, 168–172, 178, 210
 waitpid, 27, 168, 169, 171, 172, 178
 WIFSIGNALED, 171
 WNOHANG, 169
 write, 29, 42, 46, 88, 100, 110, 118, 239
 WUNTRACED, 169