Kapitel 14

Sicheres Programmieren mit C

Gerade bei systemnaher Software muss man sich auch über Sicherheit Gedanken machen. Diese Software wird oft mit Superuser-Rechten ausgeführt (z. B. *passwd*) und von „normalen"' Benutzern aufgerufen (bzw. kann zumindest von "'normalen" Benutzern beeinflusst werden). Durch die *Laufzeitumgebung* (z. B. *Umgebungsvariablen*), *Eingaben*, *Kommandozeilen-Argumente* etc. kann u. a. durch den Aufrufer auf das Programm Einfluss genommen werden. Bietet das Programm Angriffspunkte, so wird dies garantiert von Angreifern ausgenutzt, die so im schlimmsten Fall Superuser- bzw. Root-Rechte erhalten.

14.1 Typische Schwachstellen

Um gängige Schwachstellen in C kennen zu lernen, betrachten wir folgendes Beispiel-Programm:

Programm 14.1: C-Programm mit vielen Schwachstellen (*pubcat.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define BUFSIZE 256
5  #define PUBDIR "/home/thales/jmayer/pub"
6
7  int main(int argc, char *argv[]) {
8     char s[BUFSIZE];
9
10    if (argc != 2) {
11       fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
12       return 1;
13    }
14
15    sprintf(s, "cat %s/%s", PUBDIR, argv[1]);
16
17    system(s);
18
19    return 0;
20 }
```

Mit diesem Programm soll es möglich sein, Dateien aus einem öffentlichen Ordner anzeigen zu lassen. Hierzu verwenden wir die Funktion *system()* aus *stdlib.h*, die das über-

gebene Unix-Kommando ausführt. Das Kommando *cat* gibt einfach die Dateien auf die Standardausgabe aus, deren Namen es als Kommandozeilen-Argumente erhält.

Programm 14.1, das *Benutzer jmayer* erzeugt hat, wird nun übersetzt und als Kommando *pubcat* in das Verzeichnis /tmp gestellt:

```
thales$ id
uid=13847(jmayer) gid=200(sai)
thales$ gcc -Wall -o pubcat pubcat.c
thales$ cp pubcat /tmp/pubcat
thales$ ll /tmp/pubcat
-rwxr-x---    1 jmayer   sai          6774 Dec 10 13:08 /tmp/pubcat
thales$ ll /home/thales/jmayer/pub/
-rw-------    1 jmayer   sai            15 Dec 10 13:08 welcome.txt
thales$ cat /home/thales/jmayer/pub/welcome.txt
Sie sind drin!
thales$ /tmp/pubcat welcome.txt
Sie sind drin!
thales$
```

Jetzt kann auch *Benutzer mayer* das Kommando *pubcat* ausführen. Da das Heimatverzeichnis von jmayer aber niemandem Zugriff erlaubt, bleiben alle Leseversuche erfolglos:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ cat /home/thales/jmayer/pub/welcome.txt
cat: /home/thales/jmayer/pub/welcome.txt: Permission denied
thales$ /tmp/pubcat welcome.txt
cat: /home/thales/jmayer/pub/welcome.txt: Permission denied
thales$
```

Der *Benutzer jmayer* setzt aber nun das *Set-User-ID-Bit* des Kommandos *pubcat*:

```
thales$ id
uid=13847(jmayer) gid=200(sai)
thales$ chmod u+s /tmp/pubcat
thales$ ll /tmp/pubcat
-rwsr-x---    1 jmayer   sai          6774 Dec 10 13:08 /tmp/pubcat
thales$
```

Das Setzen des *Set-User-ID-Bits* hat zur Folge hat, dass das Kommando *pubcat* nicht mehr mit den Benutzerrechten des Aufrufers, sondern mit den Benutzerrechten des Datei-Eigentümers ausgeführt wird. Folglich kann nun der *Benutzer mayer* mit dem Kommando *pubcat* die Datei welcome.txt lesen, obwohl er selbst immer noch keinen Zugriff auf diese Datei hat:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ cat /home/thales/jmayer/pub/welcome.txt
cat: /home/thales/jmayer/pub/welcome.txt: Permission denied
thales$ /tmp/pubcat welcome.txt
Sie sind drin!
thales$
```

Nun tritt gleich die erste *Schwachstelle* des Programms zu Tage: Der *Dateiname wird nicht überprüft*. Das ist fatal, denn nun kann einfach mittels „..“ ins Heimatverzeichnis und von dort aus ins *.ssh*-Verzeichnis gewechselt und der *private Schlüssel* ausgegeben werden:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ /tmp/pubcat ../.ssh/id_dsa
-----BEGIN DSA PRIVATE KEY-----
[...]
-----END DSA PRIVATE KEY-----
thales$
```

Es kommt noch schlimmer! Man kann sich einfach der *Metasymbole der Shell* bedienen, um den „Dateinamen“ komplett umzufunktionieren. Zwei Shell-Kommandos kann man einfach durch einen Strichpunkt voneinander trennen. Bauen wir einen Strichpunkt in den Dateinamen ein, so wird alles danach als weiteres Kommando interpretiert. Die gemäßigte Variante ist, dass wir als zweites Kommando *id* verwenden, um die Benutzerkennung, unter der das Programm ausgeführt wird, auszugeben. Wir können aber auch das Programm *sh* verwenden. Dann haben wir eine *Shell mit den Rechten des Benutzers jmayer*:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ /tmp/pubcat 'welcome.txt; id'
Sie sind drin!
uid=15157(mayer) gid=200(sai) euid=13847(jmayer)
thales$ /tmp/pubcat 'welcome.txt; sh'
Sie sind drin!
thales$ id
uid=15157(mayer) gid=200(sai) euid=13847(jmayer)
thales$ ^D
thales$ id
uid=15157(mayer) gid=200(sai)
thales$
```

Angenommen, wir hätten gar keinen Einfluss auf die Argumente des ausgeführten Programms. Dann gibt es immer noch eine gravierende Schwachstelle. Das Kommando *cat* wird nämlich in allen Verzeichnissen, die in der Umgebungsvariable *PATH* angegeben sind gesucht, da *cat* nicht mit dem *absoluten Pfad* angegeben ist. D. h., dass ein Angreifer ganz einfach ein *eigenes Kommando cat* z. B. im Verzeichnis /tmp erzeugen kann und dann die Variable PATH auf den Wert /tmp setzen kann. Danach führt das Kommando pubcat genau das Kommando cat des Angreifers aus:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ ll /tmp/cat
-rwxr-x---   1 mayer    sai          44 Dec 10 14:08 /tmp/cat
thales$ cat /tmp/cat
#!/bin/sh
echo "Mein Programm!"
/usr/bin/id
thales$ /tmp/pubcat welcome.txt
Sie sind drin!
thales$ PATH=/tmp
thales$ /tmp/pubcat welcome.txt
Mein Programm!
uid=15157(mayer) gid=200(sai) euid=13847(jmayer)
thales$
```

Normalerweise würde ein Angreifer natürlich in seinem eigenen cat eine Shell starten, mit der er dann unter den Benutzerrechten von jmayer arbeiten kann.

Selbst wenn man den absoluten Pfad des Kommandos cat angibt, gibt es immer noch Angriffsmöglichkeiten. Wenn z. B. das Kommando */usr/local/bin/cat* innerhalb von *pubcat.c* aufgerufen würde, so könnte ein Angreifer den IFS – eine Umgebungsvariable – neu setzen. Der IFS wird von der Shell verwendet bei der Zerlegung einer Kommandozeile in das Kommando und seine Argumente. Normalerweise enthält er Leerzeichen, Tabulator und Zeilenumbruch. Setzt man den IFS jedoch auf den Wert „/”‘“, *so würde ”‘/usr/local/bin/cat* interpretiert als der Aufruf des Kommandos „usr”’ mit den Argumenten ”‘local”’, ”‘bin”’ und ”‘cat“. Dann kann man wie vorher durch Setzen von PATH ein *eigenes Kommando usr* ausführen.

Eine weitere große Sicherheitslücke von Programm 14.1 ist, dass mittels *sprintf()* das Kommando „zusammengebaut“ wird und dabei nicht darauf geachtet wird, ob der Platz in *s* ausreicht. Man kann sich ja vorstellen, dass das Argument beliebig groß werden kann. So schreibt *sprintf()* in einen Speicherbereich außerhalb des Arrays. Der Zugriff kann hier also außerhalb der Grenzen des Arrays erfolgen (*buffer overflow*). Somit kann ein Angreifer Maschinencode in diesem Array plazieren und dann noch die Rücksprungadresse (innerhalb einer Funktion) geeignet setzen, so dass sein Code angesprungen wird. Je größer das Array dimensioniert ist, desto einfacher wird es für den Angreifer, sinnvollen Maschinencode im Array zu plazieren – dann kann es auch ein etwas längeres Programm sein. ;-) Auf dieser Basis existieren viele Angriffe. Dies ging hier jedoch viel einfacher.

Ein weiterer typischer Fehler ist, wenn dem Benutzer die Möglichkeit gegeben wird, auf den Format-String bei *printf()* Einfluss zu nehmen. Dann kann er nämlich die Formatierungsangabe *%n* unterbringen, die bewirkt, dass die Anzahl der bisher ausgegebenen Zeichen in das Argument – das als Zeiger auf eine Integer interpretiert wird – geschrieben wird! Das würde man auf den ersten Blick nicht vermuten, dass *printf()* nicht nur Argumente liest, sondern evtl. auch verändert:

Programm 14.2: printf() und die Formatierungsangabe %n (*printf.c*)

```
 1  #include <stdio.h>
 2
 3  int main() {
 4      int n;
 5
 6      printf("%d%n\n", 3, &n);
 7      printf("n = %d\n", n); /* Anzahl bis %n ausgegebener Zeichen */
 8
 9      printf("%d%n\n", 33, &n);
10      printf("n = %d\n", n); /* Anzahl bis %n ausgegebener Zeichen */
11
12      return 0;
13  }
```

```
thales$ gcc -Wall printf.c
thales$ a.out
3
n = 1
33
n = 2
thales$
```

Zusammenfassung der typischen Schwachstellen von C-Programmen:

- *Indizierungsfehler bei Arrays:* Diese können einerseits direkt durch den Programmierer

passieren, wenn er auf einen Index zugreift, den es gar nicht gibt. Andererseits kann so ein Fehler auch in den Funktionen *gets()*, *sprintf()*, *strcpy()*, *strcat()*, etc. passieren.

- *Unzureichende Überprüfung von Argumenten:* In unserem Fall war das eine nicht ausreichende Überprüfung eines „Dateinamens". Beim Arbeiten mit Dateien, Ausführen von Kommandos oder Arbeiten mit Systembefehlen ist immer Vorsicht geboten, wenn der Benutzer Einfluss auf die Argumente haben kann.
- *Benutzereinfluss auf dem Format-String bei printf():* Wir haben gesehen, dass *printf()* seine Argumente nicht nur zum Lesen, sondern manchmal auch zum Schreiben verwendet. Dieses unerwartete Verhalten ist u. U. fatal, wenn der Benutzer Einfluss auf den Format-String nehmen kann.
- *Benutzung eines Zeigers nach seiner Freigabe*
- *doppelte Speicherfreigabe mit free()*

Fazit: Die Vermeidung dieser Fehler ist alles andere als trivial, wie gefundene Schwachstellen in sicherheitsrelevanter Software wie etwa ssh (*secure shell*) und ssl (*secure socket layer*) belegen. Daher ist es für systemnahe Software ratsam, auf problematische Funktionen der C-Bibliothek zu verzichten und stattdessen auf sicherere Alternativen auszuweichen.

14.2 Dynamische Strings

Es ist zwar sehr einfach möglich – mittels *calloc()* bzw. *strdup()* – einen dynamischen String anzulegen, jedoch bieten die Funktionen *strcpy()*, *strcat()*, *sprintf()*, etc. keine Möglichkeit der Überprüfung, ob die allokierte Länge nicht überschritten wird – was nicht zuletzt an der fehlenden Größeninformation bei C-Arrays liegt.

Es bietet sich also die Verwendung der Bibliothek *libowfat* (http://www.fefe.de/libowfat/) an, die von Felix von Leitner nach einem Vorbild von Dan J. Bernstein nachprogrammiert und unter die GPL (GNU General Public License) gestellt wurde. Diese Bibliothek ist bei uns lokal unter */usr/local/diet* installiert.

Bei C-Arrays fehlt im Wesentlichen die Größeninformation. Dies behebt die folgende Datenstruktur für Strings in der *stralloc-Bibliothek*:

```
typedef struct stralloc {
    char* s;  /* Zeichen des Strings (i.A. ohne '\0' am Ende) */
    unsigned int len; /* Laenge des Strings (len <= a) */
    unsigned int a; /* Laenge des Arrays s */
} stralloc;
```

Hinter *s* verbirgt sich das Zeichenarray, das im Allgemeinen nicht nullterminiert ist. Dies hat den Vorteil, dass Strings auch das Null-Byte enthalten können. Die Komponente *len* enthält die momentane Länge des Strings und *a* ist die momentante Länge des Arrays *s*; also gilt folglich immer $len \leq a$.

Da C lokale Variablen nicht automatisch initialisiert, müssen diese jeweils „von Hand" initialisiert werden:

```
stralloc sa = {0};
```

Man beachte, dass durch obige Initialisierung nicht nur *sa.s*, sondern auch *sa.len* und *sa.a* auf 0 initialisiert werden.

Im Folgenden sind die wesentlichen Funktionen der *stralloc-Bibliothek* kurz beschrieben. Der Rückgabewert dieser Funktionen ist bei Erfolg 1 und sonst 0.

int stralloc_ready(stralloc* sa,unsigned int len)
Stellt sicher, dass genügend Platz für *len* Zeichen vorhanden ist.

int stralloc_readyplus(stralloc* sa,unsigned int len)
Stellt sicher, dass genügend Platz für weitere *len* Zeichen vorhanden ist.

int stralloc_copys(stralloc* sa,const char* buf)
Kopiert den nullterminierten String *buf* nach *sa*.

int stralloc_copy(stralloc* sa,const stralloc* sa2)
Kopiert *sa2* nach *sa*.

int stralloc_cats(stralloc* sa,const char* buf)
Hängt den nullterminierten String *buf* an *sa* an.

int stralloc_cat(stralloc* sa,stralloc* sa2)
Hängt *sa2* an *sa* an.

int stralloc_0(stralloc* sa)
Hängt ein Null-Byte an *sa* an.

void stralloc_free(stralloc* sa)
Gibt den von *sa* belegten Speicher wieder frei – also den Speicher von *sa*−>*s* (und nicht den Speicher für die stralloc-Struktur).

Das folgende Programm demonstriert die Verwendung der stralloc-Bibliothek. Dieses Programm enthält auch je eine kleine Funktion, um einen stralloc-String auszugeben und eine (beliebig lange) Zeile von der Standardeingabe zu lesen.

Programm 14.3: Arbeiten mit der stralloc-Bibliothek (*stralloc.c*)

```
1  #include <stdio.h>
2  #include <stralloc.h>
3
4     /* sa auf Standardausgabe schreiben */
5  void print(stralloc *sa) {
6     int i;
7     for (i = 0; i < sa->len; i++)
8        putchar(sa->s[i]);
9  }
10
11    /* eine Zeile von der Standardeingabe lesen */
12 int readline(stralloc *sa) {
13    if (!stralloc_copys(sa, "")) return 0;
14    while (1) {
15       if (!stralloc_readyplus(sa, 1)) return 0;
16       if (fread(sa->s + sa->len, sizeof(char), 1, stdin) <= 0) return 0;
17       if (sa->s[sa->len] == '\n') break;
18       sa->len++;
19    }
20    return 1;
21 }
22
23 int main() {
24    stralloc sa = {0};
25
26    if (readline(&sa)) {
27       print(&sa); puts("");
```

```
28
29          printf("Laenge␣von␣sa:␣%d\n", sa.len);
30          stralloc_0(&sa);
31          printf("Laenge␣von␣sa:␣%d\n", sa.len);
32
33          puts(sa.s);
34
35          stralloc_free(&sa);
36      }
37
38      return 0;
39  }
```

```
thales$ gcc -Wall -I /usr/local/diet/include/ -L /usr/local/diet/lib/ \
> stralloc.c -lowfat
thales$ a.out
Meine Eingabe ...
Meine Eingabe ...
Laenge von sa: 17
Laenge von sa: 18
Meine Eingabe ...
thales$
```

Anmerkungen: Bei der Übersetzung (bzw. dem Linken) muss man dem gcc den Pfad der Include-Datei und den Pfad der Bibliothek angeben und die Bibliothek libowfat mit der *Option -lowfat* hinzubinden.

14.3 Zusammenfassung und Fazit

- *C-Arrays (und somit auch Strings) enthalten keine Größeninformation.* Dies kann leicht zu unerlaubten Zugriffen führen, die nicht abgefangen werden. Betroffen sind auch beliebte Funktionen wie etwa *strcpy()*, *strcat()*, *gets()*, *sprintf()*, etc.
- *„Traue nichts und niemandem"* ist dringend anzuraten. Deshalb sollten Eingaben aller Art (Kommandozeilen-Argumente, Umgebungsvariablen, Standardeingabe, Dateieingabe, etc.) überprüft werden.
- *Positivlisten* (Was ist erlaubt?) sind bei der Überprüfung von Eingaben *besser und sicherer als Negativlisten* (Was ist verboten?)!
- Der *wohldefinierte Bereich einer Programmiersprache* sollte nie verlassen werden (z. B. durch den Zugriff auf nicht vorhandene Array-Elemente).
- Soweit *automatische Überprüfungen* möglich sind, sollten sie verwendet werden.
- Sind die automatische Überprüfungen nicht hinreichend (wie z. B. bei Array-Zugriff), so sollte etwa auf eine *andere Bibliothek* (z. B. stralloc-Bibliothek) ausgewichen werden.

Kapitel 15

Das Aufbau des Betriebssystems Unix

15.1 Betriebssysteme allgemein

15.1.1 Definition

Die *DIN-Norm 44300* definiert wie folgt, was ein *Betriebssystem* ist:

> Zum Betriebssystem zählen die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Das Betriebssystem ist das *erste Programm*, welches *nach dem Starten eines Rechners* geladen wird (abgesehen von der Firmware). *Solange der Rechner in Betrieb ist*, läuft das Betriebssystem.

15.1.2 Aufgaben

Das Betriebssystem hat zwei zentrale *Aufgaben*:

- **Ressourcen-Management:**

  Das Betriebssystem kontrolliert alle Hardware- und Software-Komponenten eines Rechners und teilt sie *effizient* den einzelnen Nachfragern zu.

  Das Betriebssystem stellt Basis-Dienstleistungen (Dateizugriff, Prozessmanagement) zur Verfügung, auf denen Anwendungsprogramme aufsetzen können.

- **„Erweiterte Maschine“:** (sog. *Virtual Machine*)

  Das Betriebssystem besteht aus einer (oder mehreren) Schichten von Software, die über der „nackten“ Hardware liegen. Diese *Virtual Machine* ist einfacher zu verstehen und zu programmieren. Komplexe Hardware-Details verbergen sich hinter einfacheren und einheitlichen *erweiterten Instruktionen* (*Systemaufrufe*).

  Der Programmierer erhält vom Betriebssystem eine angenehmere Schnittstelle zur Hardware. Dabei wird von Hardware-Details wir *Speicherorganisation, I/O-Struktur, Bus-Struktur, etc.* abstrahiert. Der Programmierer muss sich nicht mehr um sämtliche maschinennahen Details kümmern.

15.1.3 Schichtenmodell

Das *Schichtenmodell* teilt die *„erweiterte Maschine“* in die folgenden Schichten ein (siehe Abbildung 15.1):

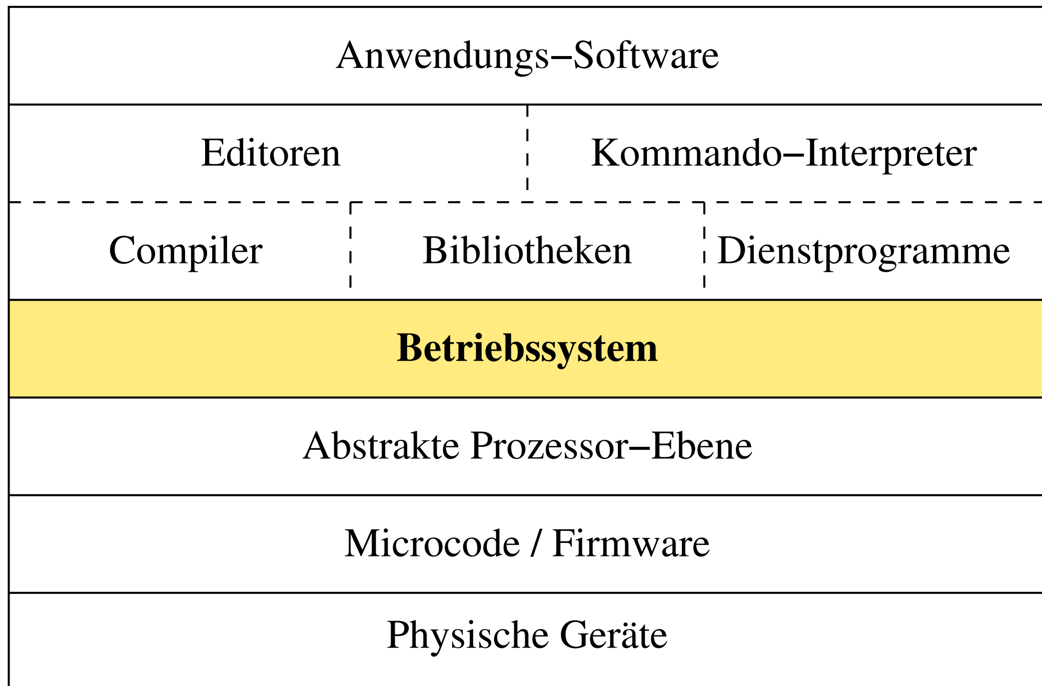

Abbildung 15.1: Das Schichtenmodell der „erweiterten Maschine“

- **Physikalische Geräte:**

  Prozessor, Festplatten, Grafikkarte, Stromversorgung, etc.

- **Microcode / Firmware:**

  Software, die die *physikalischen Geräte direkt kontrolliert* und sich teilweise direkt auf den Geräten befindet. Diese bietet der nächsten Schicht eine einheitlichere Schnittstelle zu den physikalischen Geräten. Dabei werden *einige Details* der direkten Gerätesteuerung *verborgen*.

  Beispiel: Abbildung logischer Adressen auf physikalische Adressen bei Festplatten.

- **Maschinensprache:**

  Stellt die Schnittstelle zwischen Hardware und Software dar. Je nach dem ob *RISC (Reduced Instruction Set Computer)* oder *CISC (Complex Instruction Set Computer)* zwischen 50 und 300 Instruktionen (ADD, MOVE, JUMP, etc.). Die meisten dienen zum *Kopieren von Daten* zwischen den Registern und dem Hauptspeicher, dem Ausführen von (teils bedingten) *Sprüngen* und dem Ausführen von *arithmetischen und Vergleichs-Operationen*.

- **Betriebssystem:**

  Das Betriebssystem vermittelt zwischen Anwender bzw. Anwenderprogramm und Hardware. Es verbirgt die Komplexität der Hardware vor dem Programmierer und bietet angenehmere Instruktionen für das Programmieren (*open(), lseek(), read(), close()* statt „move head of disc 1 to track 231, lower head, ...“).

- **System-Software:**

  Compiler, Editoren, Kommando-Interpreter, Dienstprogramme – gehören nicht zum Kern des Betriebssystems, werden aber meist vom Hersteller des Betriebssystem(-Kern)s mitgeliefert.

- **Anwendungs-Software:**

  Von Benutzern bzw. für Benutzer zur Lösung ihrer Probleme entwickelt (Beispiel: Textverarbeitungsprogramm).

