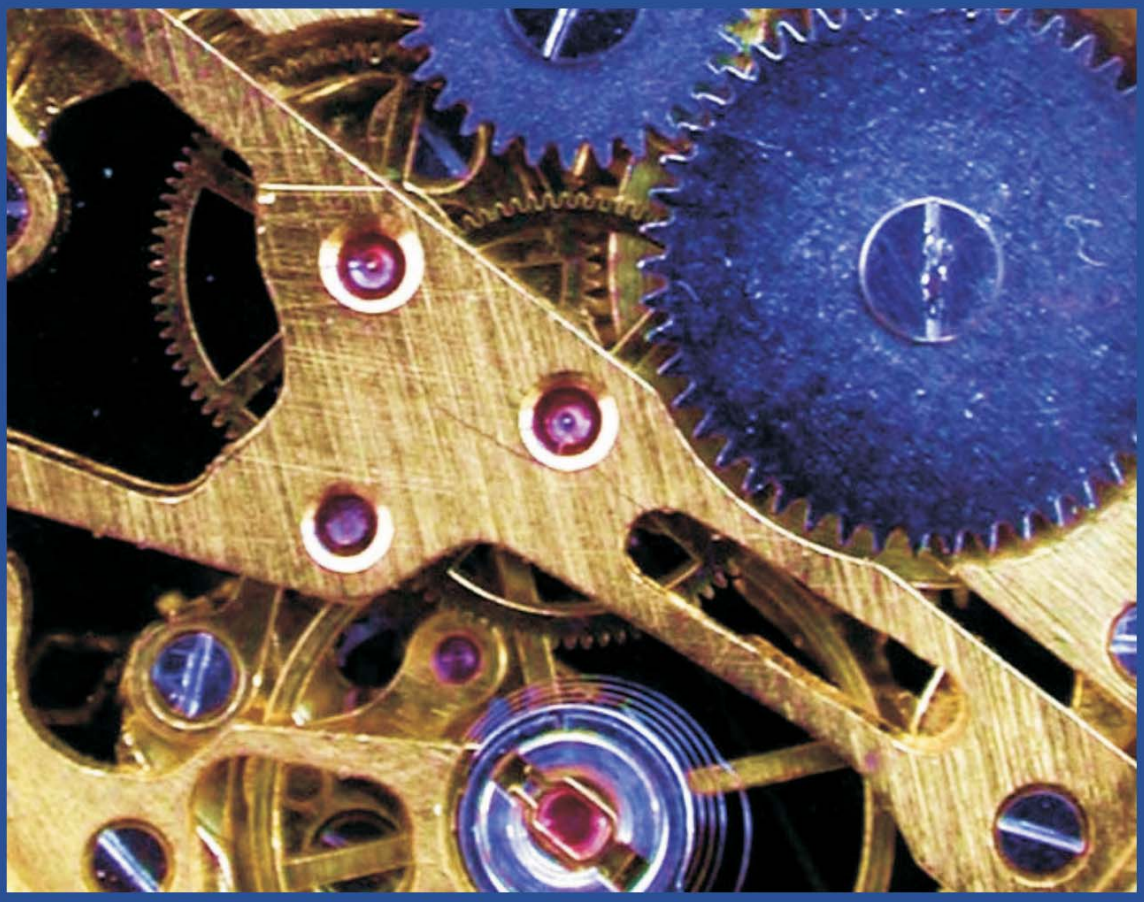
Systemnahes Programmieren

C Programmierung unter Unix und Linux



Neue überarbeitete Auflage

Inhaltsverzeichnis

Vorwort

Während das Betriebssystem UNIX schon von jeher einen großen Marktanteil im Serverbereich aufzuweisen hatte, zeigt nun Linux, dass ein entsprechendes Potential auch im Workstationbereich besteht. Nicht zu vergessen ist die Tatsache, dass das aktuelle Betriebssystem Mac OS X für Apple-Computer ebenfalls auf solide UNIX-Technologie setzt: Darwin, die grundlegende Betriebssystemschicht von Mac OS X, enthält einen Mach 3.0 Kern, der, wie Linux, ein Abkömmling von UNIX ist.

Generell kann man also sagen, dass den Varianten des UNIX-Betriebssystems auch in Zukunft eine große Bedeutung zukommen wird.

Aus diesem Grunde haben wir, die Mitarbeiter des Instituts für Technische Informatik an der Technischen Universität Wien, uns entschlossen, ein Buch über die Programmierung solcher Systeme herauszugeben.

Das vorliegende Buch ist sowohl als Einführungswerk für den Anfänger als auch als Nachschlagewerk für Fortgeschrittene konzipiert. Es werden Kenntnisse über den Aufbau des Betriebssystems und die im Betriebssystem verwendeten Mechanismen vermittelt.

Behandelt werden konkret die Synchronisation paralleler Prozesse, Interprozesskommunikation, Signale, Pipes und Systemfunktionen zur Fileverwaltung und Systemadministration. Verstärkte Berücksichtigung findet die Interprozesskommunikation unter Verwendung von Semaphoren, Shared Memory und Nachrichten.

Für all diese Techniken werden detaillierte Anleitungen zur Implementierung gegeben.

Dieses Buch ist wie folgt aufgebaut: Im folgenden Kapitel wird eine grundlegende Einführung in die Bedienung von UNIX/Linux über die Shell gegeben. Daran schließt ein Kapitel mit detaillierten Beschreibungen der Mechanismen von UNIX/Linux an. Danach folgt ein Kapitel, welches dem Leser die Programmiersprache C näherbringt. Das letzte Kapitel behandelt das Hauptthema dieses Buches, die Programmierung unter UNIX/Linux.

Dieses Buch richtet sich vor allem an echte C-Programmierer, die Systemroutinen als Kommandozeilenprogramme entwickeln wollen. In Zukunft wird die Wichtigkeit von Systemstabilität weiter anwachsen, daher wird besonderer Wert auf saubere und sichere Programmierung gelegt.

Objektorientierte Programmierung und die Programmierung von grafischen Oberflächen werden in diesem Buch nicht behandelt, für diese Aufgaben sei der Leser jedoch auf die zahlreich vorhandene ergänzende Literatur verwiesen.

Das Zustandekommen dieses Buches wurde erst durch die Zusammenarbeit vieler Leute ermöglicht. Insbesondere möchte ich an dieser Stelle die Assistenten, Professoren, Sekretärinnen, Techniker und

Tutoren des Instituts für Technische Informatik erwähnen, die im Rahmen der Lehrtätigkeit an der Technischen Universität Wien für das Zustandekommen der Inhalte dieses Buches gesorgt haben. Ich möchte ebenfalls denjenigen Studenten danken, die mit kritischen Fragen geholfen haben, die Qualität des vorliegenden Lehrwerks zu steigern. Über weitere Kommentare und Verbesserungsvorschäge sind wir dankbar. Diese können an die Email-Adresse `sysprogbuch@vmars.tuwien.ac.at` gesendet werden.

Wilfried Elmenreich

Kapitel 1

Einführung in UNIX

Das folgende Kapitel soll eine kurze Einführung in die Bedienung von Unices (UNIX und Derivate wie z.B. Linux) geben.

1.1 Erste Schritte

UNIX ist ein Mehrbenutzer- und Mehrprozess-Betriebssystem. Es ist daher erforderlich, dass zunächst jeder Benutzer dem System seine Identität bekannt gibt.

1.1.1 Login

Nachdem mit dem Terminalserver eine Verbindung zum Rechner hergestellt wurde, findet man am Bildschirm des Terminals, auf dem man arbeiten will, die Meldung

```
login:
```

vor. Es ist dies die Aufforderung, seinen Benutzernamen bekanntzugeben.

Wurde einem Benutzer vom Systemmanager keine Benutzungsberechtigung, also keine Identität, zugewiesen, kann er mit dem System nicht arbeiten.

Nach der Eingabe der Identität erscheint die Meldung

```
Password:
```

Darauf ist das geheime Losungswort einzugeben. Zum Arbeiten auf dem System sollte ein Passwort gewählt werden, das nur dem Benutzer selbst bekannt ist.

Von Zeit zu Zeit sollte man sein Losungswort ändern. Dazu dient das Kommando

```
passwd
```

für das Ändern des Passworts am lokalen Rechner beziehungsweise

```
yppasswd
```

für Rechnersysteme mit Network Information Service (NIS) Verwaltung.

Es muss zuerst das alte Passwort eingetippt werden (als Schutz vor 'netten' Kollegen, die sonst das Passwort ändern könnten, wenn sie kurz Zugang zum Terminal haben). Nun kann das (neue) Passwort eingetippt werden. Passwörter werden niemals am Bildschirm angezeigt. Daher ist (als Schutz vor Tippfehlern) das neue Passwort zweimal einzugeben.

1.1.2 Passwörter

Passwörter sollten völlig geheim gehalten werden. Damit das auch gelingt, ist es wichtig, die Methoden zu kennen, mit denen es möglich ist, fremde Passwörter herauszufinden:

- Man kann dem Benutzer beim Einloggen über die Schulter schauen. Wenn ein Passwort ein korrekt geschriebenes Wort in einer bekannten Sprache ist, reichen oft schon wenige Buchstaben, um den Rest mit Herumprobieren herauszufinden. Wenn man etwa einen Benutzer beobachtet, der die Buchstaben "reg.....ter" tippt, ist es recht naheliegend, das Wort "regenwetter" zu versuchen.

  → *Achten Sie darauf, dass Sie nicht beobachtet werden! Verwenden Sie keine Wörter, bei denen aus einem kleinen Teil der Rest leicht zu erraten ist. Verwenden Sie Passwörter, die Ihnen beim Tippen gut von der Hand gehen. Verwenden Sie Passwörter, die acht Zeichen lang sind: Wenn Sie beim Eintippen beobachtet werden, tippen Sie nach Ihrem Passwort noch eine Zeitlang unsinnig weiter, bevor sie 'return' tippen. Alle Zeichen nach dem achten werden vom Computer ignoriert, verwirren aber den Neugierigen.*

- Man kann mit relativ geringem Aufwand ein Programm schreiben, das sich wie das Login-Programm verhält, dieses auf einem Terminal starten und warten, bis jemand versucht, sich dort einzuloggen. Sobald ein Benutzer darauf hineinfällt, schreibt das Programm den Namen und das Passwort auf ein File, gibt die Meldung "login incorrect" aus und startet das echte Login-Programm.

  → *Geben Sie Ihr Passwort immer sorgfältig ein und ändern Sie es sofort, wenn Sie die Meldung "login incorrect" erhalten, obwohl Sie sicher sind, dass Sie sich nicht vertippt haben.*

- Man kann die Aufzeichnungen des Benutzers durchsuchen. Manche Menschen notieren sich Passwörter im Telefonverzeichnis (vielleicht unter 'S' wie 'Sysprog'; der Bankomatcode steht dann unter 'G' wie 'Geld').

  → *Schreiben Sie Ihr Passwort (und um Himmels Willen erst recht Ihren Bankomatcode!) nirgends auf. Wählen Sie stattdessen ein Passwort, das Sie sich leicht merken können.*

- Man kann den Benutzer fragen. Das ist nicht so lächerlich, wie es klingt. Mit telefonischen Anfragen der Art: "Guten Tag, hier spricht Gerhard Gonter vom Rechenzentrum. Wir haben ein Problem mit der Platte, auf der Ihre Daten stehen, und brauchen Ihr Passwort, um sie auf

Band speichern zu können, während wir die Platte neu formatieren" kann man erstaunliche Erfolge erzielen.

→ Geben Sie Ihr Passwort niemandem bekannt, auch wenn es sich scheinbar um eine Autorität handelt. Es gibt keine Tätigkeiten, für die der Systemadministrator Ihr Passwort kennen müsste.

- Man kann 'wahrscheinliche' Passwörter händisch durchprobieren. Besonders populär sind Namen von Freundinnen/Freunden, verkehrt geschriebene Namen, Geburtsdaten, ...

  → Verwenden Sie kein Passwort, das im Zusammenhang mit Ihrer Person steht.

- Man kann ein Programm schreiben, das eine große Anzahl von möglichen Passwörtern durchprobiert. In UNIX ist der Algorithmus, der zur Passwortverschlüsselung verwendet wird, allgemein bekannt (eine Variante des Data Encryption Standard DES). Ebenso ist das File, das die verschlüsselten Passworte enthält, in vielen UNIX-Versionen für alle Benutzer lesbar. Es ist daher mit vertretbarem Aufwand möglich, ein Programm zu schreiben, das ein (Klartext) Wort verschlüsselt und mit dem verschlüsselten Passwort eines Benutzers vergleicht. Wenn die beiden übereinstimmen, hat man das Passwort des Benutzers gefunden.

  Mit einer schnellen Implementierung des DES kann man selbst mit älteren Workstations etwa 10000 Wörter in der Sekunde überprüfen. Das reicht bei weitem nicht aus, um den gesamten Suchraum aller möglichen Passwörter zu durchsuchen: Angenomme ein Passwort ist 8 Buchstaben lang, damit ergeben sich mit den 95 druckbaren ASCII-Zeichen 95^8 Möglichkeiten. Um alle diese durchzuprobieren bräuchte man länger als 20000 Jahre.

  Aber Vorsicht: Der Duden hat nur ca. 200000 Stichwörter. Nachdem viele Rechner ein deutsches Wörterbuch on-line haben, ist die Wahrscheinlichkeit sehr groß, dass ein deutsches Wort als Passwort in weniger als vier Minuten gefunden werden kann! Für Wörter anderer Sprachen gilt das natürlich ebenso.

  → Verwenden Sie keine korrekt geschriebenen Wörter irgendeiner Sprache als Passwort! Verwenden Sie lange Passwörter! Verwenden Sie ein großes Alphabet: Groß- und Kleinbuchstaben gemischt, ebenso Ziffern und Sonderzeichen.

Fassen wir zusammen! Ein gutes Passwort muss folgende Anforderungen erfüllen:

- Es darf kein Wort einer bekannten Sprache sein.
- Es darf in keinem Zusammenhang mit der eigenen Person stehen.
- Es soll Groß- und Kleinbuchstaben, Sonderzeichen und Ziffern enthalten.
- Es soll möglichst lang sein.
- Es muss leicht zu tippen sein.
- Es muss leicht zu merken sein.
- Es muss schwer zu erraten sein.

Es scheint, dass diese Anforderungen nur schwer gemeinsam zu erfüllen sind. Es gibt aber eine Methode, mit der man sehr gute Passwörter erzeugen kann. Diese Methode wird im nächsten Abschnitt behandelt:

Akronyme als Passwörter

Eine Methode, um gut geeignete Passwörter zu finden, ist es, die Anfangsbuchstaben eines Satzes zu einem Wort zusammenzufügen und dieses Wort dann zu verwenden. Auf den vorigen Satz angewendet, ergibt diese Methode das Wort 'EMuggPzf'. Perfektionisten können dann noch einzelne Zeichen durch Ziffern und/oder Sonderzeichen ersetzen.

Passwörter, die mit dieser Methode erzeugt werden, haben folgende positive Eigenschaften:

- Sie kommen, außer durch Zufall, in keinem Wörterbuch vor[1].
- Sie sind schwer zu erraten, selbst wenn man einen Teil durch Beobachtung herausgefunden hat: Man kennt eben nur einen Teil des Wortes, aber nicht einen Teil des Sinns der dahintersteckt.
- Sie sind leicht zu merken, weil man sich beim Einloggen den erzeugenden Satz vorsagen kann (bitte nur in Gedanken!).
- Sie sind in fast beliebiger Länge erzeugbar.

1.2 Die UNIX Shell

Eine UNIX Shell ist ein Programm, das zwei Aufgaben erfüllt: Eine Aufgabe ist es, Kommandos vom Benutzer zu empfangen und dann aufgrund dieser Kommandos die entsprechenden Programme auszuführen. Zusätzlich kann eine Shell Programme (sogenannte *Shellscripts*) ausführen, die in einer speziellen Programmiersprache geschrieben sind.

1.2.1 Verschiedene Shells

Eine Shell ist kein Bestandteil des Betriebssystemkernels, sondern ein Anwenderprogramm. Man kann daher auch nicht von "der" UNIX Shell sprechen. Folgende Shells sind weitverbreitet:

sh Die 'Bourne-Shell' war die erste Shell. Sie ist auch heute noch die am weitesten verbreitete Shell. Sie ist für Shell-Programme relativ gut geeignet, aber als interaktive Shell zu unkomfortabel.

csh Die 'C-Shell' ist eine Shell mit C-ähnlicher Syntax.

ksh Die 'Korn-Shell' ist der Bourne-Shell sehr ähnlich, enthält aber auch Features der 'C-Shell'.

bash Die 'Bourne-Again-Shell' ist der Bourne-Shell sehr ähnlich, enthält auch nützliche Features der 'C-Shell' und 'Korn-Shell'. Sie ist für Einsteiger angenehmer zu bedienen.

In den folgenden Ausführungen beziehen wir uns auf die 'Bourne-Shell'; das Gesagte gilt jedoch auch für die 'Bourne-Again-Shell'.

[1] Vorsicht: Das kann sich ändern, wenn die Cracker professioneller werden!

Bei der Verwendung als Kommandointerpreter weist die Shell den Benutzer durch Ausgabe des sog. *Shell-Prompts* , (meist das Zeichen **$**) an, dass sie auf die Eingabe eines Kommandos wartet. Hierauf kann der Benutzer ein Kommando eingeben.

1.2.2 Shellvariablen

In der Shell können auch Variablen verwendet werden. Variablen sind immer vom Typ 'string' und werden durch die erste Verwendung deklariert. Die Syntax der Zuweisung ist wie folgt:

```
$ FN=/usr/users/01/s8225607
```

Damit wird der Variablen `FN` der String `/usr/users/01/s8225607` zugewiesen. *Beachten Sie, dass vor und nach dem Zuweisungsoperator '=' kein Leerzeichen stehen darf!*

Den Wert dieser Variablen erhält man mit der Konstruktion `${FN}`. Alternativ dazu kann auch einfach `$FN` verwendet werden, das ist allerdings nur dann möglich, wenn hinter dem Variablennamen keine Zeichen folgen, die zum Variablennamen gehören können. Wir empfehlen daher, immer die Schreibweise mit den geschwungenen Klammern zu verwenden.

Es gibt einige vordefinierte Shell-Variable. Die wichtigsten sind in Tabelle 1.1 zusammengefasst, und werden im Folgenden erklärt:

Variable	Wert
`HOME`	Homedirectory
`PATH`	Suchpfad für Programme
`PS1`	Prompt (meist `$`)
`PS2`	2. Prompt (meist `>`)
`IFS`	"Internal field separators"
`USER`	Benutzername
`$`	Prozessnummer der Shell (für temp-files!)
`?`	Letzter Return-Wert

Tabelle 1.1: Vordefinierte Shell-Variablen

`HOME` enthält den Namen des Homedirectories. Das ist jenes Directory, indem man sich nach dem Einloggen befindet, und in das man mit dem Kommando `cd` (ohne Argumente) zurückkehrt.

`PATH` enthält die Liste jener Directories, in denen die Shell nach ausführbaren Programmen sucht. Diese Liste ist durch Doppelpunkte getrennt. Ein typisches Beispiel wäre etwa:

```
$ echo ${PATH}

/bin:/usr/ucb:/usr/bin:/usr/users/01/s8225607/bin
```

Was das bedeutet, lässt sich an einem Beispiel einfach zeigen: Wenn der Benutzer das Programm `col` aufruft, dann testet die Shell der Reihe nach, ob folgende Programme existieren:

```
/bin/col
/usr/ucb/col
/usr/bin/col
/usr/users/01/s8225607/bin/col
```

Das erste der gefundenen Programme wird ausgeführt. Wenn keines der angeführten Programme existiert, dann liefert die Shell eine Fehlermeldung zurück:

```
$ hgfd
hgfd: not found
```

Die Variable `PS1` enthält jene Zeichenkette, die als Prompt angezeigt wird, wenn die Shell auf die Eingabe eines Kommandos wartet (meist `$␣`). Die Variable `PS2` enthält den zweiten Prompt (meist `>␣`). Dieser wird angezeigt, wenn die Eingabe des Kommandos noch nicht vollständig ist:

```
$ echo "erste zeile
> zweite zeile"
erste zeile
zweite zeile
```

Die Variable `IFS` (internal field separators) enthält die Liste der Zeichen, die von der Shell als Trennzeichen erkannt werden. Üblicherweise sind das die Zeichen Space, Tab und Newline. Das Programm 'which', das in Abschnitt 1.5 vorgestellt wird zeigt, wie man diese Variable umdefiniert.

Die Variable `USER` enthält den Usernamen des Benutzers[2]. Mit ihrer Hilfe kann man Shellprogramme, die den Namen des Benutzers benötigen so programmieren, dass sie von mehreren Benutzern ohne Änderungen verwendet werden können.

Die Variable `$` (angesprochen durch `${$}` oder `$$`) enthält die Prozessnummer (pid) der Shell. Sie kann dazu verwendet werden, eindeutige Filenamen zu erzeugen. Mehr dazu ist im Abschnitt 1.5 nachzulesen.

1.2.3 Expansion von Datei- und Kommandonamen

Es ist nicht notwendig alle Datei- und Kommandonamen in der Shell auszuschreiben. Viel schneller arbeitet man mit der Shell wenn man Namen von der Shell expandieren lässt.

Beispiel: Durch Eingabe von

```
$ z
```

gefolgt von der Tabulator Taste wird die Shell mit einem Signalton reagieren. Durch erneutes Drücken der Tabulator Taste versucht die Shell die bisher eingegebenen Zeichen eindeutig einem Datei- oder Kommandonamen zuzuordnen. Dazu durchsucht sie das aktuelle Verzeichnis und alle in der Shell-Variablen $PATH angegebenen Verzeichnisse nach entsprechenden Einträgen. Dabei werden zwei Fälle unterschieden.

[2] In manchen UNIX-Versionen heisst diese Variable `LOGNAME`.

- Falls mehrere Ergänzungen möglich sind, so listet die Shell alle Varianten am Bildschirm auf.

  `$ z`

  gefolgt durch zweimaliges Betätigen der Tabulator Taste ergibt folgenden Output:

  ```
  zcat zdiff zgrep zipcloak ...
  ```

- Wenn der Name eindeutig aufgelöst werden kann, ergänzt die Shell selbstständig.

  `$ zd`

  wiederum gefolgt durch zweimal Tabulator ergibt:

  ```
  zdiff
  ```

Dieser Mechanismus funktioniert auch bei Shell-Variablen und Home-Verzeichnissen:

`$ $PA`

ergibt:

```
$PATH
```

`$ ls ro`

ergibt:

```
ls  root/
```

1.2.4 Alias

Zur Abkürzung immer wiederkehrender Kommandofolgen lassen sich für diese sogenannte Aliasse definieren.

Mit dem Kommando `alias` definiert man einen Alias und mit `unalias` entfernt man ihn wieder entfernen. Aliasse existieren bis zum Löschen dieser oder bis zum beenden der aktiven Shell. Möchte man einen Alias permanent einrichten, trägt man die entsprechende Befehlszeile in die Datei *.profile* in seinem Home-Verzeichnis (*~/.profile*) ein.

`$ alias ll="ls -l"`

Wenn man nun `ll` tippt, wird in Wirklichkeit `ls -l` ausgeführt.

Der Aufruf von `alias` ohne Argumente bewirkt eine Auflistung aller definierten Aliasse.

`$ alias`

```
ll="ls -l"
```

```
cdproj1="cd /home/e0326432/proj1/"
```

1.2.5 Ein- und Ausgabeumleitung

Die Ausführung eines mit dem Kommando verbundenen Programmes nennen wir einen *Prozess*. Ein Prozess kommuniziert mit dem Benutzer über drei Kanäle:

- Die Standardeingabe (*stdin*, Nummer 0)
- Die Standardausgabe (*stdout*, Nummer 1)
- Die Standardfehlerausgabe für Fehlermeldungen (*stderr*, Nummer 2)

Es gilt folgende Konvention, die beim Schreiben von Kommandos zu beachten ist:

> *Argumente steuern den Ablauf der Verarbeitung. Fehlermeldungen sind immer auf die Standardfehlerausgabe zu schreiben. Wenn durch Optionen oder Argumente nichts anderes angegeben ist, sollen die zu verarbeitenden Daten von der Standardeingabe gelesen werden, und die verarbeiteten Daten auf die Standardausgabe geschrieben werden.*

Das System überprüft nicht, ob ein Verstoß gegen diese Regel vorliegt. Jeder Verstoß aber macht die Bedienung des Systems weniger flexibel und auch viel unübersichtlicher.

Es kann vorkommen, dass die Eingabe an einen Prozess doch nicht vom Benutzer, sondern von einem File erfolgen soll. Dies nennt man *Redirection* oder *Umleitung* und wird durch den Operator $<$ erreicht.

```
$ Kommando < infile
```

bewirkt, dass die Eingabe nicht vom Benutzer (über die Tastatur), sondern vom File *infile* erfolgt. Die Redirection wird von der Shell durchgeführt *bevor* das Kommando gestartet wird. Die Redirection-Operatoren und ihre Argumente scheinen auch nicht mehr in der Argumentliste auf, die dem Kommando übergeben wird.

→ *Sie müssen sich als Programmierer nicht um die Redirection kümmern.*

In analoger Weise gibt es den Operator $>$ zum "Umlenken" der Ausgabe.

```
$ Kommando > outfile
```

```
$ Kommando < infile > outfile
```

bewirken, dass die Ausgabe statt an den Benutzer auf das File outfile geschrieben wird. Dies gilt nicht für Fehlermeldungen!

Fehlermeldungen werden auf die Standardfehlerausgabe (Deskriptor Nr. 2) geschrieben. Die Umleitung der Fehlermeldungen eines Prozesses erfolgt mittels

```
$ Kommando 2> Fehlerfile
```

Soll die Ausgabe eines Prozesses an ein File *angehängt* werden, ist der Operator $>>$ zu verwenden.

Soll die Ausgabe eines Kommandos, die normalerweise auf die Standardausgabe ginge, auf die Standardfehlerausgabe umgeleitet werden, so kann dazu die Konstruktion

```
$ Kommando 1>&2
```

verwendet werden: Damit werden die Ausgaben, die auf Deskriptor 1 (also die Standardausgabe) erfolgen auf den Deskriptor 2 (also die Standardfehlerausgabe) umgeleitet. Auf diese Art sind beliebige Umleitungen möglich.

1.2.6 Pipes

Kommt es vor, dass die Ausgabe eines Prozesses die Eingabe für einen anderen Prozess sein soll, benötigt man in UNIX kein Zwischenfile, sondern verwendet die sogenannte *Pipeline*, gekennzeichnet durch '|':

$\$$ $Kommando_1 \mid Kommando_2 \mid ... \mid Kommando_n$

bewirkt die Ausführung von $Kommando_1$ (Eingabe vom Benutzer). Die Ausgabe von $Kommando_1$ ist gleichzeitig Eingabe für $Kommando_2$, das parallel zu dem $Kommando_1$ ausgeführt wird usw. Erst $Kommando_n$ liefert die Ausgabe an den Benutzer.

Während der Ausführung eines Kommandos kann der Benutzer keine weiteren Aktionen durchführen. Um ein Kommando abzubrechen, kann man `^C` (`^C` kann man durch Eingabe von STRG + C erzeugen) eintippen. Dadurch wird vom Betriebssystem an den gerade laufenden Prozess ein Signal geschickt, das den Prozess abbricht. Wenn man allerdings die Ergebnisse braucht, muss man auf die Beendigung des Kommandos warten.

Durch den Operator '**&**' wird dies vermieden:

```
$ Kommando &
```

bewirkt die Ausführung des Kommandos "im Hintergrund". Die Shell gibt die sogenannte *Prozessnummer (pid)* des Prozesses aus und erwartet sofort die nächste Eingabe. Der Benutzer kann ganz normal weiterarbeiten und muss sich nur von Zeit zu Zeit vergewissern, ob der Prozess schon beendet ist oder nicht[3]. Allerdings sollte ein Hintergrundprozess nicht direkt mit dem Benutzer kommunizieren und sich auch nicht auf Daten, die gerade vom Benutzer bearbeitet werden, beziehen. Ein Hintergrundprozess kann keine Daten vom Terminal lesen: Jeder Versuch, von der Standardeingabe zu lesen, ohne dass diese mit einem File oder einem anderen Prozess verbunden wurde, liefert sofort EOF (end of file) oder stoppt den Hintergrundprozess. Um einen im Vordergrund aktiven Job im Hintergrund fortzusetzen muss man ihn zuerst suspendieren indem man das entsprechende Signal `^Z` (STRG + Z) schickt. Dabei gibt die Shell die Job Nummer des gerade suspendierten Jobs auf der Konsole aus. Dann kann man den Prozess mit dem Kommando `bg %n` im Hintergrund fortsetzen.

[3] Die Bourne-Again-Shell meldet von selbst, wenn ein Hintergrundprozess terminiert, falls notify gesetzt wurde.

`n` Entspricht hier der Jobnummer. Um einen im Hintergrund befindlichen Job in den Vordergrund zu holen benutzt man das Kommando `fg %n`. Um die Jobnummer `n` herauszufinden verwendet man am besten `jobs`.

Einen Hintergrundprozess kann man nicht mit `^C` beenden. Stattdessen verwendet man das Kommando `kill`:

`$` *kill pid*

wobei *pid* die Prozessnummer des Hintergrundprozesses ist. Dieses Kommando schickt das Signal `SIGTERM` an den Prozess. Dieser hat die Möglichkeit, auf das Signal zu reagieren, indem er die Bearbeitung unterbricht, eventuelle temporär angeforderte Ressourcen wieder frei gibt und terminiert.

Manche Prozesse sind vom Programmierer gegen das `SIGTERM`-Signal geschützt worden. In diesem Fall kann man den Prozess mit dem Kommando

`$` *kill -KILL pid*

beenden. Dabei erhält der Prozess allerdings keine Möglichkeit mehr zum Aufräumen. Man sollte diese Möglichkeit daher sparsam verwenden.

Eingabevervielfältigung

Mit dem Kommando `tee` kann man die Standardeingabe vervielfältigen. Beispiel:

`$` *ls -l | tee datei1*

Zeigt den aktuellen Verzeichnisinhalt auf der Standardausgabe an und speichert ihn in *datei1*. Das bedeutet die Eingabe für das Kommando nach der Pipe wird vervielfältigt, einmal für die Ausgabe auf dem Bildschirm und einmal um in *datei1* geschrieben zu werden.

1.2.7 Anführungszeichen

Es gibt drei Arten von Anführungszeichen:

- Doppelte Anführungszeichen (double quotes, `"`).
- Einfache Anführungszeichen (single quotes, `'`, ‘von links unten nach rechts oben’).
- ‘Verkehrte’ Anführungszeichen (grave quotes, `` ` ``, ‘von links oben nach rechts unten’).

Die drei Arten haben in der Shell unterschiedliche Bedeutung:

1. **Double Quotes**: Fassen mehrere Wörter zu einem String zusammen. Variablensubstitution innerhalb des Strings findet statt, Generierung von Filenamen findet *nicht* statt.

2. **Single Quotes**: Fassen ebenfalls mehrere Wörter zu einem String zusammen. Weder Variablensubstitution, noch Generierung von Filenamen findet statt. Das folgende Beispiel verdeutlicht den Unterschied zwischen:

   - Keine Anführungszeichen (Zeile 1)
   - Doppelte Anführungszeichen (Zeile 2)
   - Einfache Anführungszeichen (Zeile 3)

   Nehmen Sie dazu im Folgenden an, dass im momentanen Directory die Files `a.c` und `b.c` vorhanden sind und dass der Benutzer `alex` mit dem System arbeitet.

   ```
   $ echo ${USER} *
   alex a.c b.c
   $ echo "${USER} *"
   alex *
   $ echo '${USER} *'
   ${USER} *
   ```

3. **Grave Quotes**: Dienen der sogenannten *Kommandoersetzung (Command Substitution)*. Sie werden wie folgt verwendet:

   ‘Kommando‘

   Bei der Auswertung wird zuerst das Kommando exekutiert, das zwischen den Grave Quotes steht. Dann wird das gesamte Konstrukt, inklusive der Grave Quotes, durch die Ausgabe dieses Kommandos ersetzt. Dann wird der Rest des Kommandos ausgeführt. Dazu ein Beispiel:

   ```
   $ echo ‘ls‘
   a.c b.c
   ```

   Im ersten Schritt wird das Kommando `ls` ausgeführt. Dann wird das gesamte Konstrukt `` `ls` `` durch seine Ausgabe (die Liste `a.c b.c`) ersetzt. Schlussendlich wird das so entstandene Kommando `echo a.c b.c` ausgeführt.

   Üblicherweise wird die Kommandoersetzung in Verbindung mit einer Zuweisung verwendet. Um etwa der Variablen `FILES` die Liste aller Files im momentanen Directory zuzuweisen, wird man folgendes Kommando verwenden:

   ```
   $ FILES=‘ls‘
   ```

1.2.8 Mehrere Kommandos

Die Shell unterstützt mehrere Möglichkeiten, Kommandos nacheinander und in Abhängigkeit voneinander zu starten.

```
$ ls; date
```

Die Kommandofolge führt zunächst das Kommando `ls` aus und zeigt dann das aktuelle Datum an.

```
$ ls; date > datei1
```

In *datei1* steht das aktuelle Datum.

```
$ (ls; date) > datei1
```

In *datei1* stehen nun der Verzeichnisinhalt und das aktuelle Datum. Die Klammerung bewirkt die Ausführung der eingeschlossenen Kommandos in derselben Shell, so dass diese ein Ergebnis zurückliefern. Nachfolgende Tabelle 1.2 fasst die weiteren Möglichkeiten zusammen.

Kommandofolge	Wirkung
com1; com2	Führt die Kommandos hintereinander aus
com1 && com2	Führt `com2` nur aus, wenn `com1` erfolgreich war
com1 || com2	Führt `com2` nur aus, wenn `com1` einen Fehler liefert
com1 &	Führt das Kommando als Hintergrundprozess aus
com1 & com2	Startet `com1` im Hintergrund und `com2` im Vordergrund
(com1; com2)	Startet beide Kommandos in einer Shell

Tabelle 1.2: Hintereinanderausführung von Kommandos

1.3 Das Dateisystem

UNIX ist ein "fileorientiertes" Betriebssystem, der zentrale Datentyp ist das File. Ein File ist eine unstrukturierte Zeichenfolge.

1.3.1 Dateien und Verzeichnisse

Files werden in UNIX in *Directories* (Fileverzeichnissen) zusammengefasst. Ein Directory kann beliebig viele Verweise auf andere Files enthalten; im Sinne der Strukturierung sollte man jedoch ab etwa 20 Directoryeinträgen Subdirectories anlegen. Die Namen von Files können 14 Zeichen lang sein, in vielen Versionen sogar bis zu 255 Zeichen lang.

Ausgehend vom *Rootdirectory* (mit dem Namen /, gesprochen "slash") verzweigt sich das UNIX Filesystem baumartig.

Die Selektion eines einzelnen Eintrages aus einem Directory erfolgt ebenfalls mit /. So enthält das Rootdirectory ein Subdirectory usr, das mit

```
/usr
```

(sprich “slash user”) angesprochen wird.

Enthält das Directory usr ein Subdirectory users, so kann dieses mit

```
/usr/users
```

angesprochen werden, usw.

Jeder Benutzer hat ein sogenanntes *Homedirectory*, in dem er beliebig Files und Subdirectories anlegen kann. Beim Login-Vorgang wird dem Benutzer automatisch sein Homedirectory als sogenanntes Arbeitsdirectory (*working directory*) zugewiesen, für den Benutzer *s8225607* z.B.

```
/usr/users/01/s8225607
```

Ein File bsp1.c im Arbeitsdirectory muss nicht mit dem vollen Namen

```
/usr/users/01/s8225607/bsp1.c
```

angesprochen werden, sondern kann auch (kürzer) mit

```
bsp1.c
```

angesprochen werden. Analoges gilt für Subdirectories: Enthält das Arbeitsdirectory ein Subdirectory cprog und enthält dieses Directory ein File bsp2.c, so kann dieses File sowohl mit

```
/usr/users/01/s8225607/cprog/bsp2.c
```

als auch mit

```
cprog/bsp2.c
```

angesprochen werden. Allgemein gilt, dass Namen, die *nicht* mit / beginnen, sich auf das augenblickliche Arbeitsdirectory beziehen. Diese Namen werden *relative Pfadnamen* genannt. Namen, die mit / beginnen, heißen *absolute Pfadnamen*.

Das Arbeitsdirectory kann mit cd verändert werden: Mit

```
$ cd cprog
```

wird /usr/users/01/s8225607/cprog zum Arbeitsdirectory, und das File bsp2.c wird einfach mit

```
bsp2.c
```

angesprochen. Dieses Verfahren funktioniert natürlich über beliebig viele Ebenen von Subdirectories hinweg.

Mit

```
$ cd ..
```

gelangt man wieder eine Ebene in Richtung Rootdirectory. Tatsächlich gibt es in jedem Directory standardmäßig zwei Einträge, nämlich ".." und ".". Ersterer bezieht sich immer auf das übergeordnete Directory, "." bezeichnet das augenblickliche Directory. cd ohne Argument bringt Sie in Ihr Homedirectory zurück.

1.3.2 Zugriffskontrolle

Jedes File ist in UNIX durch einen Schutzmechanismus vor Fremdzugriffen schützbar. Die Zugriffsrechte werden im Folgenden erklärt:

- Bezogen auf die Zugriffsrechte lässt sich die Gesamtheit der UNIX-Benutzer in drei Klassen unterteilen:
  - den Besitzer eines Files (**u***ser*), Üblicherweise ist das der, der das File angelegt hat.
  - die 'Gruppe' eines Files (**g***roup*). Für jedes Files ist eine Gruppennummer gespeichert – die 'Gruppe' des Files sind jene Benutzer, die sich in dieser Benutzergruppe befinden.
  - alle übrigen Benutzer (**o***thers*).
- Man kann auf ein File auf drei verschiedene Arten zugreifen:
  - Lesen (**r***ead*)
  - Schreiben (**w***rite*)
  - Ausführen (*e***x***ecute*)
- Jeder Benutzer in einer dieser drei Klassen kann prinzipiell auf ein File auf jede der drei Arten zugreifen. Ob der Zugriff von Benutzern einer Klasse auf ein Objekt in einer bestimmten Art tatsächlich möglich ist, wird über die Zugriffskontrolle entschieden.

Die Zugriffsrechte auf ein File können mit dem Kommando `ls -l` angezeigt werden. Dieses Kommando liefert zu jedem File (unter anderem) einen String der Bauart "`drwxrwxrwx`", wobei jedes dieser Zeichen auch "–" sein kann. Das erste Zeichen gibt an, ob das File ein normales File ist (`-`), oder ein spezielles File: Directories sind mit `d` gekennzeichnet, Sockets mit `s`, Symbolic Links mit `l`, Pipes mit `s`, Devices mit `c` oder `b` ('Character Special' und 'Block Special').

Die restlichen neun Zeichen zerfallen in drei Gruppen zu je drei Zeichen: Die erste Dreiergruppe gibt die Lese-, Schreib- und Ausführberechtigungen für den Eigentümer des Files an, die daraufolgende Dreiergruppe die Berechtigungen für die Gruppe, und die letzte Dreiergruppe die Berechtigungen für alle anderen Benutzer.

Für Directories haben die Zugriffsberechtigungen eine etwas andere Bedeutung:

- Lesen bedeutet, dass die Liste der Einträge gelesen werden darf (z.B. mit ls).

- Schreiben heißt jede Art von Veränderung im Directory, also das Erzeugen eines neuen oder das Löschen eines alten Files.

- Exekutieren heißt hier, dass das Directory als Arbeitsdirectory verwendet werden darf (mittels cd), und dass einzelne Einträge selektiert werden dürfen. Man darf aber die Liste der Einträge nicht lesen (ls ist nicht möglich).

Die Zugriffsberechtigungen können mit dem Kommando `chmod` geändert werden. Unglücklicherweise ist es nicht möglich, die Zugriffsberechtigungen in derselben symbolischen Form zu setzen, in der sie von `ls -l` ausgegeben werden – es stehen aber zwei andere Möglichkeiten zur Verfügung:

Die erste Möglichkeit ist, sich die neun möglichen Zugriffe als die Bits einer dreistelligen Oktalzahl vorzustellen:

Zum Glück sind nur wenige der 512 prinzipiell möglichen Kombinationen von Zugriffsrechten wirklich sinnvoll. Es ist daher möglich, sich die entsprechenden Oktalzahlen recht einfach zu merken. Meist findet man mit drei Kombinationen das Auslangen:

- `-rw-r--r--` = 644

  Diese Berechtigung wird üblicherweise für Files vergeben, die keiner Geheimhaltung bedürfen.

- `-rwxr-xr-x` = 755

  Diese Berechtigung wird für ausführbare Files vergeben, die keiner Geheimhaltung bedürfen.

- `-rw-------` = 600

  Diese Berechtigung wird für Files vergeben, die nur für den Benutzer zugänglich sein sollen.

Die zweite Möglichkeit, die Zugriffsberechtigungen zu vergeben ist oft einfacher anzuwenden. Dabei wird der gewünschte Zugriffsmodus wie folgt dargestellt:

- *Wer:* Eine beliebige Kombination von **u**ser, **g**roup und **o**thers, dargestellt durch den jeweiligen (hier fettgedruckten) Anfangsbuchstaben.

- *Operator:* Eines der Zeichen +, – oder =. + heißt, dass die entsprechende Berechtigung zu den bisherigen dazu gegeben werden soll, – heißt, dass sie gelöscht werden soll. = wird verwendet, um genau die angegebenen Berechtigungen zu vergeben und alle anderen zu löschen.

- *Berechtigung:* Eine beliebige Kombination der Buchstaben `r`, `w` und `x`, mit der bereits bekannten Bedeutung.

Dazu einige Beispiele:

- chmod u+x foo gibt die Ausführberechtigung für den Eigentümer zu den sonstigen Berechtigungen dazu.
- chmod g-r bar nimmt allen Mitgliedern der Gruppe des Files die Leseberechtigung.
- chmod o-rwx baz nimmt allen 'sonstigen' Benutzern alle Berechtigungen an diesem File.

Die Manualpage für das Kommando `chmod` gibt dafür noch weitere Beispiele an. Es ist wichtig, sich zu merken, dass alle Zugriffsberechtigungen nur vor dem unbefugten Zugriff durch nicht privilegierte Benutzer schützen. Der Systemadministrator hat Zugang zu *allen* Files.

→ *Speichern Sie keine streng vertraulichen Daten unter Ihrem Account.*

umask

Beim Anlegen einer neuen Datei werden die Zugriffsrechte mit einer Voreinstellung besetzt welche mit dem Befehl welche die Dateischutzattribute *ausschalten*. Die Standardbelegung ist üblicherweise `umask 027`, welche dem Besitzer alle Rechte und der Gruppe das Lese- und Ausführrecht einräumt – alle anderen haben keinen Zugriff auf die Dateien.

Bei gemeinsamen Arbeiten an Dateien innerhalb einer Gruppe macht es manchmal Sinn die umask auf 002 zu setzen, um den Gruppenmitgliedern Schreibrechte auf erstellte Files zu geben.

1.3.3 Erweiterte Konzepte der Zugriffskontrolle

Neben den Zugriffsberechtigungen von denen wir im vorigen Kapitel gelesen haben, gibt es auch noch andere als *nur* "`rwx`".
Um dieses erweiterte Konzept besser verstehen zu können, sehen wir uns zunächst eine Darstellung der erweiterten Zugriffsreche an:

Hier sehen wir das die Zugriffsrechte durch 4 Bitfelder dargestellt werden. Da das Kommando `ls -l` aber nur die letzten 3 Bitfelder (*ugo*) anzeigt, werden die erweiterten Rechte (*sst*) anstelle der *rwx* flags angegeben sofern sie gesetzt sind.
Das sieht dann z.B.: so aus:(`chmod 1777 datei`): `-rwxrwxrwt`

t-Bit, SUID und SGID

- **t-Bit**
  Das *t-Bit* (manchmal auch "sticky Bit" genannt) hat nur in Verbindung mit Verzeichnissen

eine Bedeutung. Normalerweise (ohne *t-Bit* auf dem Verzeichnis) können Dateien von jeder Person, die Schreibrechte auf das Verzeichnis hat, gelöscht werden. Das *t-Bit* ändert diese Regel. Ist es gesetzt, so können nur der Eigentümer der Datei (und der Verzeichniseigentümer) eine Datei löschen. Das *t-Bit* wird beispielsweise durch die Befehle "`chmod a+tw`" oder "`chmod 1777`" gesetzt.

- **s-Bit** Set-User-ID (SUID)
  Unter Linux hat jeder Benutzer eine Kennung (User-ID). Diese Kennung ist wichtig für die Zugriffsrechte auf Ressourcen (Dateien, etc ...). Man unterscheidet 2 User-IDs.
  - real User-ID (*echte Kennung*)
  - effective User-ID (*wirksame Kennung*)
- **s-Bit** Set-Group-ID (SGID)
  Das *s-Bit* kann man auch auf die Gruppe setzen. Hier werden zwei Fälle unterschieden (*Ausführbare Dateien* und *Verzeichnisse*):
  1. Ausführbare Dateien die das *s-Bit* auf die Gruppe gesetzt haben, laufen unter der Gruppen-ID des Dateieigentümers. Das ist analog zu dem, was wir oben gesehen haben, wenn das *s-Bit* auf den Eigentümer gesetzt war.
  2. Eine etwas andere Bedeutung bekommt das *s-Bit*, wenn es auf dem Gruppenfeld eines Verzeichnisses gesetzt ist. In diesem Fall wird jede neue Datei mit dieser Gruppe gesetzt gesetzt, wenn diese Datei in dem Verzeichnis erzeugt wird. (Jedoch nur, wenn der erzeugende Benutzer auch in der Gruppe ist)

     Bsp.: User alex gehört zwei Gruppen an. Gruppe `users` *und Gruppe* `sound`*. Seine erste Gruppe ist* `users` *und normalweise werden neue Dateien mit dieser Gruppe erzeugt. Gibt es jedoch ein Verzeichnis, welches zur Gruppe* `sound` *gehört und das s-Bit im Gruppenfeld gesetzt hat, dann werden in diesem Verzeichnis neue Dateien mit der Gruppenzugehörigkeit* `sound` *erzeugt.*

     Die effektive User-ID bestimmt die Dateizugriffsrechte Das *s-Bit* wird durch den Befehl "`chmod u+s`" oder z.B.: "`chmod 4755`" gesetzt.

→ *Programme die mit gesetztem s-Bit ausgeführt werden, sollten so selten wie möglich eingesetzt und sehr vorsichtig geschrieben werden um Sicherheitslücken im System zu vermeiden.*

1.3.4 Links

Mit dem Kommando `ln` kann man einen Verweis (Link) auf eine Datei erzeugen.

Zusätzliche Verweise (hardlinks) unterscheiden sich nicht vom Original. Wenn das Original gelöscht wird, bleibt die Datei erhalten solange noch mindestens ein Verzeichniseintrag existiert.

Softlinks (symbolic links) sind eigene kleine Dateien, die als Inhalt den Pfadnamen der Zieldatei erhalten. Dadurch sind sie flexibler als hardlinks. Man muss aber beachen das der softlink ungültig wird wenn das Ziel gelöscht oder umbenannt wird.

Hardlink:

```
$ ln datei1 datei1.lnk
```

Softlink: (auch: symbolic link)

```
$ ln -s datei2 datei2.lnk
```

1.3.5 Wildcards

Oft ist es notwendig, einem Kommando die Namen aller Files oder einer gewissen Auswahl von Files in einem Directory anzugeben. Die Shell bietet hier die Möglichkeit, mit Hilfe sogenannter Meta-Zeichen (*meta characters*) bestimmte Gruppen von Files auszuwählen:

- Das Zeichen ***** in einem Filenamen steht für eine beliebige (auch leere) Zeichenfolge.
- Das Zeichen **?** in einem Filenamen steht für ein einzelnes beliebiges Zeichen.
- Eine Liste von Zeichen, eingeschlossen in eckige Klammern (**[** und **]**) steht für ein beliebiges Zeichen aus dieser Liste. Die Liste kann eine Aneinanderreihung von beliebigen Zeichen oder ein Intervall von Zeichen der Form **x-y** sein, wobei x und y beliebige Zeichen sind.

Tabelle 1.3 verdeutlicht die Funktionsweise dieser Zeichen. Nehmen Sie dazu an, dass im momentanen Arbeitsdirectory folgende Files vorhanden sind:

```
$ ls
hugo hugo2 prog1.c prog2.c t1 t2 t3 t4 test test.c xyz
```

Name	wird expandiert zu
∗	alle obigen Files
t∗	t1,t2,t3,t4,test und test.c
t?	t1,t2,t3,t4
∗.c	prog1.c, prog2.c und test.c
∗2	t2 und hugo2
pr???.c	prog1.c und prog2.c
t[12]	t1 und t2
t[1-3]	t1,t2 und t3
∗[1-4].c	prog1.c und prog2.c

Tabelle 1.3: Funktionsweise der Meta-Zeichen

Die Interpretation der Meta-Zeichen erfolgt durch die Shell. Für die Kommandos sieht es so aus, als wären alle resultierenden Filenamen einzeln eingegeben worden.

→ *Sie müssen sich als Programmierer nicht um die Generierung von Filenamen kümmern.*

Diese Bequemlichkeit birgt allerdings auch Gefahren: Wenn Sie in einem Directory `rm *` tippen, dann werden *alle* Files gelöscht, die sich in diesem Directory befinden. Wenn sie das wollten, dann

ist es gut. Wenn nicht, dann eher weniger. Vorsicht ist auch beim Kommando `mv` geboten: Ein Kommando wie `mv * *.bak` kann *nicht* verwendet werden, um jedes File umzubenennen! Falls bereits ein File mit dem neuen Namen existiert, so wird es gelöscht.

1.4 Wichtige Kommandos

In diesem Abschnitt wird auf einige wichtige `UNIX` Kommandos genauer eingegangen bzw. eine Kurzbeschreibung präsentiert.

1.4.1 Die Manualpages

Ein großer Teil der Dokumentation des Systems ist am System selbst in sogenannten *Manual Pages* gespeichert. Eine Manualpage kann mit dem Kommando

`$` *man* [*section-number*] <*name*>

angezeigt werden. Das Manual ist in numerierte, sogenannte 'Sections' unterteilt. Folgende Sections sind vorhanden:

1 Commands: Das sind alle Programme, die aus der Shell als Kommandos aufgerufen werden können.

2 System Calls: Das sind Funktionen, die aus C-Programmen aufgerufen werden können und die direkt im Betriebssystemkern implementiert sind.

3 Library Functions: Das sind ebenfalls Funktionen, die aus C-Programmen aufgerufen werden können. Sie sind aber nicht im Betriebssystemkern selbst implementiert.

4 File Formats: Diese Section enthält Informationen über die Formate verschiedener Systemfiles.

5 Miscellaneous Information: Hier stehen Informationen, die sonst nirgends hin gepasst haben.

6 Games: Beschreibt Spiele und amüsante kleine Programme auf dem System.

7 Special Files: Das sind Files, die besondere Bedeutung haben (etwa Devices, die ins Filesystem abgebildet werden).

8 System Maintenance: Hier sind die Kommandos beschrieben, die zur Wartung des Systems notwendig sind.

In jeder Section ist ein Eintrag `intro` vorhanden, der eine Einführung in die Section gibt. So kann man beispielsweise mit dem Kommando

`$` *man 2 intro*

die Einführung über die System Calls abrufen. Manche der Sections sind noch weiter unterteilt: Beispielsweise gibt es die Subsection `3m` für mathematische Funktionen, oder `3s` für die Funktionen der Standard-I/O Library. Wenn die *section-number* im man-Kommando nicht angegeben wird, findet das Kommando den Eintrag mit der niedrigsten Nummer. Oft kann daher eine man-page nur gefunden werden, wenn die *section-number* angegeben wird. Ein Beispiel ist die Library Funktion `getopt`. Da es auch eine Shellprozedur `getopt` (in der Section 1) gibt, muss für die Libraryfunktion `getopt` die Section angegeben werden, also

```
$ man 3 getopt
```

1.4.2 Packen, Entpacken, Archivieren

Dateien mit gzip und bzip2 komprimieren

Um Dateien zu verschicken oder eventuell für längere Zeit zu speichern ist es vorteilhaft diese zu komprimieren damit sie weniger Ressourcen (Speicherplatz, Bandbreite) vebrauchen. Dazu können unter anderem die zwei folgenden Programme verwendet werden:

- `gzip` (GNU zip) ist unter Unix ein weit verbreitetes Komprimierungsprogramm.

  ```
  $ gzip archiv.tar
  ```

  Komprimiert *archiv.tar* und speichert es als *archiv.tar.gz*.

  ```
  $ gunzip archiv.tar.gz
  ```

  Dekomprimiert *archiv.tar.gz* in *archiv.tar*.

- `bzip2` ist ein neueres Komprimierungsprogramm als `gzip` das einen effizienteren Algorithmus verwendet der meist besser komprimiert, allerdings auch langsamer ist. `bzip2` ist auch weniger weit verbreitet als `gzip`.

  ```
  $ bzip2 archiv.tar
  ```

  komprimiert *archiv.tar* und speichert es als *archiv.tar.bz2*

  ```
  $ bunzip2 archiv.tar.bz2
  ```

  dekomprimiert *archiv.tar.bz2* in *archiv.tar*

Beide Programmen komprimieren Dateien einzeln. Dabei wird die Originaldatei durch die komprimierte Datei ersetzt und ein `.gz` bzw. ein `.bz2` an den Dateinamen gehängt. Werden mehrere Dateien als Argumente angegeben, so wird jede Datei einzeln komprimiert und ersetzt.

Um mehrere Dateien in ein Archiv zu packen, ist die Verwendung von Archivierungsprogrammen wie `tar` oder `zip` notwendig.

Archivieren mit tar

Es gibt viele Gründe warum man mehrere Dateien zu einer einzigen Datei zusammenfassen möchte. Zum Beispiel zur Archivierung von Projekten oder um viele kleine Dateien zu verschicken. Zu diesem Zweck verwendet man das Archivierungsprogramm `tar`.

Das Programm `tar` kennt sehr viele Optionen, auf die hier nicht näher eingegangen wird. Stattdessen werden im Folgenden einige nützliche Aufrufe vorgestellt.

Packen:

```
$ tar -cvvf archiv.tar datei1 datei2 datei3 ...
```

Entpacken:

```
$ tar -xvvf archiv.tar
```

Entpacken eines gzip komprimierten Archives:

```
$ tar -xvvzf archiv.tar.gz
```

Entpacken eines bzip2 komprimierten Archives:

```
$ tar -xvvjf archiv.tar
```

Inhalt eines Archives auflisten:

```
$ tar -tvvf archiv.tar
```

Die Originaldateien bleiben beim Erzeugen eines Archivs unangetastet. Bei den obigen Aufrufen werden die relativen Verzeichnispfade und die Zugriffsrechte der einzelnen Dateien im Archiv mit abgespeichert und beim Entpacken wieder hergestellt.

Archivieren und Komprimieren mit zip

`zip` ist ein Archivierungs- und Komprimierungsprogramm das auf vielen Plattformen, wie z. B. Unix, VMS, MSDOS, OS/2, Windows NT, Minix, Atari and Macintosh, Amiga and Acorn RISC OS verfügbar ist. Die Vorteile von `zip` sind die leichte Handhabbarkeit und weite Verbreitung der Methode. Gegenüber `gzip` und `bzip2` weist der Algorithmus von `zip` allerdings eine schlechtere Kompressionsrate auf.

Unter Unix und Linux werden zip-Archive mithilfe der Kommandos `zip` und `unzip` erzeugt bzw. entpackt.

Packen:

```
$ zip archiv.zip datei1 datei2 datei3 ...
```

Packen eines Verzeichnisses einschließlich aller Unterverzeichnisse:

```
$ zip -r archiv.zip verzeichnisname
```

Auflisten der Dateiein in einem zip-Archiv:

```
$ unzip -l archiv.zip
```

Entpacken eines zip-Archivs:

```
$ unzip archiv.zip
```

Die Originaldateien bleiben beim Erzeugen eines Archivs unangetastet. Im zip-Archiv werden die relativen Verzeichnispfade und die Zugriffsrechte der einzelnen Dateien mit abgespeichert und beim Entpacken wieder hergestellt.

1.4.3 Pager (cat, less, more)

Als Pager bezeichnet man ein Programm das immer eine (Bildschirm-)Seite an Informationen darstellt.

`cat` wird oft zum Anzeigen von Dateien zweckentfremdet. Die eigentliche Funktion von `cat` besteht darin, die als Argument übergebenen Dateien aneinandergehängt auszugeben. Wenn man jedoch nur eine Datei angibt, so zeigt `cat` lediglich diese an, wenn der Inhalt der Datei länger als eine Seite ist, sonst wird nur die letzte Seite angezeigt.

Das Programm `less` zeigt seitenweise Dateien an. Mittels `/muster` bzw `?muster` kann in der Datei nach `muster` gesucht werden. Im Text kann man mit der Leertaste seitenweise vorwärts scrollen. `less` beherrscht eine Unmenge an Optionen und Kommandos.

`more` ist ein Programm das `less` sehr ähnlich ist, allerdings können die Cursortasten nicht zur Navigation verwendet werden. Im Text vorwärts scrollt man mit Hilfe der Leertaste, zurück gehts mit der b Taste.

1.4.4 Weitere Kommandos

Shell-Kommandos

Ein Kommando an die Shell hat stets die Form

```
$ name arg_1 ... arg_n
```

Dabei ist *name* der Name des Kommandos (entweder der Name eines direkt interpretierten Kommandos, oder ein gültiger Filename, der das Objektfile angibt, das ausgeführt werden soll). Die arg_i sind beliebig viele (auch 0) Argumente, deren Anzahl, Form und Bedeutung vom jeweiligen Programm abhängt, das sie weiterverarbeitet.

Man kann zwei Klassen von Kommandos unterscheiden:

1. Kommandos, die vom Kommandointerpreter *Shell* direkt ausgeführt werden;
2. Kommandos, die die Ausführung von Programmen bewirken. Diese Programme sind in speziellen Directories abgespeichert.

In der Anwendung besteht zwischen diesen beiden Arten von Kommandos keinerlei Unterschied.

Direkt interpretierte Kommandos sind unter anderem:

break	continue	cd	eval	exec
exit	export	read	set	shift
times	trap	wait		

Kommandos, die weitergeleitet und als Programme exekutiert werden, gibt es viele, im Folgenden werden einige kurz erläutert, die häufig benötigt werden.

apropos Gibt Hinweise auf Manualpages, die ein Schlüsselwort enthalten.

bash Eine `sh`-kompatible Shell, welche auch nützliche Features der Korn- und C-Shell enthält.

cat Hängt Dateien aneinander und gibt sie auf die Standardausgabe aus.

clear Löscht den Bildschirm- bzw Fensterinhalt.

chmod Ändert Zugriffsberechtigungen auf Files.

cmp Vergleicht zwei Files binär.

cp Kopiert Files auf Files oder Files in ein Directory.

cut Schneidet Spalten aus Files.

date Ausgabe von Datum und Uhrzeit.

diff Vergleicht Textfiles.

echo Gibt die als Argument übergebenen Strings auf die Standardausgabe aus.

ed Der zeilenorientierte Editor. Seine Dokumentation ist hauptsächlich wegen der Erklärung der *regular expressions* interessant.

expr Ein Programm zum Auswerten von arithmetischen und logischen Ausdrücken. Vor allem in der Shellprogrammierung interessant.

file Ein Programm, das versucht zu erraten, welchen 'Typ' ein File hat, also ob es sich um Text, ein C-Programm, Binärdaten u.s.w. handelt.

find Sucht Dateien nach Namen -name, Datum -[a,c]time, Größe -size, Typ -type usw. z.B.

```
$ find pfadname -name 'dateiname'
```

durchsucht den angegebenen Pfad mit allen Unterverzeichnissen nach der angegebenen Datei und listet alle Vorkommen auf. Der Dateiname darf auch Wildcards enthalten. Mit der Option -exec ist es auch möglich einen Befehl auf alle gefundenen Dateien anzuwenden: z.B.

`$` *find pfadname -name 'dateiname' -exec* befehl *{}* \;

Die beiden Zeichen {} stehen dabei als Platzhalter für den Namen der gefundenen Datei, \; schließt den Befehl ab.

finger Ein Programm zum Abfragen von Informationen über andere Benutzer.

grep Ein Programm, mit dem es möglich ist, Zeilen aus einem File herauszuschneiden, die einer bestimmten *regular expression* (siehe **ed(1)**) entsprechen bzw. nicht entsprechen.

head `$` *head -n i datei1*

Gibt die ersten i Zeilen von datei1 aus, ohne Angabe von "-n i" werden defaultmässig 10 Zeilen ausgegeben.

id Gibt Informationen zum aktuellen Benutzer und dessen Gruppen aus.

ipcrm Dient zum Löschen eines Semaphores, einer Message Queue, oder eines Shared Memory Segments.

ipcs Gibt die Liste aller Semaphore, Message Queues und Shared Memory Segmente aus, die gerade existieren.

jobs Listet alle von der Shell gestarteten Prozesse auf.

joe Ein einfach zu bedienender Editor.

kill Das Programm mit dem Signale an Prozesse geschickt werden können.

less Befehl zum seitenweisen Anzeigen eines Files (ähnlich wie **more**, aber komfortabler).

ln Befehl zum Anlegen eines Links auf ein File.

lpr Befehl zum Ausdrucken.

ls Befehl zum Anzeigen der Liste der Files in einem Directory.

mail Befehl zum Lesen und zum Verschicken von *electronic mail*. Es ist leider recht unbequem zu bedienen.

make Ein Programm zum Ausführen dateiabhängiger Operationen wie z.B. dem automatischen Neukompilieren von Programmsystemen. Weitere Informationen zu `make` finden Sie im Kapitel 4.1.4.

man Das Programm zum Anzeigen des online Manuals (siehe nächster Abschnitt).

mkdir Das Programm zum Anlegen eines Subdirectories.

more Ein Programm zum seitenweisen Anzeigen eines Files.

mount Zum "einhängen" von Medien in die Verzeichnisstruktur.

mv Ein Programm zum Umbenennen bzw. Verschieben von Files.

passwd Das Programm zur Änderung des Passworts (siehe auch *yppasswd*)

pico Texteditor, der auch vom Mailprogramm **pine** verwendet wird.

pine Ein Programm zum Lesen und zum Verschicken von E-Mails und Postings an Newsgroups.

popd Wechselt in das zuletzt in pushd gespeicherte Verzeichnis.

ps Das Programm zum Anzeigen der Liste der aktiven Prozesse.

pushd Speichert aktuelles Verzeichnis und wechselt in in angegebenes Verzeichnis.

pwd Das Kommando zum Ausgeben des Arbeitsdirectories.

rm Das Programm zum Löschen von Files. *Achtung:* 'undelete' ist *nicht* möglich.

rmdir Das Kommando zum Löschen von (leeren) Directories.

set Zeigt alle Shellvariablen an.

sh Die Bourne-Shell.

shred Zum sicheren Vernichten einer Datei. Die Daten werden mehrmals überschrieben um Wiederherstellungsversuche zu erschweren.

sort Ein Kommando zum Sortieren von Files nach verschiedenen Kriterien.

splint Analysiert ein C-Programm und listet wahrscheinliche Programmierfehler auf

tail wie head, aber es werden die letzten Zeilen ausgegeben.

test Ein Kommando zur Auswertung von diversen logischen Bedingungen. Vor allem in der Shell-Programmierung interessant.

time Startet ein Kommando und gibt nach dessen Beendigung eine Statistik über die Laufzeit des Kommandos aus.

top Listet alle Prozesse auf und aktualisiert per Voreinstellung alle 5 Sekunden.

touch Setzt die Modifikationszeit der als Argument übergebenen Dateien auf die aktuelle Systemzeit. Existiert die angegebene Datei nicht, wird eine leere Datei erzeugt.

```
$ touch datei1 datei2 ...
```

tr Ein Programm zum Ersetzen von verschiedenen Zeichen durch andere Zeichen (kann z.B. dazu verwendet werden, alle Großbuchstaben durch Kleinbuchstaben zu ersetzen).

type Gibt an wie ein Kommando von der Shell interpretiert werden würde.

```
$ type passwd

passwd is /usr/bin/passwd

$ type test

test is a shell builtin
```

ulimit Zum Anzeigen und Ändern von User Limits. Kontrolliert die Ressourcen die einem von der Shell gestarteten Prozess zur Verfügung stehen, auf einem System das die Kontrolle gestattet.

uname Liefert Informationen zum Betriebssystem.

```
$ uname -a

Linux vmars 2.4.18-bf2.4 #1 Son Apr 14 09:53:28 CEST
2002 i686 GNU/Linux
```

Es ist nicht möglich den Namen der Linux-Distribution über uname herauszufinden, diese Information findet sich aber in der Datei /etc/issue:

```
$ cat /etc/issue

Red Hat Linux release 9 (Shrike)
Kernel \r on an \m
```

vi Der Full-Screen Editor `vi` ist mächtig, aber nicht leicht zu erlernen.

w Ein Kommando zum Ausgeben einer Liste von aktiven Benutzern.

wc Ein Kommando zum Zählen der Anzahl der Zeichen, Worte und Zeilen in einem File.

whatis Gibt eine Kurzinformation zu einem Kommando aus.

which Gibt aus, in welchem Directory ein gegebenes Kommando steht.

who Ein weiteres Programm zur Ausgabe einer Liste von aktiven Benutzern.

yppasswd Das Programm zur Änderung des Passworts im Network Information Service (NIS) (siehe auch *passwd*).

xargs Liest Argumente von der Standardeingabe ein und übergibt diese an einen angebenen Befehl: z.B.

```
$ ls | xargs -i befehl {}
```

um die Dateien des aktuellen Verzeichnisses durchzuarbeiten. Gemeinsam mit der Option -i stehen die beiden Zeichen {} als Platzhalter für das jeweilige Argument.

1.5 Shell-Programme

Shell-Programme sind Programme auf Kommandoebene. Sie enthalten Variablen, Konstruktionen zur Ablaufsteuerung sowie Kommandos an das Betriebssystem. Shell-Programme können wie andere Programme direkt durch Angabe des Programmnamens ausgeführt werden.
Es ist zu beachten, daß ein Skript ausführbar und lesbar (Skripts werden von der Shell gelesen und interpretiert) sein muss (zB.: `chmod u+xr`) um es laufen zu lassen. Ein normales Binärprogramm hingegen muss nur ausführbar sein.(zB.: `chmod u+x`)

Parameter

Ein Shell-Programm wird in Quellform in einem File abgespeichert. Der Name dieses Files ist dann der Name eines neuen Kommandos. Die beim Aufruf dieses Kommandos angegebenen Argumente (Parameter) können in dem Shell-Programm weiterverarbeitet werden.

`$1` liefert den Wert des ersten Parameters, `$2` den des zweiten, etc. `$0` liefert den Namen des Kommandos selbst. `$#` liefert die Anzahl der Parameter, `$*` enthält die Parameter `$1`, `$2`,... als String, während `$@` dieselben als Liste enthält. Der Unterschied zwischen diesen beiden Formen tritt selten in Erscheinung. Tabelle 1.4 beinhaltet dazu ein Beispiel.

$-Variable	**Wert**
$0	“say”
$1	“hello”
$2	“world”
$#	“2”
$*	“hello world”
$@	“hello” “world”

Tabelle 1.4: Belegung der Variablen nach *say hello world*

Here-Documents

Wenn in einem Shell-Programm ein Kommando mit konstanter Eingabe aufgerufen wird, so kann diese Eingabe direkt in das Programm geschrieben werden. Man bezeichnet diese Eingaben dann als *Here-Document*. Sie werden mit `<<!` an beliebiger Stelle im Shell-Programm eingeleitet und mit `!` in Spalte 1 einer nachfolgenden Quellzeile abgeschlossen. An Stelle des Rufzeichens kann jede beliebige Zeichenfolge verwendet werden.

Dazu ein Beispiel:

```
$ sort <<!
> one
> man
> clapping
> !
clapping
man
one
$
```

Dieses Konstrukt wird allerdings selten gebraucht.

Variable

Man kann in einem Shellprogramm selbst verständlich auch Shellvariable verwenden. Diese wurden schon im Abschnitt 1.2.2 beschrieben.

Flusskontrolle in Shell-Programmen

Es gibt in der Shell Kontrollflusskonstrukte zur Selektion und Iteration. Eines ist ihnen allen gemeinsam: Sie funktionieren nur dann, wenn die Zeilenumbrüche an den richtigen Stellen stehen. Halten Sie sich daher bei der Shell-Programmierung an das hier gezeigte Layout.

FOR-DO-DONE

```
for name [ in list ]
do
   command-list
done
```

name bezeichnet eine Variable; *list* ist eine (durch die Elemente von `IFS` getrennte) Liste von Strings; wird «in *list*» weggelassen, so entspricht dies «in "$@"». *command-list* ist eine Folge von Shell-Kommandos, die wiederholt ausgeführt wird.

Bei jedem Schleifendurchlauf wird die Shell-Variable *name* auf das nächste Wort innerhalb von *list* gesetzt. Im Sonderfall ohne in *list* wird die Schleife also für jedes Argument einmal ausgeführt.

Beispiel:

Programm

```
for i in one two three
do
     echo ${i}
done
```

Ausgabe

```
one
two
three
```

CASE-ESAC

case *word* in
$pattern_1$) $command - list_1$;;
$pattern_2$) $command - list_2$;;
...
esac

word ist ein String-Ausdruck; die $pattern_i$ sind jeweils beliebige Zeichenmuster, die auch *, ? und [...] enthalten dürfen; die $command - list_i$ sind wiederum Folgen von Shell-Kommandos.

Mit Hilfe von **case** ist eine Mehrfachverzweigung möglich, die auf der Übereinstimmung von Stringmustern basiert. Ausgeführt wird jene Kommandoliste, die zum *ersten* Muster gehört, das mit *word* überein stimmt.

Beispiel:

Programm

```
for i in one zork two three
do
    case ${i} in
         one)    B=1;;
         two)    B=2;;
         three)  B=3;;
         *)      B=unknown;;
    esac

    echo ${B}
done
```

Ausgabe

```
1
unknown
2
3
```

Der Stern als letztes Muster spielt die Rolle eines *catch-all*, das mit jedem String übereinstimmt. Die dazu gehörige Kommandoliste wird also immer dann ausgeführt, wenn das gegebene Wort mit keinem der anderen Muster übereinstimmt.

IF-ELSE-FI

if $command - list_{if}$
then
$command - list_{then}$
[else
$command - list_{else}$]
fi

Diese Konstruktion erlaubt eine normale bedingte Verzweigung. Dazu ein kleiner Einschub:

> Jedes Kommando liefert in UNIX einen Wert (*Exit-Status*). Dieser ist 0, falls kein Fehler bei der Ausführung des Kommandos aufgetreten ist, sonst größer als 0.
>
> Für logische Vergleiche gibt es das Kommando **test**, das das Ergebnis des Vergleichs als seinen Exit-Status mitteilt (TRUE entspricht dabei 0, FALSE einem Wert ungleich 0).

$command - list_{then}$ wird genau dann ausgeführt, wenn das letzte Kommando in $command - list_{if}$ den Exit-Status 0 liefert.

Meistens besteht deshalb $command - list_{if}$ aus einer **test**-Anweisung.

Beispiel:

Programm

```
if test "$1" = "yes"
then
      echo ja
else
      echo nein
fi
```

Ausgabe

liefert ja falls das Shellprogramm mit yes als ersten Parameter aufgerufen wurde, sonst nein

WHILE-DO-DONE, UNTIL-DO-DONE

while $command - list_{while}$
do
$command - list_{do}$
done
until $command - list_{until}$
do
$command - list_{do}$
done

Die Kommandos der Bedingung werden wie bei if-else-fi konstruiert. Die Semantik von while entspricht PASCAL. Die Semantik von until ist die von 'while not', d.h. dass der Schleifentest wie bei while *vor* der Schleife durchgeführt wird.

Kommandoersetzung

Die in Abschnitt 3 erläuterte Kommandoersetzung mit Grave Quotes kann selbstverständlich auch in Shellprogrammen verwendet werden.

WEITERE WICHTIGE SHELL-KOMMANDOS:

break *[n]*	Abbruch der gerade laufenden Schleife. Falls *n* angegeben, Sprung aus der *n*-ten Ebene heraus.
continue *[n]*	Sofortiger Beginn der nächsten Iteration der laufenden Schleife bzw. der *n*-ten umschließenden Schleife.
cd *[dir]*	Das Directory *dir* wird neues Arbeitsdirectory. Wird der Parameter *dir* nicht angegeben, so wird das Homedirectory neues Arbeitsdirectory.
exit *n*	Termination des Programms; *n* ist Exit-Status (0 im fehlerfreien Fall, positiv sonst).
read *name*	Einlesen der Variablen *name* von der Standardeingabe.
shift	Umordnen der Parameterwerte: $1 geht verloren, $1 erhält den Wert von $2, etc. Nützlich, wenn die Parameter der Reihe nach verarbeitet werden sollen.
trap *cmd sigs*	Bei Auftreten eines Signals in *sigs* wird das Kommando *cmd* ausgeführt, und danach die Programmausführung fortgesetzt. Nützlich zum Abfangen von Fehlern oder Unterbrechungen. *Beispiel*: Abfangen von Unterbrechungen vom Terminal aus. `trap 'rm -f ${TMPFILE}; exit 0;' 2 3` Hier wird bei Auftreten eines `SIGINT`-Signals (Signalnummer 2, kann durch Drücken von Control-C erzeugt werden) oder eines `SIGQUIT`-Signals (Signalnummer 3, kann durch Control-Backslash erzeugt werden) das File gelöscht, dessen Name sich in der Variablen `TMPFILE` befindet, und das Programm beendet.
wait *[pid]*	Warte auf Beendigung des Kindprozesses mit der Prozessnummer *pid*. Wenn *pid* nicht angegeben wird, wartet **wait** auf die Beendigung aller Kindprozesse.

Temporäre Files

Die Verwendung temporärer Files ist unter UNIX recht selten nötig. Meistens findet man mit dem Pipeline-Mechanismus und der Kommandoersetzung das Auslangen.

Sollte die Verwendung temporärer Files allerdings einmal unumgänglich sein, so ist Folgendes zu beachten:

- Die temporären Files dürfen *nicht* im momentanen Arbeitsdirectory angelegt werden. Wird diese Bedingung verletzt, dann funktioniert das Programm nicht, wenn es keine Schreiberlaubnis auf das Arbeitsdirectory hat.

  Es empfiehlt sich daher, temporäre Files in einem der Directories `/tmp` oder `/usr/tmp` anzulegen, die genau für diesen Zweck existieren.

- Die temporären Files müssen einen eindeutigen Namen haben, um zu verhindern, dass zwei Prozesse, die das gleiche Programm ausführen, sich gegenseitig stören.

  Das kann erreicht werden, wenn der Filename die Prozessnummer des Prozesses enthält. Auf die Prozessnummer kann ja in der Shell mit der Konstruktion `${$}` (bzw. `$$`) zugegriffen werden.

- Die temporären Files müssen bei der Terminierung wieder gelöscht werden.

Eine gute Möglichkeit, diese Bedingungen zu erfüllen, ist, folgende Gestalt für die Namen von temporären Files zu wählen:

`/tmp/`*Programmname*`.`*Prozessnummer*

Zusätzlich muss man dafür sorgen, dass das temporäre File auch bei Auftritt eines Signals gelöscht wird. Also etwa:

```
TMPFILE=/tmp/foo.$$
...
trap 'echo "Abbruch."; rm -f ${TMPFILE}; exit 0;' 1 2 3 15
catpw | sort > ${TMPFILE}
...
rm -f ${TMPFILE}
exit 0
```

Eine sicherere Methode um temporäre Dateinamen zu generieren ist durch Verwendung des Programms `mktemp` gewährleistet. (siehe "`man mktemp`" für Details)

Beispiele für Shell-Prozeduren

DAS KOMMANDO WHICH

Kommandos befinden sich in UNIX in ganz bestimmten Directories. Der Großteil davon steht in /bin bzw. /usr/bin. Nicht selten kommt es allerdings vor, dass Benutzer eigene Kommandos entwickelt haben und diese lokal in einem eigenen Directory (üblicherweise ${HOME}/bin) gespeichert haben.

Damit die Shell ganz genau weiß, welche Directories überhaupt Kommandos enthalten, gibt es im Shell-Environment eine Shell-Variable PATH, die eine *Liste* aller relevanten Directories enthält. Die einzelnen Einträge in dieser Liste sind jeweils durch einen Doppelpunkt voneinander getrennt (z.B. PATH=/bin:/usr/bin:/usr/ucb).

Bei der Ausführung eines Kommandos durchsucht die Shell sukzessive alle Directories aus dieser Liste. Die Reihenfolge entspricht dabei genau der Reihenfolge der Directory-Einträge in PATH.

Kommen Kommandos mit gleichem Namen in mehreren dieser Directories vor, so wird das zuerst gefundene Kommando ausgeführt.

Das Kommando which gibt zu einem als Argument angegebenen Kommando bekannt, welches aus der Menge der in Frage kommenden Kommandos unter Berücksichtigung von PATH von der Shell ausgeführt wird. Das Ergebnis eines Aufrufs von which ist entweder der vollständige Pfad des entsprechenden Kommandos, oder die Liste aller durchsuchten Directories, falls das Kommando nicht gefunden werden konnte (Abbildung 1.1).

```
#!/bin/sh                                              # (1)
#
#   shell version of WHICH                             # (2)
#

if test $# -ne 1                                       # (3)
then
    echo "usage: $0 cmdname" 1>&2
    exit 1
fi

IFS=${IFS}:                                            # (4)
for i in ${PATH}                                       # (5)
do
     if test -s "${i}/$1"                              # (6)
     then
          echo "${i}/$1"
          exit 0
     fi
done

echo -n "no $1 in "
echo ${PATH} | tr ':' ' '                              # (7)
exit 1
```

Abbildung 1.1: Das Kommando which

Bemerkungen zu Abbildung 1.1

1. `#!/bin/sh` in der ersten Zeile gibt an, dass zum Ausführen des Programms der Interpreter `/bin/sh` zu verwenden ist. Achtung: Dieses Konstrukt *muss* in der ersten Zeile stehen!

   Tipp: Wenn Sie mit der Ausführung eines Programms Probleme haben, geben Sie hier `#!/bin/sh -x` an. Dadurch wird jede Anweisung ausgegeben, bevor sie ausgeführt wird.

2. Texte mit vorangestellter `#` dienen nur als Kommentar und werden nicht ausgeführt.

3. `test $# -ne 1` überprüft, ob die Anzahl der Parameter ungleich eins ist. Wenn ja, wird eine Fehlermeldung auf die Standardfehlerausgabe ausgeben und das Programm mit einem Exitstatus größer als Null terminiert (es ist ein Fehler aufgetreten).

4. IFS ist ebenso wie PATH eine vordefinierte Shell-Variable. IFS steht für "internal field separators" und beinhaltet die Trennzeichen der Shell. Standardmäßig handelt es sich dabei um das Leerzeichen, das Tabulatorzeichen und das Zeilenende. Da in PATH die einzelnen Einträge durch ":" getrennt sind, wird mit der Anweisung `IFS=${IFS}:` der Doppelpunkt als zusätzliches Trennzeichen vereinbart.

5. Die `for` Schleife wird für jedes Element aus der Liste PATH - also für jedes Directory aus dieser Liste durchlaufen.

6. `test -s "${i}/$1"` stellt fest, ob das File `${i}/$1` existiert. `${i}/$1` ist dabei der aus dem gerade betrachteten Directory und dem als Argument übergebenen Kommandonamen zusammengesetzte vollständige Pfadname. Existiert unter diesem Namen ein File, so wurde das Kommando gefunden, sein Name wird ausgegeben und das Programm terminiert mittels `exit 0`. Andernfalls wird mit der Suche im nächsten Directory aus PATH fortgefahren.

7. Dieser Teil gelangt nur dann zur Ausführung, wenn die Suche in den PATH-Directories erfolglos war. In diesem Fall wird die Liste aller durchsuchten Directories, allerdings getrennt durch Leerzeichen, ausgegeben. Daraufhin terminiert das Programm mit einem Exitstatus größer als Null.

Eine Routine zum Schützen von Files vor Fremdzugriffen

Die nun vorgestellte Routine **protect** (Abbildung 1.2) ermöglicht es einem Benutzer, seine Files vor Fremdzugriffen zu schützen. Der Aufruf

`$` *protect file [file ...]*

gewährt dem Benutzer Lese- und Schreibzugriff auf alle angegebenen Files. Bei ausführbaren Files wird zusätzlich die Ausführberechtigung erteilt. Für alle anderen Benutzer (auch die der eigenen Gruppe) wird jeder Zugriff gesperrt.

`$` *protect -g file [file ...]*

gewährt dem Benutzer dieselben Rechte wie oben beschrieben, und gewährt zusätzlich den Mitgliedern der Gruppe das Leserecht und gegebenenfalls das Ausführungrecht.

Bemerkungen

1. Die Shell-Variablen für die Bitmuster der Zugriffsberechtigungen (getrennt nach exekutierbaren und nicht exekutierbaren Files) werden entsprechend der Option gesetzt. Sie werden für die nachfolgenden Aufrufe von `chmod` (zum tatsächlichen Ändern der Zugriffsberechtigungen) benötigt.

2. `shift` verschiebt die Parameter nach hinten, so dass die (bereits behandelte) Option `-g` gelöscht wird und `$1` das erste File-Argument enthält.

3. Die Konstruktion `1>&2` bewirkt, dass die Ausgabe des `echo`-Kommandos, die ja eine Fehlermeldung ist, auf die Standardfehlerausgabe geschickt wird.

4. `for FILE` bewirkt, dass die Schleife für alle Parameter der Reihe nach durchlaufen wird.

5. Der Shell-Variablen TYP wird ein String zugewiesen; und zwar jener, der vom Kommando `file` als Ergebnis geliefert wird. Die Bedeutung der "verkehrten Hochkommas" ist also, dass das umschlossene Kommando an dieser Stelle ausgeführt und sein Ergebnis als String zur Verfügung gestellt wird.

   Das Kommando `file` wird benützt, um den Typ eines Files festzustellen. Handelt es sich z.B. bei dem File *hugo* um ein ASCII file, so lautet die Ausgabe von `file`:

   ```
   hugo: ASCII text
   ```

6. Mit Hilfe der `case`-Kaskade wird nun der von `file` gelieferte und in TYP enthaltene String auf das Vorkommen von bestimmten Schlüsselwörtern untersucht. Enthält TYP beispielsweise den Teilstring ASCII (die '*' in den einzelnen Zweigen der Case-Anweisung stehen für beliebige Strings), so wird das File als nicht exekutierbar eingestuft und der Mode gemäß NOEX gesetzt.

7. Diese Stelle im Programm wird genau dann erreicht, wenn keines der vorherigen Stringmuster mit TYP überein gestimmt hat ('*' steht wie bereits erwähnt für einen beliebigen String).

   Da in diesem Fall nicht entschieden werden kann, ob es sich um ein exekutierbares oder nicht exekutierbares File handelt, wird der Mode des betroffenen Files nicht verändert. Stattdessen wird eine entsprechende Fehlermeldung ausgegeben, die den Programmnamen und den entsprechenden Filenamen enthält.

```
#!/bin/sh
#     protect - set protection bits as required

if test "$#" -gt 0 -a "$1" = "-g"               # (1)
then
    EXEC="0750"
    NOEX="0640"
    shift                                       # (2)
else
    EXEC="0700"
    NOEX="0600"
fi
if test "$#" -eq 0
then
    echo "usage: $0 [-g] file [file ...]" 1>&2  # (3)
    exit 1
fi

for FILE                                        # (4)
do
    if test -d "${FILE}"    # parameter is a directory
    then
         chmod ${EXEC} ${FILE}
    elif test -f "${FILE}"    # parameter is a file
    then
        TYP=`file "${FILE}"`                    # (5)
        case ${TYP} in                          # (6)
            *commands*)
                 chmod ${EXEC} ${FILE} ;;
            *executable*)
                 chmod ${EXEC} ${FILE} ;;
            *ASCII*)
                 chmod ${NOEX} ${FILE} ;;
            *program*)
                 chmod ${NOEX} ${FILE} ;;
            *English*)
                 chmod ${NOEX} ${FILE} ;;
            *data*)
                 chmod ${NOEX} ${FILE} ;;
            *)                                  # (7)
                 echo "$0: Unknown type - ${FILE} not changed" 1>&2;;
        esac
    else        # parameter is neither file nor directory
        echo "$0: ${FILE}: No such file or directory" 1>&2
    fi
done
exit 0
```

Abbildung 1.2: Das Shell-Skript protect

Kapitel 2

Mechanismen von UNIX/Linux

In diesem Kapitel werden die grundlegenden Konzepte von Betriebssystemen am Beispiel von UNIX erklärt. Hierbei wird spezielles Augenmerk auf *Linux* als ein Beispiel eines UNIX Betriebssystems gelegt. Manche Funktionalität ist jedoch in anderen UNIX Derivaten übersichtlicher strukturiert. In diesen Fällen wird bei der Erklärung auf diese Varianten verwiesen. Falls es für das Verständnis der einzelnen Mechanismen notwendig ist, oder wo für Betriebssysteme typische Mechanismen verwendet werden, wird auch auf Implementierungsdetails eingegangen.

2.1 Der Aufbau eines UNIX-Systems

Ein UNIX System besteht aus der Hardware, dem UNIX Kernel, einem oder mehreren Kommandointerpretern (Shells) und Systemkommandos. Applikationsprogramme können entweder direkt auf dem Systemcall-Interface des Kernels aufsetzen, sie können auf einer Shell aufsetzen (*Shell Script*) und sie können auch Systemkommandos verwenden. Die Struktur eines UNIX Systems ist Abbildung 2.1 zu entnehmen.

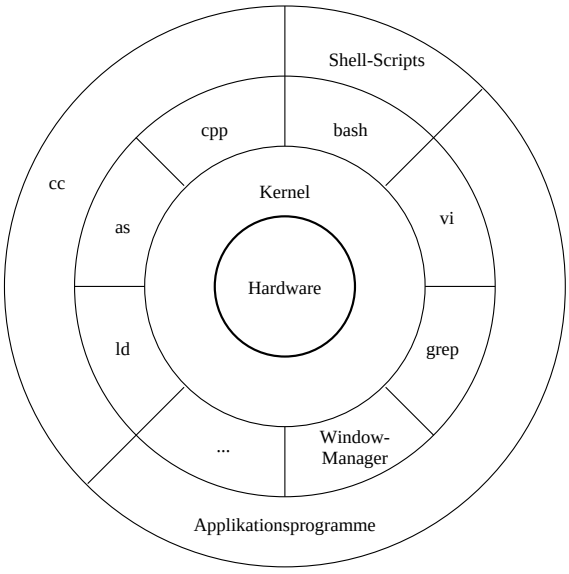


Abbildung 2.1: Struktur eines UNIX Systems

Der Kernel schirmt die Hardware von den Applikations- und Systemprogrammen ab. Er ist dafür verantwortlich, den Programmen eine von der spezifischen Hardware weitgehend unabhängige Sicht des Systems zu geben.

Der Kernel besteht hauptsächlich aus dem Filesystem, der Prozess- und Speicherverwaltung, dem Netzwerkkommunikationssystem und dem Ein-/Ausgabesystem. Der Kernel greift direkt auf die Hardware zu (MMU, Plattenkontroller, Tastaturkontroller, Bildschirmkontroller, ...), und stellt den Programmen Funktionalität auf einem höheren Niveau durch das Systemcall-Interface zur Verfügung (siehe Abbildung 2.2).

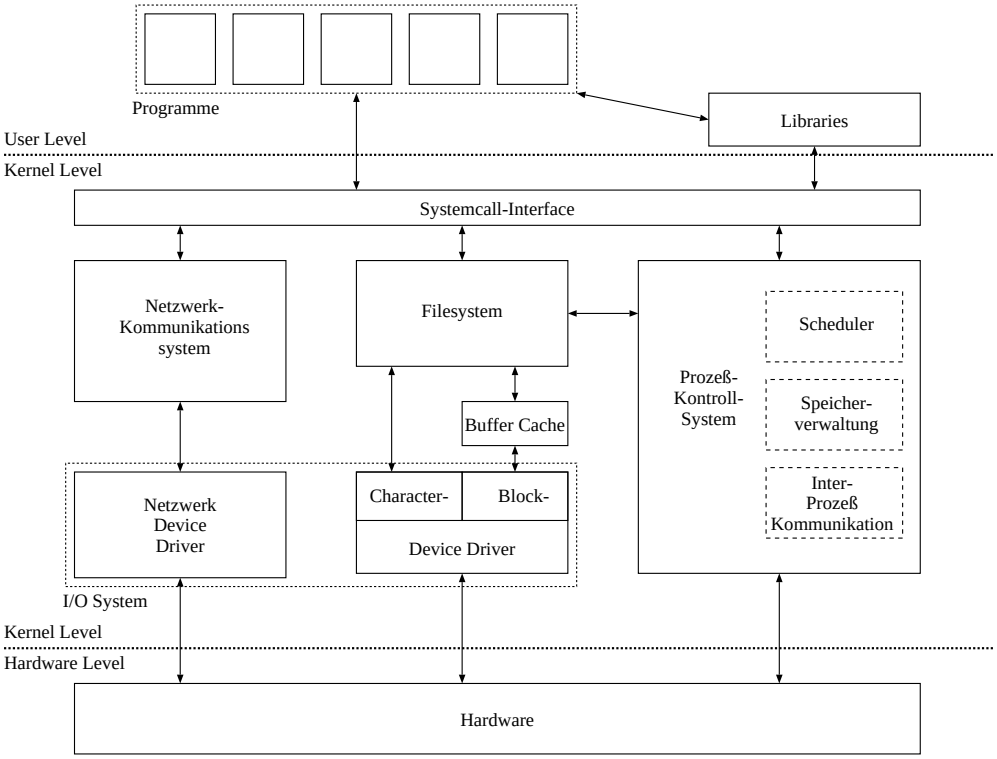


Abbildung 2.2: Struktur eines UNIX-Kernels

Das Betriebssystem ist aufgeteilt in eine Top-Half und eine Bottom-Half. Alle Teile der *Top-Half* werden von Prozessen aktiviert (Systemcalls, Traps), die Prozeduren der *Bottom-Half* durch Interrupts (Clock-Interrupt, Device-Interrupts, ...).

2.2 Filesystem

2.2.1 Struktur des Filesystems von UNIX

Das UNIX Filesystem ist *hierarchisch* organisiert. Es gibt einen Wurzelknoten, die sogenannte "root". Das Root-Directory kann, ebenso wie alle anderen *Directories*, Dateien oder weitere Directories enthalten. Ein Directory ist eine Sonderform eines *Files*, dessen Inhalt vom Betriebssystem verwendet wird, um sich im Filesystem zurechtzufinden und auf Files zuzugreifen. Außer Directo-

ries und normalen Files, die in UNIX nur eine Kette von Zeichen darstellen, die vom System nicht interpretiert werden, gibt es noch eine Reihe von *Special Files*:

Character Special Files dienen dem Zugriff auf zeichenorientierte Geräte, z.B. auf ein zeichenorientiertes Terminal, ein Modem oder auf einen Drucker, aber auch dem Zugriff auf den gesamten, physischen oder virtuellen, Hauptspeicher.

Block Special Files dienen dem Zugriff auf externe Geräte, die Daten in Blocks fixer Größe speichern und wahlfreien Zugriff darauf gestatten. Ein Beispiel dafür ist eine Festplatte.

Named Pipes (oder *FIFOs*) sind Files, die sich nach dem Öffnen wie eine Pipe verhalten, d.h., die Daten, die zuerst hineingeschrieben wurden, werden zuerst wieder gelesen.

Sockets sind Special Files zur Unterstützung der Interprozesskommunikation. Prozesse, die über Sockets miteinander kommunizieren, müssen nicht auf demselben Computer laufen, sondern können auch auf verschiedenen Maschinen exekutiert werden. Die Kommunikation erfolgt dann, für die Prozesse transparent, mit Hilfe eines Protokolls, z.B. TCP/IP.

Symbolic Links sind Verweise auf andere Files; der Symbolic Link und die Datei, auf die er verweist, müssen weder auf derselben Platte noch auf demselben Computer liegen.

Auf einer Platte können sich ein oder mehrere Filesysteme befinden. Da man immer nur Zugriff auf einen Directorybaum hat, müssen andere Filesysteme in diesen Directorybaum eingebunden werden. Erst dann kann man darauf zugreifen. Dieses Einbinden nennt man *Mounting*. Dabei wird ein neues Filesystem in ein bestehendes derart eingehängt, dass das Root-Directory des neuen Filesystems ein Directory des alten Filesystems ersetzt. Files, die sich im alten Directory befanden, sind nach dem Mounten nicht mehr ansprechbar (Abbildung 2.3).

Im Folgenden wird nun genauer auf einige konkrete Filesysteme eingegangen.

2.2.2 Das Virtuelle Filesystem (VFS)

In Linux werden derzeit etwa 20 verschiedene *physikalische Filesysteme* (`ext`, `ext2`, `ext3`, `xia`, `minix`, `umsdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs coda`, `hfs`, `reiserfs`, `ntfs`, `udf`, und `ufs`) unterstützt. Durch die Einführung des sogenannten *Virtuellen Filesystems (VFS)* zwischen dem Betriebssystem und den physikalischen Filesystemen wird von den speziellen Eigenschaften der einzelnen Filesysteme abstrahiert und eine einheitliche Schnittstelle geschaffen. Diese einheitliche Schnittstelle ermöglicht das transparente Mounting von mehreren verschiedenen physikalischen Filesystemen zur gleichen Zeit.

VFS Superblock

Jedes gemountete Filesystem wird durch einen sogenannten *VFS Superblock* repräsentiert. Dieser Superblock enthält unter anderem folgende Informationen:

- Die Device Nummer für das Device, welches das File System benützt.

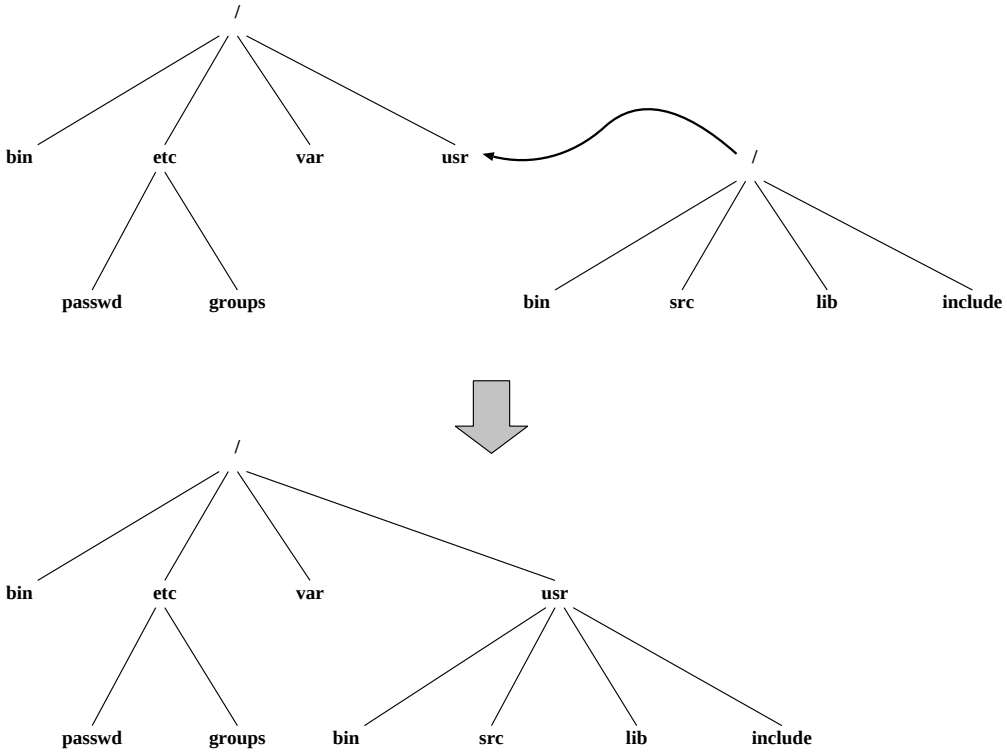


Abbildung 2.3: Mounten eines Filesystems

- Einen Zeiger auf den ersten i-node (wird im folgenden Kapitel erklärt) in dem Filesystem.
- Einen Zeiger auf den i-node des Directories, in das das File System gemounted ist.
- Die Blockgröße des Filesystems.
- Einen Zeiger auf eine Tabelle von Funktionen, die vom VFS benützt werden, um Superblöcke und i-nodes des jeweiligen physikalischen Filesystems zu lesen bzw. zu schreiben.
- Den Typ des Filesystems (`ext2`, `proc`, ...).

VFS i-node

Jedes File des VFS ist durch genau einen i-node repräsentiert. Dieser enthält folgende Informationen zum File:

- Die Device Nummer für das Device, welches das File beherbergt.
- Eine eindeutige i-node Nummer innerhalb des Filesystems. – Die Kombination aus i-node Nummer und Device Nummer ist eindeutig innerhalb des gesamten VFS.
- Den *Typ* des Files (normales File, Directory, special File).

- Die *Zugriffsrechte* auf das File. Diese geben für jede der drei Gruppen (Besitzer des Files, Gruppe des Besitzers und alle übrigen Benutzer) an, ob das File gelesen, beschrieben oder ausgeführt werden darf. Das Set-User-Id Bit gibt bei einem ausführbaren File an, ob das Programm unter den Zugriffsrechten des Besitzers des Files oder unter den Zugriffsrechten des aufrufenden Benutzers ausgeführt wird.
- Die *Anzahl der Referenzen* auf das File. Es ist möglich, dass es mehrere verschiedene Einträge in Directories gibt, die alle auf denselben i-node verweisen (sogenannte Links auf das File). Der Referenzzähler ist beim Löschen von Dateien von Bedeutung. Solange die Anzahl der Links auf eine Datei größer als eins ist, bedeutet das Löschen nur das Entfernen eines Links. Erst wenn der Referenzzähler im i-node null wird, werden auch die Datenblöcke des Files entfernt.
- Den *Besitzer* des Files.
- Die *Gruppe* des Besitzers des Files.
- *Größe* des Files.
- Zeitpunkt des *letzten Zugriffs* auf das File.
- Zeitpunkt des *letzten Schreibzugriffs* auf das File.
- Zeitpunkt der letzten *Änderung des i-nodes* des Files.
- *Anzahl der Blocks* des Files.
- Einen Zeiger auf eine Tabelle von Funktionen, die vom VFS benützt werden, um Operationen auf dem i-node (`inode_operations`-Struktur) und auf dem dem i-node zugeordneten File (`file_operations`-Struktur) durchzuführen. Diese Funktionen sind vom jeweiligen physikalischen Filesystem abhängig.
- *Filesystem-spezifischer Teil*, der zusätzliche Informationen abhängig vom jeweiligen physikalischen Filesystem enthält.

Registrierung eines physikalischen Filesystems

Während des Systemstarts oder beim Laden eines dynamisch ladbaren Filesystemmoduls erfolgt die Initialisierung der physikalischen Filesysteme und deren Registrierung beim VFS.

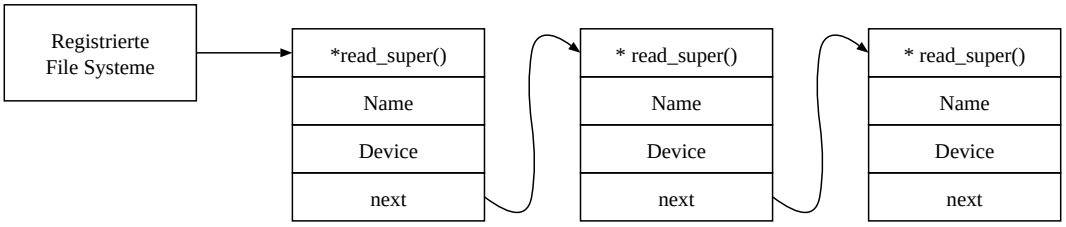


Abbildung 2.4: Registrierte Filesysteme

Jedes registrierte Filesystem ist durch eine Datenstruktur (siehe Abbildung 2.4) repräsentiert, die die folgenden Elemente enthält:

- Einen Zeiger auf eine *Funktion zum Lesen des jeweiligen VFS Superblocks* (`read_super()`).
- Den *Namen* des Filesystems (z.B. `ext2`).
- Ein Flag, welches anzeigt, ob dieses Filesystem ein Device benötigt[1].
- Einen Zeiger auf den *nächsten Filesystemeintrag*.

2.2.3 Das Second Extended Filesystem (EXT2FS)

Nach der Beschreibung des VFS als Abstraktion von den verschiedenen physikalischen Filesystemen im vorherigen Kapitel wird in diesem Kapitel das *Second Extended Filesystem (EXT2FS)* als Beispiel für das unter Linux am häufigsten verwendete physikalische Filesystem behandelt.

Plattenorganisation

Physikalisch ist jede Platte beim EXT2FS in mehrere *Zylindergruppen* unterteilt. Ein Zylinder besteht aus allen übereinanderliegenden Spuren einer Platte, d.h., auf alle Daten innerhalb eines Zylinders kann ohne Bewegung des Lesekopfs zugegriffen werden. Eine Zylindergruppe ist eine Gruppe von nebeneinanderliegenden Zylindern.

Jede Zylindergruppe (siehe Abbildung 2.5) enthält

- eine eigene, redundante Kopie des Superblocks (aus Sicherheitsgründen, da der Superblock kritische Daten enthält),
- einen sogenannten *Zylindergruppenblock*, der aus Einträgen besteht, die die Struktur der einzelnen Zylindergruppen beschreiben.
- i-nodes
- Datenblöcke.

Die Motivation für die Bildung von Zylindergruppen ist, die i-nodes und die Datenblöcke eines Files möglichst knapp beieinander zu halten, d.h. beim Zugriff auf eine Datei möglichst wenige Kopfbewegungen der Platte durchführen zu müssen. Aus Sicherheitsgründen werden die organisatorischen Daten der Zylindergruppen (Superblocks, i-nodes, ...) nicht an den Anfang der Zylindergruppen gelegt, sondern in jeder Zylindergruppe möglichst in verschiedene Sektoren und auf verschiedene Ebenen der Platte gelegt, so dass ein einfacher Hardwarefehler immer nur einen kleinen Teil dieser wichtigen Daten zerstören kann.

[1] Manche Filesysteme benötigen kein physikalisches Device (z.B. das `/proc` Filesystem).

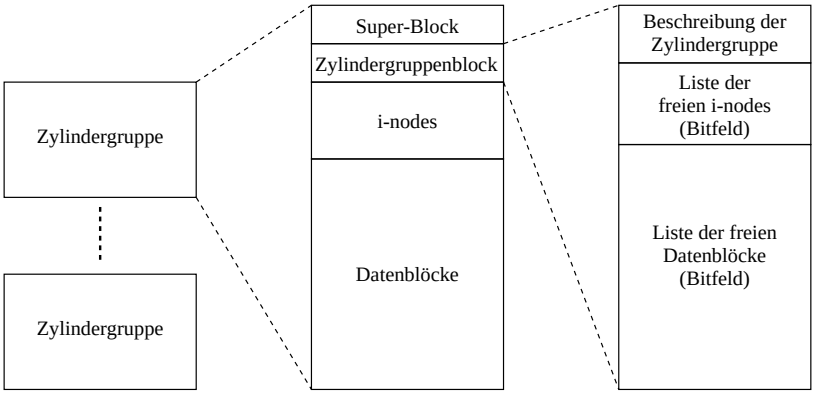


Abbildung 2.5: Plattenorganisation

Auswahl der Blockgröße

Bei der Wahl der Größe eines logischen Plattenblocks gibt es zwei Kriterien, die einander widersprechen:

1. Im Sinne einer Beschleunigung des Datentransfers ist es wichtig, möglichst große Blöcke zu verwenden. Dadurch wird erreicht, dass möglichst viele Daten eines Files in einem kontinuierlichen Vorgang gelesen werden können, und keine zeitraubenden Bewegungen des Lesekopfes dazwischen durchgeführt werden müssen.

2. Je größer die logischen Datenblöcke sind, desto größer ist auch der verschwendete Platz auf der Platte, da im Durchschnitt ein halber Block pro Datei ungenutzt bleibt. Tabelle 2.1 zeigt einen Vergleich des verschwendeten Platzes einer Platte für verschiedene Blockgrößen.

Verbrauchter Platz in MB	**Verschwendeter Platz in %**	**Organisation der Platte**
775.2	0.0	Nur Daten, ohne Zwischenräume
807.8	4.2	Nur Daten, 512-Byte Blöcke
828.7	6.9	Daten + i-nodes, 512-Byte Blöcke
866.5	11.8	Daten + i-nodes, 1024-Byte Blöcke
948.5	22.4	Daten + i-nodes, 2048-Byte Blöcke
1128.3	46.6	Daten + i-nodes, 4096-Byte Blöcke

Tabelle 2.1: Verschwendeter Plattenplatz, abhängig von der Blockgröße (ohne Fragmente)

Dieses Dilemma wurde dadurch gelöst, dass eine sehr große Blockgröße (4096 oder 8192 Bytes) gewählt wurde, und außerdem *Fragmente* eingeführt wurden. Ein Datenblock wird dann entweder ohne weitere Unterteilung einem File zugeordnet, oder er wird in eine Anzahl von Fragmenten vordefinierter Größe (z.B. 1024 Bytes) aufgespalten, die verschiedenen Files zugeordnet werden können. Dadurch, dass nur das Ende einer Datei in einem Fragment liegen kann (wird mehr als ein Fragment für den letzten Teil des Files benötigt, so müssen die entsprechenden Fragmente zusammenhängend

sein), wird der Durchsatz beim Zugriff auf das File nicht beeinträchtigt. Die Platzausnutzung ist jedoch genausogut wie bei einem Filesystem, das nur Blöcke von der Größe eines Fragmentes enthält. Der einzige Nachteil dieser Technik liegt in der etwas höheren CPU Belastung.

Tabelle 2.2 zeigt die Lese- und Schreibgeschwindigkeit in Abhängigkeit von der verwendeten Blockgröße (Filesystemtyp 4096/1024 bedeutet, dass eine Blockgröße von 4096 Bytes und eine Fragmentgröße von 1024 Bytes verwendet wird; Typ 1024 steht für ein Filesystem mit Blöcken der Größe 1024 Bytes ohne Fragmente).

Filesystemtyp	Lesegeschwindigkeit	Schreibgeschwindigkeit
1024	29 KB/s	48 KB/s
4096/1024	221 KB/s	142 KB/s
8192/1024	233 KB/s	215 KB/s

Tabelle 2.2: Performancevergleich verschiedener Filesystemtypen

Allokation von Plattenblöcken

Um die Zugriffs- und Übertragungszeit von Dateien möglichst gering zu halten, ist es notwendig, die physischen Positionen der Blöcke eines Files mit möglichst intelligenten Algorithmen zu bestimmen. Es gibt zwei Arten von Algorithmen:

Globale Algorithmen bestimmen, wo neue Directories und Files angelegt werden; sie bestimmen auch, wie die Blöcke einer Datei am besten auf der Platte angelegt werden. Es wird versucht, die i-nodes aller Files in einem Directory möglichst in derselben Zylindergruppe wie das Directory selbst anzulegen. Weiters wird versucht, die Datenblöcke eines Files in dieselbe Zylindergruppe wie den i-node eines Files zu legen. Die globalen Algorithmen versuchen aber auch, Daten die keinen logischen Zusammenhang haben, möglichst gleichmäßig über die Platte zu verteilen.

Muss ein neues Directory angelegt werden, so wird es in einer Zylindergruppe angelegt, die über möglichst viele freie i-nodes verfügt, und in der noch nicht viele Directories vorhanden sind. Dadurch soll erreicht werden, dass die Zylindergruppen nicht voll werden, und somit neue Datenblöcke eines Files oder neue i-nodes in einem Directory in der Zylindergruppe Platz finden, mit der sie die stärkste logische Verbindung aufweisen. Um das Anfüllen einer Zylindergruppe zu verhindern, werden sehr große Dateien über mehrere Zylindergruppen verteilt, auch wenn sie gerade noch in einer Zylindergruppe Platz fänden. Dies bringt kaum Nachteile für die Übertragungsgeschwindigkeit dieser Dateien, da die Kosten einer längeren Kopfbewegung verglichen mit dem Lesen von z.B. einem Megabyte Daten gering sind.

Die globalen Algorithmen versuchen außerdem noch, aufeinanderfolgende Datenblöcke eines Files so auf der Platte anzuordnen, dass sie möglichst schnell gelesen werden können (zwischen dem Lesen zweier Datenblöcke muss der Computer noch den entsprechenden Interrupt behandeln, was teilweise so lange dauern kann, dass auf der Platte unmittelbar aufeinanderfolgende Blöcke nicht in derselben Plattenumdrehung gelesen werden können; daher muss oft

ein Abstand von einem oder mehreren Blöcken zwischen zwei logisch aufeinanderfolgenden Datenblöcken eingehalten werden (Interleave Faktor)).

Lokale Algorithmen sind für die tatsächliche Zuteilung eines Plattenblocks an eine Datei zuständig. Da die globalen Algorithmen teilweise Blöcke anfordern, die bereits belegt sind (würden die globalen Algorithmen alle Informationen über die Plattenbelegung verwenden, würden sie sehr zeitaufwendig werden), werden die lokalen Algorithmen dazu verwendet, Blöcke zuzuteilen, die möglichst nahe an den Angeforderten liegen. Dazu kann auch, wenn weder im Zylinder noch in der Zylindergruppe des angeforderten Blocks noch Platz vorhanden ist, eventuell die ganze Platte durchsucht werden. Um die Wahrscheinlichkeit dieses Falles sehr gering zu halten, darf das Filesystem nur bis zu einem bestimmten Prozentsatz (z.B. 90 %) gefüllt sein. Nach Überschreiten dieser Grenze darf nur mehr ein Prozess mit Superuser-Berechtigung neue Blöcke anfordern.

EXT2FS Superblock

Der EXT2FS Superblock dient zur Beschreibung der Größe und der Struktur des Filesystems. Aufgrund seiner Wichtigkeit für das Filesystem ist er in jedem Zylindergruppenblock repliziert vorhanden. Er enthält die folgenden Einträge:

- Eine *Filesystem ID* (magic number), die es der Software die für das Mounten zuständig ist ermöglicht, zu prüfen ob es sich tatsächlich um ein EXT2FS handelt.
- Eine *Revisionsnummer*, die es der Mounting Software gestattet festzustellen, welche Dienste von dieser Version des Filesystems unterstützt werden.
- Eine *Zylindergruppen Nummer*, die die Zylindergruppe identifiziert, die diesen Superblock enthält.
- Die *Blockgröße* in Bytes (z.B. 1024).
- Die *Anzahl der Blöcke* pro Zylindergruppe.
- Die *Anzahl der freien Blöcke*.
- Die *Anzahl der unbenutzten i-nodes*.
- Die *i-node Nummer des ersten i-nodes* des File Systems[2].

[2]Für das Rootfilesystem ist das der i-node für den Directory Eintrag des '/' Directories.

EXT2FS i-node

Wie beim VFS, wird auch beim EXT2FS jedes File durch genau einen i-node repräsentiert. Dieser enthält zusätzlich zu den Informationen, die bereits im VFS i-node enthalten sind, noch *Zeiger auf die Datenblöcke* des jeweiligen Files. Hierbei zeigen die ersten zwölf Zeiger direkt auf Datenblöcke, während der 13. Zeiger auf einen *einfach indirekten Block*, der 14. Zeiger auf einen *zweifach indirekten Block* usw. zeigt. Indirekte Blöcke enthalten ihrerseits wiederum Zeiger auf Datenblöcke bzw. auf weiter indirekte Blöcke (siehe Abbildung 2.6).

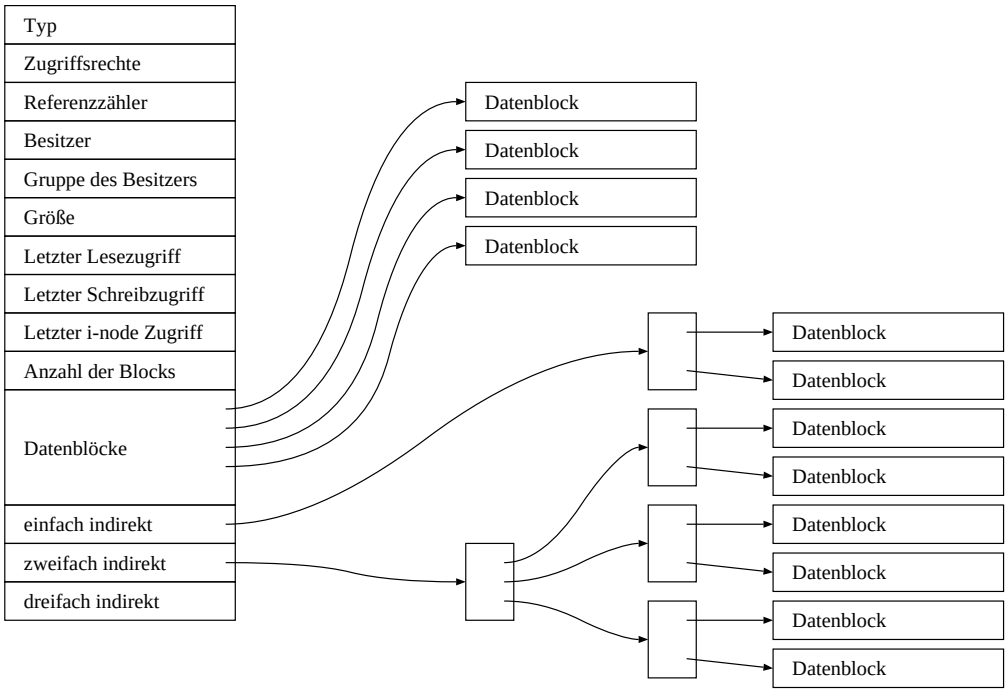


Abbildung 2.6: Struktur eines EXT2FS i-nodes

Diese Aufteilung in direkte und (mehrfach) indirekte Datenblöcke hat zur Folge, dass Zugriffe auf Files, die maximal zwölf Datenblöcke benötigen, schneller erfolgen als Zugriffe auf Files, die mehr als zwölf Datenblöcke (und somit auch indirekte Blöcke) benützen.

EXT2FS Zylindergruppenblock

Dieser Block ist (wie der Superblock) redundant in jeder Zylindergruppe vorhanden. Er besteht aus einem Eintrag für jede Zylindergruppe, wobei diese Einträge selbst wiederum folgende Informationen enthalten:

- Die *Anzahl der freien Blocks* in der Zylindergruppe.
- Die *Anzahl der freien i-nodes* in der Zylindergruppe.
- Die *Anzahl der benützten Directories* in der Zylindergruppe.

- Die Blocknummer des Blocks, der zur Speicherung der *Block Bitmap*[3] verwendet wird.
- Die Blocknummer des Blocks, der zur Speicherung der *i-node Bitmap*[4] verwendet wird.
- Die Blocknummer des Blocks, der den Start der *i-node Tabelle* beherbergt.

Directorystruktur

Die einzigen Informationen über ein File, die sich im Directory befinden, sind der Name des Files und die Nummer des dem File zugeordneten i-node. Filenamen im EXT2FS können bis zu 255 Bytes lang sein. Damit in den Directories nicht zuviel Platz verschwendet wird, werden nicht für jeden Filenamen 255 Bytes reserviert. Statt dessen wird nur der tatsächlich benötigte Platz verwendet. Jeder Directoryeintrag besteht aus vier Komponenten:

- Die Nummer des i-nodes des Files.
- Die Länge des Filenamens.
- Die Länge des gesamten Directoryeintrags.
- Der Filename.

Wird ein Filename gelöscht, so wird die Nummer des i-nodes auf 0 gesetzt. Die Länge des gelöschten Eintrages wird zur Größe des vorherigen Eintrages addiert. Dadurch wird eine allzu große Fragmentierung des Speicherplatzes vermieden. Wird ein neuer Eintrag gebraucht, so kann von einem Eintrag, der mehr Platz reserviert als er benötigt, ein neuer Eintrag abgespalten werden.

Um das Schreiben eines Directoryeintrages als *atomare Operation* durchzuführen, die auch bei einem eventuellen Systemabsturz entweder ganz oder gar nicht durchgeführt wird, werden Directoryeinträge nicht über Blockgrenzen des unterlagerten Speichermediums hinweg geschrieben. Abbildung 2.7 zeigt Beispiele für die Struktur eines Directories.

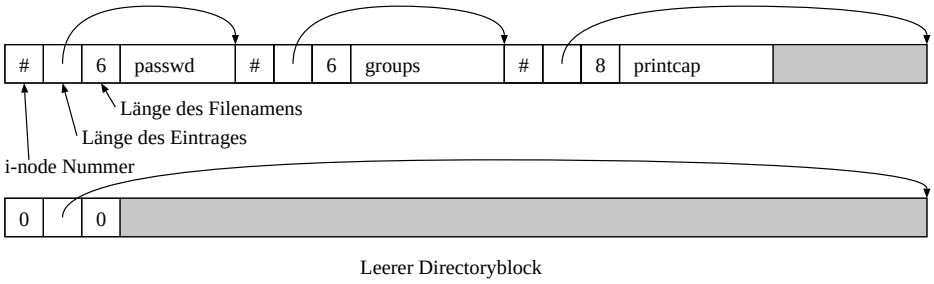


Abbildung 2.7: Struktur von Directoryeinträgen

[3] In dieser Bitmap ist für jeden Block genau ein Bit reserviert, welches gesetzt ist, wenn der Block benützt wird.
[4] analog zur Block Bitmap

2.2.4 Das `/proc` Filesystem (ProcFS)

Das `/proc` Filesystem (ProcFS) ist ein klassisches Beispiel für ein Filesystem, dem kein physikalisches Device zugeordnet ist. Weder die Inhalte des `/proc` Directories, noch die seiner Subdirectories existieren physikalisch. Die Filesystemfunktionen, die das ProcFS beim VFS registriert, generieren vielmehr alle Directoryeinträge des ProcFS "on-the-fly", wenn das VFS i-nodes des ProcFS lesen möchte.

Der Nutzen des ProcFS besteht nun darin, dass es eine Schnittstelle zum Kernel mit Filesystemsyntax bereitstellt. Eine Applikation, die Informationen über Interna des Kernel benötigt, braucht jetzt nicht mehr `/dev/kmem`[5] zu lesen und zu wissen, an welchen Positionen im Speicher sich die gewünschten Informationen befinden, sondern kann sich auf das Lesen der Files und Directories des ProcFS beschränken.

Das ProcFS enthält ein Subdirectory für jeden laufenden Prozess[6], welches wiederum Files und Directories enthält. Beispiele für diese Files bzw. Directories sind:

`stat` Dieses File enthält Informationen über den Status des jeweiligen Prozesses (z.B. Prozess ID, Prozesszustand (running, waiting, swapping, zombie, ...) oder das benützte TTY).

`lib` Dieses Directory enthält einen Eintrag für jede Shared Library, die der Prozess benützt.

`exe` Dieser Eintrag ist ein Link auf das ausführbare File des Prozesses.

Weiters sind im ProcFS Files zu finden, die generelle (d.h. prozessunabhängige) Informationen enthalten. Beispiele hierfür sind:

`loadavg` Dieses File beinhaltet Informationen über die durchschnittliche CPU Belastung.

`meminfo` Dieses File enthält Daten über die derzeitige Speicherauslastung des Systems.

`version` Diese File enthält den Versionsstring der aktuell laufenden Version des Betriebssystems.

2.2.5 Parametrisierung des Filesystems

Um den Zugriff auf das Filesystem optimal auf die gegebene Hardware (Platte und Computer) abzustimmen, werden bei manchen Filesystemen (z.B. beim Filesystem von 4.3BSD) im Superblock des Filesystems Parameter abgespeichert, die den Zugriff auf die Platte steuern. Einige dieser Parameter sind:

- minimaler Abstand von Plattenblöcken, bei dem aufeinanderfolgende Blöcke ohne dazwischenliegende Plattenumdrehung gelesen werden können.
- Zeit zum Umschalten auf einen anderen Lese/Schreibkopf (diese Zeit ist meistens 0).

[5] Dieses Device bietet Zugriff auf den vom Kernel benutzten Speicher.
[6] Der Name des Subdirectories ist die jeweilige Prozess ID.

- Zeit, die der Computer zum Behandeln eines Interrupts nach dem Lesen bzw. Schreiben eines Blocks benötigt.
- Umdrehungsgeschwindigkeit der Platte.
- Anzahl der Blöcke pro Spur.

2.2.6 Interne Verwaltung von Files

Tabellen zum Zugriff auf Dateien

Es gibt drei Tabellen, die für den Zugriff auf Dateien gebraucht werden (siehe auch Abbildung 2.8)

- Die File-Descriptor Tabelle liegt im Kontext eines Prozesses und enthält einen Eintrag für jedes offene File des Prozesses. Die Zahl, die bei einem `open()` oder `dup()` Systemcall zurückgeliefert wird, ist der dem File entsprechende Index in dieser Tabelle.
- Jeder File-Descriptor enthält einen Zeiger auf einen Eintrag in der File-Tabelle, die sich im Kontext des Kernels befindet. Durch jedes `open()` eines Prozesses wird ein neuer Eintrag in dieser File-Tabelle erzeugt; beim Aufruf von `dup()` wird jedoch nur ein neuer File-Descriptor erzeugt, der auf denselben Eintrag in der File-Tabelle wie der originale File-Descriptor zeigt. In der File-Tabelle ist der Zugriffsmodus auf das File (`read`, `write`, ...) und die momentane Position des Lese- bzw. Schreibzeigers im File abgespeichert.
- Jeder Eintrag in der File-Tabelle zeigt auf einen Eintrag in der i-node Tabelle. Diese befindet sich resident im Kernel und beinhaltet genau einen Eintrag für jedes File, das mindestens einen Prozess im System geöffnet hat. Ist eine Datei mehrmals geöffnet, auch von verschiedenen Prozessen, so existiert trotzdem nur ein Eintrag in der i-node Tabelle.

Die Eigenheiten dieser Tabellenorganisation müssen bei der Programmierung berücksichtigt werden, da ansonsten unvorhergesehene Effekte auftreten können:

Nach Aufruf des `dup()` Systemcalls existieren zwei File-Descriptoren, die auf denselben Eintrag in der File-Tabelle zeigen. Da die momentane Position in der Datei in der File-Tabelle abgespeichert ist, bedeutet dies, dass beim alternierenden Lesen von den beiden Descriptoren keine Daten doppelt gelesen werden; beim Schreiben auf die beiden Descriptoren gehen demzufolge auch keine Daten durch Überschreiben der auf den anderen Descriptor geschriebenen Daten verloren (dies gilt nicht nur für `dup()`, sondern auch für `fork()`, da dabei ebenso nur die File-Descriptoren verdoppelt werden). Besondere Vorsicht ist bei der Verwendung von Bibliotheksfunktionen für den Dateizugriff angebracht. Da diese Funktionen Daten normalerweise intern puffern (`fopen()`, `getc()`, `putc()`, ...), werden nach einem `fork()` auch diese Puffer dupliziert. Dadurch werden die noch im Puffer enthaltenen Daten von beiden Prozessen gelesen, nach dem erneuten Füllen der Puffer enthalten diese jedoch verschiedene Daten.

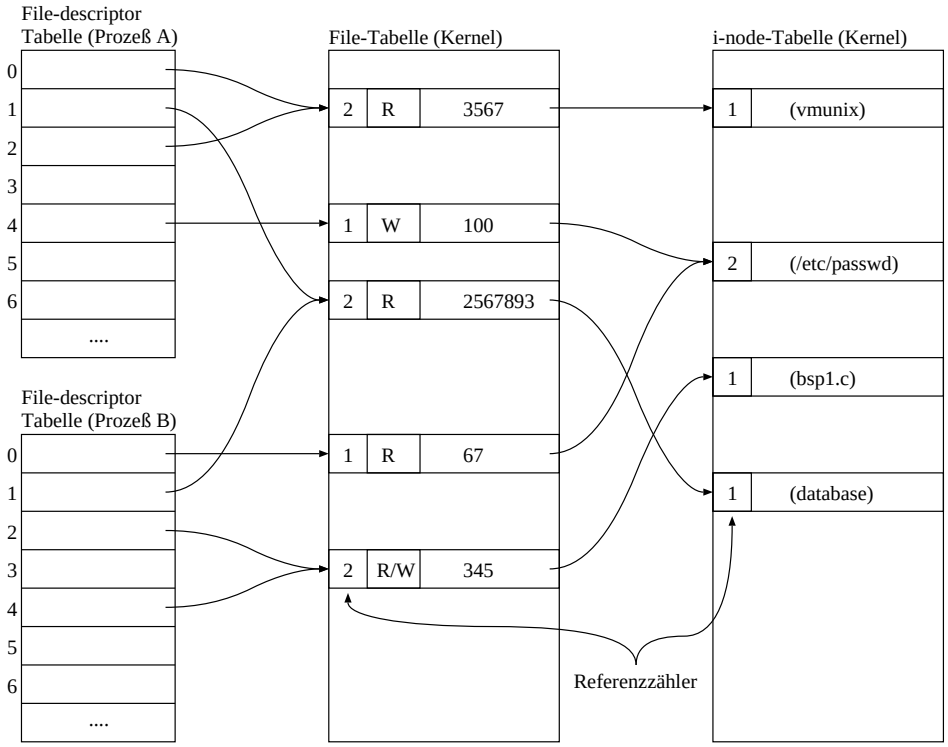


Abbildung 2.8: Tabellen für den Zugriff auf Dateien

Verwaltung der i-nodes

Die i-nodes aller geöffneten Files befinden sich resident in einer Liste im Kernel. i-nodes, die nicht mehr aktiv sind[7], bleiben noch solange in dieser Liste enthalten, bis ihr Platz für einen neuen i-node benötigt wird. Um einen möglichst schnellen Zugriff auf die i-nodes zu ermöglichen, ist die Liste der i-nodes im Kernel als Hashtabelle realisiert. Als Hashcode wird die Nummer des i-nodes zusammen mit der Device-Number des Gerätes verwendet (da auf verschiedenen Geräten gleiche i-nodes vorkommen) (siehe Abbildung 2.9).

Buffer von nicht aktiven i-nodes sind zusätzlich noch in einer linearen Liste verkettet. Wird ein neuer i-node benötigt, so wird dafür der Puffer am Anfang dieser Liste verwendet, und der darin enthaltene i-node aus dem Hauptspeicher entfernt. Ein i-node, der vom aktiven in den inaktiven Zustand wechselt, wird an das Ende dieser Liste angehängt. Dadurch wird für die Verwaltung der inaktiven i-nodes ein LRU Algorithmus realisiert. Beim Öffnen eines Files wird jedesmal zuerst in der Hashtabelle nachgesehen, ob der entsprechende i-node bereits im Hauptspeicher ist (aktiv oder inaktiv).

Ist dies der Fall, so wird dieser verwendet, und es wird nur der Referenzzähler inkrementiert. Ist der i-node nicht im Hauptspeicher, so wird er von der Platte geladen und ersetzt den i-node am Beginn der Liste der inaktiven i-nodes.

[7] Anzahl der Referenzen ist Null

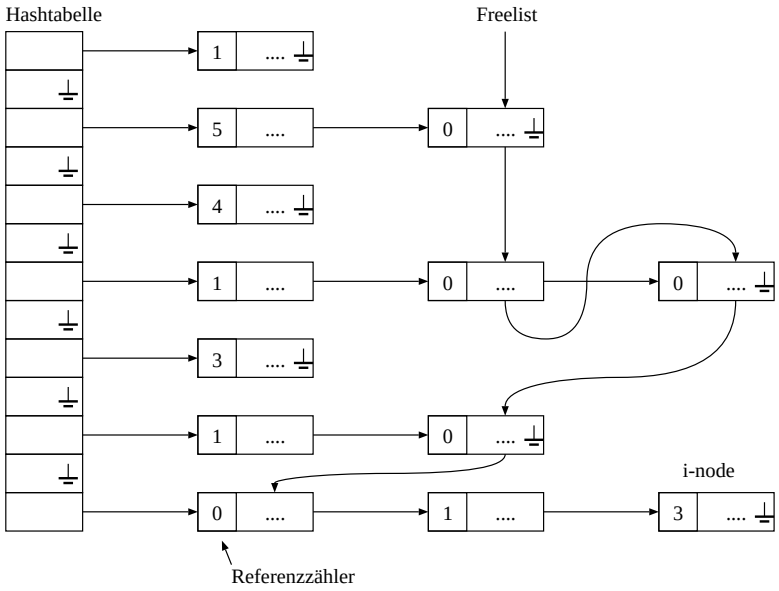


Abbildung 2.9: i-node Tabelle im Hauptspeicher

Systemprozeduren zum Zugriff auf Dateien

Der Kernel enthält einige Prozeduren[8], die für das Öffnen, Lesen, Schreiben, etc. von Dateien zuständig sind:

`namei()` wandelt einen Pfadnamen zu einem File (relativ oder absolut) in die Nummer des entsprechenden i-nodes um. Untersuchungen haben gezeigt, dass fast ein Viertel der gesamten Exekutionszeit des Kernels auf diese Prozedur entfällt. Mechanismen zur Steigerung der Geschwindigkeit dieser Routine tragen also sehr viel zur Performancesteigerung des Gesamtsystems bei. `namei()` besteht im Wesentlichen aus zwei verschachtelten Schleifen. Die äußere Schleife behandelt jede Komponente des Pfadnamens, die innere Schleife sucht im momentanen Directory nach genau dieser Komponente des Pfadnamens. Zwei Arten von Caches wurden eingeführt, um diese Routine zu beschleunigen:

- Im ersten Cache ist der Offset des letzten übersetzten Namens im Directory enthalten. Da einige Programme (z.B. `ls -l`) einen Namen nach dem anderen im selben Directory referenzieren, kann bei dem folgenden Directoryeintrag von der letzten Position weg gesucht werden, was die Performance steigert.
- Der zweite Cache enthält die letzten durchgeführten Übersetzungen eines Namens zu einem i-node (für jede Komponente des Pfadnamens). Jedesmal, wenn eine dieser Komponenten erneut referenziert wird, entfällt das Abarbeiten der inneren Schleife.

`rwip()` wandelt einen Offset in einem File in eine logische Blocknummer um; sie bestimmt, welche Blocks des Files gelesen bzw. geschrieben werden müssen.

[8]Die Namen der Prozeduren, die hier verwendet werden, sind an 4.3BSD angelehnt. Manche der Funktionen tragen in Linux andere Namen, erfüllen aber die gleiche Aufgabe.

`bmap()` wandelt die logische Blocknummer eines Files in eine physikalische Blocknummer des Filesystems um. Diese Prozedur interpretiert die direkten und indirekten Datenblockzeiger im i-node des entsprechenden Files. Muss (beim Schreiben) ein neuer Block angefordert werden, so wird dies ebenfalls von dieser Routine durchgeführt.

`bread()` erhält als Eingabewert die physikalische Nummer eines Blocks und liefert einen Zeiger auf den entsprechenden Puffer im Hauptspeicher zurück. Ist der Block in keinem Puffer vorhanden, so wird der Block von der Platte gelesen.

`bwrite()` schreibt einen Puffer auf die Platte. Es gibt noch zwei andere Formen des Schreibens auf Platte: während `bwrite()` auf die Beendigung des Transfers wartet (synchrones Schreiben), löst `bawrite()` einen asynchronen Schreibvorgang aus. `bdwrite()` markiert einen Puffer nur als *dirty*, d.h. er enthält Daten, die auf die Platte geschrieben werden müssen. Physikalisch wird der Block allerdings erst geschrieben, wenn entweder der Puffer gebraucht wird, oder wenn ein `sync()` Systemcall durchgeführt wird (alle 30 Sekunden durch den `update` Prozess).

Auf alle belegten Puffer wird über eine Hashtabelle zugegriffen. Der Hashcode wird aus der physikalischen Blocknummer und der Gerätenummer gebildet, da dadurch jeder Puffer eindeutig identifiziert wird. Zusätzlich sind alle Puffer (auch die unbenutzten) in zwei Listen organisiert, aus denen jedesmal, wenn ein Puffer benötigt wird, einer entnommen wird (siehe Abbildung 2.10):

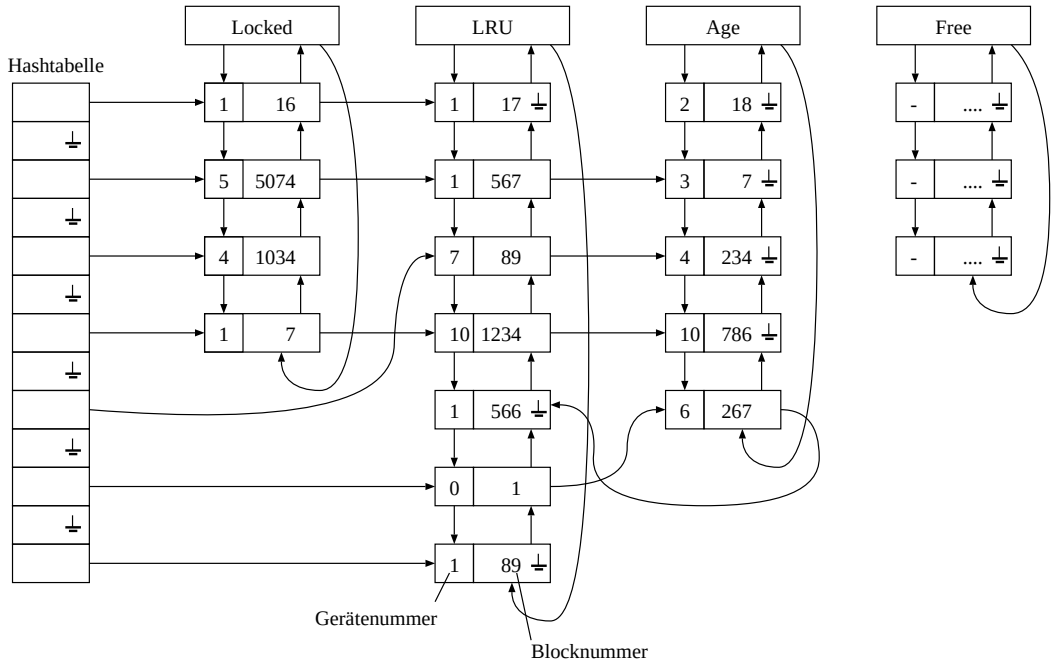


Abbildung 2.10: Organisation der Platten-Puffer im Speicher

- In der *Free-Liste* sind alle leeren Puffer gespeichert.
- Ist die Free-Liste leer, so wird der Puffer am Anfang der *LRU-Liste* verwendet. Jeder Puffer, auf den zugegriffen wird, wird immer an das Ende dieser Liste gestellt. Da dies auch für

Puffer gilt, die sich bereits in der LRU-Liste befinden, ist sichergestellt, dass immer der Puffer abgegeben wird, auf den am längsten nicht zugegriffen wurde.

In 4.3BSD UNIX gibts es zusätzlich noch zwei weitere Listen von Platten-Puffern, um eine möglichst effiziente Handhabung derselben zu gewährleisten:

- In die *Age-Liste* kommen alle Puffer, die zwar gelesen, aber noch nie benötigt wurden (z.B. durch ein '*read-ahead*'); bei einem Zugriff auf eine Datei werden außer dem verlangten Block unter bestimmten Bedingungen auch noch einige direkt darauffolgende Blöcke gelesen, da die Wahrscheinlichkeit, dass diese benötigt werden, relativ groß ist. Außerdem sind die zusätzlichen Kosten für das Lesen aufeinanderfolgender Blöcke nicht sehr groß. Ist die Wahrscheinlichkeit, dass ein solcher Puffer benötigt wird, eher groß, so wird er ans Ende der Age-Liste gehängt, sonst wird er an deren Anfang gestellt. Falls nun ein neuer Puffer benötig wird und die Free-Liste leer ist, wird beim Vorhandensein einer Age-Liste der Puffer aus dieser statt aus der LRU-Liste entnommen.
- Eine weitere Liste, die in 4.3BSD zwar konzeptionell vorhanden ist, jedoch nicht benutzt wird, ist die *Locked-Liste*. In ihr befinden sich alle Blöcke, die nicht aus dem Speicher entfernt werden. Diese Liste könnte z.B. für die Speicherung der Superblöcke verwendet werden.

2.3 Prozessverwaltung

Ein *Prozess* ist die Ausführung eines Programms. In einem Multitasking-System können mehrere Prozesse zur gleichen Zeit aktiv sein. Die Verwaltung der Prozesse ist die Aufgabe des Betriebssystems und besteht aus folgenden Teilaufgaben:

- Starten von Prozessen
- Beenden von Prozessen
- Ausführung von Programmen (Files) im Kontext eines Prozesses
- Scheduling von Prozessen
- Signalbehandlung

Dieses Kapitel beschreibt die Aufgaben des Betriebssystems bei der Prozessverwaltung am Beispiel von Linux.

2.3.1 Struktur von Prozessen

Der *Kontext* eines Prozesses ist die Information, die über einen Prozess gespeichert wird. Er besteht aus den folgenden Teilen:

- Dem Zustand des Prozesses in der *Benutzerebene*, also dem Teil des Adressraumes, der für das ausgeführte Programm selbst wichtig ist.
- Dem Zustand des Prozesses auf der *Betriebssystemebene*. Dieser enthält die Parameter für das Scheduling, Informationen über belegte Ressourcen, über die Speicherbelegung des Prozesses, sowie Information zur Identifikation des Prozesses.