15.2 Unix-Schalenmodell

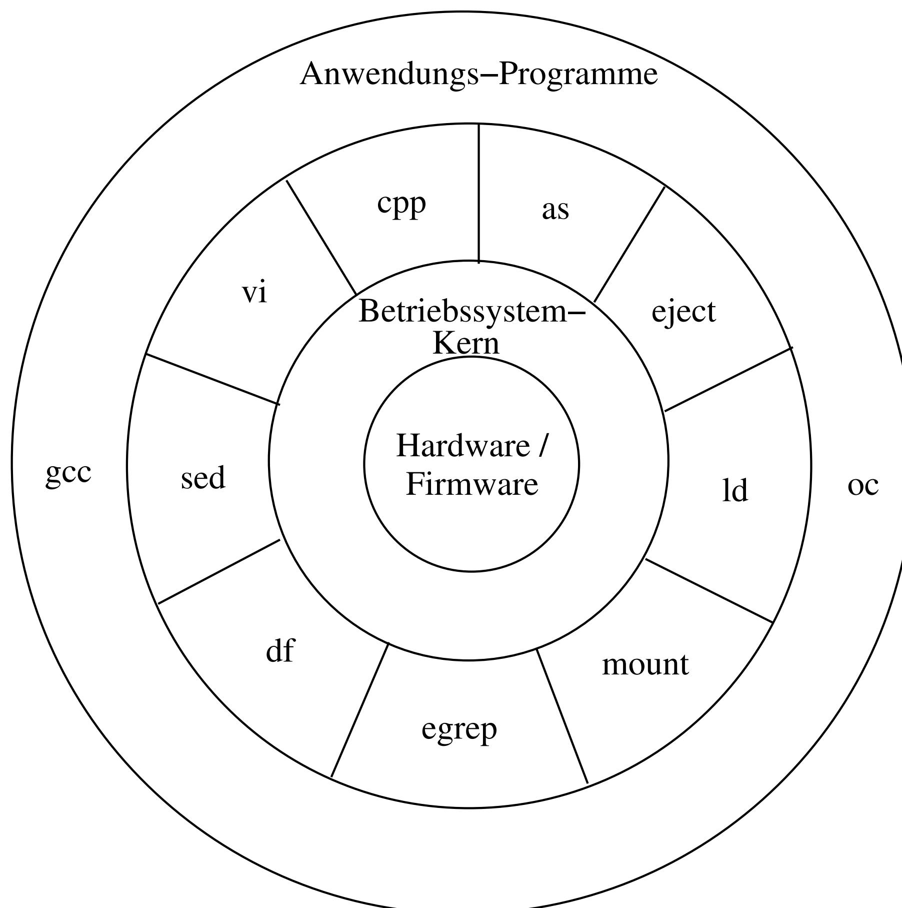

Abbildung 15.2: Das Unix-Schalenmodell

Abbildung 15.2 zeigt das *Unix-Schalenmodell*. Die wesentlichen Punkte sind:

- **Abstraktion:**

  Das Betriebssystem übernimmt die Kommunikation mit der Hardware. Es isoliert die Programme von Hardware-Details. Der Programmierer kann mit abstrakten Objekten (z. B. Filedeskriptor) statt mit hardware-nahen Details (z. B. Plattenadressen) arbeiten. Dadurch werden Programme hardware-unabhängig und somit leichter portierbar.

- **Interaktion:**

  Programme kommunizieren mit dem Betriebssystem-Kern über klar definierte Systemaufrufe (*System Calls*), um Dienstleistungen zu erhalten und Daten auszutauschen.

- **Kombination:**

  Benutzerprogramme und (System-)Kommandos befinden sich in derselben Schicht. Sie können aufeinander aufbauen, sich gegenseitig aufrufen oder Daten miteinander austauschen.

15.3 Interner Aufbau von Unix

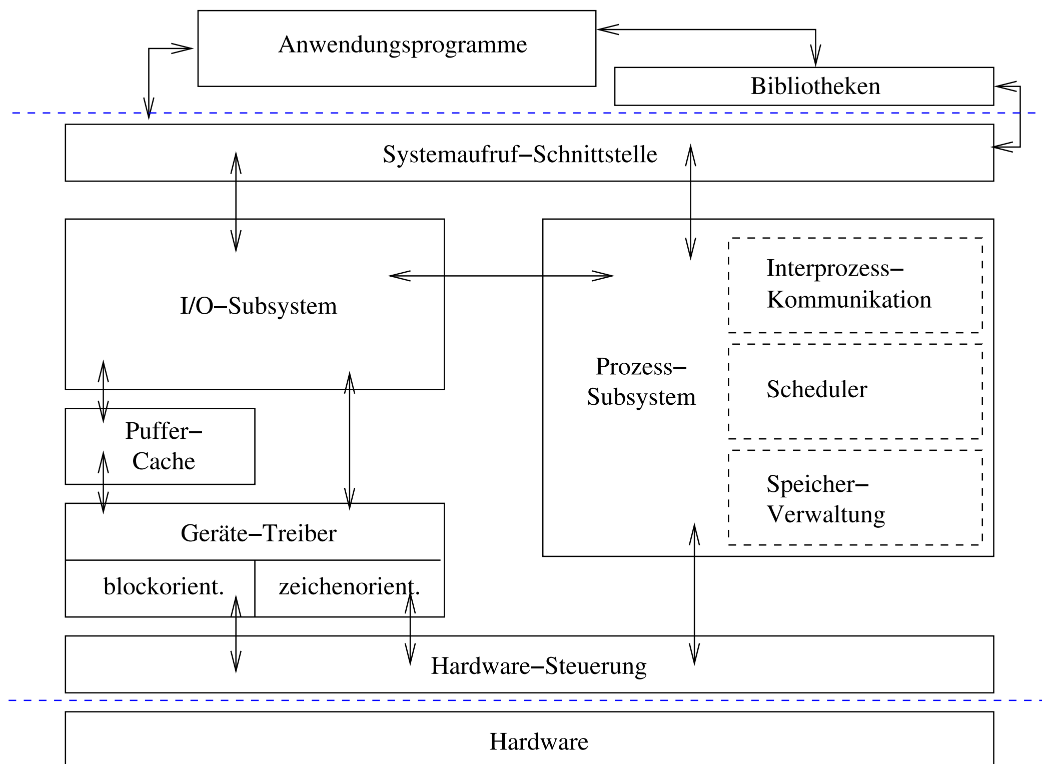

Abbildung 15.3: Der interne Aufbau von Unix

Der interne Aufbau des Betriebssystems Unix ist in Abbildung 15.3 veranschaulicht.

- **Systemaufrufe (system calls):**

  Die Systemaufrufe bilden die *Schnittstelle* zwischen dem Betriebssystem-Kern und den Anwendungsprogrammen. Über sie können die *Dienstleistungen des Betriebssystems* in Anspruch genommen werden. In einem Programm sind Systemaufrufe einfach *Funktionsaufrufe*.

- **Subsysteme:**

  Intern kann man den Betriebssystem-Kern in *zwei Subsysteme* aufteilen – das *Ein-/Ausgabe-Subsystem* (oder *I/O-Subsystem* oder *I/O Subsystem*) und das *Prozess-Subsystem*.

Ersteres ist verantwortlich für alles, was mit Dateien zu tun hat. Letzteres ist verantwortlich für Programmausführung, Speicherzuteilung, etc. Die Systemaufrufe lassen sich je einem dieser beiden Subsysteme zuordnen.

- **Zugriffskontrolle:**

  Die *Systemaufrufe* bieten den *einzig möglichen Zugang zu den Ressourcen* (Hardware). Es gibt keinen direkten Zugriff der Benutzerprogramme auf die Ressourcen (CPU, Hauptspeicher, Festplatten, etc.).

- **Transparenz:**

  Die Dienstleistungen des Betriebssystem-Kerns sind transparent. (*Beispiel:* Alles ist eine Datei: „Echte Datei", Gerät, Pipe, Netzwerkverbindung, etc.)

Kapitel 16

Das I/O-Subsystem

16.1 Dateien

16.1.1 Was ist eine Datei?

Was ist eigentlich eine Datei? Unter UNIX ist eine Datei wie folgt definiert: „Every file is a sequence of bytes.“ Also: *Eine Datei ist einfach eine Folge von Bytes*.

Zu einer Datei gehören

- ein oder auch mehrere *Namen*,
- der *Inhalt* und *Aufbewahrungsort* (Menge von Blöcken auf der Platte, etc.) und
- *Verwaltungsinformationen* (Besitzer, erlaubter Zugriff, Zeitstempel, Länge, Dateityp, etc.).

UNIX verlangt und unterstellt bei regulären (d. h. gewöhnlichen) Dateien *keinerlei Struktur* und unterstützt auch keine. Die Konzepte variabel oder konstant langer Datensätze (Records) sind im Kernel von UNIX nicht implementiert. *Eine Datei wird abstrakt als Datenstrom repräsentiert.*

Neuartig unter UNIX ist, dass praktisch *alle Objekte des Systems durch Dateien repräsentiert* und somit abstrahiert sind. Hinter einer Datei kann sich Plattenplatz, ein Gerät, eine Pipeline (zur Prozesskommunikation), etc. „verstecken“.

16.1.2 Aufgaben des Betriebssystems

Im Zusammenhang mit Dateien hat das UNIX-Betriebssystem folgende Aufgaben:

- Speichern und Wiederbeschaffen von Benutzerdaten
- Bereitstellen von Platten-Speicherplatz
- Verwalten von freiem Plattenplatz
- Zugriff auf Geräte ermöglichen
- Kontrolle der Zugriffsrechte auf Dateien

16.1.3 Dateioperationen

Der Zugriff auf Dateien erfolgt immer einheitlich mit denselben Dateioperationen (öffnen, schließen, lesen, schreiben, etc.) via Systemaufruf.

Beispiele für relevante Systemaufrufe:

chdir	Katalog wechseln
chmod	Zugriffsrechte ändern
chown	Besitzer ändern
close	Datei schließen
dup	Filedeskriptor duplizieren
dup2	Filedeskriptor duplizieren
fnctl	Eigenschaften einer offenen Datei ändern
ioctl	Ähnlich zu fcntl, aber Geräte-spezifisch
link	Neuer Name für bestehende Datei
lseek	Positionieren in einer Datei
mknod	Besondere Dateien erzeugen
mount	Dateisystem in bestehendes „einhängen“
open	Öffnen/Erzeugen einer Datei
pipe	Namenlose Pipeline erzeugen
read	Lesen aus einer Datei
stat	Attribute einer Datei lesen
umount	Dateisystem „aushängen“
unlink	Namen entfernen
write	Schreiben in eine Datei

16.1.4 Dateitypen

Obwohl Dateien als Abstraktionen für alle Objekte des Systems dienen, benötigt UNIX nur wenige verschiedene *Dateiarten* (*file types*):

- *Gewöhnliche Datei* (*ordinary file* oder *regular file*):
  Container für jede Art von Daten (sowohl Text als auch binär!)

- *Verzeichnis* bzw. *Katalog* (*directory file*):
  Verzeichnis für die Namen von Dateien und zugehörige Verweise auf „Verwaltungsinformation“ (*Inode*). Ein Verzeichnis hat immer eine fest vorgegebene Struktur.

- *Gerätedatei* (*device special file*: character special file / block special file):
  Geräte und Hardware allgemein; zu Dateien abstrahiert (Terminal, Drucker, Platten, Hauptspeicher, etc.)

- *Symbolischer Link* (*symbolic link*):
  Verweis auf den Pfadnamen einer anderen (u. U. nicht existierenden) Datei

- *Benannte Pipeline* (*named pipe*; *FIFO*):
  Zur unidirektionalen Prozesskommunikation

- *Socket*:
  Kommunikationsstruktur; insb. zur Kommunikation in Rechnernetzen

- und einige wenige mehr (System V, BSD)

16.1.5 Gerätedateien

Es gibt zwei Arten von Geräten:

- **Zeichengeräte** (*character devices*):
  Dies sind Geräte auf die ungepuffert zugegriffen werden kann, gemäß dem Modell „unformatierter sequentieller Bytestrom“.
  Beispiele: Terminal, Drucker

- **Blockgeräte** (*block devices*):
  Dies sind Geräte, die Daten in beliebig adressierbaren Blöcken speichern und übertragen können.
  Beispiel: Festplatte

Das I/O-Subsystem benutzt einen Puffermechanismus beim Zugriff auf Daten, die sich auf einem Blockgerät (z. B. Festplatte) befinden, den sog. *Puffer-Cache*.

Die Module des Kernels, welche unmittelbar auf die physikalischenn Geräte zugreifen, heißen *Gerätetreiber*. Sie haben eine *Geräte-unabhängige Schnittstelle* „nach oben" zu den übrigen Kernel-Routinen und eine *Geräte-abhängige Schnittstelle* „nach unten" zur Hardware hin.

16.2 Dateisysteme

16.2.1 Arten von Dateisystemen

Man unterscheidet prinzipiell drei Arten von Dateisystemtypen:

- **Plattenbasierte Dateisysteme:**
  Die wichtigsten Typen (im folgenden weiter betrachtet) sind die plattenbasierenden wie z. B. das *UFS* (*Unix File System*), *FAT* (*File Allocation Table* – von DOS oder Windows) und *NTFS* (*New Technology File System*), die auf Festplatten, Disketten oder CDs (ISO-Dateisystem) angelegt werden.

- **Netzwerk-Dateisysteme:**
  Andere Dateisysteme basieren auf Netzwerken; dazu zählen die Typen *NFS* (*Network File System*) oder *RFS* (*Remote File System*). *NFS* ist das Dateisystem, welches im *WiMa-Netz (!)* (z. B. für Heimatverzeichnisse) Verwendung findet.

- **Pseudo-Dateisysteme:**
  Schließlich gibt es eine Reihe spezieller Dateisystemtypen, die unter dem Begriff *Pseudodateisysteme* (*Pseudo Filesystems*) zusammengefasst werden.
  Beispiel: Das Prozess-Dateisystem (*procfs*)

16.2.2 Netzwerk-Dateisysteme

16.2.2.1 Allgemeines

Über ein Netzwerk verteilte Dateisysteme basieren auf dem sogenannten *Client/Server-Prinzip*. Der Server ist dabei der Rechner, der die Dateisysteme, die er lokal verwaltet, anderen Rechner verfügbar macht. Der Client ist der Rechner, der diese Leistung auf einem Server nutzt. Mischformen sind durchaus üblich: Rechner, die selbst anderen Rechnern Teile ihres Dateisystems zur Verfügung stellen, selbst auch welche von anderen Rechnern erhalten (Beispiel: unsere Server thales, turing, etc.). Es können nur lokale Dateisysteme weitergegeben werden. Abbildung 16.1 zeigt ein Beispiel für ein verteiltes Dateisystem (nach [Handschuch 1993]).

16.2.2.2 Network File System (NFS)

Das *Network File System* (*NFS)* wurde von *Sun Microsystems Inc.* für die Kopplung von Rechnern mit unterschiedlicher Hardware und unterschiedlichem Betriebssystem entwickelt. Damit sollte über das Netzwerk der Zugriff eines Rechners auf die Festplatten eines anderen Rechners ermöglicht werden: Daten lesen und verarbeiten, auch wenn diese in einem

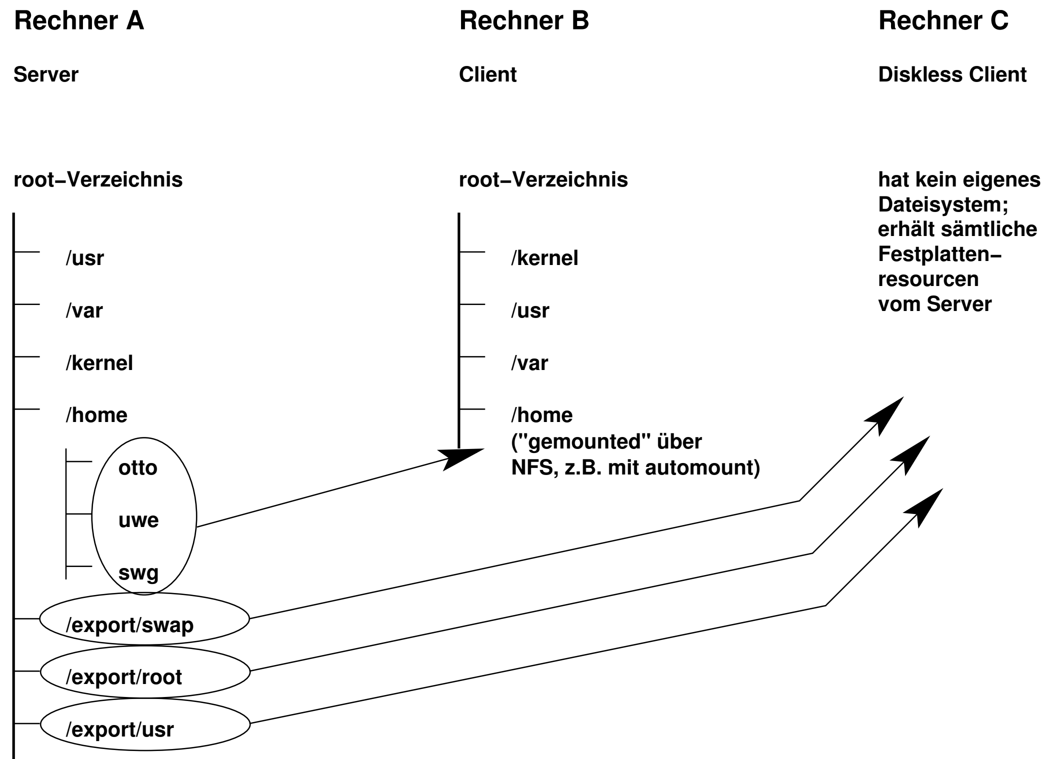

Abbildung 16.1: Beispiel für ein Netzwerk-Dateisystem

Format abgelegt sind, die nur das dortige Betriebsystem „kennt“. Die notwendige Umsetzung der Datenstruktur wird vom *XDR-Protokoll* (*eXternal Data Representation*) vorgenommen; die Steuerung des Zugriffs erfolgt über *RPC* (*remote procedure call*). Der Zugriff wird transparent durchgeführt, also ohne Eingabe von besonderen Befehlen. Systemaufrufe wie *read()* oder *write()*, die einem remote-System gelten, werden *auf dem Klienten wie lokale Operationen behandelt*, aber über das Netzwerk weitergeleitet.

Die NFS-Implementierung von Sun unterstützt sog. zustandslose Server:

- Das Öffnen einer Datei mittels *open()* wird zunächst im Client zurückgehalten, falls sich diese Datei auf einem remote-Dateisystem befindet.

- Die Übertragung der für *open()* spezifischen Daten geschieht erst bei einem nachfolgenden *read()*- oder *write()*-Aufruf, zusammen in einer Transaktion.

- Ein „Absturz“ des remote-Dateisystems zwischen *open()* und z. B. *read()* führt so zu keinen Problemen, die Zustände sind lokal.

- NFS nimmt keine Verwaltung des Zugriffs von Clients vor, den Clients ist ebenso der Zustand des Servers nicht bekannt.

- Die Anforderungen nach einem NFS-Zugriff durch einen Client sind alle unabhängig voneinander, und mit sämtlichen Informationen ausgestattet, die der Server für die Erfüllung der Aufgabe braucht.

- Problematische Zustände pflanzen sich nicht im Netzwerk fort.

- Dadurch ist eine konsistente Arbeitsweise des Servers gewährleistet, da ein Auftrag immer vollständig abgearbeitet werden kann, sobald er über das Netz beim Server eintrifft.

Funktionsweise von NFS-Anforderungen:
Wird von einem Prozess (NFS-Client) eine Zugriffsanforderung auf eine Datei abgesetzt, so wird dies als Betriebssystemaufruf an die Dateisystem-Schnittstelle interpretiert. Hier wird mit Hilfe des *VFS* (*Virtual File System*) festgestellt, wie die gewünschte Datei erreichbar ist: lokal oder über das Netz. Wenn lokal erreichbar, so wird sie mit Hilfe des entsprechenden Plattentreibers von einer lokalen Platte geladen. Ist die Datei nur über das Netz erreichbar, so wird der Betriebssystemaufruf des Client über NFS an die Dateisystemschnittstelle des Servers weitergeleitet und dort behandelt. Auf dem Server ist die Datei dann lokal erreichbar, die Datei kann gelesen und das Ergebnis an den Client über das Netz zurückgesandt werden. Dabei werden immer nur die Daten übertragen, die auch direkt zur Ausführung eines Befehls benötigt werden. Bei einem Zugriff auf eine große Datei wird diese bei einem Zugriff nicht in voller Länge übertragen, sondern nur der Teil, der für die Verarbeitung gerade benötigt wird.
Konsequenzen:

- Über NFS können entfernte Dateien schnell gelesen und geschrieben werden, andere Operationen wie das *Sperren von Dateien* (*File Locking*) sind tunlichst zu vermeiden, da das NFS-Protokoll selbst kein Sperren von Dateien vorsieht.
- Änderungen in Dateien wie in deren Inodes erfolgen zeitverzögert – so kann es durchaus sein, dass zwei verschiedene Prozesse ein- und dieselbe Datei Löschen können, ohne dass einer von ihnen eine Fehlermeldung „Datei nicht vorhanden“ bekommt!
- Das *Sperren von Dateien* wird bei NFS über den *NFS lock daemon* realisiert; dies ist ein Prozess, der im Auftrag von Prozessen auf anderen Rechnern das Sperren von Dateien übernimmt. (Bei der Verwendung von *lockf()*, das bei entfernten Dateien den NFS lock daemon verwendet, kann es zu Prozessen kommen, die temporär – selbst mit `kill -9` – nicht beendet werden können!)

16.2.2.3 Remote File System (RFS)

Das *RFS* (*Remote File System*) ist ähnlich dem NFS. Es kann im Gegensatz dazu aber nur in homogenen Netzen verwendet werden (alle Systeme müssen Unix System V als Betriebssystem haben, da keine Datentyp-Konvertierungen durchgeführt werden). Dieses System von *Sun Microsystems Inc.* hat sich allerdings in der Praxis nicht durchgesetzt und scheint vom Markt verschwunden zu sein.

16.2.2.4 AFS

AFS ist ein verteiltes Dateisystem, das Rechnern in einem Netzwerk ermöglicht, effizient ihre Dateisystem-Ressourcen zu teilen. Es basiert auf einem verteilten Dateisystem, das ursprünglich als „*Andrew File System*“ an der *Carnegie Mellon Universität* für den Einsatz als hochschulweites Dateisystem für ca. 2000 Arbeitsplätze entwickelt wurde. „Andrew“ war dort die Bezeichnung eines entsprechenden Forschungsprojekts. Das System wird (im Gegensatz zu NFS) inzwischen kommerziell weiterentwickelt und vertrieben (Fa. Transarc) und ist für alle gängigen UNIX-Plattformen verfügbar. Die AFS-Entwicklung ist als Basis für das *Distributed File System* (*DFS*) in den OSF/DCE-Standard eingebracht worden (*OSF*: *Open Software Foundation*, *DCE*: *Distributed Computing Environment*).

Der einheitliche Namensraum von AFS erlaubt es, Dateien unter gleichem Namen von allen Rechnern anzusprechen. Weiterhin ist es möglich, dass die Administrationseinheiten

(eine Abteilung oder andere organisatorische Einheit), *Zellen* (*cells*) genannt, ihre lokalen Datenräume in einem globalen Datenraum zuammenführen. Damit können Benutzer vollkommen transparent auf Datenbestände auf entfernt liegenden Rechnern zugreifen – ohne Kenntnisse über die Lokation der Rechner, nur über den entsprechenden Pfadnamen. AFS folgt ebenfalls dem *Client/Server-Konzept*.

16.2.3 Pseudo-Dateisysteme

16.2.3.1 Das tmpfs-Dateisystem

Ein temporäres Dateisystem kann im Hauptspeicher des Rechners (und damit lokal!) angelegt werden. Diese Art von Dateisystem wird oft auch als *RAM-Disk* bezeichnet. Der Katalog */tmp* wird typischerweise als *tmp-Dateisystem* (oder auch *tmpfs*) angelegt.

Dieses Dateisystem kann wie ein gewöhnliches Dateisystem auch mit Dateien belegt und mit Unterkatalogen organisert werden. Man beachte, dass auf diese Dateien der Zugriff lokal und nicht über NFS erfolgt! Zu beachten ist aber auch, dass alles in diesem Dateisystem nach einem Neustart des Rechners oder nach einem *umount* nicht mehr vorhanden ist, entsprechend der Philosophie „temporär“. Man kann es aber sehr wohl zur Synchronisation mittels *link* oder *open* (also Sperren durch Nutzen von dateiorientierten Systemaufrufen) nutzen, weil kein NFS-Zugriff erfolgt!