Es ist wichtig, sich den Unterschied zwischen einem *ausführbaren File* und einem *Prozess* vor Augen zu halten. Abbildung 2.11 zeigt den Zusammenhang zwischen den beiden: Eine ausführbare Binärdatei[9] besteht aus einem *Header*, der Informationen über die Art der Datei sowie über die Länge der einzelnen Teile enthält. Auf den Header folgt das *Textsegment*, das den Programmcode enthält. Darauf folgt das *Datensegment* mit den initialisierten Variablen und die *Symboltabelle* mit Informationen zum Debugging. Beim Laden eines Programms werden die Text- und Datensegmente in den Speicher geladen und das *bss-Segment* angelegt, das die uninitialisierten Variablen enthält. Dazu wird ein Bereich von der Länge des bss-Segments (steht im Header) mit Nullen angefüllt. Der Header wird beim Laden des Programms zwar verwendet, aber selbst nicht in den Speicher kopiert, ebensowenig wie die Symboltabelle.

Es ist aufwendig, beim Start eines Programms das gesamte Text- und Datensegment auf einmal in den Speicher zu laden. Linux verwendet daher die Technik des *demand paging*, bei der einzelne Seiten erst dann geladen werden, wenn sie auch tatsächlich referenziert werden. Details über diese Strategie finden sich in Kapitel 2.4.

Der Stapel (*Stack*) des Prozesses wird beim Laden ebenfalls mit Nullen initialisiert. Er wächst von oben nach unten und wird im Bedarfsfall automatisch vergrößert. Das bss-Segment kann zur dynamischen Anforderung von Speicherplatz nach oben erweitert werden (*Heap*). Dazu dient der Systemcall `sbrk()`, der z.B. von der Routine `malloc()` verwendet wird. Zwischen Heap und Stack befinden sich die *shared libraries*, die der Prozess benützt. Oberhalb des Stacks befinden sich der *Argument Vector* und der *Environment Pointer*, die beim Start des Prozesses vom System initialisiert werden. Die sogenannte *User Area* und der *per-Process Kernel Stack* werden vom Betriebssystem zur Verwaltung des Prozesses gebraucht. Sie werden im Folgenden noch genauer beschrieben. Es ist wichtig, sich daran zu erinnern, dass alle bis jetzt beschriebenen Strukturen im virtuellen Speicher liegen, also unter Umständen weder immer im Hauptspeicher (*resident*), noch in einer vorgegebenen Reihenfolge vorhanden sein müssen.

2.3.2 Prozesszustände

In Linux kann sich ein Prozess in einem der folgenden unterschiedlichen Prozesszustände befinden:

Waiting Prozess wartet auf ein Ereignis. Je nachdem, ob das Warten durch ein Signal unterbrochen werden kann oder nicht, unterscheidet Linux zwischen *interruptable waiting* und *non-interruptable waiting*.

Running Prozess ist lauffähig.

[9] Linux unterstützt zur Zeit zwei verschiedene Binärformate (a.out und ELF), wobei sich letzteres durch seine größere Flexibilität auszeichnet.

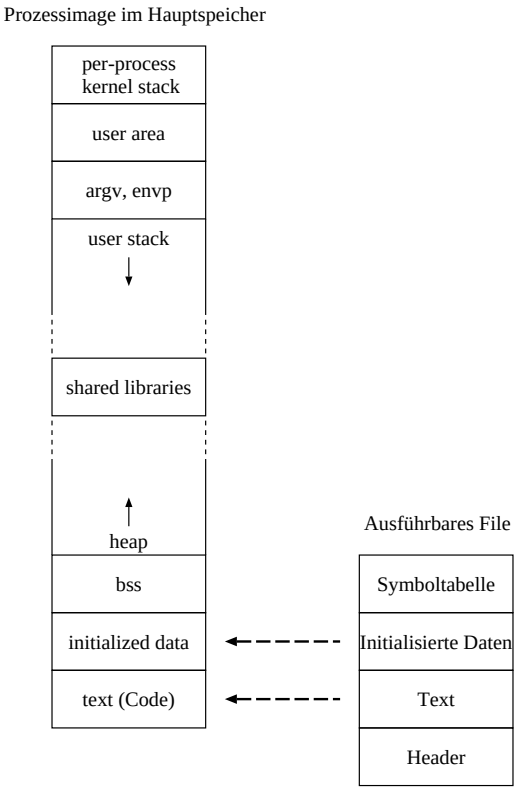


Abbildung 2.11: Struktur eines Prozesses im Speicher und des entsprechenden ausführbaren Files

Swapping Prozess wird gerade erzeugt.

Zombie Prozess hat terminiert, aber der Status wurde noch nicht vom Elternprozess empfangen.

Stopped Prozess wurde gestoppt.

2.3.3 Datenstrukturen zur Prozessverwaltung

Das Betriebssystem verwendet zur Verwaltung der Prozesse einige Datenstrukturen. Zum Teil wurden diese schon bei der Struktur von Prozessen kurz angesprochen. Sie werden hier im Detail erläutert. Einige dieser Strukturen werden nur während der Ausführung des entsprechenden Prozesses gebraucht und können daher ausgelagert werden. Andere müssen zu jedem Zeitpunkt im Hauptspeicher resident sein.

Die `task_struct`-Struktur

Dabei handelt es sich um eine Datenstruktur, die immer im Kernel resident sein muss. Ihre Information lässt sich in mehrere Kategorien einteilen:

Scheduling Die Prioritäten, die CPU-Auslastung und die Zeit, die der Prozess “geschlafen” (Zustand *waiting*) hat.

Identifikation Die Prozessnummer, die Nummer des Elternprozesses und die Nummer des Benutzers (*Real User Id*).

Speicherverwaltung Informationen für die Verwaltung des virtuellen Speichers. Auf diese Informationen wird in Kapitel 2.4 im Detail eingegangen.

Synchronisation Die sogenannte *Wait Queue*, die die Bedingung angibt, auf die der Prozess wartet.

Signale Die Signale, die an den Prozess geschickt wurden, und die Aktionen, die auszuführen sind, wenn der Prozess ein Signal empfängt.

Resource Accounting Informationen über die Betriebsmittelverwendung des Prozesses und über seine Plattenlimits (*Disk Quota*).

Timer-Verwaltung Die Zeit bis ein Timer abläuft.

Process Control Block Der Zustand des Prozesses im User-Mode bzw. im Kernel-Mode wird bei einem *Kontextwechsel* im sogenannten PCB (*Process Control Block*) gespeichert. Er besteht im Wesentlichen aus einem prozessorabhängigen Teil (etwa den Registern des Prozessors, den Stapelzeigern (Stack Pointers), dem Programmzähler (Program Counter), dem Statuswort des Prozessors und den Segmentregistern). Zusätzlich enthält er Informationen über die *Page Tables* für die virtuelle Speicherverwaltung sowie ein kurzes Codestück, das für die Signalbehandlung gebraucht wird und den Zustand des Prozesses in Bezug auf Systemcalls.

Deskriptorentabelle Eine Tabelle aller verwendeten Filedeskriptoren des Prozesses.

Kernelstack Betriebssystemstack für die Abarbeitung von Systemcalls und Traps. Der Stack für das Betriebssystem wird nur für die Abarbeitung von Systemcalls und Traps verwendet (*Top Half*). Für die Abarbeitung von Interruptroutinen (*Bottom Half*), die keinem Prozess direkt zugeordnet werden können, existiert ein eigener Stack, der sogenannte *Interrupt Stack*.

Für jeden Prozess im System existiert eine eigene `task_struct`-Struktur. Die Menge aller vorhandenen `task_struct`-Strukturen ist als doppelt verkettete lineare Liste organisiert. Alle Prozesse, die sich im Zustand *running* befinden, sind außerdem noch in der Liste der lauffähigen Prozesse (*Run Queue*). Weiters besitzt jeder Prozess Zeiger auf seinen *Elternprozess*, auf seinen *jüngsten Kindprozess* und auf jeweils einen seiner *jüngeren bzw. älteren Geschwister*.

Jeder Prozess hat eine eindeutige Nummer, die sogenannte *Prozessnummer* (*Process Identifier PID*). Diese wird vom Kernel und von der Applikation zur Identifikation des Prozesses verwendet. Zwei Prozessnummern sind für einen Prozess von besonderer Bedeutung und werden daher in der `task_struct`-Struktur gespeichert: Die eigene Prozessnummer (`pid`) und die Prozessnummer des Elternprozesses (`ppid`).

Prozesse werden zu sogenannten *Prozessgruppen* (*Process Groups*) zusammengefasst. Die Nummer der Prozessgruppe steht in dem Feld `pgrp`. Die Prozesse einer Prozessgruppe sind untereinander mit Zeigern verkettet. Prozessgruppen werden dazu verwendet, mehrere Prozesse, z.B. in bezug auf die Signalbehandlung, zusammenzufassen (etwa alle Prozesse in einer Pipeline). Die *Signalbehandlung* erlaubt, dass Signale an alle Prozesse in einer Prozessgruppe geschickt werden (siehe Kapitel 2.6.1).

In Linux wird zwischen *Real-Time-Prozessen*[10] und *Non-Real-Time-Prozessen* unterschieden. Erstere haben eine sogenannte Real-Time-Priorität (`rt_priority`), während letzteren eine Non-Real-Time-Priorität (`priority`) zugeordnet ist. Die Auswirkungen dieser beiden Prioritäten auf die Prozessorzuteilung wird im Kapitel 2.3.4 genauer behandelt.

Speicherverwaltung

Die wichtigsten Datenstrukturen für die Verwaltung des Speichers eines Prozesses sind die Seitentabellen (*Page Tables*) und die Text-Struktur (*Text Structure*). Die Einträge der Seitentabelle (*Page Table Entries*, `pte`) geben an, welche Teile des virtuellen Speicherraums des Prozesses sich gerade im physikalischen Hauptspeicher befinden und wo diese im Hauptspeicher zu finden sind. Vor der Exekution eines Prozesses müssen die Seitentabellen geladen werden, und die Speicherverwaltungseinheit der Hardware (*Memory Management Unit*, MMU) muss entsprechend programmiert werden. Die Text-Struktur beschreibt den Aufbau des geladenen Codes eines Programms. Sie wird dazu verwendet, den Code eines Programms mehreren Prozessen gleichzeitig zur Verfügung zu stellen.

Die Datenstrukturen für die Speicherverwaltung werden in Kapitel 2.4 noch genauer behandelt. Zunächst genügt es, anzunehmen, dass der Speicher für die Ausführung eines Prozesses einfach vorhanden ist. Abbildung 2.12 zeigt die verschiedenen Datenstrukturen für die Prozessverwaltung und deren Zusammenhang.

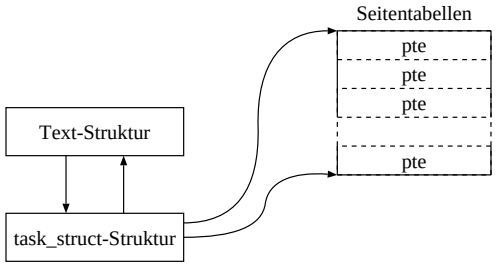


Abbildung 2.12: Datenstrukturen zur Prozessverwaltung

2.3.4 Schedulingmechanismen

UNIX (und auch Linux) ist ein Multitasking-Betriebssystem. Das heißt, dass mehrere Prozesse zur gleichen Zeit aktiv sein können. Es ist die Aufgabe des Betriebssystems, die CPU zwischen den einzelnen Prozessen umzuschalten. Dieses Umschalten zwischen Prozessen wird als *Kontextwechsel* (*context switch*) bezeichnet. Es gibt im wesentlichen zwei Arten des Kontextwechsels:

- Die CPU wird zwischen zwei Prozessen umgeschaltet
- Ein Prozess wird unterbrochen, weil die CPU zur Behandlung eines asynchronen Ereignisses gebraucht wird (etwa ein Interrupt von einem Device).

[10] Der Begriff Real-Time bezieht sich in diesem Kontext auf *Soft Real-Time* und hat nur entfernt mit dem Konzept von harten Echtzeitsystemen zu tun.

Die Kontextwechsel zwischen Prozessen kann man noch weiter differenzieren, je nachdem, aus welchem Grund der Kontextwechsel auftritt:

Freiwillig Der Prozess braucht die CPU zur Zeit nicht mehr, weil er auf das Eintreten irgendeiner Bedingung wartet (etwa die Beendigung einer Ein/Ausgabeoperation, auf das Freiwerden eines Betriebsmittels, `sleep()`, ...).

Unfreiwillig Die Zeitscheibe des Prozesses ist abgelaufen, oder ein Prozess mit höherer Priorität steht zur Abarbeitung bereit.

Für jede Art von Kontextwechsel gibt es eine eigene Methode ihn auszuführen: Freiwillige Kontextwechsel werden ausgeführt, indem der Prozess selbst die Routine `sleep_on()` aufruft. Ein unfreiwilliger Kontextwechsel wird ausgeführt, wenn der Kernel die Routine `schedule()` aufruft. Die asynchronen Kontextwechsel werden von der Hardware selbst ausgeführt (etwa über Interrupts).

Beim Kontextwechsel muss zuerst die Zustandsinformation des alten Prozesses abgespeichert werden (in der `task_struct`-Struktur):

- Der *User-Mode Prozessor-Status* wird bei jedem Eintritt in den Kernel auf den Kernel-Stack gerettet. Da ein Kontextwechsel nur im Kernel-Mode stattfindet (entweder durch Aufruf von `sleep_on()` in einem Systemcall oder infolge eines Hardware-Interrupts), ist der User-Mode Prozessor-Status beim Auftreten des Kontextwechsels bereits gespeichert.
- Der *Kernel-Mode Prozessor-Status* wird im PCB der `task_struct`-Struktur abgespeichert, wenn der Prozess den Prozessor freiwillig durch Aufruf von `sleep_on()` abgibt. Ansonsten kann ein Prozess, der sich im Kernel-Mode befindet, nicht unterbrochen werden, hat also auch keinen Kernel-Mode Prozessor-Status.

Nun wird der nächste abzuarbeitende Prozess bestimmt. Die `task_struct`-Struktur und die Text-Struktur eines Prozesses sind immer resident. Die Daten für die virtuelle Adressumsetzung müssen nur dann neu berechnet werden, wenn der Prozess als Ganzes (inklusive Seitentabellen) ausgelagert wurde, ansonsten bleiben sie gültig.

Kontextwechsel auf niedriger Ebene

Weil alle relevanten Informationen über einen Prozess in der `task_struct`-Struktur gespeichert sind, ist es für einen Kontextwechsel ausreichend, die `task_struct`-Struktur des einen Prozesses gegen die des anderen Prozesses auszutauschen. Das geschieht dadurch, dass die virtuelle Adresse der `task_struct`-Struktur für die gerade laufenden Prozesse immer auf dieselbe Stelle gesetzt wird.

Freiwilliger Kontextwechsel

Wann immer die Ausführung eines Prozesses blockiert wird, weil ein Ereignis, das der Prozess zum Weiterarbeiten benötigt, noch nicht eingetreten ist, tritt ein freiwilliger Kontextwechsel auf. Dazu

ruft der Prozess die Routine `sleep_on()` auf. Die Parameter dieser Routine sind die sogenannte *Wait Queue* und eine Priorität. Jede Wait Queue steht für eine Klasse von Ereignissen. Typische Wait Queues sind etwa:

`wait_chldexit` wartet auf die Terminierung eines Kindprozesses.

`sleeper` wartet darauf, dass ein Semaphore freigegeben wird.

`wait_for_request` wartet auf das Ende einer Block-IO-Operation (etwa eines Plattenzugriffs).

Jede Kernelprozedur kann *eigene Wait Queues* definieren. Da eine Wait Queue einfach als Adresse im Adressbereich der Prozedur definiert wird, können keine Konflikte zwischen Wait Queues verschiedener Prozeduren auftreten.

Die Funktion der `sleep_on()`-Routine wird hier beschrieben:

1. Setze den Prozesszustand auf *waiting*.
2. Sichere die Prozessorflags.
3. Sperre die Interrupts, die eine Änderung der Prozesszustände bewirken könnten (d.h. setze den Prozessor auf eine entsprechend hohe Hardwareprioritätsstufe).
4. Trage den Prozess in die Wait Queue ein.
5. Lasse die gesperrten Interrupts wieder zu.
6. Rufe `schedule()` auf, um einen anderen lauffähigen Prozess zu schedulen[11].
7. Sperre abermals die Interrupts.
8. Entferne den Prozess aus der Wait Queue.
9. Setze die gesicherten Prozessorflags.

Prozesse können entweder unterbrechbar oder ununterbrechbar schlafen. Üblicherweise schlafen Prozesse auf einer nicht unterbrechbaren Prioritätsebene, außer wenn erwartet wird, dass das erwartete Ereignis erst später eintritt.

Wenn eine Bedingung eintritt, auf die einer oder mehrere Prozesse warten, wird vom System die Routine `wake_up()` aufgerufen. Der Parameter von `wake_up()` ist die Wait Queue. `wake_up()` weckt alle Prozesse, die auf dieser Wait Queue schlafen, in der Reihenfolge von vorne nach hinten.

Hierbei werden für jeden Prozess, der geweckt werden muss, die nachfolgenden 5 Schritte ausgeführt:

[11] Dadurch, dass der Zustand des aktuellen Prozesses *waiting* ist, wird er solange nicht gescheduled, bis ein anderer Prozess `wake_up()` mit der entsprechenden Wait Queue als Parameter aufruft.

1. Sichere die Prozessorflags.
2. Sperre die Interrupts.
3. Setze den Prozesszustand auf *running*.
4. Trage den Prozess in die Run Queue ein.
5. Setze die gesicherten Prozessorflags.

Synchronisation

Bei der Synchronisation im Kernel zum Zugriff auf Ressourcen sind zwei Fälle zu unterscheiden:

- Bei der Synchronisation zwischen zwei Prozessen werden zwei Flags `locked` und `wanted` verwendet. Wenn ein Prozess auf eine Ressource zugreifen will, überprüft er, ob das `locked`-Flag gesetzt ist. Ist es nicht gesetzt, setzt er es und verwendet die Ressource. Wenn es jedoch gesetzt ist, setzt er das `wanted`-Flag und ruft `sleep_on()` mit einer Wait Queue auf, die die Ressource beschreibt. Wenn ein Prozess eine Ressource nicht mehr benötigt, löscht er das `locked`-Flag. Wenn das `wanted`-Flag gesetzt ist, ruft er zusätzlich `wake_up()` auf, um die wartenden Prozesse zu wecken.
- Bei der Synchronisation zwischen einem Prozess und einer Interruptroutine kann dieses Schema nicht verwendet werden, weil eine Interruptroutine nicht verzögert werden darf. Um das Warten in einer Interruptroutine zu vermeiden, wird während des Zugriffs von Prozessen auf gemeinsame Datenstrukturen die Hardwarepriorität so gesetzt, dass gar keine Interrupts auftreten können.

Selbstverständlich funktioniert der Ansatz mit der Prozessorpriorität nur bei Einzelprozessorsystemen. Bei Multiprozessorsystemen sind andere Ansätze, wie etwa die Verwendung von Semaphoren, notwendig.

Schedulingalgorithmen

Wie bereits erwähnt unterscheidet Linux zwischen *Real-Time-Prozessen* und *Non-Real-Time-Prozessen*. Ersteren wird entsprechend ihrer Real-Time-Priorität eine fixe Zeitscheibe zugeteilt. Die Non-Real-Time-Prozesse bekommen ebenfalls abhängig von ihrer Non-Real-Time-Priorität eine (ebenfalls fixe) Zeitscheibe zugeteilt.

Der eigentliche Scheduling Algorithmus ist *Round-Robin*-basiert, wobei eine Bevorzugung der höherprioren Prozesse dadurch erreicht wird, dass höherpriore Prozesse eine längere Zeitscheibe erhalten.

Der Algorithmus kann in folgende drei Stufen unterteilt werden:

1. *Handling des aktuellen Prozesses*

- Wenn die verbleibende Zeitscheibe des aktuellen Prozesses gleich Null ist, dann wird seine verbleibende Zeitscheibe gleich seiner Priorität gesetzt und er ans Ende der Run Queue gestellt.
- Wenn der Status des aktuellen Prozesses *interruptable waiting* ist, und er ein Signal erhalten hat, dann wird sein Status auf *running* gesetzt.
- Wenn der Status des aktuellen Prozesses verschieden von *running* ist, dann wird er von der Run Queue entfernt.

2. *Selektion des nächsten Prozesses*

   - Es wird der Prozess mit der höchsten `goodness` aus der Run Queue gewählt, wobei bei gleichen `goodness` Werten Prozesse, die weiter vorne in der Queue stehen, selektiert werden.
   - Ist die Run Queue leer, oder existiert kein Prozess mit einem `goodness` Wert größer als Null, dann wird der sogenannte *idle process* gescheduled, dessen einzige Aufgabe darin besteht, CPU Zyklen zu vernichten.

3. *Aktivierung des selektierten Prozesses*

   - Falls der selektierte Prozess verschieden vom aktuell Laufenden ist, wird ein Kontext-Wechsel durchgeführt.

Die `goodness` eines Prozesses gibt an, wie gut der jeweilige Prozess vom Scheduler behandelt werden soll. Die berechnet sich wie folgt:

- Falls der Prozess ein Real-Time-Prozess ist, dann ist seine `goodness` konstant 1000 + `rt_priority`. Dadurch wird sichergestellt, dass Real-Time-Prozesse im allgemeinen vor Non-Real-Time-Prozessen gescheduled werden.
- Der gerade laufende Prozess erhält als `goodness` den um eins inkrementierten Wert seiner noch verbleibenden Zeitscheibe.
- Bei allen anderen Prozessen ist die `goodness` gleich dem Wert ihrer noch verbleibenden Zeitscheibe.

2.3.5 Erzeugen von Prozessen

Ein neuer Prozess wird in Linux mit dem Systemcall `fork()` generiert. Der Kindprozess ist eine genaue Kopie des Prozesses, der `fork()` aufgerufen hat (des *Elternprozesses*). Der einzige Unterschied in den Prozessen ist der Wert, den `fork()` zurückliefert: Dieser ist im Kindprozess Null und im Elternprozess die Prozessnummer des Kindprozesses.

Folgende Schritte sind zum Erzeugen eines Prozesses notwendig:

1. Anlegen und Initialisieren einer neuen `task_struct`-Struktur.

2. Duplizieren der `task_struct`-Struktur des Elternprozesses.

3. Erzeugen einer Referenz auf die Resourcen (z.B. Virtueller Speicher, Filedeskriptoren, ...) des Elternprozesses.

4. Anlegen einer Wait Queue (`wait_chldexit`) um auf die Terminierung der eigenen Kindprozesse warten zu können.

5. Scheduling des Kindprozesses.

Im ersten Schritt wird zuerst eine neue `task_struct`-Struktur angelegt. Dann wird der größte Teil der `task_struct`-Struktur des Elternprozesses auf die neue Struktur kopiert. Die wesentlichste Ausnahme ist die Prozessnummer. Zum Kopieren der einzelnen Teile der `task_struct`-Struktur werden eigene Routinen verwendet. Schließlich wird der neue Prozess auf die Run Queue gesetzt.

2.3.6 Beendigung von Prozessen

Die Beendigung eines Prozesses ist ein zweistufiger Vorgang: In der ersten Stufe werden alle Ressourcen, die noch von dem Prozess belegt sind, freigegeben und der Exit-Status des Prozesses in seiner `task_struct`-Struktur gespeichert. Der Prozess wird dann von der Liste der lauffähigen Prozesse genommen. Sein Zustand wird auf *zombie* gesetzt. Anschließend wird der Elternprozess mittels eines Signals von der Terminierung in Kenntnis gesetzt und mittels `wake_up()` mit der Wait Queue `wait_chldexit` des Elternprozesses als Parameter aufgeweckt.

Die zweite Stufe tritt dann auf, wenn der Elternprozess `wait()` (oder auch `wait3()`) aufruft. Hierbei ruft der Elternprozess `sleep_on()` mit der Wait Queue `wait_chldexit` als Parameter auf. Wenn nun ein Kindprozess terminiert und den Elternprozess mittels `wake_up()` aufweckt, dann wird die gesamte Information über den Status des Kindprozesses an den Elternprozess übermittelt. Erst dann wird der Kindprozess wirklich gelöscht und seine `task_struct`-Struktur wieder freigegeben.

2.3.7 Threads

In moderneren Unix-Systemen, gibt es neben Prozessen noch eine weitere Möglichkeit, Nebenläufigkeit zu erreichen: *Threads*. Das traditionelle Prozessmodell wird dabei dadurch erweitert, dass in einem Prozess mehrere Threads ablaufen können, die sich alle Betriebsmittel (Adressraum, offene Filehandles, ...) teilen, jedoch jeweils einen eigenen Stack haben.

Der Vorteil in der Verwendung von Threads gegenüber Prozessen und etwa Shared Memory als Kommunikationsmittel ist, dass Threads in vielen Betriebsystemimplementierungen „billiger“ zu erzeugen und zu verwalten sind. Ausserdem ist durch die gemeinsame Nutzung eines Adressraums keine explizite Erzeugung eines Shared-Memory-Bereichs nötig. Die gemeinsame Nutzung des Adressraums hat freilich den Nachteil, dass ein Bug in einem Thread den gesamten Prozess, mitsamt allen anderen Threads, die darin ablaufen, zum Absturz bringen kann.

Das POSIX-Programmierinterface für Threads wurde mit der SUSv2 (Single Unix Specification) 1997 standardisiert; es wird daher von fast allen neueren Unix-Varianten unterstützt. Im folgenden sollen die wichtigsten Funktionen des POSIX-Thread-API kurz vorgestellt werden:

```
int pthread_create (pthread_t * thread,
                    pthread_attr_t *attr,
                    void * (*start_routine)(void *),
                    void * arg);

void pthread_exit (void *retval);

int pthread_join (pthread_t thread, void **thread_return);
```

`pthread_create()` erzeugt einen neuen Thread, der parallel zum aufrufenden Thread abläuft. Im neuen Thread wird `start_routine` mit `arg` als Argument aufgerufen. Der neue Thread läuft solange bis `start_routine` zurückkehrt oder `pthread_exit()` aufruft. Mit `pthread_join()` kann auf die Beendigung eines Threads gewartet werden; der Rückgabewert des Threads wird in `thread_return` abgelegt.

Das POSIX Thread API stellt auch Mutexes (binäre Semaphoren, siehe auch Abschnitt 2.6.2) zur Verfügung:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock (pthread_mutex_t *mutex);

int pthread_mutex_trylock (pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Mit Hilfe von *Condition Variables* kann auf ein Ereignis gewartet werden:

```
int pthread_cond_init (pthread_cond_t *cond,
                       pthread_condattr_t *cond_attr);

int pthread_cond_signal (pthread_cond_t *cond);

int pthread_cond_broadcast (pthread_cond_t *cond);

int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex);

int pthread_cond_timedwait (pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec *abstime);

int pthread_cond_destroy (pthread_cond_t *cond);
```

Mit `pthread_cond_init()` wird eine Condition Variable initialisiert, mit `pthread_cond_destroy()` die Resourcen wieder freigegeben. `pthread_cond_wait()` führt eine atomare Unlock-Operation auf dem übergebenen Mutex aus (der Mutex muss also vor dem Aufruf gesperrt sein) und wartet, bis mit `pthread_cond_signal()` oder `pthread_cond_broadcast()` die Condition Variable signalisiert wird. Bevor `pthread_cond_wait()` zurückkehrt, wird der Mutex wieder gesperrt.

Weiters erlaubt das POSIX Thread API die Verwaltung von Thread-spezifischen Daten (*Thread specific data*):

```
int pthread_key_create (pthread_key_t *key,
                        void (*destr_function) (void *));

int pthread_key_delete (pthread_key_t key);

int pthread_setspecific (pthread_key_t key, const void *pointer);

void * pthread_getspecific (pthread_key_t key);
```

Mit `pthread_key_create()` wird ein Schlüssel erzeugt, unter dem jeder Thread mit `pthread_setspecific()` und `pthread_getspecific()` individuell Daten speichern bzw. auslesen kann; die unter einem Schlüssel erreichbaren Daten sind dabei vom aufrufenden Thread abhängig, d.h. der Rückgabewert von `pthread_getspecific()` mit demselben Key aus zwei verschiedenen Threads liefert i.a. zwei unterschiedliche Ergebnisse.

2.4 Speicherverwaltung

Die Speicherverwaltung ist ein zentraler Teil jedes Betriebssystems. Moderne Computersysteme haben eine Hierarchie von verschiedenen Speichertypen (typischerweise Cache, Hauptspeicher, Plattenspeicher, Remote Fileserver). Die Aufgabe der Speicherverwaltung ist es, die verschiedenen Typen von Speichern als homogenen Speicher (*virtueller Speicher*) erscheinen zu lassen. Dabei werden verschiedene Anforderungen an das System gestellt:

- Der Adressraum des virtuellen Speichers soll größer als der tatsächlich vorhandene Hauptspeicher sein.
- Der virtuelle Speicher soll nicht wesentlich langsamer als der Hauptspeicher sein.
- Es sollen mehr Prozesse zur gleichen Zeit aktiv sein können, als im Hauptspeicher Platz haben.
- Der Speicherbereich eines Prozesses soll vor dem Zugriff anderer Prozesse geschützt werden.
- Die Erweiterung des Speicherraums soll ‘transparent’, d.h. unsichtbar für den Benutzer, erfolgen.

Die (historisch) ersten Ansätze zur besseren Ausnutzung des Adressraumes basierten auf dem Konzept der *Overlays*. Dabei wurden unter der Kontrolle des Programms Teile, die momentan nicht gebraucht wurden, ausgelagert. Heute werden vor allem zwei Arten der virtuellen Speicherverwaltung (bzw. eine Kombination der beiden) verwendet: *Swapping* und *Paging*.

Swapping ist eine Methode der Speicherverwaltung, bei der ganze Prozesse aus dem Hauptspeicher auf den Sekundärspeicher ausgelagert werden (*Swap Out*) und vor ihrer Fortsetzung wieder eingelagert werden (*Swap In*). Swapping ist relativ einfach zu implementieren, es erfüllt aber nicht alle der oben angesprochenen Anforderungen. So müssen Prozesse immer noch zur Gänze im Hauptspeicher resident sein, um abgearbeitet werden zu können. Außerdem ist das Ein- und Auslagern eines ganzen Prozesses eine zeitaufwendige Sache.

Paging überwindet die Probleme, die beim Swapping vorhanden sind. Es benötigt allerdings Unterstützung durch Spezialhardware (*Memory Management Unit*, MMU). Beim Paging wird der physikalische Speicher in *Seitenrahmen* gleicher Größe unterteilt. Jeder dieser Seitenrahmen kann eine *Seite* aufnehmen. Die Adressen im virtuellen Adressraum werden von der MMU in physikalische Adressen umgerechnet. Beim Paging müssen nicht alle Seiten eines Prozesses zur gleichen Zeit im Hauptspeicher vorhanden sein. Statt dessen wird jedesmal, wenn auf eine Adresse zugegriffen wird, die nicht im Hauptspeicher vorhanden ist, die entsprechende Seite von einem speziellen Bereich des Sekundärspeichers, dem sogenannten *Swap Space* gelesen. Umgekehrt werden Seiten, die nicht mehr gebraucht werden, in den Swap Space zurückgeschrieben (sofern sie verändert wurden).

Im Rest dieses Kapitels wird beschrieben, wie die Speicherverwaltung in Linux implementiert ist.

2.4.1 Virtueller Adressraum

Die Speicherverwaltung von Linux teilt den gesamten virtuellen Adressraum in zwei Teile. Ein Teil beinhaltet den Kernel, während der andere Teil den (User-)Prozess beherbergt.

Bei Prozessoren, die verschiedene Adressmodi unterstützen (z.B. der Intel 80386) kann der Kernel sogar im sogenannten *physikalischen Adressmodus* laufen, während die Userprozesse im *virtuellen Adressmodus* abgearbeitet werden. Der Hauptunterschied zwischen den beiden Modi ist die Adressumsetzung. Während im virtuellen Modus eine Adressumsetzung durch das Betriebssystem (unterstützt durch die jeweilige Prozessorhardware) nötig ist, um eine virtuelle Adresse in eine physikalische Adresse umzuwandeln, ist das beim physikalischen Adressmodus nicht nötig.

2.4.2 Adressumsetzung

Um nun auf eine Speicherstelle im virtuellen Adressraum zugreifen zu können, muss das System eine *Adressumsetzung* durchführen. Dabei wird die virtuelle Adresse in eine physikalische Adresse umgerechnet. Diese Umsetzung wird von der MMU durchgeführt, die dazu auf sogenannte *Seitentabellen* (*Page Tables*) zugreift. Die Seitentabelle gibt für jede Seite des virtuellen Speichers an, ob sie gerade im physikalischen Speicher vorhanden ist, und wenn ja, in welchem Seitenrahmen.

Diese Umsetzung ist abhängig vom verwendeten Prozessor. Um trotzdem eine gewisse Portabilität des Kernelcodes zu erreichen, abstrahiert der Linux Kernel durch die Einführung von *Makros zur Adressumsetzung*. Diese Makros müssen für jeden Prozessor implementiert werden und verdecken hauptsächlich folgende Parameter vor dem Kernel:

- Anzahl der Seitentabellen
- Struktur eines Seitentabelleneintrags
- Struktur einer virtuellen Adresse

Im Folgenden wird nun genauer auf die Adressumsetzung beim Intel 80386 eingegangen.

Abbildung 2.13: Virtuelle Adresse am Intel 80386

Jede virtuelle 32-bit Adresse (siehe Abbildung 2.13) wird dabei wie folgt betrachtet: Die Bits 22 bis 31 geben den Index im sogenannten *Seitentabellenverzeichnis (Page Directory)* an. Die Bits 12 bis 21 spezifizieren den Index in der Seitentabelle, wo der Eintrag (Abbildung 2.14 skizziert einen solchen Seitentabelleneintrag) für die jeweilige virtuelle Adresse zu finden ist. Die Bits null bis elf definieren den *Offset* in der jeweiligen physikalischen Seite.

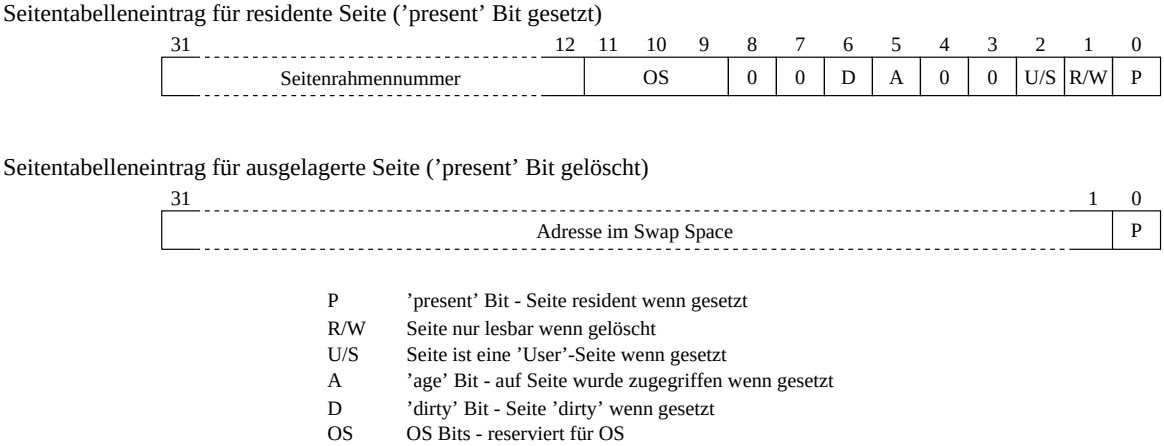


Abbildung 2.14: Seitentabelleneintrag des Intel 80386

Die Adressumsetzung geht nun (vereinfacht) wie folgt vor sich:

1. Die MMU addiert den Index im Seitentabellenverzeichnis (Bits 22 bis 31) zur Basisadresse des selbigen[12], um die Adresse des jeweiligen Eintrags im Seitentabellenverzeichnis zu erhalten. – Dieser Eintrag definiert die Basisadresse der dem Eintrag zugeordneten Seitentabelle.

[12] Diese Basisadresse steht in der `task_struct`-Struktur.

2. Die Addition von Basisadresse der Seitentabelle und Seitentabellenindex (Bits 12 bis 21) liefert die Adresse des Seitentabelleneintrags.

3. Nun erfolgt eine Überprüfung (durch die MMU) anhand des Seitentabelleneintrags, ob der Prozess berechtigt ist, auf die Seite zuzugreifen (`R/W` und `U/S` Bits werden geprüft), und ob die Seite im Hauptspeicher vorhanden ist (*present*-Bit).

4. Sind alle diese Bedingungen erfüllt, dann liest die MMU aus dem Seitentabelleneintrag die Nummer des Seitenrahmens, multipliziert sie mit der Größe einer Seite (4 KByte beim Intel 80386) und addiert den Offset innerhalb der Seite (Bits null bis elf).

Um den durchschnittlichen Zeitaufwand für die Adressumsetzungen zu senken, verwaltet die Hardware einen Cache von Seitentabelleneinträgen, den sogenannten *translation lookaside buffer*. Wenn ein Eintrag in diesem Cache gefunden wird, wird der Zugriff auf den entsprechenden Seitentabelleneintrag im Hauptspeicher vermieden. Die Verwaltung des Cache bringt allerdings Probleme der Konsistenz mit sich, weil das Betriebssystem sicherstellen muss, dass die Einträge im Cache mit denen im Speicher konsistent sind und bleiben.

2.4.3 Seitenfehler

Wenn bei der Adressumsetzung festgestellt wird, dass eine Seite nicht im Hauptspeicher vorhanden ist (das *present*-Bit im Seitentabelleneintrag ist nicht gesetzt), dann liegt ein *Seitenfehler* (*Page Fault*) vor. In diesem Fall muss die Abarbeitung des Programms unterbrochen werden, und die geforderte Seite in den Hauptspeicher gebracht werden[13]. Dazu muss das System etwaige Seiteneffekte der Instruktion, die den Seitenfehler verursacht hat, rückgängig machen, einen freien Seitenrahmen finden, die Seite in diesen Seitenrahmen laden, die Seitentabelle entsprechend modifizieren und schließlich die Instruktion neu starten.

2.4.4 Austauschstrategie

Das Finden eines leeren Seitenrahmens ist natürlich dann ein Problem, wenn kein Seitenrahmen mehr frei ist. Daher muss das System nicht mehr gebrauchte Seiten auf den Sekundärspeicher auslagern. Der Bereich des Sekundärspeichers, der dafür vorgesehen ist, heißt *Swap Space*. Es werden allerdings nur jene Seiten in den Swap Space ausgelagert, die verändert wurden (bei denen das *dirty*-Bit gesetzt ist). Alle anderen Seiten werden einfach verworfen, weil sie sowieso z.B. vom ausführbaren File des Prozesses wieder eingelagert werden können.

Prinzipiell wäre es möglich, Seiten erst dann auszulagern, wenn ein leerer Seitenrahmen gebraucht wird. In Linux wird aber ein anderer Ansatz gewählt: Es gibt einen eigenen Systemprozess, den *Pagedaemon* (`kswapd`), der periodisch alle Seitenrahmen überprüft und jene Seiten auslagert, die vermutlich in nächster Zukunft nicht mehr gebraucht werden. Welche der im Hauptspeicher vorhandenen Seiten ausgelagert wird, wird von der *Austauschstrategie* entschieden. Das Ziel ist es natürlich, eine Seite auszulagern, deren nächste Verwendung möglichst weit in der Zukunft liegt. Das

[13] In diesem Fall geben die Bits eins bis 31 des Seitentabelleneintrags die Adresse der Seite im Swap Space an.

ist bei dynamischen Systemen im Allgemeinen nicht möglich[14]. Daher verwendet man Algorithmen (LRU, LFU, ...), die Informationen aus der Vergangenheit benutzen, um eine Seite zum Page-Out auszuwählen.

Eine der besten Strategien ist LRU (*Least Recently Used*). Das heißt, es wird jene Seite ausgelagert, deren letzte Referenz am weitesten in der Vergangenheit liegt. Bei der praktischen Implementierung dieses Algorithmus tritt jedoch folgendes Problem auf: Die Hardware der meisten Systeme speichert keine Information über den Zeitpunkt des letzten Zugriffs auf eine Seite. Meistens wird nur in einem Bit gespeichert, ob auf diese Seite überhaupt zugegriffen wurde (Referenzbit, *Page Reference Bit*).

Dieses Problem kann mit dem *Clock-Algorithmus* gelöst werden. Dieser Algorithmus benötigt nur ein Referenzbit (*age*-Bit (siehe Abbildung 2.14)) und zeigt ein sehr ähnliches Verhalten wie LRU. Dabei wird der physikalische Speicher zyklisch untersucht (so wie der Zeiger einer Uhr über das Zifferblatt streicht). Beim ersten Umlauf werden alle Referenzbits gelöscht. Beim zweiten Umlauf werden alle Seiten, deren Referenzbit immer noch gelöscht ist, ausgelagert. Das sind alle, die seit dem letzten Umlauf nicht verwendet worden sind. In der Praxis verhält sich dieser Algorithmus sehr ähnlich wie LRU: Statt wiederholt die Seite auszulagern, die am längsten nicht verwendet wurde, werden alle Seiten ausgelagert, die "lange" nicht verwendet wurden. So wie eben beschrieben, hat der Algorithmus allerdings folgenden Nachteil: Vom Start des Algorithmus bis zum Entfernen der ersten Seiten dauert es mindestens eine ganze Umlaufzeit des Zeigers im Hauptspeicher. Der *Two-Handed Clock-Algorithmus* vermeidet das. Dabei kreisen zwei Zeiger in konstantem Abstand. Der erste löscht die Referenzbits und der zweite untersucht, ob die Bits immer noch gelöscht sind.

Linux verwendet den *Single-Handed Clock-Algorithmus* zum Austausch nicht benutzter Seiten, während unter 4.3BSD der *Two-Handed Clock-Algorithmus* seinen Einsatz findet[15].

Der Pagedaemon passt sein Verhalten der Menge des verfügbaren Speichers an. Wenn viel freier Speicher vorhanden ist, dann schläft der Pagedaemon. Erst wenn der freie Speicher unter den Schwellwert *free_pages_low* sinkt, beginnt der Pagedaemon mit dem Clock-Algorithmus. Die Anzahl der Seiten, die der Pagedaemon versucht, zwischen zwei Aktivierungen freizugeben, ist indirekt proportional zum freien Speicherplatz.

2.4.5 Demand Paging

Systeme ohne virtuelle Speicherverwaltung müssen beim Start eines Programms das gesamte Programm in den Hauptspeicher laden. Im Gegensatz dazu ermöglicht die Verwendung von Paging eine Optimierung, das sogenannte *Demand Paging*. Dabei werden Seiten erst dann in den Speicher geladen, wenn sie wirklich gebraucht werden. Wenn ein Prozess erzeugt wird, wird im Swap Space Platz für den gesamten Prozess reserviert. Die Seitentabelleneinträge für den Programmcode und den initialisierten Datenbereich werden speziell markiert (*Fill-from-Text*). Bei einem Zugriff auf eine virtuelle Adresse in diesem Bereich wird dann die entsprechende Seite direkt aus dem ausführbaren File in den physikalischen Speicher eingelagert. Der uninitialisierte Datenbereich wird ähnlich behandelt. Bei ihm wird beim ersten Zugriff auf eine Seite einfach ein Seitenrahmen mit Nullen angefüllt. Diese Sonderbehandlung findet allerdings nur beim ersten Einlagern jeder Seite statt: Ausgelagert werden die Seiten wie beim normalen Paging in den Swap Space.

[14] zumindest nicht ohne Verwendung von `/dev/crystal-ball`

[15] Der Abstand zwischen den Zeigern berträgt in 4.3BSD ca. 1000 Seiten.

Eine weitere Optimierung ist das *Prefetching* bei Demand Paging. Wenn bei einem Seitenfehler eine Seite von einem ausführbaren File eingelagert wird, dann werden die benachbarten Seiten automatisch mit eingelagert. Die Idee dabei ist, dass aufgrund der Lokalität diese Seiten ohnehin bald eingelagert werden müssten.

2.4.6 Die Verwaltung des Hauptspeichers

Bis jetzt haben wir uns mit der Verwaltung des virtuellen Speichers beschäftigt. Dabei haben wir das Problem ignoriert, wie eigentlich die Seitenrahmen des physikalischen Speichers selbst verwaltet werden. Das geschieht mit Hilfe der `mem_map`-Struktur. Diese besteht aus einer Liste von `mem_map_t` Einträgen[16] welche folgende wichtige Elemente beinhalten:

- Die *Anzahl der Referenzen* auf diesen Seitenrahmen.
- Das *Alter des Seitenrahmens* (die Zeit, seit der auf den Seitenrahmen nicht mehr zugegriffen wurde).
- Die *Nummer des physikalischen Seitenrahmens*, den dieser Eintrag beschreibt.

Zusätzlich zur `mem_map`-Struktur existiert noch eine Liste aller freien Seitenrahmen (`free_area`-Struktur). Diese Struktur beschreibt *Blöcke von Seitenrahmen*. Der erste Eintrag beschreibt Blöcke der Größe von einem Seitenrahmen, der zweite Eintrag Blöcke der Größe von zwei Seitenrahmen, der Dritte Blöcke der Größe von vier Seitenrahmen. . . . Jeder Eintrag verweist auf eine doppelt verkettete *Liste von freien Blöcken* der jeweiligen Größe. Zusätzlich enthält jeder Eintrag auch noch einen Zeiger auf eine *Bitmap*, die den Allokationsstatus der Blöcke der jeweiligen Größe beschreibt (wenn z.B. Bit n gesetzt ist, dann ist der n-te Block noch nicht alloziert). Abbildung 2.15 verdeutlicht den Aufbau der `free_area`-Struktur.

Zur Allokation von freien Seitenrahmen wird in Linux das sogenannte *Buddy Verfahren* benutzt. Bei diesem Verfahren werden prinzipiell Blöcke der Größe 2^n Seitenrahmen zugeteilt. Wenn ein freier Block gewünschter Größe (z.B. 2) vorhanden ist, dann wird er vom System zur Verfügung gestellt[17]. Falls dem nicht so ist, wird ein freier Block der nächst größeren Einheit (z.B. 4) in zwei gleich große Blöcke unterteilt und die `free_area`-Struktur entsprechend aktualisiert. Einer der beiden Blöcke (die ja nun die gewünschte Größe haben) wird nun vom System zur Verfügung gestellt[18]. Falls auch kein freier Block der nächst größeren Einheit vorhanden ist, so wiederholt sich der Unterteilungsvorgang rekursiv bis zur maximalen Blockgröße (gesamter physikalischer Speicher).

Bei der Freigabe eines Blocks wird umgekehrt versucht, mehrere benachbarte kleine Blöcke zu einem großen Block zusammenzufassen[19].

[16] Ein Eintrag pro Seitenrahmen

[17] Der Block wird als *belegt* in der jeweiligen Bitmap markiert und aus der Liste der freien Blöcke entfernt.

[18] Wiederum wird der Block als *belegt* in der jeweiligen Bitmap markiert und aus der Liste der freien Blöcke entfernt.

[19] Nach dem Zusammenfassen mehrerer Blöcke muß wieder die Bitmap und die Liste der freien Blöcke aktualisiert werden.

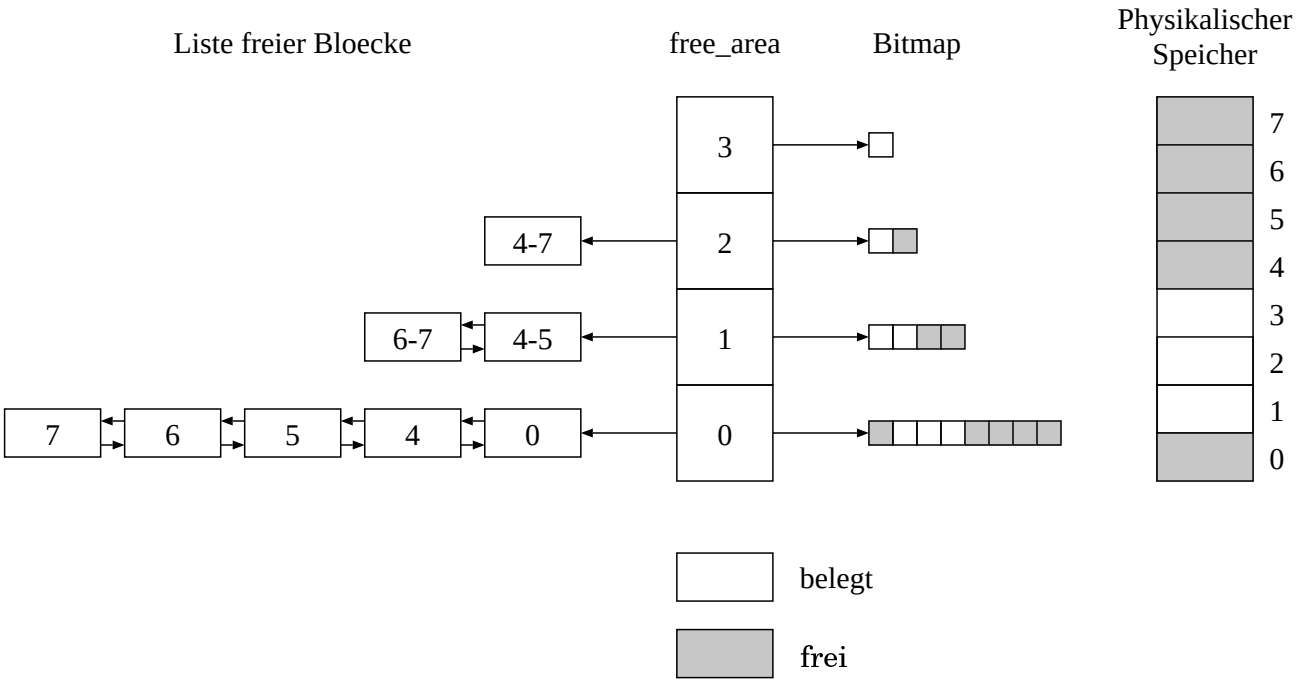


Abbildung 2.15: Verwaltung des Hauptspeichers

2.4.7 Speicherverwaltung beim Erzeugen von Prozessen

Beim Aufruf von `fork()` wird eine identische Kopie eines Prozesses erzeugt. Dazu sind folgende Schritte notwendig:

- Im Swap Space wird Platz für den Kindprozess angelegt.
- Ein neuer `task_struct` Tabelleneintrag wird geschaffen.
- Die Seitentabellen für den Kindprozess werden angelegt.
- Der gesamte Adressraum wird dupliziert. Bei *Fill-on-Demand* Seiten, auf die noch nicht zugegriffen wurde, genügt es, den entsprechenden Seitentabelleneintrag zu kopieren. Alle anderen Seiten müssen kopiert werden.

Es ist offensichtlich, dass das Duplizieren des Adressraumes zeitaufwendig ist. Das ist umso schlimmer, da die meisten Kindprozesse bald nach dem Aufruf von `fork()` mittels `exec()` einen neuen Adressraum anlegen.

Eine effizientere Implementierung von `fork()` benützt das *copy-on-write* Verfahren. Dabei werden nur die Seitentabelleneinträge kopiert, wodurch dann beide Prozesse auf dieselben Seiten zugreifen. Dieses Verfahren findet auch in Linux Verwendung.

Erst wenn einer der beiden Prozesse auf eine Seite schreiben möchte, wird diese dupliziert. Der Vorteil dieses Verfahrens ist, dass nur Seiten kopiert werden, bei denen es auch wirklich notwendig ist.

2.4.8 Speicherverwaltung beim Ausführen von Programmfiles

Beim Ausführen eines exekutierbaren Files wird der gesamte Adressraum des aufrufenden Prozesses gegen einen anderen Adressraum ausgetauscht. Dieser Austausch wird in folgenden Schritten durchgeführt:

- Im Swap Space wird Platz für die neuen Daten- und Stacksegmente angelegt.
- Der Adressraum des Prozesses wird freigegeben, mit Ausnahme der `task_struct`-Struktur.
- Die Seitentabellen werden auf die richtigen Größen für den neuen Adressraum gebracht.
- Die Text-Struktur des Files wird gesucht und, falls sie noch nicht vorhanden ist, neu angelegt. Ebenso werden die Seitentabellen für das Textsegment wie folgt initialisiert:
  - Wenn bereits ein anderer Prozess dieses File ausführt, werden die Seitentabelleneinträge einfach von diesem Prozess kopiert (aus dem Speicher oder vom Swap Space).
  - Sonst werden die Seitentabelleneinträge als *Fill-from-Text* markiert.

2.5 Verwaltung der Peripherie

2.5.1 Struktur des Input/Output (IO) Systems in UNIX

Der Zugriff auf die Peripherie in einem UNIX System geschieht prinzipiell über Device Driver. Es lassen sich im Wesentlichen drei verschiedene Arten von Device Drivers unterscheiden: Character Device Driver, Block Device Driver und Network Interface Driver. Während der Benutzer auf die ersten beiden Typen mehr oder weniger direkt zugreifen kann, wird auf Network Interface Driver nur von Netzwerkprotokollen zugegriffen. Mit Netzwerkprotokollen wird wiederum nur über Sockets kommuniziert. Block Devices können zwar direkt angesprochen werden, im Normalfall wird aber über das Filesystem auf sie zugegriffen (siehe Abbildung 2.16).

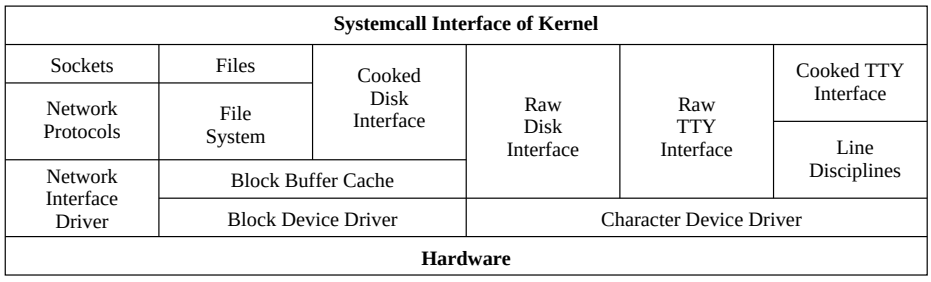


Abbildung 2.16: Struktur des UNIX IO Systems

Der direkte Zugriff auf ein Device geschieht stets über ein Special File. Diese Files sind normalerweise im Directory `/dev` abgelegt (`/dev/tty`: Terminal, `/dev/lp`: Drucker, `/dev/hda1`: IDE Festplattenpartition, ...).

Jedem Device ist eine Major Device Number und eine Minor Device Number zugeordnet, durch die es eindeutig identifiziert werden kann.

Major Device Number Diese Nummer spezifiziert den Typ eines Devices, genauer gesagt, sie spezifiziert den zu verwendenden Device Driver (es ist auch möglich, für verschiedene Devicetypen nur einen Device Driver zu verwenden, z.B. für `/dev/mem` und `/dev/kmem`).

Minor Device Number Diese Nummer hingegen spezifiziert das genaue Gerät aus der Menge der mit dem entsprechenden Treiber verbundenen Geräte, das zu verwenden ist. Sie wird, außer im Treiber selbst, sonst in keinem Teil des Kernels interpretiert und wird dem Treiber als Parameter übergeben. Es ist auch möglich, dass eine Minor Device Number nur einen Teil eines Gerätes kennzeichnet, z.B. eine Partition einer Festplatte.

2.5.2 Device Driver

Ein UNIX Device Driver ist eine Sammlung von Routinen, die benötigt werden, um auf ein Peripheriegerät zugreifen zu können. Abhängig von der Art des Treibers (Block- oder Character) müssen diese Routinen eine gewisse, vorgegebene Funktionalität bereitstellen. Im Kernel sind zwei Tabellen vorhanden (jeweils eine für Block- (`blkdevs`) bzw. Character Driver (`chrdevs`)), die für jeden Device Driver genau einen Eintrag (`device_struct`-Struktur) enthalten. Jeder dieser Einträge enthält den Namen des Treibers und einen Zeiger auf eine Tabelle mit den Einsprungadressen der vom Treiber bereitgestellten Routinen (`file_operations`-Struktur).

Sowohl in der Tabelle für Character Driver als auch in der für Block Driver ist Platz für ein Maximum von `MAX_CHRDEV` bzw. `MAX_BLKDEV` Treiber reserviert. Der Index für die Einträge in diese Tabellen ist die Major Device Number.

Die *Registrierung* von Treibern erfolgt durch den Aufruf der Routine `register_chrdev()` bzw. `register_blkdev()`, welcher die Major Device Number, der Name des Treibers und ein Zeiger auf die `file_operations`-Struktur als Parameter übergeben wird. Diese Registrierung kann nun *statisch* bei der Initialisierung des Kernels oder aber auch *dynamisch* während des Betriebs (mittels der Funktion `request_module()()`) erfolgen.

Dadurch ist es möglich, neue Treiber zum Kernel hinzuzufügen, ohne im Besitz des Source Codes des Kernels zu sein. Es genügt, einen eigenen dynamisch ladbaren Treiber zu schreiben. Dieser wird dann bei Bedarf vom Kernel geladen und in die Tabelle der vorhandenen Treiber eingetragen.

Struktur eines Device Drivers

Normalerweise besteht ein Device Driver aus einer *Top Half* und einer *Bottom Half*. Die Top Half eines Treibers läuft im Kontext des aufrufenden Prozesses. Sie wird während der Exekution eines System Calls aufgerufen. Sollen Daten von einem Gerät gelesen oder auf ein Gerät geschrieben werden, so wird entweder der Datentransfer zum/vom Gerät initiiert, oder, wenn gerade ein anderer Transfer in Gang ist, die Anforderung in eine Liste zur späteren Behandlung geschrieben. Wird gelesen oder ein synchroner Schreibvorgang durchgeführt, so wird in jedem Fall auf die Fertigstellung des Transfers gewartet (die Prozedur führt ein `sleep_on()` aus). Nach Beendigung des

Datentransfers wird vom Gerät ein Interrupt ausgelöst, durch den die Bottom Half des Treibers aufgerufen wird. Die Bottom Half läuft in einem eigenen Kontext und asynchron zum Benutzerprozess. Nachdem der Interrupt behandelt wurde, weckt die Bottom Half die Top Half wieder auf (Prozedur `wake_up()`) und überprüft, ob noch eine weitere Anforderung zum Datentransfer vorliegt. Ist dies der Fall, so wird diese behandelt. Danach terminiert die Bottom Half.

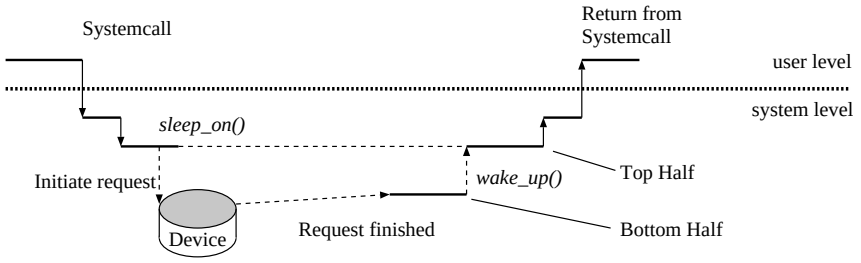


Abbildung 2.17: Durchführung eines Datentransfers von einem Gerät

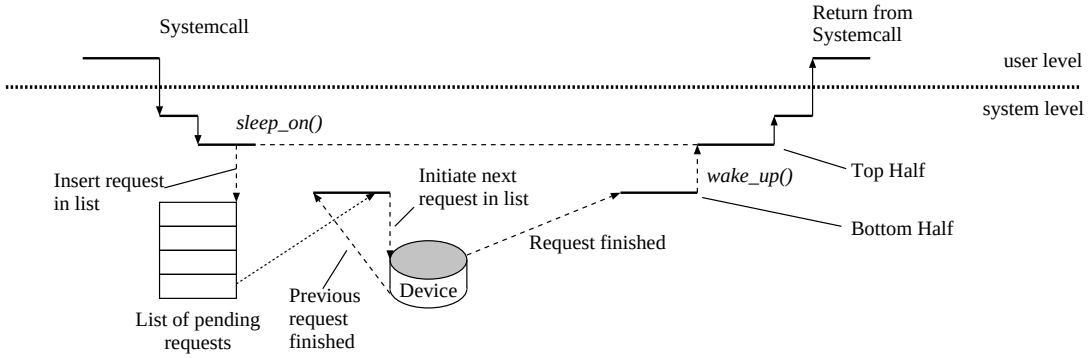


Abbildung 2.18: Durchführung eines Datentransfers von einem Gerät (verzögert)

Die Kommunikation zwischen den beiden Teilen des Treibers erfolgt über Datenstrukturen, auf die beide Prozeduren Zugriff haben. Um den Zugriff auf diese Strukturen ununterbrechbar (atomic) zu machen, ist es notwendig, die Priorität des Prozesses soweit anzuheben, dass er nicht durch einen ankommenden Interrupt unterbrochen werden kann. Abbildungen 2.17 und 2.18 zeigen den Ablauf von Datentransfers von einem Gerät, Abbildung 2.19 einen nicht-synchronen Datentransfer zu einem Gerät.

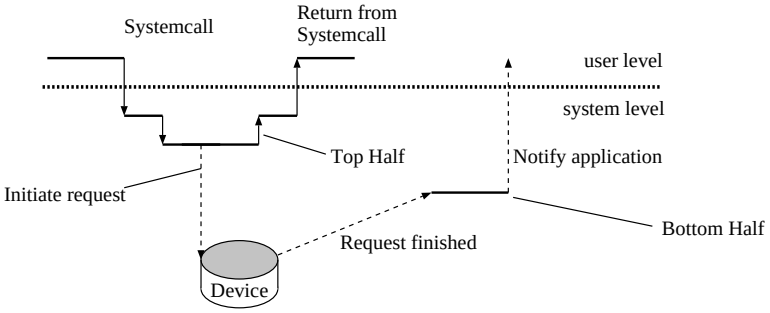


Abbildung 2.19: Durchführung eines nicht-synchronen Datentransfers zu einem Gerät

Software Devices

Es gibt einige Treiber, die nicht mit einem Peripheriegerät verbunden sind. Diese werden als *Software Devices* bezeichnet. Ein Charakteristikum dieser Treiber ist, dass sie über *keine Bottom Half* verfügen.

Die folgende (unvollständige) Liste bietet einige Beispiele für Software Devices:

`/dev/mem` Über dieses Device ist ein direkter Zugriff auf den physikalischen Hauptspeicher möglich. Damit kann direkt auf Programme und Daten des Kernels, aber auch von Benutzerprogrammen zugegriffen werden. Ist dieses Device für alle Benutzer lesbar, so ist dies eine potentielle Sicherheitslücke, da jeder Benutzer auf geheime Daten eines Programmes eines anderen Benutzers zugreifen kann.

`/dev/kmem` Dieses Device hat dieselben Eigenschaften wie `/dev/mem`, mit dem Unterschied, dass auf den virtuellen Hauptspeicher anstatt auf den physikalischen Hauptspeicher zugegriffen wird.

`/dev/null` Dieses Device ist eine Datensenke. Alle Daten, die darauf geschrieben werden, verschwinden. Wird von diesem Device gelesen, so wird nur End of File geliefert.

`/dev/zero` Dieses Device ist eine Datenquelle. Jeder Lesezugriff auf dieses Device produziert die gewünschte Anzahl von Null-Bytes zurück. Dieses Device liefert nie End of File.

`/dev/random` Dieses Device, das auf moderneren Unix-Systemen anzutreffen ist, liefert eine Folge von (echt) zufälligen Bytes. Wenn der Entropie-Pool des Systems erschöpft ist, blockiert das Device Lesezugriffe, bis wieder Zufallsdaten vorhanden sind.

`/dev/urandom` Dieses arbeitet wie `/dev/random`, liefert aber Pseudozufallszahlen, wenn der Entropie-Pool des Systems erschöpft ist.

Als eine weitere Anwendung eines Software Devices wäre z.B. die Implementierung einer RAM Disk möglich.

Block Device Driver

Ein Block Device ist gekennzeichnet durch eine Organisation in Blöcken fixer Größe, auf die wahlfrei zugegriffen werden kann. Die klassische Anwendung für ein Block Device ist eine Festplatte, aber auch Magnetbänder sind normalerweise als Block Devices organisiert. Ein Block Device Driver ist dafür zuständig, dass die Anforderungen zum Lesen und Schreiben von Blöcken in einer sinnvollen und effizienten Reihenfolge ausgeführt werden.

Auf jedes Block Device wird mittels einer gewissen *Strategie* zugegriffen. Die einfachste Möglichkeit ist, alle Übertragungsanforderungen in der Reihenfolge ihres Eintreffens zu behandeln. Da aber normalerweise die *Suchzeiten* einen großen Teil der Gesamtübertragungszeit der Daten ausmachen, ist es viel effizienter, die Anforderungen an das Gerät so zu ordnen, dass die Suchzeiten möglichst minimiert werden.

Die bekanntesten Strategien zum Zugriff auf Block Devices sind die Folgenden:

Elevator Algorithmus Bei diesem Algorithmus wird immer der Request als nächstes bearbeitet, der in der momentanen Richtung als nächster folgt. Ist in der augenblicklichen Richtung kein Request mehr vorhanden, so wird die Richtung gewechselt. Die Bewegung des Schreib/Lesekopfes kann dabei mit der Bewegung eines Aufzuges verglichen werden.

C-SCAN Algorithmus Dieser funktioniert ähnlich dem Elevator Algorithmus, mit dem Unterschied, dass Requests nur bei einer festgelegten Richtung des Kopfes behandelt werden. Bewegt sich der Kopf in die andere Richtung, werden keine Requests behandelt. Die Vorteile gegenüber dem Elevator Algorithmus sind, dass

1. die inneren und äußeren Blöcke nicht benachteiligt werden.
2. die Dichte der Requests am Anfang und am Ende einer Kopfbewegung über die Platte im Mittel gleich groß ist.

Eine graphische Darstellung der beiden Algorithmen ist in Abbildung 2.20 zu finden. Sie zeigt schematisch den Weg des Schreib/Lesekopfs der Platte; wie in der Zeichnung rechts zu erkennen ist, verändert sich durch den „Scan“ (gestrichelte Linie) die Abarbeitungsreihenfolge der Aufträge.

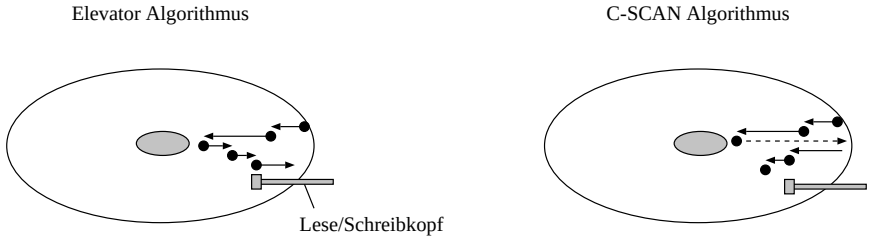


Abbildung 2.20: Vergleich von Elevator- und C-SCAN Algorithmus

Die wichtigsten Routinen eines Block Device Drivers sind:

`open()` Diese Routine bereitet das Device auf spätere Zugriffe vor. Beim Zugriff auf das Block Device über das Filesystem wird diese Routine nur einmal – zum Zeitpunkt des Mountens des Filesystems – aufgerufen. Beim ersten Aufruf dieser Routine nach dem Systemstart werden die internen Datenstrukturen initialisiert.

`release()` Der Aufruf dieser Routine zeigt dem Treiber an, dass das Gerät momentan von keinem Prozess benutzt wird. Erst durch ein neuerliches `open()` kann es wieder verwendet werden. D.h., es gibt keine eins zu eins Abbildung von `open()` und `release()` Aufrufen. `open()` wird für jeden Prozess aufgerufen, der das Gerät benutzen möchte, `release()` wird erst aufgerufen, nachdem alle Prozesse, die ein `open()` aufgerufen haben, das Gerät wieder abgegeben haben.

`strategy()` Diese Routine dient zur Durchführung aller Ein- und Ausgaben auf das Gerät. Sie ist dafür zuständig, alle ankommenden Requests zu ordnen (siehe obige Algorithmen) und für die Interruptroutine aufzubereiten (bzw. einen Request zu starten, wenn das Gerät nicht belegt ist). Da für Lese- und Schreibzugriffe dieselben Strategien anzuwenden sind, gibt es nur eine Routine, die beide Typen von Anforderungen behandelt.

`ioctl()` Mit dieser speziellen Operation können verschiedene Parameter beeinflusst werden.

Als Parameter wird allen diesen Prozeduren die Major und die Minor Device Number des angesprochenen Gerätes übergeben. Der `strategy()` Routine wird außerdem noch ein Zeiger auf eine Struktur übergeben, die einen Systempuffer beschreibt. In diesen Puffer werden die gelesenen Daten geschrieben bzw. werden die auf die Platte zu schreibenden Daten daraus entnommen. Außerdem sind in dieser Struktur alle Daten vorhanden, die benötigt werden, um den Request durchzuführen (Nummer des zu transferierenden Sektors, Lese- oder Schreiboperation, ...). Durch die Verwendung derselben Struktur für die Pufferverwaltung und die Parameterübergabe an den Treiber können unnötige Kopieroperationen vermieden werden.

Character Device Driver

Ein Character Device Driver wird für jede Art von Kommunikation mit der Peripherie verwendet, die nicht in das Schema von Block Devices passt (die Kommunikation über Netzwerke erfolgt normalerweise auch nicht über Character Device Driver, sondern über spezielle Mechanismen, wie z.B. Sockets). Dennoch kann man zwei Hauptgruppen von Character Devices unterscheiden:

Raw Devices dienen ebenso wie Block Devices dem wahlfreien Zugriff auf Peripheriegeräte. Dabei wird jedoch der Block Buffer Cache umgangen, und das Peripheriegerät muss auch nicht, wie bei Block Devices, in Blöcken fixer Größe organisiert sein. Geräten, auf die standardmäßig als Block Device zugegriffen wird, ist üblicherweise auch ein Raw Device zugeordnet. Bei der Verwendung solcher Geräte muss jedoch beachtet werden, dass beim Zugriff auf ein- und denselben Block über das Block bzw. Raw Device verschiedene Daten gelesen werden können, da das Raw Device den Block Buffer Cache umgeht.

Character Oriented Devices dienen dem Zugriff auf zeichen- oder zeilenorientierte Geräte, wie Terminals oder Drucker.

Die Routinen, mit denen auf Raw Devices und auf Character Oriented Devices zugegriffen wird, sind prinzipiell dieselben, jedoch werden bei Raw Devices einige nicht benutzt:

`open(),release()` Diese Routinen haben im großen und ganzen dieselben Funktionen wie bei den Block Devices. Kann auf ein Gerät sowohl als Block als auch als Character Device zugegriffen werden, so sind diese Routinen normalerweise für beide Treiber identisch.

`write()` Diese Routine dient zum Schreiben von Daten auf Character Devices. Die Daten werden üblicherweise in einen privaten Puffer des Treibers kopiert; die Bottom Half des Treibers greift darauf zu und gibt sie – einzeln oder blockweise – auf das Gerät aus.

`read()` Zeichen, die die Interruptroutine in einen privaten Puffer des Treibers kopiert hat, werden an das aufrufende Programm übergeben.

`ioctl()` Eine spezielle Operation (Setzen oder Lesen von Parametern, Ausschalten von Geräten, Positionierung von Magnetbändern, ...) wird durchgeführt.

`select()` Es wird überprüft, ob zu lesende Daten vorhanden sind, bzw. ob Platz zum Schreiben von Daten verfügbar ist. Diese Routine wird in Zusammenhang mit dem `select()` Systemcall verwendet. Sie kann nicht für Raw Devices verwendet werden (da keine Daten gepuffert werden).

Der Datenaustausch mit den Character Device Routinen geschieht über die `uio`-Struktur. Diese liefert alle nötigen Informationen, um einen Datentransfer mit einem Gerät durchzuführen.

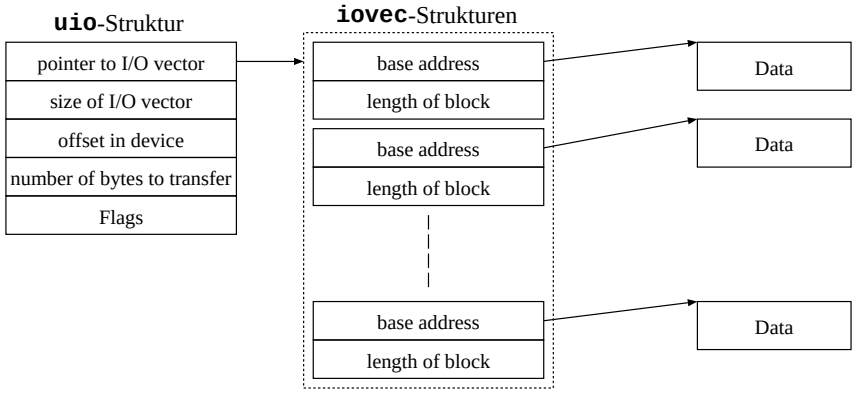


Abbildung 2.21: Struktur (`uio`-Struktur) zum Datenaustausch mit einem Character Device Driver

Sie enthält die Länge der zu übertragenden Daten, die Position im Gerät, wohin die Daten geschrieben bzw. woher sie gelesen werden sollen (nur bei Devices mit wahlfreiem Zugriff) und einen Zeiger auf ein Feld für scatter/gather IO. Dieses Feld besteht aus einer Anzahl von Strukturen (`iovec`-Struktur), die jede die Adresse und die Länge eines Datenblocks spezifizieren (das ist dieselbe Struktur, die bei den `readv()` und `writev()` Systemcalls verwendet wird). Dadurch ist es möglich, Daten, die an unterschiedlichen Adressen im Speicher stehen, mit einem Systemcall (in einer Atomic Action) zu übertragen. Die `uio`-Struktur enthält weiters zwei Flags, die bestimmen, ob es sich um eine Lese- oder Schreiboperation handelt, und ob die Datenpuffer im Benutzer- oder im Kerneldatenbereich liegen. Liegen sie im Kerneldatenbereich, so können die Daten direkt aus den Puffern gelesen oder in die Puffer geschrieben werden, da der Treiber ja auch im Kernel-Kontext exekutiert wird, ansonsten müssen spezielle Systemprozeduren für das Kopieren der Daten vom bzw. in den Benutzerdatenbereich verwendet werden (siehe Abbildung 2.21).

Line Disciplines Character Device Driver, die mit Terminals verbunden sind, haben viele Aufgaben durchzuführen, die weder vom verwendeten Treiber noch vom Programm, das vom Terminal liest oder auf das Terminal schreibt, abhängen (Behandlung von *erase* und *kill* Zeichen, Behandlung von Interruptsequenzen (Ctrl-c), getippte Zeichen auf den Bildschirm schreiben (*echo*), ...). Diese Routinen, die im Datenstrom zwischen Systemcall und Bildschirmtreiber liegen, werden *Line Disciplines* genannt.

Line Disciplines sind Kernel-Routinen, die vom Bildschirmtreiber nach dem Empfang eines Zeichens und vor dem Senden eines Zeichens aufgerufen werden, und diese Zeichen entsprechend gewisser Regeln behandeln. Sie werden aber sonst vom Kernel nicht weiter unterstützt, d.h., der Treiber selbst muss die entsprechenden Routinen der Line Discipline aufrufen. Mittels `ioctl()` Aufrufen kann einem Treiber mitgeteilt werden, die Line Discipline zu wechseln.

Line Disciplines können außer für die Kommunikation mit einem Terminal auch für andere Zwecke, zum Beispiel für die Durchführung eines Datenübertragungsprotokolls, verwendet werden. Ein Beispiel dafür ist die in einigen UNIX Versionen enthaltene SLIP (Serial Line IP) Line Discipline.

2.5.3 Streams

Das klassische Device Driver Schema von UNIX enthält einige konzeptionelle Schwächen, vor allem die Kommunikation über Netzwerke ist über Character Devices kaum effizient durchzuführen:

- Character Devices behandeln den Datenstrom zeichenweise, die Kommunikation über Netzwerke ist aber nachrichtenorientiert. Das beeinträchtigt die Performance, außerdem ist das Beibehalten von Nachrichtengrenzen für gewisse Anwendungen unerlässlich.
- Für verschiedene Treiber werden oft dieselben Protokolle verwendet. Der Mechanismus, den UNIX zur Verfügung stellt, um die Mehrfachimplementierung gleicher Protokolle zu vermeiden, sind die Line Disciplines. Da Protokolle aber oft auf anderen Protokollen aufbauen (z.B. TCP/IP, OSI), muss es möglich sein, mehrere Protokolle hintereinander auf einem Treiber aufzusetzen. Das ist bei Line Disciplines nicht der Fall. Eine prinzipielle Schwäche von Line Disciplines ist, dass sie vom Treiber selbst aufgerufen werden müssen, obwohl sie logisch zwischen Treiber und Systemcall liegen würden. Die Benutzung von Line Disciplines ist daher nicht transparent für den Treiber.

Um diese Schwächen zu beheben, wurde in System V das Konzept der *Streams* eingeführt. Streams sind eine Verallgemeinerung von Line Disciplines, zeichnen sich jedoch durch große Modularität und gute Performance aus.

Ein Stream ist eine *bidirektionale Verbindung* zwischen einem Prozess und einem Treiber. Ein Stream besteht aus einer Anzahl von *Modulen*; das erste ist der *Stream Head*, das letzte ein *Device Driver*. Jedem der Module sind zwei Queues, eine zur Eingabe, die andere zur Ausgabe, zugeordnet. Daten wandern entweder vom Stream Head durch alle Module zum Treiber, oder vom Treiber durch alle Module bis zum Stream Head. Ein Modul empfängt Daten vom vorhergehenden Modul, bearbeitet sie und übergibt sie danach dem nächstfolgenden Modul.

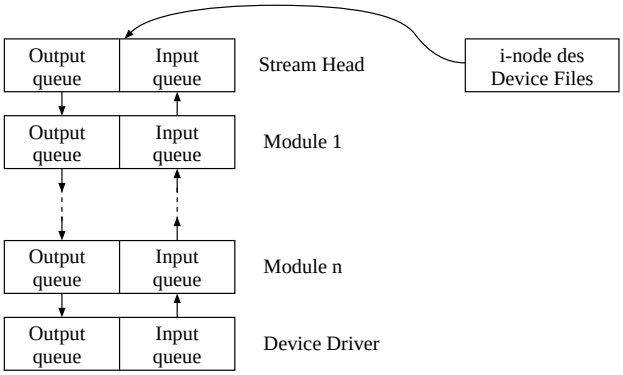


Abbildung 2.22: Aufbau von Streams

Streams scheinen im Filesystem als *Character Special Files* auf, und sie verhalten sich aus der Sicht des Anwenders auch wie solche (ausgenommen einige `ioctl()` Aufrufe). Abbildung 2.22 zeigt den Aufbau von Streams.

In einer *Queue* werden Daten nicht als unstrukturierte Folge von Zeichen abgespeichert, vielmehr enthält jede Queue eine beliebige Anzahl von Nachrichten (z.B. erzeugt jeder einzelne `write()` Systemcall eine Nachricht; von den unterlagerten Modulen kann jedoch eine Nachricht in mehrere Nachrichten aufgespalten werden, bzw. können mehrere Nachrichten in eine zusammengefasst werden), die wieder aus einer Anzahl von Datenblöcken besteht.

Den Nachrichten eines Streams können *Prioritäten* zugewiesen werden. Höherpriore Nachrichten werden in den Queues immer vorgereiht. Auf diese Weise können z.B. Kontrolldaten schnell weitergegeben werden oder *out-of-band* Daten übertragen werden (siehe Kapitel 2.6.3).

Jedes Modul wird nach außen durch vier Prozeduren repräsentiert:

`open(),close()` Diese Prozeduren entsprechen im Wesentlichen den gleichnamigen Prozeduren von Treibern.

`put()` Über diese Prozedur wird einem Modul eine Nachricht übergeben. Sie kann entweder von einem anderen Modul oder von einem Systemcall aufgerufen werden. Diese Nachricht wird entweder in der dem Modul zugewiesenen Queue zur späteren Behandlung gespeichert oder sofort an das nächste Modul weitergegeben. Wird die Nachricht in irgendeiner Weise bearbeitet, so ist zu beachten, dass die Bearbeitungsroutine nicht blockierend sein darf, da die `put()` Routine auch im Kontext eines Interrupts aufgerufen werden kann (ein Interrupt darf nicht blockieren!).

`service()` Diese Prozedur ist für das Bearbeiten von Nachrichten, die in der lokalen Queue eines Moduls gespeichert sind, zuständig. Kann die `put()` Prozedur eine Nachricht nicht sofort behandeln, weil entweder noch mehr Nachrichten für die Bearbeitung benötigt werden, oder weil die Daten vom nächstfolgenden Modul nicht angenommen werden können (z.B. weil dessen Puffer voll ist), so werden sie in der Queue gespeichert. Die `service()` Routinen aller Module, deren Queues nicht leer sind, werden von einem speziellen Scheduler, der nur für diese Routinen zuständig ist, aktiviert und verwaltet (es wäre auch möglich, diese Routinen wie normale Tasks zu schedulen. Da aber sehr viele dieser Module gleichzeitig aktiv sein können, wäre der Overhead für den Scheduler sehr groß). Die `service()` Routinen können im Gegensatz zu den `put()` Routinen auch blockierend sein, d.h., sie können auf Ereignisse warten (Timeout, nächstes Modul ist aufnahmebereit, neue Nachricht kommt an, ...).

Konfiguration von Streams

Nach dem Öffnen eines Special Devices, das einen Stream repräsentiert, besteht der Stream üblicherweise nur aus einem Stream Head und einem Treiber. Der Stream Head ist für alle Module gleich und hat zwei Funktionen:

- Führt ein Systemcall ein `write()` auf dem Stream aus, so übergibt der Stream Head die Daten ohne Bearbeitung an das nächstfolgende Modul. Ist dieses nicht aufnahmefähig, so werden die Daten in der internen Queue gespeichert und die `service()` Routine wird aktiviert.

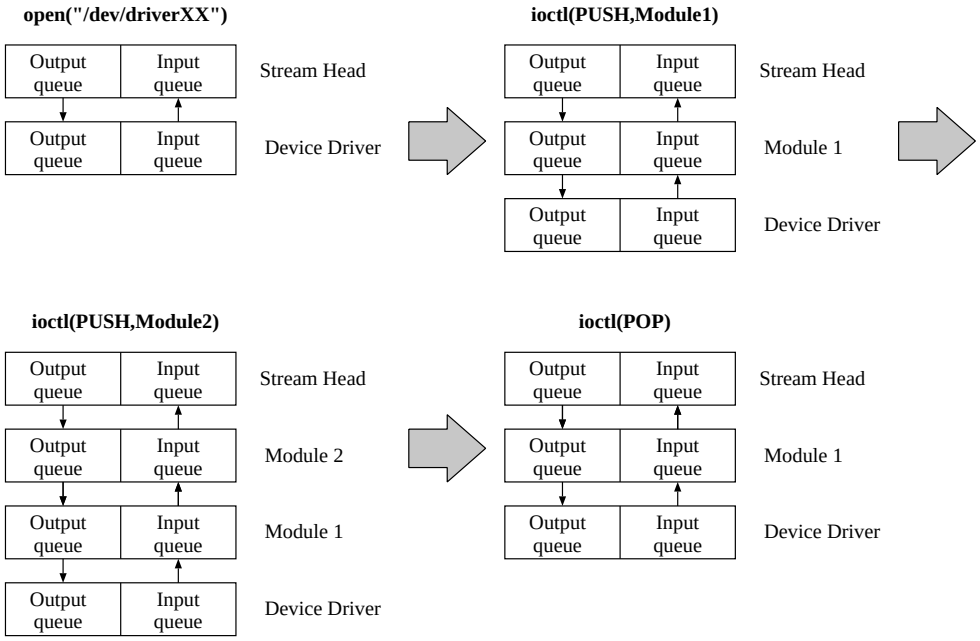


Abbildung 2.23: Hinzufügen und Entfernen von Stream-Modulen

- Werden Daten von einem darunterliegenden Modul empfangen, so werden sie in der internen Queue gespeichert, wo sie von einem `read()` Systemcall ausgelesen werden können.

Ein neues Modul wird in einen Stream mittels eines `ioctl()-PUSH` Systemcalls hinzugefügt, mittels `ioctl()-POP` wird ein Modul entfernt. Von diesen Funktionen ist jeweils nur das direkt dem Stream-Head nachfolgende Modul betroffen (Es kann nur nach dem Stream-Head ein neues Modul eingefügt werden, bzw. es kann nur das oberste Modul entfernt werden). Werden mehrere Module in einem Stream eingefügt, ist daher die Reihenfolge der `ioctl()`-PUSH Aufrufe entscheidend.

In neueren UNIX Versionen ist es auch möglich, Multiplexer-Module zu implementieren. Dabei ist ein Modul mit mehreren darunterliegenden bzw. darüberliegenden Modulen verbunden. Dabei hat sich das Multiplexer Modul darum zu kümmern, dass die Nachrichten an das richtige Modul weitergeleitet werden.

2.5.4 Pseudo TTYs

In einigen UNIX Versionen (und auch in Linux) gibt es einen speziellen Typ von Character Special Devices, die sogenannten *Pseudoterminals (PTYs)*. Die Existenz von Pseudoterminals begründet sich historisch aus der Ablöse der mit einem Mainframe verbundenen physikalischen Terminals durch (virtuelle) Terminals z.B. für das X Window System (`xterm`) und die Möglichkeit sich via `telnet` oder `ssh` mit dem Rechner zu verbinden. Anstatt eines physikalischen Terminals übernimmt ein Prozess die Rolle der Hardware.

Ein Pseudo TTY besteht aus zwei Character Special Files, `/dev/tty??` und `/dev/pty??`, die miteinander verbunden sind.

Die mit `/dev/pty` bezeichnete Seite des PTY nennt man *Master*, die mit `/dev/tty` bezeichnete

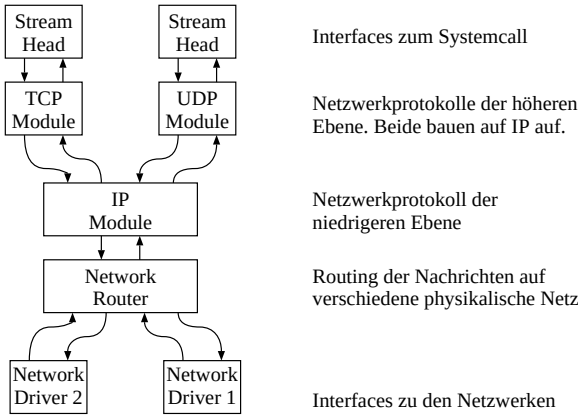


Abbildung 2.24: Beispiel für die Anwendung von Multiplex Modulen

Seite *Slave*. Die Slave-Seite verhält sich wie ein normales Terminal Device File. Allerdings werden die Zeichen, die auf dieses File geschrieben werden, nicht an ein "echtes" (Hardware-) Terminal geschickt, sondern können von einem Prozess von der Master-Seite gelesen werden. Umgekehrt stammen die Zeichen, die von der Slave-Seite gelesen werden können, nicht von einem echten Terminal, sondern sind von einem Prozess auf der Master-Seite geschrieben worden. Die Master-Seite kann immer nur von einem Prozess exklusiv geöffnet werden. Dieser Prozess ist für die Verarbeitung aller Zeichen zuständig, die über das Pseudoterminal fließen. Die Slave-Seite lässt sich, wie jedes Terminal, mehrfach öffnen.

Pseudoterminals werden verwendet, wenn man mit einem Programm, das direkt mit einem Terminal verbunden sein muss (etwa ein interaktives Programm, beispielsweise ein Editor) kommunizieren möchte, ohne das Programm zu verändern. Man schreibt dann ein Programm, das die Master-Seite des Pseudoterminals öffnet und sorgt dafür, dass das andere Programm die Slave-Seite des Pseudoterminals als *Controlling Terminal* ansieht. So funktioniert etwa beim X Windows System die Terminalemulation `xterm`. `xterm` ist ein Programm, das VT 100 Terminalsteuerungssequenzen versteht und in einem Window auf einer Workstation ein Terminal simuliert. Dazu öffnet `xterm` ein `pty`/`tty`-Paar und spaltet sich in zwei Prozesse. Der Elternprozess liest von der Tastatur und schreibt die Zeichen auf die Master-Seite des PTY. Außerdem liest er von der Master-Seite und zeigt die gelesenen Zeichen in einem Fenster an. Der Kindprozess löst sich vom momentanen Controlling Terminal, öffnet die Slave-Seite des PTY und setzt sein `stdin`, `stdout` und `stderr` auf dieses. Dann führt er eine Shell aus. Für die Shell und für alle von ihr gestarteten Prozesse verhält sich die Slave-Seite genau wie ein normales Terminal. Abbildung 2.25 vergleicht ein solches Szenario mit der Verwendung eines "normalen" Terminals.

In neueren Versionen von Linux[20] wurde das Namensschema für Pseudo TTYs dem von System V bzw. Unix98 angepaßt. In diesen Versionen wird statt mehreren Device Files für die Master Seite (`/dev/pty??`) ein einziges (gemeinsames) Device File (`/dev/ptmx`) verwendet. Das Öffnen dieses Device Files bewirkt die Generierung eines neuen (bisher unbenutzten) PTYs. Die Device Files für die Slave Seite befinden sich in einem gesonderten Verzeichnis (`/dev/pts`) und werden mit Dezimalzahlen benannt (z.B. `/dev/pts/42`). Auf diese Weise wurde einerseits das bisherige Problem des limitierten Namensraums für Pseudo TTYs beseitigt, andererseits gibt dieses Namens-

[20] ab Kernel Version 2.2.x

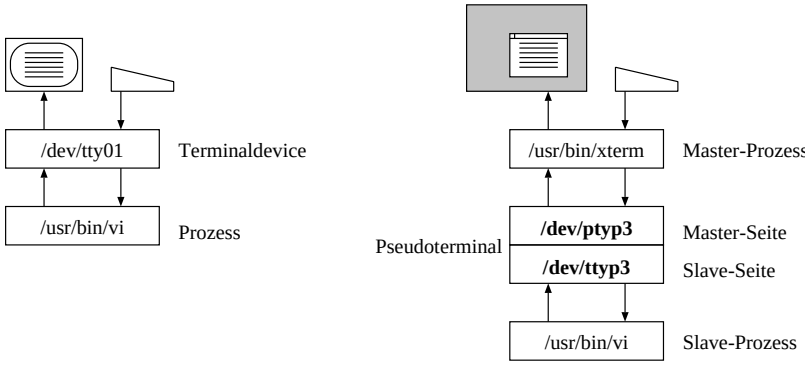


Abbildung 2.25: Szenario für die Verwendung von `ttys` und `ptys`

schema dem Kernel die Möglichkeit, automatisch (on-demand) neue Device Files für die Slaves zu generieren.

2.6 Interprozesskommunikation

In den ersten UNIX Versionen (bis AT&T UNIX Version 7) gab es zur Synchronisation von Prozessen und zur Kommunikation zwischen Prozessen zwei Mechanismen: Signale und Pipes. Beide Mechanismen weisen, zumindest in ihrer ursprünglichen Form, entscheidende Schwächen auf:

- Signale waren ursprünglich nur zur Behandlung von Ausnahmezuständen, z.B. dem Versuch eines Benutzers, ein Programm zu terminieren, gedacht. Da es jedoch keine anderen Mechanismen zur Synchronisation von Prozessen gab, wurden Signale dafür 'zweckentfremdet'. Es ist jedoch nicht möglich, mit Version 7 Signalen eine zuverlässige Prozesssynchronisation durchzuführen. Um dies zu ändern, wurde in BSD UNIX der Signalmechanismus vollkommen neu implementiert.

- Unnamed Pipes können nur zur Kommunikation zwischen einem Prozess und einem 'Nachfahren' dieses Prozesses verwendet werden (außer die Pipe ist in einem Vorgängerprozess beider Prozesse erzeugt worden). Die Kommunikation zwischen zwei Prozessen eines Benutzers, die unabhängig voneinander in der Shell gestartet wurden, oder gar zwischen den Prozessen zweier verschiedener Benutzer, ist nicht möglich. Dieser Nachteil wird allerdings von *Named Pipes* behoben. Named Pipes werden explizit mit `mkfifo()` angelegt. Das resultierende *special file* kann dann von unabhängigen Prozessen zum Lesen bzw. Schreiben geöffnet werden. Ein weiterer gravierender Nachteil von Pipes (sowohl Named als auch Unnamed) ist, dass sie nur lokal verwendet werden können. Zwei Prozesse auf verschiedenen Computern, die durch ein Netzwerk miteinander verbunden sind, können nicht über Pipes miteinander kommunizieren.

In den neueren UNIX-Versionen wurden verschiedene Mechanismen eingeführt, um diese Schwächen in der Interprozesskommunikation zu umgehen:

- In *BSD* wurden *zuverlässige Signale* eingeführt, die folgende Schwächen von unzuverlässigen Signalen beseitigten:
  - Signale konnten verlorengehen
  - Signale konnten abgefangen oder ignoriert, aber nicht blockiert werden
  - Die Aktion des Signals wurde bei jedem Auftreten zurückgesetzt

  Die Kommunikation zwischen verschiedenen Prozessen, auch auf verschiedenen Computern, wurde durch die Einführung der *Sockets* ermöglicht.
- In *System V* wurden *Semaphore*, *Shared Memory* und *Message Queues* eingeführt. Damit ist die Synchronisation und Kommunikation von beliebigen Prozessen, die allerdings alle auf demselben Computer ausgeführt werden müssen, möglich. Kommunikation zwischen Prozessen auf verschiedenen Computern ist mit *Streams* (vgl. Kapitel 2.5.3) möglich.

Linux unterstützt sowohl die BSD IPC Mechanismen (zuverlässige Signale und Sockets), als auch die System V Mechanismen (Semaphore, Shared Memory und Message Queues). System V Streams werden zur Zeit in Linux nicht unterstützt.

2.6.1 Signale

Signale werden in UNIX sowohl zur Behandlung asynchroner Ereignisse (Prozessor Exception, Signal von der Tastatur, Aufruf von `kill()` zum Terminieren eines Prozesses, ...), als auch zur Synchronisation von parallelen Prozessen verwendet.

Das Senden eines Signals an einen Prozess kann, in Abhängigkeit vom jeweiligen Signal selbst und von Aktionen, die der Prozess zur Behandlung des Signals durchgeführt hat, verschiedene Reaktionen hervorrufen:

- Das Signal wird ignoriert.
- Der Prozess wird terminiert.
- Ein core-File wird erzeugt und der Prozess wird terminiert.
- Der Prozess wird gestoppt.
- Eine benutzerdefinierte Signalbehandlungsroutine wird aufgerufen.

Signale können von Benutzerprogrammen blockiert werden (außer einigen Signalen mit spezieller Bedeutung). Das bedeutet, dass die entsprechenden Signale zwar empfangen und gespeichert werden, die entsprechende Aktion aber erst dann ausgeführt wird, wenn die *Blockierung* wieder aufgehoben wird. Der Systemcall `sigpause()` führt zwei Aktionen durch: Zuerst wird eine neue Signalmaske gesetzt, danach wird auf das Eintreffen eines Signals gewartet. Dadurch kann verhindert werden, dass ein Signal "verlorengeht", wie das in früheren UNIX Versionen möglich war (siehe Abbildung 2.26).

```
/*
 * Die Uebertragung des Signals wird blockiert
 */
sigsetmask(1 << (SIGUSR1 - 1));

[...]

/*
 * Trifft das Signal ein, wird es gespeichert
 */

[...]

/*
 * Die Blockierung des Signals wird aufgehoben.
 * Wenn das Signal bereits eingetroffen ist, wird
 * die Exekution sofort fortgesetzt, ansonsten wird
 * sie fortgesetzt, sobald das Signal eintrifft
 */
sigpause(0);
```

Abbildung 2.26: Sichere Übertragung eines Signals (Empfänger)

Das Signal-API

Zum Senden von Signalen an andere Prozesse dient die Funktion `kill()`:

```
int kill (pid_t pid, int signum);
```

Ein Aufruf sendet das Signal *signum* zum Prozess *pid*. Für *signum* sind in `<signal.h>` Konstantenten definiert, etwa `SIGTERM`, `SIGINT`, `SIGKILL`, ...

Zum Einrichten von Signal-Handlern gibt es zwei Funtionen; `signal()` ist in ANSI-C definiert, `sigaction()` ist Teil des POSIX-Standards.

```
typedef void (*sighandler_t) (int);

sighandler_t signal (int signum, sighandler_t action);
```

Die Definition von `sighandler_t` ist nicht Teil des Standards, wird aber auf GNU-Systemen zur Verfügung gestellt. Das erste Argument ist wieder eine Konstante für die Signalnummer, das zweite die Funktion die bei Eintreffen des spezifizierten Signals ausgeführt werden soll. Dieser Funktion wird dann die Signalnummer als Argument übergeben. Anstatt einer Funtion können allerdings auch folgende Konstanten benutzt werden:

`SIG_DFL` Standard-Aktion für das Signal. Oft ist das die Programmbeendigung.

`SIG_IGN` Signal ignorieren.

`signal()` gibt die zuletzt aktive Signalbehnadlungsroutine zurück oder liefert im Fehlerfall `SIG_ERR`.

`sigaction()` erlaubt mehr Kontrolle als `signal()` – so kann etwa angegeben werden welche Signale während des Aufrufs des Handlers blockiert werden sollen.

```
struct sigaction
{
  sighandler_t sa_handler;
  sigset_t sa_mask;
  int sa_flags;
};

int sigaction (int signum,
               const struct sigaction *action,
               struct sigaction *old_action);
```

`sigaction()` setzt die in *action* spezifizierte Signalbehandlungsroutine in Kraft und speichert die zuvor aktive Routine und die zugehörigen Parameter in *old_action*. Die `struct sigaction` setzt sich zusammen aus: `sa_handler`, der dem *action*-Argument von `signal()` entspricht, `sa_mask`, das die Signale spezifiziert, die während des Ablaufs der Signalbehandlunsgroutine blockiert werden sollen und `sa_flags`, in dem verschiedene Flags, die auf des Verhalten des Signals einen Einfluss haben, angegeben werden können.

Implementierung von Signalen

In der `task_struct`-Struktur befinden sich zwei Bitfelder `signal` und `blocked`, die für jedes Signal angeben, ob es gerade *pending* ist oder ob es *blockiert* werden soll. Weiters existiert ein Array von Zeigern auf *benutzerdefinierte Signalbehandlungsroutinen*.

Das Senden eines Signals an einen Prozess wird von zwei Routinen (`send_sig()` und `generate()`) durchgeführt. Diese führen folgende Aktionen aus (siehe Abbildung 2.27):

1. Ist der Prozess im Zustand *stopped*, so wird er nur durch die Signale `SIGCONT` und `SIGKILL` (da dieses Signal auch gestoppte Prozesse terminiert) aufgeweckt.
2. Ist das empfangene Signal ein `SIGSTOP`-Signal, so geht der Prozess in den Zustand *stopped* über (ein gesetztes `SIGCONT`-Signal wird hierbei gelöscht).
3. Ist das Signal nicht blockiert und soll das Signal ignoriert werden, dann ist die Signalbehandlung beendet.
4. Nun wird das Signal in die Menge der wartenden Signale eingetragen.
5. Ist der Prozess im Zustand *interruptable waiting*, so wird er aufgeweckt.

```
void send_sig(unsigned long sig, struct task_struct * p)
{
    if ((sig == SIGKILL) || (sig == SIGCONT))          /* (1) */
    {
        if (p->state == STOPPED)
        {
            wake_up_process(p);
        } /* if */
    } /* if */

    if (sig == SIGSTOP)                                /* (2) */
    {
        p->signal &= ~(1 << (SIGCONT - 1));
    } /* if */

    generate(sig, p);
} /* send_sig() */

void generate(unsigned long sig, struct task_struct * p)
{
    unsigned long mask = 1 << (sig - 1);

    if (!(p->blocked & mask) &&                        /* (3) */
        (p->sig[sig]->action == SIG_IGN))
    {
        return;
    } /* if */

    p->signal |= mask;                                 /* (4) */

    if ((p->state == INTERRUPTABLE_WAITING) &&         /* (5) */
        (p->signal & ~p->blocked))
    {
        wake_up_process(p);
    } /* if */
} /* generate() */
```

Abbildung 2.27: Behandlung von Signalen – Teil 1

Jedesmal, wenn ein Prozess neu gescheduled wird, überprüft er, ob ein Signal in der Menge der wartenden Signale eingetragen und nicht maskiert ist. Ist dies der Fall, so wird eine von zwei möglichen Aktionen ausgeführt (siehe Abbildung 2.28):

1. Ist das Signal vom Benutzer nicht abgefangen, so wird das *Programm terminiert*, eventuell nach dem Erzeugen eines core-Images[21].

[21] Die Möglichkeiten 'Stoppen des Prozesses' und 'Ignorieren des Signals' müssen an dieser Stelle nicht mehr be-

```
void do_signal(unsigned long oldmask, struct pt_regs * regs)
{
    unsigned long sig;

    for (sig = 1; sig < NSIG; sig++)
    {
        unsigned long mask = 1 << (sig - 1);

        if ((current->signal & mask) && !(current->blocked & mask))
        {
            if (current->sig[sig]->action == SIG_DFL)  /* (1) */
            {
                if (default_action[sig] == CORE)
                {
                    core_dump(signr, regs);
                }

                do_exit(signr);
            } /* if */
            else                                        /* (2) */
            {
                p->blocked |= mask;
                setup_frame(current->sig[signr]->action, regs,
                            signr, oldmask);
            } /* else */
        } /* if */
    } /* for */
} /* do_signal() */
```

Abbildung 2.28: Behandlung von Signalen – Teil 2

2. Eine vom *Benutzer definierte Signalbehandlungsroutine* wird aufgerufen. Dazu wird zuerst im Bitfeld `blocked` das soeben empfangene Signal blockiert, damit rekursive Aufrufe der Signalbehandlungsroutine verhindert werden. Dann wird der User-Stack mittels der Routine `setup_frame()`[22] derart manipuliert, dass sofort nach der Rückkehr des Prozesses in den User-Modus die Signalbehandlungsroutine exekutiert wird (die Überprüfung, ob ein Signal empfangen wurde, geschieht natürlich noch im Kernel-Kontext des Prozesses).

2.6.2 Semaphore, Shared Memory, Message Queues

In System V wurden neue Mechanismen zur Interprozesskommunikation eingeführt, die einige der Schwächen der bis dahin vorhandenen Mechanismen ausgemerzt haben: Semaphore, Shared Memo-

trachtet werden, da sie bereits durch `send_sig()` behandelt wurden.

[22] Eigentlich wird hier zuerst die Routine `handle_signal()` aufgerufen, welche ihrerseits schließlich `setup_frame()` aufruft.

ry und Message Queues. Sie ermöglichen die Synchronisation und Kommunikation zwischen unabhängigen Prozessen. Sie sind sehr zuverlässig zu verwenden (vergleiche z.B. Semaphore mit den traditionellen UNIX Signalen) und ermöglichen teilweise sehr effizienten Datenaustausch (Shared Memory). Der Nachteil dieser Mechanismen ist, dass sie nur lokal auf einem Computer eingesetzt werden können, und nicht netzwerkfähig sind.

Semaphore können sowohl als binäre Semaphore als auch als mehrwertige Semaphore verwendet werden. Durch die Möglichkeit der Verwendung von Semaphorefeldern, die in *Atomic Actions* abgefragt bzw. gesetzt werden können, können leichter Synchronisationsmechanismen ohne die Gefahr eines Deadlocks implementiert werden.

Shared Memories können direkt in die Adressbereiche der Tasks, die sie verwenden, eingebunden werden. Dabei können sie entweder vorhandene Adressbereiche überdecken, oder sie können an das Datensegment des Prozesses angefügt werden. Beliebig viele Prozesse können auf ein- und dasselbe Shared Memory zugreifen.

Message Queues werden verwendet, um zwischen beliebig vielen Prozessen eines Computers Nachrichten auszutauschen. Es ist möglich, Nachrichtentypen zu vergeben; diese können verwendet werden, um speziell Nachrichten dieses Typs zu empfangen, oder um Nachrichten mit Prioritäten zu versehen.

Details über die Implementierung dieser System V IPC Mechanismen in Linux können in [Rus98] gefunden werden.

2.6.3 Sockets

Um die traditionellen Schwächen von UNIX in der Interprozesskommunikation zu überwinden, wurde in BSD UNIX der *Socket* Mechanismus eingeführt. Sockets ermöglichen die Kommunikation zwischen unabhängigen Prozessen, welche auch auf verschiedenen Computern exekutiert werden können. Da Sockets auch ein allgemeines Interface zu Netzwerkprotokollen darstellen, kann unter Benutzung von Sockets auch mit Computern, die nicht unter UNIX laufen, kommuniziert werden.

Sockets stellen einen sehr allgemeinen Mechanismus zur Interprozesskommunikation dar, der andere Mechanismen, wie z.B. Pipes, überflüssig macht (tatsächlich sind in BSD Pipes durch Sockets realisiert[23]).

Sockets bieten den Vorteil der *Transparenz*, d.h., die Kommunikation ist unabhängig davon, ob die beiden Prozesse auf derselben Maschine ablaufen oder nicht. Ferner ist die Kommunikation über Sockets bis zu einem gewissen Grad *kompatibel* zu den schon vorher in UNIX vorhandenen Befehlen zur IO: Die `read()` und `write()` Systemcalls können weiterhin verwendet werden. Ein Unterschied besteht jedoch in der Syntax beim Verbindungsaufbau (der `open()` Systemcall kann für Sockets nicht verwendet werden).

Ein Socket stellt den Endpunkt einer Kommunikationsverbindung dar. Bei der Erzeugung mittels des Systemcalls `socket()` werden ihm eine Domain, ein Typ und ein Protokoll zugewiesen.

[23] In Linux sind Pipes mittels des VFS (siehe Kapitel 2.2.2) implementiert.

```
int socket(
    int domain,
    int type,
    int protocol);
```

Die *Domain* gibt an, in welchem Bereich eine Verbindung zum Socket aufgebaut werden kann. Linux unterstützt derzeit u.a. folgende Domains:

AF_UNIX die UNIX-Domain. Diese Domain erstreckt sich über einen UNIX Rechner. Zwei Prozesse auf demselben Computer können über einen Socket in der UNIX-Domain miteinander kommunizieren.

AF_INET die Internet-Domain. Diese Domain fasst alle Computer, die an das Internet angeschlossen sind, zusammen. Da das Internet ein weltumspannendes Netz ist, können zwei Prozesse auf verschiedenen Computern (sogar in verschiedenen Ländern oder auf verschiedenen Kontinenten) über einen Socket in der Internet-Domain miteinander kommunizieren.

AF_AX25 Diese Domain benützt Amateur Radio X25 als Übertragungsprotokoll.

AF_IPX Diese Domain verwendet Novell IPX zur Nachrichtenübertragung.

AF_APPLETALK Die Übertragung in dieser Domain passiert unter Verwendung von APPLETALK.

AF_X25 Das X25 Protokoll wird in dieser Domain verwendet.

AF_INET6 Diese Domain unterstützt IPv6.

Der *Typ* spezifiziert, welche Eigenschaften die Kommunikation über den Socket aufweisen soll:

SOCK_STREAM (Stream Socket) gewährleistet eine zuverlässige Kommunikation. Es gehen weder Daten verloren, noch werden welche dupliziert oder vertauscht. Daten werden als kontinuierlicher Strom übertragen, Grenzen zwischen Paketen werden nicht mitübertragen.

Die Übertragung von '*out-of-band data*' wird unterstützt (Das sind Daten, die außerhalb des normalen Datenstroms übertragen und empfangen werden können).

Bis auf die 'out-of-band' Daten entspricht dieser Kommunikationstyp der Funktionalität von Pipes.

SOCK_DGRAM (*Datagram Socket*) stellt einen unzuverlässigen Mechanismus zur Übertragung einzelner Datenpakete dar. Im Gegensatz zum Stream Socket wird keine virtuelle Verbindung zwischen Client- und Serverprozess aufgebaut.

SOCK_RAW (*Raw Socket*) ist ein Mechanismus zum direkten Zugriff auf ein unterlagertes Kommunikationsprotokoll (während z.B. in der Internet Domain ein Stream Socket auf TCP/IP[24] und ein Datagram Socket auf UDP/IP[25] aufsetzt, setzt der Raw Socket direkt auf IP, dem Internet Protokoll, auf).

[24] *Transmission Control Protocol* on *Internet Protocol*
[25] *User Datagram Protocol* on Internet Protocol

SOCK_RDM (*Reliable Delivered Message Socket*) ist dem Datagram Socket sehr ähnlich. Allerdings wird bei diesem Typ von Socket garantiert, dass die Daten beim Empfänger ankommen.

SOCK_SEQPACKET (*Sequenced Packet Socket*) ist dem Stream Socket ähnlich, unterscheidet sich aber dadurch, dass keine 'out-of-band' Daten unterstützt werden, und dass die Grenzen zwischen Paketen erhalten bleiben.

SOCK_PACKET (*Packet Socket*) ist kein standard BSD Socket Typ, sondern eine Linux-spezifische Erweiterung, die den direkten Zugriff auf Pakete auf Deviceebene ermöglicht.

Das *Protokoll* gibt an, welches Protokoll für die Kommunikation zu verwenden ist. Wird dieser Parameter nicht spezifiziert (= 0), so wird, abhängig von der gewählten Domain, ein entsprechendes Protokoll ausgewählt.

Soll eine Verbindung zu einem Socket aufgebaut werden können, so muss ihm eine Adresse zugewiesen werden können (Dies ist normalerweise nur bei Serverprozessen nötig; Clientprozesse bekommen im allgemeinen eine Adresse vom System zugewiesen. Ihre Adresse muss nicht von vornherein bekannt sein, da sie ja von sich aus eine Verbindung zu einem Server aufbauen). Abhängig von der Domain des Sockets, haben die Adressen vollkommen verschiedene Formate.

In der UNIX Domain ist eine Adresse z.B. ein Eintrag im Filesystem (Socket Special File), in der Internet Domain setzt sich eine Socketadresse aus der 32-Bit Internetadresse[26] des Computers und einer 16-Bit Portnummer zusammen. Dies wird durch den `bind()` Systemcall bewerkstelligt.

```
int bind(
    int socket,
    struct sockaddr * addr,
    int addrlen);
```

Mit den Funktionen `inet_pton()` und `inet_ntop()` können Zeichenketten zu IPv4 oder IPv6 Internetadressen umgewandelt werden, respektive umgekehrt.

```
int inet_pton(
    int address_family,
    const char *string,
    void *adress_buffer);

const char *inet_ntop(
            int address_family,
            const void *address_buffer,
            char *string,
            size_t len);
```

Verlangt der gewählte Kommunikationstyp den Aufbau einer *virtuellen Verbindung* zwischen den beiden Sockets, so wird diese vom Clientprozess durch den `connect()` Systemcall angefordert.

[26] In IPv6 werden mittlerweile 128-Bit Adressen benutzt.

```
int connect(
    int socket,
    const struct sockaddr * address,
    size_t address_len);
```

Exekutiert der Serverprozess einen `accept()` Systemcall, so kann die virtuelle Verbindung zum anfordernden Clientprozess aufgebaut werden.

```
int accept(
    int socket,
    struct sockaddr * address,
    size_t * address_len);
```

Durch `accept()` wird ein neuer Socket kreiert, der als tatsächlicher Endpunkt für die aufgebaute Verbindung dient. Dadurch wird ermöglicht, dass ein Server mehrere virtuelle Verbindungen zu verschiedenen Clients gleichzeitig unterhält. Vor dem ersten Aufruf von `accept()` muss der Server noch `listen()` aufrufen.

```
int listen(
    int socket,
    int backlog);
```

Ab diesem Zeitpunkt ist der Socket bereit, ankommende Requests von Clientprozessen zu behandeln. Der Parameter `backlog` gibt an, wieviele unbehandelte Requests der Socket maximal verwalten können soll.

Ist eine virtuelle Verbindung zwischen zwei Sockets aufgebaut, so können die Prozesse anschließend über `read()` und `write()` miteinander kommunizieren. Wird jedoch spezielle Funktionalität gefordert (bearbeiten von *out-of-band* Daten, non-blocking IO, ...), oder wird ein Kommunikationstyp ohne virtuelle Verbindung verwendet, so müssen andere Systemcalls zum Senden und Empfangen von Nachrichten verwendet werden:

- `send()` und `recv()` verhalten sich wie `write()` und `read()`, mit der Ausnahme, dass über einen zusätzlichen Parameter Flags übergeben werden können (z.B. Lesen/Schreiben von 'out-of-band' Daten).
- `sendto()` und `recvfrom()` erlauben zusätzlich die Angabe (bzw. Abfrage) der Adresse des Kommunikationspartners. Dies ist bei Kommunikationstypen ohne virtuelle Verbindung notwendig.
- Mit `sendmsg()` und `recvmsg()` ist außerdem *Scatter/Gather IO*[27] und die Übertragung von Zugriffsrechten möglich.

[27] Die Nachricht kann aus mehreren, an verschiedenen Stellen im Speicher liegenden Teilen zusammengesetzt werden. Alle diese Teile können mit einem Systemcall gesendet werden (Atomizität des Sendens der Nachricht ist gewährleistet!).

- `getsockopt()` und `setsockopt()` können verwendet werden, um Socketeinstellungen auszulesen und zu verändern; so kann beispielsweise die Sende- (`SO_RCVBUF`) und Empfangspufferlänge (`SO_SNDBUF`) angepasst werden.

In den Abbildungen 2.29, 2.30 und 2.31 ist der Ablauf der Kommunikation über Sockets graphisch dargestellt (Die Parameter der Systemcalls wurden der besseren Übersicht halber vereinfacht dargestellt).

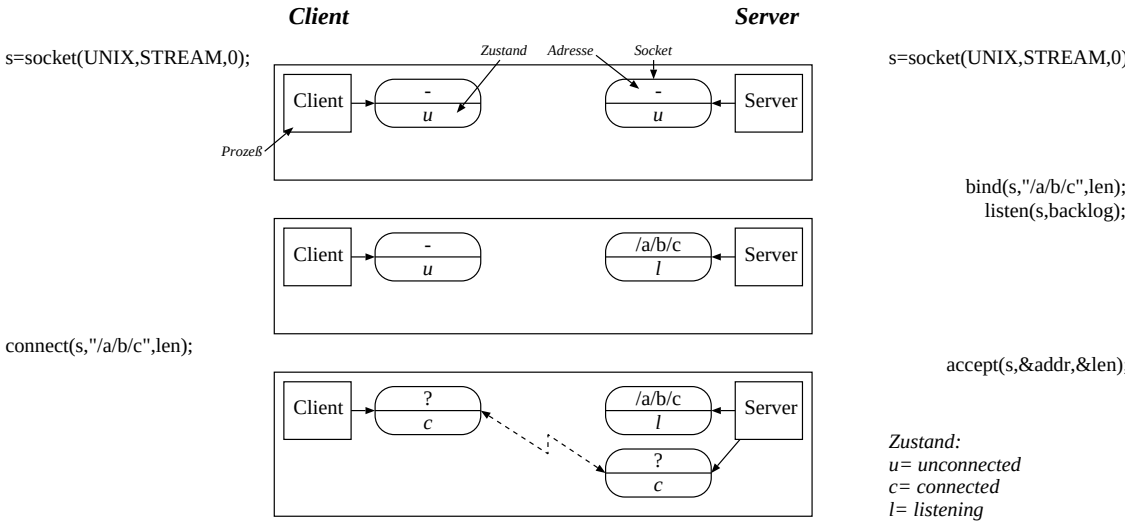


Abbildung 2.29: Kommunikation zweier Prozesse über Sockets in der UNIX-Domain

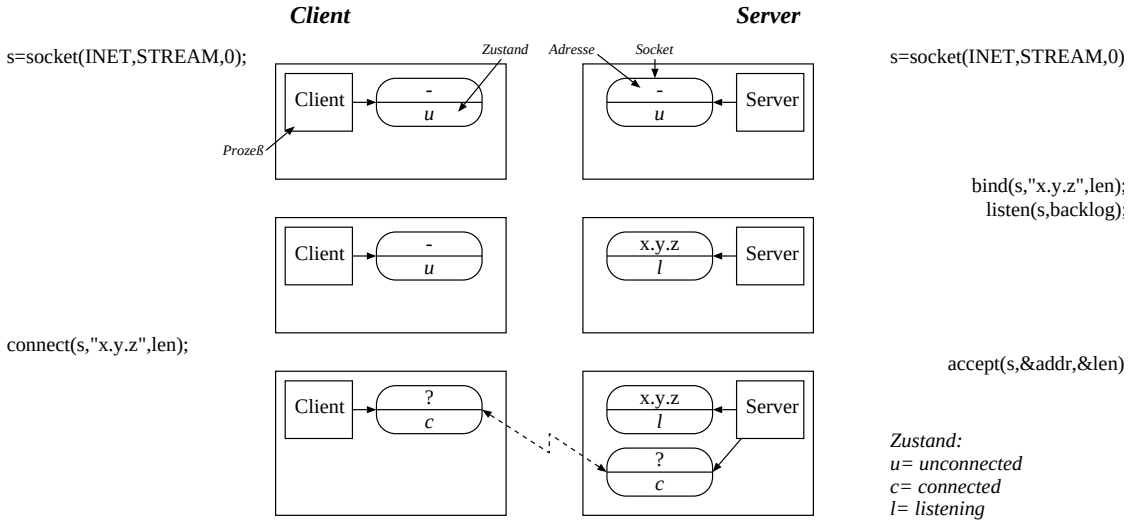


Abbildung 2.30: Kommunikation zweier Prozesse in der Internet Domain über Stream Sockets

Multicasting

Die bisher besprochene Socket-Funktionalität ermöglicht die bidirektionale Kommunikation zweier Prozesse. Multicasting erweitert dieses Konzept auf einen beliebig viele Empfänger, wobei der Sen-

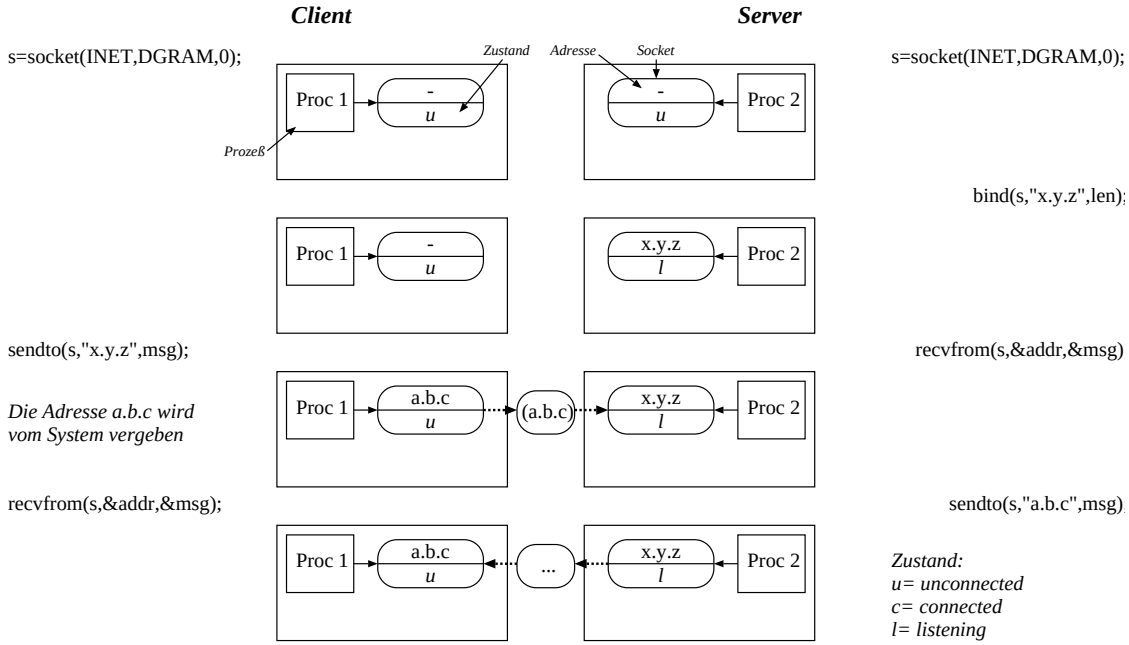


Abbildung 2.31: Kommunikation zweier Prozesse in der Internet Domain über Datagram Sockets

der auch Empfänger sein kann. IP Multicast ist ausschließlich Datagramm-orientiert; zuverlässige, Stream-orientierte Verbindungen werden nicht bereitgestellt.

Der Empfängermenge ist eine IP Multicast (Gruppen-) Adresse zugeordnet, an die ein Sender ohne zusätzliche Vorkehrungen Pakete senden kann, entsprechende Unterstützung des Betriebssystems und Netzwerks vorausgesetzt.

Ein Empfänger muss erst seine Mitgliedschaft zu der Multicast Gruppen-Adresse mit `setsockopt()` bekanntgeben, bevor er Pakete empfangen kann.

```
struct ip_mreq mreq;

/* Multicast Gruppen-Adresse setzen */ inet_pton(AF_INET,
"224.0.0.1", &mreq.imr_multiaddr); /* Interface setzen */
mreq.imr_interface = INADDR_ANY;

setsockopt(sock,IPPROTO_IP,IP_ADD_MEMBERSHIP,&mreq,sizeof(mreq));
```

Linux und die meisten modernen Unix-Derivate unterstützen Multicast sowohl auf Sender- als auch Empfängerseite.

Implementierung von Sockets

Bezogen auf das ISO Referenzmodell entsprechen Sockets dem Session Layer. Sie bauen direkt auf den Netzwerkprotokollen (TCP/IP, UDP/IP, . . .) auf, und stellen eine Schnittstelle für Applikationsprozesse bereit. Daten fließen von der Applikation durch die Sockets in die Netzwerkschicht, und umgekehrt (siehe Tabelle 2.3).

Für jeden Socket, der neu kreiert wird, wird eine Datenstruktur im Adressraum des Kernels angelegt. Diese Datenstruktur existiert solange, bis der Socket wieder gelöscht wird (`close()` Systemcall). Da für jeden in einem Prozess existierenden Socket ein File-Descriptor existiert, gibt es auch einen Eintrag in der File-Tabelle. Wie auch bei offenen Dateien zeigt ein Zeiger in der File-Tabelle auf einen Eintrag in der i-node Tabelle. Im Fall von Sockets enthält dieser i-node im Filesystem-spezifischen Teil eine Socket-Datenstruktur (`socket`). Diese enthält z.B. den Typ des Sockets, das verwendete Protokoll, den Zustand des Sockets (verbunden, nicht verbunden, im Verbindungsaufbau, im Verbindungsabbau, Puffer voll, ...), Information über aufgetretene Fehler, die Adresse des Sockets und einen Zeiger auf die empfangenen oder zu sendenden Daten.

ISO Referenzmodel	UNIX Model	Beispiele
Application Layer	Benutzerprogramme und Bibliotheken	telnet, ftp, http, rlogin, rcp,..
Presentation Layer		
Session Layer	Sockets	SOCK_STREAM
Transport Layer	Netzwerk Protokolle	TCP, UDP
Network Layer		IP
Data Link Layer	Netzwerk Interfaces	Ethernet Treiber
Physical Layer	Netzwerk Hardware	Ethernet Controller

Tabelle 2.3: Entsprechung von Sockets im ISO Referenzmodell

Da Länge und Struktur von Socket-Adressen, abhängig von der Domain, stark differieren, wird die Adresse nicht in der Socket-Struktur direkt abgespeichert. Stattdessen enthält sie einen Zeiger auf einen Puffer, indem die Adresse abgespeichert ist. Die Struktur von Socket-Adressen wird durch das erste Feld der Adresse bestimmt, das die Bezeichnung der Domain enthält. In Abbildung 2.32 sind zwei Beispiele von Adressen in unterschiedlichen Domains gegeben.

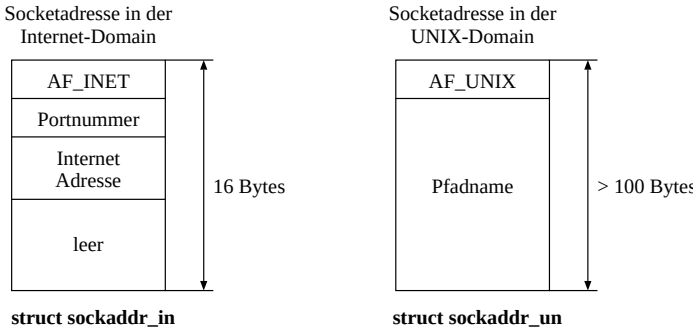


Abbildung 2.32: Beispiele für Adressen von Sockets

Jeder Socket enthält einen Zeiger auf eine Liste, in der die empfangenen bzw. zu sendenden Daten gespeichert sind. Diese Liste ist aus sogenannten *Socket Buffern* (`sk_buff`) aufgebaut. Ein `sk_buff` ist eine Struktur fixer Größe. Er enthält, neben Platz für Daten, die *Länge* der Daten (`len`), einen Zeiger auf den *Beginn des Datenbereiches* (`head`), einen Zeiger auf das *Ende des Datenbereiches* (`end`), einen Zeiger auf den *eigentlichen Beginn der Daten* (`data`), einen Zeiger auf das *eigentliche Ende der Daten* (`tail`) und *Zeiger auf den vorhergehenden* (`prev`) bzw. *nachfolgenden Socket Buffer* (`next`).

Socket Buffer sind auf zwei Arten verbunden: in Ketten und in Listen, diese Organisation ist notwendig, da bei einigen Sockettypen (Datagram, Sequenced Packet) die Grenzen von Datenpaketen bewahrt werden müssen. Daten, die in einer *Kette* von `sk_buffs` abgespeichert sind, werden als unstrukturierte Folge von Bytes behandelt. Bei einem Stream-Socket sind alle Daten in einer einzigen Kette gespeichert. Mehrere Ketten können zu *Listen* verbunden sein. Die Elemente einer Liste stellen die Datenpakete einer Kommunikation dar (siehe Abbildung 2.33).

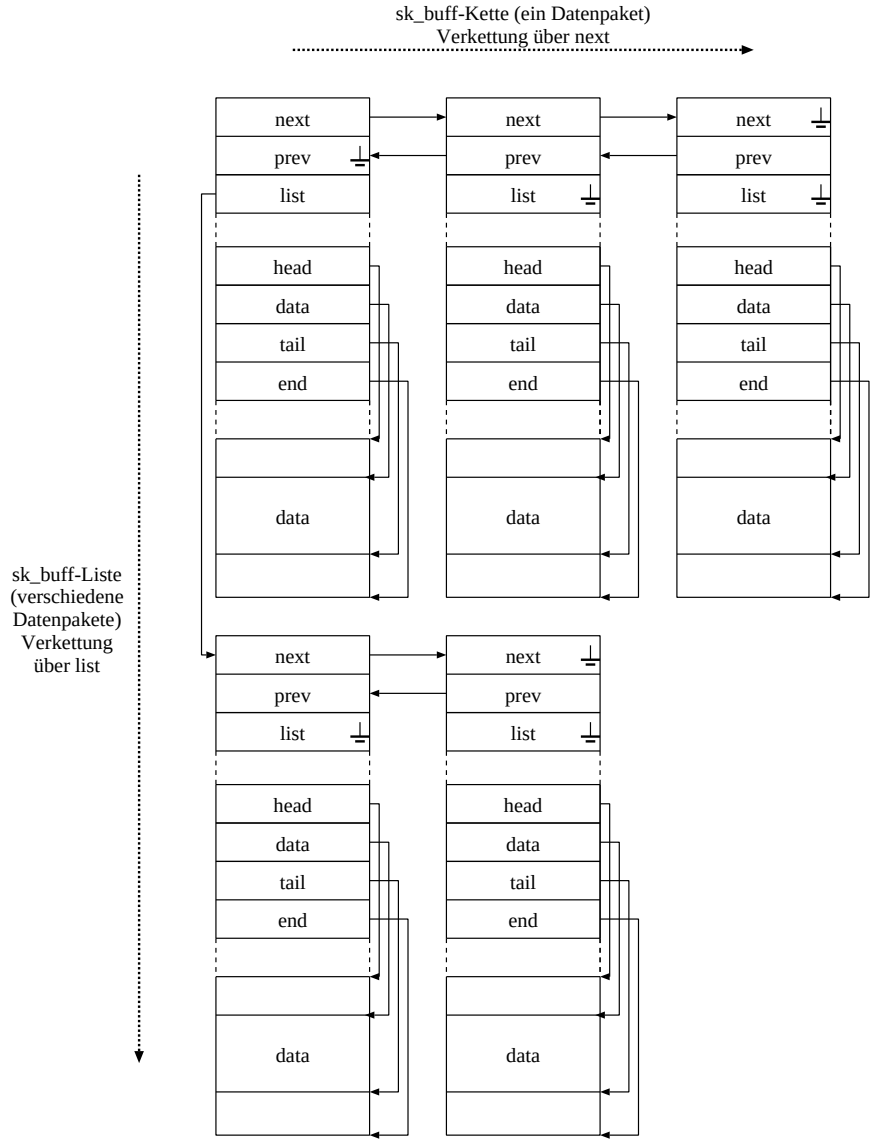


Abbildung 2.33: Organisation von Socket Buffern

Der Zugriff auf Daten durch die Angabe der vier Zeiger (`head`, `data`, `tail` und `end`) hat noch einen enormen Vorteil: Da beim Weitergeben der Daten durch die verschiedenen Protokollschichten normalerweise nur an bestehende Datenpakete vorne und/oder hinten protokollspezifische Daten angehängt bzw. weggeschnitten werden, ist es möglich, das Kopieren der Daten bei der Übergabe an eine andere Protokollschicht zu vermeiden. Es genügt, die vier Zeiger entsprechend anzupassen und eventuell noch die Protokolldaten hinzuzufügen (von den Protokollen in den höheren Schichten muss natürlich entsprechend viel Platz vor den eigentlichen Nutzdaten freigehalten werden).

Genau für diesen Zweck sind in Linux bereits Routinen zum Hinzufügen und Entfernen von Protokollheadern vorhanden:

`push()` bewegt den `data` Zeiger näher zum Beginn des Datenbereiches und schafft somit Platz für zusätzliche Protokollheader. Der `len` Eintrag wird entsprechend inkrementiert.

`pull()` bewegt den `data` Zeiger näher zum Ende des Datenbereiches und kann somit zum Entfernen von Protokollheadern verwendet werden. Der `len` Eintrag wird entsprechend dekrementiert.

`put()` bewegt den `tail` Zeiger näher zum Ende des Datenbereiches. Dies kann zum Hinzufügen von Protokollinformation am Ende der Daten verwendet werden. Der `len` Eintrag wird entsprechend inkrementiert.

`trim()` bewegt den `tail` Zeiger näher zum Beginn des Datenbereiches. Dies kann zum Entfernen von Protokollinformation am Ende der Daten verwendet werden. Der `len` Eintrag wird entsprechend dekrementiert.

Die Aufgabe der Kernelprozedur zum *Senden* von Daten über einen Socket besteht hauptsächlich darin, die Datenpakete an das beim Kreieren des Sockets angegebene unterlagerte Protokoll weiterzugeben. Außerdem muss überprüft werden, ob die Menge der zum Senden gepufferten Daten eines Sockets durch die neue Sendeanforderung über eine festgelegte Grenze hinaus ansteigen würde. In diesem Fall müssen die zu sendenden Daten in mehrere Teile aufgespalten werden (bei Protokollen, in denen keine Paketgrenzen beibehalten werden müssen), oder es muss solange gewartet werden, bis genügend Platz vorhanden ist, um das Senden durchführen zu können.

Die Kernelprozedur zum *Empfangen* von Daten von einem Socket überprüft zuerst, ob Daten auf dem Socket vorhanden sind. Ist dies der Fall, so wird die angeforderte Menge von Daten (sofern vorhanden) übergeben und aus dem Socket entfernt. Sollen Paketgrenzen beibehalten werden, so wird jedesmal genau ein Paket übergeben. Ist der bereitgestellte Puffer zu klein, so gehen die restlichen Daten des Paketes verloren.

2.7 Networking

2.7.1 TCP/IP

Networking unter UNIX baut auf den Internet-Protokollen auf, die unter Aufsicht der DARPA (Defense Advanced Research Projects Agency) des DoD (Department of Defense) entwickelt wurden. Die Protokolle sind in sogenannten RFCs (Request for Comments) spezifiziert. Die Internet-Protokolle erlauben die gesicherte Übertragung von Daten über Systeme, die aus zusammengeschalteten, paketorientierten Netzwerken bestehen. Das größte solche Netzwerk ist das *Internet*, das den Protokollen den Namen gegeben hat.

TCP (*Transmission Control Protocol*) ist ein verbindungsorientiertes, zuverlässiges Protokoll. Es baut auf einem (unter Umständen unzuverlässigen) Datagram-Service auf. Dieses Service wird

vom Protokoll IP (Internet Protocol) zur Verfügung gestellt. IP selbst baut auf den unterlagerten Netzwerkprotokollen (etwa Ethernet oder Token Ring) auf. Abbildung 2.34 zeigt die Internet-Protokollschichten und ihre Entsprechung im OSI Schichtenmodell.

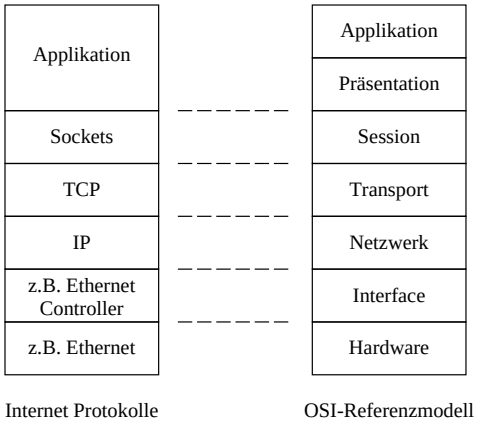


Abbildung 2.34: Schichtenmodell der Internet-Protokolle

IP

IP *Internet Protocol* stellt ein Service zur Verfügung, mit dem Pakete an Hosts geschickt werden können, ohne auf die physikalische Struktur der unterlagerten Netzwerke eingehen zu müssen. Die Struktur des Netzwerks und eventuelle Beschränkungen der verwendeten Teilnetze bleiben dem Benutzer von IP verborgen.

Die Hauptaufgaben von IP sind:

Routing Pakete müssen auf dem Weg zu ihrem Ziel über Netzwerke und *Gateways* geschickt werden.

Fragmentierung Wenn ein beteiligtes Netzwerk die gewünschte Paketgröße nicht unterstützt, müssen die zu übertragenden Pakete in kleinere Pakete zerlegt und auf der Empfängerseite wieder zusammengesetzt werden.

IP ermöglicht es somit, einzelne Pakete (Datagrams) von einem Rechner an einen anderen zu schicken. IP sorgt aber **nicht** für eine gesicherte Übertragung. Es gibt *keine Acknowledgements*, *keine Fehlerkontrolle* für die Daten, *keine Wiederholung* im Fehlerfall und *keine Flusskontrolle*.

Wie jedes Protokoll schickt auch IP mit jedem Paket Protokollinformation in einem Header mit. Abbildung 2.35 zeigt das Format eines IP-Headers.

Die wesentlichsten Felder des IP-Headers werden im Folgenden kurz erklärt. Eine vollständige Erklärung findet sich in [Pos81a]:

IHL ist die Länge des IP-Headers.

Version	IHL	Typ des Services	Gesamtlänge
Identifikation			Fragment-Flags und Offset
Lebenszeit		Protokoll	Prüfsumme für Header
Quelladresse			
Zieladresse			
Optionen			

Abbildung 2.35: Header eines IP-Pakets

Identifikation wird gemeinsam mit Fragment-Flags und Offset zum Fragmentieren und Zusammensetzen von IP-Paketen verwendet.

Lebenszeit Die Lebenszeit eines Pakets wird vom Sender des Pakets auf einen bestimmten Wert gesetzt. Jeder Rechner, der das Paket verarbeitet, vermindert den Wert dieses Felds um die Anzahl der Sekunden, die er für die Verarbeitung benötigt hat, mindestens jedoch um eins[28]. Das Paket wird zerstört, wenn seine Lebenszeit abgelaufen ist. Das hilft, eine Überlastung nach Netzwerkausfällen zu vermeiden.

Zieladresse Diese wird von Gateways zum Finden einer Route zum Zielrechner gebraucht, und vom Zielrechner selbst, um zu erkennen, dass er der Empfänger des Pakets ist.

IPv6

IPv6 ist der Nachfolger von IP(v4). Der Adressraum wurde von 32 Bit auf 128 Bit erweitert[29] und das Protokoll in Bezug auf andere Anforderungen, die sich im Lauf der Zeit herauskristallisiert haben und von IPv4 nicht gut genug erfüllt werden verbessert. So bietet IPv6 bessere Unterstützung u.a. in den Bereichen Sicherheit, Mobilität und Routing.

TCP

TCP (*Transmission Control Protocol*) [Pos81b] ist ein verbindungsorientiertes Protokoll, das die gesicherte Übertragung eines Datenstroms zwischen Rechnern implementiert. Es baut auf einem ungesicherten Datagram Service auf, wie es z.B. von IP angeboten wird. Zu diesem Zweck muss TCP folgende Teilprobleme lösen:

Datenübertragung TCP muss den Datenstrom in Pakete zerlegen und diese beim Empfänger wieder zu einem Datenstrom zusammensetzen.

Zuverlässigkeit TCP muss Fehler der unterlagerten Protokolle tolerieren (beschädigte, duplizierte, verlorene oder umgeordnete Pakete). Dazu wird ein *PAR-Protokoll*[30] mit Sequence-Nummern verwendet.

[28] in der Praxis wird die TTL von jedem Rechner um eins vermindert

[29] Durch das starke Wachstum des Internet und hohe Fragmentierung des zur Verfügung stehenden IPv4-Adressraums wurde dieser knapp

[30] Positive Acknowledgement or Retransmission

Flusskontrolle TCP schickt bei den Acknowledges Informationen darüber mit, wieviele Pakete der Empfänger noch akzeptieren kann.