Der Vorteil des *tmp-Dateisystems* liegt in der *Performance* (Hauptspeicherzugriff!).

16.2.3.2 Das proc-Dateisystem

Das *proc-Dateisystem* (kurz: *procfs*) dient zur Verwaltung von Prozessen. Es ist ein virtuelles Dateisystem, d. h. es belegt keinen „echten“ Platz auf der Platte, sondern existiert im Hauptspeicher. Die Prozesse sind nach ihrer Prozess-ID (PID) geordnet darin abgelegt. Programme wie *ps* greifen darauf zu.

16.2.4 Das Unix-Dateisystem (UFS)

16.2.4.1 Prinzipieller Aufbau

Ein UNIX-System enthält in der Regel mehrere Unix-Dateisysteme, von denen eines das *Wurzeldateisystem* (*root file system*) ist, welches beim Wurzelkatalog „/“ eingehängt ist.

Abbildung 16.2 zeigt die Grobstruktur eines Unix-Dateisystems mit den folgenden we-

Boot–block	Super–block	Inode–Liste	Datenblöcke

Abbildung 16.2: Grobstruktur des Unix-Dateisystems

sentlichen Elementen:

- **Bootblock**:
  Beim Wurzeldateisystem enthält dieser Block den sog. *bootstrap code*, der beim „Hochfahren“ des Betriebssystems als erstes geladen wird. Bei anderen Dateisystemen ist er zwar vorhanden, wird aber nicht verwendet.

- **Superblock**:
  Enthält die folgenden Verwaltungsinformationen des Dateisystems:

  – Größe des Dateisystems

  - Anzahl freier Blöcke
  - Liste der freien Blöcke
  - Zeiger auf den nächsten freien Block in der Liste der freien Blöcke (hier: Zeiger = Index!)
  - Größe der Inode-Liste
  - Anzahl freier Inodes
  - Liste der freien Inodes
  - Zeiger auf nächste freie Inode in der Liste der freien Inodes (hier wiederum: Zeiger = Index!)
  - „Sperr-Felder“ für Liste der freien Blöcke / Inodes
  - Anzeigefeld, ob Superblock verändert wurde

- **Inode-Liste**:
  Zu jeder Datei gehört genau eine *Inode* und eine (fast) beliebige Anzahl an Datenblöcken. Die Inode enthält die Verwaltungsinformationen, welche zu dieser Datei gehören. Eine besondere Inode ist die *Wurzel-Inode* (*root inode*), die den Wurzelkatalog repräsentiert und beim Zusammenfügen (⇝ *mount*) von Dateisystemen wichtig ist.

- **Datenblöcke**:
  Ein Datenblock kann entweder zur Adressierung (⇝ *indirekte Adressierung*) oder zum Speichern von *Dateiinhalt* (auch *Verzeichniseinträge*) verwendet werden.

Alle anderen Dateisysteme (außer dem Wurzel-Dateisystem) werden explizit mit dem Kommando *mount* – normalerweise dem Superuser vorbehalten – in das Wurzel-Dateisystem eingefügt (oder in bereits so eingefügte Dateisysteme eingefügt). Deshalb muss beim Arbeiten z. B. mit Disketten das darauf enthaltene Dateisystem (i. A. vom Typ FAT) explizit „ge*mount*et“ werden. Den Wurzelkatalog eine Dateisystems kann man nur auf einen Katalog (⇝ *mount point*) eines bereits „ge*mount*eten"' Dateisystems "'*mount*en“. Mit *umount* kann ein Dateisystem wieder ausgehängt werden, wenn es nicht mehr in Benutzung und nicht das Wurzel-Dateisystem ist.

Aus Performance-Gründen werden Schreiboperationen über einen Puffer im Hauptspeicher abgewickelt. Das System synchronisiert selbständig in gewissen Zeitabständen den Inhalt dieses Puffer mit den Dateisystemen auf den Festplatten – dies geschieht mit dem Kommando *sync*.

Das Kommando *df* gibt die Anzahl der freien Plattenblöcke und Indexverweise in angemeldeten Dateisystemen oder Verzeichnissen aus. Das Kommando *du* (*disk usage*) gibt die Plattenbelegung durch Dateien in Einheiten von 512-Byte-Blöcken an. Bei Verwendung der Option „-k“ wird die Dateigröße in Kilobytes angegeben. (Weitere Infos zu diesen Kommandos gibt’s wie immer auf den Manual-Seiten.)

Beispiel:

```
chomsky$ ls
kap.tex  pics  progs
chomsky$ du -k
4       ./progs
36      ./pics
64      .
chomsky$
```

16.2.4.2 Inodes

Die Datenstruktur *Inode* ist die interne Repräsentation einer Datei. Die *Inode-Nummer* ist ein Index in der Inode-Liste und ist die *eindeutige Identifikation* einer Datei (innerhalb eines Dateisystems), *nicht* der Dateiname.

In der *Inode* sind sind sämtliche Informationen enthalten, die eine Datei beschreiben (siehe z. B. *Kommando ls -lai*):

Inode-Inhalt	**Beispiel**
Eigentümer (owner)	swg
Gruppe (group)	sai
Typ (type)	„normale“ Datei (regular file)
Rechte (rights)	`rw-r--r--`
Letzter Lesezugriff auf den Dateiinhalt (last access)	Wed Jun 21 09:25:11 2000
Letzte Änderung am Dateiinhalt (last modification)	Sun May 21 11:50:15 2000
Letzte Änderung an der Inode (last change)	Wed Jun 21 05:24:20 2000
Anzahl der Dateinamen (link count)	2
Größe (size)	600 Bytes
Blockadressen (block addresses)	...

Adressierung Die Datenblöcke, die zu einer Datei gehören werden direkt bzw. (bis zu dreifach) indirekt adressiert. In der Inode selbst gibt es eine Reihe Einträgen für direkte Adressierung und je einen Eintrag für einfach, zweifach und dreifach indirekte Adressierung (siehe Abbildung 16.3). Bei indirekter Adressierung stehen in Datenblöcken, sog.

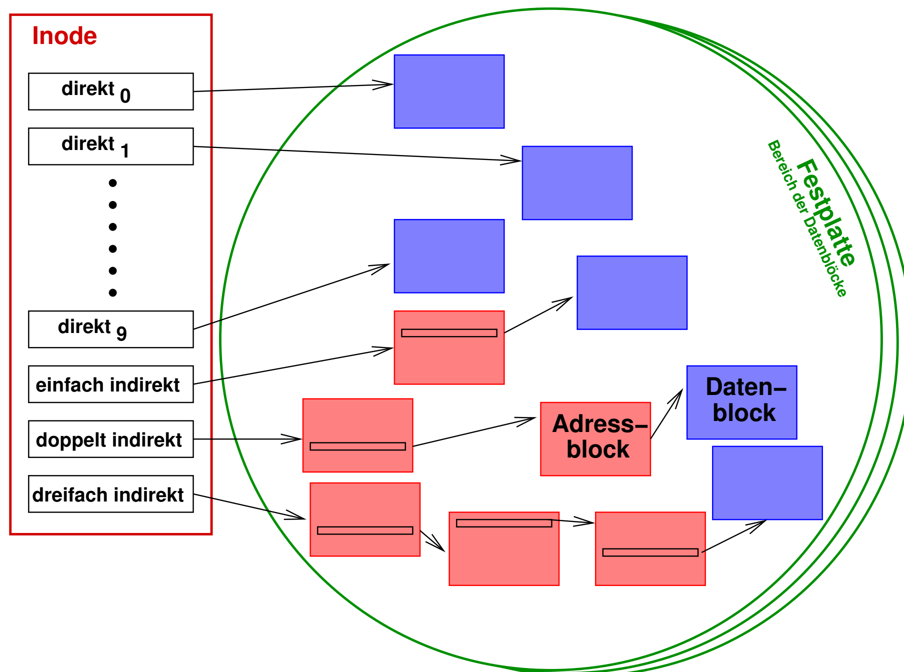

Abbildung 16.3: Adressierung der Datenblöcke

Adressblöcke, wiederum Adressen weiterer Daten- oder Adressblöcke. (Datenblöcke sind im Prinzip Arrays von Blockadressen.)

Durch die einfach, doppelt und dreifach indirekte Adressierung können – je nach gewählter Blockgröße – theoretisch Dateigrößen von bis zu 100 GB erreicht werden. Die eigentliche *Beschränkung* liegt allerdings in der *Angabe der Dateigröße*: Mit 32 Bits können nur Dateigrößen bis 4 GB spezifiziert werden! (Ab Solaris 8 wird die Dateigröße mit 64 Bit beschrieben und dadurch könnten die Dateigrößen theoretisch im Terabytebereich liegen.)

Zugriffsschutz

- **Benutzergruppen**:
  Zur Identifikation eines Benutzers gehört der *Benutzername* (*user name*) und der *Name der Gruppe* (*group name*).

  Während der Benutzername pro System eindeutig ist, fassen die Gruppen jeweils mehrere Benutzer (zum Beispiel alle Mitarbeiter eines bestimmten Projekts oder alle Mitarbeiter einer Abteilung) zu einer Einheit zusammen.

  Ein Benutzer kann eine eigene Gruppe bilden oder zusammen mit anderen Benutzern in einer oder in mehreren verschiedenen Gruppen eingetragen sein. Die Gruppenzuordnung jedes Benutzers lässt sich dynamisch verändern. Die jeweils aktuelle Gruppenzugehörigkeit wirkt sich auf die Zugriffsrechte des Benutzers aus.

- **Benutzerklassen**:
  Die Zugriffsrechte für alle Objekte (d. h. Dateien) des Dateisystems sind nach drei Klassen gestaffelt und für jede Klasse individuell definierbar.

  – Klasse 1: *Eigentümer* (*user*)
    Die Rechte des Besitzers der Datei
  – Klasse 2: *Gruppe* (*group*)
    Die Rechte für die Mitglieder in dieser Gruppe (außer dem Besitzer der Datei)
  – Klasse 3: *Andere* (*other*)
    Die Rechte aller nicht durch die Klassen 1 und 2 erfassten Benutzer

  Wie die Benutzer des Systems haben auch alle Dateien einen Benutzernamen und eine Gruppe. Der Benutzername definiert den Besitzer. Der Gruppenname bestimmt die Gruppe für den Zugriffsschutz.

- **Individueller Schutz**:
  Der Schutzmechanismus in UNIX ermöglicht das individuelle Festlegen von Zugriffsrechten für jedes Objekt (d. h. Datei) im Dateibaum. Der Besitzer kann bei jedem Objekt die Rechte für jede der drei Benutzerklassen separat festlegen – mit dem *Kommando chmod*.

  Für jede der drei Benutzerklassen kann jedes der folgenden drei *Zugriffsrechte* gesetzt bzw. nicht gesetzt sein:

  – **r** (read)
    Berechtigt zum Lesen einer Datei oder der Einträge eines Katalogs.
  – **w** (write)
    Berechtigt zum Verändern einer Datei oder eines Katalogs. Bei Katalogen erlaubt dies primär das Hinzufügen von neuen Einträgen. Details zum Löschen von Einträgen s.u.
  – **x** (execute)
    Berechtigt zum Ausführen einer Datei oder zum Wechseln in einen Katalog (*Kommando cd*).

- **Zugriff auf eine Datei**:
  Ein Prozess (= Ausführung eines Programms) hat vier IDs:

- reale Benutzer-ID (real user ID)
- real Gruppen-ID (real group ID)
- effektive Benutzer-ID (effective user ID)
- effektive Gruppen-ID (effective group ID)

Für den Zugriff auf Dateien sind die effektiven IDs wichtig.

Ein Prozess erhält nun unter folgenden Bedingungen Zugriff auf eine Datei:

1. Ist die effektive Benutzer-Id 0 (superuser), so ist ein Zugriff immer erlaubt.
2. Stimmt die effektive Benutzer-ID des Prozesses mit dem Besitzer der Datei überein und sind die Zugriffsrechte entsprechend dem Zugriff (read, write) richtig gesetzt, so ist Zugriff möglich.
3. Stimmt die effektive Benutzer-ID des Prozesses *nicht* mit dem Besitzer der Datei überein
und
stimmt die effektive Gruppen-ID des Prozesses mit der Gruppen-ID des Besitzers der Datei überein
und
und stimmen Zugriffswunsch und Zugriffsrechte überein,
so ist der Zugriff erlaubt.
4. Stimmt die effektive Benutzer-ID des Prozesses *nicht* mit dem Besitzer der Datei überein
und
stimmt die effektive Gruppen-ID des Prozesses *nicht* mit der Gruppen-ID des Besitzers der Datei überein
und
und stimmen Zugriffswunsch und Zugriffsrechte für Andere überein,
so ist der Zugriff erlaubt.

- **Konsequenzen**:
  Dieser Mechanismus kann sowohl einzelne Dateien als auch ganze Teilbäume vor dem Zugriff einiger, vieler oder aller Benutzer schützen.

  In früheren UNIX Versionen genügte bereits das Schreib-Recht für einen Katalog, um alle darin befindlichen Dateien löschen zu können. In neueren UNIX Versionen kann der Besitzer des Katalogs durch Setzen des sog. *Sticky-Bits* (mit dem Kommando *chmod +t*) verlangen, dass zusätzlich mindestens eine der folgenden Bedingungen erfüllt ist, bevor ein Benutzer eine Datei aus dem Katalog löschen kann:

  - Benutzer ist Besitzer der Datei
  - Benutzer ist Besitzer des Katalogs
  - Benutzer hat Schreib-Recht auf die Datei
  - Benutzer ist Super-User

  Will der Katalogbesitzer diese Sicherheitsmaßnahme aktivieren, muss er das Kommando *chmod +t* auf seinen Katalog anwenden.

  Der *Superuser* ist ein besonders ausgezeichneter Benutzer (Benutzername *root*, id=0), bei dem sämtliche Mechanismen des Zugriffsschutzes *nicht* greifen.

Formen einer Inode Abhängig vom aktuellen Speicherort gibt es die Datenstruktur „Inode“ in zwei verschiedenen Ausprägungen:

- *Arrayelement in der Inode-Liste* eines Dateisystems
- Inode als *Element der Kernel-Inode-Tabelle* im Hauptspeicher (*in-core inode*)

Beide Ausprägungen sind funktional identisch, nur im Aufbau etwas unterschiedlich.

Zusätzliche Einträge der *in-core inode*:

- Status
  - Inode gesperrt
  - Prozess wartet auf Freigabe
  - Inhalt ist anders als bei *disk inode*
  - Datei ist ein *mount point*
- Logische Geräte-Nummer
- Eigene *Inode-Nummer*
- Referenz-Zähler – gibt die Zahl der aktiven Instanzen dieser Datei an (⇝ in der OFT, kommt noch!)

Auf diesen (öffentlichen) Teil der Inode (siehe */usr/include/sys/stat.h*) kann man vom Programm aus zugreifen:

```
/* sys/stat.h: stat structure, used by stat(2) and fstat(2) */
struct stat {
  dev_t   st_dev;   /*device number*/
  ino_t   st_ino;   /*inode number*/
  mode_t  st_mode;  /*mode and type of file*/
  nlink_t st_nlink; /*number of links to file*/
  uid_t   st_uid;   /*user ID of the file's owner*/
  gid_t   st_gid;   /*group ID of the file's group*/
  dev_t   st_rdev;  /*ID of (raw)device if block/char special*/
  off_t   st_size;  /*file size in bytes*/
  time_t  st_atime; /*time of last access*/
  time_t  st_mtime; /*time of last data modification*/
  time_t  st_ctime; /*time of last file status change*/
};
```

Die Komponente *st_mode* enthält sowohl den Dateityp, als auch die Zugriffsrechte – bitweise kodiert in einer Integer (siehe Abb. 16.4 und Abb. 16.5).

Außer den Zugriffsrechten und dem Dateityp gibt es noch drei weitere Bits:

- **Set-UID-Bit** (UID = user id)
  Wir ein Programm ausgeführt, bei dem dieses Bit gesetzt ist, so erhält der Prozess (= Ausführung des Programms) die effektive Benutzer-ID (effective user id) des Besitzers der Programm-Datei. (Beispiel: Kommando *passwd*)
- **Set-GID-Bit** (GID = group id)
  Hier muss man zwischen Verzeichnissen und „normalen“ Dateien unterscheiden. Handelt es sich um eine „normale“ Datei, also um ein Programm, so wird – analog zum Set-UID-Bit – bei der Ausführung die effektive Gruppen-ID auf die Gruppe der Programm-Datei gesetzt. Im Falle eines Verzeichnisses erhalten alle in diesem Verzeichnis angelegten Dateien die Gruppe des Verzeichnisses als Gruppe.

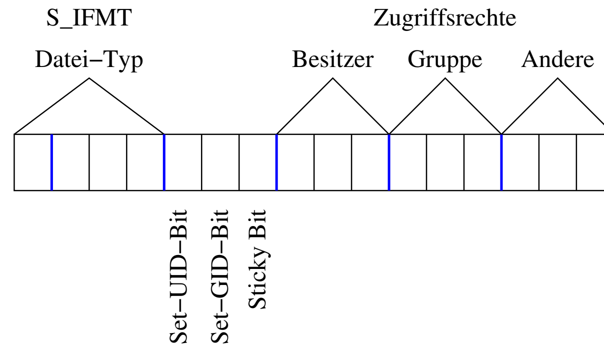

Abbildung 16.4: Kodierung der Komponente `st_mode`

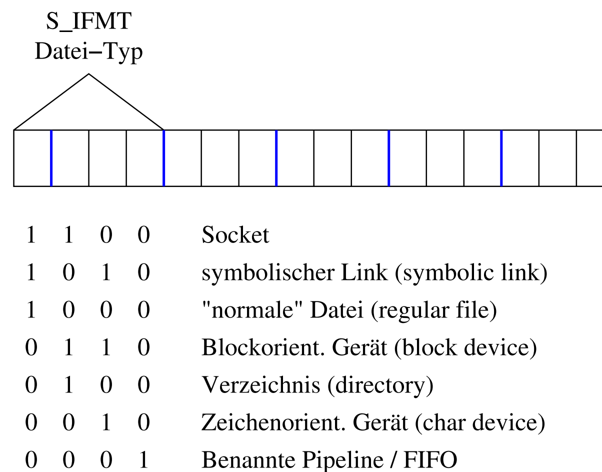

Abbildung 16.5: Kodierung des Dateityps in `st_mode`

- **Sticky-Bit**
  Bewirkt bei Verzeichnissen das Einschalten einer zusätzlichen „Sicherung“ gegen das unerlaubte Löschen von Dateien – wie bereits besprochen. Handelt es sich um ein ausführbares Programm (als *read-only*, d. h. der Programm-Text darf nicht verändert werden), so verbleibt eine Kopie des Programm-Textes im Swap-Bereich (nach Beendigung des Programms) und kann beim nächsten Start schneller geladen werden.

Kommando / Bibliotheksfunktion zum Ändern der Datei-Zugriffsrechte:
chmod / ***int chmod(const char *path, mode_t mode);***

Etwas Spielerei mit den Rechten:

```
spatz$ mkdir tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users 11276 2005-01-16 17:40 a.out
drwxrwxr-x  2 swg users    48 2005-01-16 17:47 tmp
spatz$ chmod +t tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users 11276 2005-01-16 17:40 a.out
drwxrwxr-t  2 swg users    48 2005-01-16 17:47 tmp
spatz$ cd tmp
spatz$ > xyz
spatz$ ls -l
total 0
-rw-rw-r--  1 swg users 0 2005-01-16 17:48 xyz
spatz$ cd ..
spatz$ chmod g+s tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users 11276 2005-01-16 17:40 a.out
drwxrwsr-t  2 swg users    72 2005-01-16 17:48 tmp
spatz$ groups
users uucp dialout audio video
spatz$ chown :dialout tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users   11276 2005-01-16 17:40 a.out
drwxrwsr-t  2 swg dialout    72 2005-01-16 17:48 tmp
spatz$ id
uid=1000(swg) gid=100(users) groups=14(uucp),16(dialout),
17(audio),33(video),100(users)
spatz$ cd tmp
spatz$ ls -l
total 0
-rw-rw-r--  1 swg users 0 2005-01-16 17:48 xyz
spatz$ > uvw
spatz$ ls -l
total 0
-rw-rw-r--  1 swg dialout 0 2005-01-16 17:51 uvw
-rw-rw-r--  1 swg users   0 2005-01-16 17:48 xyz
spatz$
```

Folgendes Programm verwendet den Systemaufruf *stat()*:
int stat(const char *file_name, struct stat *buf);

stat() erwartet als ersten Parameter einen Dateinamen und liefert dann im zweiten Parameter vom Typ *struct stat* die Informationen über die Datei. Im Fehlerfall ist der Rückgabewert negativ.

Programm 16.1: Informationen über Dateien ermitteln – mittels stat() (*stat.c*)

```
 1 #include <sys/stat.h> // fuer struct stat
 2 #include <unistd.h> // fuer stat(.)
 3 #include <time.h> // fuer ctime(.)
 4 #include <stdlib.h> // fuer exit(.)
 5 #include <stdio.h>
 6
 7 int main(int argc, char **argv) {
 8     int i; char *s;
 9     struct stat statbuf;
10     for (i = 1; i < argc; i++) {
11         printf("Datei␣\"%s\":\n", argv[i]);
12
13         if (stat(argv[i], &statbuf) < 0)
14             perror("stat"), exit(1); // Abbruch im Fehlerfall
15
16         /* ctime(.) erzeugt aus der Datenstruktur time_t einen String */
17         printf("␣␣␣␣␣␣␣␣letzter␣Zugriff:␣%s",
18                ctime(&statbuf.st_atime));
19         /* statbuf.st_mtime und statbuf.st_ctime
20          * koennen analog behandelt werden. */
21
22         switch (statbuf.st_mode & S_IFMT) {
23             case S_IFREG : s = "normale␣Datei"; break;
24             case S_IFDIR : s = "Verzeichnis"; break;
25             case S_IFCHR : s = "zeichenorient.␣Geraet"; break;
26             case S_IFBLK : s = "blockorient.␣Geraet"; break;
27             case S_IFLNK : s = "symbolischer␣Link"; break;
28             case S_IFIFO : s = "FIFO"; break;
29             case S_IFSOCK : s = "Socket"; break;
30             default: s = "_unbekannt_"; break;
31         }
32         printf("␣␣␣␣␣␣␣␣Datei−Typ:␣%s\n", s);
33     }
34     return 0;
35 }
```

```
chomsky$ gcc stat.c
chomsky$ a.out stat.c /dev/tty /dev/hda .
Datei "stat.c":
        letzter Zugriff: Sun Jan 11 23:14:34 2004
        Datei-Typ: normale Datei
Datei "/dev/tty":
        letzter Zugriff: Sun Jan 11 23:14:17 2004
        Datei-Typ: zeichenorient. Geraet
Datei "/dev/hda":
        letzter Zugriff: Sat Jul 29 14:48:22 2000
        Datei-Typ: blockorient. Geraet
Datei ".":
        letzter Zugriff: Sun Jan 11 23:16:04 2004
        Datei-Typ: Verzeichnis
chomsky$
```

Anmerkungen: Mit der Funktion *ctime()* (⇝ *time.h*) kann ein Zeitstempel (Datum und Uhrzeit) als String formatiert werden – mit fest vorgegebenem Format. Außerdem stehen Konstanten *S_IFMT*, *S_IFREG*, etc. bereit, um den Typ einer Datei auf einfache Art und Weise mit einer bitweisen Und-Operation zu ermitteln. Beim Fehlschlagen eines Systemaufrufs kann mittels *perror()* eine ausführliche Fehlermeldung ausgegeben werden – später mehr dazu. Mit der Bibliotheksfunktion *exit()* kann man ein Programm mit dem angegebenen Exit-Status beenden (was innerhalb von *main()* gleichbedeutend einem return ist).