Multiplexen TCP ermöglicht es mehreren Prozessen gleichzeitig auf das Netzwerk zuzugreifen. Dazu stellt es sogenannte *Ports* zur Verfügung. Ein solcher Port auf einem Rechner entspricht einem Socket.

Verbindungen TCP verwaltet den Aufbau und Abbau von Verbindungen zwischen Sockets.

Um das Routing und die Fragmentierung muss sich TCP nicht mehr kümmern, weil diese Aufgaben schon von IP behandelt werden. TCP ist wesentlich komplexer als IP (der RFC hat 85 Seiten, der für IP nur 45). Daher ist auch die Header-Information von TCP komplexer als die von IP. Abbildung 2.36 zeigt das Format eines TCP-Headers.

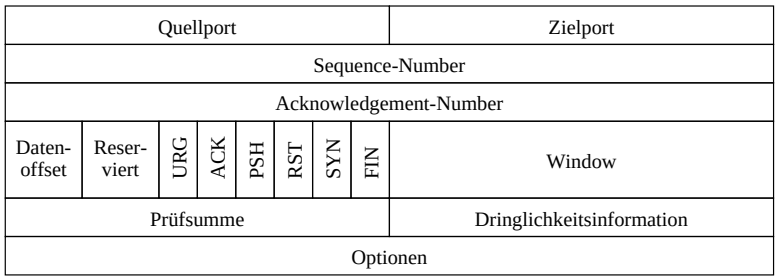


Abbildung 2.36: Header eines TCP-Pakets

Der TCP-Header enthält weder die Adresse des Quell- noch die des Zielrechners, weil diese ja schon im IP-Header gespeichert sind.

Sequence-Nummer Diese wird zum Erkennen von doppelt gesendeten Paketen verwendet.

Acknowledgement-Nummer Wird zur Implementierung des *Sliding-Window*-Protokolls verwendet (d.h., dass mehrere Pakete geschickt werden können, bevor ein Acknowledge vorliegen muss). Sie gibt die Sequence-Nummer des Pakets an, das als nächstes erwartet wird.

Window Die Anzahl von Bytes, die noch akzeptiert werden können. Dient zur Flusskontrolle.

Prüfsumme Dient zum Erkennen von Übertragungsfehlern im Header oder in den Nutzdaten.

Die Vorgänge beim Verbindungsaufbau und -abbau sind komplex, und deren Beschreibung liegt nicht im Rahmen der Vorlesung. Interessierte seien auf die entsprechenden RFCs verwiesen.

UDP

UDP (*User Datagram Protocol*) ist ein einfaches Datagram-Protokoll, das wie TCP auf IP aufbaut. Es ist ungesichert, verbindungslos und paketorientiert. Es erweitert das Service von IP nur in zwei Bereichen:

- Es stellt Ports mit Multiplexing und Demultiplexing zur Verfügung.
- Es stellt eine Prüfsumme für die Daten zur Verfügung.

2.7.2 Netzwerkdateisysteme

Die hohe Geschwindigkeit und Zuverlässigkeit von lokalen Netzwerken macht es attraktiv, über das Netzwerk auf Files zuzugreifen, die auf anderen Rechnern stehen. Damit kann Plattenplatz gespart werden, und eine Anzahl von Workstations kann als "homogenes" System verwendet werden. Es gibt unter UNIX im Wesentlichen zwei Implementierungen dieses Konzepts: NFS (*Network File System*), das von SUN Microsystems entwickelt wurde, und RFS (*Remote File Sharing*), das von AT&T entwickelt wurde. Beide erlauben es, Files auf anderen Rechnern so anzusprechen, als ob sie auf dem lokalen Rechner vorhanden wären. Beide bauen auf einem Client/Server Modell auf. Dabei werden die Rechner, die ihre Files anderen Rechnern zur Verfügung stellen, Server genannt, die Rechner, die diese Files in ihre lokale Directorystruktur einhängen, (*mount*) Clients. Um die Zugriffsrechte sinnvoll verwalten zu können, muss es eine Abbildung zwischen den Benutzern des Servers und des Clients geben. Die Unterschiede zwischen den beiden Systemen werden im Folgenden beschrieben.

NFS

In NFS erfolgt die Kommunikation zwischen Server und Client über ein *Remote Procedure Call* Protokoll (RPC), das auf UDP/IP aufbaut. Zusätzlich werden die Probleme mit unterschiedlicher Datenrepräsentation in den verschiedenen Rechnern mittels einer genormten externen Datenrepräsentation (XDR, *External Data Representation*) vermieden.

NFS verwendet das Konzept des *Stateless Server*. D.h., dass ein Server (möglichst) keine Informationen über den Zustand der Clients speichert. Zusätzlich wird versucht, die Operationen des Servers idempotent zu machen. Zusammen erleichtern diese Maßnahmen die Behandlung von Ausfällen: Wenn ein Server ausfällt, braucht der Client seine (idempotenten) Requests nur solange wiederholen, bis er eine positive Antwort erhält. Auch das Wiederaufsetzen nach dem Ausfall wird viel einfacher, weil der neue Server keine Zustandsinformation vom 'abgestürzten' Server übernehmen muss.

Das Konzept der Stateless Server bringt allerdings auch einige Probleme mit sich:

- *Remote Devices* und andere *Special Files* werden nicht unterstützt (weil `ioctl()` einen inneren Zustand erzeugt).
- Die Datenintegrität ist nicht sichergestellt.

NFS operiert auf der Filesystem-Ebene. D.h., dass ein Rechner nur ein ganzes Filesystem zur Verfügung stellen (*exportieren*) kann. Wenn innerhalb dieses Filesystems selbst ein Filesystem auf einem dritten Rechner gemounted ist, so kann dieses nicht angesprochen werden (keine *Multi-Hop*-Verbindungen). NFS geht davon aus, dass die Benutzer auf den verschiedenen Rechnern identisch sind (die einzige unterstütze Abbildungsfunktion ist die *identische Abbildung*). Die Namensgebung erfolgt über ein Host/Filesystem-Paar: Das Filesystem `/var` auf einem Rechner `ali` kann als `ali:/var` angesprochen werden. Filesysteme können zum Lesen und Schreiben, oder nur zum Lesen exportiert werden. Zur Steigerung der Effizienz verwalten die Clients einen Cache von Zugriffen.

NFS ist ein "Bestseller", es gibt Implementierungen für eine Vielzahl von Systemen, unter anderem auch für PCs.

RFS

RFS baut auf gesicherten Verbindungen (TCP/IP) auf. Die Server von RFS halten Zustandsinformation, daher funktioniert auch der Zugriff auf *Special Files*. Des Weiteren ist es möglich, mit *Multi-Hops* auf Daten zuzugreifen (A greift mit RFS auf eine Directoryhierarchie von B zu, die Files von C enthält).

Im Gegensatz zu NFS operiert RFS auf der Directory-Ebene, d.h., dass ein Server, unabhängig von der Aufteilung des Directorybaums in Filesysteme, beliebige Directorybäume zur Verfügung stellen kann.

Weiters werden *beliebige Abbildungen* zwischen den Benutzern des Servers und des Clients unterstützt. Die Namensgebung erfolgt über symbolische Namen, die über das DNS (Domain Name Service) verbreitet werden.

Die Konzepte von RFS sind sauberer als die von NFS, dennoch ist seine Verbreitung relativ beschränkt (im Wesentlichen auf System V UNIX).

SMB

SMB (Server Message Block) ist ein ursprünglich von IBM stammendes Netzwerkprotokoll, das vor allem in der Windows-Welt Verbreitung gefunden hat. Unter Unix wird SMB von der Samba-Suite[31] implementiert. Über SMB lassen sich u.a. Dateien und Drucker über das Netzwerk ansprechen. SMB kann auf TCP/IP, NetBEUI oder IPX aufbauen.

Verteilte Dateisysteme

Ein verteiltes Dateisystem ist ein Netzwerkdateisystem, das sich über mehrere Rechner erstreckt, dem Benutzer aber als ein einziges Dateisystem erscheint. Fortgeschrittenere Features von verteilten Dateisystemen sind u.a. Caching, asynchrone Operationen, Fehlertoleranz und ein „*disconnected mode*“, d.h. der Benutzer kann mit dem Dateisystem arbeiten, wenn er nicht mit dem Netzwerk verbunden ist, und die Daten werden synchronisiert, wenn wieder eine Netzwerkverbindung besteht. Beispiele für verteilte Dateisysteme sind u.a. Coda und AFS.

2.7.3 Packet Filter

Netzwerkprotokolle sind üblicherweise im Kernel implementiert. Sollen Protokolle auch von Applikationsprogrammen implementiert werden können, so muss der Kernel entsprechende Mechanismen zum direkten Zugriff auf die Netzwerkhardware bereitstellen.

Wird auf dieses Netz ausschließlich über dieses Applikationsprogramm zugegriffen, so genügt es, alle am Netz empfangenen Pakete an dieses Programm zu schicken. Gibt es aber mehrere Applikationsprogramme, die ihre Protokolle auf demselben Netz abwickeln, so müssen die empfangenen Pakete entsprechend gewisser Bedingungen an diese Programme weitergeleitet werden (das Senden von Paketen stellt kein Problem dar).

[31] `http://www.samba.org`

Der *Packet Filter* Mechanismus dient dazu, dem Kernel die Informationen zu geben, welche Pakete an welche Programme weitergeleitet werden sollen. Der ursprünglich an der Carnegie-Mellon Universität entwickelte *Enet Packet Filter* wurde auf BSD UNIX portiert und entwickelte sich danach in die *ULTRIX Packet Filter* bei DEC, in *STREAMS NIT* unter SunOS und in die *Berkeley Packet Filter* (BPF) unter BSD UNIX. Da in Linux ebenfalls Berkeley Packet Filter Verwendung finden wird nun genauer auf diese Art der Packet Filter eingegangen.

Berkeley Packet Filter sind in Linux kein Teil des Kernels, sondern werden als Bibliotheksfunktionen der sogenannten Packet Capture Library (`libpcap`)[32] mit der eigentlichen Applikation gelinkt und laufen auf User Level ab. Es gibt allerdings auch einen Packet Filter unter Linux, der im Kernel abläuft – `netfilter`. Pakete müssen erst diesen passieren, bevor sie an Applikationen und deren Packet Filter (sofern vorhanden) weitergereicht werden. Die Regeln in den Kerneltabellen, die mit dem zu `netfilter` gehörenden Kommandozeilenprogramm `iptables` manipuliert werden, gelten global für alle Applikationen.

Die Benutzung von Berkeley Packet Filtern geschieht allerdings auf folgende Art:

1. Ein Netzwerkdevice (etwa `eth0`) wird der Funktion `pcap_open_live()` geöffnet.
2. Nachdem man einen gültigen `pcap_t` Zeiger für das Device erhalten hat, kann man das Filterprogramm, das auf dieses Device angewendet werden soll, setzen. Ein Filterprogramm ist in einer eigenen, registerorientierten Programmiersprache (die mit Hilfe von C Makros ausgedrückt wird) geschrieben, die es erlaubt, beliebige Felder eines ankommenden Paketes auf bestimmte Bedingungen zu prüfen (siehe Listing 2.1).
3. Der Empfang eines Paketes kann vom Applikationsprogramm auf verschiedene Arten festgestellt werden:
   - Das Programm ruft die Funktion `pcap_next()` auf. Diese kehrt zurück (und liefert die Anzahl der Bytes zurück, die vom Filterprogramm akzeptiert wurden), sobald ein Paket empfangen wurde.
   - Mit `pcap_disatch()` und `pcap_loop()` kann man eine Funktion aufrufen lassen (bei `pcap_loop` wiederholt) wenn ein Paket empfangen wurde.
4. Da die `libpcap` nur den Empfang behandelt, muss zum Senden wie schon beschrieben ein Socket geöffnet und dann `write()` verwendet werden.

Listing 2.1 zeigt ein Paketfilterprogramm mit den dazu gehörigen `pcap_`- Funktionsaufrufen. Das Filterprogramm akzeptiert nur ICMP (Internet Control Message Protocol, u.a. von `ping` verwendet) Pakete. Falls ein Paket akzeptiert wird, liefert der Filter `PACKET_DATA_SIZE` als Länge der weiterzureichenden Daten zurück. Ansonsten wird null zurückgeliefert. Die Logik wird in Abbildung 2.37 verdeutlicht.

Die erste Instruktion des Filterprogramms lädt (`BPF_LD`) das Halbwort (`BPF_H`), das am absoluten (`BPF_ABS`) Byteoffset zwölf im Paket steht, in den Akkumulator der Filtermaschine. Die zweite Instruktion überspringt (`BPF_JMP`) für den Fall, dass der Wert im Akkumulator gleich (`BPF_JEQ`) der

[32] Diese Bibliothek ist ein Teil der `tcpdump` Distribution.

Konstanten (`BPF_K`) `ETHERTYPE_IP` ist, die nächsten null, ansonsten die nächsten drei Instruktionen. Die dritte Instruktion lädt (`BPF_LD`) das Byte (`BPF_H`), das am absoluten (`BPF_ABS`) Byteoffset 23 im Paket steht, in den Akkumulator. Die vierte Instruktion überspringt (`BPF_JMP`) für den Fall, dass der Wert im Akkumulator gleich (`BPF_JEQ`) der Konstanten (`BPF_K`) `IPPROTO_ICMP` ist, die nächsten null Instruktionen. Für den Fall, dass der Wert ungleich der Konstanten ist, wird eine Instruktion übersprungen. Die fünfte Instruktion retourniert (`BPF_RET`) die Größe des Paketes (Header + ARP Daten) als Konstante (`BPF_K`) und beendet somit das Filterprogramm (das Paket wurde akzeptiert). Die sechste und letzte Instruktion retourniert (`BPF_RET`) die Konstante (`BPF_K`) 0, falls das Paket nicht akzeptiert wurde.

Die von UNIX standardmäßig unterstützten Netzwerkprotokolle (TCP/IP, UDP/IP, ...) sind direkt im Kernel implementiert. Es wäre zwar möglich, auch diese über Packet Filter abzuwickeln, da diese Protokolle aber sehr oft benutzt werden, wurde dies aus Performancegründen nicht realisiert.

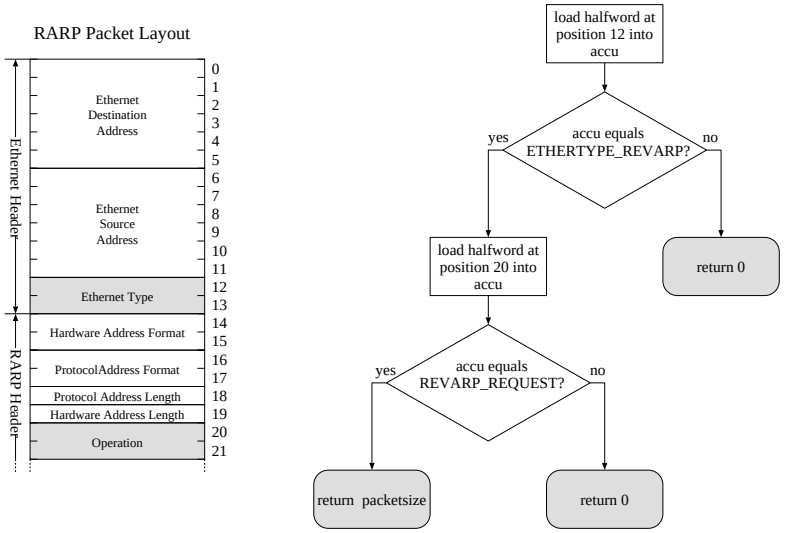


Abbildung 2.37: Beispiel eines Filterprogrammes

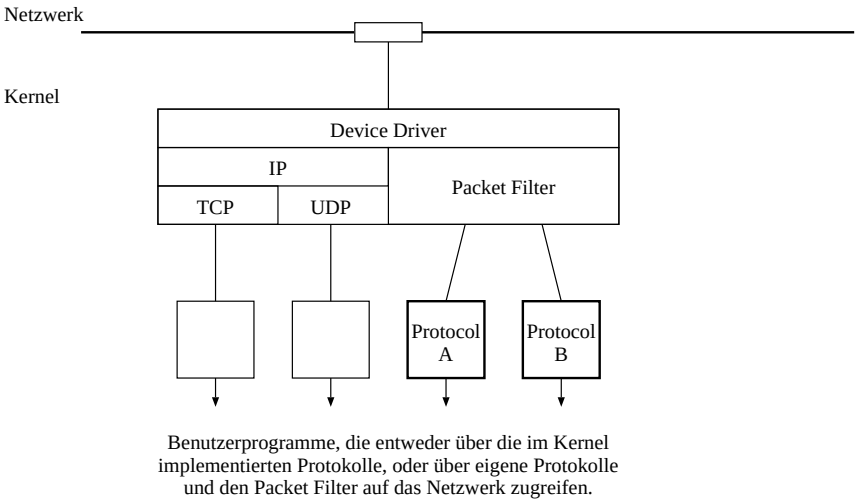


Abbildung 2.38: Einbindung von Packet Filtern in den UNIX Kernel

Der Durchsatz eines Packet Filters ist zwar, verglichen mit einem Szenario, indem das Verteilen der Pakete von einem Benutzerprozess durchgeführt wird, sehr hoch, er reicht aber doch nicht an den Durchsatz der im Kernel implementierten Protokolle heran. Auch kann ein Benutzerprozess kein "*Service*", wie es zum Beispiel bei der Anwendung von Sockets gebraucht wird, bereitstellen.

Da sowohl die im Kernel implementierten Standardprotokolle als auch Packet Filter auf die Netzwerktreiber zugreifen müssen, müssen diese Mechanismen parallel im Kernel existieren. Jene Pakete, die von den Standardprotokollen verarbeitet werden, werden bereits von diesen "konsumiert" und gelangen nicht zum Packet Filter.

Abbildung 2.38 zeigt die Einbindung des Packet Filter Mechanismus in den Kernel.

Filterprogramm zum erkennen von ICMP Paketen

Listing 2.1: Ein ICMP Packet Filter

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stropts.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>

#include <net/if.h>
#include <netinet/if_ether.h>

#include <pcap.h>

#define PACKET_DATA_SIZE (sizeof(struct bpf_hdr) \
                          + sizeof(struct ether_header) + 128)
#define FILTER_PROGRAM_LENGTH 6
#define SNAPLEN 512
#define TMOUT   100
#define IPPROTO_ICMP 0x01

/*
 * filter program
 */
static struct bpf_insn sFilterInstructions[FILTER_PROGRAM_LENGTH] =
{
    BPF_STMT(BPF_LD + BPF_H + BPF_ABS, 12),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, ETHERTYPE_IP, 0, 3),
    BPF_STMT(BPF_LD + BPF_B + BPF_ABS, 23),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, IPPROTO_ICMP, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, PACKET_DATA_SIZE),
    BPF_STMT(BPF_RET + BPF_K, 0)
};
```

```
static struct bpf_program sFilterProgram =
{
    FILTER_PROGRAM_LENGTH,
    sFilterInstructions
};

static pcap_t *pCapture = NULL;
const char * pcCmnd = "<not set>";

static void terminate(int nExitStatus)
{
    if (pCapture != NULL)
        pcap_close(pCapture);

    exit(nExitStatus);
}

static void handle_signal(int nSignal)
{
    if (printf("%s: Caught signal %d! - Cleaning up ...\n",
               pcCmnd, nSignal) < 0)
    {
        (void) fprintf(stderr, "%s: Cannot write to stdout - %s!\n", pcCmnd,
                       strerror(errno));
        terminate(EXIT_FAILURE);
    }

    if (fflush(stdout) == EOF)
    {
        (void) fprintf(stderr, "%s: Cannot flush stdout - %s!\n",
                       pcCmnd, strerror(errno));
        terminate(EXIT_FAILURE);
    }

    terminate(EXIT_SUCCESS);
}

static void dump_packet(u_char *user_data,
                        const struct pcap_pkthdr *pkt_hdr,
                        const u_char *pkt_data)
{
    printf("detected ICMP packet\n");
}

int main(int argc, char ** argv)
{
    u_int nDataLinkType = DLT_NULL;
```

```
char pcErrBuf[PCAP_ERRBUF_SIZE];
pcCmnd = argv[0];

/* install signal handlers */
(void) signal(SIGINT, handle_signal);
(void) signal(SIGQUIT, handle_signal);
(void) signal(SIGTERM, handle_signal);

if (argc != 2)
{
    (void) fprintf(stderr, "Usage: %s <interface>!\n", argv[0]);
    terminate(EXIT_FAILURE);
}

/* open packet filter */
if ((pCapture = pcap_open_live(argv[1], SNAPLEN, 0,
                               TMOUT, pcErrBuf)) == NULL)
{
    (void) fprintf(stderr, "%s: Cannot open packet filter - %s!\n",
                   argv[0], pcErrBuf);
    terminate(EXIT_FAILURE);
}

/* get and print out data link layer type */
nDataLinkType = pcap_datalink(pCapture);

(void) printf("Connected to ");
switch (nDataLinkType)
{
    case DLT_EN10MB:
        (void) printf("Ethernet");
        break;
    default:
        (void) printf("unknown data link layer type");
        break;
}
(void) printf("\n");


/* install the filter program */
if (pcap_setfilter(pCapture, &sFilterProgram) == -1)
{
    (void) fprintf(stderr,
                   "%s: Cannot install the filter program - %s!\n",
                   argv[0],
                   pcap_geterr(pCapture));
    terminate(EXIT_FAILURE);
```

```
    }

    /* loop forever (well, until an error or signal occurs, actually) */
    if (pcap_loop(pCapture, -1, dump_packet, NULL) < 0)
    {
        (void) fprintf(stderr, "%s: Error reading packets - %s!\n",
                       argv[0], pcap_geterr(pCapture));
        terminate(EXIT_FAILURE);
    }

    terminate(EXIT_SUCCESS);
}
```

Kapitel 3

Die Programmiersprache C

C gehört (wie z.B. Modula, Pascal und Ada) zu den Algol-ähnlichen Programmiersprachen. Es enthält alle Merkmale höherer Programmiersprachen (Komplexe Datentypen, Schleifen, Funktionen, Modularisierbarkeit), die es ermöglichen, lesbare und portable Programme zu schreiben. Es enthält aber auch genügend "low level"-Elemente, um in vielen Fällen Assemblersprachen zu ersetzen. Dementsprechend weit gefächert ist der Einsatzbereich von C, der von Betriebssystemen (UNIX wurde fast zur Gänze in C geschrieben), über Systemsoftware (Compiler, Editoren) zu Anwenderprogrammen aller Art reicht. Ebenso weit wie die Anwendungsbereiche der in C geschriebenen Software ist auch die Hardware gestreut, für die C-Compiler geschrieben wurden. Es gibt C-Compiler für 8-bit Microcontroller ebenso wie für Supercomputer. Am weitesten verbreitet ist C aber im Workstation- und PC-Bereich.

C wurde nicht wie manche andere Sprachen von einer Person oder einem Komitee fix und fertig entworfen, sondern ist im Laufe der Zeit gewachsen. Die erste Version wurde Anfang der 70er-Jahre von Dennis Ritchie für die Implementierung von UNIX auf der PDP-11 entwickelt. Diese Version war als "maschinenunabhängiger Assembler" gedacht und ließ dem Programmierer entsprechend viele Freiheiten. Später wurde das Typkonzept der Sprache etwas ernster genommen, sowie einige Details der Syntax geändert. Diese Sprache wurde dann in [Ker78] vorgestellt. Verschiedene Implementierungen hielten sich jedoch nicht exakt an diese Sprachdefinition, sondern erweiterten die Sprache oder legten missverständliche Stellen unterschiedlich aus. Vor allem aber entstanden umfangreiche Bibliotheken, d.h., Sammlungen von Funktionen. Um 1983 bildete daher das American National Standards Institute (ANSI) eine Arbeitsgruppe, um C zu standardisieren. Diese Arbeitsgruppe konzentrierte sich darauf, die existierende Sprache und die Bibliotheken zu standardisieren, führte aber auch ein paar neue Konzepte ein. 1989 wurde die Arbeit an diesem C-Standard abgeschlossen. Wir werden in diesem Kapitel ANSI-C behandeln.

Dieser Abschnitt des Skriptums soll einerseits dem mit anderen höheren Programmiersprachen (insbesondere Modula-2) vertrauten Leser einen raschen Einstieg in C ermöglichen, andererseits dem C-Programmierer als kurzes Referenzhandbuch dienen. Um diese beiden entgegengesetzten Ziele unter einen Hut zu bringen, stellt das nächste Kapitel anhand eines konkreten Beispiels die grundlegenden Konzepte von C vor. Daran schließt eine Beschreibung der Sprache und der Standard-Bibliothek an. Beide sind insofern vollständig, als sie alle Konstrukte und Funktionen erwähnen, aber nur die gebräuchlicheren im Detail behandeln.

Diese Einführung in C ist notwendigerweise zu kurz, um alle Aspekte der C-Programmierung zu behandeln. Als Einführung in C mit vielen Programmbeispielen sei die zweite Ausgabe von Kernighan und Ritchie [Ker88] empfohlen. Eine wesentlich detaillierte Beschreibung von C (und den Unterschieden zwischen verschiedenen Compilern) findet sich in [Har91]. Der ernsthafte C-Programmierer wird es hin und wieder nötig finden, die Sprachdefinition selbst [C89] zu Rate zu ziehen.

3.1 Ein C-Programm

In diesem Kapitel werden wir Schritt für Schritt ein C-Programm entwickeln. Diese Vorgehensweise ist für eine Einführung in eine Programmiersprache eher ungewöhnlich, sie hat aber gegenüber der herkömmlichen Methode mehrere Vorteile:

- Sie braucht wenig Platz. Damit muss jemand kein 200-Seiten-Buch lesen, bevor er sein erstes Programm schreiben kann. Wir gehen davon aus, dass der Leser bereits die prinzipielle Struktur höherer Programmiersprachen des Algol-Typs (Algol, Pascal, C, Modula, neuere Basic-Dialekte, ...) kennt und an der Frage “Wie mache ich das in C?” interessiert ist.
- Wir können auch Themen, die nicht eigentlich zur Programmiersprache gehören, wie Kommentare, Programmierstil und Fehlerbehandlung, sinnvoll behandeln. 30-Zeilen-Programme, die zur Demonstration eines Features einer Sprache geschrieben wurden, haben mit Programmen, die tatsächlich verwendbar sind, meist nicht viel zu tun. Kommentare im Code und saubere Strukturierung sind bei diesen Programmen aufgrund ihrer Kürze meist unnötig. Außerdem wird das Programm ohnehin im Text detailliert beschrieben. Ebenso fällt Fehlerbehandlung meist unter den Tisch, da sich diese Programme “auf das Wesentliche” beschränken.
- Die Reihenfolge, in der Konstrukte eingeführt werden, ergibt sich aus dem Programm.

Wir wollen aber nicht verschweigen, dass diese Methode auch Nachteile hat:

- Das Programm ist die meiste Zeit nicht lauffähig und tut erst ganz am Schluss das, was es tun soll.
- Die Reihenfolge, in der Konstrukte eingeführt werden, ist nicht unbedingt nach dem Schwierigkeitsgrad sortiert.

3.1.1 Problemstellung

Das Programm soll den Inhalt von Files lesen und ausgeben. Dabei sollen die Zeilen fortlaufend nummeriert werden, und am Schluss soll eine Liste aller Wörter sowie der Zeilen, in denen sie vorgekommen sind, ausgegeben werden. Die Namen der Files sollen dem Programm als Argumente übergeben werden. Den UNIX-Konventionen entsprechend soll von der Standard-Eingabe (normalerweise also vom Benutzerterminal) gelesen werden, wenn das Programm ohne Argumente oder mit dem Parameter “−” aufgerufen wird. Die Ausgabe soll auf die Standard-Ausgabe erfolgen.

Da wir den Zweck unseres Programms gleich im Programm festhalten wollen (für den Fall, dass wir das File in ein paar Jahren auf einer verstaubten Diskette wiederfinden und uns wundern, was "`xref.c`" denn eigentlich machen sollte), kommen wir zum ersten Element der Sprache C: dem Kommentar. Ein Kommentar beginnt mit der Zeichenfolge "`/*`" und setzt sich bis zur Zeichenfolge "`*/`" fort. Er kann beliebige Zeichen (auch Newlines) enthalten, und wird vom Compiler wie ein einzelnes Leerzeichen behandelt. Kommentare können in C nicht geschachtelt werden, d.h., die erste Zeichenfolge "`*/`" beendet den Kommentar, auch wenn zuvor beliebig viele "`/*`" eingefügt wurden.

Unser File `xref.c` besteht also bis jetzt aus dem Kommentar:

```
/*** program xref
 *
 * Author:      Peter Holzer   (hp@vmars.tuwien.ac.at)
 * Date:        1992-02-14
 * Purpose:     Print files with line numbers and generate a cross
 *              reference listing of all words in the files.
 * Usage:       xref [file ...]
 */
```

und weil wir uns über die grobe Struktur unseres Programms schon im Klaren sind, fügen wir noch zwei Kommentare hinzu:

```
/* read and print all files */
/* print cross reference listing */
```

Stil

Programmkommentare sollten auf jeden Fall den Autor des Programms, das Datum (und eventuell die Version), den Zweck des Programms und Hinweise zu seiner Verwendung enthalten. Es ist auch empfehlenswert, Programme auf Englisch zu kommentieren, da Englisch die Sprache der Informatik ist und Informatikerdeutsch aufgrund der vielen englischen Fachausdrücke ohnehin ein seltsames Kauderwelsch aus Englisch und Deutsch ist.

3.1.2 Statements und Funktionen

Bevor wir uns nun in medias res stürzen, müssen wir kurz die wesentlichen Elemente von C vorstellen und ein paar Begriffe erklären, die wir im Weiteren verwenden werden.

Konstanten sind konstante Werte, die im Programm vorkommen. Das können Zahlen sein wie `0`, `1`, `2`, `42`, `3.14`, `6.67E-11`, Zeichenkonstante wie `'a'` oder Strings wie `"Hallo"`.

Identifier bestehen aus Buchstaben, Ziffern und Underscores (_), wobei das erste Zeichen keine Ziffer sein darf: z.B. `a`, `printf`, `int`, `new_xref`, `NULL`, `_exit`, `XCopyArea`. Groß- und Kleinbuchstaben werden unterschieden. Namen, die mit Underscores beginnen, sind für das System reserviert und sollten *nicht* verwendet werden.

Operatoren sind Symbole für Operationen, die in der Sprache definiert sind, z.B. Addition (+), Subtraktion (−), Vergleich auf Gleichheit (==) und Zuweisung (=).

Ausdrücke (Expressions) bestehen aus Konstanten, Identifiern und Operatoren: z.B. `34`, `-1`, `a[i]`, `(p = malloc (100)) == NULL`

Anweisungen (Statements) bestehen im einfachsten Fall aus einem Ausdruck, gefolgt von einem Strichpunkt. Mehrere Statements können zu einem Block-Statement gemacht werden, indem sie in geschwungene Klammern eingeschlossen werden; außerdem gibt es komplexe Statements wie Schleifen und Verzweigungen. Beispiele:

```
a = 0;
printf ("hallo");
if (a > b) { a = a - b; } else { b = b - a; }
```

Funktionen Ein C-Programm besteht aus einer Anzahl von *Funktionen*, die den *Procedures* von Modula entsprechen. Die Definition einer Funktion besteht aus einem *Header*, der beschreibt, wie die Funktion aufgerufen wird, und einem *Body*, der beschreibt, was die Funktion tut.

3.1.3 Ausdrücke, Typen und Variablen

Jedes nicht-triviale C-Programm manipuliert während der Abarbeitung *Objekte*[1]. Jedes Objekt hat einen Typ (der während der gesamten Lebensdauer des Objekts gleichbleibt) und einen Wert (der sich ändern kann). Objekte, die über einen Namen ansprechbar sind, werden *Variablen* genannt. Variablen können hinsichtlich ihrer Sichtbarkeit (globale und lokale Variablen und Parameter), ihrer Lebensdauer (statische und automatische Variablen), und ihres Typs unterschieden werden.

C besitzt eine Vielzahl von Typen. Die grundlegenden Typen sind ganzzahlige Typen in verschiedenen Größen (`char`, `short`, `int` und `long`, jeweils mit und ohne Vorzeichen) und Fließkommazahlen in verschiedenen Größen (`float`, `double` und `long double`)[2]. Von diesen können weitere Typen abgeleitet werden: Mehrere Elemente desselben Typs können zu einem *Array* zusammengefasst werden und mehrere Elemente verschiedenen Typs zu einer *Structure* (die dem *Record* in Modula entspricht). Außerdem gibt es zu jedem Typ einen *Pointer*-Typ, der die Adresse eines Objekts dieses Typs aufnehmen kann. Auf diese Weise können beliebig komplexe Datentypen erzeugt werden.

Auch Ausdrücke haben einen Typ, der von den Operatoren und den Typen der Operanden abhängt. Typen können in andere umgewandelt werden. Dies kann explizit durch den Programmierer geschehen, oder der Compiler kann selbstständig Typumwandlungen einführen, wenn dies notwendig ist. Dafür kann es zwei Gründe geben:

- Ein Operator wird mit zwei Ausdrücken verschiedenen Typs verwendet (z.B. Addition von einem `int` und einem `double` oder Zuweisung von einem `double` an einen `long`). In diesem Fall wird einer der beiden Typen in den anderen umgewandelt, bevor die Operation ausgeführt wird.
- In einem Ausdruck wird der Wert eines Objekts verwendet, das einen Typ hat, der in Ausdrücken nicht vorkommen kann. So gibt es zum Beispiel keine Ausdrücke vom Typ `char`

[1] Trotz des Namens haben diese nichts mit objektorientierter Programmierung zu tun.

[2] Im Unterschied zu Modula gibt es in C *keinen* boolschen Typ. Vergleichs- und logische Operatoren liefern in C ein Ergebnis vom Typ `int`.

oder `short`. Werte von Objekten dieses Typs werden sofort in den Typ `int` umgewandelt. Auch Ausdrücke vom Array-Typ gibt es keine. Der des Arrays steht stattdessen für einen Pointer auf das Element Nr. 0.

Die Typ-Umwandlungen werden im Allgemeinen so ausgeführt, dass der Programmierer vom Ergebnis möglichst wenig überrascht wird. Die genauen Regeln sind in Kapitel 3.2.4 angegeben.

Als Beispiel wollen wir die Typumwandlungen betrachten, die bei der Auswertung des Ausdrucks `a[i] = 2.5 * b[i]` auftreten (Abb. 3.1). Wir nehmen an, dass `a` und `b` Arrays von `float` sind und `i` vom Typ `short` ist:

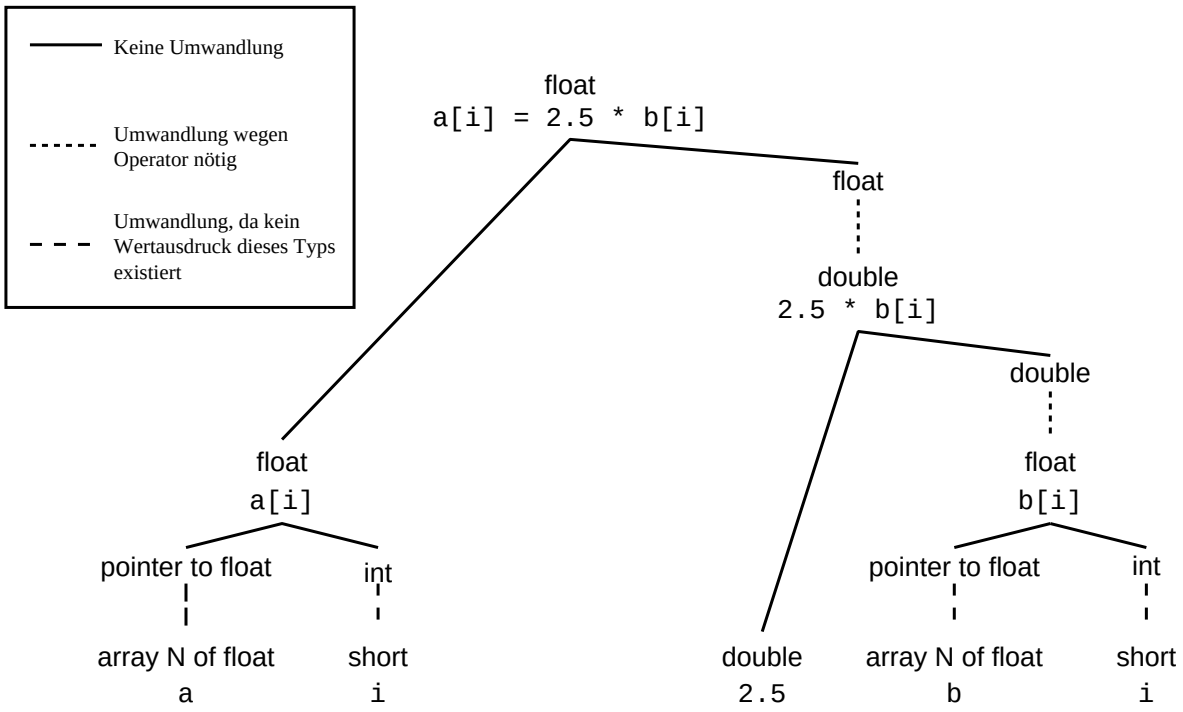


Abbildung 3.1: Typkonversionen bei Auswertung eines Ausdrucks

`a` ist ein Array von `float`. Stattdessen wird ein Pointer auf `a[0]` verwendet, also ein pointer to float. `i` ist ein `short` und wird laut ANSI Standard in ein `int` umgewandelt. Der Index-Operator braucht zwei Operanden, wobei einer vom Typ pointer to T, der andere ganzzahlig sein muss, und liefert dann ein Objekt vom Typ T, in unserem Fall also `float`. Der Wert `2.5` ist eine Fließkommazahl, die laut C-Standard als `double` interpretiert wird. `b[i]` wird analog zu `a[i]` ausgewertet. Da nun ein `double` (`2.5`) mit einem `float` (`b[i]`) multipliziert werden soll, wird zuvor `b[i]` ebenfalls in ein `double` umgewandelt. Als Ergebnis der Multiplikation erhält man ebenfalls ein `double`. Um dieses dem `float`-Objekt `a[i]` zuweisen zu können, muss es in `float` umgewandelt werden.

Natürlich kann ein guter Compiler viele dieser Umwandlungen vermeiden, das Ergebnis muss aber so sein, als ob er sie ausführen würde.

Liefern diese Typumwandlungen nicht das gewünschte Ergebnis, so kann der Programmierer mit einem sogenannten *Cast* eine explizite Typumwandlung erzwingen. Bei einem Cast wird der Name

des Typs, in den das Ergebnis eines Ausdrucks umgewandelt werden soll, in runden Klammern eingeschlossen vor den Ausdruck geschrieben.

Implizite Typkonvertierung kann manchmal zu unbeabsichtigten Effekten führen. Daher sollte im Zweifelsfall durch Casts selbst eine Typkonvertierung vorgenommen werden; dabei sollte man stets die Auswirkungen dieser Konvertierungen im Auge behalten.

Beispiele

Der Ausdruck `q = a / b` würde, wenn `a` und `b` vom Typ `int` und `q` vom Typ `double` sind, eine ganzzahlige Division ergeben und das Ergebnis in einen `double`-Wert umwandeln.

Soll bereits die Division in Fließkommaarithmetik durchgeführt werden, so muss mindestens einer der Operanden in `double` umgewandelt werden: `q = (double) a / b`.

3.1.4 Deklarationen und Definitionen

Wir haben nun viel von Objekten und Ausdrücken erzählt, haben auch ein paar Beispiele für ihre Verwendung gegeben, aber eines haben wir noch völlig verschwiegen: Wie sage ich dem Compiler, dass ich etwa eine Variable namens `i` vom Typ `int` haben will, und `a` ein Array mit 3 Elementen vom Typ `double` sein soll?

Im Unterschied zu Modula, wo eine Variablendefinition aus einer Liste von Variablen, gefolgt von ihrem Typ besteht, folgt in C auf einem Basistyp, eine Liste von Ausdrücken, die für jede Variable angeben, was man mit ihr machen kann, um auf den Basistyp zu kommen.

Einige Beispiele:

C-Deklaration	Bedeutung	Entsprechende Modula-Deklaration
int i;	i ist ein `int`	i: INTEGER;
unsigned int u;	u ist ein `unsigned int`	u: CARDINAL;
double a [10];	Jedes der 10 Elemente von a ist ein double	a: ARRAY [0..9] OF LONGREAL;
char *p;	Wenn p dereferenziert wird, erhält man einen char	p: POINTER TO CHAR;

Außerdem unterscheidet C zwischen Deklarationen und Definitionen. Deklarationen deklarieren, dass eine Funktion oder ein Objekt des angegebenen Typs existiert, Definitionen erzeugen das Objekt selbst. Jede Definition ist auch eine Deklaration, nicht aber umgekehrt. Bei Definitionen kann eine Variable auch *initialisiert* werden, indem der Wert der Variablen hinter einem Zuweisungsoperator (=) angegeben wird.

Stil

Die Tatsache, dass links nur der Basistyp steht, sollte man auch optisch hervorheben, indem man den Dereferenzierungsoperator * zur Variable und nicht zum Typ schreibt. Also nicht:

```
char*   p, c;
```

was den Leser des Programms zu der irrigen Ansicht verleiten könnte, sowohl `p` als auch `c` seien vom Typ pointer to char, sondern:

```
char    *p, c;
```

was deutlich die Tatsache hervorhebt, dass nur `p` ein Pointer ist.

Nachdem wir uns solcherart mit Theorie gewappnet haben, wollen wir den ersten Schritt in die Praxis tun:

3.1.5 Die Funktion `main`

Wie bei Variablen bestehen auch Funktionsdeklarationen aus einem Basistyp und einem Ausdruck, der beschreibt, wie aus der Funktion der Basistyp gewonnen werden kann. In runden Klammern neben dem Funktionsnamen werden (durch Beistriche getrennt) die Parameter definiert. Bei Funktions*definitionen* wird der abschließende Strichpunkt der Deklarationen durch eine Folge von Anweisungen in geschwungenen Klammern ersetzt.

Das "Hauptprogramm" eines C-Programms heißt konventionsgemäß `main`. Die Funktion `main` wird beim Programmstart vom System aufgerufen. `main` hat zwei Parameter: der erste enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde und der zweite die Argumente selbst. Diese Parameter werden üblicherweise mit den Namen `argc` und `argv` bezeichnet. Main liefert außerdem einen ganzzahligen Wert zurück, der vom Betriebssystem als Fehlercode verwendet werden kann.

```
int main (int argc, char **argv)
{
    /* read and print all files */
    /* print cross reference listing */
    return 0;
}
```

Achtung Diese Form der Funktionsdefinition, bei der die Typen der Parameter in den runden Klammern angegeben wurden, wurde vom ANSI-Komitee von C++ übernommen. Früher wurden nur die Namen der Parameter in die runden Klammern geschrieben und die Definitionen unmittelbar danach. Bei Funktionsdeklarationen wurden überhaupt keine Parameter angegeben. Um mit alten Compilern kompatibel zu sein, ist diese Syntax nach wie vor zulässig, allerdings werden dann Anzahl und Typ der Argumente nicht überprüft. Sie sollte daher nur verwendet werden, wenn Kompatibilität mit alten Compilern nötig ist.

Was den C-Neuling nun verblüfft, ist die Tatsache, dass ein pointer to pointer to char verwendet wird, um die Argumente an `main` zu übergeben. Da die Argumente Strings sind, und Strings Arrays von Zeichen, wäre ein zweidimensionales Array von Zeichen doch eher zu erwarten (bzw. ein Pointer auf Arrays von Characters, da C keine Ausdrücke (und daher auch keine Parameter) vom Array-Typ kennt). Zum Zeitpunkt der Kompilation des Programms ist allerdings weder die Anzahl noch die Länge der einzelnen Argumente bekannt. Um Länge und Anzahl der Argumente nicht unnötig einzuschränken, müsste das Array also sehr groß sein. Stattdessen wird folgende Methode gewählt: In ein Array aus Pointern to `char` werden Pointer auf die ersten Zeichen des Argument-Strings eingetragen. Ein Pointer auf das erste Element dieses Arrays wird dann an `main` übergeben (Abb. 3.2).

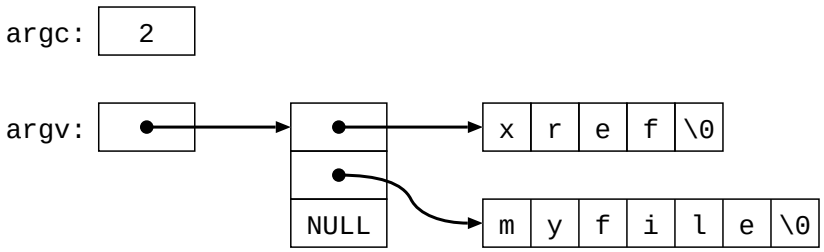


Abbildung 3.2: argv und argc beim Aufruf `xref myfile`

Achtung Obwohl Arrays als Parameter nicht erlaubt sind, können Parameter-Definitionen wie Arraydefinitionen aussehen. Der C-Compiler schreibt diese in die entsprechenden Pointerdefinitionen um. Es ist also möglich statt `char *s` auch `char s[]` oder sogar `char s[80]` zu schreiben. Wir empfehlen jedoch, dies nicht zu tun, da es kaum leserlicher ist (Ist jetzt `char *argv[]` ein Pointer auf ein Character Array oder ein Array von Character Pointern?), aber beim (menschlichen) Leser ungerechtfertigte Erwartungen auslösen kann (insbesondere, wenn eine Längenangabe in eckigen Klammern geschrieben wird, die der Compiler dann einfach ignoriert). *If you lie to the compiler it will get its revenge!*

3.1.6 Statements, Schleifen und Verzweigungen

Der Body einer Funktion besteht aus einer Folge von Statements. Unsere Funktion enthält bereits ein Statement, ein `return`-Statement. Es weist den Computer an, diese Funktion zu verlassen und den angegebenen Wert an die aufrufende Funktion zu liefern. Im Unterschied zu Modula werden in C Statements nicht durch einen Strichpunkt getrennt, sondern durch ihn abgeschlossen, d.h., auch am Ende des letzten Statements einer Sequenz gehört ein Strichpunkt.

Wir werden nun schön langsam unsere Kommentare durch Statements ersetzen, damit nicht nur menschliche Leser mit dem Programm etwas anfangen können, sondern auch der Compiler. Im ersten Schritt werden wir den Kommentar

```
/* read and print all files */
```

etwas verfeinern. Das Programm soll seine Argumente als Namen von Inputfiles verwenden, sofern Argumente übergeben wurden und sonst `stdin`. In C-Notation lautet das:

```
if (/* There are any arguments */)
{
    /* Take each argument as a file name,
     * and print the contents of the file
     */
}
else
{
    /* print what we get from standard input
     */
}
```

Das `if`-Statement in C hat also die Form:

`if (`*Bedingung*`)` *Statement*$_1$ [`else` *Statement*$_2$]

Wenn der Ausdruck *Bedingung* einen Wert ungleich 0 hat, wird *Statement*$_1$ ausgeführt, sonst *Statement*$_2$. Sollen in einem der Zweige mehrere Statements stehen, so müssen diese in geschwungene Klammern eingeschlossen werden.

Stil

Die Blockstruktur des Programms sollte auch optisch wahrnehmbar sein. Darum werden die Zweige eines `if`-Statements (und ebenso die Bodies von Schleifen und Funktionen) eingerückt. Wieweit eingerückt wird und wo die Klammern gesetzt werden, ist weitgehend Geschmacksache, aber es sollte deutlich (3–8 Zeichen) und konsistent eingerückt werden.

Es ist außerdem empfehlenswert, geschwungene Klammern auch dann zu verwenden, wenn der Zweig nur aus einem einzelnen Statement besteht. Beim Hinzufügen von Statements vergisst man nämlich zu leicht die Klammern, und dann entstehen Programmfragmente wie das Folgende, die absolut nicht das tun, was man von ihnen erwartet:

```
if (a)
    printf ("yes\n");
else
    printf ("no\n");
    printf ("Oh, no!!\n");
```

Wird der `then`-Zweig ausgeführt, so lautet die Programmausgabe:

```
yes
Oh, no!!
```

da das dritte `printf` nicht mehr zum `else`-Zweig gehört.

> Wenn an einem existierenden Programm Änderungen vorgenommen werden, so ist unbedingt der Stil des Programms beizubehalten. Es gibt nichts Unleserlicheres als Programme, an denen mehrere Programmierer mit unterschiedlichen Stilen gearbeitet haben.

Wie aus Abb. 3.2 hervorgeht, zählt der Programmname als Argument. Die Abfrage, ob "interessante" Argumente übergeben wurden, lautet also:

```
if (argc > 1)
```

Unser nächstes Problem ist es nun, an diese Argumente heranzukommen. Das Argument Nummer 0 (der Programmname) ist kein Problem, da `argv` ja darauf zeigt. Wie aber bekommen wir die restlichen Argumente? Die Antwort ist einfach. Da `argv` (= `argv + 0`) auf das Argument 0 zeigt, zeigt `argv + 1` auf das erste, `argv + 2` das zweite, und so weiter. Ganz allgemein können wir also mit `*(p + i)` das `i`-*te Element* in einem Array ansprechen, wenn `p` ein Pointer auf das Element Nr. 0 ist. Da diese Syntax eher hässlich ist, existiert als "Abkürzung" dazu der *Index-Operator* `[]`. `p[i]` hat die gleiche Bedeutung wie `*(p + i)`. Wir können also `argv[1]`, `argv[2]`, etc. für das erste, zweite, etc. Argument schreiben. Um alle Argumente ansprechen zu können, brauchen wir eine Schleife. Davon gibt es in C drei Arten: `do`-, `while`- und `for`-Schleifen.

Eine `while`-Schleife, die für jedes Argument `printfile` aufruft, sieht folgendermaßen aus:

```
i = 1;
while (i < argc)
{
    printfile (argv[i]);
    i ++;
}
```

Das ließe sich auch als `for`-Schleife schreiben:

```
for (i = 1; i < argc; i ++)
{
    printfile (argv[i]);
}
```

Da wir in unserem Fall schon wissen, dass mindestens ein Argument existiert, ist auch die `do`-Schleife äquivalent:

```
i = 1;
do
{
    printfile (argv[i]);
    i ++;
}
while (i < argc);
```

Wie die Syntax vermuten lässt, prüft die `do`-Schleife die Bedingung erst nach dem ersten Durchlauf, die `while`-Schleife dagegen davor. Die `for`-Schleife ist nichts anderes als eine andere Schreibweise für eine `while`-Schleife. Initialisierung, Schleifenbedingung, und Update können beliebige

Ausdrücke enthalten (oder sogar leer sein. In diesem Fall setzt der Compiler eine Konstante ungleich 0 ein), es gibt auch keine ausgezeichnete "Laufvariable", für die spezielle Regeln gelten wie in Modula.

Der ++-Operator ist eine von vier Möglichkeiten in C, eine Variable zu erhöhen. Dies kann (wie in anderen Programmiersprachen auch) durch `i = i + 1;` geschehen. Etwas Schreibarbeit spart man sich, indem man `i += 1;` schreibt. Noch etwas kürzer, aber völlig äquivalent ist `++ i;` oder `i ++;`. Steht der ++-Operator innerhalb eines Ausdrucks, so ist es wichtig, ob ++ vor oder nach der Variable geschrieben wird. Wenn das ++ nach der Variable geschrieben wird, wird der Ausdruck mit der noch unveränderten Variable ausgewertet und anschliessend die Variable um eins erhöht. Ein ++ vor der Variable bedeutet eine Auswertung des Ausdrucks mit dem bereits inkrementierten Wert der Variable. Im vorliegenden einfachen Fall sind alle diese Schreibweisen äquivalent, und es bleibt dem persönlichen Geschmack überlassen, welche gewählt wird. In komplizierteren Ausdrücken kann aber sehr wohl die eine oder andere Form vorzuziehen sein.

Da wir ja festgelegt haben, dass keine Argumente das gleiche bewirken sollen wie ein einzelnes Argument "−", rufen wir im `else`-Zweig einfach `printfile` mit der Stringkonstanten `"-"` auf.

Bevor wir nun darangehen, die Funktion `printfile` zu entwerfen, führen wir eine globale Variable `char *cmnd` ein (außerhalb von `main` zu definieren), der wir ganz am Anfang `argv[0]` zuweisen. Wir werden diese Variable noch in diversen Funktionen für Fehlermeldungen verwenden. Global wird diese Variable definiert, um sie nicht an jede Funktion übergeben zu müssen, die eine Fehlermeldung enthält. Nun aber zu `printfile`:

```
/*** function printfile
 *
 * Purpose:     Print file and line numbers for a file.
 *              Store line numbers of all words, so that a cross
 *              reference can be printed later.
 * In:          filename:       Name of the file to be printed.
 *                              - means stdin.
 */

void printfile (const char *filename)
{
    /* Open file if necessary and possible */
    /* For each line in the file:
     *     print it.
     *     for each word in the line:
     *         insert the word and the line number into the cross
     *         reference list.
     */
    /* close file */
}
```

Der Typ `void` (engl. Nichts) zeigt an, dass die Funktion keinen Wert zurückliefert (Sie entspricht also einer Pascal-Prozedur). Wir definieren `filename` als pointer to const char, um anzuzeigen, dass die Funktion diesen String nicht verändert.

Achtung Sollte versucht werden, eine als `const` deklarierte Variable zu verändern, so wird vom Compiler eventuell falscher Code erzeugt, der zum Absturz des Programms führen kann.

3.1.7 Ein- und Ausgabe

Es gibt in C (ebenso wie in Modula, aber im Gegensatz zu Pascal, BASIC oder FORTRAN) keine Ein- oder Ausgabestatements. Ein- und Ausgabe werden von *Library-Funktionen* erledigt. Alle diese Funktionen operieren auf sogenannten *Streams*. Ein Stream ist eine sequentielle Folge von Zeichen. Von einem Stream kann man zeichenweise lesen oder zeichenweise auf ihn schreiben. Unter bestimmten Umständen kann auch im Stream positioniert werden.

Jedem C-Programm stehen am Anfang drei Streams zur Verfügung: Die Standard-Eingabe (`stdin`), die Standard-Ausgabe (`stdout`) und die Standard-Fehlerausgabe (`stderr`). Weitere Streams können durch *Öffnen* von Files erzeugt werden.

Da die Anzahl der Streams pro Prozess begrenzt ist, sollte jedes File geschlossen werden, sobald es nicht mehr gebraucht wird.

Gehen wir also daran, den Stream mit dem File, dessen Name in `filename` übergeben wurde, zu verbinden. Zuerst deklarieren wir uns einen Stream:

```
    FILE        *fp;
```

Dieser Stream wird nun geöffnet, wobei überprüft wird, ob dabei ein Fehler aufgetreten ist. Im Fehlerfall wird eine Fehlermeldung ausgeben, und das Programm terminiert mit dem bereits definierten Wert `EXIT_FAILURE`:

```
if ((fp = fopen (filename, "r")) == NULL)
{
    (void) fprintf (stderr, "%s: cannot open %s: %s\n",
                    cmnd, filename, strerror (errno)   );
    exit (EXIT_FAILURE);
}
```

Die Funktion `fopen` *öffnet* ein File, d.h., sie erzeugt einen neuen Stream und verbindet ihn mit dem File. Falls sie das File nicht öffnen konnte, liefert sie den Wert `NULL` zurück und setzt die globale Variable `errno`, um den Grund für das Versagen anzuzeigen.

Achtung Im ANSI C-Standard ist nicht explizit festgehalten, das die Variable `errno` bei `fopen` im Fehlerfall gesetzt wird (siehe `errno.h` und Kapitel 3.4.15). In der bei den Übungen verwendeten Umgebung wird `errno` gesetzt und kann deshalb für eine aussagekräftige Fehlermeldung herangezogen werden.

`fprintf` ist die flexibelste der Ausgabefunktionen in C. Sie benötigt mindestens zwei Argumente (einen Stream und einen String) und kann noch beliebig viele weitere Argumente akzeptieren. Sie gibt den String auf den Stream aus, wobei das %-Zeichen eine Sonderbedeutung hat: Es leitet einen

sogannten *Format-Specifier* ein, der angibt, welchen Typ das nächste Argument hat und wie es auszugeben ist. Das genaue Format dieser Format-Specifier ist in Kapitel 3.4.9 beschrieben, hier wollen wir nur `%s` (String) und `%d` (`int`, der als Dezimalzahl auszugeben ist) erwähnen.

Stil

Die Methode, in einer Bedingung den Return-Wert einer Funktion einer Variablen zuzuweisen und gleich abzuprüfen, widerspricht zwar der generellen Faustregel, Seiteneffekte in Ausdrücken zu vermeiden, ist aber so weit verbreitet, dass sie durchaus akzeptabel ist.

Fehlermeldungen sind prinzipiell auf `stderr` zu schreiben.

Das Format der Fehlermeldungen:

Programmname : *Was hat nicht funktioniert* : *Warum*

ist das in Unix Übliche. Es ist auch recht gut für Programme geeignet, die nicht an ein Betriebssystem gebunden sind. Im Allgemeinen sollte man sich an bestehende Konventionen halten.

Ein Programm, das aufgrund eines Fehlers abbricht, sollte einen *positiven Exit-Status* haben.

Die Funktion `fprintf` liefert als Return-Wert die Anzahl der Zeichen die geschrieben wurden oder einen negativen Wert, wenn ein Fehler aufgetreten ist. In der Regel ist dieser Returnwert (wie der Returnwert jeder anderen Funktion auch) zu überprüfen, um festzustellen, ob die gewünschte Operation fehlerfrei ausgeführt wurde. Bei der Ausgabe einer Fehlermeldung kann der Returnwert von `fprintf()` jedoch ignoriert werden, indem man ihn auf `void` castet (Eine Fehlerbehandlung für einen Fehler beim Ausgeben der Fehlermeldung scheint nicht sinnvoll). Das Cast vor dem `fprintf` sagt dem Compiler: “Ja, ich habe darüber nachgedacht und ich bin mir sicher, dass ich den Return-Wert ignorieren will.”.

Nachdem wir das File geöffnet haben, lesen wir es zeilenweise, geben jede Zeile nummeriert auf `stdout` aus und schließen das File wieder:

```
while (fgets (line, sizeof (line), fp) != NULL)
{
    /* print the line number and the line */
    if (printf ("%6d  %s", lineno, line) < 0) {
            (void) fprintf (stderr, "%s: standard output error: %s\n",
                            cmnd, strerror (errno));
            exit (EXIT_FAILURE);
    }
    /*  for each word in the line
     *          insert the word and the line number into the cross
     *          reference list.
     */
    lineno ++;
}
```

```
if (ferror (fp))
{
    (void) fprintf (stderr, "%s: file read error: %s\n",
                    cmnd, strerror (errno));
    exit (EXIT_FAILURE);
}
(void) fclose (fp);
if (fflush (stdout) == EOF)
{
    (void) fprintf (stderr, "%s: standard output flush error: %s\n",
                    cmnd, strerror (errno));
    exit (EXIT_FAILURE);
}
```

`fgets` ist eine Library-Funktion, die Zeichen von einem Stream liest. Es werden Zeichen bis zum ersten Auftreten eines *Newline-Characters* gelesen; der zweite Parameter, prinzipiell verringert um eins, gibt an, wieviele Zeichen maximal gelesen werden, falls kein *Newline-Character* vorkommt. Die eingelesene Zeichenkette wird durch ein `\0` abgeschlossen. `printf` entspricht `fprintf`, gibt aber immer auf `stdout` aus. Da `fgets` sowohl `NULL` am Ende des Files, als auch im Fehlerfall liefert, muss mit `ferror` abgefragt werden, ob tatsächlich ein Fehler aufgetreten ist. Die Fehlerabfrage bei `fclose` kann bei Files, die nur zum Lesen geöffnet wurden, entfallen, da in diesem Fall nur Daten gelesen und nicht verändert werden. `fflush` sorgt dafür, dass die gepufferten Daten sofort auf `stdout` geschrieben werden.

3.1.8 Strings und Characters

Was wir bis jetzt noch ignoriert haben ist, dass der Filename "–" eine besondere Bedeutung haben soll. In diesem Fall wollen wir nicht ein File namens "–" öffnen, sondern `stdin` verwenden. Wir müssen also zwei Strings auf Gleichheit überprüfen. Den Gleichheitsoperator können wir dafür nicht verwenden, da dieser ja die Pointer auf die Strings vergleichen würde. Der Vergleich muss also zeichenweise erfolgen. Da diese Operation sehr oft benötigt wird, ist sie bereits in der Standard-Library (`<string.h>`) enthalten: Die Funktion `strcmp` nimmt zwei Strings als Argumente und liefert einen negativen Wert, wenn der erste String lexikographisch kleiner, einen positiven, wenn der erste String größer ist als der zweite, und 0, wenn beide gleich sind. Mit

```
if (strcmp (filename, "-") == 0)
{
    fp = stdin;
}
else
{
    if ((fp = fopen ( ... , "r") ...
```

können wir also den Filenamen "–" speziell behandeln. Ebenso wollen wir `fp` nur dann schließen, wenn wir es auch vorher geöffnet haben, was ja bei `stdin` nicht der Fall ist (zur Fehlerabfrage von `fclose` siehe letztes Beispiel Punkt 2.1.7):

```
if (fp != stdin) { (void) fclose (fp); }
```

3.1.9 Der Präprozessor

Die Sprache C hat ein Feature, das bei höheren Programmiersprachen eher selten zu finden ist, aber bei Assemblern weit verbreitet ist: Eine Makrosprache, mit der Textersetzung im Programm möglich ist. Diese Makrosprache ist im Gegensatz zum eigentlichen C zeilenorientiert, und alle Kommandos dieser Sprache beginnen mit einem "`#`" (*hash mark*). In vielen C-Compilern werden diese Textersetzungen von einem eigenen Programm, dem *Präprozessor* durchgeführt.

Der Präprozessor wird für 4 Zwecke verwendet:

1. Zur Definition von Konstanten.
2. Zur Definition von funktionsähnlichen Makros.
3. Zum Inkludieren von Files
4. Zum optionalen Kompilieren von Teilen des Programms.

Die ersten beiden Punkte werden durch die `#define`-Direktive implementiert. Diese hat entweder die Form

`#define` *Name Ersatztext*

(um ein objektähnliches Makro wie z.B. eine Konstante zu definieren) oder die Form

`#define` *Name* `(` *Parameter* `)` *Ersatztext*

um ein funktionsähnliches Makro zu definieren.

Im folgenden Text wird dann jedes Vorkommnis von *Name* (und den in Klammern folgenden Parametern bei Funktionsmakros) durch *Ersatztext* ersetzt. Zu beachten ist, dass dies ein reiner Textersatz ist, der sich nicht um die Syntax von C kümmert.

In unserem Programm können wir diese Methode dazu verwenden, um den Ausdruck `strcmp (filename, "-") == 0`, der nicht allzuviel darüber aussagt, ob die Strings gleich oder ungleich sein sollen (noch schlimmer in dieser Hinsicht ist der häufig verwendete Ausdruck `!strcmp(a,b)`, der eher impliziert, dass die Strings ungleich sein sollen) durch den leserlicheren Ausdruck `STREQ (filename, "-")` zu ersetzen. Wir definieren uns daher ein *Makro* `STREQ`[3], um unsere Absicht klarer zu machen.

```
#define STREQ(a,b)  (strcmp((a),(b)) == 0)
```

[3] STRing EQual.

Stil

Es ist üblich, alle Makros in Großbuchstaben zu schreiben, um sie deutlich von Variablen und Funktionen zu unterscheiden. Außerdem sollte man alle Argumente klammern, um Seiteneffekte – bedingt durch unterschiedliche Prioritäten von übergebenen Ausdrücken – zu vermeiden.

Außerdem haben wir diverse Typen, Funktionen und Variablen aus der Standard-Library verwendet. All diese Funktionen sind in sogenannten Header-Files[4] (die ungefähr den *Definition Modules* von Modula entsprechen) deklariert. Diese Header-Files müssen inkludiert werden, bevor die Funktionen verwendet werden:

```
#include <errno.h>  /* for errno */
#include <stdio.h>  /* for fopen, fclose, FILE, fprintf */
#include <stdlib.h> /* for exit */
#include <string.h> /* for strcmp */
```

Auch hier wird einfach die `#include`-Zeile durch den Inhalt des entsprechenden Files ersetzt. Um unangenehme Überraschungen zu vermeiden, sollten daher alle Header-Files am Anfang des C-Files vor allen eigenen Definitionen inkludiert werden.

Damit haben wir ein lauffähiges Programm, das die als Argumente übergebenen Files mit Zeilennummerierung auf `stdout` schreibt.

3.1.10 Noch einmal Strings

Nachdem wir diesen Erfolg genügend gefeiert haben, und das Ausgeben von Files mit Zeilennummern fad geworden ist, können wir uns dem zweiten (und etwas schwierigeren) Problem zuwenden, dem Erzeugen der Cross-References.

Dazu müssen wir jede eingelesene Zeile in Wörter zerlegen. Bevor wir das aber tun können, brauchen wir eine Definition für Wort. Für natürlichsprachige Texte ist wohl die Definition “eine Folge von Buchstaben” am besten geeignet, für C-Programme wäre “eine Folge von Buchstaben, Ziffern und Underscores, wobei das erste Zeichen keine Ziffer sein darf” geeigneter, und für andere Texte gelten wieder andere Regeln. Der Einfachheit halber nehmen wir ein Wort als Folge von Buchstaben an.

Außerdem müssen wir uns nun um die interne Darstellung von Strings kümmern. Bisher haben wir nur Strings von Library-Funktionen bekommen und an solche weitergegeben. Wenn wir den String aber in einzelne Wörter zerlegen wollen, so müssen wir zumindest das Ende erkennen können. Wie in Modula werden auch in C Strings mit einem Null-Character (`'\0'`) abgeschlossen.

Das nächste Problem besteht darin, zu erkennen, ob ein Zeichen ein Buchstabe ist oder nicht. Im Unterschied zu Modula ist C nicht auf einem bestimmten Zeichensatz definiert. Abfragen wie

```
if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
```

die auf ASCII-Systemen funktionieren, können auf anderen Systemen entweder gar nicht funktionieren (wenn z.B. ’Z’ kleiner ist als ’A’) oder in speziellen Fällen nicht funktionieren (Umlaute

[4] Weil sie nur die Header von Funktionen enthalten, aber nicht deren Bodies

liegen bei den meisten Zeichensätzen nicht zwischen A und Z, Der EBCDIC-Zeichensatz hat “Löcher” zwischen den Buchstaben, ...). Um diese Probleme zu umgehen, gibt es einige Funktionen zum Klassifizieren von Zeichen im Include-File `<ctype.h>`. Eine davon ist `isalpha`, die testet, ob das Argument ein Buchstabe ist.

```
/*
 * decompose the line into words and insert each word and the
 * line number into the cross reference list.
 */

inword = FALSE;
for (p = line; *p != '\0'; p ++)
{
    if (inword && ! isalpha (*p))
    {
        /*
         * we are just past the end of a word. terminate it and
         * insert it into list.
         */

        *p = '\0';
        new_xref (word, lineno);
        inword = FALSE;
    }
    else if (! inword && isalpha (*p))
    {
        /*
         * We are at the start of word here. Memorize the
         * position and the fact that we are now inside a word.
         */

        word = p;
        inword = TRUE;
    }
}

/*
 * If the line did end in a letter, we have not yet stored the
 * last word. Normally this cannot happen, since each line ends
 * in '\n', which is not a letter. The last line could lack the
 * trailing '\n', however, or we could just have split a very
 * long line. I think it is better to get some (possibly) bogus
 * words than to lose some, so I store what I've got.
 */

if (inword) new_xref (word, lineno);
```

3.1.11 Typedefs und Enums

In diesem Programmstück haben wir jetzt ganz offensichtlich eine boolsche Variable (`inword`) verwendet, obwohl wir doch vorher festgestellt haben, dass es diesen Typ in C nicht gibt. Es gibt nun mehrere Möglichkeiten, das in C zu erreichen. Man kann `inword` einfach als `int` definieren und zwei symbolische Konstanten `FALSE` und `TRUE` einführen:

```
#define FALSE 0
#define TRUE 1
```

Allerdings sagt die Definition `int inword;` nichts darüber aus, dass diese Variable nur als boolsche Variable verwendet wird. Dies kann man ausdrücken, indem man sich einen Typ `bool` definiert:

```
typedef int bool;
```

und dann `inword` als `bool` definiert. Dies ist für den menschlichen Leser Signal genug, der Compiler kann aber `int` und `bool` immer noch nicht unterscheiden, da in C `typedef`'s nur neue Namen für Typen einführen, aber keine neuen Typen. Dieses Manko kann dadurch ausgeglichen werden, dass wir einen *Aufzähl-Typ*, der nur die Werte FALSE und TRUE annehmen kann, einführen:

```
typedef enum {FALSE, TRUE} bool;
```

was der Modula-Definition

```
TYPE bool = (FALSE, TRUE);
```

entspricht. Wie in Modula werden auch in C den Konstanten einer enum-Deklaration aufsteigende Ordnungszahlen zugeordnet (`FALSE` ist also 0 und `TRUE` 1). Der Compiler könnte nun immer dann, wenn einer Variable vom Typ `bool` ein anderer Wert als 0 oder 1 zugewiesen wird (oder werden könnte), eine Warnung produzieren.

Achtung Enums sind unmittelbar nach dem Erscheinen von [Ker78] in C aufgenommen worden. Die Compiler, die zwischen dieser Zeit und dem Erscheinen von [C89] geschrieben wurden, interpretieren diesen Typ daher sehr unterschiedlich. Das reicht von sehr lockerer Handhabung (alle enums sind vom Typ `int`) bis zu noch restriktiveren Regeln als sie etwa in Modula üblich sind (einer enum Variable kann man ihre Mitglieder zuweisen, und man kann sie in Vergleichen verwenden, und sonst nichts). ANSI-C legt fest, dass jede enum ein eigener, ganzzahliger Typ ist. Damit können enums überall verwendet werden, wo auch andere ganzzahlige Typen erlaubt sind, der Compiler kann aber Wertebereiche überprüfen.

3.1.12 Compilation Units

Wie in Modula kann auch in C ein Programm aus mehreren Source-Files bestehen. Das hat mehrere Vorteile:

- Teile des Programms können unabhängig voneinander kompiliert werden. Bei Änderungen muss nicht das ganze Programm neu kompiliert werden, sondern nur das geänderte Source

File (Stellen Sie sich vor, Sie müssten bei jeder Änderung in Ihrem Programm die gesamte C-Library neu kompilieren!).

- Details der Implementierung können vor anderen Programmteilen versteckt werden. Wenn nur einige wenige Funktionen direkt auf eine Datenstruktur Zugriff haben, so kann diese Datenstruktur mit relativ geringem Aufwand geändert werden.
- Ein abgeschlossenes Modul, das bestimmte Aufgaben erfüllt, kann in anderen Programmen wiederverwendet werden. Aus einem monolithischen Programm einzelne Funktionen herauszulösen ist dagegen im Allgemeinen mit großem Aufwand verbunden [Spe88].

Auf Grund dieser Überlegungen werden auch wir die Funktionen zur tatsächlichen Verwaltung der Cross-Referenzliste in eine eigene *Compilation Unit* (dies ähnelt dem Konzept des *Moduls* in Modula) packen. Wir brauchen eine Funktion `new_xref`, die ein neues Paar (Wort, Zeilennummer) in die Liste der Cross-References einträgt und eine Funktion `print_xref`, die die Cross-References ausgibt, sowie eine Datenstruktur, die die Liste der Cross-References selbst enthält.

Stil

Obwohl C im Unterschied zu Modula keine Unterscheidung von Implementation und Definition Modules kennt und den Programmierer auch nicht zur Verwendung von Include-Files zwingt, hat es sich als praktisch herausgestellt, die Struktur von Modula-Programmen zu imitieren. Zu jedem C-File (das einem Implementation Module entspricht), sollte ein Header-File geschrieben werden, das die *Deklarationen* aller Funktionen, Variablen, Typen und Konstanten, die dieses C-File exportiert, enthält. Verwendet nun ein anderes C-File diese Funktionen, Variablen, etc., so sollte es das Header-File inkludieren und *nicht* entsprechende `extern`-Deklarationen enthalten. Auch das C-File selbst soll sein Header-File inkludieren. Auf diese Weise kann der Compiler Inkonsistenzen zwischen Modulen entdecken.

3.1.13 Structures

Datenstrukturen sind ein heikles Thema und sollten gut durchdacht sein, da eine ineffiziente Repräsentation der Daten durch noch so gefinkelte Programmierung nicht mehr auszugleichen ist.

Wie sollen wir also unsere Cross-Referenz-Liste darstellen? Wir nehmen an, dass jedes Wort mehrfach vorkommt, also scheint eine Struktur, die aus einem String und einem Array von Zeilennummern besteht, zur Repräsentation eines Wortes gut geeignet. Die gesamte Liste ist dann ein Array solcher Strukturen. Um Wörter, die bereits in der Liste sind, schnell finden zu können, sorgen wir dafür, dass die Liste jederzeit geordnet ist, indem wir neue Wörter an der richtigen Stelle einfügen.

Unsere Strukturdefinition sieht also wie folgt aus:

```
typedef struct wordentryT
{
    char    word [WORDLEN + 1];
    int     nr_numbers;
    int     number [MAXREPEAT];
} wordentryT;
```

`word` muss deshalb Länge `WORDLEN␣+␣1` haben, um das Wort mit `\0` abschließen zu können.

Hier möge der Leser[5] kurz innehalten, und sich überlegen, ob er nicht bessere Lösungen findet.

Haben Sie eine bessere Lösung gefunden? Ja? Lesen Sie trotzdem weiter!

```
/*** module xreflist
 *
 * Author:      Peter Holzer    (hp@vmars.tuwien.ac.at)
 * Date:        1992-02-16
 * Version:     1.0
 * Purpose:     Maintain a list of (word, line) pairs as they are
 *              needed for cross references.
 * Internals:   Words and lines are stored in fixed size arrays.
 *              We assume that many words will be repeated and that
 *              the list is finally printed in alphabetical order.
 *              Therefore we keep the words sorted all the time and
 *              have a list of numbers associated with each word.
 */


/*** Includes ----------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "xref.h"
#include "xreflist.h"
```

Header-Files, die nicht vom System bereitgestellt werden sondern zur Applikation gehören, werden in Anführungszeichen statt in spitze Klammern eingeschlossen.

`xref.h` enthält die Typendefinition von `bool`, die von `xref.c` exportierte Variable `cmnd`, sowie ein paar Makros, um das Programm leserlicher zu gestalten.

`xreflist.h` enthält die beiden von `xreflist.c` exportierten Funktionen. Es enthält auch die Kommentare, die vor den entsprechenden Funktionen stehen mit der Ausnahme, dass der Algorithmus und Zugriffe auf modullokale Variablen *nicht* beschrieben sind — d.h., das Headerfile enthält nur die Informationen, die für den Benutzer des Moduls wichtig sind.

```
/*** Macros ------------------------------------------------------*/
#define WORDLEN     15  /* more than 99% of all English words  */
#define MAXREPEAT   400 /* on how many lines can a word appear */
#define MAXWORDS    3000

#define NUMBERS_PER_LINE 8

typedef struct wordentryT
{
```

[5] Beiderlei Geschlechts.

```
    char    word [WORDLEN + 1];
    int     nr_numbers;
    int     number [MAXREPEAT];
} wordentryT;

static wordentryT list [MAXWORDS];
static int nr_words;
```

Normalerweise werden Funktionen und Variablen, die außerhalb einer Funktion definiert werden *exportiert*, d.h., sie können von anderen Modulen importiert werden. Um dies zu verhindern, werden sie als `static` definiert und sind nur mehr in diesem Modul verfügbar.

```
/*** Functions ----------------------------------------------------*/
/*** function new_xref
 *
 * Purpose:     Insert new (word, line) pair into xref list.
 * Algorithm:   Look for word using binary search. If already in list
 *              just add new line number. If not move all greater
 *              entries one back and insert new entry.
 * Changes:     list
 *              nr_words
 */

void new_xref (char const *word, int line)
{
    int first = 0;
    int last = nr_words;
    int mid;
    int cmp = 1;

    /*
     * perform the binary search.
     * (Algorithm from Wirth: _Programmieren in Modula 2_ p.41)
     * first will point to the word if it is already in the list,
     * or to the element before which it should be inserted.
     */

    while (first < last)
    {
        mid = (first + last) / 2;
        cmp = strncmp (word, list [mid].word, WORDLEN);
        if (cmp <= 0)
        {
            last = mid;
        }
        else
        {
            first = mid + 1;
```

```
        }
    }

    if (strncmp (word, list [first].word, WORDLEN) != 0)
    {
        /* Insert new word */
        if (nr_words < MAXWORDS)
        {
            memmove (list + first + 1, list + first,
                     (nr_words - first) * sizeof (wordentryT));
            strncpy (list [first].word, word, WORDLEN);
            list[first].nr_numbers = 1;
            list[first].number[0] = line;
            nr_words ++;
        }
        else
        {
            fprintf (stderr, "%s: Too many words. Change MAXWORDS in "
                     __FILE__ " and recompile.\n", cmnd);
```

Aufeinanderfolgende Strings werden vom C-Compiler zusammengefügt. Auf diese Art können Strings, die zu lang für eine Zeile sind, einfach auf mehrere Zeilen aufgeteilt werden. `__FILE__` ist ein vordefiniertes Makro, das den Namen des Source-Files enthält.

```
            exit (EXIT_FAILURE);
        }
    }
    else
    {
        /* Add new line number */
        if (list[first].nr_numbers < MAXREPEAT)
        {
            list[first].number[list[first].nr_numbers++] = line;
        }
        else
        {
            fprintf (stderr, "%s: Too many repetitions. Change MAXREPEAT in "
                     __FILE__ " and recompile.\n", cmnd);
            exit (EXIT_FAILURE);
        }
    }
}


/*** function print_xref
 *
 * Purpose:     Print cross reference list (rather obvious :-)
 * Algorithm:   Just go through list in two nested loops.
```

```
 * Uses:        list
 *              nr_words
 */

void print_xref (void)
{
   int w;
   int l;       /* schlechtes Beispiel ! */
```

Stil

Die Verwendung von kurzen Variablennamen für lokale Variablen (insbesondere Laufvariablen) ist im Allgemeinen durchaus zulässig. Der hier verwendete Variablenname `l` sollte aber vermieden werden, da er kaum von der Ziffer `1` unterscheidbar ist.

```
   /* a nice header */

   printf ("\nCross References:\n");

   /* for all the words */

   for (w = 0; w < nr_words; w ++)
   {
       printf ("%-*s", WORDLEN, list[w].word);

       /* print all the line numbers */

       for (l = 0; l < list[w].nr_numbers; l ++)
       {
           printf ("%8d", list[w].number[l]);
           if (l % NUMBERS_PER_LINE == NUMBERS_PER_LINE - 1 &&
               l != list[w].nr_numbers - 1)
           {
               /* line full and still something to print
                * --> start new line
                */
               printf ("\n%-*s", WORDLEN, " ");
           }
       }
       printf ("\n");
   }
}
```

Die Konstanten MAXREPEAT und MAXWORDS wurden so gewählt, dass ein ca. 1000 Zeilen langer englischer Text vom Programm bearbeitet werden kann. Wenn wir uns nun Speicherbedarf ($\approx$ 5 Megabytes bei 32-Bit `int`s) und Laufzeit (ca. 30 Minuten auf einer DECstation 5000/120) ansehen, so werden wir den Verdacht nicht los, dass wir etwas falsch gemacht haben.

Das Problem hat zwei Ursachen. Ursache Nummer eins ist, dass jedes Mal, wenn ein neues Wort auftaucht, alle bereits eingetragenen Wörter, die im Alphabet später vorkommen, um eine Position nach hinten verschoben werden müssen. Das wäre noch nicht so schlimm, wenn alle Felder, die auf diese Weise verschoben werden, tatsächlich sinnvolle Daten enthielten. Das ist aber nicht der Fall. In typischen Texten kommen die meisten Wörter sehr selten vor (bei unserem Test-Text etwa drei Mal) und einige wenige sehr häufig. Das Programm verbringt also den Löwenanteil seiner Zeit damit, Speicherbereiche zu kopieren, die gar keine Daten enthalten! Wenn wir es also schaffen, zu jedem Wort nur soviele Zeilennummern abzuspeichern, wie tatsächlich benötigt werden, so haben wir sowohl Platz als auch CPU-Zeit gespart.

Und damit kommen wir zum letzten Kapitel unserer Einführung in C:

3.1.14 Dynamische Speicherverwaltung

Wir haben ganz am Anfang festgestellt, dass Objekte, die über einen Namen ansprechbar sind Variablen genannt werden und damit impliziert, dass es auch namenlose Objekte gibt. Diese Objekte müssen explizit erzeugt und wieder zerstört werden und sind über ihre Adresse ansprechbar.

Dafür stellt C drei Standard-Funktionen zur Verfügung:

malloc erzeugt ein Objekt der angegebenen Größe.

free zerstört ein mit `malloc` erzeugtes Objekt und gibt den Speicherplatz wieder frei.

realloc ändert die Größe eines mit `malloc` erzeugten Objekts. Dabei kann es nötig sein, ein neues Objekt zu erzeugen und das alte frei zu geben. Nach `realloc` sind also alle Pointer, die noch auf das alte Objekt zeigen, ungültig.

Im Fehlerfall liefern `malloc` und `realloc` einen Null-Pointer zurück.

Der Operator `sizeof` liefert die Größe seines Arguments. Er ist sowohl auf Typen als auch auf Ausdrücke anwendbar und ersetzt damit die Modula-Prozeduren SIZE und TSIZE.

Gegenüber Modula und Pascal hat C den Vorteil, dass durch die nahe Verwandtschaft von Pointern und Arrays sehr einfach Arrays variabler Größe implementiert werden können. So wird mittels

```
int *p;
p = (int *) malloc (n * sizeof (int));
```

ein Array für `n ints` angelegt, das über den Pointer `p` in völlig gewohnter Weise angesprochen werden kann (`p[i]` ist also das `i`-te Element). Nachdem der Rückgabewert der Funktion `malloc` vom Typ `void` ist, wird hier noch explizit eine Typumwandlung in den Typ des Pointers `p (int *)` durchgeführt. Sollte sich im Laufe des Programms herausstellen, dass `n` Elemente doch nicht genug waren, so kann man die Größe des Arrays einfach verdoppeln:

```
n *= 2;
p = (int *) realloc (p, n * sizeof (int));
```

Man beachte, dass der alte und der neue Wert von `p` unterschiedlich sein können.

Gehen wir also daran, unser Modul `xreflist.c` komplett neu zu implementieren. Wir haben festgestellt, dass das Hauptproblem war, dass wir für jedes Wort 400 Zeilennummern vorgesehen haben, aber die meisten nur sehr wenige Zeilennummern brauchten. Um dieses Problem zu umgehen, ersetzen wir in `wordentryT` das Array `number` durch einen Pointer und allozieren jeweils soviel Platz wie wir brauchen. Das zweite Problem war, dass jeweils beim Einfügen eines neuen Wortes durchschnittlich die Hälfte aller bereits eingefügten Wörter verschoben werden müssen. Dies können wir verhindern, ohne auf die Vorteile einer binären Suche verzichten zu müssen, indem wir die Wörter in einem binären Baum anordnen[6].

Diese Änderungen verringern die benötigte CPU-Zeit auf weniger als eine Sekunde und den Speicherbedarf auf 280 Kilobyte.

```
/*** module xreflist
 *
 * Author:      Peter Holzer    (hp@vmars.tuwien.ac.at)
 * Date:        1992-02-17
 * Version:     2.0
 * Purpose:     Maintain a list of (word, line) pairs as they are
 *              needed for cross references.
 * Internals:   Words are stored in a binary tree.
 *              Each word has a ``dynamic array'' of line numbers
 *              attached to it.
 */


/*** Includes ----------------------------------------------------*/
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "xref.h"
#include "xreflist.h"


/*** Macros ------------------------------------------------------*/

#define INDENT  16
#define LINELEN 80
#define NUMLEN  8

/*** Types -------------------------------------------------------*/

typedef struct wordentry wordentryT;
```

[6]Wir gehen dabei davon aus, dass die Wörter in zufälliger Reihenfolge im Text vorkommen. Bei einer sortierten Liste von Wörtern würde unser Baum zu einer linearen Liste degenerieren. Ein balancierter Baum würde aber den Rahmen dieses Einführungsbeispiels sprengen.

```
struct wordentry
{
    char        *word;
    int         nr_alloc;    /* numbers allocated */
    int         nr_numbers;  /* numbers actually used */
    int         *number;
    wordentryT  *left;
    wordentryT  *right;
};

static wordentryT *tree;
```

```
/*** Functions ---------------------------------------------------*/

/*** function find_word
 *
 * Purpose:     Find an entry in the binary tree. If the entry is not
 *              in the tree already, it is created.
 * Algorithm:   Create new node if an empty tree is passed,
 *              else compare word with word in the node and search
 *              left or right subtree as necessary.
 * In:          word:   string to look for.
 * In/Out:      root:   points to a pointer to the root node of the
 *                      tree to be searched.
 * Returns:     pointer to a node containing the string word.
 */

static wordentryT *find_word (const char *word, wordentryT **root)
{
    wordentryT  *node = *root;
```

C kennt im Gegensatz zu Modula nur Wert-Parameter, aber keine Variablenparameter. Soll eine Funktion einen Parameter ändern, so muss ein Pointer auf diesen Parameter übergeben werden. `wordentryT **root` entspricht also ungefähr `VAR root: POINTER TO wordentryT` in Modula. Im Gegensatz zu Modula muss aber der Pointer explizit an die Funktion übergeben und in der Funktion explizit dereferenziert werden. Um dies zu vermeiden, führen wir eine zusätzliche Variable `node` ein, und weisen `*root` erst vor dem Ausstieg aus der Funktion den neuen Wert zu.

```
    if (node == NULL)
    {
        /*
         * empty tree -- create node
         */
        if ((node = (wordentryT *)malloc (sizeof (wordentryT))) == NULL)
        {
            (void)fprintf (stderr, "%s: cannot create new tree node: %s\n",
                    cmnd, strerror (errno));
            delete_tree(*root);
```

```
            exit (EXIT_FAILURE);
        }
        if ((node->word = (char *)malloc (strlen (word) + 1)) == NULL)
        {
            (void)fprintf (stderr,
                    "%s: cannot create word field in new tree node: %s\n",
                    cmnd, strerror (errno));
            delete_tree(*root);
            exit (EXIT_FAILURE);
        }
        strcpy (node->word, word);
        if ((node->number = (int *)malloc (sizeof (int))) == NULL)
        {
            (void)fprintf (stderr,
                    "%s: cannot create number field in new tree node: %s\n",
                    cmnd, strerror (errno));
            delete_tree(*root);
            exit (EXIT_FAILURE);
        }
        node->nr_numbers = 0;
        node->nr_alloc = 1;
        node->left = NULL;
        node->right = NULL;

        /* and return it */
        *root = node;
        return *root;
    }
    else
    {
        int cmp = strcmp (word, node->word);

        if (cmp < 0)
        {
            return find_word (word, & (node->left));
        }
        else if (cmp > 0)
        {
            return find_word (word, & (node->right));
        }
        else
        {
            return node;
        }
```

Stil

C kennt im Gegensatz zu Modula kein `ELSIF`. Da aber der `else`-Zweig ein `if`-Statement sein kann, lässt sich dieses einfach simulieren. In diesem Fall wird auf ein weiteres Einrücken des `else`-Zweiges verzichtet.

```
    }
}
```

```
/*** function new_xref
 *
 * Purpose:     Insert new (word, line) pair into xref list.
 * Algorithm:   Traverse tree to find word. If already in list
 *              just add new line number. If not create new entry.
 * Changes:     tree
 */

void new_xref (char const *word, int line)
{
    wordentryT  *node;

    node = find_word (word, &tree);

    /* Expand number list if necessary */
    if (node->nr_numbers >= node->nr_alloc)
    {
        node->nr_alloc *= 2;

        if ((node->number = (int *)
             realloc (node->number,
                      node->nr_alloc * sizeof (*node->number))
            ) == NULL)
        {
            (void) fprintf (stderr,
                     "%s: cannot expand number field: %s\n",
                     cmnd, strerror (errno));
            delete_tree(tree);
            exit (EXIT_FAILURE);
        }
    }

    /* and insert line number */
    node->number[node->nr_numbers++] = line;
}
```

```
/*** function print_tree
 *
 * Purpose:     print the tree
 * Algorithm:   for each node print left subtree, info in the node
```

```
 *              and right subtree.
 * In:          node:   root node of the tree.
 */

static void print_tree (wordentryT *node)
{
    int col;    /* current printing column  */
    int lni;    /* line number index    */

    if (node == NULL) return; /* empty tree */

    print_tree (node->left);

    if (printf ("%-*s", INDENT, node->word) < 0)
    {
      (void)fprintf(stderr, "%s: Cannot print to stdout: "
                    "%s\n", cmnd, strerror(errno));
      delete_tree(node);
      exit(EXIT_FAILURE);
    }
    col = strlen (node->word);
    if (col < INDENT) col = INDENT;

    /* print all the line numbers */
    for (lni = 0; lni < node->nr_numbers; lni ++)
    {
        if ((col += NUMLEN) > LINELEN)
        {
            if (printf ("\n%-*s", INDENT, "") < 0)
            {
                (void)fprintf(stderr, "%s: Cannot print to stdout: "
                              "%s\n", cmnd, strerror(errno));
                delete_tree(node);
                exit(EXIT_FAILURE);
            }
            col = INDENT + NUMLEN;  /* to match condition */
        }
        if (printf ("%*d",  NUMLEN, node->number[lni]) < 0)
        {
            (void)fprintf(stderr, "%s: Cannot print to stdout: "
                          "%s\n", cmnd, strerror(errno));
            delete_tree(node);
            exit(EXIT_FAILURE);
        }
    }
    if (printf ("\n") < 0)
    {
```

```
      (void)fprintf(stderr, "%s: Cannot print to stdout: "
                    "%s\n", cmnd, strerror(errno));
      delete_tree(node);
      exit(EXIT_FAILURE);
    }
    if (fflush(stdout) < 0)
    {
      (void)fprintf(stderr, "%s: Cannot flush stdout: "
                    "%s\n", cmnd, strerror(errno));
      delete_tree(node);
      exit(EXIT_FAILURE);
    }

    print_tree (node->right);
}
```

```
/*** function print_xref
 *
 * Purpose:     Print cross reference list (rather obvious :-)
 * Algorithm:   print_tree does all the work
 * Uses:        list
 *              nr_words
 */

void print_xref (void)
{
    if (printf ("\nCross References:\n") < 0)
    {
      (void)fprintf(stderr, "%s: Cannot print to stdout: "
                    "%s\n", cmnd, strerror(errno));
      delete_tree(tree);
      exit(EXIT_FAILURE);
    }
    print_tree (tree);
}
```

```
/*** function delete_tree
 *
 * Purpose:     Deletes the elements of the tree in order to free all
 *              dynamic memory
 * Algorithm:   Traverses the tree and deletes every single entry
 * Uses:
 *
 */

void delete_tree (wordentryT *p)
{
```

```
    delete_tree(p->left);
    delete_tree(p->right);
    free((void *)p->word);
    free((void *)p->number);
    free((void *)p);
}
```

3.2 Sprachbeschreibung

Nachdem im vorigen Kapitel die Sprache C anhand eines Beispiels vorgestellt wurde, um dem Leser ein Gefühl für die Sprache zu geben, werden wir nun eine exakte (wenn auch nicht ganz vollständige) Beschreibung der Sprache C liefern.

3.2.1 Translation Phases

Die Übersetzung eines C-Programms läuft in 8 Phasen ab. Das bedeutet nicht, dass jeder C-Compiler aus 8 Passes bestehen muss. Meist werden mehrere Phasen zu einem Pass zusammengefasst oder eine besonders komplexe Phase in mehrere Passes aufgespalten. Traditionellerweise besteht ein C-Compiler aus einem Präprozessor, dem eigentlichen Compiler, einem Assembler und einem Linker. Auch wir verwenden hier diese Einteilung.

Präprozessor

1. Wenn nötig, werden Zeichen aus dem Source-File in eine interne Darstellung übersetzt. Trigraphs[7] werden in die entsprechenden Zeichen übersetzt.
2. Zeilen, die mit einem Backslash enden, werden mit der folgenden Zeile vereinigt.
3. Das Source File wird in *Preprocessing Token* und *White Space Characters* zerlegt. Kommentare werden durch einzelne Spaces ersetzt.
4. Präprozessor-Anweisungen werden ausgeführt und Makros expandiert. Auf Files, die mittels `#include` eingelesen werden, werden die Phasen 1 bis 4 angewendet.

Compiler und Assembler

5. Zeichen und Escape-Sequenzen werden in Zeichen des Zielcomputers übersetzt.
6. Aufeinanderfolgende String-Konstanten werden vereinigt.
7. Das Programm wird analysiert und übersetzt.

[7] Da die Programmiersprache C Zeichen enthält, die nicht in allen üblichen Zeichensätzen enthalten sind, wurden für die fehlenden Zeichen sogenannte *Trigraphs* eingeführt (`??!` = `|`, `??'` = `^`, `??(` = `[`, `??)` = `]`, `??-` = `~`, `??/` = `\`, `??<` = `{`, `??=` = `#`, `??>` = `}`). Diese sollten aber nur in Notfällen verwendet werden.

Linker

8. Referenzen zu Objekten und Funktionen in anderen Compilation Units werden aufgelöst und ein lauffähiges Programm wird erstellt.

3.2.2 Typen und Konstanten

C verwendet eine relativ große Anzahl von Typen. Abb. 3.3 gibt einen Überblick über die Typen.

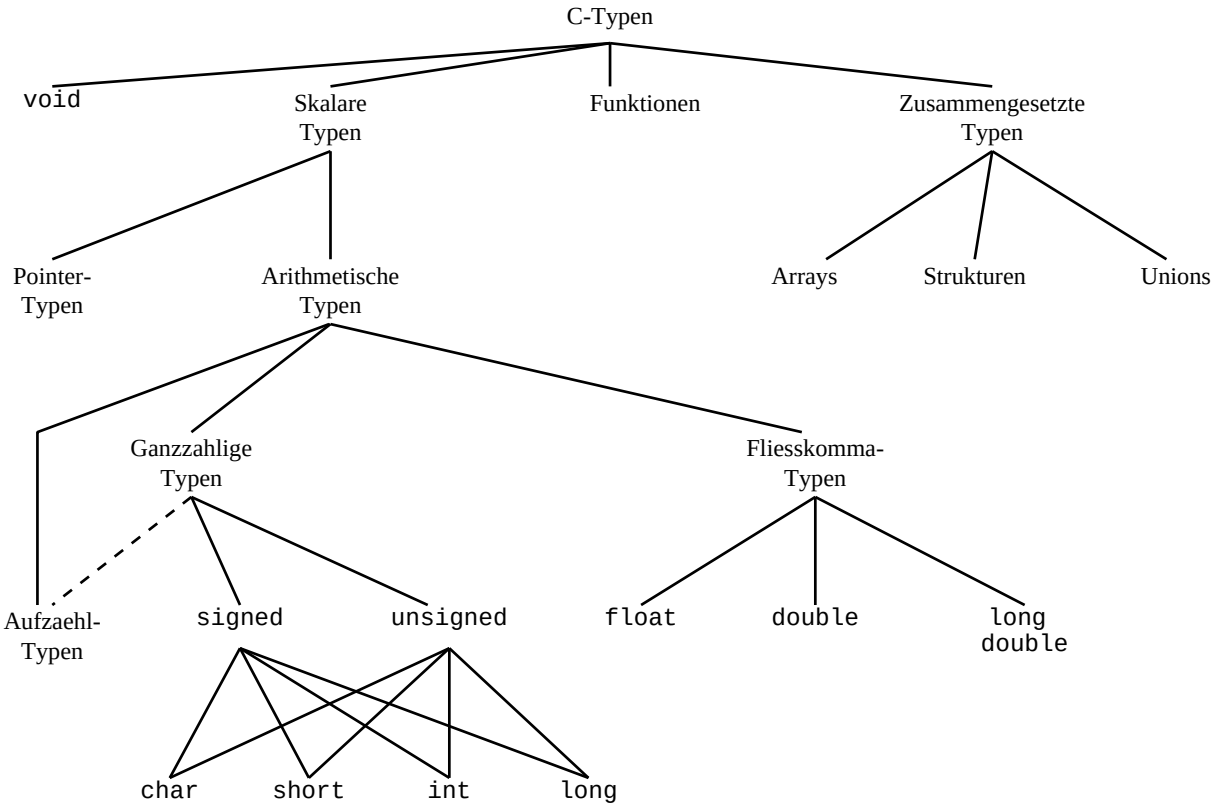


Abbildung 3.3: Überblick über die Typen in C

void

`void` zeigt die Abwesenheit eines Typs an. Es hat drei Anwendungsbereiche:

- Eine Funktion, die keinen Wert zurückliefert, hat Typ `void`.
- Das Wort `void` in einer Parameterliste zeigt an, dass eine Funktion keine Parameter hat (Im Gegensatz zur leeren Parameterliste, die (in einer Deklaration) anzeigt, dass eine Funktion eine unbekannte, aber fixe Anzahl von Parametern hat).
- Ein Pointer auf `void` kann auf beliebige Daten zeigen.

Ganzzahlige Typen

Es gibt in C vier ganzzahlige Typen (*integral types*): `char`, `short`, `int` und `long`. Alle diese Typen können mit (`signed`) und ohne (`unsigned`) Vorzeichen verwendet werden, wobei `signed` als default angenommen wird (außer bei `char`, wo es dem Compiler überlassen bleibt, ob er `signed` oder `unsigned` als default annimmt). Die Rechenregeln für diese Typen verlangen eine binäre Darstellung, wobei bei den signed Typen jede der drei üblichen Darstellungen (Zweier-Komplement, Einer-Komplement oder Sign-Magnitude) möglich ist. Tabelle 3.1 enthält die *minimalen* Wertebereiche der ganzzahligen Typen.

Typ	Minimum	Maximum
signed char	−128	127
unsigned char	0	255
char[a]	0	127
short	−32 768	32 767
unsigned short	0	65 535
int	−32 768	32 767
unsigned int	0	65 535
long	−2 147 483 648	2 147 483 647
unsigned long	0	4 294 967 295

[a] Eigentlich ist der Wertebereich von `char` entweder gleich dem von `signed char` oder `unsigned char`. Da dies aber vom Compiler abhängt, ist es sinnvoll, `char` als Schnittmenge von `signed` und `unsigned char` zu betrachten. In alten Compilerversionen wurde `char` praktisch ausschließlich als `signed` behandelt.

Tabelle 3.1: Wertebereiche der ganzzahligen Typen

`char` und `short`-Typen existieren nur als Objekt-Typen aber nicht als Wert-Typen. Wird der Wert eines solchen Objektes verwendet, so wird er in `int` umgewandelt, wenn der gesamte Wertebereich dieses Typs in einem `int` dargestellt werden kann, sonst in `unsigned int`[8].

Portabilität Manche ältere Compiler unterstützen nur eine Untermenge dieser Typen. Insbesondere das Keyword `signed` wird nur von wenigen Vor-ANSI-Compilern verstanden.

Viele ältere Compiler wandeln `unsigned short` und `unsigned char` *immer* in `unsigned int` um, nicht nur, wenn der Wert nicht in `int` dargestellt werden könnte. Diese Methode (unsigned preserving) wird auch in [Ker78] verwendet.

Konstante Ganzzahlige Konstanten können als dezimale, oktale oder sedezimale (bzw. hexadezimale) Zahlen sowie als Characters geschrieben werden.

Dezimale Konstanten haben die Form:

(`1—9`*){*`0—9`*}*

[8] Das führt dazu, dass der Ausdruck `((unsigned short)0 - 1 < 0)` auf Maschinen wo `sizeof (short) < sizeof (int)` 1 ergibt, auf Maschinen, wo beide gleich sind, jedoch 0.

\'	Einfaches Hochkomma
\"	Doppeltes Hochkomma
\a	Alarm (Bell)
\b	Backspace
\f	Formfeed
\n	Newline
\r	Return
\t	Horizontal Tab
\v	Vertical Tab
ooo	Zeichen mit Code *ooo* oktal. Immer 3 Ziffern
\x*xx*	Zeichen mit Code *xx* sedezimal.

Tabelle 3.2: Escape-Sequenzen in Character- und String-Konstanten

Dabei dienen runde Klammern dazu, einen Bereich von Zeichen anzugeben: *(*1—9*)* steht für eine Ziffer die größer oder gleich 1 und kleiner oder gleich 9 ist. Eine Menge von Zeichen, die in geschwungenen Klammern angegeben ist, steht für beliebig viele (auch null) Zeichen aus der angegebenen Menge.

Oktale Konstanten haben die Form:

0*{*0—7*}*

und sedezimale Konstanten haben die Form:

*(*0x/0X*)(*0—9/a—f/A—F*){*0—9/a—f/A—F*}*

Character-Konstanten schließlich werden in einfache Hochkommas eingeschlossen und bestehen entweder aus einem einzelnen Zeichen (außer Hochkomma und Newline) oder einer der Escape-Sequenzen in Tabelle 3.2.

Aufzähltypen

Aufzähltypen werden mittels einer `enum`-Deklaration erzeugt. Diese hat folgende Form, wobei zwischen eckigen Klammern angegebene Teile optional sind:

`enum` *enum-tag* `{` *identifier [= const-expression] { , identifier [= const-expression] }* `};`

Jede enum-Definition erzeugt einen neuen Typ und ganzzahlige Konstanten $identifier_i$, die – bei 0 beginnend – durchnummeriert werden. Optional können die Werte dieser Konstanten auch geändert werden.

Beispiele

`enum bool { FALSE, TRUE };` erzeugt einen neuen Typ `enum bool` und zwei Konstanten `FALSE` (mit Wert 0) und `TRUE` (mit Wert 1).

`enum colors { RED = 1, GREEN = 2, BLUE = 4 };` erzeugt einen Typ `enum colors` und drei Konstanten mit den angegebenen Werten.

In `enum people { BILLY = 17, BOBBY, JACK = 53, JANE = -3, JIM = 17};` haben sowohl `BILLY` als auch `JIM` den Wert 17 (Achtung, der Compiler gibt keine Warnung aus), und `BOBBY` hat den Wert 18 (durch automatische Zuweisung durch den Compiler).

Portabilität `enum`s werden von Vor-ANSI-Compilern auf sehr unterschiedliche Art implementiert. Portable Software muss daher so geschrieben werden, dass sie frei von Operationen ist, die von bestimmten Annahmen über Implementierungsdetails ausgehen.

Fließkommazahlen

C kennt drei Typen von Fließkommazahlen: `float`, `double` und `long double`. Der Wertebereich muss mindestens $10^{\pm 37}$ betragen, die Genauigkeit bei `float` 6 Dezimalstellen, bei `double` und `long double` 10 Dezimalstellen. `float` existiert nicht als Typ von Wertausdrücken. Wird ein Objekt vom Typ `float` in einem Wertausdruck verwendet, so wird es in einen `double`-Wert umgewandelt.

Portabilität `long double` ist eine Erfindung des ANSI-Kommittees.

Pointer

Zu jedem Typ gibt es einen Pointer-Typ, der die Adresse eines Objekts oder einer Funktion dieses Typs aufnehmen kann. Als Sonderfall kann der Typ pointer to void die Adresse eines beliebigen Objekts aufnehmen.

Pointer auf verschiedene Typen sind nicht zuweisungskompatibel (Ausnahme: `void *` ist mit allen Pointern auf Datentypen zuweisungskompatibel, aber nicht mit Pointern auf Funktionen).

Der Null-Pointer (d.h., ein Pointer, der auf kein Objekt zeigt) wird durch die *Konstante* `0` (üblicherweise `NULL` geschrieben) dargestellt. Man beachte, dass diese Konstante abhängig vom Kontext in einen Null-Pointer des benötigten Typs umgewandelt wird. Fehlt dieser Kontext (z.B. `NULL` als Argument einer Funktion für die kein Prototyp existiert) so ist das Ergebnis undefiniert.

Portabilität `void *` wurde von ANSI als generischer Pointer eingeführt. Ältere Compiler verwenden stattdessen `char *`.

Mit casts erzwungene Zuweisungen von unterschiedlichen Pointer-Typen können illegale Pointer erzeugen. Portabel ist nur die Zuweisung von Pointern, die auf "größere" Datentypen zeigen, auf Pointer, die auf "kleinere" Datentypen zeigen, möglich. Zuweisungen in die andere Richtung können zu Fehlern im Alignment (in der Ausrichtung) führen. Dieser Fall kann z.B. eintreten, wenn der Wert eines Character-Pointers, der auf eine ungerade Byteadresse zeigt, an einen Integer-Pointer, der stets auf eine Wort– oder Langwortadresse zeigen muss, zugewiesen wird. Die Ergebnisse beim Dereferenzieren solcher Pointer sind natürlich von der internen Darstellung der Datentypen abhängig.

Arrays

Ein Array ist eine Ansammlung mehrerer Elemente gleichen Typs, die über einen Index angesprochen werden können. Der Element-Typ kann ein beliebiger kompletter Objekt-Typ sein. Die Anzahl der Elemente muss ein konstanter, ganzzahliger Ausdruck sein. Hat das Array N Elemente, so sind die Elemente von 0 bis $N-1$ durchnummeriert. Die Adresse des N-ten Elements ist gültig, kann aber nicht mehr dereferenziert werden. Dies kann das Programmieren von Schleifen (und deren Abbruchbedingungen), in denen ein Pointer über alle Elemente eines Arrays fortgeschaltet wird, erleichtern.

Bei der *Deklaration* eines Arrays kann die Anzahl der Elemente des Arrays weggelassen werden. Ein Array ohne Anzahl der Elemente ist kein kompletter Typ.

Beispiele

`char s[MAXLINE];` definiert `s` als Array von `MAXLINE` Characters. `double vector[3];` Array `vector` von 3 `double`s. `extern char *messages[];` deklariert `messages` als Array unbekannter Größe von `char *`. `int matrix[10][20];` definiert ein Array `matrix` von 10 Elementen, wobei jedes ein Array von 20 `int`'s ist.

Arrays können nicht in Wertausdrücken vorkommen. Stattdessen wird ein Pointer auf das nullte Element verwendet.

Strukturen

Eine Struktur ist eine Ansammlung mehrerer Elemente eventuell unterschiedlichen Typs, die über einen Namen angesprochen werden können. Elemente einer Struktur können jeden kompletten Objekttyp haben.

Eine Strukturdefinition hat folgende Form:

`struct` *struct-tag* `{` *Elementdefinitionen* `}`

Jede Strukturdefinition erzeugt einen neuen Typ, der mit keinem anderen Typ zuweisungskompatibel ist, und einen eigenen Namespace für seine Elemente (d.h., zwei Elemente von verschiedenen Strukturen können den selben Namen haben).

Beispiele

`struct complex { double x, y; };` erzeugt einen Typ `struct complex`. `struct complex a;` definiert eine Variable `a` dieses Typs. Das könnte auch durch `struct complex {double x, y; } a;` geschehen.

```
struct nodeT
{
    struct nodeT *left, *right;
    char         *name;
    int           value;
};
```

definiert eine `struct nodeT`, die zwei Pointer `left` und `right` auf Objekte desselben Typs, sowie einen `char *` und einen `int` enthält. Obwohl die struct nodeT zum Zeitpunkt der Definition von `left` und `right` noch nicht komplett ist, können Pointer auf sie definiert werden.

Portabilität In [Ker78] haben die Elemente aller Strukturen einen gemeinsamen Namespace.

Da der Compiler an beliebigen Stellen in der Struktur "Füllbytes" einfügen kann, um nachfolgende Elemente auf legale Adressen zu bringen, ist die Verwendung von Strukturen zur Darstellung externer Daten problematisch.

Bitfields

Bei ganzzahligen Elementen von Strukturen kann die Größe in Bit angegeben werden. Der Compiler kann dann mehrere dieser *Bitfields* in ein Maschinenwort packen. Bei Bitfields können zwei Sonderfälle auftreten:

- Ein Bitfield kann keinen Namen haben. Ein namenloses Bitfield kann verwendet werden, um eine Anzahl von Bits zu überspringen.
- Ein Bitfield von Länge 0 Bit weist den Compiler an, alle weiteren Bits in diesem Maschinenwort zu überspringen. Ein solches "leeres" Bitfield *darf keinen* Namen haben.

Beispiele

```
struct modeT
{
    unsigned int    type  :4;
    unsigned int    suid  :1;
    unsigned int    guid  :1;
    unsigned int    sticky:1;
    unsigned int    user  :3;
    unsigned int    group :3;
    unsigned int    world :3;
}
```

Dies definiert eine Struktur, die die gleiche Information auf gleichem Platz (aber nicht unbedingt in gleicher Darstellung) enthält wie das `st_mode`-Feld in einem Unix-Inode. Eine solche Definition könnte Zugiffe auf diese Information lesbarer gestalten (z.B. `if (sb.st_mode.type == REGULAR)` statt `if ((sb.st_mode & S_IFMT) == S_IFREG)`), andererseits würde sie die gemeinsame Bearbeitung mehrerer Felder (z.B. der 3 Permission-Felder) unmöglich machen.

Portabilität Der Typ `int` kann in Bitfields ein Vorzeichen haben oder nicht. Wenn ein Vorzeichen benötigt wird, so ist explizit das Keyword `signed` zu verwenden.

ANSI C sieht die Typen `int, signed int`, und `unsigned int` für Bitfields vor. Viele ältere Compiler unterstützen nur `int` und/oder `unsigned int`, manche aber beliebige ganzzahlige Typen.

Die maximale Länge eines Bitfields und die Anordnung von Bitfields in einem Maschinenwort hängen vom Compiler ab.

Unions

Unions sind "Überlagerungen" von Typen. Eine Union entspricht einer Struktur mit dem Unterschied, dass alle Elemente den selben Speicherplatz belegen. Eine Union kann daher zu jedem Zeitpunkt nur eines ihrer Elemente enthalten.

Beispiele

```
union yystype
{
    long    iconst;
    char   *ident;
    double  fconst;
};
```

definiert eine Union, die einen long, einen char * oder einen double Wert aufnehmen kann. Eine solche union könnte in einem Compiler vorkommen und den Wert oder Namen eines Tokens enthalten.

```
union wordT
{
    unsigned int  word;
    unsigned char byte [sizeof (int)];
};
```

definiert eine Union, über die man ein Maschinenwort sowohl als ganzes, als auch als einzelne Bytes ansprechen kann (Das setzt natürlich voraus, dass ein `int` wirklich genau ein Maschinenwort lang ist).

3.2.3 Definitionen und Deklarationen

Scope, Lebensdauer und Linkage

Der Teil eines Programms, in dem eine Deklaration sichtbar ist, nennt man *Scope*. Deklarationen innerhalb eines Block-Statements haben Block-Scope, d.h., sie sind bis zum Ende des Blocks sichtbar. Deklarationen außerhalb von Funktionen haben File-Scope, d.h., sie sind bis zum Ende des Files sichtbar. In engem Zusammenhang damit steht die *Linkage*. Objekte mit *externer Linkage* können von anderen Modulen mittels `extern`-Deklaration importiert werden, solche mit *interner Linkage* dagegen nicht.

Objekte unterscheiden sich durch ihre *Lebensdauer*. Objekte mit statischer Lebensdauer werden beim Programmstart erzeugt (und initialisiert), Objekte mit automatischer Lebensdauer hingegen werden jedes Mal neu erzeugt (und eventuell initialisiert), wenn das Blockstatement, indem sie definiert sind, ausgeführt wird und danach wieder zerstört.

Deklarationen haben in C folgende Form:

Basistyp Deklarator { `,` *Deklarator* } `;`

Der Basistyp kann wiederum aus einem *Storage-Class Specifier*, einem *Type Specifier* und einem *Type Qualifier* bestehen.

Storage-Class Specifier

Ein *Storage-Class Specifier* ist eines der Keywords `auto, extern, register, static, typedef`. Er gibt an, wie ein Objekt in Bezug auf seine Linkage und Lebensdauer zu behandeln ist (siehe Tabelle 3.3).

Typedef bestimmt natürlich nicht wirklich die Storage-Class eines Objekts oder einer Funktion, sondern definiert einen neuen Typnamen.

	Block Scope			File Scope		
	Linkage	Dauer	Definition	Linkage	Dauer	Definition
`auto`	Keine	automatisch	ja	—	—	—
`register`	Keine	automatisch	ja	—	—	—
`extern`	?	statisch	nein	extern	statisch	ja
`static`	Keine	statisch	ja	intern	statisch	ja

Tabelle 3.3: Auswirkungen der Storage-Class Specifier und Ort der Vereinbarung auf Linkage und Lebensdauer

Type Specifier

Ein Type Specifier ist `void`, ein ganzzahliger oder Fließkommatyp, eine Struktur oder Union oder ein mit `typedef` definierter Typname.

Type Qualifier

Type Qualifier in C sind `const` und `volatile`. `const` hat die Bedeutung “dieses Objekt darf nicht vom Programm aus geändert werden”, `volatile` bedeutet “dieses Objekt kann seinen Wert unvorhersehbar ändern”.

Type Qualifier können auch im Deklarator vorkommen. Um festzustellen, auf welchen Teil des Typs sich der Qualifier bezieht, ist die Deklaration von innen nach außen zu lesen.

Beispiele

`volatile sig_atomic_t flag;`[9] definiert eine Variable `flag` vom Typ `sig_atomic_t`, deren Wert sich jederzeit ändern kann. Variablen, die von Signal-Handlern geändert werden, um anzuzeigen, dass ein Signal aufgetreten ist, sollten so deklariert werden.

`const char *s` definiert einen Pointer to char, der nicht dazu verwendet werden kann, den String, auf den er zeigt, zu verändern. Dem Pointer selbst kann aber ein neuer Wert zugewiesen werden.

`char *const s` definiert einen Pointer to char, dem keine Werte zugewiesen werden können. Der String, auf den er zeigt, kann aber verändert werden.

`char const *const s` schließlich ist ein Pointer, der nicht verändert werden kann und der auch nicht dazu verwendet werden kann, den String, auf den er zeigt, zu verändern.

`const volatile int *status_port;` ist ein Pointer auf eine Speicherstelle, die zwar vom Programm nicht verändert werden darf, aber ihren Wert "von selbst" ändern kann. Z.B. ein Read-Only-Register eines Devices.

Deklaratoren

Ein Deklarator besteht aus einem Ausdruck, der festlegt, wie aus dem Basistyp der Typ der Variable konstruiert wird. Es gibt vier Formen von Deklaratoren:

Identifier

Identifier hat den Basistyp.

Deklarator [*konstanter-ganzzahliger-Ausdruck*]

Deklarator ist ein Array mit *konstanter-ganzzahliger-Ausdruck* Elementen des Basistyps. In Deklarationen und initialisierten Definitionen kann die Größenangabe entfallen. Im letzteren Fall wird die Größe des Arrays aus der Anzahl der Elemente in der Initialisierung bestimmt.

Deklarator (*Parameterliste*)

Deklarator ist eine Funktion, die den Basistyp zurückliefert und die in `()` angegebenen Parameter hat. Fehlt diese Liste, so wird eine unbekannte, aber fixe Anzahl von Parametern angenommen.

⋆ *Deklarator* oder
⋆ *Type-Qualifier Deklarator*

Deklarator ist ein Pointer (möglicherweise `const` oder `volatile`) auf den Basistyp.

[9] `sig_atomic_t volatile flag;` hat dieselbe Bedeutung.

Diese Konstrukte können beliebig geschachtelt werden. Runde Klammern können verwendet werden, um Prioritäten zu setzen.

Beispiele

`char c` ist ein `char`.

`char c[128]` ist ein Array von 128 `char`.

`char *c` ist ein Pointer auf `char`.

`char *a[128]` ist ein Array von 128 `char *`.

`char (*a)[128]` ist dagegen ein Pointer auf ein Array von 128 `char`.

`void *f(int)` ist eine Funktion, die einen Parameter vom Typ `int` hat und einen Wert vom Typ `void *` zurückliefert.

`void (*f(int))` deklariert einen Pointer auf eine Funktion mit einem Argument vom Typ `int` welche keinen Rückgabewert hat. Die Funktion kann mittels `(*f)(arg)` oder in der Kurzform `f(arg)` aufgerufen werden.

`void (*signal (int sig, void (* func)(int)))(int)` deklariert eine Funktion `signal`, die zwei Argumente hat: Das erste (`sig`) ist ein `int`, und das zweite (`func`) ist ein Pointer auf eine Funktion, die ein Argument vom Typ `int` hat und keinen Returnwert liefert. Der Returnwert von `signal` ist ein Pointer auf die übergebene Funktion mit Parametertyp `int`. Etwas übersichtlicher lässt sich `signal` deklarieren, wenn man vorher einen Typ `sighandler_t` definiert:

`typedef void (*sighandler_t) (int sig);` Dann kann signal mittels `sighandler_t signal (int sig, sighandler_t func);` definiert werden.

3.2.4 Ausdrücke

Objekte, Ausdrücke und Typen

Jeder Ausdruck in C hat einen Wert, einen Typ und einen Kontext. Der Kontext sagt aus, ob der Ausdruck ein Wert- oder ein Objektausdruck ist. Ein Objektausdruck ist ein Ausdruck, der ein Objekt, also einen Speicherbereich bezeichnet.

Syntaktisch sind Objektausdrücke einer Untermenge der Wertausdrücke: Jeder gültige Objektausdruck kann in einem Wert-Kontext verwendet werden (z.B. kann `a` sowohl die Variable `a` als auch ihren Wert bezeichnen). Es wird dann nicht das Objekt selbst, sondern nur sein Wert genommen. Dagegen gibt es sehr wohl Wertausdrücke, die keine Objektausdrücke sind (z.B. `(a+b)` oder `3.14`).

Die in Objektausdrücken erlaubten Typen sind hingegen eine Obermenge der in Wertausdrücken erlaubten. Die "kurzen" ganzzahligen Typen (`char`, `short`) und `float` existieren als Objekttypen[10],

[10]Um Speicherplatz zu sparen und um möglichst alle Typen, die die Maschine kennt, direkt ansprechen zu können.

in Berechnungen werden sie jedoch immer in `int` bzw. `double` umgewandelt. Auch Arrays existieren nur als Objekttypen. Im Wertkontext werden sie durch einen Pointer auf ihr nulltes Element ersetzt. Dies ist die erste Art von Typ-Umwandlung, die in C vorkommt[11].

Die zweite Art von Typ-Umwandlungen kommt dadurch zustande, dass viele Operatoren nicht auf allen möglichen Kombinationen von Typen definiert sind. Im Gegensatz zu manchen anderen Programmiersprachen erzeugt aber der Versuch, einen Integer mit einer Fließkommazahl zu multiplizieren, keinen Fehler. Der Compiler fügt einfach die entsprechende Umwandlung ein. In arithmetischen Ausdrücken wird dabei immer der kleinere in den größeren Typ umgewandelt, wobei Floating-Point-Typen größer sind als ganzzahlige, und die `unsigned`-Typen größer als die entsprechenden `signed`-Typen sind.

Diese beiden Umwandlungsarten genügen in den meisten Fällen. In einigen Fällen ist es aber notwendig, explizit einen Typ in einen anderen umzuwandeln. Dazu dient ein sogenannter *cast*[12]. *Achtung: Cast*s sind wirklich nur dort zu verwenden, wo es absolut notwendig ist. Ansonsten sollten die Typen von Variablen von vornherein so gewählt werden, dass sie mit den Operationen, die auf diesen Variablen ausgeführt werden, kompatibel sind.

Die Operatoren

C hat insgesamt 46 Operatoren auf 15 Prioritätsstufen, die 1, 2 oder 3 Argumente verlangen. Bei gleicher Priorität werden manche Operatoren von links nach rechts (*linksassoziativ*: `a - b - c` ist gleich bedeutend mit `(a - b) - c`), andere von rechts nach links zusammengefasst (*rechtsassoziativ*: `a = b = c` ist gleich bedeutend mit `a = (b = c)`).

Im Folgenden sind die Operatoren nach fallender Priorität aufgeführt. Wo nicht anders notiert, sind sowohl die Argumente der Operatoren als auch ihr Ergebnis Wertausdrücke.

Selektionsoperatoren

Selektionsoperatoren haben die höchste Priorität und sind linksassoziativ.

`(` *exp* `)` Runde Klammern werden zum Gruppieren von Ausdrücken verwendet. Der Typ des Ausdrucks ändert sich dadurch nicht.

exp_1 `[` exp_2 `]` Eckige Klammern werden für Subskripts verwendet. Eine der beiden Ausdrücke muss vom Typ pointer to T sein, der andere von einem ganzzahligen Typ. Das Ergebnis ist ein Objektausdruck vom Typ T. $exp_1[exp_2]$ ist äquivalent zu $*(exp_1 + exp_2)$.

exp `(` *argumentlist* `)` Funktionsaufruf. Ist *exp* vom Typ pointer to function returning T, und ist *argumentlist* eine durch Kommas getrennte Liste von Ausdrücken richtigen Typs und richtiger Anzahl für diese Funktion, so ist das Ergebnis vom Typ T.

[11] Ähnlich liegt der Fall bei Funktionen. Eine Funktion und ihre Adresse sind in der Verwendung völlig gleichwertig. Beide können einer entsprechenden Pointer-Variablen zugewiesen oder aufgerufen werden. Da Funktionen aber keine Objekte sind, kann man hier nicht von einer Typumwandlung sprechen.

[12] Von *coerce*, erzwingen. Die Wandlung des Wortes coerced zu cast (Gipsverband) ist ein gutes Beispiel für die Mächtigkeit und Gefährlichkeit dieses Konstrukts.

`exp . struct-member` Hat *exp* den Typ struct T und ist *struct-member* ein Mitglied von struct T, so hat das Ergebnis Typ und Wert des Strukturelementes. Ist *exp* ein Objektausdruck, so ist auch der gesamte Ausdruck ein Objektausdruck.

`exp -> struct-member` Hat *exp* den Typ pointer to struct T und ist *struct-member* ein Mitglied von struct T, so hat das Ergebnis Typ und Wert des Strukturelementes. Das Ergebnis ist ein Objektausdruck.

Postfixoperatoren

`exp ++` Erhöht den Wert des Objektes *exp* um 1. Das Ergebnis des gesamten Ausdrucks *exp*++ ist der Wert, den *exp* vorher hatte.

`exp --` Erniedrigt den Wert des Objektes *exp* um 1. Das Ergebnis des Ausdrucks *exp*− ist der Wert, den *exp* vorher hatte.

Achtung: Die kompakte Schreibweise von Inkrement bzw. Dekrementoperationen verlockt immer wieder dazu, die resultierenden Ausdrücke auch in komplizierteren Ausdrücken zu verwenden, deren Semantik jedoch schwer ersichtlich ist (Man betrachte z.B. den Ausdruck i++ + i−. Ist das Resultat $2i$ oder $2i$+1?). Es muss daher immer darauf geachtet werden, dass Programme nicht auf Grund von verkürzten Schreibweisen an Lesbarkeit und Verständlichkeit verlieren.

Präfix- oder Unäre Operatoren

Diese Operatoren haben alle ein Argument. Sie sind rechtsassoziativ.

`++ exp` Erhöht den Wert des Objektes *exp* um 1. Das Ergebnis des Ausdrucks ++*exp* ist der neue Wert des Objektes.

`-- exp` Erniedrigt den Wert des Objektes *exp* um 1. Das Ergebnis des Ausdrucks ist der neue Wert des Objektes.

`sizeof exp` Liefert die Größe des Objekts vom selben Typ wie *exp*. Das Ergebnis ist vom Typ *size_t*.

`sizeof (type)` Liefert die Größe eines Objekts des angegebenen Typs.

`~ exp` Bitweise Negation. *exp* muss ein ganzzahliger Typ sein.

`! exp` Logische Negation. *exp* muss einen arithmetischen oder Pointer-Typ haben. Das Ergebnis ist vom Typ `int` und 1, wenn *exp* den Wert 0 hat, sonst 0. Äquivalent zu `(exp == 0)`

`- exp` Arithmetische Negation. *exp* muss einen arithmetischen Typ haben.

`+ exp` Tut nichts.

`& exp` Addressoperator. Ist *exp* ein Objektausdruck vom Typ T, so ist das Ergebnis die Adresse dieses Typs und ein Wertausdruck vom Typ pointer to T.

`* exp` Dereferenzierungsoperator. Ist *exp* ein Ausdruck vom Typ pointer to T der auf ein Objekt zeigt, so ist das Ergebnis ein Objektausdruck vom Typ T, der dieses Objekt beschreibt.

`( type ) exp` Cast. Wandelt den Ausdruck *exp* in einen Ausdruck des Typs *type* um. *type* und *exp* können beliebige arithmetische oder Pointer-Typen haben.

Multiplikative Operatoren

Multiplikative Operatoren sind `*` (Multiplikation), `/` (Division) und `%` (Rest). Multiplikation und Division sind auf allen arithmetischen Typen definiert, der Rest nur auf ganzzahligen. Bei ganzzahliger Division hängt es von der Implementierung ab, in welche Richtung gerundet wird. Die Gleichung `a % b == a - (a / b) * b` muss aber immer erfüllt sein.

Additive Operatoren

Additive Operatoren sind + und −. Auf arithmetische Typen angewendet, entsprechen sie den üblichen mathematischen Funktionen.

+ kann außerdem auf einen Pointer `p` und einen ganzzahligen Ausdruck `i` angewendet werden. Das Ergebnis ist ein Pointer, der `i` Elemente[13] hinter `p` zeigt. `p + i` ist also äquivalent zu `&p[i]`.

− kann zur Berechnung der Differenz zweier Pointer verwendet werden. Beide Pointer müssen vom selben Typ sein und in dasselbe Objekt (Array) zeigen. Das Ergebnis ist vom Typ `ptrdiff_t`.

Shift-Operatoren

« shiftet den linken Operanden um die im rechten Operanden angegebene Anzahl von Bits nach links, wobei 0-Bits nachgeschoben werden.

» shiftet entsprechend nach rechts, wobei 0-Bits nachgeschoben werden, wenn der linke Operand nicht negativ ist.

Shift-Operatoren sind nur auf ganzzahligen Typen definiert. Wird um mehr als die Breite des Typs oder um einen negativen Betrag geshiftet, so ist das Ergebnis undefiniert. Wird eine negative Zahl nach rechts geshiftet, so hängt das Ergebnis von der Implementierung ab.

Relationale Operatoren

<, <=, >= und > liefern das `int`-Ergebnis 1, wenn die Bedingung zutrifft, sonst 0. Alle arithmetischen Typen können miteinander verglichen werden. Zwei Pointer können miteinander verglichen werden wenn sie vom selben Typ sind und in dasselbe Objekt (z.B. Array) zeigen.

Gleichheit und Ungleichheit

== (Test auf Gleichheit) und `!=` (Test auf Ungleichheit) liefern das `int`-Ergebnis 1, wenn der Test zutrifft, sonst 0. Alle arithmetischen Typen können miteinander verglichen werden. Zwei Pointer können miteinander verglichen werden, wenn sie vom selben Typ sind oder einer vom Typ pointer

[13] Nicht Bytes!

to void ist. Zwei Pointer sind gleich, wenn sie auf dasselbe Objekt zeigen. Kein Pointer auf ein Objekt ist gleich dem Null-Pointer.

Bitweises Und

`&` ist auf ganzzahlige Typen definiert. Im Ergebnis ist jedes Bit gesetzt, das in beiden Operanden gesetzt ist.

Bitweises Exklusiv-Oder

`^` ist auf ganzzahlige Typen definiert. Im Ergebnis ist jedes Bit gesetzt, das in genau einem der Operanden gesetzt ist.

Bitweises Oder

`|` ist auf ganzzahlige Typen definiert. Im Ergebnis ist jedes Bit gesetzt, das in mindestens einem der Operanden gesetzt ist.

Logisches Und

`&&` liefert 1, wenn beide Operanden ungleich 0 sind. Der linke Operand wird zuerst ausgewertet. Ist er 0, so wird der zweite Operand nicht ausgewertet.

Logisches Oder

`||` liefert 1, wenn mindestens einer der Operanden ungleich 0 ist. Der linke Operand wird zuerst ausgewertet. Ist er ungleich 0, so wird der zweite Operand nicht ausgewertet.

Bedingte Auswertung

exp_1 `?` exp_2 `:` exp_3
Zuerst wird exp_1 ausgewertet. Ist es ungleich 0, so wird exp_2 ausgewertet, sonst exp_3. Das Ergebnis ist das des zuletzt ausgewerteten Teilausdrucks. exp_1 kann einen arithmetischen oder Pointer-Typ haben, exp_2 und exp_3 können von beliebigem Typ sein (inklusive `void`), müssen aber den selben Typ haben. Rechts assoziativ.

Zuweisung

Zuweisungen (=) sind rechtsassoziativ. Beliebige arithmetische Typen können einander zugewiesen werden. Beliebige Pointer-Typen können einem `void *` zugewiesen werden, und ein `void *` kann einem beliebigen Pointer-Typ zugewiesen werden. Werte von Pointer, Struktur, und Union-Typen können Objekten desselben Typs zugewiesen werden.

Zu den binären Operatoren `*`, `/`, `%`, `+`, `-`, «, » existiert jeweils ein verkürzter Zuweisungsoperator *op*=. Der Ausdruck *a op= b* ist äquivalent zu *a = (a) op (b)*, mit der Ausnahme, dass *a* nur ein Mal ausgewertet wird.

Sequentielle Auswertung

Der Komma-Operator (`,`) wertet zuerst sein linkes und dann sein rechtes Argument aus. Das Ergebnis ist das Ergebnis des rechten Arguments. Links-assoziativ.

3.2.5 Programmstruktur

Ein Statement kann in C folgende Formen annehmen:

Label `:` *Statement*
[Expression]`;`
`return` *[Expression]*`;`
`goto` *[Label]*`;`
`break ;`
`continue ;`
If-Statement
Switch-Statement
While-Statement
Do-Statement
For-Statement
Block-Statement

Expression-Statements

Die Expression wird ausgewertet, das Ergebnis ignoriert.

Achtung Viele Compiler warnen allerdings, wenn in einer Expression ein Wert berechnet wird, der nicht verwendet (oder durch expliziten Cast auf `void` verworfen) wird.

Labels

Es gibt zwei Arten von Labels. `case`-Labels bestehen aus dem Keyword `case` gefolgt von einer ganzzahligen Konstante oder dem Keyword default. Sie können nur innerhalb von Switch-Statements vorkommen. Sprunglabels dagegen haben die Form eines Identifiers, und können mittels `goto` angesprungen werden. Sie sind innerhalb der gesamten Funktion sichtbar.

Sprungbefehle

`return` *`[expression]`* Verlässt die Funktion und liefert den Wert von *Expression* an die aufrufende Funktion.

`goto` *`label`* Springt zum genannten Label. Dieses Konstrukt darf in der Übung nicht verwendet werden.

`break` Verlässt das innerste umgebende Switch- oder Schleifenstatement.

`continue` Überspringt den Rest des Schleifenbodies und beginnt nächste Iteration.

Verzweigungen

`if` `(` *Expr* `)` *Statement*$_1$ *[* `else` *Statement*$_2$ *]*

Wertet *Expr* aus. Ist der Wert ungleich 0, so wird *Statement*$_1$ ausgeführt, sonst *Statement*$_2$ (wenn vorhanden).

`switch` `(` *Expr* `)` *Statement*

Wertet *Expr* aus und setzt dann die Ausführung beim entsprechenden Case-Label in *Statement* fort. Ist kein passendes Label vorhanden, so wird beim `default`-Label fortgesetzt. Ist auch kein `default`-Label vorhanden, so wird *Statement* übersprungen. In der Praxis hat das Switch-Statement meist die Form:

```
switch (expr)
{
    case CONST1:
    case CONST2:
        statement;
        ...
        break;
    case CONST3:
        ...
        break;
    default:
        statement;
        ...
}
```

Achtung Eine Besonderheit des Switch-Statements ist, dass die Programmausführung im Body bis zum Ende oder nächsten `break` ausgeführt wird. Das Code-Segment

```
switch (c)
{
    case '\n':
        newline ++;
        /* FALLTHROUGH */
    case ' ':
    case '\t':
        whitespace ++;
        break;
    default:
        printing ++;
}
```

erhöht also `newline` *und* `whitespace`, wenn c ein `'\n'` ist, nur `whitespace`, wenn c ein Blank oder Tab ist, und in allen anderen Fällen `printing`. Da die-

ser Effekt nur selten beabsichtigt ist, ist er mit einem Kommentar wie zum Beispiel `/*FALLTHROUGH*/` zu kennzeichnen.

Schleifen

`while (` *Expr* `)` *Statement*

Expr wird vor jedem Schleifendurchlauf ausgewertet. Ist es null, so wird die Schleife abgebrochen.

`do` *Statement* `while (` *Expr* `);`

Expr wird nach jedem Schleifendurchlauf ausgewertet. Ist es null, so wird die Schleife abgebrochen.

`for (` $Expr_1$ `;` $Expr_2$ `;` $Expr_3$ `)` *Statement*

Entspricht der While-Schleife:
$Expr_1$ `;`
`while (`$Expr_2$`) {`
    Statement
    $Expr_3$ `;`
`}`

außer, dass bei einem `continue` in der Schleife auch $Expr_3$ noch ausgeführt wird.

Der Präprozessor

C weist eine Eigenheit auf, die bei höheren Programmiersprachen eher selten, bei Assemblern aber weit verbreitet ist: Mit Hilfe von Präprozessordirektiven können Makros definiert, andere Files inkludiert, und Teile eines Programms von der Kompilation ausgeschlossen werden.

Im Gegensatz zum eigentlichen Compiler arbeitet der Präprozessor zeilenorientiert, d.h. jede Präprozessordirektive muss in einer eigenen Zeile stehen. Alle Direktiven beginnen mit einem "`#`".

Portabilität Viele ältere C-Compiler erlauben keine Leerzeichen vor dem "`#`", d.h. "`#`" muss in der ersten Spalte stehen.

Inklusion von Files

Mit der Direktive `#include` können andere Files inkludiert werden. Dies wird vor allem dazu verwendet, um Konstanten und Deklarationen in einem zentralen File abzulegen, das dann von mehreren C-Files inkludiert wird. Der Filename kann entweder in doppelte Anführungszeichen "`" "`" oder spitze Klammern "<>" eingeschlossen sein. Anführungszeichen bedeuten ein vom Benutzer erstelltes File, spitze Klammern ein zum Compiler gehöriges. Der Filename kann auch durch Expansion eines Makros erzeugt werden.

Bedingte Kompilation

Mit Hilfe der `#if`, `#ifdef` und `#ifndef` Direktiven ist es möglich, bestimmte Teile eines Programms nur zu kompilieren, wenn eine bestimmte Bedingung erfüllt ist. Die Direktiven haben folgendes Format:

```
#if Ausdruck
[ C-Code ]
[ #elif Ausdruck ]*
[ C-Code ]
[ #else ]
[ C-Code ]
#endif
```

und

```
( #ifdef oder #ifndef ) Makroname
[ C-Code ]
[ #else ]
[ C-Code ]
#endif
```

Ausdruck ist ein konstanter C-Ausdruck, der in `long`-Arithmetik ausgewertet wird, wobei alle Operatoren außer sizeof und dem unären `&` erlaubt sind. Zusätzlich existiert ein Operator `defined`, der 1 liefert, wenn sein Argument ein definiertes Makro ist, sonst 0. Kommen im Ausdruck undefinierte Makros vor, wird stattdessen der Wert 0 angenommen. Der erste Block, für den der Ausdruck in der vorhergehenden `#if` oder `#elif` ungleich 0 war, wird kompiliert. Trifft das auf keinen Ausdruck zu, so wird der Block nach dem `#else` kompiliert.

`#ifdef` *Macroname* ist eine Abkürzung für `#if defined (`*Macroname*`)`, `#ifndef` *Macroname* für `#if ! defined (`*Macroname*`)`.

Portabilität `#elif` ist eine Erfindung des ANSI-Komittees.

Zeilennummern und Filename

Mit der `#line`-Direktive können Zeilennummer und Filename geändert werden. Dies ist praktisch, wenn das C-File von einem Programm (z.B. lex oder yacc) aus einem anderen File generiert wurde, und sich Fehlermeldungen des Compilers nicht auf das C-File sondern das ursprüngliche File beziehen sollen. Die Syntax lautet:

`#line` *Zeilennummer ["Filename"]*

wobei der Filename auch entfallen kann.

Pragmas

Pragmas sind Anweisungen für den Compiler. Ein Pragma besteht aus dem Keyword `#pragma` und beliebigen Tokens. Unbekannte Pragmas werden vom Compiler ignoriert.

Portabilität Pragmas wurden vom ANSI-Komittee eingeführt. Die Regel, dass unbekannte Pragmas ignoriert werden sollen, bietet nur wenig Schutz vor unerwünschten Seiten-

effekten, da gleiche Pragmas auf unterschiedlichen Compilern verschiedene Effekte haben können. Sie sollten daher immer in `#if...#endif` eingeschlossen werden.

Leere Direktive

Zeilen die nur ein "`#`" enthalten, werden ignoriert.

Makros

Mittels `#define` können Makros definiert werden. Es gibt zwei Arten von Makros, objektähnliche und funktionsähnliche. Sie unterscheiden sich dadurch, dass funktionsähnliche Makros Argumente haben. Die Definition hat folgendes Format

`#define` *`Macroname Ersatztext`*

für objektähnliche, und

`#define` *`Macroname`* `(`*`Parameterliste`*`)` *`Ersatztext`*

für funktionsähnliche Makros (kein Space zwischen *Macroname* und der öffnenden Klammer der Parameterliste). Die Parameterliste ist eine Liste von Identifiern, die durch Kommas getrennt sind. Jedes weitere Vorkommen des Makros im Programmtext wird dann durch *Ersatztext* ersetzt, wobei bei funktionsähnlichen Makros die Parameter durch die Argumente ersetzt werden. Wird während der Expansion eines Makros dasselbe Makro noch einmal erzeugt, wird es nicht mehr expandiert (es kann also zu keiner endlosen Rekursion kommen); außerdem werden expandierte Makros nicht mehr auf Präprozessordirektiven untersucht (es ist also nicht möglich, ein Makro zu definieren, das zu einem `#define` expandiert wird). Während der Expansion eines Makros erkennt der Präprozessor auch zwei zusätzliche Operatoren "`#`" und "`##`". Der erste ist ein unärer Operator, der aus einem Präprozessortoken einen String macht, der zweite ist ein binärer Operator, der seine beiden Operanden zu einem Token vereinigt.

Vordefinierte Konstanten

`__STDC__` hat bei Compilern, die dem ANSI-Standard entsprechen, den Wert 1.

`__DATE__` und `__TIME__` sind Strings, die Datum und Uhrzeit der Übersetzung enthalten.

`__FILE__` und `__LINE__` sind Strings, die Namen des Sourcefiles und die momentane Zeilennummer enthalten. Dies kann für Debuggingausgaben verwendet werden. Siehe auch `assert`.

3.3 Tücken im Umgang mit C

Die Programmiersprache C ist eine ausdrucksstarke und vielseitige Programmiersprache, welche dem Anwender viele Möglichkeiten offenlässt, um eine Programmieraufgabe zu lösen. Gerade diese Eigenschaften machten C zu einer äußert beliebten und erfolgreichen Programmiersprache, andererseits führen viele der in C angebotenen Möglichkeiten oft zu ungewollten Effekten. Aufgrund der der vielen Freiheiten, die C erlaubt, ist es oft schwer diese Programmierfehler vorzeitig zu entdecken und zu beheben.

Im Folgenden werden einige der häufigsten Ursachen für Programmierfehler im Umgang mit C diskutiert und Maßnahmen zur Vermeidung dieser Probleme vorgestellt. Der letzte Teil dieses Kapitels beschreibt dabei zusätzlich noch Probleme der Portabilität von C-Programmen, welche durch fehlende Festlegungen, bzw. bewusst offen gelassene Aspekte des ANSI-Standards auftreten.

3.3.1 Die Tücken des Präprozessors

Der Präprozessor kann auf vielfältige Weise eingesetzt werden, zum Beispiel zur Definition von Konstanten welche innerhalb eines Programms konsistent gehalten werden müssen, um gewisse Teile des Programms auszukommentieren oder für die Einsetzung von Makrofunktionen, welche teilweise die fehlende Funktionalität einer `inline`-Funktion nachbilden. Durch die Verwendung von Makros kann der C-Code auch komplett unleserlich gestaltet werden, dieses als *Obfuscation* bezeichnete Prinzip ist sogar Mittelpunkt eines alljährlichen Wettbewerbs.[14]

Textersetzung für Konstanten

Bei der Verwendung von Makros zur Definition von Konstanten ist zu beachten, dass bei der Auflösung der Makros durch den Präprozessor keine numerische Auswertung sondern nur eine reine Textersetzung stattfindet. Das folgende Codestück liefert daher nicht dass gewünschte Ergebnis.