16.2.4.3 Verzeichnisse

Verzeichnisse oder *Kataloge* (*Directories*) sind besondere Dateien mit einer *festen Struktur*. Sie enthalten Einträge mit den Elementen *Dateiname* und *Inode-Nummer*. (Jede Datei hat genau eine Inode, aber evtl. mehrere Dateinamen.)

Beispiel:

Lokaler Dateiname	Verweis auf Inode-Liste (Index)
.	4711
..	4709
onefile	47119
otherfile	471110
subdir_1	471130

Jeder Katalog enthält mindestens zwei besondere Einträge, nämlich „.”‘ und ”‘..”‘. ”‘.“ ist ein Verweis auf den aktuellen Katalog; „..“ ist ein Verweis auf den Vaterkatalog.

Damit ergibt sich die bekannte Benutzer-Sicht: Hierarchisches Dateisystem oder Dateibaum! Die Blätter im Baum sind *normale Dateien*, *benannte Pipelines* oder *Gerätedateien*, die anderen Knoten *Verzeichnisse*.

Mit den Systemaufrufen *opendir()*, *readdir()* (und *closedir()*) kann man die Einträge in einem Verzeichnis wie folgt ermitteln:

Programm 16.2: Dateien in einem Verzeichnis ermitteln (*dir.c*)

```
1 #include <sys/types.h>
2 #include <dirent.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 int main(int argc, char **argv) {
```

```
 7      DIR * dir;
 8      struct dirent *entry;
 9
10      if (argc != 2) {
11          fprintf(stderr, "Usage:␣%s␣<directory>\n", argv[0]);
12          exit(1);
13      }
14
15      if (!(dir = opendir(argv[1])))
16          perror("opendir"), exit(2);
17      while ((entry = readdir(dir)))
18          printf("%ld␣:␣%s\n", entry−>d_ino, entry−>d_name);
19      closedir(dir);
20
21      return 0;
22  }
```

```
thales$ gcc dir.c
thales$ a.out .
2497008 : .
2496977 : ..
2497010 : dir.c
2496000 : a.out
thales$
```

Anmerkungen: Der Rückgabewert von *opendir()* ist genau dann ungleich dem Nullzeiger, wenn das Öffnen erfolgreich war. Andernfalls liegt ein Fehler vor. Die Funktion *readdir()* signalisiert im Falle eines Nullzeigers als Rückgabewert entweder einen Fehler oder, dass kein weiterer Eintrag mehr vorhanden ist. Mit Hilfe von *stat()* könnte man nun zu jedem Verzeichniseintrag wie gehabt weitere Informationen „beschaffen“.

Dateinamen Dateinamen bestehen aus Zahlen, Buchstaben und Sonderzeichen. Erlaubt sind im Prinzip alle Zeichen außer „/”' (Slash) und ”'\0“ (Null-Byte). Verschiedene Zeichen tragen jedoch eine Sonderbedeutung bei der Kommandoeingabe (Shell) und eignen sich daher nur bedingt für Dateinamen.

Dateinamen sind in älteren UNIX-Versionen auf eine Länge von 14 Zeichen begrenzt, in neueren Versionen auf 256 (genauer: 255 + Null-Byte) Zeichen oder mehr.

Standard-Verzeichnisse

- **root directory**:
  Der Katalog am oberen Ende des Dateibaums / heißt Wurzel (*root*). Dieser Katalog ist direkter oder indirekter Vorgänger aller Dateien und Verzeichnisse im gesamten Dateibaum und hat selbst keinen Vorgänger.

- **top level directories**:
  Im oberen Bereich des Dateibaums befinden sich Kataloge mit verschiedenen Systemdaten.

  – */bin* — ausführbare Programme (*binaries*)

  – */lib* — Bibliotheken und Hilfsdateien

  – */tmp* — Platz für temporäre Dateien aller Benutzer – bei einem *Netzwerk-Dateisystem* wie hier im WiMa-Net ist dieser Unterkatalog immer lokal auf der jeweiligen

Maschine, liegt also (aus Performance-Gründen) auf einer an dieser Maschine angehängten Platte oder im Hauptspeicher (⇝ tmpfs)

- */home* — Heimatverzeichnisse
- */etc* — Konfigurationsdateien und Systemverwaltungskommandos
- */proc* — Prozess-Pseudo-Dateisystem (*procfs*); benutzt als Schnittstelle zu den Kernel Datenstrukturen (mehr z.B. `man -S5 proc`)
- */dev* — Gerätedateien (*device special files*)
- */usr* — Benutzerdaten und eher lokale Systemdateien (/usr/bin, /usr/lib, /usr/tmp /usr/man)
- */usr/spool* — Zwischenablage für Druckaufträge, externe Post, usw. (*spooling area*)
- */usr/local* — lokale Software, meist ebenfalls unterteilt in Kataloge mit den Namen *bin, lib, man* und *src*.

16.2.4.4 Links

Hardlinks Mit folgendem Kommando kann man einen *Hardlink* erzeugen:

```
ln name neuername
```

Beim Erzeugen eines Hardlinks wird nur ein *neuer Verzeichniseintrag mit derselben Inode-Nummer* angelegt. (Link – genauer Hardlink – ist also ein Synonym für Verzeichniseintrag!)

Die bisher über den Namen `name` erreichbaren Daten sind jetzt auch über den Namen `neuername` erreichbar. Beim Zugriff besteht zwischen `name` und `neuername` keinerlei Unterschied, da letztendlich auf den gleichen physikalischen Speicherplatz auf der Festplatte zugegriffen wird.

Hardlinks können nur auf „normale" Dateien oder Gerätedateien eingerichtet werden. Beide Objekte, `name` und `neuername`, müssen sich im *gleichen Dateisystem* befinden, da ja über die Inode-Nummer referenziert wird. Der Eintrag `name` muss bereits existieren.

Der Linkzähler (link count) in der Inode gibt an, wieviele Dateinamen dieselbe Datei referenzieren (z. B. „."' und "'.." in einem Unterverzeichnis beziehen sich auf dieselbe Datei).

Beispiel für das Anlegen eines Hardlinks:

```
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:12 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:11 ..
 314005 -rw-r--r--   1 jmayer   users           0 Jan 12 00:12 hallo.txt
chomsky$ ln hallo.txt neu.txt
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:12 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:11 ..
 314005 -rw-r--r--   2 jmayer   users           0 Jan 12 00:12 hallo.txt
 314005 -rw-r--r--   2 jmayer   users           0 Jan 12 00:12 neu.txt
chomsky$
```

Anmerkungen: In der ersten Spalte der Ausgabe von *ls -lai* steht die *Inode-Nummer* und in der dritte Spalte steht der *Link-Zähler* (*link count*).

Softlinks Mit folgendem Kommando kann man einen *Softlink* bzw. *symbolischen Link* erzeugen:

```
ln -s name neuername
```

Beim Erzeugen eines Softlinks wird eine *neue Datei* (vom *Typ „Softlink“*) erzeugt, deren *Inhalt der Name der referenzierten Datei* ist.

Das bisher über den Namen `name` erreichbare Daten-Objekt ist jetzt auch über `neuername` erreichbar. Falls das Daten-Objekt `name` existiert, so bestehen beim Zugriff auf diese Daten via `neuername` aus Sicht des Benutzer keinerlei Unterschiede zum Zugriff über `name`. Im Kernel laufen jedoch zusätzliche Operationen ab (für `neuername` wird eine Inode angelegt!). Softlinks sind folglich „teurer“ als Hardlinks.

Der Eintrag `name` muss nicht unbedingt existieren. Softlinks können außerdem auch auf Verzeichnisse eingerichtet werden und können auch auf andere Dateisysteme verweisen. Der Link-Zähler (link count) wird logischerweise nicht erhöht.

Beispiel für das Anlegen eines Softlinks:

```
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:14 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:13 ..
 314005 -rw-r--r--   1 jmayer   users           0 Jan 12 00:12 hallo.txt
chomsky$ ln -s hallo.txt neu.txt
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:14 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:13 ..
 314005 -rw-r--r--   1 jmayer   users           0 Jan 12 00:12 hallo.txt
 314007 lrwxrwxrwx   1 jmayer   users           9 Jan 12 00:14 neu.txt -> hallo.txt
chomsky$
```

Konsequenzen

- Durch Hardlinks wird aus dem Dateibaum ein zyklenfreier gerichteter Graph.
- Durch Softlinks können auch Zyklen im Dateisystem entstehen (durch Softlinks auf Verzeichnisse)!

16.3 Systemaufrufe für I/O-Verbindungen – Erster Teil

16.3.1 Öffnen von Dateiverbindungen – open()

int open(const char *path, int oflag);
int open(const char *path, int oflag, mode_t mode);

Der Systemaufruf *open()* mit *zwei* Parametern öffnet eine existierende Datei. Die Variante mit *drei* Parametern ist für den Fall bestimmt, dass eine Datei beim Öffnen zusätzlich angelegt wird.

Der erste Parameter ist der *Name der zu öffnenden Datei*.

Der zweite Parameter enthält die *oflags*, die die Art der I/O-Verbindung und ihre Eigenschaften spezifizieren. *open()* kennt drei Arten von I/O-Verbindungen:

oflag	Verbindungsart
O_RDONLY	nur lesen
O_WRONLY	nur schreiben
O_RDWR	lesen und schreiben

Die *oflags* können nicht nur die Werte *O_RDONLY*, *O_WRONLY* oder *O_RDWR* annehmen, sondern differenziertere Flagwerte. Diese entstehen durch (bitweises) Oder-Verknüpfung von einem oder mehreren der folgenden Flagwerte zur Verbindungsart:

oflag	**Bedeutung**
O_CREAT	Falls die Datei nicht existiert, wird sie angelegt. Nur bei diesem Flagwert findet das dritte Argument Verwendung.
O_TRUNC	Falls die Datei bereits existiert, wird sie auf Länge 0 verkürzt, d. h., der alte Inhalt wird „gelöscht".
O_APPEND	Dieses Flag stellt sicher, dass jede Schreib-Operation am Ende der Datei erfolgt, auch bei konkurrierenden Schreib-Operationen verschiedener Prozesse.
O_EXCL	Zusammen mit dem *O_CREAT*-Flag stellt dieses Flag sicher, dass *open()* fehlschlägt, wenn `path` bereits existiert.
O_NONBLOCK *O_NDELAY*	Prozess blockiert nicht beim Öffnen eines noch nicht bereiten IPC-Kanals (FIFO). Zusätzlich wirkt sich dieses Flag auf nachfolgende *read()* und *write()* Systemaufrufe aus.
O_SYNC	Bei jedem *write()* auf diese Dateiverbindung blockiert der Prozess bis die Schreib-Operation physikalisch abgeschlossen ist.

Beliebige Kombinationen sind möglich, aber nicht alle sind sinnvoll! *fcntl.h* enthält für *oflag* die Definitionen der symbolischen Konstanten (Makros).

Der dritte Parameter von *open()* ist nur relevant, wenn eine Datei geöffnet werden soll, die noch nicht existiert. In diesem Fall soll die Datei ggf. angelegt werden (über den zweiten Parameter festzulegen – durch Oder-Verknüpfung mit *O_CREAT*); dabei müssen dann die Zugriffsrechte auf diese neue Datei durch den dritten Parameter festgelegt werden (in der Form `rwxrwxrwx` für Eigentümer, Gruppe und Andere und zwar binär kodiert – wie gehabt bei *stat()*, nur ohne Dateityp etc.).

Eine erfolgreiche Ausführung des Systemaufrufs liefert einen *Filedeskriptor* zurück (genauer: den *kleinsten freien* Filedeskriptor). Dieser ist eine kleine, nicht-negative Zahl, die bei allen nun folgenden I/O-Operationen (*read()*, *write()*, …, *close()*) diese I/O-Verbindung identifiziert. Sie stellt einen Index in eine entsprechende Tabelle (UFDT) dar. *stdin* steht für den Filedeskriptor 0, *stdout* für 1 und *stderr* für 2. Diese sind beim Start eines Prozesses typischerweise bereits geöffnet.

Im Falle eines Fehlers liefert *open()* einen negativen Wert.

Beispiele:

- Eine Datei zum Schreiben anlegen. Wenn sie bereits existiert, dann soll sie auf Länge 0 verkürzt werden:

```
int fd;
if ((fd = open(dateiname, O_WRONLY | O_CREAT | O_TRUNC, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror(öpen()");
␣␣␣␣␣␣␣␣exit(1);
␣␣␣␣}
```

- Eine Datei, die bereits existiert, zum Lesen und Schreiben öffnen. Alle Schreiboperationen sollen am Dateiende erfolgen.

```
int fd;
if ((fd = open(dateiname, O_RDWR | O_APPEND, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror(öpen()");
```

```
␣␣␣␣␣␣␣␣exit(1);
␣␣␣␣}
```

- Eine Datei, die noch nicht existieren darf, zum Schreiben öffnen. (Existiert die Datei bereits, so führt dies zu einem Fehler!)

```
int fd;
if ((fd = open(dateiname, O_WRONLY | O_CREAT | O_EXCL, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror(öpen()”);
␣␣␣␣␣␣␣␣exit(1);
␣␣␣␣}
```

Programm 16.3 öffnet ein Datei names „myfile.txt" zum Lesen und Schreiben. Existiert diese Datei noch nicht, so wird sie angelegt, andernfalls auf Länge 0 verkürzt.

Programm 16.3: Öffnen einer Datei – mittels open() (*open.c*)

```
 1  #include <fcntl.h>
 2  #include <unistd.h>
 3  #include <stdlib.h>
 4  #include <stdio.h>
 5
 6  int main() {
 7      int fd;
 8
 9      if ((fd = open(”myfile.txt”, O_RDWR | O_CREAT | O_TRUNC, 0664)) < 0)
10          perror(”open”), exit(1);
11
12      /* Arbeiten mit der Dateiverbindung fd ... */
13
14      close(fd);
15
16      return 0;
17  }
```

Anmerkungen: Die *Zugriffsrechte* werden am einfachsten als *Oktalzahl* (wie hier geschehen) angegeben. Hier wurden die Zugriffsrechte `rw-rw-r--` gesetzt, die aber noch durch die sogenannte *umask* modifiziert werden. Die Zugriffsrechte, die bei open gesetzt werden sollen, werden mit dem *Komplement der umask mit bitweisem Und verknüpft*. (**Beispiel:** In vorigem Programm sollten die Zugriffsrechte 0664 gesetzt werden; 0664 & ~*umask* ist 0664 & ~022 = 0644 im Fall, dass *umask* = 022 gesetzt ist, was bewirkt, dass die Schreibrechte für die Gruppe und Andere entfernt werden). Dabei können höchstens Rechte weggenommen werden, aber keine dazu kommen. Die *umask* (*file mode creation mask*) kann mit dem Systemaufruf *umask()* und dem Kommando *umask* gesetzt werden (siehe Manpages).

16.3.2 Schließen von Dateiverbindungen – close()

int close(int fd);

Beendet die I/O-Verbindung, die durch den übergebenen Filedeskriptor repräsentiert wird, sofern kein weiterer Filedeskriptor für dieselbe I/O-Verbindung existiert. Im Fehlerfall liefert *close()* einen negativen Wert.

Da der Kernel die Zahl der I/O-Verbindungen pro Prozess limitiert ist es ratsam, nicht mehr gebrauchte I/O-Verbindungen mit *close()* zu schließen.

Terminiert ein Prozess, so führt der Kernel implizit einen *close()* Systemaufruf für alle noch offenen I/O-Verbindungen aus.

16.3.3 Duplizieren von Filedeskriptoren – dup(), dup2()

int dup(int fd);
int dup2(int oldfd, int newfd);

Der Systemaufruf *dup()* erhält als Argument einen Filedeskriptor und liefert als Rückgabewert einen neuen Filedeskriptor als Duplikat zurück; dieser ist der kleinste freie Filedeskriptor! Das Duplikat bezeichnet exakt dieselbe I/O-Verbindung. Im Fehlerfall liefert *dup()* einen negativen Wert. Der Systemaufruf *dup2()* kopiert den Filedeskriptor *oldfd* nach *newfd* und schließt ggf. vorher den Filedeskriptor *newfd*, falls er eine I/O-Verbindung repräsentierte. *dup2()* liefert ebenfalls den neuen Filedeskriptor und im Fehlerfall −1.

Das folgende Programm lenkt mit Hilfe von *dup()* die Standardausgabe (*stdout*) in eine Datei namens „stdout.txt" um.

Programm 16.4: Umlenkung der Standardausgabe – mittels dup() (*dup.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5
6  int main() {
7      int fd;
8
9      /* Datei, auf die die Standardausgabe umgelenkt werden soll, oeffnen */
10     if ((fd = open("stdout.txt", O_WRONLY | O_CREAT | O_TRUNC, 0664)) < 0)
11         perror("open"), exit(1);
12
13     close(1); /* stdout schliessen */
14
15     dup(fd); /* fd duplizieren => 1 ist der kleinste freie! */
16     close(fd); /* fd schliessen, da doppelt */
17
18     /* Ausgabe auf stdout => geht in die Datei stdout.txt! */
19     puts("Hallo␣Welt!");
20
21     return 0;
22 }
```

Erläuterung: Zunächst wird eine neue Dateiverbindung eröffnet (zum Schreiben). Danach wird Filedeskriptor 1 (*stdout*) geschlossen und dies ist jetzt der kleinste freie. Somit liefert der folgende *dup()*-Aufruf als Ergebnis den Filedeskriptor 1. Alle Schreiboperationen nach *stdout* erfolgen danach in die Datei mit Namen „stdout.txt". Der Filedeskriptor fd ist nun ein unnötiges Duplikat und kann nun geschlossen werden. Es gibt ja immer noch einen Filedeskriptor für diese Dateiverbindung, nämlich den Filedeskriptor 1. (Die Aufgabe ließe sich auch kürzer mit einem *close(1)* gefolgt von *open()* lösen. ;-)

16.3.4 Informationen über Dateien und I/O-Verbindungen – stat(), etc.

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

Der Systemaufruf *stat()* – der in Programm 16.1 auf Seite 188 bereits verwendet und zuvor besprochen wurde – liefert Informationen zu einer Datei in dem übergebenen Puffer `buf`. Der Rückgabewert ist im Fehlerfall negativ.

Die beiden Systemaufrufe *stat()* und *lstat()* sind nahezu identisch. Sie verhalten sich bei *symbolischen Links* jedoch unterschiedlich. Mit *stat()* erhält man bei einem symbolischen Links über die Datei, auf die der Link „zeigt", wohingegen man von *lstat()* Informationen über den symbolischen Link selbst erhält.

Mit dem Systemaufruf *fstat()*, der als erstes Argument nicht einen Dateinamen, sondern einen Filedeskriptor erwartet, kann man Informationen über eine offen Dateiverbindung erhalten.

Das folgende Programm ermittelt – mit Hilfe des Systemaufrufs *fstat()* – den Typ der Datei, die hinter der Standardeingabe steckt:

Programm 16.5: Typ der Standardeingabe ermitteln – mittels fstat() (*fstat.c*)

```
1  #include <sys/stat.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  int main() {
7     char *s;
8     struct stat statbuf;
9
10    if (fstat(0, &statbuf) < 0)
11       perror("fstat"), exit(1);
12
13    switch (statbuf.st_mode & S_IFMT) {
14       case S_IFREG : s = "normale Datei"; break;
15       case S_IFDIR : s = "Verzeichnis"; break;
16       case S_IFCHR : s = "zeichenorient. Geraet"; break;
17       case S_IFBLK : s = "blockorient. Geraet"; break;
18       case S_IFLNK : s = "symbolischer Link"; break;
19       case S_IFIFO : s = "FIFO"; break;
20       case S_IFSOCK : s = "Socket"; break;
21       default: s = "_unbekannt_"; break;
22    }
23    printf("Datei-Typ: %s\n", s);
24
25    return 0;
26 }
```

```
thales$ gcc -Wall fstat.c
thales$ a.out
Datei-Typ: zeichenorient. Geraet
thales$ a.out < stat.c
Datei-Typ: normale Datei
thales$
```

16.3.5 Zugriff auf Verzeichnisse – readdir(), etc.

DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);

Mit den Systemaufrufen *opendir()*, *readdir()* und *closedir()* können die Einträge eines Verzeichnisses gelesen werden. Dies wurde in Abschnitt 16.2.4.3 auf Seite 189 bereits erläutert und mit dem Programm 16.2 demonstriert. Im Fehlerfall liefern diese Systemaufrufe einen Null-Zeiger oder einen negativen Wert.

Jeder Verzeichniseintrag wird in einer Struktur vom Typ *struct dirent* geliefert:

```
struct  dirent {
    ino_t           d_ino;      /* Inode-Nummer */
    off_t           d_off;      /* (interne Bedeutung) */
    unsigned short  d_reclen;   /* Groesse des Eintrags */
    char            d_name[1];  /* Dateiname */
};
```

Diese Struktur enthält in der Komponente *d_ino* die *Inode-Nummer* und in der Komponente *d_name* den *Dateinamen*. Die Komponente *d_reclen* enthält die Größe der Struktur (die nicht fest ist, bedingt durch die variable Größe von *d_name*). Die Komponente *d_off* ist nur von interner Bedeutung für das Dateisystem.

16.3.6 Schreiben in I/O-Verbindungen – write()

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

write() schreibt aus dem Hauptspeicher in eine I/O-Verbindung. *nbyte* bestimmt die Datenmenge (Anzahl der zu schreibenden Bytes), *buf* ist die Startadresse im Hauptspeicher und der Filedeskriptor *fd* identifiziert die I/O-Verbindung.

Eine *write()*-Operation ist ein 1:1 Kopiervorgang. Es findet keinerlei Daten-Konvertierung oder Veränderung statt. Führt die I/O-Verbindung zu einer gewöhnlichen Datei (regular file), so bestimmt der sog. *File Offset Pointer* im Open-File-Table-Slot (OFT – siehe weiter hinten) das Ziel des Kopiervorgangs. Die übertragenen Bytes überschreiben bereits vorher gespeicherte Daten. Anschließend zeigt der File Offset Pointer hinter das letzte übertragene Byte.

Der Rückgabewert von *write()* ist die Anzahl der geschriebenen Bytes bzw. −1 im Fehlerfall.

Achtung: Die Modellannahme, dass die Daten nach Abschluß des *write()* Systemaufrufs physikalisch auf die Platte geschrieben wurden, ist nicht vollständig korrekt. Das I/O-Subsystem kopiert die Daten in den Puffer-Cache und gibt die Kontrolle zurück, etwa mit folgender Meldung:

> *"'I've taken note of your request, and rest assured that your file deskriptor is OK, I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll put your data on the disk where it belongs. If I discover an error then I'll try to print something on the console, but I won't tell you about it (indeed, you may have terminated by then). If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request. You may ask no further questions. Trust me. And thank me for the speedy reply."'* [Marc J. Rochkind, Seite 29]

Mit dem Flag *O_SYNC* könnte man dies verhindern. Ein deutlicher Performanceverlust wäre jedoch die Folge.

Folgendes Programm verwendet *write()*, um einen String (und einen abschließenden Zeilenumbruch) auf die Standardausgabe (Filedeskriptor 1) auszugeben:

Programm 16.6: Ausgabe eines Strings mit Hilfe von write() (*write.c*)

```
1  #include <string.h>
2  #include <unistd.h>
3
4  int main() {
5      char *s = "Hallo Welt!";
6      int len = strlen(s);
7
8      /* Ausgabe von s mit abschliessendem Newline aus stdout */
9      write(1, s, len);
10     write(1, "\n", 1);
11
12     return 0;
13 }
```

Da *write()* unter Umständen auch weniger Bytes schreiben kann, als gefordert ist, führt folgendes Programm solange *write()* Systemaufrufe aus, bis alle Daten geschrieben sind.

Programm 16.7: Ausgabe eines Strings mit Hilfe von write() (*write1.c*)

```
1  #include <string.h>
2  #include <unistd.h>
3
4  int dowrite(int fd, const void *buf, size_t nbyte) {
5      int ret, n = nbyte;
6      const char *cbuf = buf;
7      /* solange write() aufrufen, bis alle Daten geschrieben sind */
8      do {
9          ret = write(fd, cbuf, nbyte);
10         cbuf += ret; /* Adresse erhoehen um Anzahl der geschr. Bytes */
11         nbyte -= ret; /* und Anzahl zu schreibender Bytes entspr. verringern */
12     } while (ret > 0 && nbyte > 0);
13     return ret < 0 ? ret : n;
14 }
15
16 int main() {
17     char *s = "Hallo Welt!";
18     int len = strlen(s);
19
20     /* Ausgabe von s mit abschliessendem Newline aus stdout */
21     dowrite(1, s, len);
22     dowrite(1, "\n", 1);
23
24     return 0;
25 }
```

16.3.7 Lesen aus I/O-Verbindungen – read()

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

read() liest (maximal) *nbyte* viele Bytes von der I/O-Verbindung, die der Filedeskriptor *fd* bezeichnet, in den Puffer mit der Startadresse *buf*. Die Datenübertragung beginnt mit dem Byte, auf das der *File Offset Pointer* im OFT-Slot zeigt. Nach Abschluß des *read()* Systemaufrufs zeigt der *File Offset Pointer* auf das Byte, welches auf das letzte übertragene Byte folgt.

Der Rückgabewert 0 zeigt *EOF* (*End Of File*) und −1 einen Fehler an. Andernfalls gibt *read()* die Anzahl der gelesenen Bytes (nicht mehr als *nbyte*) zurück.

Folgendes Programm verwendet *read()*, um (wie *fgets()*) eine Zeile (aber maximal *buflen*−1 Zeichen) von der Standardeingabe zu lesen (*readline()* ist ein *fgets()*-Variante für Filedeskriptoren):

Programm 16.8: Einlesen einer Zeile von der Standardeingabe – mittels read() (*read.c*)

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4
 5 #define BUFLEN 128
 6
 7 /* liest eine Zeile von der I/O−Verbindung, die fd bezeichnet,
 8  * aber maximal buflen−1 viele Zeichen
 9  */
10 int readline(int fd, char *buf, int buflen) {
11     int res, len;
12
13     /* eine Zeile (aber max. BUFLEN−1 Zeichen) von stdin lesen */
14     for (len = 0; len < buflen−1; len++) {
15         /* ein Zeichen von stdin in den Puffer lesen */
16         if ((res = read(fd, buf+len, 1)) < 0)
17             return −1;
18         /* bei EOF oder '\n' ist Schluss mit dem Einlesen */
19         if (res == 0 || buf[len] == '\n')
20             break;
21     }
22
23     /* String noch sauber mit einem Null−Byte terminieren */
24     buf[len] = '\0';
25
26     return len;
27 }
28
29 int main() {
30     char buf[BUFLEN];
31
32     if (readline(0, buf, BUFLEN) < 0)
33         perror("readline"), exit(1);
34
35     /* ... und auf stdout ausgeben */
36     puts(buf);
37
38     return 0;
39 }
```

Besonderheit: Nicht-blockierendes Lesen (non-blocking read)

Durch die Abstraktion aller Datenquellen/-ziele zu Dateien kann mit *read()* nicht nur von Plattendateien, sondern auch von Terminals, Pipes und Stream-Devices gelesen werden.

Wenn über eine derartige I/O-Verbindung gerade keine Daten zur Verfügung stehen und ein Prozess führt darauf einen *read()* Systemaufruf aus, dann blockiert der Prozess bis Daten eintreffen. Wurde jedoch beim *open()* oder durch *fcntl()* das Flag *O_NDELAY / O_NONBLOCK* für diese I/O-Verbindung gesetzt, so kehrt *read()* ohne zu blockieren zurück. Bei Terminals und Pipes produziert *read()* als Rückgabewert 0, was ein Entdecken von **EOF** unmöglich macht. Dieses Problem vermeidet *read()* bei den neueren Stream-Devices, hier kommt der Wert −1 zurück und *errno* enthält den Wert *EAGAIN*.

16.3.8 Fehlerbehandlung bei Systemaufrufen – perror()

Jeder Systemaufruf zeigt mit seinem *Rückgabewert* an, ob die Ausführung erfolgreich war oder ob eine Fehlersituation aufgetreten ist. Normalerweise liefert ein Systemaufruf im Fehlerfall entweder einen *negativen Wert (oder sogar genau* −1*)* oder einen *Null-Zeiger* – abhängig vom Typ des Rückgabewertes. Ein stabil kodiertes Programm überprüft den Rückgabewert bei jedem Systemaufruf.

Programm 16.9: Fehlerbehandlung bei Systemaufrufen – Die Erste (*error1.c*)

```
 1  #include <stdio.h> /* wg. perror() */
 2  #include <fcntl.h>
 3  #include <unistd.h>
 4  #include <stdlib.h>
 5
 6  int main() {
 7      int fd;
 8      if ((fd = open("file", O_RDWR | O_CREAT | O_EXCL | O_TRUNC, 0600)) < 0)
 9          perror("open"), exit(1);
10      close(fd);
11      return 0;
12  }
```

```
chomsky$ gcc -Wall error1.c
chomsky$ ls
a.out  error1.c
chomsky$ a.out
chomsky$ ls
a.out  error1.c  file
chomsky$ a.out
open: File exists
chomsky$
```

Wenn der System Call *open()* (exklusives Erzeugen einer Datei, siehe *O_EXCL*) bei der Ausführung im Kernel Mode in irgendwelche Probleme läuft, liefert er *-1* als Ergebnis. Dies tritt z. B. auf, wenn die Datei bereits existiert.

Fragen:

Auf welches Problem traf *open()*?

Woher weiß *perror()*, welches Problem aufgetreten ist?

Antwort: *errno*

Tritt bei der Ausführung eines Systemaufrufs ein Fehler auf, liefert der Aufruf einen eigentlich unmöglichen Wert zurück. Dies ist meist *-1* oder ein *Null-Zeiger*. Die Details beschreibt die Definition der einzelnen Systemaufrufe im Kapitel 2 der *Reference Manuals* (siehe auch *man -s 2 intro* ↪ *Introduction to system calls and error numbers*).

Neben diesem Fehlerindikator macht der Kernel in der globalen Variable *errno* eine Fehlernummer verfügbar. Mit Hilfe dieser Fehlernummer kann *perror()* bzw. *strerror()* eine entsprechende Fehlermeldung ausgeben bzw. erzeugen.

Programm 16.10: Fehlerbehandlung bei Systemaufrufen – Die Zweite (*error2.c*)

```
 1 #include <stdio.h>
 2 #include <fcntl.h>
 3 #include <unistd.h>
 4 #include <stdlib.h>
 5 #include <errno.h> /* wg. errno */
 6 #include <string.h> /* wg. strerr() */
 7
 8 int main() {
 9     int fd;
10     if ((fd = open("file", O_RDWR | O_CREAT | O_EXCL | O_TRUNC, 0600)) < 0) {
11         perror("open");
12         printf("errno = %d\n", errno);
13         printf("Fehler = %s\n", strerror(errno));
14         exit(1);
15     }
16     close(fd);
17     return 0;
18 }
```

```
chomsky$ gcc -Wall error2.c
chomsky$ ls
a.out  error2.c
chomsky$ a.out
chomsky$ ls
a.out  error2.c  file
chomsky$ a.out
open: File exists
errno = 17
Fehler = File exists
chomsky$
```

Die Datei */usr/include/errno.h* definiert symbolische Konstanten (Makros) für alle vom Kernel benutzten Fehlernummern.

errno	Makro	Fehlermeldung
1	EPERM	Operation not permitted
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
	…	
19	ENODEV	No such device
20	ENOTDIR	Not a directory
	…	
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
	…	
64	ENONET	Machine is not on the network
65	ENOPKG	Package not installed
	…	

Anwendung von errno:
Wird ein "langsamer" Systemaufruf (z. B. *read()*) durch ein Interrupt-Signal unterbrochen, so liefert er -1 als Fehlerindikator zurück, setzt aber *errno* auf den Wert *EINTR*. Dies bedeutet, dass eigentlich *kein Fehler* aufgetreten ist, sondern der Systemaufruf nur „gestört“ worden war. Das Programm kann nun entscheiden, ob es den Systemaufruf erneut aufsetzt oder auf die Unterbrechung hin anders weiterarbeitet.

```
/* ... */

do {
    nread = read(fd, buf, BUFSIZE);
} while (nread < 0 && errno == EINTR); /* ... nur unterbrochen? */

if (nread < 0)
    perror("read"), exit(1); /* ... also echter Fehler */

/* ... */
```

16.4 Datenstrukturen für I/O-Verbindungen

16.4.1 UFDT, OFT und KIT

Der Kernel verwaltet in seinem Bereich des Hauptspeichers eine Reihe von Datenstrukturen, die die Systemaufrufe beim Arbeiten mit Dateien benutzen (siehe Abbildung 16.6).

- **UFDT: User File Descriptor Table**
  Die *UFDT* (*User File Descriptor Table*) identifiziert alle offenen I/O-Verbindungen eines Prozesses. Der Kernel legt diese Tabelle für jeden Prozeß (aber im Kernel-Adressraum) an. Sie gehört zum Kontext des Prozesses. Jeder *Filedeskriptor* eines Prozesses ist ein Index in seine UFDT. Als einzige Komponente enthalten die Slots der UFDT einen Zeiger in die *OFT* (*Open File Table*) des Kernels.

- **OFT: Open File Table**
  Die *OFT* (*Open File Table*) existiert nur einmal im Adressraum des Kernels. Alle Slots der *OFT* enthalten

  – einen Zeiger in die *KIT* (*Kernel Inode Tabelle*),

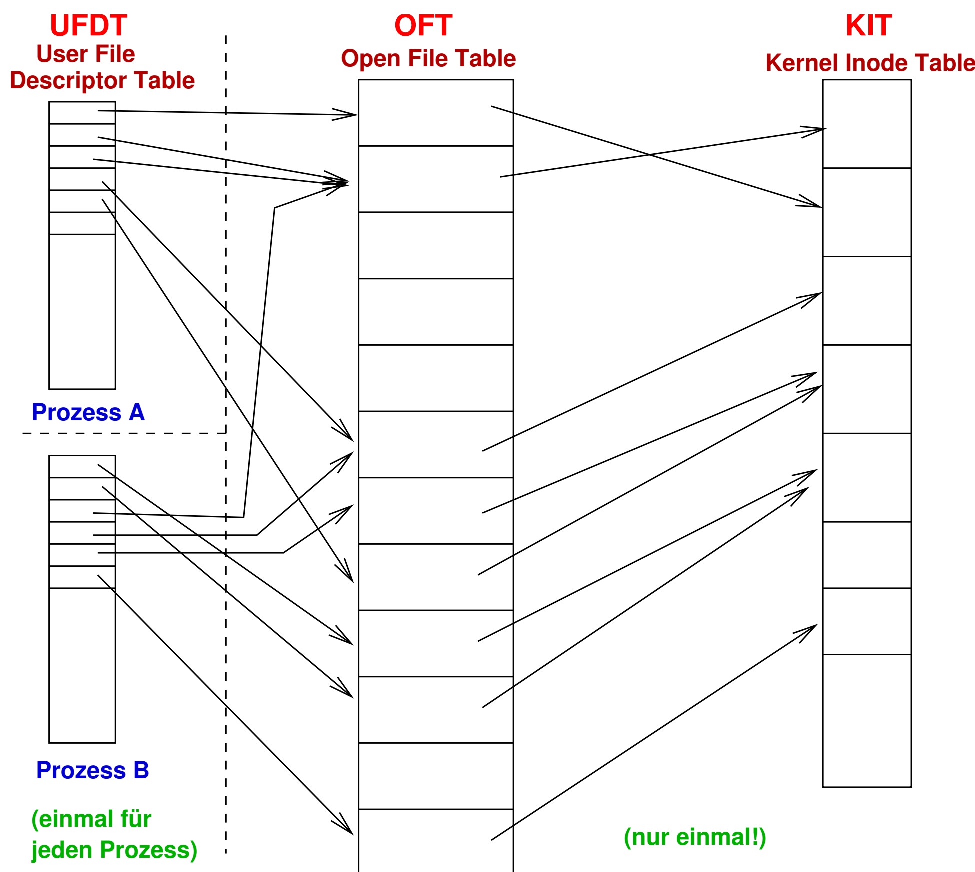

Abbildung 16.6: Datenstrukturen für I/O-Verbindungen

- die *oflags*,
- einen *File Offset Pointer*
- und einen Referenz-Zähler, der angibt, wieviele UFDT-Slots auf diesen OFT-Slot verweisen.

Die *oflags* stammen aus dem *open()*-Aufruf (bzw. aus dem letzten *fcntl()*-Aufruf). Der *File Offset Pointer* legt fest, bei welchem Byte die nächste *read()*- oder *write()*-Operation stattfindet.

- **KIT: Kernel Inode Table**
  Die *KIT* (*Kernel Inode Table*) existiert nur einmal im System. Der Kernel koordiniert darüber sämtliche Zugriffe auf Dateien. Die *in-core* Version der Inodes enthält alle Komponenten der Plattenversion plus Informationen, die nur bei offenen I/O-Verbindungen von Bedeutung sind: ein Referenz-Zähler (der angibt wieviele OFT-Slots auf diesen KIT-Slot verweisen), die Gerätenummer, …
  Jede offene Datei belegt genau einen Slot, unabhängig wie oft und von wievielen Prozessen sie geöffnet wurde.

16.4.2 Interne Abläufe bei den Systemaufrufen

Um die Integrität der I/O-Datenstrukturen zu gewährleisten, behält der Kernel sämtliche Tabellen in seinem Adreßraum. Prozesse können deshalb nur durch Systemaufrufe auf

diese Tabellen einwirken.

16.4.2.1 Systemaufruf open()

open() belegt immer je einen Slot in der UFDT und der OFT. Der Referenz-Zähler im OFT-Slot wird mit 1 initialisiert. Bestand keine Verbindung zu der Datei, belegt *open()* auch einen Slot in der KIT und initialisiert diese neue in-core Inode (und deren Referenz-Zähler auf 1). Besteht bereits eine Verbindung zu der Datei (vom selbem oder einem anderen Prozess), wird der schon für die Datei reservierte KIT-Slot verwendet. Der Kernel installiert nur den Zeiger vom OFT-Slot auf den KIT-Slot und inkrementiert den Referenz-Zähler in dieser in-core Inode.

Folge: Mehrere OFT-Slots können auf den selben KIT- Slot zeigen. Der Kernel hält von jeder Platten-Inode maximal eine Kopie „*in-core*“.

16.4.2.2 Systemaufruf close()

close() gibt einen Slot in der UFDT frei und dekrementiert den Referenz-Zähler im OFT-Slot. Wird dieser Referenz-Zähler dadurch zu 0, kann auch der OFT-Slot freigegeben und der Referenz-Zähler im KIT-Slot dekrementiert werden. Wird nun dieser Referenz-Zähler (in der KIT) zu 0, so gibt der Kernel auch den KIT-Slot frei. Sollte in der Inode der *Link Count* 0 sein, so kann der Kernel jetzt auch die Platten-Inode und den Plattenplatz freigeben.

16.4.2.3 Systemaufruf dup()

Der Systemaufruf *dup()* (siehe auch *dup2()*) belegt den niedersten, noch freien Slot in der UFDT, kopiert den Inhalt eines bereits belegten Slots dorthin und inkrementiert den Referenz-Zähler im OFT-Slot, auf den nun beide UFDT-Slots zeigen.

Folge: Mehrere UFDT-Slots eines Prozesses können auf den selben OFT-Slot zeigen.

16.4.2.4 Systemaufruf fork()

Der Systemaufruf *fork()* erzeugt einen neuen Prozess, indem er den kompletten Kontext des ausführenden Prozesses verdoppelt. Dabei muss der Kernel u. A. sowohl die UFDT kopieren als auch die Referenz-Zähler in den entsprechenden OFT-Slots inkrementieren.

Folge: Mehrere UFDT-Slots aus verschiedenen aber verwandten Prozessen können auf den selben OFT-Slot zeigen (Vererbung). Diese Prozesse können sich dadurch einen File Offset Pointer teilen.

16.4.2.5 Beispiel

Im Folgenden soll die Belegung der **Kerneltabellen** über eine Folge von Systemaufrufen gezeigt werden. *Der Ausgangszustand ist in Abbildung 16.7 dargestellt.*

Vater:

fd1 = *open*(”file1”, *O_RDONLY*);
fd2 = *open*(”file2”, *O_RDWR*);

Dabei wird jeweils ein UFDT-Slot und ein OFT-Slot angelegt und der Referenz-Zähler im OFT-Slot auf 1 gesetzt. Da beide Dateien noch nicht offen waren wir auch je ein Eintrag in die KIT hinzugefügt (mit Referenz-Zähler 1). *Danach sieht die Situation wie in Abbildung 16.8.*

Vater:

fork();

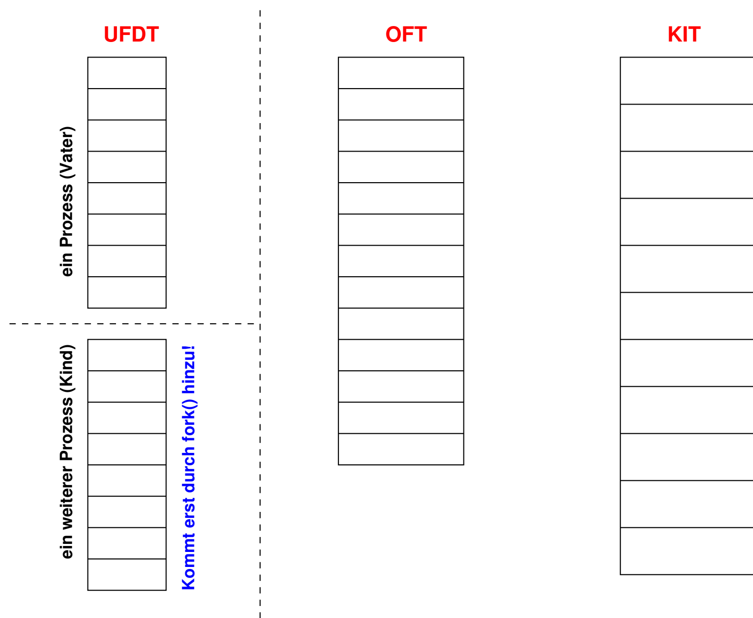

Abbildung 16.7: Kerneltabellen – Ausgangszustand

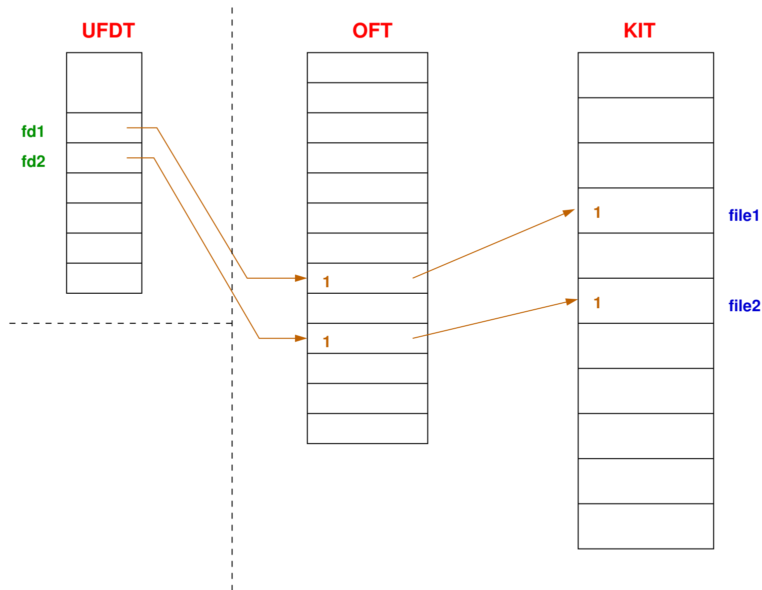

Abbildung 16.8: Kerneltabellen (1)

Durch den Systemaufruf *fork()* wird der Kontext des Prozesses dupliziert und es entsteht so ein weiterer Prozess. Dessen UFDT-Einträge verweisen auf dieselben OFT-Slots, deren Referenzähler dabei dementsprechend erhöht wurden. *Danach sieht die Situation wie in Abbildung 16.9.*

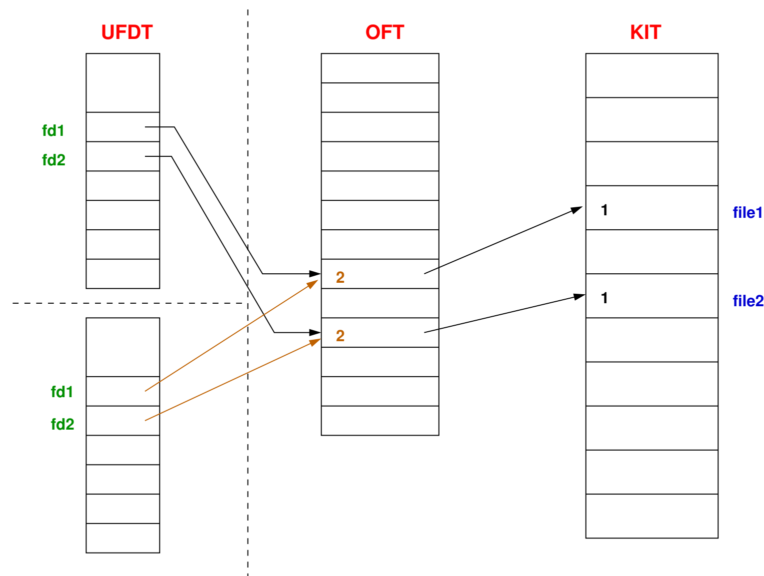

Abbildung 16.9: Kerneltabellen (2)

Vater:

$fd3 = open(''\text{file3}'', O_WRONLY);$

Kind:

$fd3 = open(''\text{file3}'', O_RDONLY);$

Durch jedes der beiden *open()* Systemaufrufe wird bei Vater und Kind je ein neuer UFDT- und eine neuer OFT-Slot allokiert und der Referenz-Zähler im OFT-Slot auf 1 gesetzt. Beim ersten *open()* wird die Inode in den Speicher geholt und von der OFT ein Verweis auf diesen KIT-Eintrag gesetzt – und schließlich der Referenz-Zähler im KIT-Eintrag auf 1 gesetzt. Beim zweiten *open()* wird nur ein weiterer Verweis auf denselben KIT-Slot erzeugt und der Referenz-Zähler im KIT-Slot inkrementiert. *Danach sieht die Situation wie in Abbildung 16.10.*

Vater:

$fd4 = open(''\text{file4}'', O_RDONLY);$

Kind:

$fd4 = open(''\text{file5}'', O_RDONLY);$

Danach sieht die Situation wie in Abbildung 16.11.