```
#define STRINGLEN 50+1

 /* Speicherverbrauch für 10 Strings */
 m = STRINGLEN * 10;
```

Nach dem Durchlauf des Präprozessors steht im Ausdruck `50+1 * 10`, was dann zu einem Ergebnis von 60 anstelle von 510 führt. Dieses Problem wird umgangen, indem der Ausdruck in der Makrodefinition mit einer Klammer versehen wird:

```
#define STRINGLEN (50+1)
```

Makros und Seiteneffekte

Ein ähnliches Problem tritt bei der unachtsamen Definition von Makrofunktionen auf:

```
#define SQUARE(x) x*x

 a = SQUARE(3+4);
 b = SQUARE(a++);
```

Nach dem Durchlauf des Präprozessors steht in den Ausdrücken `a = 3+4*3+4;` und `b = a++*a++;` was zu einem falschem Ergebnis bzw. im zweiten Fall sogar zu einer doppelten Inkremenierung der Variable a führt.

Das korrekte Ergebnis für die Variable a kann auch hier durch eine entsprechende Klammernsetzung erreicht werden:

[14] The International Obfuscated C Code Contest. `http://www.ioccc.org/`

```
#define SQUARE(x) ((x)*(x))
```

Die ungewünschten Seiteneffekte bei Inkrementations- und Dekrementationsoperatoren bleiben jedoch problematisch. Hier gilt es, entweder das Makro so zu schreiben, dass jedes Argument nur einmal ausgewertet wird, oder, wenn dies nicht möglich ist, muss auf die Verwendung von Argumenten mit Seiteneffekten in Makros verzichtet werden.

Mit der Einführung des C99 Standards [C99] besteht auch die Möglichkeit, eine Funktion durch Angabe des Schlüsselworts `inline` als Funktion mit Inline-Expansion zu definieren, was einen effizienten Funktionsaufruf ohne Probleme mit Seiteneffekten ermöglicht.

Makros anstelle von Typdefinitionen

Werden Makros anstelle von Typdefinitionen verwendet, so kann das ebenfalls zu unerwünschten Effekten führen:

```
#define POINTER struct foo *
POINTER x,y;
```

Nur die zuerst deklarierte Variable `x` ist hier wirklich vom Typ `foo *` die Variable `y` ist hingegen vom Typ `foo`. Um solche Probleme zu vermeiden, ist es besser, die `typedef`-Anweisung zu verwenden, um beide deklarierten Variablen mit dem Typ `foo *` zu erhalten.

```
typedef struct foo *POINTER
POINTER x,y;
```

3.3.2 Lexikalische Fehlerquellen

Als erster Teil des Kompilierungsprozesses wird eine lexikalische Analyse durchgeführt. Diese betrachtet die Reihenfolge der Zeichen, die das Programm bilden und unterteilt sie in so genannte Token. Ein Token ist eine Reihenfolge von einem oder mehr Zeichen, die eine (verhältnismäßig) konstante Bedeutung in der Sprache haben, die kompiliert wird. Gewisse Programmkonstrukte können aber durch unterschiedliche Interpretation der Token ein anderes Ergebnis als das vom Programmierer beabsichtigt ergeben.

= ist nicht gleich ==

C verwendet = zur Zuweisung von Werten, die Vergleichsoperation hingegen wird mittels == durchgeführt. Andere Programmiersprachen wie Pascal und Ada verwenden hingegen := für die Zuweisung und = als Vergleichsoperator. Außerdem ist die Zuweisung in C gleichzeitig auch ein Operator, welcher den zugewiesenen Wert zurückgibt und somit Mehrfachzuweisungen wie `a=b=c` ermöglicht.

Somit kann die Zuweisung in C aber auch in einem logischen Ausdruck verwendet werden, ohne dass eine syntaktische Verletzung des Standards auftritt. Die folgende if-Abfrage ist lexikalisch korrekt, aufgrund der Zuweisung ist die Bedingung aber immer wahr:

```
if (a=1) foo();
```

Viele Compiler warnen den Programmierer, wenn eine Zuweisung in einer Bedingung eingesetzt wird. Diese Warnungen sollten nicht ignoriert werden. Will man tatsächlich eine Zuweisung in einer Bedingung effizient einsetzen, so sollte man den folgenden Vergleich explizit machen, um die Compilerwarnung zu vermeiden.

Statt

```
if (a=b) foo();
```

sollte man somit

```
if ((a=b)!= 0) foo();
```

verwenden.

Logische versus bitweise Operatoren

Ein anderes Problem liegt in der Verwechslung der Operatoren `&&`, `||`, `!` mit `&`, `|`, `~`. Die ersten unterscheiden bei ihren Argumenten hierbei nur zwischen 0 und ungleich 0, wohingegen die einfach geschriebenen Operatoren `&`, `|` und `~` bitweise arbeiten. Das folgende Listing zeigt den semantischen Unterschied anhand eines Beispiels auf:

```
a=1;
b=2;

if (a && b) foo();
if (a & b) bar();
```

Die Funktion `foo()` wird aufgerufen, die Funktion `bar` hingegen nicht, denn die bitweise Und-Verknüpfung von a und b ergibt 0.

Tokenbildung

Die Anweisung

```
x= y---z;
```

ist lexikalisch korrekt, jedoch ist den meisten Betrachtern nicht klar, ob hier y postdekrementiert oder z präinkrementiert werden soll:

```
x= y-- - z; /* erste Interpretationsmöglichkeit */
x= y - --z; /* zweite Interpretationsmöglichkeit */
```

Hierbei geht der Compiler nach folgenden Regeln vor:

- es wird immer zuerst die Interpretation mit dem größten zusammenhängenden Token gewählt
- bei mehreren Interpretationsmöglichkeiten von Token gleicher Länge wird von links nach rechts vorgegangen.

Daher wird ein Standard C-Compiler hier die erste Interpretation wählen, jedoch ist nicht klar ob das auch die Intention des Programmierers war. Zur Vermeidung solcher Mehrdeutigkeiten sollten daher entsprechende Leerzeichen und Klammern in Ausdrücken gesetzt werden.

Unechte Kommentarzeichen

Das folgende Codebeispiel zeigt ein weiteres Problem, das bei der Interpretation von mehrzeichigen Token auftreten kann zeigt.

```
y = x/*p; /* p ist ein Zeiger auf den Divisor */
```

In diesem Fall wird das erste $/*$ im Code als Beginn eines Kommentars interpretiert. Wahrscheinlich erhält man beim Kompilieren hier eine Fehlermeldung die auf das fehlende Semikolon zurückgeht, da dieses ja lexikalisch innerhalb eines Kommentarblocks liegt.

Dieses Problem lässt sich dadurch vermeiden, indem zwischen den einzelnen Token ein Abstand gesetzt wird:

```
y = x / *p; /* p ist ein Zeiger auf den Divisor */
```

Stringkonstanten

Strings werden mittels doppelten Anführungszeichen ” deklariert, Zeichen hingegen mittels ’. Ein einzelnes Zeichen in einfachen Anführungszeichen entspricht einer Ganzzahl, welche den entsprechenden Zeichencode (prinzipiell systemabhängig, aber zumeist in ASCII-Kodierung) enthält. Viele Compiler unterstützen auch die Angabe von mehreren Zeichen in einfachen Anführungszeichen, welche dann einen Integer ergeben, der byteweise aus den Zeichencodes der angegebenen Zeichen zusammengesetzt ist.

Die Angabe einer Zeichenkette in doppelten Anführungszeichen definiert hingegen einen Adresspointer vom Typ `char *` welcher auf eine Stringkonstante mit dem angegebenen Inhalt einschließlich eines abschließenden Nullzeichens zeigt.

Wird von einer Funktion eine Zeichenkette erwartet, so ist diese immer mit doppelten Anführungszeichen anzugeben, auch wenn die zu übergebende Zeichenkette nur aus einem Zeichen besteht.

```
printf('\n'); /* falsch, führt möglicherweise zu einem Segmentation Fault */
printf("\n"); /* korrekt, es wird ein Zeiger auf "\n\0" übergeben */
```

Bei der Definition eines Strings über `char *` ist zu beachten, dass nur ein Pointer erzeugt wird, aber kein Speicherbereich für den reserviert wird. Die folgenden Anweisungen führen daher zu einem Laufzeitfehler:

```
char *s;

if (strcmp(s,"foobar") != 0) ...
strcpy(s,"ABCDEF");
```

Nach der Definition ist s undefiniert. Die `strcmp`-Funktion würde daher versuchen, diesem undefinierten Zeiger zu folgen, was zu einen Lesezugriff auf einen nicht vorher festgelegten Speicherbereich führt und wahrscheinlich eine Speicherschutzverletzung zur Folge hat. Der Aufruf der `strcpy`-Funktion führt sogar einen Schreibzugriff auf einen nicht festgelegten Speicherbereich aus.

Korrekterweise muss daher der Zeiger auf einen definierten Speicherbereich gesetzt werden:

```
char *s,*t,*u;

s=NULL;                      /* s wird mit dem NULL-Zeiger initialisiert      */
t="ABCDEF";                  /* s zeigt jetzt auf eine Stringkonstante        */
u=malloc(100*sizeof(char)); /* für u 100 Bytes Speicher reservieren           */
```

Integerkonstanten

Integerkonstanten können in C auf unterschiedliche Arten (Dezimal, Zeichencode, Oktal, Hexadezimal) angegeben werden, was die Lesbarkeit des Quellcodes unterstützt. Dezimalkonstante dürfen aber nie mit einer führenden Null geschrieben werden, da sie sonst als Oktalzahl interpretiert werden. Kommen in der Zahl die Ziffern 8 und 9 nicht vor, so bleibt der Fehler vom Compiler unbemerkt:

```
printf("Weihnachten ist am %d.%d",024,012);
```

ergibt nicht wie gewünscht den 24.12., sondern verlegt das frohe Fest auf den 20. Oktober.

Lesbarkeit

Eindeutige, aber grafisch schwer zu unterscheidende Zeichen sollten bei der Benennung von Variablen und Konstanten vermieden werde. Insbesondere gilt das für das große "`O`"welches leicht mit der Ziffer "`0`"verwechselt werden kann und das kleine "`l`", welches leicht mit der Ziffer "`1`"verwechselt werden kann. Als Modifier für den Typ `long` sollte immer das große "`L`"angegeben werden.

3.3.3 Syntaktische Fehlerquellen

Syntaktische Fehler betreffen Missinterpretationen bei den Definitionen, Deklarationen, Ausdrücken und Aussagen eines Programms.

Deklarationen

Die Programmiersprache C erlaubt es, eine Deklaration auf unterschiedliche Art und Weise auszudrücken, was oft zu schwer verständlichen Konstrukten führt. Andererseits können fast alle Deklarationen so geschrieben werden, wie die Variable auch verwendet wird. Der Lesbarkeit wegen sollte daher eine Deklaration so geschrieben werden, wie die Variable, Struktur oder Funktion auch verwendet wird!

Elemente in Deklarationen werden nach einer vorgegebenen Reihenfolge aufgelöst (siehe S. 150 zu den Prioritäten der Operatoren). Fehlende Kenntnis dieser Operatorenpriorität führen allerdings oft zu Programmierfehlern in Deklarationen.

Die folgenden beiden Deklarationen haben zum Beispiel unterschiedliche Bedeutung:

```
int *g();  /* eine Funktion, die einen Pointer auf Integer zurückgibt     */
int(*h)(); /* ein Pointer auf eine Funktion, die einen Integer zurückgibt */
```

Vorrangregeln bei Operatoren

Die Auswertungsreihenfolge der Operatoren ist nicht immer intuitiv wie das folgende Beispiel zeigt. Die folgende Anweisung dient dazu, das Ergebnis einer bitweisen Und-Verknüpfung auf ein Ergebnis ungleich 0 zu prüfen:

```
if (MASK & x) foo();
```

Um den Code lesbarer zu gestalten, wird nun die Abfrage auf ein Ergebnis ungleich Null explizit formuliert, was aber zu einem unterschiedlichen Programmverhalten führt:

```
if (MASK & x != 0) foo();
```

Der Operator `!=` bindet nämlich stärker als das bitweise Und. Darum wird hier zuerst überprüft, ob x ungleich 0 ist und das Ergebnis (0 oder 1) dann bitweise mit MASK Und-verknüpft. Dieser Fehler ist schwer zu entdecken, da er sich in manchen Fällen (z.B. für `x=0` oder `x=1`) nicht manifestiert. Die korrekte Umsetzung erfordert eine zusätzliche Klammerung wie im folgenden Beispiel angegeben:

```
if ((MASK & x) != 0) foo();
```

Die Auswertungsreihenfolge von Shift-Operatoren kann zu fehlerhaften Code führen, wenn übersehen wird, dass die Shift-Operatoren schwächer binden als `+`, `-`, `*` und `/`:

```
result = bit2 * 1<<1 + bit1;   /* wird als (bit2*1) << (2+bit1) interpretiert
   */
result = bit2 * (1<<1) + bit1; /* liefert gewünschtes Ergebnis */
```

Überflüssige und fehlende Strichpunkte

Ein zusätzliches Semikolon hat meist wenig Einfluss auf das Programmverhalten, da es wie eine Leeranweisung behandelt wird. An der falschen Stelle gesetzt oder ausgelassen kann es jedoch den gesamten Programmablauf ändern, wie in folgendem Codebeispiel zu sehen ist:

```
if (x[i] > big);
  big = x[i];
```

In diesem Fall wird die “if” Operation als abgeschlossen betrachtet, die zweite Zeile wird unabhängig von Resultat der Abfrage ausgeführt. Soll die Ausführung der zweiten Zeile vom Ergebnis der Abfrage abhängen, wie es die Einrückung suggeriert, so muss das Semikolon weggelassen werden. Zur besseren Übersicht wäre zudem noch eine Klammerung des if-Ausdrucks zu empfehlen:

```
if (x[i] > big) {
  big = x[i];
}
```

Das folgende Beispiel (aus [Koe89]) zeigt einen unerwünschten Effekt durch das Weglassen eines Strichpukts nach einer Deklaration:

```
struct foo {
  ...
}

f()
{

}
```

Da die Funktion `f` ohne Rückgabewert deklariert wurde (hier sollte besser `void f()` stehen), interpretiert der Compiler die Struktur `foo` als den Typ des Rückgabewerts der Funktion.

Tücken bei switch

Ein vergessenes `break` in einem `switch`-Statement wird vom Compiler nicht erkannt, da das Weglassen von `break` auch bewusst als Fallthrough-Mechanismus zum effizienten Lenken des Programmablaufs verwendet werden kann. Um solche Fehler zu vermeiden, sollte jedes `case`-Statement mit einem `break` abgeschlossen werden oder mit einem Kommentar, welcher auf den bewussten Verzicht von `break` hinweist.

Dangling else Problem

Besteht der Block, der nach einer `if` Anweisung ausgeführt werden soll, nur aus einem Befehl, so kann die Klammerung weggelassen werden. Bei verschachtelten if-Anweisungen führt dies jedoch zu einem Interpretationsproblem für einen menschlichen Betrachter.

```
if (a == b)
  if (c == 0) foo();
else {
  a = b + c;
  bar();
}
```

Hier ist unklar, zu welchem `if` die `else` Anweisung im oberen Fall wirklich gehört. Tatsächlich gehört das `else` immer zum direkt davorliegenden `if`, jedoch suggeriert die Einrückung in obigem Beispiel etwas anderes.

Durch eine geeignet gesetzte Klammerung kann die Funktion übersichtlicher und eindeutiger gestaltet werden:

```
if (a == b) {
```

```
  if (c == 0) foo();
}
else {
  a = b + c;
  bar();
}
```

3.3.4 Semantische Fehlerquellen

Ein im Sinne der Sprachspezifikation korrekt geschriebenes Programm kann dennoch seinen Zweck für den es implementiert worden ist, verfehlen. Im Folgenden werden typische semantische Fehler, welche bei der C-Programmierung auftreten, vorgestellt.

Auswertungsreihenfolge

Die Reihenfolge, in der Argumente innerhalb eines Ausdrucks abgearbeitet werden ist nur für die Operatoren `&&`, `||`, die bedingte Auswertung und den Komma-Operator eindeutig definiert; jeweils das linke Argument wird zuerst ausgewertet; sobald das Ergebnis des Gesamtausdrucks durch den ausgewerteten Teilausdruck definiert ist, werden die weiteren Argumente nicht mehr ausgewertet.

Bei allen anderen Operatoren darf der Compiler die Auswertereihenfolge beliebig wählen.

Die folgende Schleife zum Kopieren eines Arrays

```
while (i<n)
  y[i] = x[i++];
```

kann daher je nach Compilerlaune funktionieren oder auch nicht. Um sicherzugehen, sollte die Inkrementierung der Zählervariable in einer eigenen Anweisung erfolgen.

Pointer versus Strings

Strings werden in C als Pointer auf ein Array von Characters dargestellt. Der Pointer bildet dabei die Handhabe um auf den String zugreifen zu können, tatsächlich enthält er aber eine Adresse, die auf den String zeigt, nicht den String selbst.

Ein Vergleich zwischen zwei Pointern anstatt eines Stringvergleichs liefert daher nur eine Aussage über die (Un-)gleichheit zweier Pointer, die Gleichheit des Inhalts muss mit der Funktion `strcmp` überprüft werden.

```
char *p = "foo";

if ( p == "foo") ...            /* falsch */
if ( strcmp (p,"foo") == 0) ... /* korrekt */
```

Derselbe Umstand ist beim Kopieren eines Strings zu beachten. Die folgende Befehle erzeugen zwei Pointer auf denselben Speicherbereich, anstatt einer Kopie des Strings (diese wäre mittels `strcpy` zu erzeugen).

```
char *p, *q;
p = "foo";
q = p;
```

3.3.5 Portabilität von C-Programmen

Um ein C-Programm unabhängig von Plattform und Compiler zu erstellen sind die folgenden plattformabhängigen Eigenschaften zu beachten.

Typgrößen und Überlauf

Um auf unterschiedlichen Plattformen effizient zu sein, sind gewisse Eigenschaften, wie zum Beispiel die Größe der Datentypen nicht eindeutig im ANSI C Standard festgelegt.

So gilt für die Ganzzahltypen, dass der Typ `short` weniger oder gleich viele Bits hat wie der normale Integer und dieser wiederum weniger oder gleich viele Bits wie `long`. ANSI C definiert ein Minimum von 16 Bits für `short` und normalen Integer und ein Minimum von 32 Bit für den `long` Typ.

Der Datentyp `char` muss mindestens 7 Bit besitzen, die meisten Compiler realisieren Character als 8-Bit Integer. Leider ist dabei nicht definiert, ob Characters als vorzeichenbehaftete oder vorzeichenlose Ganzzahl interpretiert werden.

Auf manchen Plattformen wie zum Beispiel dem avr-gcc Compiler hat des weiteren der Datentyp `double` so wie `float` nur 32 Bit.

Um plattformunabhängig zu bleiben, sollte der Ganzzahlüberlauf vermieden werden und für jeden Datentyp nur die minimal garantierte Wortbreite angenommen werden.

Rechtsverschiebungsoperator

Beim Shift nach rechts auf vorzeichenbehafteten Datentypen werden die hereingeschobenen Bits compilerabhängig entweder mit einer Kopie des Vorzeichenbits oder mit 0 aufgefüllt.

Um diese Mehrdeutigkeit zu verhindern, sollte der Rechtsverschiebungsoperator nur auf vorzeichenlose Datentypen angewandt werden, bzw. vorzeichenbehaftete Datentypen auf vorzeichenlose konvertiert werden.

Ganzzahldivision und Rest

Bei einer Ganzzahldivision erhält man im Allgemeinen einen gerundeten Quotienten `q` und einen Rest `r`, welche sich wie folgt errechnen:

```
int divisor, dividend;

q = divisor / dividend;
r = divisor % dividend;
```

Wir betrachten nun die folgenden Eigenschaften:

1. `q*dividend + r == divisor`
2. Der Rest sollte immer größer gleich 0 und kleiner als der Dividend sein.
3. Es sollte gelten `(-divisor)/dividend == -(divisor/dividend)`

Während Eigenschaft 1 immer gilt, schließen sich Eigenschaft 2 und 3 bei Verwendung von negativen Zahlen gegenseitig aus. Sei zum Beispiel der Divisor -3 und der Dividend 2, dann gilt entweder `q=-2, r=1` (Eigenschaft 2 erfüllt) oder `q=-1, r=-1` (Eigenschaft 3 erfüllt). Die meisten Compiler geben die zweite Eigenschaft auf und garantieren das Assoziativprinzip der Negation. Ein Compiler der die Eigenschaft 3 aufgibt und dafür einen positiven Rest für positive Dividenden garantiert, entspricht aber ebenso dem ANSI-Standard.

Um die Kompatibilität von C-Programmen zu erhalten, sollte sich ein Programmierer daher weder auf Eigenschaft 2 noch auf Eigenschaft 3 verlassen.

3.4 Die C-Library

Zur Sprache C gehört auch eine umfangreiche Library. In diesem Kapitel sollen die wichtigsten Funktionen kurz vorgestellt werden.

3.4.1 errno.h

Deklariert die globale Variable `errno`, die von verschiedenen Funktionen im Fehlerfall gesetzt wird (siehe auch Kapitel 3.4.15), sowie alle Werte, die diese annehmen kann, als Konstante.

3.4.2 assert.h

```
void assert(int condition)
```

Wenn `condition` nicht zutrifft, gibt `assert` eine Fehlermeldung aus, die *expression*, Name des Source-Files und Zeilennummer enthält, und veranlasst einen Programmabbruch mittels `abort`. Es kann dazu verwendet werden, um Annahmen über das Verhalten des Programms zu dokumentieren die zur Laufzeit überprüft werden sollen (jedoch: siehe Notiz unten). Damit können Fehler, die sonst erst viel später zu falschen Ergebnissen führen, leichter eingegrenzt und “unmögliche” Fehler gefunden werden.

Achtung Ist bei der Inklusion von `<assert.h>` das Makro `NDEBUG` definiert, so macht assert nichts. Das Argument von `assert` darf daher keine Seiteneffekte haben.

Beispiele

```
switch(one_or_two)
{
    case 1:
        do_something ();
        break;
    case 2:
        do_something_else ();
        break;
    default:
        assert (0);
}
```

zeigt, dass im angegebenen Beispiel der default-Zweig der Schleife nie erreicht werden soll. Wenn es doch passieren sollte (auf Grund eines Programmierfehlers), so bricht das Programm ab.

3.4.3 ctype.h

Enthält diverse Funktionen zum Testen von Attributen von Zeichen und zum Umwandeln von Zeichen. Alle Testfunktionen liefern einen Wert $\neq 0$, wenn das Attribut des Tests zutrifft, sonst 0.

`int isalnum (int c)` testet, ob `c` ein Buchstabe oder eine Ziffer ist.

`int isalpha (int c)` testet, ob `c` ein Buchstabe ist.

`int iscntrl (int c)` testet, ob `c` ein Steuerzeichen ist.

`int isdigit (int c)` testet, ob `c` eine Ziffer ist.

`int isgraph (int c)` testet, ob `c` ein sichtbares druckbares Zeichen ist (also kein Steuer- oder Leerzeichen).

`int islower (int c)` testet, ob `c` ein Kleinbuchstabe ist.

`int isprint (int c)` testet, ob `c` ein druckbares Zeichen ist (inklusive Space).

`int ispunct (int c)` testet, ob `c` ein Satzzeichen ist.

`int isspace (int c)` testet, ob `c` ein Leerzeichen ist.

`int isupper (int c)` testet, ob `c` ein Großbuchstabe ist.

`int isxdigit (int c)` testet, ob `c` eine Hexadezimalziffer ist.

`int toupper (int c)` Wenn `c` ein Kleinbuchstabe ist, wird der entsprechende Großbuchstabe zurückgeliefert, sonst `c`.

`int tolower (int c)` Wenn `c` ein Großbuchstabe ist, wird der entsprechende Kleinbuchstabe zurückgeliefert, sonst `c`

Diese Funktionen hängen von der aktuellen *Locale* (siehe Kapitel 3.4.4) ab. In der "C"-Lokale sind folgende Bedingungen erfüllt:

`islower (c)` : `c` ist aus `a...z`

`isupper (c)` : `c` ist aus `A...Z`

`isspace (c)` : `c` ist aus `' ','\f','\n','\r','\t','\v'`

```
isprint (c) : isgraph (c) || c == ' '
```

```
isalpha (c) : islower (c) || isupper (c)
```

```
ispunct (c) : isprint (c) && ! isalnum (c)
```

Andere Locales können von diesen Regeln abweichen. Z.B. könnte (und sollte) eine deutsche Locale Umlaute als Buchstaben erkennen.

3.4.4 locale.h

Die *Locale* bestimmt lokale Eigenschaften des Environments. Dazu gehören der Zeichensatz (z.B. sind Umlaute Buchstaben?), die Sortierreihenfolge (z.B. sollen Groß- und Kleinbuchstaben unterschiedlich sortiert werden? Wie werden Umlaute sortiert? . . .) und das Zahlenformat (Wird ein Punkt oder ein Komma als Dezimalkomma verwendet?). All diese Kategorien beeinflussen nur das Verhalten des Programms zur Laufzeit, nicht seine Kompilation. Beim Start ist jedes Programm in der `"C"`-Locale, andere Locales müssen explizit gesetzt werden.

`char * setlocale (int category, const char *locale);` Dient zum Setzen einer Locale (oder einzelne der oben angeführten Kategorien). Da Locales (außer "C") nicht von C definiert sind, sei hier auf Compiler- und Betriebssystemdokumentation verwiesen.

`struct lconv *localeconv (void);` Dient zum Abfragen gewisser Eigenschaften der aktuellen Locale. Auch hier sei auf Compiler- und Betriebssystemdokumentation verwiesen.

3.4.5 math.h

Deklariert ein Makro `HUGE_VAL`, das den größten positiven darstellbaren `double`-Wert (möglicherweise $+\infty$) darstellt und diverse mathematische Funktionen. Alle diese Funktionen setzen `errno` auf `ERANGE`, wenn das Ergebnis nicht als `double` darstellbar ist, bzw. auf `EDOM`, wenn die Funktion auf dem Eingabewert nicht definiert ist.

Trigonometrische Funktionen

Alle Winkel werden in Radiant angegeben.

`double acos (double x);` Arcus cosinus

`double asin (double x);` Arcus sinus

`double atan (double x);` Arcus tangens

`double atan2 (double y, double x);` Winkel des Strahls, der von (0, 0) durch (x, y) geht.

`double cos (double x);` Cosinus

`double sin (double x);` Sinus

`double tan (double x);` Tangens

Hyperbolische Funktionen

`double cosh (double x);` Cosinus hyperbolicus

`double sinh (double x);` Sinus hyperbolicus

`double tanh (double x);` Tangens hyperbolicus

Logarithmische und Exponentialfunktionen

`double exp (double x);` e^x

`double frexp (double value, int *exp);` Zerlegt `value` in eine Mantisse (im Bereich $[\frac{1}{2}, 1)$) und einen Exponenten zur Basis 2. Wenn jedoch `value` den Wert 0 hat, sind sowohl die Mantisse als auch der Exponent 0. `frexp` liefert den Wert der Mantisse als Funktionswert und speichert den Exponenten in den `int`, auf den `exp` zeigt.

`double ldexp (double x, int exp);` $\mathtt{x} \cdot 2^{\mathtt{exp}}$

`double log (double x);` Natürlicher Logarithmus.

`double log10 (double x);` Logarithmus zur Basis 10.

`double modf (double value, double *iptr);` Zerlegt `value` in einen ganzzahligen und einen Nachkommaanteil mit gleichem Vorzeichen. Liefert den Nachkommaanteil als Return-Wert und speichert den ganzzahligen Anteil in den `double`, auf den `iptr` zeigt.

Potenzfunktionen

`double pow (double x, double y);` x^y

`double sqrt (double x);` $\sqrt{x}$

Diverses

`double ceil (double x);` Liefert den kleinsten ganzzahligen Wert der nicht kleiner als `x` ist.

`double fabs (double x);` Liefert den Absolutbetrag von `x`.

`double floor (double x);` Liefert den größten ganzzahligen Wert der nicht größer als `x` ist.

`double fmod (double x, double y);` Liefert den Rest der Division `x/y`. Für den Return-Wert $r = x - iy$ gilt: $i \in \mathbf{Z}$, $\mathtt{r} < \mathtt{y}$ und $rx \geq 0$.

3.4.6 setjmp.h

Definiert den Typ `jmp_buf`, das Makro `int setjmp (jmp_buf env)` und die Funktion `void longjmp (jmp_buf env, int val)`. Diese können für Sprünge aus einer Funktion hinaus verwendet werden.

Faustregel: Nicht verwenden.

Faustregel für Experten: Nur im Notfall verwenden.

3.4.7 signal.h

`sig_atomic_t` ist ein ganzzahliger Typ auf den *atomic* zugegriffen werden kann (und wird). Globale Variablen, die von einer Signalbehandlungsroutine manipuliert werden, sollten diesen Typ haben und `volatile` sein (da sich der Wert einer solchen Variable ändern kann, ohne dass der Compiler dies aufgrund des lokalen Programmflusses erkennen kann).

`void (*signal (int sig, void (* func)(int)))(int);` (siehe auch Seite 149) installiert eine Funktion `func`. Diese Funktion wird aufgerufen, wenn das Signal `sig` eintrifft. Wird `SIG_IGN` als zweiter Parameter übergeben, so wird das Signal ignoriert; `SIG_DFL` stellt das default-Verhalten wieder her. Folgende Signale sind unabhängig vom Betriebssystem definiert:

`SIGABRT` Interner Programmabbruch, wie von `abort` ausgelöst.

`SIGFPE` Arithmetische Exception (z.B. Division durch 0)

`SIGILL` Illegale Instruktion.

`SIGINT` Programmabbruch durch den Benutzer.

`SIGSEGV` Illegaler Speicherzugriff.

`SIGTERM` Externer Programmabbruch.

Wenn `signal` erfolgreich ist, so wird der zuletzt gesetzte Signal-Handler zurückgeliefert, sonst `SIG_ERR`. Im Fehlerfall wird errno gesetzt. Für Beispiele zur Verwendung siehe Kapitel 4.4.5.

Bei Eintreffen eines Signals, für das ein Handler registriert wurde, wird, abhängig vom Betriebssystem, entweder der Handler für dieses Signal auf `SIG_DFL` zurückgestellt, oder ein weiteres Auftreten dieses Signals für die Dauer der Handlerfunktion unterbunden. Dann wird die Handlerfunktion mit einem Parameter (der Nummer des Signals) aufgerufen. Wurde das Signal nicht durch `raise` oder `abort` ausgelöst, so führt die Verwendung von Libraryfunktionen (außer `signal` zum erneuten Setzen einer Handlerfunktion für das behandelte Signal) und die Verwendung statischer Variablen, die nicht vom Typ `volatile sig_atomic_t` sind, zu undefiniertem Verhalten.[15]

`int raise (int sig);` sendet das Signal `sig` an den aufrufenden Prozess.

Im Fehlerfall wird ein Wert ungleich 0 zurückgeliefert, sonst 0.

3.4.8 stdarg.h

Makros zum Verarbeiten von Argumentlisten variabler Länge.

`void va_start (va_list ap, parmN);` setzt `ap` auf das erste Argument hinter *parmN*, wobei *parmN* der Name des letzten bekannten Arguments der Funktion sein muss.

type `va_arg (va_list ap, type);` liefert das nächste Argument, das vom Typ *type* sein muss und setzt `ap` auf das Folgende.

`void va_end (va_list ap);` muss vor einem `return` aufgerufen werden, wenn vorher `va_start` aufgerufen wurde.

Beispiel zur Ein-/Ausgabe

Die folgende Funktion entspricht `printf`, schreibt aber auf `stderr` statt auf `stdout`:

```
int eprintf (const char *fmt, ...)
{
    va_list ap;
    int     rc;

    va_start (ap, fmt);
    rc = vfprintf (stderr, fmt, ap);
```

[15] Der ANSI-Standard garantiert also nur, dass es möglich ist, aus einem Signalhandler heraus ein Flag zu setzen. In der Praxis ist die Verwendung von Libraryfunktionen meist möglich, obwohl sich in der Dokumentation selten Hinweise darauf finden, ob ein korrektes Funktionieren der Funktionen gewährleistet ist.

```
    va_end (ap);
    return rc;
}
```

Die folgende Funktion stellt eine stark vereinfachte Version von `printf` dar:

```
void simple_printf (const char *fmt, ...)
{
    va_list  ap;
    int      i;
    char     *s;
    char     *p;

    va_start (ap, fmt);
    for (p = fmt; *p; p ++)
    {
        if (p == '%')
        {
            switch (* ++ p)
            {
                case 'd':
                    ... print_decimal (va_arg (ap, int)) ...
                    break;
                case 'c':
                    /* int, not char! */
                    ... putchar (va_arg (ap, int)) ...
                    break;
                case 's':
                    for (s = va_arg (ap, char *); * s; s ++)
                    {
                        ... putchar (*s) ...
                    }
                    break;
                default:
                    ... putchar (*p) ...
            }
        }
        else
        {
            ... putchar (*p) ...
        }
    }
    va_end (ap);
}
```

3.4.9 stdio.h

Streams und Files

Ein *Stream* ist eine sequentielle Folge von Zeichen. Er kann mit einem Ein- oder Ausgabegerät oder einem *File* verbunden sein. Auf jedem Stream kann man entweder sequentiell Zeichen schreiben, oder sequentiell Zeichen lesen. Auf Streams, die mit Files (oder Geräten die direkten Zugriff erlauben) verbunden sind, kann außerdem der Schreib-Lesezeiger auf eine andere Stelle im File positioniert werden.

Es existieren zwei Arten von Streams, Text-Streams und Binäre Streams. Text-Streams bestehen aus Zeilen, die durch `'\n'` getrennt sind. Sind in der externen Repräsentation Zeilen nicht durch ein einzelnes Zeichen getrennt, so führen die Ein- und Ausgabefunktionen entsprechende Umwandlungen durch. Auf binären Streams hingegen werden keine Umwandlungen durchgeführt.

Streams sind üblicherweise gepuffert, d.h. Zeichen, die mit den stdio-Funktionen geschrieben werden, werden nicht unbedingt sofort auf das entsprechende File oder Ausgabemedium geschrieben, und Aufrufe von Lesefunktionen können große Blöcke in interne Puffer lesen, aus denen weitere Lesefunktionen bedient werden. Es gibt 3 Arten von Pufferung:

Keine Pufferung Jede Schreiboperation wird sofort ausgeführt. Leseoperationen fordern vom Betriebssystem nur soviele Zeichen wie notwendig an.

Zeilenpufferung Schreiboperationen werden ausgeführt, sobald ein `'\n'` geschrieben wird, oder von einem nicht voll gepufferten Stream gelesen wird[16].

Volle Pufferung Schreib- und Leseoperationen werden nur wenn notwendig ausgeführt (z.B. wenn der Puffer voll ist).

Außerdem werden Schreiboperationen immer dann ausgeführt, wenn der Puffer voll ist, bevor eine Leseoperation auf einen gepufferten Stream ausgeführt wird und wenn der Stream geschlossen wird. Leseoperationen werden immer dann ausgeführt, wenn der Eingabepuffer leer ist.

Die Pufferung von Ein- und Ausgabestreams kann zu drei Problemen führen:

- Die Ausgabe wird verzögert. Das kann dazu führen, dass Ausgaben zu unerwarteten Zeitpunkten (z.B. am Ende des Programms) oder überhaupt nicht (z.B. weil das Programm inzwischen abgestürzt ist — Vorsicht bei Debugging-Ausgaben) erfolgen.
- Schreiboperationen werden in mehrere Portionen aufgespalten. Wenn mehrere Programme auf das selbe File schreiben, kann deren Ausgabe vermischt werden.
- Ein Programm kann mehr lesen als erwünscht. Dies tritt vor allem auf, wenn ein Programm andere Programme aufruft, um Teile seiner Eingabe zu verarbeiten.

[16] Bei einem interaktiven Programm wird `stdout` also immer geflusht, wenn von `stdin` gelesen wird. Ein `fflush` nach einem Prompt ist also nicht notwendig.

Die ersten beiden Probleme können leicht durch `fflush` und Puffer entsprechender Größe vermieden werden, für das dritte gibt es keine gute Lösung (es tritt allerdings auch selten auf).

Drei Streams sind beim Start jedes Programmes verfügbar: `stdin`, `stdout` und `stderr` (siehe auch Kapitel 3.1.7). Der Stream `stderr` ist nicht voll gepuffert (im Allgemeinen ungepuffert). Die Streams `stdin` und `stdout` sind genau dann voll gepuffert, wenn sie mit keinem interaktiven Gerät (z.B. Terminal) verbunden sind, sonst sind sie im Allgemeinen zeilengepuffert.

Zusätzliche Streams können durch Öffnen von Files erzeugt werden. Files werden über einen Namen (der als `char [FILENAME_MAX]` darstellbar ist) identifiziert.

Operationen mit Files

`int remove (const char *filename);` Löscht ein File. Liefert einen Wert $\neq$ (!) 0, wenn ein Fehler auftritt.

`int rename (const char *old, const char *new);` Benennt ein File um. Liefert einen Wert $\neq$ (!) 0, wenn ein Fehler auftritt. In diesem Fall existiert das File noch unter seinem ursprünglichen Namen.

`FILE *tmpfile (void);` Erzeugt ein temporäres File und öffnet es im `"wb+"` Modus. Wenn das File geschlossen wird, wird es automatisch gelöscht. Im Fehlerfall liefert `tmpfile` einen Null-Pointer.

`char *tmpnam (char *s);` Liefert einen gültigen Filenamen, der keinem existierenden File entspricht. Die Funktion kann in jedem Programm bis zu `TMP_MAX` mal aufgerufen werden.

Wenn das Argument ein Null-Pointer ist, erzeugt `tmpnam` den Filenamen in einem internen Puffer. Sonst wird angenommen, dass das Argument auf ein Char-Array von mindestens `L_tmpnam` Zeichen zeigt und der Filename wird in dieses Array geschrieben. Auf jeden Fall retourniert `tmpnam` einen Pointer auf den erzeugten Filenamen.

Filezugriffsfunktionen

`int fclose (FILE *stream);` Schließt ein File. Zum Schreiben gepufferte Daten werden geschrieben, zum Lesen gepufferte werden verworfen, der Puffer wird freigegeben, und der Stream vom File getrennt. Nach einem `fclose` ist der Stream nicht mehr verwendbar. Tritt ein Fehler auf, so retourniert `fclose` den Wert `EOF`, sonst 0.

`int fflush (FILE *stream);` Sendet alle Daten die sich im Ausgabepuffer des Streams befinden an das Betriebssystem. Ist `stream` ein Null-Pointer so werden *alle* Ausgabestreams geflusht.

Tritt ein Fehler auf, so retourniert `fflush` den Wert `EOF`, sonst 0.

`FILE *fopen (const char *filename, const char *mode);` Öffnet ein File im angegebenen Modus und liefert den erzeugten Stream, oder einen Null-Pointer, wenn ein Fehler aufgetreten ist.

Die möglichen Modi sind in Tabelle 3.4 zusammengefasst.

Modus	Beschreibung
r	Öffne existierendes File zum Lesen
w	Erzeuge neues File zum Schreiben
a	Öffne oder erzeuge File zum Anhängen
r+	Öffne existierendes File zum Lesen und Schreiben
w+	Erzeuge neues File zum Lesen und Schreiben
a+	Öffne oder erzeuge File zum Lesen und Anhängen

Tabelle 3.4: Modi für `fopen`

Files werden normalerweise als Text-Streams geöffnet. Wird ein `b` an den Modus angehängt, werden sie als Binärstreams geöffnet.

Ist ein File zum Anhängen geöffnet, so werden alle Schreiboperationen am Ende des Files ausgeführt, ohne Rücksicht auf Positionier-Operationen.

Wenn ein Stream zum Lesen und Schreiben geöffnet ist, so muss zwischen Lese- und Schreiboperationen ein `fflush` oder eine Positionierfunktion aufgerufen werden.

Nach dem Öffnen sind Streams voll gepuffert, wenn sie nicht mit einem interaktiven Gerät verbunden sind.

`FILE *freopen (const char *filename, const char *mode, FILE *stream);`
Öffnet das angegebene File und verbindet es mit `stream`. Ist `stream` bereits geöffnet, so wird er vorher geschlossen.

Im Fehlerfall wird `NULL` retourniert, sonst `stream`.

`void setbuf (FILE *stream, char *buf);` Ist `buf` `NULL`, so wird `stream` nicht gepuffert, sonst ist `stream` gepuffert mit dem Puffer `buf` mit einer Größe von `BUFSIZ` Bytes.

Achtung Der Puffer `buf` muss natürlich vorher mit der entsprechenden Größe angelegt werden.

`int setvbuf (FILE *stream, char *buf, int mode, size_t size);` Setzt Art der Pufferung und den Puffer.

Ist mode `_IONBF`, so ist `stream` ungepuffert, ist es `_IOLBF`, so ist `stream` zeilengepuffert, und ist es `_IOFBF`, so ist es voll gepuffert. Ist `buf` $\neq$ `NULL`, so wird das Character Array der Größe `size`, auf das `buf` zeigt, als Puffer verwendet.

Portabilität Viele Implementierungen allozieren einen Puffer entsprechender Größe, wenn `size` $\neq 0$ aber `buf` = `NULL`. Dieses Verhalten ist aber nicht garantiert.

Formatierte Ein- und Ausgabe

`int fprintf (FILE *stream, const char *format, ...);` Schreibt seine Argumente auf `stream`. Der String `format` besteht aus Characters, die einfach auf `stream` ausgegeben werden, und *Format-Anweisungen*, die mit einem `%`-Zeichen beginnen und bestimmen, wie die weiteren Argumente behandelt werden.

Nach einem %-Zeichen folgen:

- Null oder mehr Flags (in beliebiger Reihenfolge), die folgende Wirkung haben:
  - `-` Linksbündige Ausgabe.
  - `+` Nichtnegative Zahlen beginnen mit einem +-Zeichen.
  - *`space`* Nichtnegative Zahlen beginnen mit einem Space.
  - `#` Oktalzahlen werden mit führender `0` ausgegeben, Sedezimalzahlen mit führendem `0x`, Floatingpoint-Zahlen enthalten immer einen Dezimalpunkt
  - `0` Zahlen werden bis zur Feldbreite mit führenden Nullen aufgefüllt.
- Eine optionale Feldbreite.

  Hat das Ergebnis der Konversion weniger Characters als die Feldbreite, so wird auf der linken Seite mit Spaces aufgefüllt.

  Die Feldbreite kann entweder als positive Zahl oder als Stern ausgedrückt werden. Im letzteren Fall wird das nächste Argument (das vom Typ `int` sein muss), als Feldbreite genommen.
- Eine optionale Genauigkeit.

  Bei ganzen Zahlen die Mindestanzahl von Ziffern. Bei Fließkommazahlen die Anzahl der Ziffern hinter dem Dezimalpunkt. Bei Strings die Anzahl der Zeichen, die maximal ausgegeben werden sollen.

  Die Genauigkeit besteht aus einem Punkt, gefolgt von entweder einer positiven Zahl oder einem Stern. Im letzteren Fall wird das nächste Argument (das vom Typ `int` sein muss), als Feldbreite genommen.
- Ein optionaler Typ-Modifier.

  Ein `h` drückt aus, dass das nächste Argument ein `short int` (`d`, `i` Konversion), `unsigned short int` (`o`, `u`, `x`, `X` Konversion), bzw. `short int *` (`n` Konversion) ist.

  Ein `l` drückt aus, dass das nächste Argument ein `long int` (`d`, `i` Konversion), bzw. `unsigned long int` (`o`, `u`, `x`, `X` Konversion) ist.

  Ein `L` drückt aus, dass das nächste Argument ein `long double` (`e`, `E`, `f`, `g`, `G` Konversion) ist.
- Eine Konversions-Anweisung:
  - `d,i` Das nächste Argument ist vom Typ `int` und wird als Dezimalzahl (ev. mit führendem Vorzeichen) ausgegeben.
  - `o,u,x,X` Das nächste Argument ist vom Typ `unsigned int` und wird als Oktal-Dezimal-, Sedezimal-Zahl mit Kleinbuchstaben, bzw. Sedezimalzahl mit Großbuchstaben ausgegeben.

`f` Das nächste Argument ist vom Typ `double` und wird in der Form *[−] ddd.ddd* ausgegeben. Wird keine Genauigkeit angegeben, so werden 6 Nachkommastellen gedruckt. Ist die Genauigkeit 0, so wird auch kein Dezimalpunkt gedruckt.

`e,E` Das nächste Argument ist vom Typ `double` und wird in der Form *[−] d.ddd*`e`±*dd* ausgegeben. Wird keine Genauigkeit angegeben, so werden 6 Nachkommastellen gedruckt. Ist die Genauigkeit 0, so wird auch kein Dezimalpunkt gedruckt. Der Exponent hat mindestens zwei Stellen. Die `E`-Konversion druckt ein `E` statt einem `e` vor dem Exponenten.

`g,G` Druckt im `f`, `e`, oder `E` Format, je nachdem, welches kürzer ist.

`c` Das nächste Argument ist vom Typ `int` und wird als Character gedruckt.

`s` Das nächste Argument ist vom Typ `char *` und zeigt auf einen String, der ausgegeben wird.

`p` Das nächste Argument ist vom Typ `void *` und wird auf eine Form, die von der Implementation abhängig ist, ausgegeben.

`n` Das nächste Argument ist vom Typ `int *`. Die Anzahl der bisher ausgegebenen Zeichen wird in den Integer auf den dieses Argument zeigt, geschrieben. Es erfolgt keine Ausgabe.

`%` Ein `%`-Zeichen wird ausgegeben.

Der Return-Wert ist die Anzahl der Zeichen, die ausgegeben wurden, oder ein negativer Wert, wenn ein Fehler aufgetreten ist.

`int fscanf (FILE *stream, const char *format, ...);` liest von `stream`, wandelt die gelesenen Zeichen entsprechend den Format-Anweisungen in `format` um und weist diese Werte den Argumenten zu.

Alle Zeichen in `format` außer Leerzeichen und % müssen in der angegebenen Reihenfolge von `stream` gelesen werden.

Jede Folge von Leerzeichen in `format` muss null oder mehr Leerzeichen in `stream` entsprechen.

Das %-Zeichen leitet eine Format-Anweisung ein und ist gefolgt von:

- Einem optionalen `*`, der eine Zuweisung unterdrückt.
- Einer optionalen, positiven Feldweite.
- Einem optionalen Typmodifier. Dieser hat die gleiche Bedeutung wie bei `fprintf`. Zusätzlich bedeutet `l` bei `e`, `f`, und `g`-Konversionen, dass das Argument ein `double *` und kein `float *` ist.
- Ein Zeichen, das die Art der Konversion bestimmt:

  `d` Eine dezimale Integerzahl, möglicherweise mit Vorzeichen. Das Argument ist ein `int *`.

  `i` Eine Integerzahl im selben Format wie von `strtol` mit Basis `0` verlangt. Das Argument ist ein `int *`.

  `o` Eine oktale Integerzahl, möglicherweise mit Vorzeichen. Das Argument ist ein `unsigned int *`.

`u` Eine dezimale Integerzahl, möglicherweise mit Vorzeichen. Das Argument ist ein `unsigned int *`.

`x,X` Eine sedezimale Integerzahl, möglicherweise mit Vorzeichen. Das Argument ist ein `unsigned int *`.

`e,f,g,E,G` Eine Fließkommazahl im selben Format wie von `strtod` verlangt. Das Argument ist ein `float *`.

`s` Ein String ohne Leerzeichen. Das Argument ist ein `char *`.

`[` Ein String der nur aus den angegebenen Zeichen besteht. Dem `[` folgt eine Liste von Zeichen oder Ranges (der Form *a*−*b*), gefolgt von `]`. Ist das erste Zeichen der Liste ein `^`, so werden alle Zeichen akzeptiert, die *nicht* in der Liste vorkommen. Das Argument ist ein `char *`.

`c` Ein String der durch die Feldlänge angegebenen Länge (default 1). Das Argument ist ein `char *`.

`p` Ein Pointer der selben Form wie von `fprintf` ausgegeben. Das Argument ist ein `void **`.

`n` Die Anzahl der bisher gelesenen Zeichen wird in den `int` auf den das nächste Argument zeigt, geschrieben. Dabei findet keine Umwandlung statt. Das Argument ist ein `int *`.

`%` Ein %-Zeichen wird von `stream` gelesen. Es findet keine Umwandlung statt.

Wird ein unerwartetes Zeichen von `stream` gelesen, so wird es in den Stream zurückgestellt und `fscanf` bricht ab.

Der Returnwert ist die Anzahl der erfolgreich umgewandelten Felder, oder EOF, wenn ein Lesefehler vor der Umwandlung des ersten Feldes auftrat.

Achtung `fscanf` kann, wenn bei String-Leseoperationen keine Feldweite angegeben wird, über das Ende des bereitgestellten Arrays hinausschreiben.

Da der erste Character, der nicht erfolgreich umgewandelt werden konnte, wieder in den Stream zurückgestellt wird, können Eingabefehler zu Endlosschleifen führen.

Aus diesen Gründen ist es besser, statt `fscanf` die Funktion `fgets`, gefolgt von `sscanf` zu verwenden.

`int sprintf (char *s, const char *format, ...);` entspricht `fprintf`, schreibt aber in den String `s` und nicht auf einen Stream.

Achtung Es ist darauf zu achten, dass der String, auf den `s` zeigt, lange genug ist, um den gesamten Output von `sprintf` (inklusive `'\0'`) aufzunehmen.

`int sscanf (const char *s, const char *format, ...);` entspricht `fscanf`, liest aber vom String `s` statt von einem Stream.

`int printf (const char *format, ...);` entspricht `fprintf`, schreibt aber auf `stdout`.

`int vfprintf (FILE *stream, const char *format, va_list arg);` entspricht `fprintf`, hat aber statt einer variablen Argumentliste ein Argument, das mittels `va_start` aus einer variablen Argumentliste erzeugt wurde.

`int vprintf (const char *format, ...);` entspricht `vfprintf`, schreibt aber auf `stdout`.

`int vsprintf (char *s, const char *format, ...);` entspricht `vfprintf`, schreibt aber in den String `s` statt auf einen Stream.

Achtung Es ist darauf zu achten, dass der String auf den `s` zeigt, lange genug ist, um den gesamten Output von `vsprintf` (inklusive `'\0'`) aufzunehmen.

Beispiele

Im Folgenden ist ein Beispiel für der Verwendung der Funktion `vfprintf` in einer Fehlerausgabefunktion gezeigt:

```
const char    *Cmd = "";

void error(int ecode, const char *fmt, ...)
{
 va_list        args;

 va_start(args, fmt);
 (void) fprintf(stderr, "%s: ", Cmd);
 (void) vfprintf(stderr, fmt, args);
 va_end(args);
 exit(ecode);
}

int main(int argc, char **argv)
{
 ...

 /* setze den Kommandonamen fuer die Fehlerausgabefunktion */
 Cmd = argv[0];

 ...
 if (chmod(filename, 0700) < 0)
        error(EXIT_FAILURE, "chmod for '%s' failed: %s\n",
              filename, strerror(errno));
 ...
 return EXIT_SUCCESS;
}
```

Zeichenweise Ein- und Ausgabe

`int fgetc (FILE *stream);` Liest ein Zeichen von `stream`. Liefert dieses Zeichen (als `unsigned char` umgewandelt in eine `int`) bzw. `EOF` wenn kein Zeichen gelesen werden kann.

Achtung diese Funktion liefert immer einen Wert vom Typ `int` zurück.

`char *fgets (char *s, int n, FILE *stream);` Liest maximal eine Zeile *inklusive* `'\n'` von `stream` in `s` ein. Maximal `n - 1` Zeichen werden gelesen, und ein `'\0'`-Character wird unmittelbar hinter das letzte gelesene Zeichen geschrieben.

Im fehlerfreien Fall retourniert die Funktion `s` (falls mindestens 1 Zeichen gelesen wurde). Wenn das Ende der Datei erreicht wurde und keine Zeichen in das Array gelesen wurden, oder wenn ein Lesefehler passiert ist, liefert die Funktion `NULL` zurück!

`int fputc (int c, FILE *stream);` schreibt das Zeichen `c` (in `unsigned char` umgewandelt) auf `stream`. Tritt dabei ein Fehler auf, so wird `EOF` zurückgeliefert, sonst `c`.

`int fputs(const char *s, FILE *stream);` schreibt den String, auf den `s` zeigt, auf `stream`.

Der Return-Wert ist `EOF` im Fehlerfall, sonst ein nicht-negativer Wert.

`int getc (FILE *stream);` Entspricht `fgetc`, kann aber als Makro implementiert sein, das sein Argument mehr als einmal auswertet.

`int getchar (void);` Entspricht `getc (stdin);`

`char *gets (char *s);` Liest von `stdin` in das Array, auf das s zeigt, bis ein `'\n'` oder `EOF` gelesen wurde. `'\n'` wird verworfen und ein `'\0'` wird unmittelbar hinter das letzte gelesene Zeichen geschrieben.

Der Return-Wert enspricht dem von `fgets`.

Achtung Ist die Eingabezeile länger als das Array, auf das `s` zeigt, so werden nachfolgende Speicherbereiche überschrieben. Daher sollte statt `gets` stets `fgets` verwendet werden.

`int putc (int c, FILE *stream);` Entspricht `fputc`, kann aber als Makro implementiert sein, das sein zweites Argument mehr als einmal auswertet.

Achtung Seiteneffekte können bei Verwendung von `putc` daher unerwartete Auswirkungen haben.

`int putchar (int c);` Entspricht `putc (c, stdout);`

`int puts(const char *s);` schreibt den String, auf den `s` zeigt, gefolgt von einem `'\n'`, auf `stdout`.

Der Return-Wert ist `EOF` im Fehlerfall, sonst ein nicht-negativer Wert.

`int ungetc (int c, FILE *stream);` Stellt `c` in den Stream zurück. Nachfolgende Leseoperationen lesen zurückgestellte Zeichen in umgekehrter Reihenfolge. Das mit dem Stream verbundene File wird dadurch nicht geändert, und nachfolgende Positionierfunktionen verwerfen zurückgestellte Zeichen. Die Anzahl der Zeichen, die zurückgestellt werden können, kann begrenzt sein, ist aber mindestens 1.

Ein erfolgreiches `ungetc` löscht den EOF-Indikator. Für Binär-Streams wird die File-Position (so vorhanden) um 1 verringert, für Text-Streams ist sie unbestimmt, bis alle zurückgestellten Zeichen wieder gelesen sind.

Wenn zuviele Zeichen zurückgestellt werden, liefert `ungetc` `EOF`.

Direkte Ein- und Ausgabe

`size_t fread (void *ptr, size_t size, size_t nmemb, FILE * stream);` Liest `nmemb` Elemente von jeweils `size` Bytes Größe von `stream` und schreibt sie in das Array, auf das `ptr` zeigt.

Die Anzahl der erfolgreich gelesenen Elemente wird zurückgeliefert (welche im Fall, dass das Fileende erreicht wurde, oder im Fehlerfall kleiner als `nmemb` sein kann).

`size_t fwrite (void *ptr, size_t size, size_t nmemb, FILE * stream);` Schreibt `nmemb` Elemente von jeweils `size` Bytes Größe aus dem von Array, auf das `ptr` zeigt, auf `stream`.

Die Anzahl der erfolgreich geschriebenen Elemente wird zurückgeliefert (welche im Fehlerfall kleiner als `nmemb` ist).

Positionierfunktionen

Positionierfunktionen dienen dazu, den Schreib-Lese-Zeiger eines Streams auf eine andere Stelle zu positionieren. Dies funktioniert im Allgemeinen nur bei Files und I/O-Devices, die direkten Zugriff erlauben.

`int fseek (FILE *stream, long int offset, int whence);` Setzt die Position für Lese-Schreibe-Operationen auf `offset`, relativ zum Startpunkt `whence`. `whence` kann folgende Werte annehmen:

`SEEK_SET` `offset` bezieht sich auf den Anfang des Files.

`SEEK_CUR` `offset` bezieht sich auf die aktuelle Position.

`SEEK_END` `offset` bezieht sich auf das Ende des Files.

Für binäre Streams wird `offset` in Bytes gemessen. `SEEK_END` muss für binäre Strings keine sinnvollen Ergebnisse liefern.

Für Text-Streams müssen nur Positionierungen auf den Anfang des Files und Positionen, die vorher mittels `ftell` ermittelt wurden, funktionieren, wobei der `whence` Parameter den Wert `SEEK_SET` haben muss.

Der Return-Wert ist 0, wenn erfolgreich.

`long int ftell (FILE *stream);` Ermittelt die Position im File und liefert sie als Return-Wert.

Im Fehlerfall liefert `ftell -1L` und setzt `errno` auf einen positiven Wert.

`void rewind (FILE *stream);` Entspricht `(void) fseek (stream, 0L, SEEK_SET);` und löscht zusätzlich den Fehlerindikator (siehe auch `clearerr`).

`int fgetpos (FILE *stream, fpos_t *pos);` Ermittelt die Position im File und speichert sie in `pos`.

Der Return-Wert ist 0, wenn erfolgreich. Im Fehlerfall wird ein Wert ungleich 0 retourniert und `errno` auf einen positiven Wert gesetzt.

`int fsetpos (FILE *stream, const fpos_t *pos);` Setzt die Position auf eine vorher mit `fgetpos` ermittelte Position `*pos`.

Im Fehlerfall liefert `fsetpos` einen Wert $\neq 0$ und setzt `errno` auf einen positiven Wert.

Portabilität `fsetpos` und `fgetpos` wurden vom ANSI-Komitee eingeführt, um Positionieroperationen auch in Files durchführen zu können für die ein `long`-Wert zur Angabe der Position nicht mehr ausreicht.

Fehlerbehandlung

`void clearerr (FILE *stream);` löscht EOF- und Fehlerindikator für den Stream.

`int feof (FILE *stream);` Liefert einen Wert $\neq 0$, wenn der EOF-Indikator des Streams gesetzt ist.

`int ferror (FILE *stream);` Liefert einen Wert $\neq 0$, wenn der Fehlerindikator des Streams gesetzt ist.

`void perror (const char *s);` Schreibt eine Fehlermeldung auf `stderr`. Wenn `s` `NULL` ist, so ist es äquivalent zu `(void)fprintf (stderr,"%s\n", strerror (errno));`, sonst zu `(void)fprintf (stderr,"%s: %s\n", s, strerror (errno));`.

3.4.10 stdlib.h

Pseudozufallszahlen

`int rand (void);` Liefert eine Pseudozufallszahl im Intervall [0,`RAND_MAX`].

`void srand (unsigned int seed);` Initialisiert den Pseudozufallszahlengenerator.

Umwandlungsfunktionen

`double atof (const char *nptr);` Äquivalent zu `strtod (nptr, (char **) NULL);` (siehe unten)

Achtung `atof` retourniert im Fehlerfall `0.0`, was nicht von der der erfolgreichen Konversion der Zahl 0 zu unterscheiden ist. Daher sollte `strtod` verwendet werden.

`int atoi (const char *nptr);` Äquivalent zu `(int) strtol (nptr, (char **) NULL, 10);` (siehe unten)

Achtung `atoi` retourniert im Fehlerfall `0`, was nicht von der der erfolgreichen Konversion der Zahl 0 zu unterscheiden ist. Daher sollte `strtol` verwendet werden.

`long atol (const char *nptr);` Äquivalent zu `strtol (nptr, (char **) NULL, 10);` (siehe unten)

Achtung `atol` retourniert im Fehlerfall `0`, was nicht von der der erfolgreichen Konversion der Zahl 0 zu unterscheiden ist. Daher sollte `strtol` verwendet werden.

`double strtod (const char *nptr, char ** endptr);` Wandelt den Anfang des Strings, auf den `nptr` zeigt, in eine `double` Zahl um. Führende Leerzeichen werden ignoriert. Wenn für `endptr` nicht der Null-Pointer übergeben wurde, so wird ein Pointer auf das erste Zeichen, das nicht umgewandelt werden konnte, in das `char *`-Objekt geschrieben, auf das `endptr` zeigt.

Die Form von Strings, die als Fließkommazahlen akzeptiert werden, hängt von der aktuellen Locale ab. In der `"C"`-Locale müssen sie die Form einer Fließkomma-Konstanten haben.

Der Return-Wert ist der Wert der umgewandelten Zahl. Wurde keine Umwandlung durchgeführt so ist er 0. Bei Overflow wird der Wert $\pm$`HUGE_VAL` geliefert und `errno` auf `ERANGE` gesetzt. Bei Underflow wird 0 geliefert und `errno` auf `ERANGE` gesetzt.

`long strtol (const char *nptr, char ** endptr, int base);` Interpretiert den Anfang des Strings, auf den `nptr` zeigt als ganzzahlige Zahl zur Basis `base` (ev. mit Vorzeichen) und wandelt ihn in eine `long` Zahl um. Führende Leerzeichen werden ignoriert. Die Buchstaben `A`–`Z` bzw. `a`–`z` werden als Ziffern mit den Werten 10–35 interpretiert (der Maximalwert für `base` ist daher 36). Ziffern deren Wert $\geq$ der Basis ist, werden nicht akzeptiert. Ist `base` $=$ 0, so gelten die selben Regeln wie für ganzzahlige Konstanten in C (s. S. 141). Wenn für `endptr` nicht der Null-Pointer übergeben wurde, so wird ein Pointer auf das erste Zeichen, das nicht umgewandelt werden konnte in das `char *`-Objekt geschrieben, auf das `endptr` zeigt.

In anderen Locales als der `"C"`-Locale können zusätzliche Strings akzeptiert werden.

Der Return-Wert ist der Wert der umgewandelten Zahl. Wurde keine Umwandlung durchgeführt so ist er 0. Bei Overflow wird der Wert `LONG_MAX` bzw. `LONG_MIN` geliefert und `errno` auf `ERANGE` gesetzt.

`unsigned long strtoul (const char *nptr, char ** endptr, int base);`
Wie `strtol`, liefert aber einen `unsigned long` Wert und im Falle eines Overflows den Wert `ULONG_MAX`.

Beispiele
Hier ein Beispiel für die Verwendung von `strtol`:

```
char *eptr,
     *toconvert;
long value;
...
/* setze errno auf 'no error' */
errno = 0;

/* konvertiere den String in toconvert zu einem long */
value = strtol(toconvert, &eptr, 10);
if ( (eptr == toconvert) || (*eptr != '\0') )
{
    /* FEHLERHAFTER STRING */
    ...
}
if (errno == ERANGE)
{
    /* OVERFLOW */
    ...
}
```

Speicherverwaltung

Jeder Aufruf von `malloc`, `calloc`, und `realloc` erzeugt ein eigenes Objekt, dessen Alignment für jeden Datentyp genügt, für den der reservierte Speicher ausreicht. Kann nicht genügend Platz alloziert werden, so wird der Null-Pointer retourniert, sonst ein Pointer auf den allozierten Speicherplatz.

`void *calloc (size_t nmemb, size_t size);` Alloziert Speicher für `nmemb` Elemente von jeweils `size` Größe und füllt diesen mit 0-Bits.

Achtung Der allozierte Speicher ist für alle ganzzahligen Typen mit 0 initialisiert, jedoch nicht notwendigerweise für Pointer und Fließkommazahlen. Diese müssen explizit initialisiert werden.

`void free (void *ptr);` Gibt einen vorher allozierten Speicherbereich frei. `ptr` muss ein Pointer sein, der von `malloc`, `calloc` oder `realloc` geliefert wurde, oder der Null-Pointer (der ignoriert wird).

`void *malloc (size_t size);` Alloziert Speicher für ein Element der Größe `size`. Der Speicher wird nicht initialisiert.

`void *realloc (void *ptr, size_t size);` Ändert die Größe eines vorher allozierten Speicherbereichs auf `size`. Ist dies nicht möglich, so wird ein neuer Speicherbereich der Größe `size` alloziert, Daten im alten Speicherbereich werden in den neuen kopiert, der alte Speicherbereich wird freigegeben und ein Pointer auf den neuen Speicherbereich wird zurückgeliefert. Ist auch das nicht möglich, so wird der Null-Pointer zurückgeliefert, und der ursprüngliche Speicherbereich bleibt intakt.

Achtung Bei manchen Speicherverwaltungsverfahren kann auch das *Verkleinern* eines Speicherbereichs fehlschlagen. Der Return-Wert ist daher auf jeden Fall zu überprüfen.

Environment

`void abort (void);` Löst einen abnormalen Programmabbruch durch das Signal `SIGABRT` aus.

`int atexit (void (* func)(void));` Registriert eine Funktion, die bei normalem Programmabbruch ausgeführt werden soll.

`void exit (int status);` Löst einen normalen Programmabbruch aus. Zuerst werden alle Funktionen, die mittels `atexit` registriert wurden in umgekehrter Reihenfolge ihrer Registrierung aufgerufen, dann werden alle offenen Streams geschlossen und die Kontrolle wird an das Betriebssystem zurückgegeben. Wenn Status den Wert `0` oder `EXIT_SUCCESS` hat, so wird dem Betriebssystem mitgeteilt, dass das Programm erfolgreich terminiert ist, ist `status` `EXIT_FAILURE`, so wird eine erfolglose Termination gemeldet.

`char *getenv(const char *name);` Liefert den Wert der Environment-Variablen `name`, oder einen Null-Pointer, wenn diese nicht existiert.

`int system(const char *string);` Ruft einen Kommandointerpreter mit dem Kommando `string` auf. Der Return-Wert dieser Funktion ist implementierungsabhängig.

Suchen und Sortieren

```
void *bsearch (const void *key, const void *base, size_t nmemb,
    size_t size, int (*compar)(const void *, const void *));
```

Durchsucht ein aufsteigend sortiertes Feld `base`, das aus `nmemb` Elementen der Größe `size` besteht, nach einem Eintrag, der gleich dem Element ist, auf das `key` zeigt.

Für jeden Vergleich wird die Funktion `compar` aufgerufen. Diese muss einen negativen Wert liefern, wenn das erste Element kleiner ist als das zweite, einen positiven, wenn das erste Element größer ist, und 0, wenn beide gleich sind.

Der Returnwert ist ein Pointer auf ein passendes Element, oder `NULL`, wenn kein passendes Element gefunden wurde.