Kind:

$close(fd1);$

Vater:

$close(fd2);$

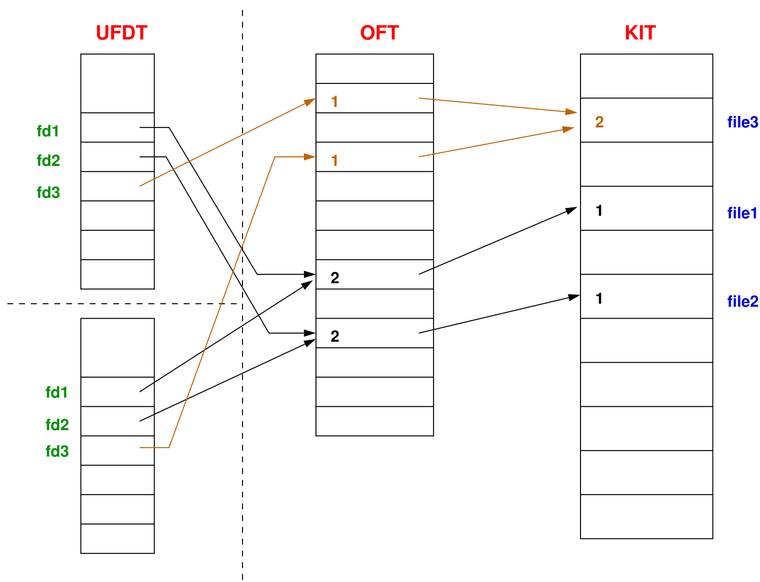

Abbildung 16.10: Kerneltabellen (3)

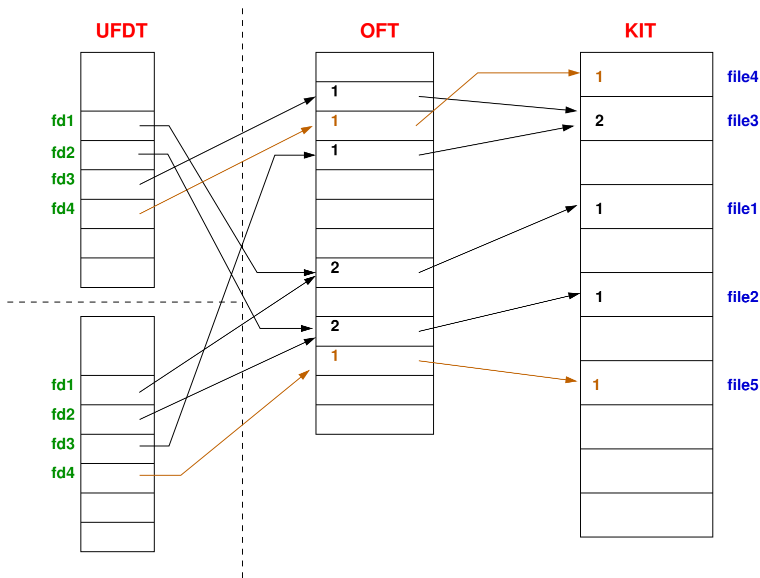

Abbildung 16.11: Kerneltabellen (4)

Nach jedem *close()* wird ein UFDT-Slot frei und der Referenz-Zähler im OFT-Slot dekrementiert. Da der Referenz-Zähler jeweils noch größer als 0 ist, bleibt der OFT-Slot bestehen. *Danach sieht die Situation wie in Abbildung 16.12.*

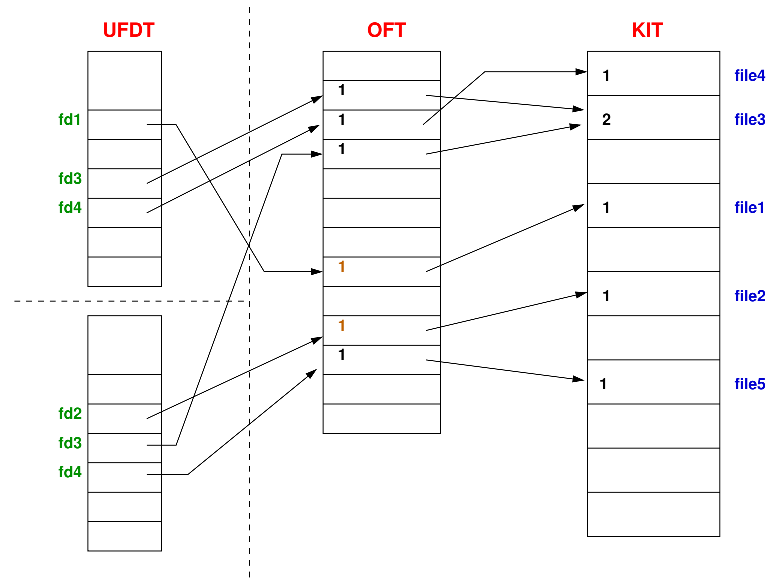

Abbildung 16.12: Kerneltabellen (5)

Kind:

fd5 = dup(fd4);

dup() liefert den kleinsten freien Filedeskriptor. Dieser ist ja fd1, den wir eben geschlossen haben. Der neue UFDT-Slot zeigt auf den selben OFT-Slot. Deshalb muss auch der Referenz-Zähler im OFT-Slot erhöht werden. *Danach sieht die Situation wie in Abbildung 16.13.*

Kind:

close(fd4);
close(fd5);

Beim ersten *close()* wird nur der UFDT-Slot freigegeben, da der Referenz-Zähler im OFT-Slot immer noch größer 0 (nämlich 1) ist. Nach dem zweiten *close()* ist der Referenz-Zähler im OFT-Slot gleich 0 und somit wird der OFT-Slot freigegeben und der Referenz-Zähler im KIT-Slot dekrementiert. Dieser Referenz-Zähler geht nun auch auf 0 und somit wird der KIT-Slot freigegeben ($\leadsto$ auf Platte, wenn Link Count größer 0 und sonst Datei freigeben).

Vater:

close(fd1);
close(fd4);

Nach jedem dieser *close()*-Aufrufe werden alle Slots freigegeben, da alle Referenz-Zähler auf 0 gehen. *Danach sieht die Situation wie in Abbildung 16.14.*

Kind:

close(fd2);
close(fd3);

Alle UFDT-, OFT-, und KIT-Slot werden nach dem ersten *close()* entfernt, da alle Referenz-Zähler auf 0 absinken. Bei dem zweiten *close()* fällt der UFDT-Slot weg und auch der OFT-Slot, da dieser Referenz-Zähler auf 0 absinkt. Der Referenz-Zähler im KIT-Eintrag ist jedoch noch 1, so dass der KIT-Eintrag bleibt.

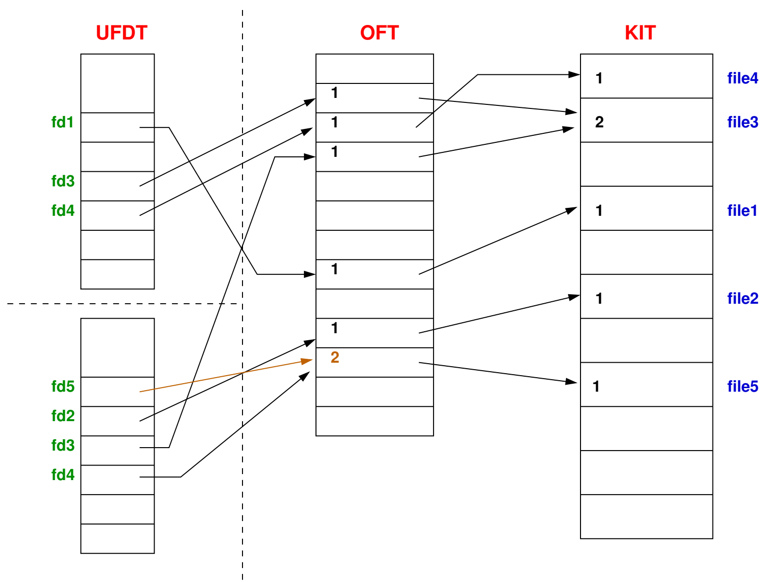

Abbildung 16.13: Kerneltabellen (6)

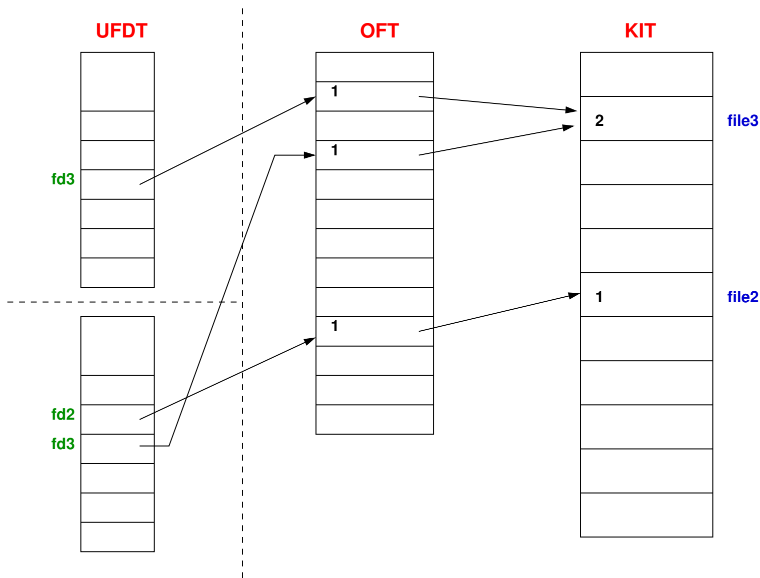

Abbildung 16.14: Kerneltabellen (7)

Vater:

close(*fd*3);

Bei diesem *close()* werden nun alle Slots freigegeben, da der Referenz-Zähler sowohl im OFT-Slot als auch im KIT-Slot auf 0 absinkt.

Jetzt sind alle Tabellen wieder leer (siehe Ausgangszustand in Abbildung 16.7).

16.5 Systemaufrufe für I/O-Verbindungen – Zweiter Teil

16.5.1 Positionieren in Dateien – lseek()

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

lseek() verändert den *File Offset Pointer* OFT-Slot. Diesen neuen Startpunkt für die nächste *read()*- oder *write()*-Operation bestimmt *lseek()* aus *offset* und *whence*:

whence	Neue Position
SEEK_SET	Dateianfang plus *offset*
SEEK_CUR	momentane Position plus *offset*
SEEK_END	Dateiende plus *offset*

offset kann auch eine negative Zahl sein. Die symbolischen Konstanten (Makros) für *whence* sind in der Header-Datei *unistd.h* definiert.

lseek() kann nicht auf alle I/O-Verbindungen angewendet werden. Bei einem Terminal oder einer Pipe hat der *File Offset Pointer* keine sinnvolle Bedeutung. *lseek()* signalisiert deshalb einen Fehler mit dem Rückgabewert −1. Im Erfolgsfall gibt *lseek()* den neuen Wert des *File Offset Pointer* zurück.

Das folgende Programm demonstriert die Verwendung von *lseek()*:

Programm 16.11: Positionieren in Dateien – mittels lseek() (*lseek.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5
6  int main() {
7      char buf[11];
8      int fd, pos;
9      if ((fd = open("file", O_RDWR | O_CREAT | O_TRUNC, 0664)) < 0)
10         perror("open"), exit(1);
11
12     pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
13     printf("Position: %d\n", pos);
14
15     if (write(fd, "0123456789\n", 11) != 11) /* String schreiben */
16         perror("write"), exit(2);
17
18     pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
19     printf("Position: %d\n", pos);
20
21     lseek(fd, 2, SEEK_SET);  /* 2 Bytes hinter den Anfang */
22
23     if (write(fd, "xx", 2) != 2)  /* String schreiben */
```

```
24          perror("write"), exit(2);
25
26      pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
27      printf("Position: %d\n", pos);
28
29      lseek(fd, -4, SEEK_END);  /* 4 Bytes vor das Ende */
30
31      if (write(fd, "aa", 2) != 2)  /* String schreiben */
32          perror("write"), exit(2);
33
34      pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
35      printf("Position: %d\n", pos);
36
37      lseek(fd, 0, SEEK_SET); /* an den Anfang positionieren */
38      read(fd, buf, 10);  /* 10 Zeichen lesen */
39      buf[10] = '\0';  /* String mit Null-Byte terminieren */
40      puts(buf);   /* ... und schliesslich ausgeben */
41
42      pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
43      printf("Position: %d\n", pos);
44
45      return 0;
46  }
```

```
chomsky$ gcc -Wall lseek.c
chomsky$ a.out
Position: 0
Position: 11
Position: 4
Position: 9
01xx456aa9
Position: 10
chomsky$ cat file
01xx456aa9
chomsky$
```

Implizites lseek:

Enthalten die *oflags* im OFT-Slot einer I/O-Verbindung *O_APPEND*, so verursacht jeder *write()* Systemaufruf vor dem Datentransfer eine implizite *lseek()*-Operation, die den *File Offset Pointer* zuverlässig ans Ende der Datei positioniert.

Zwei explizite Systemaufrufe, ein *lseek()*, gefolgt von einem *write()*, sind in einem Multiuser-/Multitasking-Betriebssystem wie UNIX nicht notwendig eine *atomare* Operation. Ein konkurrierender Prozess könnte seinen *write()* Systemaufruf zwischen dem *lseek()* und dem *write()* Systemaufruf des ersten Prozesses ausführen. Dadurch wäre der *File Offset Pointer* des ersten Prozesses nicht mehr auf das neue Ende der Datei positioniert und der erste Prozess würde die Daten des zweiten Prozesses überschreiben. Das *O_APPEND* Flag ermöglicht dagegen das sichere und geordnete Anfügen an eine Datei. Der implizite *lseek()* und der Systemaufruf *write()* laufen als eine atomare Operation ab.

16.5.2 Erzeugen von Links – link(), symlink()

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

Mit *link()* kann man einen Hardlink erzeugen – analog zum Kommando *ln*. Mit *symlink()* kann man – analog zum Kommando *ln* mit der Option *-s* – einen Softlink erzeugen. Der Rückgabewert ist jeweils 0 bei Erfolg und −1 bei Misserfolg.

Das Anlegen eines Links funktioniert nur dann, wenn noch keine Datei mit diesem Namen bereits existiert (⇝ zur Synchronisation geeignet, genauso wie *open()* mit *O_EXCL* – später mehr dazu).

16.5.3 Entfernen von Dateinamen – unlink()

int unlink(const char *pathname);

Der Systemcall *unlink()* entfernt den angegebenen Dateinamen. Konsequenterweise wird der Link Count in der Inode dekrementiert. Sinkt der Link Count dabei auf 0, so werden die Inode und die Datenblöcke sofort freigegeben, wenn die Inode nicht in der KIT ist. Andernfalls werden die Inode und die Datenblöcke dann freigegeben, wenn die Inode aus der KIT entfernt wird (nachdem der Referenz-Zähler auf 0 gesunken ist). Im Erfolgsfall liefert *unlink()* 0 und −1 bei einem Fehler.

Besonderheit:
Das I/O-Subsystem verzögert das Freigeben von Inode und Plattenplatz solange noch ein Prozess die Datei geöffnet hat. Da jedoch der Verzeichnis-Eintrag entfernt wurde, ist ein weiterer Zugriff auf die Datei, mit *open()* oder *stat()*, über den Namen nicht mehr möglich. Die Filedeskriptoren der Prozesse bilden die „letzte" Verbindung zu der Datei. Sie können für *read()*, *write()*, *fstat()*, ..., und *close()* benutzt werden. Nach dem letzten expliziten oder impliziten *close()* Systemaufruf mit einem Filedeskriptor, der noch auf die Datei verwiesen hat, gibt das I/O-Subsystem die Daten endgültig frei.

Dieses Verhalten des Kernels vereinfacht den Gebrauch (und das ordentliche Entfernen) von *temporären Dateien*. Das folgende Programm demonstriert das Arbeiten mit temporären Dateien:

Programm 16.12: Erzeugen einer temporären Datei (*temp.c*)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4  #include <stdlib.h>
5
6  #define TMPNAME "/tmp/jmayerXXXXXX" /* XXXXXX am Ende wg. mktemp! */
7  #define BUFSIZE 128
8
9  int main() {
10     char tmp[BUFSIZE];
11     int fd;
12
13     /* temp. Dateiname mit mktemp() erzeugen;
14      * "XXXXXX" muss am Ende des Templates stehen und wird ersetzt */
15     strcpy(tmp, TMPNAME);
16     if (!mktemp(tmp)) /* Parameter von mktemp() wird veraendert! */
17         perror("mktemp"), exit(1);
18
19     /* temporaere Datei oeffnen -- aber nur, wenn sie noch nicht existiert */
20     if ((fd = open(tmp, O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0600)) < 0)
21         perror("open"), exit(2);
22
23     /* danach Dateinamen loeschen -- I/O-Verb. zur Datei immer noch via fd! */
```

```
24      unlink(tmp);
25
26      /* Arbeiten mit der temp. Datei */
27
28      /* nun aber wirklich die Inode und den Plattenplatz freigeben */
29      close(fd);
30
31      return 0;
32  }
```

Zum Erzeugen des Dateinamens wird die Funktion *mktemp()* verwendet:

char *mktemp(char *template);

Diese Funktion erwartet, dass am Ende von *template* der String „XXXXXX" steht. Dieser wird ersetzt durch eine Zeichenfolge, so dass der Dateiname noch nicht existiert. Dabei wird der Parameter verändert! Der Rückgabewert ist ungleich dem Null-Zeiger, wenn alles ohne Schwierigkeiten verlief. Im Falle eines Fehlers liefert diese Funktion den Null-Zeiger.

Der Prozess kann die temporäre Datei benutzen, wenn er jedoch – egal aus welchem Grund – terminiert, gibt das I/O-Subsystem automatisch allen Plattenplatz frei, den die Datei belegt hat. Es entstehen keine unliebsamen „Überbleibsel".

Temporäre Dateien legt man (aus Performance-Gründen) immer im Katalog */tmp* an, da dieser lokal auf der Maschine ist und meist sogar im Hauptspeicher ($\leadsto$ *tmpfs*).

16.5.4 Ändern der oflags – fcntl()

int fcntl(int fd, int cmd, long arg);

fcntl() verändert die Eigenschaften von bereits offenen I/O-Verbindungen. Er manipuliert die *oflag*-Bits im OFT-Slot. Damit können nachträglich Flagbits wie *O_APPEND* oder *O_SYNC* für eine I/O-Verbindung gesetzt oder gelöscht werden.

Der Parameter *cmd* kann entweder den Wert *F_GETFL* oder den Wert *F_SETFL*. Im ersten Fall liefert *fcntl()* die oflags als Rückgabewert und ignoriert den dritten Parameter. Im zweiten Fall setzt *fcntl()* die oflags auf den in *arg* angegebenen Wert. Der Rückgabewert −1 signalisiert eine Fehlersituation.

Programm 16.13: oflags verändern – mittels fcntl() (*fcntl.c*)

```
 1  #include <stdio.h>
 2  #include <fcntl.h>
 3  #include <unistd.h>
 4
 5  void clear_append(int fd) {
 6      int oflags;
 7          /* oflags mittels fcntl() ermitteln */
 8      if ((oflags = fcntl(fd, F_GETFL, 0)) < 0)
 9          perror("fcntl"), exit(1);
10
11      /* Ist O_APPEND gesetzt? Wenn nein, dann fertig! */
12      if ((oflags & O_APPEND) != O_APPEND)
13          return;
14
15      oflags &= ~O_APPEND; /* O_APPEND loeschen */
16
17      /* veraenderte oflags setzen */
18      if (fcntl(fd, F_SETFL, oflags) < 0)
```

```
19             perror("fcntl"), exit(2);
20    }
21
22    int main() {
23        char buf[20];
24        int fd, n;
25
26        if ((fd = open("file", O_RDWR | O_CREAT | O_TRUNC | O_APPEND, 0644)) < 0)
27            perror("open"), exit(3);
28
29        write(fd, "aaaaa", 5); /* "aaaaa" in die Datei schreiben */
30
31        lseek(fd, 0, SEEK_SET);
32        write(fd, "bbbb", 4); /* haengt "bbbb" an -- wg. O_APPEND! */
33
34        clear_append(fd);  /* O_APPEND loeschen */
35
36        lseek(fd, 0, SEEK_SET);
37        write(fd, "ccc", 3); /* jetzt wird "ccc" tatsaechlich am Anfang geschr. */
38
39        lseek(fd, 0, SEEK_SET); /* vom Anfang an ... */
40        n = read(fd, buf, 19); /* auslesen ...*/
41        buf[n] = '\0';
42        puts(buf);   /* ... und ausgeben */
43
44        close(fd);   /* wuerde auch implizit passieren */
45
46        return 0;
47    }
```

```
chomsky$ gcc -Wall fcntl.c
chomsky$ a.out
cccaabbbb
chomsky$ cat file; echo
cccaabbbb
chomsky$
```

16.5.5 ioctl()

int ioctl(int fd, int cmd, long arg);

ioctl() verändert die Eigenschaften von offenen I/O-Verbindungen zu Geräten oder IPC-Kanälen. Ein Prozess kann damit direkt auf einen *Gerätetreiber* Einfluss nehmen. Deshalb sind auch die erlaubten Kommandos sehr abhängig von den einzelnen Gerätetreibern und davon, welche Treibermodule in den Kernel konfiguriert wurden (siehe dazu die entsprechenden Manpages!). Die Verwendung ist analog zu *fcntl()*.

Typischerweise will man bei Passworteingaben nicht haben, dass der eingegebene Text auch gleichzeitig ausgegeben wird. Die wir mit folgendem Programm, das *ECHO* aus- und wieder einschaltet, erreicht.

Programm 16.14: Echo aus- und anschalten – mittels ioctl() (*echo.c*)

```
 1 #include <unistd.h>
 2 #include <termio.h>
 3 #include <stdio.h>
 4
 5 void echo_off(int fd) {
 6     struct termio buf;
 7     /* Flags lesen ... */
 8     if (ioctl(fd, TCGETA, &buf) < 0)
 9         perror("ioctl"), exit(1);
10
11     /* ECHO nicht gesetzt? => nichts zu tun */
12     if ((buf.c_lflag & ECHO) != ECHO)
13         return;
14
15     buf.c_lflag &= ~ECHO; /* ECHO loeschen */
16
17     /* Flags setzen ... */
18     if (ioctl(fd, TCSETA, &buf) < 0)
19         perror("ioctl"), exit(2);
20 }
21
22 void echo_on(int fd) {
23     struct termio buf;
24     /* Flags lesen ... */
25     if (ioctl(fd, TCGETA, &buf) < 0)
26         perror("ioctl"), exit(1);
27
28     /* ECHO gesetzt? => nichts zu tun */
29     if ((buf.c_lflag & ECHO) == ECHO)
30         return;
31
32     buf.c_lflag |= ECHO; /* ECHO setzen */
33
34     /* Flags setzen ... */
35     if (ioctl(fd, TCSETA, &buf) < 0)
36         perror("ioctl"), exit(2);
37 }
38
39 int main() {
40     char buf[128];
41
42     printf("Name: ");
43     fgets(buf, 128, stdin);
44     buf[strlen(buf)-1] = '\0'; /* Newline entfernen */
45     printf("Ihr Name: %s\n", buf);
46
47     printf("Passwort: ");
48     echo_off(0);  /* Echo ausschalten */
49     fgets(buf, 128, stdin);
50     buf[strlen(buf)-1] = '\0'; /* Newline entfernen */
51     echo_on(0);   /* ... und Echo wieder einschalten */
52     puts("");
53     printf("Ihr Passwort: %s\n", buf);
```

```
54
55     return 0;
56 }
```

16.6 Synchronisation

16.6.1 Generelles

Problem: Nebenläufigkeit

Mehrere Prozesse arbeiten lesend und schreibend auf einer Datei. Dabei kann es durchaus passieren, dass ein Prozess einen Datensatz liest, der während dessen von einem anderen Prozess in Teilen verändert wird. Mit einem einfachen Beispiel soll das Problem verdeutlicht werden.

Beispiel:

Mehrere Prozesse benötigen einen fortlaufenden Zähler, z. B. um für Datensätze einen internen Schlüssel zu vergeben; dazu lesen sie die Zahl für den nächsten Schlüssel aus einer Datei und schreiben ihn inkrementiert wieder zurück. In der folgenden Implementierung wird zwischen dem Lesen der Zahl und dem Zurückschreiben eine zufällig gewählte Zeit gewartet. Dieses Warten (Suspendieren) des Prozesses kann mit *sleep()* realisiert werden; die Zufallszahl für die Anzahl der zu wartenden Sekunden kann mit dem Pseudo-Zufallszahlengenerator *random()*, dessen Startwert mit der Funktion *srandom()* gesetzt werden kann, bestimmt werden:

Programm 16.15: Inkrementieren ohne Synchronisation (*nolock.c*)

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <string.h>
7
8  #define MAXDELAY 5
9  #define MAXREPEAT 5
10 #define BUFSIZE 128
11 #define SEQFILE "seqno"
12
13 /* Zaehler lesen, delay Sekunden schlafen und dann erhoehen */
14 void incr(int delay) {
15     int fd, seqno, n;
16     char buf[BUFSIZE];
17
18     if ((fd = open(SEQFILE, O_RDWR)) < 0)
19         perror("open"), exit(2);
20
21         /* Dateiinhalt in den Puffer einlesen */
22     if ((n = read(fd, buf, BUFSIZE)) < 0)
23         perror("read"), exit(3);
24         /* ... und mit Null-Byte terminieren */
25     buf[n] = '\0';
26
27         /* Integer parsen */
```

```
28      if (sscanf(buf, "%d", &seqno) != 1)
29          fprintf(stderr, "sscanf␣error\n"), exit(4);
30
31      printf("gelesen:␣%d\n", seqno);
32
33      seqno++; /* Zaehler erhoehen */
34
35      sleep(delay); /* delay Sekunden schlafen */
36
37      sprintf(buf, "%03d\n", seqno); /* neuen Zaehler als String */
38      n = strlen(buf);
39
40      lseek(fd, 0, SEEK_SET);  /* an den Dateianfang gehen */
41
42      if (write(fd, buf, n) != n)  /* ... und den Zaehler schreiben */
43          perror("write"), exit(5);
44
45      close(fd);
46  }
47
48  int main(int argc, char **argv) {
49      int i, delay;
50
51      if (argc != 2) {
52          fprintf(stderr, "Usage:␣%s␣<seed>\n", argv[0]);
53          exit(1);
54      }
55
56      /* Pseudo−Zufallszahlengenerator initialisieren */
57      srandom(atoi(argv[1]));
58
59      /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
60      sleep(delay = random() % (MAXDELAY+1));
61
62      for (i = 1; i <= MAXREPEAT; i++) {
63          /* gemeinsamen Zaehler erhoehen */
64          incr(delay);
65          /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
66          sleep(delay = random() % (MAXDELAY+1));
67      }
68
69      return 0;
70  }
```

```
thales$ gcc -Wall -o nolock nolock.c
thales$ cat seqno
001
thales$ nolock 2 > out1 & nolock 5 > out2 &
[1] 14289
[2] 14290
thales$
[1]-  Done                    nolock 2 >out1
[2]+  Done                    nolock 5 >out2
thales$ cat out1
gelesen: 1
gelesen: 3
gelesen: 4
gelesen: 6
gelesen: 7
thales$ cat out2
gelesen: 1
gelesen: 2
gelesen: 4
gelesen: 5
gelesen: 6
thales$ cat seqno
008
thales$
```

Wie man hier leicht sieht, steht der gemeinsame Zähler nach der Ausführung nicht auf 11. Jeder der beiden Prozesse sollte aber um 5 weiterzählen. Folgendes Fehlerszenario ist hierbei (mehrmals) aufgetreten:

1. Prozess A liest den Zähler
2. Prozess B liest den Zähler
3. Prozess A schreibt den inkrementierten Zähler
4. Prozess B schreibt den inkrementierten Zähler

Dabei erhalten A und B denselben Zählerwert und B überschreibt die Änderung von A (hier nicht tragisch). Eigentlich müsste das Lesen des Zählers und das Schreiben des inkrementierten Zählerwertes „auf einmal“ erfolgen, so dass kein anderer Prozess in der Zeit etwas schreiben oder lesen kann. Prozess A müsste also die Datei vor dem Lesen für sich sperren können und würde dann erst nach dem Schreiben des inkrementierten Wertes diese Sperre wieder aufheben.

Lösung: Semaphoren

Eine Semaphore ist eine „Ampel“, die immer nur *einem* Prozess grün (Datei benutzen erlaubt) und allen anderen Prozessen „rot“ (Benutzung z. Zt. nicht erlaubt) signalisiert.

Alle Prozesse verpflichten sich, nur bei „grün“ bestimmten Code auszuführen, z. B. auf eine gemeinsame Datei zuzugreifen. Dadurch, dass immer nur ein Prozess „grün“ erhält (*mutual exclusion*), synchronisieren sich die Prozesse über die Semaphore.

Eine Semaphore bewacht die Anweisungen des sog. *kritischen Bereichs*. Bevor ein Prozess den kritischen Bereich betritt, muss er sich von der Semaphore „grün"' signalisieren lassen (warten, bis "'grün“ gesetzt ist). Bei Betreten des kritischen Bereichs wird die Semaphore auf „rot“ gesetzt, wenn der Prozess den kritischen Teil verläßt, teilt er dies der Semaphore mit (wieder auf „grün“ setzen). Anschließend kann die Semaphore einem anderen Prozess „grün“ signalisieren. Der Notation des Informatikers *Dijkstra* folgend nennt man diese Operationen *P(sema)* (Semaphore für sich reservieren – *protect*) und *V(sema)* (Semaphore freigeben – holl. *vrej*).

In einem Programm könnte dies etwa folgendermaßen aussehen:

```
/* ... */

P(sem); /* Semaphore reservieren */

    /*
     * kritischer Bereich
     */

V(sem); /* Semaphore freigeben */

/* ... */
```

Anmerkung: Ist die Semaphore auf „rot", so wartet der Prozess (Operation *P()*) so lange, bis sie (von wem auch immer) auf „grün" gesetzt wird.

Problem: Deadlock

- Zwei Prozesse arbeiten z. B. gleichzeitig auf zwei Dateien.
- Jeder hat eine Datei gesperrt und wartet darauf, dass der andere seine Datei freigibt, um weiterarbeiten zu können.

16.6.2 Synchronisation mit open() und O_EXCL

In folgendem Programm sind die beiden Funktionen *my_lock()* und *my_unlock()* neu hinzugekommen. Außerdem wurde um die Anweisungen in *incr()* eine *P()*- und eine *V()*-Operation eingebaut. Der Rest ist identisch mit vorigem Beispiel. (Die *P()*-Operation besteht aus einem *open()*-Aufruf mit *O_EXCL*. Die *V()*-Operation besteht aus einem *unlink()*-Aufruf.)

Programm 16.16: Inkrementieren mit Synchronisation (*lock.c*)

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <errno.h>
8
9  #define MAXDELAY 5
10 #define MAXREPEAT 5
11 #define BUFSIZE 128
12 #define SEQFILE "seqno"
13 #define SEMA "/tmp/mysem"
14 #define MAXTRIES 5
15 #define WAITTIME 1
16 #define P(sem) if (!my_lock(sem)) \
17                         fprintf(stderr, "my_lock failed\n"), exit(6);
18 #define V(sem) if (!my_unlock(sem)) \
19                         fprintf(stderr, "my_unlock failed\n"), exit(7);
20
21 /* Zugriff ueber die Semaphore sem (Datei!) synchronisieren
22  * Rueckgabewert: != 0 bei Erfolg
```

```
23    */
24  int my_lock(char *sem) {
25      int ret, tries = 0;
26          /* Datei exklusiv anlegen => falls sie existiert gibt's einen Fehler;
27           * das Ueberpruefen, ob die Datei existiert, und das Anlegen ist eine
28           * _atomare_ Operation bei open() mit O_EXCL! */
29      while ((ret = open(sem, O_RDWR | O_CREAT | O_EXCL, 0664)) < 0 && errno ==
            EEXIST) {
30          if (++tries > MAXTRIES) /* ... das ganze nur MAXTRIES Mal versuchen */
31              return 0;
32          sleep(WAITTIME); /* ... und dazwischen immer ein kleines Schlaefchen */
33      }
34      return ret >= 0;
35  }
36
37  /* Semaphore wieder freigeben (d.h. Datei loeschen)
38    * Rueckgabewert: != 0 bei Erfolg
39    */
40  int my_unlock(char *sem) {
41      return unlink(sem) == 0;
42  }
43
44  /* Zaehler lesen, delay Sekunden schlafen und dann erhoehen */
45  void incr(int delay) {
46      int fd, seqno, n;
47      char buf[BUFSIZE];
48
49      P(SEMA) /* kritischen Bereich beginnen; Lock anfordern */
50
51      puts("BEGIN␣kritischer␣Bereich");
52
53      if ((fd = open(SEQFILE, O_RDWR)) < 0)
54          perror("open"), exit(2);
55
56          /* Dateiinhalt in den Puffer einlesen */
57      if ((n = read(fd, buf, BUFSIZE)) < 0)
58          perror("read"), exit(3);
59          /* ... und mit Null-Byte terminieren */
60      buf[n] = '\0';
61
62          /* Integer parsen */
63      if (sscanf(buf, "%d", &seqno) != 1)
64          fprintf(stderr, "sscanf␣error\n"), exit(4);
65
66      printf("gelesen:␣%d\n", seqno);
67
68      seqno++; /* Zaehler erhoehen */
69
70      sleep(delay); /* delay Sekunden schlafen */
71
72      sprintf(buf, "%03d\n", seqno); /* neuen Zaehler als String */
73      n = strlen(buf);
74
```

```
75      lseek(fd, 0, SEEK_SET);  /* an den Dateianfang gehen */
76
77      if (write(fd, buf, n) != n)  /* ... und den Zaehler schreiben */
78          perror("write"), exit(5);
79
80      close(fd);
81
82      puts("END kritischer Bereich");
83
84      V(SEMA) /* kritischen Bereich verlassen; Lock wieder freigeben */
85  }
86
87  int main(int argc, char **argv) {
88      int i, delay;
89
90      if (argc != 2) {
91          fprintf(stderr, "Usage: %s <seed>\n", argv[0]);
92          exit(1);
93      }
94
95      /* Pseudo-Zufallszahlengenerator initialisieren */
96      srandom(atoi(argv[1]));
97
98      /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
99      sleep(delay = random() % (MAXDELAY+1));
100
101     for (i = 1; i <= MAXREPEAT; i++) {
102         /* gemeinsamen Zaehler erhoehen */
103         incr(delay);
104         /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
105         sleep(delay = random() % (MAXDELAY+1));
106     }
107
108     return 0;
109 }
```

```
thales$ gcc -Wall -o lock lock.c
thales$ cat seqno
001
thales$ lock 2 > out1 & lock 5 > out2 &
[1] 27992
[2] 27993
thales$
[1]-  Done                    lock 2 >out1
[2]+  Done                    lock 5 >out2
thales$ cat out1
BEGIN kritischer Bereich
gelesen: 1
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 4
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 6
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 8
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 10
END kritischer Bereich
thales$ cat out2
BEGIN kritischer Bereich
gelesen: 2
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 3
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 5
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 7
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 9
END kritischer Bereich
thales$ cat seqno
011
thales$
```

Anmerkung: Die zur Synchronisation verwendete Datei wird im Katalog */tmp* angelegt, um sicher zu gehen, dass sie lokal auf dem Rechner ist und kein NFS im Spiel ist.

16.6.3 Synchronisation mit lockf()

int lockf(int fd, int function, off_t size);

File Locking und *Record Locking* wurde erst relativ spät in UNIX eingeführt. Einige ältere System V Versionen erhielten einen System Call *lockf()*, neuere (aktuelle) Versionen erweitern die Funktionalität von *fcntl()* und stellen eine Bibliotheksfunktion *lockf()* zur Verfügung.

Im einfachsten Fall kann *lockf()* eine Datei komplett vor simultanem Zugriff schützen

und somit als Semaphore brauchbar machen. *lockf()* bietet aber zusätzlich die Möglichkeit, auf genau definierten Teilen einer Datei exklusiven Zugriff durchzusetzen.

lockf() unterstützt folgende Kommandos (Parameter *function*):

Kommando	*Bedeutung*
F_LOCK	Einen Bereich der Datei für exklusiven Gebrauch nur durch den ausführenden Prozess reservieren.
F_ULOCK	Vorher reservierten Bereich der Datei wieder freigeben.
F_TEST	Einen Bereich der Datei auf Reservierungen durch andere Prozesse überprüfen.
F_TLOCK	Falls in dem Bereich keine Reservierung durch einen anderen Prozess besteht, den Bereich für den ausführenden Prozess reservieren, sonst Rückkehr mit Fehleranzeige. Prozess blockiert nicht!

Der Bereich beginnt beim *File Offset Pointer* und hat die Länge *size*. Ein Wert von 0 steht für unendlich. In diesem Fall reicht der Bereich also immer bis zum Ende der Datei – egal wie groß die Datei wird.

Der Rückgabewert von *lockf()* ist 0 im Erfolgsfall und -1, falls ein Fehler aufgetreten ist. Die beiden Werte *EACCES* und *EAGAIN* signalisieren bei *F_TEST* bzw. *F_TLOCK*, dass eine Reservierung nicht möglich ist.

Ein Aufruf mit *F_LOCK* blockiert so lange, bis die Reservierung gesetzt werden kann. Um das Blockieren zu verhindern, könnte man eine Sequenz von *lockf()*-Aufrufen mit *F_TEST* und *F_LOCK* benutzen. Zwischen den beiden Aufrufen kann aber ein anderer Prozess erfolgreich *F_LOCK* anwenden ($\leadsto$ keine *atomare Operation*), worauf der erste Prozess doch blockiert. Abhilfe schafft hier *F_TLOCK*, was *F_TEST* und *F_LOCK* zu einer *atomaren, nicht blockierenden Operation* zusammenfasst.

Die *P()*-Operation besteht aus (evtl. wiederholten) *lockf()*-Aufrufen mit *F_TLOCK* als Kommando.

Die *V()*-Operation besteht aus einem *lockf()*-Aufruf mit *F_ULOCK* als Kommando.

Das folgende Programm demonstriert die Verwendung von *lockf()* an dem bekannten Beispiel:

Programm 16.17: Inkrementieren mit Synchronisation – mittels lockf() (*lockf.c*)

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <errno.h>
8
9  #define MAXDELAY 5
10 #define MAXREPEAT 5
11 #define BUFSIZE  128
12 #define SEQFILE  "seqno"
13 #define MAXTRIES 5
14 #define WAITTIME 1
15 #define P(sem)  if (!my_lock(sem)) \
16                        perror("my_lock"), exit(6);
17 #define V(sem)  if (!my_unlock(sem)) \
18                        perror("my_unlock"), exit(7);
```

```
19
20  /* Zugriff ueber die Semaphore sem (Filedeskriptor) synchronisieren
21   * Rueckgabewert: != 0 bei Erfolg
22   */
23  int my_lock(int sem) {
24      int ret, tries = 0;
25      /* zuerst auf den Anfang der Datei positionieren und dann mittels
26       * lockf() bis zum Dateiende alles sperren
27       */
28      while ((ret = lseek(sem, 0, SEEK_SET)) == 0  /* an den Anfang gehen ... */
29              && (ret = lockf(sem, F_TLOCK, 0)) < 0 /* ... und bis zum Ende sperren */
30              && (errno == EACCES || errno == EAGAIN)) {
31          if (++tries > MAXTRIES) /* ... das ganze nur MAXTRIES Mal versuchen */
32              return 0;
33          sleep(WAITTIME); /* ... und dazwischen immer ein kleines Schlaefchen */
34      }
35      return ret == 0;
36  }
37
38  /* Semaphore wieder freigeben (d.h. Datei loeschen)
39   * Rueckgabewert: != 0 bei Erfolg
40   */
41  int my_unlock(int sem) {
42      return lseek(sem, 0, SEEK_SET) == 0  /* an den Anfang gehen ... */
43              && lockf(sem, F_ULOCK, 0) == 0; /* ... und bis zum Ende freigeben */
44  }
45
46  /* Zaehler lesen, delay Sekunden schlafen und dann erhoehen */
47  void incr(int delay) {
48      int fd, seqno, n;
49      char buf[BUFSIZE];
50
51      if ((fd = open(SEQFILE, O_RDWR)) < 0)
52          perror("open"), exit(2);
53
54      P(fd) /* kritischen Bereich beginnen; Lock anfordern */
55
56      puts("BEGIN␣kritischer␣Bereich");
57
58          /* Dateiinhalt in den Puffer einlesen */
59      if ((n = read(fd, buf, BUFSIZE)) < 0)
60          perror("read"), exit(3);
61          /* ... und mit Null-Byte terminieren */
62      buf[n] = '\0';
63
64          /* Integer parsen */
65      if (sscanf(buf, "%d", &seqno) != 1)
66          fprintf(stderr, "sscanf␣error\n"), exit(4);
67
68      printf("gelesen:␣%d\n", seqno);
69
70      seqno++; /* Zaehler erhoehen */
71
```

```
72        sleep(delay); /* delay Sekunden schlafen */
73
74        sprintf(buf, "%03d\n", seqno); /* neuen Zaehler als String */
75        n = strlen(buf);
76
77        lseek(fd, 0, SEEK_SET);  /* an den Dateianfang gehen */
78
79        if (write(fd, buf, n) != n)  /* ... und den Zaehler schreiben */
80              perror("write"), exit(5);
81
82        puts("END kritischer Bereich");
83
84        V(fd) /* kritischen Bereich verlassen; Lock wieder freigeben */
85
86        close(fd);
87  }
88
89  int main(int argc, char **argv) {
90        int i, delay;
91
92        if (argc != 2) {
93              fprintf(stderr, "Usage: %s <seed>\n", argv[0]);
94              exit(1);
95        }
96
97        /* Pseudo-Zufallszahlengenerator initialisieren */
98        srandom(atoi(argv[1]));
99
100       /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
101       sleep(delay = random() % (MAXDELAY+1));
102
103       for (i = 1; i <= MAXREPEAT; i++) {
104             /* gemeinsamen Zaehler erhoehen */
105             incr(delay);
106             /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
107             sleep(delay = random() % (MAXDELAY+1));
108       }
109
110       return 0;
111 }
```

Anmerkung: Die wesentlichen Änderungen sind die andere Implementierung von *my_lock()* und *my_unlock()* – unter Verwendung von *lockf()* und *lseek()* (um die gesamte Datei zu sperren) – und die Verwendung des Filedeskriptors zum Sperren. Deswegen müssen jetzt die *P()*- und die *V()*-Operation innerhalb von *open()* und *close()* sein.

```
thales$ gcc -Wall -o lockf lockf.c
thales$ cat seqno
001
thales$ lockf 2 > out1 & lockf 5 > out2 &
[3] 16357
[4] 16358
thales$
[3]-  Done                    lockf 2 >out1
[4]+  Done                    lockf 5 >out2
thales$ cat out1
BEGIN kritischer Bereich
gelesen: 1
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 4
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 6
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 8
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 10
END kritischer Bereich
thales$ cat out2
BEGIN kritischer Bereich
gelesen: 2
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 3
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 5
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 7
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 9
END kritischer Bereich
thales$ cat seqno
011
thales$
```

Anhang

Literatur

M. J. Bach: *The Design of the UNIX Operating System.* Prentice Hall, 1986.

D. Comer: *Operating System Design: The XINU Approach.* Prentice Hall, 1984.

P. A. Darnell und P. E. Margolis: *C: A Software Engineering Approach.* Springer, Dritte Auflage, 2001.

D. Goldberg: *What every computer scientist should know about floating-point arithmetic.* ACM Computing Surveys, Jahrgang 23, Heft 1 vom März 1991, Seiten 5-48.

T. Handschuch: *Solaris 2 f"ur den Systemadministrator.* Solaris Galerie, IWT-Verlag, 1993.

S. P. Harbison, G. L. Steele: *C: A Reference Manual.* Fünfte Auflage, Prentice Hall, 2002.

H. Herold: *Linux-Unix-Shells.* Addison-Wesley, 1999.

B. W. Kernighan und R. Pike: *Der UNIX-Werkzeugkasten.* Hanser, 1986.

B. W. Kernighan und D. Ritchie: *Programmieren in* C. Hanser, Zweite Auflage, 1990.

D. E. Knuth: *The Art of Computer Programming*, *Volume* 2, *Seminumerical Algorithms*, Addison-Wesley, Dritte Auflage, 1997.

A. Koenig: *C Traps and Pitfalls.* Prentice Hall, 1989.

M. Rochkind: *UNIX-Programmierung f"ur Fortgeschrittene.* Hanser Verlag, 1988.

W. R. Stevens: *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

B. Stroustrup: *The Design and Evolution of* C++. Addison-Wesley, 1994.

Sun Microsystems: *Code Conventions for the Java Programming Language.* Zu finden im Web unter http://java.sun.com/docs/codeconv/.

A. S. Tanenbaum: *Operating Systems - Design and Implementation.* Prentice Hall, 1987.
⇝ **Bitte auf die jeweils aktuellste Auflage achten!**

Abbildungsverzeichnis

1.1 Entwicklungsbeziehungen einiger Programmiersprachen 2

2.1 Anweisungsblock 10

7.1 Datentypen – Eine Übersicht 51
7.2 *big* vs. *little endian* 62
7.3 Konvertierungen zwischen numerischen Datentypen 64
7.4 Repräsentierung eines dreidimensionalen Vektors im Speicher 72
7.5 Funktionsaufrufe und lokale Variablen 85

8.1 Integration nach der Trapezregel 97

9.1 Aufteilung des Adressraums zu Beginn 102
9.2 Dynamisch belegter Speicher im Adressraum 104
9.3 Ring der freien Speicherflächen zu Beginn 107
9.4 Speicherverwaltungsstruktur nach der ersten Belegung einer Speicherfläche 107
9.5 Speicherverwaltungsstruktur nach drei Speicherbelegungen 107
9.6 Speicherverwaltungsstruktur nach einer Speicherfreigabe 107
9.7 Suche nach freien Flächen entsprechend des *circular first fit*-Verfahrens (Teil 1) 108
9.8 Suche nach freien Flächen entsprechend des *circular first fit*-Verfahrens (Teil 2) 108
9.9 Zusammenlegung benachbarter freier Speicherflächen 108

10.1 Die Repräsentierung von *argv* am Beispiel von *„gcc -Wall -std=gnu99 hello.c“* 119

12.1 Der Weg von der Quelle zum ausführbaren Programm 140
12.2 Übersetzungsvorgang bei ausgelagerten Deklarationen 142

15.1 Das Schichtenmodell der „erweiterten Maschine“ 170
15.2 Das Unix-Schalenmodell 171
15.3 Der interne Aufbau von Unix 172

16.1 Beispiel für ein Netzwerk-Dateisystem 178
16.2 Grobstruktur des Unix-Dateisystems 180
16.3 Adressierung der Datenblöcke 182
16.4 Kodierung der Komponente `st_mode` 186
16.5 Kodierung des Dateityps in `st_mode` 186
16.6 Datenstrukturen für I/O-Verbindungen 203
16.7 Kerneltabellen – Ausgangszustand 205
16.8 Kerneltabellen (1) 205
16.9 Kerneltabellen (2) 206
16.10 Kerneltabellen (3) 207
16.11 Kerneltabellen (4) 207

16.12 Kerneltabellen (5) 208
16.13 Kerneltabellen (6) 209
16.14 Kerneltabellen (7) 209

Beispiel-Programme

2.1 Hello World – Erste Version 5
2.2 Hello World – Verbesserte Version 6
2.3 Berechnung von Quadratzahlen mit einer for-Schleife 7
2.4 Berechnung von Quadratzahlen mit einer while-Schleife 7
2.5 Euklidischer Algorithmus 8
2.6 Euklidischer Algorithmus als Funktion 8
3.1 Verwendung der define-Direktive 15
3.2 Verwendung der *include*-Direktive 16
3.3 Eine winzige Header-Datei 16
4.1 Ausgabe mit *puts()* und *fputs()* 17
4.2 Ausgabe mit *puts()* und *printf()* 20
4.3 Eingabe mit *scanf()* 22
4.4 Eingabe mit *gets()* und *fgets()* 23
5.1 Geschachtelte **if**-Anweisungen mit **else** 26
5.2 Sauber geklammerte **if**-Anweisungen mit **else** 27
5.3 **else**-**if**-Kette 27
5.4 **while**: Zählen von Leerzeichen 28
5.5 **do-while**: Zählen von Leerzeichen bis zum Zeilenende 29
5.6 **do-while**: Überzählige Leerzeichen herausfiltern 29
5.7 Verwendung von **continue** 31
5.8 Zusammenhang zwischen **for**, **while** und **continue** 31
5.9 Verwendung von **break** 32
5.10 Beispiel für die **switch**-Anweisung 32
5.11 Beispiel für die **switch**-Anweisung, bei der mehrere Fälle gemeinsam behandelt werden 33
6.1 Verwendung unärer Operatoren 41
6.2 Verwendung binärer Operationen 44
6.3 Der Modulo-Operator 46
6.4 Verwendung des Komma-Operators 49
6.5 Zuweisungen 50
7.1 Zeichen als ganzzahlige Werte 55
7.2 Problematik von Rundungsfehlern 57
7.3 Problematik der Gleichheit bei Gleitkommazahlen 58
7.4 Verwendung von Aufzählungstypen 60
7.5 Verwendung von Zeigern 61
7.6 Zeiger-Arithmetik 62
7.7 Implizite Konvertierungen 66
7.8 Arbeiten mit Konstanten 67
7.9 Vektoren und Zeiger 68
7.10 Indizierungsfehler bei Vektoren 69
7.11 Parameterübergabe bei Feldern 70
7.12 Zeichenketten und Zeichenketten-Konstanten 74

7.13 Kopieren, Vergleichen, etc. von Zeichenketten 75
7.14 Verwendung diverser Funktionen für Zeichenketten 78
7.15 Verwendung der Funktion *strtok()* 79
7.16 Rekursive Strukturen 82
7.17 Zuweisung von Verbundtypen 82
7.18 Verbundtypen als Funktionsargumente 83
7.19 Verbunde als Ergebnis von Funktionen 84
7.20 Verwendung eines varianten Verbunds 86
8.1 Rekursive Berechnung der Fibonacci-Zahlen 92
8.2 Vertauschen der Werte zweier Variablen 93
8.3 Konflikt zwischen impliziter Deklaration und expliziter Definition einer Funktion 94
8.4 Vorab-Deklarationen bei Funktionen 95
8.5 Funktionszeiger: Integration nach der Trapezregel 97
9.1 Lineare Listen in C 100
9.2 Die Größe einer Kachel 101
9.3 Anordnung des Programmtexts, der globalen Variablen und des Laufzeitstapels im Adressraum 102
9.4 Dynamisches Belegen von Speicher mit Hilfe von *sbrk()* 105
9.5 Beispiel für eine dynamische Speicherverwaltung 108
9.6 Zufälliges Mischen ganzer Zahlen mit Hilfe eines dynamischen Vektors . . 114
9.7 Arbeiten mit Speicher-Operationen 118
10.1 Ausgabe der Kommandozeilenargumente und des Kommandonamens . . . 120
10.2 Alternative Bearbeitung der Kommandozeilenargumente 121
10.3 Ein vereinfachtes *grep* 122
10.4 Eine verbesserte Fassung von *grep* mit beliebig langen Eingabezeilen 122
10.5 Ein vereinfachtes *grep* mit Optionen 123
11.1 Definition und Verwendung von Makros 129
11.2 Verfolgung des Programmverlaufs mit Hilfe von Makros 131
11.3 Mehrfache Definition eines Makros 132
11.4 Zurücknahme einer Makrodefinition 132
11.5 Zusicherungen überprüfen 133
11.6 Überdefinierbare Makros 133
11.7 Makrodefinitionen in Abhängigkeit von der Kommandozeile 134
11.8 Bedingte Übersetzung 135
12.1 Beispiele für Deklarationen und Definitionen 137
12.2 Beispiel für eine Variablendeklaration 137
12.3 Übersetzungseinheit mit Deklarationen fremder Variablen und Funktionen 138
12.4 Übersetzungseinheit mit extern nutzbaren Definitionen 138
12.5 Gemeinsam genutzte Header-Datei mit der Deklaration der ggf-Funktion . 141
12.6 Getrennt übersetzbare Funktion zur Berechnung des ggT 141
12.7 Getrennt übersetzbares Hauptprogramm zur Berechnung des ggT 141
12.8 Einfaches makefile für das ggT-Beispiel 143
12.9 Allgemeine makefile-Vorlage mit Unterstützung von *makedepend* 146
12.10 Speicherklasse static für lokale Variablen 147
13.1 Einzeilige Kommentare 150
13.2 Mischen von Deklarationen/Definitionen und Anweisungen 150
13.3 Variablen in for-Schleifen 150
13.4 Arrays variabler Länge 151
13.5 Flexibles Array-Element in Strukturen 151
13.6 Nicht-konstante Initialisierer 152
13.7 Initialisierer für Arrays 153
13.8 Bereiche bei der switch-Anweisung 154

13.9 Boolesche Variablen 154
13.10 Große Integer 155
13.11 Funktion snprintf() 155
13.12 Variable Anzahl von Argumenten bei Makros 156
13.13 assert mit Name der aktuellen Funktion 157
13.14 Inline-Funktionen 157
14.1 C-Programm mit vielen Schwachstellen 161
14.2 printf() und die Formatierungsangabe %n 164
14.3 Arbeiten mit der stralloc-Bibliothek 166
16.1 Informationen über Dateien ermitteln – mittels stat() 188
16.2 Dateien in einem Verzeichnis ermitteln 189
16.3 Öffnen einer Datei – mittels open() 194
16.4 Umlenkung der Standardausgabe – mittels dup() 195
16.5 Typ der Standardeingabe ermitteln – mittels fstat() 196
16.6 Ausgabe eines Strings mit Hilfe von write() 198
16.7 Ausgabe eines Strings mit Hilfe von write() 198
16.8 Einlesen einer Zeile von der Standardeingabe – mittels read() 199
16.9 Fehlerbehandlung bei Systemaufrufen – Die Erste 200
16.10 Fehlerbehandlung bei Systemaufrufen – Die Zweite 201
16.11 Positionieren in Dateien – mittels lseek() 210
16.12 Erzeugen einer temporären Datei 212
16.13 oflags verändern – mittels fcntl() 213
16.14 Echo aus- und anschalten – mittels ioctl() 214
16.15 Inkrementieren ohne Synchronisation 216
16.16 Inkrementieren mit Synchronisation 219
16.17 Inkrementieren mit Synchronisation – mittels lockf() 223

Index

!, 41
(, 25, 37, 40, 68, 91
), 25, 37, 40, 68, 91
*, 18, 40, 43, 61, 68
*=, 49
+, 18, 40, 43
++, 39, 40
+=, 49
„ 35, 37, 48, 59, 61, 80, 91
-, 18, 40, 43
–, 39, 40
-=, 49
->, 37, 40
., 18, 37, 40
..., 91
/, 43, 44
/*…*/, 11
/=, 49
:, 25, 48, 80
;, 9, 25, 80
<, 43
«=, 49
<=, 43
«, 43, 44
=, 10, 49, 59, 61
==, 10, 43
>, 43
>=, 43
»=, 49
», 43, 44
?, 48
?:, 48
[, 37
, 18
#, 14, 18
%, 18, 43, 46
%=, 49
%p, 19
&, 8, 40, 44
&=, 49
&&, 43, 44
_, 11
_Alignas, 11, 158
_Alignof, 11, 159
_Atomic, 11
_Bool, 11, 52, 53
_Complex, 11, 55
_Generic, 11
_Imaginary, 11
_Noreturn, 11
_Static_assert, 11
_Thread_local, 11, 159
|, 44
||, 44
^, 44
[, 68, 91
], 68, 91
^, 43
^=, 49
|, 43
|=, 49
||, 43
~, 40
{, 10, 37, 59, 80, 86
}, 10, 37, 59, 80, 86
], 37

0, 18

abstract-declarator, 91
add-op, 43
additive-expression, 43
address-expression, 37, 40
Adressoperator, 8, 21, 40
Adressraum, 101
Aggregat, 38
alignas, 158
alignof, 159
Anweisung
- break, 32
- case, 32
- continue, 31
- do-while, 29
- for, 30
- if, 26
- switch, 32
- while, 28

Anweisungsblock, 9
argc, 119

Argumente
  Kommandozeilen-, 119
argv, 120
Array, 68
  mehrdimensional, 71
array-declarator, 68
array-qualifier, 68
array-qualifier-list, 68
array-size-expression, 68
assert, 114
assignment-expression, 35, 48, 49, 68
assignment-op, 35, 49
Assoziativität, 39
Aufzählungsdatentypen, 59
Ausdruck, 35
Ausgabe, 17
Auswahloperator, 48
Auswertungsreihenfolge, 43
auto, 11, 88, 147
automatische Speicherbereinigung, 105

bedingter Ausdruck, 48
Bezeichner, 11
big endian, 63
Binder, 103, 139
binäre Operatoren, 42
Bitfeld, 44
bitwise-and-expression, 43
bitwise-negation-expression, 37, 40
bitwise-or-expression, 43
bitwise-xor-expression, 43
Blockstruktur, 9
Boehm-Demers-Weiser Speicherbereiniger, 106
bool, 11, 26, 44
bool-type-specifier, 52
break, 11, 25, 32, 33
break-statement, 10, 25
bss, 103

C-Compiler
  gcc, 5
C-Präprozessor
  cpp, 14
C11, 11
C89, 5, 10, 14, 43, 98
C99, 5, 6, 10, 11, 14, 19, 26, 30, 46, 53, 65, 92, 129
call by reference, 93
call by value, 92
calloc(), 99
case, 11, 25, 32, 33
case-label, 25
cast-expression, 40, 43
char, 11, 29, 52–54, 89, 117–120
CHAR_BIT, 54
character-constant, 37
character-type-specifier, 52
chmod, 186
chmod(), 186
comma-expression, 35, 48
complex-type-specifier, 55
component-selection-expression, 37
compound-literal, 37
compound-statement, 10, 91
conditional-expression, 35, 48, 49
conditional-statement, 10, 25
const, 11, 67, 68
constant, 37
constant-expression, 25, 80, 91
continue, 11, 25, 31
continue-statement, 10, 25
ConvChar, 18
ConvSpec, 18
cpp, 14

d, 18
dangling else, 11
Datentyp
  _Complex, 55
  double, 55
  enum, 59
  float, 55
  int, 10
  struct, 80
  union, 86
  void, 92
Datentypen, 51
  skalare, 52
declaration, 9, 10, 25, 61, 89
declaration-list, 91
declaration-or-statement, 10
declaration-or-statement-list, 10
declaration-specifiers, 9, 61, 67, 91
declarator, 61, 80, 89, 91
deep-copy, 38
default, 11, 25
default-Fall, 33
default-label, 25
define, 14, 128
Definition, 137, 138
Deklaration, 61, 137, 138
Dereferenzierungsoperator, 40
Diagnoseausgabe, 17
Digit, 18
direct-abstract-declarator, 91
direct-component-selection, 37
direct-declarator, 61, 68, 91

Direktive
  define, 14
  include, 16
do, 11, 25, 29
do-statement, 25
do-while, 29
double, 11, 19, 55, 57, 58, 65, 139
dyadische Operatoren, 42
dynamische Vektoren, 114
dynamische Zeichenketten, 117

edata, 103
Einer-Komplement, 53
Eingabe, 17
else, 11, 25–28
end, 103
enum, 11, 59
enumeration-constant, 59
enumeration-constant-definition, 59
enumeration-definition-list, 59
enumeration-tag, 59
enumeration-type-definition, 59
enumeration-type-reference, 59
enumeration-type-specifier, 52, 59
equality-expression, 43
equality-op, 43
etext, 103
Exit-Status, 6
expression, 25, 35, 37, 48, 59
expression-list, 37
expression-statement, 10, 25
extern, 11, 88, 103, 137

false, 26, 135
Feld, 68
fgets(), 23
Flag, 18
float, 11, 55–57, 65
floating-constant, 37
floating-point-type-specifier, 52, 55
for, 7, 11, 25, 30, 31
for-statement, 25
fprintf(), 21
fputs(), 17
free, 105
free(), 99
function-call, 37
function-declarator, 68, 91
function-def-specifier, 91
function-definition, 9, 91
function-specifier, 9, 61, 67, 91
Funktion, 91
  als Parameter, 96
  eingebettet, 14
Funktionszeiger, 96

garbage collection, 105
gcc, 5, 14
getchar(), 28
gets(), 22
Gleitkommazahlen, 55
goto, 11, 25
goto-statement, 10, 25
Grammatik
  abstract-declarator, 91
  add-op, 43
  additive-expression, 43
  address-expression, 37, 40
  array-declarator, 68
  array-qualifier, 68
  array-qualifier-list, 68
  array-size-expression, 68
  assignment-expression, 35, 48, 49, 68
  assignment-op, 35, 49
  bitwise-and-expression, 43
  bitwise-negation-expression, 37, 40
  bitwise-or-expression, 43
  bitwise-xor-expression, 43
  bool-type-specifier, 52
  break-statement, 10, 25
  case-label, 25
  cast-expression, 40, 43
  character-constant, 37
  character-type-specifier, 52
  comma-expression, 35, 48
  complex-type-specifier, 55
  component-selection-expression, 37
  compound-literal, 37
  compound-statement, 10, 91
  conditional-expression, 35, 48, 49
  conditional-statement, 10, 25
  constant, 37
  constant-expression, 25, 80, 91
  continue-statement, 10, 25
  ConvChar, 18
  ConvSpec, 18
  declaration, 9, 10, 25, 61, 89
  declaration-list, 91
  declaration-or-statement, 10
  declaration-or-statement-list, 10
  declaration-specifiers, 9, 61, 67, 91
  declarator, 61, 80, 89, 91
  default-label, 25
  Digit, 18
  direct-abstract-declarator, 91
  direct-component-selection, 37
  direct-declarator, 61, 68, 91

do-statement, 25
enumeration-constant, 59
enumeration-constant-definition, 59
enumeration-definition-list, 59
enumeration-tag, 59
enumeration-type-definition, 59
enumeration-type-reference, 59
enumeration-type-specifier, 52, 59
equality-expression, 43
equality-op, 43
expression, 25, 35, 37, 48, 59
expression-list, 37
expression-statement, 10, 25
Flag, 18
floating-constant, 37
floating-point-type-specifier, 52, 55
for-statement, 25
function-call, 37
function-declarator, 68, 91
function-def-specifier, 91
function-definition, 9, 91
function-specifier, 9, 61, 67, 91
goto-statement, 10, 25
identifier, 37, 59, 68, 80, 86, 88, 91
identifier-list, 91
indirect-component-selection, 37
indirection-expression, 37, 40
init-declarator, 61, 68
init-declarator-list, 61
initial-clause, 25
initialized-declarator-list, 9
initializer, 61
initializer-list, 37
integer-constant, 37
integer-type-specifier, 52
iterative-statement, 10, 25
label, 25
labeled-statement, 10, 25
logical-and-expression, 43
logical-negation-expression, 37, 40
logical-or-expression, 43, 48
MinWidth, 18
mult-op, 43
multiplicative-expression, 43
named-label, 25
null-statement, 10, 25
parameter-declaration, 91
parameter-list, 91
parameter-type-list, 91
parenthesized-expression, 37
pointer, 61, 91
postdecrement-expression, 37
postfix-expression, 37
postincrement-expression, 37
Precision, 18
predecrement-expression, 37, 40
preincrement-expression, 37, 40
primary-expression, 37
relational-expression, 43
relational-op, 43
return-statement, 10, 25
shift-expression, 43
shift-op, 43
signed-type-specifier, 52
simple-declarator, 68
SizeModifier, 18
sizeof-expression, 37, 40
specifier-qualifier-list, 80
statement, 10, 25
storage-class-specifier, 9, 61, 67, 88
string-constant, 37
struct-declaration, 80
struct-declaration-list, 80, 86
struct-declarator, 80
struct-declarator-list, 80
structure-type-specifier, 52, 80
subscript-expression, 37
switch-statement, 10, 25
top-level-declaration, 9
translation-unit, 9
type-name, 37, 40
type-qualifier, 9, 61, 67, 80
type-qualifier-list, 61
type-specifier, 9, 52, 61, 67, 68, 80, 88, 89
typedef-name, 52, 88
unary-expression, 35, 37, 40, 49
unary-minus-expression, 37, 40
unary-plus-expression, 37, 40
union-type-specifier, 52, 86
unsigned-type-specifier, 52
void-type-specifier, 52
while-statement, 25

h, 18
Header-Dateien, 16, 140
hh, 18

i, 18
identifier, 37, 59, 68, 80, 86, 88, 91
identifier-list, 91
IEC 60559, 56
IEEE Std 1003.1, 79
IEEE-754, 56
if, 11, 25–28, 48
ifdef, 133
implizite Konvertierung, 65
include, 16, 127

indirect-component-selection, 37
indirection-expression, 37, 40
Infix-Notation, 42
init-declarator, 61, 68
init-declarator-list, 61
initial-clause, 25
initialized-declarator-list, 9
initializer, 61
initializer-list, 37
inline, 11, 14, 91, 129
int, 6, 10, 11, 19, 26, 28, 29, 40, 41, 52–54, 59, 61, 63, 65, 66, 68, 71, 72, 88–90
integer-constant, 37
integer-type-specifier, 52
iterative-statement, 10, 25

j, 18
Java, 6, 119

K&R-Standard, 1
Kachel, 101
Komma-Operator, 48
Kommandozeilenoptionen, 123
Kommandozeilenparameter, 119
Kommentar, 11
    /*…*/, 11
Kommutativität, 43
Komplement
    logisches, 41
Konstante, 7
Konvertierung
    implizit, 65
Konvertierungen, 63

L, 18
l, 18
label, 25
labeled-statement, 10, 25
Laufzeitstapel, 101, 104
ld, 103, 139
Leerzeichen, 11
Links-Wert, 35, 65
little endian, 63
ll, 18
logical-and-expression, 43
logical-negation-expression, 37, 40
logical-or-expression, 43, 48
lokale Variable, 62
long, 11, 52–55, 57, 66
long double, 57
long int, 19
long long int, 65

m4, 13
main(), 6, 119
make, 143
Makro, 128
    define, 14
    definieren, 128
    Existenz, 133
    include, 16
Makroprozessor, 13
malloc, 105
malloc(), 99
Matrix, 71
matrix, 72
mdb, 103
memcmp(), 117
memcpy(), 117
memmove(), 117
memset(), 117
Menge, 44
MinWidth, 18
Modulo-Operator
    F-Definition, 46
    nach Euklid, 46
    T-Definition, 46
monadische Operatoren, 39
mult-op, 43
multiplicative-expression, 43
my_calloc(), 101

named-label, 25
Namen, 11
NaN, 56
new, 99
nm, 138
null, 52
Null-Byte, 74
null-statement, 10, 25

o, 18
Operator
    !, 41
    − >, 81
    *, 40, 98
    +, 62
    ++, 39, 40
    „ 48
    -, 62
    –, 39, 40
    ->, 40
    ., 40
    /, 44
    «, 44
    =, 10, 49
    ==, 10
    », 44

?:, 48
%, 46
&, 8, 40, 44
&&, 44
|, 44
||, 44
^, 44
sizeof, 41
Operatoren, 39
bitweise, 44
dyadisch, 42
logische, 44
monadisch, 39
Optionen
Kommandozeilen-, 123

parameter-declaration, 91
parameter-list, 91
parameter-type-list, 91
Parameterübergabe, 8
main(), 119
parenthesized-expression, 37
pmap, 103
pointer, 61, 91
POSIX, 79
POSIX_C_SOURCE, 79
postdecrement-expression, 37
postfix-expression, 37
Postfix-Operator, 39
Postfix-Operatoren, 39
postincrement-expression, 37
Precision, 18
predecrement-expression, 37, 40
preincrement-expression, 37, 40
primary-expression, 37
printf, 65
printf(), 7, 17
Formate, 17
Prozedur, 91
Präfix-Operatoren, 40
Präprozessor, 79
puts(), 6, 17

Rang, 66
Rang bei numerischen Datentypen, 65
realloc, 122
realloc(), 100, 114
Rechts-Wert, 36, 65
Referenz-Parameter, 93
register, 11, 88
relational-expression, 43
relational-op, 43
restrict, 11, 67, 68
return, 6, 11, 92
return-statement, 10, 25

scanf(), 8, 21
Schiebe-Operatoren, 44
Schleife
do-while, 29
for, 7, 30
while, 7, 28
Schlüsselwort
_Alignas, 11, 158
_Alignof, 11, 159
_Atomic, 11
_Bool, 11, 52, 53
_Complex, 11, 55
_Generic, 11
_Imaginary, 11
_Noreturn, 11
_Static_assert, 11
_Thread_local, 11, 159
alignas, 158
alignof, 159
auto, 11, 88, 147
bool, 11
break, 11, 25, 32
case, 11, 25
char, 11, 29, 52–54, 89, 117–120
const, 11, 67, 68
continue, 11, 25, 31
default, 11, 25
do, 11, 25, 29
double, 11, 19, 55, 57, 58, 65, 139
else, 11, 25–28
enum, 11, 59
extern, 11, 88, 103, 137
false, 135
float, 11, 55–57, 65
for, 7, 11, 25, 30, 31
goto, 11, 25
if, 11, 25–28, 48
inline, 11, 91, 129
int, 6, 10, 11, 19, 26, 28, 29, 40, 41, 52–54, 59, 61, 63, 65, 66, 68, 71, 72, 88–90
long, 11, 52–55, 57, 66
long double, 57
long int, 19
long long int, 65
matrix, 72
new, 99
null, 52
register, 11, 88
restrict, 11, 67, 68
return, 6, 11, 92
short, 11, 52–54, 59, 66, 81

- short int, 19
- signed, 11, 52–54, 66
- signed char, 52
- sizeof, 11, 39–41
- static, 11, 68, 88, 147, 148
- struct, 11, 80, 86, 89, 90
- switch, 11, 25, 32, 33
- thread_local, 159
- true, 135
- typedef, 11, 88
- union, 11, 86
- unsigned, 11, 52–54, 63, 65, 66, 117, 118
- unsigned char, 19, 52
- unsigned long, 41
- void, 11, 92, 93, 103
- volatile, 11, 67, 68
- while, 7, 25, 28, 29, 31

Schlüsselworte, 11
Schnittstellensicherheit, 140, 143
Schnittstellenänderung, 143
Semikolon, 26
shallow-copy, 38
Shell, 6
Shellvariable
- ?, 6

shift-expression, 43
shift-op, 43
short, 11, 52–54, 59, 66, 81
short int, 19
signed, 11, 52–54, 66
signed char, 52
signed-type-specifier, 52
simple-declarator, 68
size, 103
size_t, 77
SizeModifier, 18
sizeof, 11, 39–41
sizeof-expression, 37, 40
snprintf(), 155
Solaris, 79
specifier-qualifier-list, 80
Speicher
- belegen, 99
- freigeben, 99

Speicherbereinigung, 105
Speicherklasse
- auto, 11, 147
- extern, 103
- register, 11

sprintf(), 155
Standardausgabe, 17
Standardeingabe, 17
stat(), 188
statement, 10, 25
static, 11, 68, 88, 147, 148
stderr, 17
stdin, 17
stdout, 17
storage-class-specifier, 9, 61, 67, 88
strcasecmp(), 77
strcat(), 77
strchr(), 77
strcmp(), 77
strcpy(), 77
strcspn(), 77
strdup(), 117
string-constant, 37
Strings, 74
strings.h, 76
strlen(), 77
strpbrk(), 77
strspn(), 77
strstr(), 77
strtok(), 77
struct, 11, 80, 86, 89, 90
struct-declaration, 80
struct-declaration-list, 80, 86
struct-declarator, 80
struct-declarator-list, 80
structure-type-specifier, 52, 80
subscript-expression, 37
switch, 11, 25, 32, 33, 154
switch-statement, 10, 25

t, 18
template, 14
thread_local, 159
top-level-declaration, 9
translation-unit, 9
Trapezregel, 96
true, 26, 135
Typ-Konvertierungen, 63
Typdefinition, 88
type-name, 37, 40
type-qualifier, 9, 61, 67, 80
type-qualifier-list, 61
type-specifier, 9, 52, 61, 67, 68, 80, 88, 89
typedef, 11, 88
typedef-name, 52, 88

u, 18
unary-expression, 35, 37, 40, 49
unary-minus-expression, 37, 40
unary-plus-expression, 37, 40
ungetc(), 30
union, 11, 86
union-type-specifier, 52, 86

unsigned, 11, 52–54, 63, 65, 66, 117, 118
unsigned char, 19, 52
unsigned long, 41
unsigned-type-specifier, 52
Unterstrich, 11
unär, 39

Variable
    globale, 7
    lokale, 7, 62
Variante Verbünde, 86
Vektor, 68
    mehrdimensional, 71
Vektoren
    dynamisch, 114
Verbundtypen, 80
Vereinbarung, 9
Vergleichsoperator, 10
void, 11, 92, 93, 103
void-type-specifier, 52
volatile, 11, 67, 68
Vorrang, 39

wchar_t, 54
Werteparameter, 92
Wertzuweisung, 49
while, 7, 25, 28, 29, 31
while-statement, 25
white space characters, 11

X, 18
x, 18

z, 18
Zeichen, 54
Zeichenketten, 74
    dynamisch, 117
    Funktionen, 76
Zeiger, 8
Zeiger-Arithmetik, 62
Zeigertypen, 61
Zuweisung, 10, 49
Zuweisungsoperator, 49
Zweier-Komplement, 53, 63

Übersetzungsabhängigkeiten, 143
Übersetzungseinheit, 138