```
void qsort (const void *base , size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
```

Sortiert ein Feld aufsteigend.

Für jeden Vergleich wird die Funktion `compar` aufgerufen. Diese muss einen negativen Wert liefern, wenn das erste Element kleiner ist als das zweite, einen postiven, wenn das erste Element größer ist, und 0, wenn beide gleich sind.

Beispiel für `qsort` und `bsearch`

Das folgende Beispielfragment zeigt die Verwendung von `qsort` und `bsearch`. Es wird angenommen, dass bei der Initialisierung nur eindeutige Schlüssel verwendet werden.

```
...
typedef struct
{
    int key,
        value;
} elem_t;
...
int compare(const void *a, const void *b)
{
 const elem_t *ea, *eb;

 ea = a; /* Umwandlung in den richtigen Typ */
 eb = b;
 if (ea->key == eb->key) return 0;
 if (ea->key < eb->key) return -1;
 return 1;
}

void foobar(void)
{
 ...
 elem_t table[MAX],
        search,
        *result;
 ...
 /* initialisiere die Tabelle */
 ...
 /* sortiere die Eintraege nach key */
 qsort(table, sizeof(table)/ sizeof(table[0]), sizeof(table[0]), compare);
 ...
 /* suche den entsprechenden key aus der Tabelle */
 search.key = 4711;
 result = bsearch(&search,
                  table,
                  sizeof(table)/ sizeof(table[0]),
```

```
                  sizeof(table[0]),
                  compare);
 if (result == NULL)
 {
     ...   /* KEY NOT FOUND */
 }
 else
 {
     ... printf("%d\n", result->value) ...
 }
 ...
}
```

Arithmetische Funktionen

`int abs (int j);` Liefert den Absolutbetrag des Arguments.

`div_t div (int numer, int denom);` Liefert Quotient und Rest der ganzzahligen Division `numer/denom`, wobei der Quotient immer auf 0 zu gerundet wird.

`long labs (long j);` Liefert den Absolutbetrag des Arguments.

`ldiv_t ldiv (long numer, long denom);` Liefert Quotient und Rest der ganzzahligen Division `numer/denom`, wobei der Quotient immer auf 0 zu gerundet wird.

Achtung Manualpages und `stdlib.h` nicht standard konform!
ANSI Standard: `ldiv_t ldiv(...)` bzw. `div_t div(...)`
Manualpage und `stdlib.h`: `struct ldiv_t(...)` bzw. `struct div_t div(...)`

Multibyte-Character- und String-Funktionen

Diese Funktionen arbeiten mit Multibyte Characters und Strings (für Systeme, bei denen nicht alle Zeichen in einem `char` dargestellt werden können). Multibyte-Characters sind Folgen von `char`s, die als ein Zeichen aufgefasst werden. Ein sogenannter 'weiter' Character (Typ `wchar_t`) dagegen ist groß genug um alle Zeichen aufzunehmen.

Hier seien nur die Funktionen aufgezählt und auf die Handbücher solcher Systeme verwiesen.

`int mblen (const char *s, size_t n);` Anzahl der Bytes in einem Multibyte Character.

`int mbtowc (wchar_t *pwc, const char *s, size_t n);` Wandelt einen Multibyte- Character in einen weiten Character um.

`int wctomb (char *s, wchar_t wchar);` Wandelt einen weiten Character in einen Multibyte-Character um.

`size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);` Wandelt einen Multibyte String in einen String aus weiten Characters um.

`size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);` Wandelt einen String aus weiten Characters in einen Multibyte String um.

3.4.11 string.h

Kopierfunktionen

`void *memcpy (void *dst, const void *src, size_t n);` Kopiert `n` Bytes von `src` nach `dst`. Die Bereiche dürfen sich *nicht* überlappen.

Der Returnwert ist `dst`.

`void *memmove (void *dst, const void *src, size_t n);` Kopiert `n` Bytes von `src` nach `dst`. Die Bereiche dürfen sich überlappen.

Der Returnwert ist `dst`.

`char *strcpy (char *dst, const char *src);` Kopiert den mit `'\0'` abgeschlossenen String von `src` nach `dst`. Die Bereiche dürfen sich *nicht* überlappen.

Der Returnwert ist `dst`.

Achtung Ist der String auf den `src` zeigt länger als der Speicherbereich auf den `dst` zeigt oder nicht mit `'\0'` abgeschlossen, so können andere Daten überschrieben werden.

`char *strncpy (char *dst, const char *src, size_t n);` Kopiert einen String von `src` nach `dst`. Ist der String `n` Characters lang oder länger, so werden nur `n` Characters kopiert und der Zielstring ist **nicht** mit `'\0'` abgeschlossen. Ist er kürzer, so wird `dst` mit `'\0'`-Characters auf Länge `n` aufgefüllt. Die Bereiche dürfen sich *nicht* überlappen.

Der Returnwert ist `dst`.

Zusammenhängen von Strings

`char *strcat (char *dst, const char *src);` Hängt den mit `'\0'` abgeschlossenen String `src` an den mit `'\0'` abgeschlossenen String `dst` an. Der Null-Character (`'\0'`) von `dst` wird durch das erste Zeichen von `src` überschrieben. Die Bereiche dürfen sich *nicht* überlappen.

Der Returnwert ist `dst`.

Achtung Ist der Speicherbereich, auf den `dst` zeigt, kürzer als die Summe der Längen der Strings `src` und `dst`, oder ist `src` nicht mit `'\0'` abgeschlossen, so können andere Daten überschrieben werden.

`char *strncat (char *dst, const char *src, size_t n);` Hängt den mit `'\0'` abgeschlossenen String `src` an den mit `'\0'` abgeschlossenen String `dst` an, wobei höchstens `n` Characters (ohne `'\0'`) kopiert werden. Das Ergebnis wird mit einem Null-Character abgeschlossen. Die Bereiche dürfen sich *nicht* überlappen.

Der Returnwert ist `dst`.

Vergleichsfunktionen

`int memcmp (const void *s1, const void *s2, size_t n);` Vergleicht die ersten `n` Zeichen von `s1` mit den ersten `n` Zeichen von `s2`. Die Vergleiche werden ausgeführt als ob die Zeichen vom Typ `unsigned char` wären, und das erste unterschiedliche Zeichen bestimmt den Returnwert.

Ist `s1` $<$ `s2`, so ist der Return-Wert negativ, ist `s1` $>$ `s2`, so ist er positiv. Sind beide gleich ist er 0.

`int strcmp (const char *s1, const char *s2);` Vergleicht die Strings `s1` und `s2`. Die Vergleiche werden ausgeführt als ob die Zeichen vom Typ `unsigned char` wären, und das erste unterschiedliche Zeichen bestimmt den Returnwert.

Ist `s1` $<$ `s2`, so ist der Return-Wert negativ, ist `s1` $>$ `s2`, so ist er positiv. Sind beide gleich ist er 0.

`int strcoll (const char *s1, const char *s2);` Vergleicht die Strings `s1` und `s2`. Die Vergleiche werden entsprechend der aktuellen Locale ausgeführt. In der `"C"`-Locale ist `strcoll` äquivalent zu `strcmp`.

Ist `s1` $<$ `s2`, so ist der Return-Wert negativ, ist `s1` $>$ `s2`, so ist er positiv. Sind beide gleich ist er 0.

`int strncmp (const char *s1, const char *s2, size_t n);` Entspricht `strcmp` jedoch werden maximal `n` Zeichen verglichen.

`size_t strxfrm (char *dst, const char *src, size_t n);` Konvertiert den String `src` und kopiert das Ergebnis in String `dst`. Die Umwandlung erfolgt solcherart, dass der Vergleich zweier umgewandelter Strings mittels `strcmp` das selbe Ergebnis liefert wie der Vergleich der Originalstrings mittels `strcoll`. Es werden maximal `n` Zeichen (inklusive `'\0'`) in `dst` geschrieben. Ist `n` 0, so kann `dst` ein Null-Pointer sein.

Der Return-Wert gibt die Länge des nach `dst` kopierten Strings an. Bietet `dst` nicht genug Platz, so wird die benötigte Länge des Strings zurückgeliefert (mittels `malloc (strxfrm (NULL, src, 0) + 1)` kann also genügend Speicherplatz zur Umwandlung von `src` reserviert werden).

Suchfunktionen

`void *memchr (const void *s, int c, size_t n);` Liefert einen Pointer auf das erste Vorkommnis von `c` in den ersten `n` Zeichen des Objekts auf das `s` zeigt, oder einen Null-Pointer, wenn `c` nicht gefunden wurde.

`char *strchr (const char *s, int c);` Liefert einen Pointer auf das erste Vorkommnis von `c` im String `s`, oder einen Null-Pointer, wenn `c` nicht gefunden wurde.

`size_t strcspn (const char *s1, const char *s2);` Liefert die Länge des Segments von `s1`, das nur aus Zeichen, die nicht in `s2` vorkommen, besteht.

`char *strpbrk (const char *s, const char *s2);` Liefert einen Pointer auf das erste Vorkommnis eines Zeichens aus `s2` im String `s`, oder einen Null-Pointer, wenn keines gefunden wurde.

`char *strrchr (const char *s, int c);` Liefert einen Pointer auf das letzte Vorkommnis von `c` im String `s`, oder einen Null-Pointer, wenn `c` nicht gefunden wurde.

`size_t strspn (const char *s1, const char *s2);` Liefert die Länge des ersten Segments von `s1`, das nur aus Zeichen aus `s2` besteht.

`char *strstr (const char *s1, const char *s2);` Liefert einen Pointer auf das erste Vorkommnis des Strings `s2` im String `s1`, oder einen Null-Pointer, wenn keines gefunden wurde.

`char *strtok (char *s1, const char *s2);` Eine Folge von Aufrufen von `strtok` zerlegt einen String `s1` in einzelne Token, die durch Zeichen aus `s2` getrennt sind.

Wird `strtok` mit einem Wert für `s1` ungleich `NULL` aufgerufen, so wird zuerst in dem String, auf den `s1` zeigt das erste Zeichen gesucht, das nicht in `s2` vorkommt. Kommt kein solches Zeichen vor, so wird ein Null-Pointer zurückgeliefert. Sonst ist dieses Zeichen der Anfang des ersten Tokens. Dann wird nach dem ersten Zeichen aus `s2` weitergesucht. Wird ein solches gefunden, so wird es durch einen Null-Character ersetzt und ein Pointer auf das diesem Zeichen folgende wird in einer statischen Variable gespeichert. Wird keines gefunden, so wird ein Pointer auf den abschließenden Null-Character gespeichert. In beiden Fällen wird ein Pointer auf das erste Token zurückgeliefert.

Ist `s1` der `NULL`-Pointer, so erfolgt die selbe Operation auf den intern gespeicherten Pointer.

Beispiele

Der folgende Code zerlegt einen String in durch Leerzeichen getrennte Felder und gibt sie durch Newlines getrennt aus:

```
#define SEP " \n\r\t\v"
for (p = strtok (s, SEP); p != NULL; p = strtok (NULL, SEP))
{
    printf ("%s\n", p);
}
```

Diverses

`void *memset (void *s, int c, size_t n);` Kopiert das Zeichen `c` in jedes der ersten `n` Zeichen des Objekts auf das `s` zeigt.

`char *strerror (int errnum);` Liefert eine zum Fehler `errnum` passende Fehlermeldung.

`size_t strlen (const char *s);` Liefert die Länge eines Strings.

3.4.12 time.h

Dieses Headerfile definiert drei Datentypen und diverse Funktionen zur Verwaltung der Zeit. Die Datentypen sind:

`clock_t` Anzahl von Clock-Ticks.

`time_t` Zeit in interner Darstellung.

`struct tm` Zeit in Kalenderdarstellung. Die Struktur enthält folgende Felder vom Typ `int`:

`tm_sec`	Sekunden (0–61) (inkl. zwei Schaltsekunden)
`tm_min`	Minuten (0–59)
`tm_hour`	Stunden (0–23)
`tm_mday`	Tag im Monat (1–31)
`tm_mon`	Monat (0–11, Jänner = 0!)
`tm_year`	Jahr (1900 = 0!)
`tm_wday`	Wochentag (0–6, Sonntag = 0)
`tm_yday`	Tag im Jahr (0–365, 1. Jänner = 0)
`tm_isdst`	> 0: Sommerzeit, $= 0$: keine Sommerzeit, < 0: unbekannt

`clock_t clock (void);` Liefert die CPU-Zeit in Clock-Ticks seit einem beliebigen Zeitpunkt in der Vergangenheit, der jedoch während der Programmausführung gleich bleibt. Die Anzahl der Clock-Ticks pro Sekunde liefert das Makro `CLOCKS_PER_SEC`. Im Fehlerfall ist der Returnwert `(clock_t) -1`.

`double difftime (time_t time1, time_t time0);` Liefert die Differenz `time1 - time0` in Sekunden.

`time_t mktime (struct tm *timeptr);` Errechnet aus den Feldern von `*timeptr` einen `time_t`-Wert. `*timeptr` enthält die Lokalzeit, `tm_wday` und `tm_yday` werden ignoriert und alle Felder können Werte außerhalb ihres Wertebereichs annehmen. `mktime` rechnet solche Bereichsüberschreitungen auf die anderen Felder auf und korrigiert sie entsprechend. Ebenso werden `tm_wday` und `tm_yday` eingefügt.

Der Returnwert ist der errechnete `time_t`-Wert, oder `time_t␣-1`, wenn kein Wert errechnet werden kann.

`time_t time (time_t *timer);` Liefert die aktuelle Kalenderzeit als Returnwert. Ist `timer` $\neq$ `NULL`, so wird diese Zeit auch noch in das Objekt gespeichert auf das `timer` zeigt (`t = time (NULL);` und `(void) time (&t);` haben also den selben Effekt).

`char *asctime (const struct tm *timeptr);` Wandelt die Zeit in `*timeptr` in einen String der Form `Sun␣Mar␣␣8␣22:59:35␣1992\n\0` um.

`char *ctime (const time_t *timer);` Ist äquivalent zu `asctime (localtime (timer));`

`struct tm *gmtime (const time_t *timer);` Spaltet die Zeit `*timer` in die einzelnen Felder einer `struct␣tm` auf, ausgedrückt als UTC (Universal Time Coordinated, hieß früher Greenwich Mean Time).

Liefert einen Pointer auf die Struktur, oder einen Null-Pointer, wenn eine Umrechnung in UTC nicht möglich ist.

`struct tm *localtime (const time_t *timer);` Spaltet die Zeit `*timer` in die einzelnen Felder einer `struct tm` auf (ausgedrückt als lokale Zeit) und liefert einen Pointer auf diese Struktur.

`size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr);` Erzeugt unter Kontrolle von `format` einen String `s` von maximal `maxsize` Zeichen Länge (inklusive `'\0'`). Alle Zeichen in Format außer % werden nach `s` kopiert, die folgenden Formatanweisungen werden, wie in Tabelle 3.5 aufgelistet, durch die in `*timeptr` enthaltenen Felder ersetzt. `strftime` liefert die Länge des entstehenden Strings oder 0, wenn der String länger als `maxsize` geworden wäre.

`%a`	abgekürzter Wochentag
`%A`	voller Wochentag
`%b`	abgekürzter Monatsname
`%B`	voller Monatsname
`%c`	Datum und Zeit
`%d`	Monatstag als Dezimalzahl (01–31)
`%H`	Stunde (00–23)
`%I`	Stunde (01–12)
`%j`	Tag des Jahres (001–366)
`%m`	Monat (01–12)
`%M`	Minute (01–59)
`%p`	Vormittag/Nachmittag
`%S`	Sekunde (00–61)
`%U`	Woche im Jahr, beginnend mit Sonntag (00–53)
`%w`	Wochentag als Zahl (0–6)
`%W`	Woche im Jahr, beginnend mit Montag (00–53)
`%x`	Datum
`%X`	Zeit
`%y`	Jahr ohne Jahrhundert
`%Y`	Jahr mit Jahrhundert
`%Z`	Zeitzone
`%%`	%-Zeichen

Tabelle 3.5: Formatanweisungen für `strftime`

3.4.13 Makros

Makro	Header-Files	Wert	Beschreibung
BUFSIZ	stdio.h	≥ 256	Größe des Buffers für setbuf
CLOCKS_PER_SEC	time.h		Anzahl der Clock-Ticks pro Sekunde
EDOM	errno.h	> 0	Unerlaubter Eingabewert
EOF	stdio.h	< 0	End of File oder I/O-Error
ERANGE	errno.h	> 0	Over- oder Underflow
HUGE_VAL	math.h	> 0	Größter darstellbarer double-Wert
EXIT_FAILURE	stdlib.h		Argument für exit
EXIT_SUCCESS	stdlib.h		Argument für exit
MB_CUR_MAX	stdlib.h		Maximale Anzahl von Bytes in einem Multibyte Character
NULL	stddef.h, stdio.h, stdlib.h, string.h, time.h	0, 0L, oder (void*)0	Null Pointer Konstante
RAND_MAX	stdlib.h	≥ 32767	Maximaler Wert der von rand geliefert wird
SEEK_CUR	stdio.h		Aktuelle Position
SEEK_END	stdio.h		Fileende
SEEK_SET	stdio.h		Fileanfang
SIG_DFL	signal.h		Signal soll nicht abgefangen werden
SIG_ERR	signal.h		Fehler
SIG_IGN	signal.h		Signal soll ignoriert werden
SIGABRT	signal.h		Interner Programmabbruch, wie von abort ausgelöst
SIGFPE	signal.h		Arithmetische Exception (z.B. Division durch 0)
SIGILL	signal.h		Illegale Instruktion
SIGINT	signal.h		Programmabbruch durch den Benutzer
SIGSEGV	signal.h		Illegaler Speicherzugriff
SIGTERM	signal.h		Externer Programmabbruch
TMP_MAX	stdio.h	≥ 25	Anzahl der Aufrufe von tmpnam
_IONBF	stdio.h		Ungepufferte I/O
_IOLBF	stdio.h		Zeilengepufferte I/O
_IOLBF	stdio.h		Zeilengepufferte I/O
offsetof	stddef.h		Offset eines Felds in einer Struktur

Tabelle 3.6: Liste aller von Standard-Headerfiles definierten Makros

3.4.14 Typen

Typ	Header-Files	Eigenschaften	Beschreibung
`FILE`	stdio.h	—	Streams werden durch `FILE *` beschrieben
`clock_t`	time.h	arithmetisch	Clock-Ticks
`fpos_t`	stdio.h	—	Offset in einem File
`size_t`	stdio.h, stdlib.h, string.h, time.h	unsigned, ganzzahlig	Größe eines Objekts
`wchar_t`	stdlib.h	ganzzahlig	Erweiterter Character-Typ
`div_t`	stdlib.h		Ergebnis einer Division: `int quot; int rem`
`ldiv_t`	stdlib.h		Ergebnis einer Division: `long quot; long rem`
`struct lconv`	locale.h	Struktur	Enthält Informationen über die aktuelle Locale
`struct tm`	time.h	Struktur	Kalender-Zeit: alle Felder sind vom Typ `int` `tm_sec, tm_min, tm_hour, tm_mday, tm_mon` (Seit Jänner), `tm_year` (seit 1900), `tm_wday` (seit Sonntag), `tm_yday` (seit 1. Jänner), `tm_isdst` (Sommerzeit)
`time_t`	time.h	arithmetisch	Kalender-Zeit

Tabelle 3.7: Liste aller von Standard-Headerfiles definierten Typen

3.4.15 Fehlercodes

Function	Return Value	Errno	Function	Return Value	Errno
acos	impl.def.	EDOM	asin	impl.def.	EDOM
atan2	impl.def.	EDOM	atof	0	
atoi	0		atol	0	
bsearch	NULL		calloc	NULL	syscall?
clock	(clock_t)-1		cosh	HUGE_VAL	ERANGE
exp	HUGE_VAL	ERANGE	fclose	EOF	syscall
fflush	EOF	syscall	fgetc	EOF	syscall
fgetpos	$\neq 0$	impl.def.	fgets	NULL	syscall
fopen	NULL	syscall	fputc	EOF	syscall
fputs	EOF	syscall	fread	$<$ nmemb	syscall
fseek	$\neq 0$	syscall	fsetpos	$\neq 0$	impl.def.
ftell	-1L	impl.def.	fwrite	$<$ nmemb	syscall
getc	EOF	syscall	getchar	EOF	syscall
getenv	NULL		gets	NULL	syscall
gmtime	NULL		ldexp	HUGE_VAL	ERANGE
log	impl.def.	EDOM	log10	impl.def.	EDOM
malloc	NULL	syscall?	mblen	-1	
mbstowcs	(size_t)-1		mbtowc	-1	
memchr	NULL		mktime	(time_t)-1	
pow	HUGE_VAL	ERANGE	printf	<0	syscall
putc	EOF	syscall	putchar	EOF	syscall
puts	EOF	syscall	raise	$\neq 0$	
realloc	NULL	syscall?	remove	$\neq 0$	syscall
rename	$\neq 0$	syscall	scanf	EOF	syscall
setvbuf	$\neq 0$	syscall?	signal	SIG_ERR	impl.def.
sinh	$\pm$HUGE_VAL	ERANGE	sqrt	impl.def.	EDOM
strchr	NULL		strftime	0	
strrchr	NULL		strstr	NULL	
strtod	0, $\pm$HUGE_VAL	ERANGE	strtok	NULL	
strtol	0, LONG_MIN, LONG_MAX	ERANGE	strtoul	0, LONG_MAX	ERANGE
strxfrm	$\geq$ n		system	0, impl.def.	
time	(time_t)-1		tmpfile	NULL	syscall
ungetc	EOF	syscall	wcstombs	(size_t)-1	
wctomb	-1				

Tabelle 3.8: Liste aller von Library-Funktionen gelieferten Fehler-Codes

Da die von verschiedenen Libraryfunktionen gelieferten Fehlercodes historisch gewachsen, und nicht sehr einheitlich sind, sind sie in Tabelle 3.8 aufgelistet. Dabei bedeutet *impl.def.*, dass der Standard festlegt, dass es einen oder mehrere Werte geben muss (die in der Compiler-Dokumentation beschrieben sind), aber keine weiteren Garantien über den Wert abgibt. *Syscall* bedeutet, dass der Standard überhaupt keinen Wert garantiert, aber auf Unix-Systemen (und solchen, deren C-Libraries Unix ähneln, wie etwa MS-DOS) der darunterliegende Systemaufruf `errno` meistens einen brauchbaren Wert zuweist.

Kapitel 4

Programmierung unter UNIX/Linux

Auch wenn die Programmiersprache C (als ANSI-C) genormt wurde, so gibt es doch einige Punkte während der Entwicklungsphase von Programmen, die von Betriebssystem zu Betriebssystem differieren. Diese Punkte betreffen einerseits die Hilfsmittel, die bei der Programmentwicklung zur Verfügung gestellt werden und andererseits die nicht (ANSI-)standardgemäßen Funktionen, die in C-Programmen, die auf einem bestimmten Betriebssystem entwickelt werden, eingebaut werden können/sollen/müssen.

Phasen der Programmerstellung

- Die Eingabe von Programmen.
- Das Kompilieren von Programmen.
- Sourcecodeanalyse
- Das Warten und Entwickeln von Programmen.
- Das Debuggen von Programmen.
- Der Aufruf von Programmen.

UNIX-spezifische Funktionen in C-Programmen

- UNIX Systemcalls.
- UNIX Bibliotheksfunktionen.

4.1 Phasen der Programmerstellung

UNIX und Linux stellen eine sehr leistungsfähige Programmentwicklungsumgebung zur Verfügung, die vor allem darauf aufbaut, dass es für den Programmierer sehr viele *Tools* gibt, die er bei der Entwicklung und Wartung von Programmen verwenden kann. Aufgrund dieses Konzepts wird aber auch

klar, dass die Entwicklungsumgebung eher auf den erfahrenen Programmierer zugeschnitten ist und nicht so sehr auf den 'Einsteiger', der z.B. mit einer graphischen, menügesteuerten Entwicklungsumgebung am Anfang viel leichter zurechtkommen wird. Die Effizienz der Programmentwicklung unter Unices hängt zu einem nicht unwesentlichen Teil von der genauen Kenntnis der entsprechenden Tools ab. Einige dieser Tools wurden bereits im allgemeinen Kapitel zu UNIX beschrieben, diejenigen Tools, die für den C-Programmierer von besonderer Bedeutung sind, werden im Folgenden beschrieben.

4.1.1 Die Eingabe von Programmen

Im Grunde kann ein C-Programm mit einem beliebigen Texteditor erstellt und bearbeitet werden.

Im Folgenden werden die Editoren `emacs`, `pico`, `joe` und `vi` vorgestellt, die auf fast allen Installationen verfügbar sind. Auf den Editor `emacs` wird genauer eingegangen.

Editoren Kommandoübersicht

In der folgenden Tabelle bedeutet `C-`<Buchstabe>, dass das Kommando durch gleichzeitiges Drücken der Tasten Control und <Buchstabe> angewählt wird. `C-q` bedeutet also, dass die Tasten Control und `q` gemeinsam gedrückt werden. Bei `C-k e` werden zuerst Control und `k` gemeinsam gedrückt, dann (nach Loslassen der beiden Tasten) die Taste `e`. Der Eintrag '-' bei einer Aktion bedeutet, dass diese vom betreffenden Editor nicht unterstützt wird. Die Kommandos, die den Editor `vi` betreffen, setzen voraus, dass sich der `vi` im *Kommandomodus* befindet (siehe unten).

Aktion	`emacs`	`joe`	`pico`	`vi`
Online Hilfe	C-h t	C-k h	C-g	-
Datei öffnen	C-x C-f	C-k e	C-r	:r
Dateibrowser	C-x d	-	C-r C-t	-
Datei speichern	C-x C-s	C-k d	C-o (C-t)	:w<RETURN>
Speichern und Beenden	C-x C-c	C-k x	C-x	:x<RETURN>
Text suchen	C-s	C-k f	C-w	/
Block markieren Anfang	Maus links	C-k b	C-^	-
Block markieren Ende	Maus rechts	C-k k	C-k (F9)	-
Block kopieren	C-y	C-k c	C-u (F10)	-

Tabelle 4.1: Editoren - Kommandos im Vergleich

Beim Editor `emacs` können all diese Funktionen auch über Menüs erreicht werden.

Der Editor `joe`

Mit der Tastenkombination C-k h kann ein Fenster mit den Tastenkombinationen für die Kommandos von `joe` ein- und ausgeblendet werden.

Einige nützliche Kommandozeilenparameter für den Editor `joe`:

+ <**Zeilennummer**> Der Editor lädt die angegebene Datei und positioniert den Cursor in der Zeile <Zeilennummer>. Eingabe in der Kommandozeile:

```
joe +25 test.c
```

-linums Vor jeder Zeile wird die Zeilennummer eingeblendet.

Der Editor `pico`

`pico` ist der Editor, der beim Mailprogramm `pine` verwendet wird. Die wichtigsten Kommandos werden am unteren Fensterrand stets angezeigt und sind durch die Tastenkombination Control-Taste und entsprechende Buchstabentaste anwählbar.

Nützliche Kommandozeilenparameter:

-w Wordwrap ausschalten. Diese Option ist beim Editieren von Programmcode interessant. Es wird verhindert, dass `pico` automatisch die Zeilen umbricht.

+ <**Zeilennummer**> Zeilennummer anspringen. Bei der Eingabe

```
pico +25 test.c
```

wird die Datei `test.c` geladen und der Cursor in die Zeile 25 positioniert.

-x Die untersten zwei Zeilen mit den Tastenkommandos werden ausgeblendet, so dass mehr Platz zur Anzeige des Textes zur Verfügung steht.

-m aktiviert die Mausunterstützung wenn `pico` in einem XTerminal läuft.

Der Editor `vi`

`vi` ist einer der gängigsten Editoren in der Unix-Welt. Der Editor ist sehr mächtig, die Bedienung jedoch gewöhnungsbedürftig.

Der Editor kann sich in einem von vier verschiedenen Modi befinden:

- Im *Kommandomodus* können Sie Kommandos, die auf bereits eingegebenem Text operieren (Bewegen des Cursors, Kopieren von Text, Text Suchen und Ersetzen, ...), ausführen. Mit den Kommandos `a`, `i`, `o` oder `O` gelangen Sie in den *Eingabemodus*. Durch Eingabe eines Doppelpunktes kommt man in den *Kommandozeilenmodus*.
- Im *Eingabemodus* können Sie Text eingeben. Sie bleiben solange in diesem Modus, bis Sie <`ESC`> drücken.
- Im *Kommandozeilenmodus* können Aktionen wie Suchanfragen, Speichern, Beenden, etc. durchgeführt werden.

- Mittels Eingabe von `! <Kommando>` gelangt man vom *Kommandozeilenmodus* schließlich in den vierten Modus, den *Shell-Escape Modus* zum Ausführen von Shell-Befehlen.

Weitere Informationen erhält man über die Manualpages oder durch Eingabe von `vitutor`. Dabei wird ein Lernprogramm für `vi` aufgerufen, das eine ausführliche Beschreibung des `vi` enthält.

Der Editor `emacs`

`emacs` ist ein leistungsfähiger Editor, der als *Freie Software* unter der GNU General Public License erhältlich erhältlich ist. Die X-Windows Version bietet eine komfortable Benutzeroberfläche mit Menüs und Scrollbars, die einen schnellen Einstieg in die Arbeit mit dem Programm ermöglicht. Bei den einzelnen Menüpunkten sind auch die Tastaturkürzel für die Kommandos aufgeführt.

Tastatursteuerung des Editors `emacs`

`Emacs`-Befehle beinhalten im Allgemeinen die CONTROL-Taste sowie die META-Taste. Folgende Abkürzungen werden verwendet:

C-<Taste> bedeutet, dass die CONTROL-Taste gedrückt sein muss, während man die Taste <Taste> drückt. Beispiel: `C-f` CONTROL-Taste gedrückt halten und dann die `f`-Taste drücken.

M-<Taste> bedeutet, dass die linke Alt-Function-Taste gedrückt wird und danach die Taste <Taste> eingegeben.

Im Text kann wie gewohnt mit den Cursortasten navigiert und mit den Tasten `Bild auf` und `Bild ab` Bildschirmweise gescrollt werden.

Das `emacs`-Tutorial (Im Menü *Help*) beeinhaltet Informationen über die wichtigsten Tastaturkommandos. Der Anhang enthält eine Übersicht über die gebräuchlichsten `emacs`-Kommandos.

Nützliche Tastaturkommandos

Abbrechen von Kommandos Jedes Kommando kann mit `C-g` (Taste `Ctrl` und Taste `g` gemeinsam drücken) abgebrochen werden. Sollte man also einmal ein falsches Kommando starten, das dann unbedingt eine Eingabe in der *Kommandoeingabezeile* erwartet, kann man die Sache mit `C-g` beenden.

Inkrementelle Suche `C-s` sucht ausgehend von der aktuellen Cursorposition die in der *Kommandoeingabezeile* (Unterste Zeile im Programmfenster) eingegebene Buchstabenfolge.

Split-Fenster ausschalten `C-x 1`, um nur einen Buffer in einem Fenster anzuzeigen.

Zeilenanzeige Die Anzeige der Zeilennummer in der Statuszeile kann mit `M-x line-number-mode` ein- bzw. ausgeschaltet werden.

`emacs` Menüstruktur

Die über Menü adressierbaren Kommandos decken alle zum Arbeiten erforderlichen Funktionen ab. Benötigt eine Funktion noch zusätzliche Parameter, so werden diese in der *Kommandoeingabezeile* (unterste Zeile im Programmfenster, unterhalb der invers dargestellten *Statuszeile*) abgefragt.

Das Menü Buffers

Eine in `emacs` geladene Datei wird mit einem Buffer assoziiert. Text kann zwischen verschiedenen Buffers kopiert werden.

Je nachdem ob ein oder mehrere *Frames* (Programmfenster) geöffnet sind, präsentiert sich das Menü Buffers entweder als Liste aller aktuell geladenen Dateien oder als Menü mit zwei Untermenüs:

Buffers Die Liste der aktuell geladenen Dateien. Es entspricht dem Hauptmenü Buffers, wenn nur ein Programmfenster offen ist. Hier kann man selektieren, welcher Buffer im Programmfenster zum Editieren angezeigt werden soll

Frames Eine Liste der geöffneten Programmfenster von emacs. Hier kann zwischen den Fenstern gewechselt werden.

Öffnen und Sichern von Dateien

Im Menü File finden sich die Kommandos *Open File ...* und *Save Buffer*:

Open File... In der *Kommandoeingabezeile* (Unterste Zeile im Fenster) ist der Dateiname bzw. Suchpfad (Unix-Pfad!!) einzugeben. Bei Leereingabe wird der Verzeichniseditor *DirEd* geladen, der das Browsen durch den Verzeichnisbaum und die Selektion einer Datei zum Laden erlaubt. Die Selektion erfolgt stets mit der mittleren Maustaste. Die Datei wird unmittelbar geladen und im aktuellen Fenster angezeigt.

Open Directory... Wie *Open File...* nur dass sofort *DirEd* geladen wird, mit dem dann eine Datei selektiert und geladen werden kann.

Save Buffer Der Buffer, in dem gerade gearbeitet wurde, wird auf Platte zurückgeschrieben. Wurde noch kein Dateiname vergeben, wird in der *Kommandoeingabezeile* ein Dateiname verlangt.

Kill Buffer Der Buffer, in dem gerade gearbeitet wurde, wird gelöscht und aus der Liste der aktuell geladenen Dateien gelöscht.

Arbeiten mit mehreren Fenstern

Man kann Dateien in mehreren Programmfenstern (*Frames*) anzeigen lassen, was z.B. beim Zusammenkopieren von Text aus mehreren Quelldateien angenehm ist.

Im Menü *File* finden sich die Kommandos zum Öffnen und Schließen von Programmfenstern:

Make New Frame Ein neues Programmfenster wird geöffnet und der gerade bearbeitete Buffer in diesem angezeigt.

Delete Frame Das Programmfenster, in dem gerade gearbeitet wird, wird geschlossen.

Kopieren von Text

Das Menü *Edit* enthält Kommandos zum Kopieren von Text.

Für *Cut*, *Copy* und *Clear* wird der Textblock, auf dem die Operation ausgeführt werden soll, mit der Maus markiert. Das Markieren erfolgt entweder durch Ziehen der Maus bei gedrückter linker Taste oder durch Drücken der linken Taste am Blockbeginn und Drücken der rechten Taste am Blockende.

Select and Paste zeigt eine Liste mit den Textblöcken, die durch Markieren mit der Maus angewählt wurden. Aus dieser kann mit der Maus ein Textblock zum Einfügen gewählt werden.

Online Hilfe

Im Menü *Help* von `emacs` gibt es den Punkt *Emacs Tutorial*, das einen Überblick über die gebräuchlichsten Kommandos und eine Einführung in die Tastatursteuerung des Editors `emacs` gibt.

Kompilieren

Emacs kann auch so konfiguriert werden, dass Makefiles direkt vom Editor aus aufgerufen werden können, wie bei gängigen integrierten Entwicklungsumgebungen üblich. In dem Fall gibt es, wenn eine C-Sourcedatei geladen ist, ein Menü mit den Einträgen *Build* und *Clean*, die die Einträge `all` und `clean` des Makefiles aufrufen. Zu beachten ist, dass sich das Makefile im selben Verzeichnis wie der aktuell bearbeitete Buffer befindet.

Bearbeiten von C-Quelltext

Wenn entsprechend konfiguriert, unterstützt Emacs auch eine Reihe von nützlichen Kommandos, die einfaches Navigieren im C-Text (*Forward/Backward Statement/Conditional*), und Auskommentieren (*Comment Out Region*) und Einrücken von Quelltext-Blöcken (*Indent Expression*) ermöglichen.

Konsistentes Einrücken eines markierten Blockes C-Quelltext kann durch das Kommando *Indent Region* erreicht werden. Dafür sind `M-C-\` gleichzeitig zu drücken (`Alt_Function-Ctrl-\`).

Zum Einrücken einer C-Expression wird der Cursor auf die öffnende Klammer der Expression gesetzt und dann *Indent Expression* entweder per Menü (siehe oben) oder mit dem Kommando `M-C-q` aufgerufen.

Das Einrücken einer Zeile erfolgt durch das Drücken von `TAB` an irgendeiner Position in der Zeile.

4.1.2 Das Kompilieren von Programmen

Um ein Programm lauffähig zu machen, ist es notwendig, den Quellcode zu *kompilieren* und zu *linken*.

Die meisten verfügbaren Compiler unterstützen einen Modus, in dem sie Programme streng nach dem ANSI-C Standard [C89] ('*traditional*') übersetzen. Im Allgemeinen übersetzt ein Compiler sowohl ANSI-C Programme als auch C Programme, die nicht dem ANSI-C Standard gehorchen.

Um ein Modul zu kompilieren, wird der Compiler mit folgenden Optionen aufgerufen:

```
gcc -ansi -pedantic -Wall -g -c <Modulname>
```

Der Modulname muss der Name einer Datei sein, die C-Sourcecode enthält und er muss auf `.c` enden. Nach dem erfolgreichen Kompilieren eines Moduls wird eine Datei erzeugt, die denselben Namen wie das entsprechende Modul trägt, aber an Stelle von `.c` mit `.o` endet. Diese Datei wird das *Object-File* des Moduls genannt.

Die Optionen beim Aufruf des Compilers bedeuten Folgendes:

-ansi Teilt dem Compiler mit, Programme im ANSI-Standard zu akzeptieren.

-pedantic Das Programm wird daraufhin überprüft, ob es Konstrukte enthält, die entweder zu einem unspezifizierten Verhalten des Programmes führen oder Stellen enthält die eine Portierung auf ein anderes System verhindern. In diesen Fällen liefert der Compiler eine *Warning*- Nachricht. Ein Programm sollte so geschrieben sein, dass es nach obigem Aufruf keine Warnings produziert.

-Wall Diese Option aktiviert eine Vielzahl von Warning-Nachrichten, die bei sauberer Programmierung zu vermeiden sind. Es gibt aber noch eine weitere Klasse von Warning-Nachrichten, die mit dieser Option *nicht* aktiviert werden, da diese Warnings auch in sauber strukturierten Programmen auftreten können.

-g Es wird Code zum symbolischen Debuggen erzeugt. Diese Option wird benötigt, wenn Sie das Programm mit `dbx` (oder einem anderen Debugger) debuggen möchten.

-c Es wird nur ein Modul eines Programmes und kein komplettes Programm kompiliert. Diese Option können Sie nur dann weglassen, wenn entweder das Programm nur aus einem Modul besteht oder Sie in der Kommandozeile die Namen aller Module, aus denen das Programm besteht, angeben.

Weitere Optionen, die beim Kompilieren eines Moduls oft nützlich sind:

-O Es wird optimierter Programmcode erzeugt. Das ausführbare Programm wird schneller, allerdings dauert auch das Erzeugen des ausführbaren Programms länger.

-I <Include-Directory> Als Argument dieser Option wird ein Directory angegeben, in dem sich Header-Dateien befinden. Befindet sich in einem Programm eine `#include` Anweisung, so wird dieses Directory vor den Directories, in denen normalerweise nach Header-Dateien gesucht wird, durchsucht.

-D<**Macro-Definition**> Ein Makro wird definiert. Eine Makrodefinition wird dann beim Aufruf des Programmes angegeben, wenn, abhängig davon, ob ein bestimmtes Makro definiert ist, unterschiedlicher Code erzeugt werden soll. Weitverbreitet ist die Benutzung des `DEBUG`-Makros. Im Programm befinden sich mehrere Sequenzen folgender Form:

```
#ifdef DEBUG
/* print debugging messages */
...
#endif DEBUG
```

Wird das Programm nun mit `-DDEBUG` kompiliert, so werden diese Debuggingausgaben aktiviert, ansonsten nicht.

-E Es wird nur der Präprozessor aufgerufen und die Ausgabe des Präprozessors auf die Standardausgabe geschrieben. Das Programm wird nicht kompiliert. In bestimmten Fällen kann dies für die Fehlersuche nützlich sein.

`gcc` dient nicht nur zum Kompilieren von Modulen, sondern auch zum Linken von vorher kompilierten Modulen zu einem ausführbaren Programm. Die Syntax dafür sieht folgendermaßen aus:

```
gcc -g -o <Programmname> <Object-Files> <Libraries>
```

Dabei bedeuten die Parameter Folgendes:

-o <**Program-Name**> Der Name des exekutierbaren Programms wird angegeben. Fehlt diese Option, so wird der Standardname `a.out` vergeben. Die `-o` Option sollte aber beim Linken immer angegeben werden.

<**Object-Files**> Hier müssen die Namen der Object-Files aller Module, aus denen das Programm besteht, angegeben werden.

<**Libraries**> Ruft ein Programm eine Funktion auf, die nicht in einem Modul des Programms selbst definiert ist, so muss der Code dieser Funktion aus einer Library genommen werden. Normalerweise betrifft dies den Programmierer nicht, da es eine Standardlibrary (`-lc`) gibt, in der der Compiler beim Linken selbsttätig nach undefinierten Funktionen sucht. In dieser Library sind fast alle normalerweise benötigten Funktionen enthalten. Ist eine Funktion nicht in dieser Library enthalten, so ist das im Allgemeinen in der Manual-Page der Funktion angegeben. Ein Beispiel, in dem eine Library explizit angegeben werden muss, ist die Verwendung von komplexeren Funktionen mit Fließkommazahlen. In diesem Fall muss die Mathematik-Library `-lm` spezifiziert werden.

4.1.3 Statische Sourcecodeanalyse mit `splint`

Als Hilfsmittel zur Fehlervermeidung stehen dem C-Programmierer neben den Compilerwarnings auch eine Vielzahl von Programmen zur statischen Sourcecodeanalyse zur Verfügung. Solche Programme analysieren C-Quellcode und weisen auf wahrscheinliche Programmierfehler hin, wenn

zum Beispiel typische Konstrukte der Programmiersprache C in unüblicher Art und Weise angewendet wurden (siehe dazu auch Kapitel 3.3 Tücken im Umgang mit C). Zusätzlich werden diverse Annotationen in C-Kommentaren interpretiert, um so zwischen der absichtlichen und der versehentlichen Verwendung eines Konstrukts zu unterscheiden und nur bei letzterer eine Fehlerwarnung auszugeben.

Im Folgenden wird die Anwendung eines solchen Tools auf ein mit typischen Fehlern behaftetes Programmbeispiel gezeigt. Zur Analyse wurde das Programm `splint` verwendet, welches eine indirekte Weiterentwicklung des 1979 erschienen UNIX-Tools `lint` ist. Splint ist freie Software die unter der GNU General Public License veröffentlicht ist.

Das folgende Programm sollte eigentlich in einer Schleife zeichenweise von der Eingabe lesen und für jeden Zeilenwechsel eine entsprechende Meldung ausgeben. Das Zeichen 'x' soll das Programm beenden. Alle anderen Zeichen werden direkt ausgegeben.

Das Programm ist allerdings mit typischen Fehlern beim Programmieren in C behaftet: die Variable c wird gelesen, ohne vorher initialisiert zu sein, nach dem `while`-Statement ist ein Strichpunkt zuviel, der eine Endlosschleife bedingt, bei der Zuweisung von `getchar` findet eine implizite Typkonversion von int nach char statt, die `if`-Bedingung enthält eine Zuweisung anstatt eines Vergleichs und im `switch`-Konstrukt fehlt ein `break`:

```
int main () {
  char c;
  while (c != 'x');
  {
    c = getchar ();
    if (c = 'x') return 0;
    switch (c)
      {
      case '\n':
      case '\r':
        printf ("Zeilenwechsel\n");
      default:
        printf ("%c",c);
      }
  }
  return 0;
}
```

Der Quellcode dieses Programm kann durch Aufruf von

```
splint <Dateiname.c>
```

auf wahrscheinliche Fehler und Schwachstellen hin analysiert werden. Während ein typischer C-Compiler wie gcc hier nur vor der Zuweisung in der if-Anweisung warnt, findet splint in dem Programm sechs verdächtige Codestellen (die Ausgabe wurde der Übersichtlichkeit wegen um erklärende Kommentare gekürzt):

```
Variable c used before definition
Suspected infinite loop.  No value used in loop test (c) is
Assignment of int to char: c = getchar()
```

```
Test expression for if is assignment expression: c = 'x'
Test expression for if not boolean, type char: c = 'x'
Fall through case (no preceding break)
```

Tatsächlich korrespondieren die beanstandeten Punkte im Quellcode mit den Programmierfehlern. Zum Vergleich dazu das korrekte Programm, welches von splint nicht beanstandet wird. Der Kommentar /*@fallthrough@*/ weist splint darauf hin, dass das Weglassen des break-Statements an dieser Stelle bewusst erfolgt.

```
int main () {
  char c;
  do
  {
    c = (char) getchar ();
    if (c == 'x') return 0;

    switch (c)
      {
      case '\n':              /*@fallthrough@*/
      case '\r':
        printf ("Zeilenwechsel\n");
        break;
      default:
        printf ("%c",c);
      }
  }
  while (c != 'x');
  return 0;
}
```

4.1.4 Das Entwickeln und Warten von Programmen (`make`)

Es gibt in UNIX ein sehr mächtiges Werkzeug zum Entwickeln und Warten von Programmen, das Kommando `make`. Dieses Kommando liest in einer Datei, die standardmäßig `Makefile` heißt, Informationen darüber, wie Zieldaten (z. B. ausführbare Programme) aus Quelldateien erzeugt werden können, wobei die dazu spezifizierten Anweisungen nur dann ausgeführt werden, wenn die Zieldatei älter als eine der Quelldateien ist. Somit ist `make` ein sehr flexibles und allgemeines Werkzeug, das jedoch hier speziell im Einsatz für die Erstellung von ausführbaren Programmen aus Quelldateien beschrieben wird. Das heißt, in unserem Fall liest `make` aus einer Datei Informationen darüber, wie aus Quelldateien ein ausführbares Programm erzeugt werden kann. Diese Informationen werden dazu benutzt, um alle Dateien (Object-Files, ausführbares Programm, ...), die automatisch erstellt werden können und die nicht mehr up-to-date sind, neu zu generieren (kompilieren, linken).

Dazu ein Beispiel:

Ein Programm besteht aus den Modulen `x.c`, `y.c` und `z.c` mit den entsprechenden Header-Dateien `x.h`, `y.h` und `z.h`. Dabei wird jede Header-Datei vom entsprechenden C-File inkludiert, und außerdem wird `y.h` von `x.c` und `z.h` von `y.c` inkludiert. Das exekutierbare Programm heißt `xyz`.

Wird nun die Datei `y.c` geändert, so muss das Object-File zu `y.c` (`y.o`) neu erzeugt werden, und ebenso das exekutierbare Programm `xyz` aus den Object-Files `x.o`, `y.o` und `z.o`.

Wird z.B. die Datei `y.h` geändert, so müssen alle Module, die diese Datei inkludieren (`y.c` und `x.c`) neu kompiliert werden und außerdem müssen die Object-Files neu gelinkt werden.

Wie sieht nun ein Makefile aus, das ein solches Programm beschreibt?

Ein Makefile besteht aus einer Liste von Einträgen, die beschreiben, wie jeweils eine Datei neu erzeugt werden kann. Außerdem wird in jedem Eintrag noch angegeben, nach der Änderung welcher Dateien die angegebene Datei neu erzeugt werden muss.

Ein Eintrag sieht folgendermaßen aus:

```
Target : Dependencies
<tab> Kommando
...
```

`Target` ist entweder der Name einer Datei, die durch die folgenden Kommandos neu erzeugt wird, oder es ist ein Label.

`Dependencies` ist eine Liste, die Dateinamen und Namen von Targets enthält. Diese Liste hat folgende Bedeutung: Zuerst werden die Regeln für alle Targets, die in der Liste der Dependencies enthalten sind, ausgeführt. Anschließend wird überprüft, ob mindestens eine der in der Liste der Dependencies angeführten Dateien (oder Targets) geändert wurde, nachdem `Target` das letzte Mal erzeugt wurde. Ist dies der Fall, so werden alle Kommandos, die sich nach der Zeile mit dem Target befinden, ausgeführt.

ACHTUNG! Jede Zeile mit einem Kommando *muss* mit einem Tabulator beginnen! Die erste Zeile nach `Target`, die nicht mit einem Tabulator beginnt, gibt das Ende der Liste der Kommandos an. Ein häufiger Fehler bei Makefiles ist die Verwendung von Leerzeichen statt Tabulatoren.

Ist ein Target kein Dateiname, so spricht man von einem so genannten *Label*. In diesem Fall wird genauso vorgegangen wie bei einer Datei, mit dem Unterschied, dass die Kommandos, die dem Target folgen, auf jeden Fall ausgeführt werden. Die Verwendung von Labels als Targets dient vor allem dazu, um mit dem `make` Kommando verschiedene Aktionen, die im Makefile beschrieben sind, auszuführen.

Wird das Kommando `make` ohne Parameter aufgerufen, so wird versucht, das erste Target im entsprechenden Makefile zu bilden. Es kann aber auch ein Target als Parameter für `make` angegeben werden. In diesem Fall wird versucht, dieses Target zu bilden.

Es ist Konvention, dass folgende zwei Labels in einem Makefile angegeben sind:

all: Dieses Label dient dazu, um das ganze Programm (bzw. Projekt) zu erstellen. Als Dependencies von `all:` sollen alle exekutierbaren Dateien angegeben werden, die zu dem Programmpaket

gehören. `all:` muss immer das erste Target in einem Makefile sein, damit bei Eingabe von `make` ohne Parameter immer das gesamte Programmpaket neu erstellt wird.

clean: Dieses Label dient dazu, um alle Dateien, die aus irgend einer anderen Datei automatisch erzeugt werden können, zu löschen. Dies sind normalerweise alle Object-Files und die exekutierbaren Programme.

Ein Versuch, ein einfaches Makefile zu schreiben, sieht wie folgt aus:

```
##
##  Project:       xyz
##  File:          Makefile
##  Version:       1.1
##

all: xyz

xyz: x.o y.o z.o
        gcc -o xyz x.o y.o z.o

x.o: x.c x.h y.h
        gcc -ansi -pedantic -Wall -g -c x.c

y.o: y.c y.h z.h
        gcc -ansi -pedantic -Wall -g -c y.c

z.o: z.c z.h
        gcc -ansi -pedantic -Wall -g -c z.c

clean:
        rm -f x.o y.o z.o xyz
```

Noch zwei Anmerkungen zu dem Makefile:

- Ist eine Zeile länger als die Bildschirmbreite, so kann man sie auf mehrere Zeilen aufteilen, wobei jede Zeile bis auf die letzte mit einem Backslash (\) enden muss.
- Das Kommentarzeichen in Makefiles ist '#'.

Dieses einfache Makefile enthält einige fehleranfällige Konstrukte. So müssen bei der Regel für `xyz` die Dateien `x.o`, `y.o` und `z.o` zweimal angeführt werden: einmal unter `Dependencies` und nochmals unter dem `Kommando`.

```
xyz: x.o y.o z.o
        gcc -o xyz x.o y.o z.o
```

Dies kann z.B. bei Hinzufügen von neuen Dateien dazu führen, dass eine Datei bei einer Auflistung vergessen wird. Eine bessere Lösung dafür ist die Verwendung von Variablen. Diese können mit

einer Zeichenfolge belegt und an einer beliebigen Stelle substituiert werden. In Verbindung mit der `xyz` Regel aus dem Beispiel erhält man folgendes:

```
OBJS = x.o y.o z.o

xyz: $(OBJS)
        gcc -o xyz $(OBJS)
```

Variablennamen sind case-sensitive und können mit `$` gefolgt vom Namen in Klammern durch die definierte Zeichenfolge substituiert werden. Es dürfen weder `:`, `#`, = Zeichen, noch führende oder nachfolgende Leerzeichen bei Namen vorkommen.

Es gibt zwei verschiedene Möglichkeiten der Variablenzuweisung, die sich in der Auflösung der Substitution unterscheiden.

Die *rekursiv aufgelösten Variablen* werden mit dem Zeichen = zugewiesen. Diese Variablen werden bei der Substitution so lange rekursiv substituiert, bis kein weiterer Variablenname enthalten ist.

Dazu ein Beispiel:

```
eins = $(zwei)
zwei = $(drei)
drei = text
```

Bei Auflösung der Variable `eins` erhält man die Zeichenkette „text“, da die Variable `eins` die Auflösung der Variable `zwei` enthält. Welche wiederum die Auflösung von Variable `drei` bewirkt, die letztendlich die Zeichenkette repräsentiert.

Die zweite Art von Variablen sind die *einfach aufgelösten Variablen*. Die Zuweisung erfolgt mit den beiden Zeichen `:=`. Dabei wird die Zeichenkette *einmalig* aufgelöst und der Variable zugewiesen. Solche Variablen beinhalten *keine* Referenzen zu anderen Variablen, sondern nur ihre Inhalte zum Zeitpunkt der Zuweisung.

Beispiel:

```
eins := text
zwei := $(eins) ausgabe
eins := danach
```

ist gleich zu

```
zwei := text ausgabe
eins := danach
```

Diese beiden Abfolgen haben am Ende die gleichen Variableninhalte, da beim ersten Block Variable `zwei` keine Referenz auf Variable `eins` enhält, sondern nur deren Inhalt zu dem Zuweisungszeitpunkt substituiert wird.

Ein weiteres Problem könnte die `clean` Regel auslösen. Wenn eine Datei mit dem gleichen Namen im Verzeichnis existiert, dann wird diese Regel nicht mehr ausgeführt. Da es für `clean` keine Abhängigkeit gibt, wird immer angenommen die Datei ist aktuell und das Kommando wird nie ausgeführt. Aus diesem Grund gibt es die Möglichkeit sogenannte *Phony Targets* explizit zu deklarieren. Diese Targets beziehen sich nicht auf eine Datei und werden auch ausgeführt wenn eine Datei mit selben Namen existiert bzw. aktuell ist.

```
.PHONY: clean

clean:
        rm -f x.o y.o z.o xyz
```

Built-in rules

Das make-Kommando beinhaltet im Allgemeinen von vornherein eine Menge von fertigen Regeln, die sogenannten *built-in rules*. Diese Regeln werden von Make angewendet, wenn es keine expliziten Regeln zum Erstellen eines Targets gibt.

Die *built-in rules* können mittels des Befehls

```
make -p
```

aufgelistet werden.

Um unerwünschte Effekte durch *built-in rules* zu vermeiden, sollte man die gewünschten *built-in rules* explizit mittels `.SUFFIXES:` auswählen, z.B.:

```
# use only the following built-in rules
.SUFFIXES: .c .cpp
```

Wird das Schlüsselwort `.SUFFIXES:` ohne Argumente angegeben, so werden sämtliche eingebauten Regeln außer Kraft gesetzt.

Fehlt die Angabe von `.SUFFIXES:` komplett, so sind alle eingebauten Regeln per Voreinstellung aktiv. Dies ist eine potentielle Fehlerquelle, wenn man zum Beispiel ein Listingsfile mit der Endung `.l` erzeugt, da `make` eine eingebaute Regel für Dateien des Programmes `lex` besitzt. `lex` ist ein Programm, mit dem Quellcode für andere Programme (Scanner) erzeugt werden kann. Als Eingabedatei dient hierzu eine Datei mit der Endung `.l`, als Ausgabedatei erzeugt `lex` eine Datei mit der Endung `.c`. Somit würde bei Vorhandensein einer `.l`-Datei die gleichnahmige `.c`-Datei überschrieben werden!

Darum sollte `.SUFFIXES:` immer angegeben werden!

Bedingte Ausführung von Befehlen:

In einem Makefile können auch Konditionale auftreten. Die Syntax dafür sieht folgenderweise aus:

```
Konditional-Anweisung
    Textblock wenn erfüllt
else
    Textblock wenn nicht erfüllt
endif
```

Es gibt vier mögliche `Konditional-Anweisungen`:

`ifeq (`*`arg1`*`, `*`arg2`*`)`
: Expandiert die beiden Argumente und vergleicht diese. Bei Gleichheit wird der `erfüllt` Textblock und bei Ungleichheit der `nicht erfüllt` Textblock ausgeführt.

`ifneq (`*`arg1`*`, `*`arg2`*`)`
: Expandiert die beiden Argumente und vergleicht diese. Bei Ungleichheit wird der `erfüllt` Textblock und bei Gleichheit der `nicht erfüllt` Textblock ausgeführt.

`ifdef `*`Variablenname`*
: Wenn der Inhalt der Variable `Variablenname` nicht leer ist, wird der `erfüllt` Textblock und ansonsten der `nicht erfüllt` Textblock ausgeführt.

`ifndef `*`Variablenname`*
: Wenn der Inhalt der Variable `Variablenname` leer ist, wird der `erfüllt` Textblock und ansonsten der `nicht erfüllt` Textblock ausgeführt.

Am Beginn der `Konditional-Anweisung`-Zeile sind zusätzliche Leerzeichen erlaubt. Tabulatorzeichen hingegen sind am Beginn der Zeile verboten (Es könnte ansonsten zur Verwechslung mit Kommandozeilen kommen).

Die foreach-Funktion:

```
foreach(var,list,command)
```

Die Funktion `foreach` mit den Parametern (`var`,`list`,`command`) führt mehrfach Substitutionen an einem Textblock durch. Dabei werden zuerst die ersten zwei Parameter `var` und `list` aufgelöst (`command` wird vorerst nicht aufgelöst). `var` durchläuft nun jedes Wort in `list`, wobei jedesmal `command` neu aufgelöst wird.

Einige Funktionen zur Stringmanipulation:

Es gibt weiters die Möglichkeit mit speziellen Funktionen Textmanipulationen durchzuführen. Mit der Syntax

```
$(Funktionsname Argumente)
```

wird die Funktion `Funktionsname` aufgerufen und der Funktionsaufruf (ähnlich einer Variablensubstitution) durch den Rückgabewert ersetzt.
Die `Argumente` bilden die Eingabeparameter der Funktion und können durch Leer- oder Tablulatorzeichen vom Funktionsnamen getrennt werden. Mehrere Argumente werden durch Beistriche getrennt. Argumente können weitere Funktionsaufrufe oder Variablennamen sein und werden in der Reihenfolge ihres Auftretens aufgelöst.

`subst` *`from,to,text`*
: Ersetzt in `text` alle Vorkommen von `from` durch `to`.

`patsubst` *`pattern,replacement,text...`*
: Sucht nach Wörtern in `text`, die durch Whitespaces getrennt sind und `pattern` entsprechen. Gefundene Wörter werden durch `replacement` ersetzt. Hierbei kann `pattern` Wildcards in Form von einem `%`-Zeichen enthalten. Dieses Zeichen kann einer beliebigen Anzahl von Zeichen gleich sein. Wenn `replacement` ebenfalls ein `%`-Zeichen enthält, wird der Teil, der in `pattern` mit dem Zeichen übereinstimmt, eingesetzt. Es wird jeweils nur das erste Vorkommen von `%` als Wildcard gewertet. Jedes weiter Auftreten wird als normales Zeichen interpretiert. `%`-Zeichen können mit einem vorangestellten Backslash ('\') *escaped* werden.

  Eine einfachere Methode mit gleichem Ergebnis ist die Verwendung von *Substitutions Referenzen*:

  `$(`*`var`*`:`*`pattern=replacement`*`)`

  ist gleich bedeutend mit

  `$(patsubst` *`%pattern,%replacement,($var)`*`)`

`strip` *`string`*
: Entfernt alle führenden und nachfolgenden Whitespaces von `string` und ersetzt innere Sequenzen von Whitespaces durch ein einzelnes Leerzeichen.

`filter` *`pattern...,text...`*
: Überprüft, durch Whitespace getrennte, Wörter in `text`, ob sie einem der Wörter von `pattern` entsprechen. Alle Wörter die nicht übereinstimmen werden entfernt. Dabei können die `pattern` Argumente wieder mit dem `%`, wie bei `patsubst`, versehen werden.

`filter-out` *`pattern...,text...`*
: Ähnlich wie `filter`, nur werden dabei die `pattern` entsprechenden Wörter aus `text` entfernt.

Einige Funktionen zur Dateinamenbehandlung:

`dir` *`names...`*
: Extrahiert das Verzeichnis von den Dateien die in `names` stehen (bis zum letzten Slash). Wenn ein Dateiname keinen Slash enthält wird './' ausgegeben.

`suffix` *`names...`*
: Extrahiert die Erweiterung von Dateinamen aus `names`. Hat die Datei keine Erweiterung wird ein Leerstring ausgegeben.

`wildcard` *`pattern`*
: Gibt Dateinamen aus die `pattern` entsprechen. `pattern` kann dabei die typischen Wildcard Zeichen (wie z.B. in der Shell verwendet) enthalten. Das Ergebnis wird durch Leerzeichen getrennt.

`addsuffix` *`suffix`*,*names...*
: `Names` entspricht einer durch Leerzeichen getrennten Liste von Namen. Das Ergebnis von `addsuffix` sind alle namen um `suffix` erweitert mit einem einzigen Leerzeichen getrennt.

Funktionen zur Fehlerbehandlung:

`error` *`text...`*
: Der Aufruf dieser Funktion erzeugt einen fatalen Fehler und make wird beendet. Vor dem Abbruch des Programms wird `text` noch aufgelöst und ausgegeben.

`warning` *`text...`*
: Diese Funktion ist ähnlich der `error` Funktion, nur das make nicht beendet wird. Es wird `text` aufgelöst, ausgegeben und das Makefile weiter abgearbeitet. An die Stelle des Funktionsaufrufs wird eine leere Zeichenkette gesetzt.

Abschließend noch ein Beispiel eines komplexeren Makefile mit Kommentaren und Erklärungen:

```
#   Project:     Makefile
#
#   Date:        $Date: 2003/11/18 13:00:04 $
#   Author:      Wolfgang Haidinger
#

# Projectname
PRJNAME    = ExampleTimer

# select warnlevel & flags
WARNLEVEL  = -Wall -Wstrict-prototypes
CFLAGS     = $(WARNLEVEL) -Os

# Compiler name
CC = gcc

# listing the objects instead of the source files are preferred
# to highlight the following problem:
# when having a directory containing a file TuWas.c and Tuwas.S
# TuWas.c compiles to TuWas.o & TuWas.S overwrites TuWas.o
```

```
# OBJ should contain all your object files in the current directory
# while SUBOBJS should contain all the objects which are not in
# the current directory but required for linking
OBJS       = $(PRJNAME).o
SUBOBJS    =

#============================================================
# usually there is no need to change anything below this line
# but be carefull: the trojans placed the horse in the city
# because of their lack of knowledge ...
#============================================================

# since just the objects are provided, make has to generate the
# source file names from the objects
EXISTC     = $(wildcard *.c) # a list of all .c files of current dir
EXISTS     = $(wildcard *.S) # a list of all .S files of current dir
POTC       = $(OBJS:.o=.c) # all potentially .c source files
POTS       = $(OBJS:.o=.S) # all potentially .S source files
CFILES     = $(filter $(EXISTC), $(POTC)) # intersection (existing .c files)
SFILES     = $(filter $(EXISTS), $(POTS)) # intersection (existing .S files)
```

In den Variablen `EXISTC` und `EXISTS` werden zunächst alle „.c“ bzw. „.S“ Dateien des aktuellen Verzeichnisses gelistet. Danach werden den Variablen `POTC` und `POTS` potentielle, projektabhängige „.c“ bzw. „.S“ Dateien, durch Ersetzen der Dateierweiterung „.o“ von `OBJS`, zugewiesen. Durch die `filter` Funktion werden nun aus den potentiellen Dateien alle nicht existierenden Dateien entfernt und übrig bleiben jeweils „.c“ bzw. „.S“ Dateien, die für das Projekt kompiliert werden müssen.

```
# if there exist a .c and a .S file compiling to the same .o file stop!
BOTH = $(filter $(CFILES:.c=), $(SFILES:.S=)) # xxx.c & xxx.S?
ifneq "$(strip $(BOTH))" ""
  $(error Both $(addsuffix .c, $(BOTH)) and $(addsuffix .S,$(BOTH)) exist.)
endif
```

Jetzt werden in `BOTH` Dateinamen gespeichert, von denen es jeweils welche mit „.c“ und mit „.S“ Erweiterung gibt. Dazu wird wieder die `filter` Funktion verwendet. Parameter dafür sind die Variablen `CFILES` und `SFILES` bei denen jeweils die Erweiterung durch einen Leerstring ersetzt wurde. Gibt es nun gleiche Namen, so werden diese von `filter` nicht gelöscht und in `BOTH` gespeichert. Das folgende Konditional ist wahr, wenn die Variable `BOTH` nicht leer ist und es wird die `error` Funktion aufgerufen, welche die weitere Abarbeitung des Makefiles abbricht.

```
SUBDIRS    = $(dir $(SUBOBJS)) # just the dirs "Sub1/ Sub2/ ..."

# .d files are used to handle the
#include dependencies
# (so a altered header file will cause a recompilation)
# DFILES contains a list of all (existing & required) *.d files
DEXIST     = $(wildcard *.d) # list of all existing .d files
DREQUIRED  = $(CFILES:.c=.d) $(SFILES:.S=.d) # list of all required .d files
DFILES     = $(filter $(DEXIST),$(DREQUIRED)) # intersection
```

```
# list of all non-file targets
.PHONY: all makesub lc lmc clean mostlyclean distclean ainst finst einst

# 1st the main target
all: $(DREQUIRED) makesub $(PRJNAME).hex $(PRJNAME).eep

# include just existing D-files
ifneq "$(strip $(DFILES))" ""
  include $(DFILES)
endif
```

Mit dem `include` Kommando wird die Ausführung des aktuellen Makefiles unterbrochen und die als Argument angegebenen Makefiles abgearbeitet. Es können mehrere durch Leerzeichen getrennte Dateinamen angegeben werden. Nach deren Ausführung wird das aktuelle Makefile fortgesetzt. Mit einem '-' Zeichen vor dem `include` Kommando werden Fehlermeldungen bei nicht existierenden Dateien vermieden.

```
$(PRJNAME):  $(OBJS) $(SUBOBJS)
    $(CC) -Wl,-Map=$@.m $(LDFLAGS) -o $@ $(OBJS) $(SUBOBJS)

# do not use any built-in rules
.SUFFIXES:

# declare pattern rules
%.d:    %.c
# the compiler generates output of the following format:
# "Test.o: Test.c Header1.h Header2.h ..."
# sed is used to convert this into:
# "Test.o Test.d: Test.c Header1.h Header2.h ..."
# to declare the dependencies of Test.d too
    $(CC) $(DOPT) $(CFLAGS) $<|sed 's,\($*\).o[ :]*,\1.o $@: ,g' > $@

%.d:    %.S
# the compiler generates output of the following format:
# "Test.o: Test.S Header1.h Header2.h ..."
# sed is used to convert this into:
# "Test.o Test.d: Test.S Header1.h Header2.h ..."
# to declare the dependencies of Test.d too
    $(CC) $(DOPT) $(CFLAGS) $<|sed 's,\($*\).o[ :]*,\1.o $@: ,g' > $@

%.o:    %.c
    $(CC) $(CFLAGS) -Wa,-a=$*.l -c -o $@ $<

%.o:    %.S
    $(CC) $(CFLAGS) -Wa,-a=$*.l -c -o $@ $<

%.i:    %.c
    $(CC) $(CFLAGS) -E -o $@ $<
```

```
%.s:    %.c
    $(CC) $(CFLAGS) -S -o $@ $<
```

Spezielle *Pattern Rules* ermöglichen das Schreiben von Regeln für bestimmte Typen von Dateien, ohne dabei für jede Datei speziell eine Regel oder Abhängigkeit anzugeben. Make unterstützt dabei Regeln für einige Dateitypen (z.B. *.c, *.o, *.f,...). Wenn Dateien, auf denen eine dieser Regeln zutrifft, in einer `Dependency` oder einem `Target` vorkommen, überprüft make automatisch die implizite Regel für diese Dateien. Make verwendet spezielle Variablen zur Ausführung der `Kommandos` bei impliziten Regeln (z.B. $(CC) für das Kompilieren von *.c Dateien) die definiert werden können. Im obigen Codeblock werden eigene implizite Regeln, mit einer Variable für die Angabe des C-Compilers, bereitgestellt. Das Wildcardzeichen % Zeichen kann dabei jeder beliebigen Zeichenfolgen (auch Leerstrings) entsprechen. Es wird nur das erste Vorkommen von % als Wildcard gewertet.

```
# intended:
# TMPCLEAN = cd $(dir); make clean\n
# due to the trailing newline character a define is
# required instead of "="
define TMPMAKE
@cd $(dir); make

endef
define TMPCLEAN
@cd $(dir); make clean

endef
define TMPMOSTLY
@cd $(dir); make mostlyclean

endef
define TMPDIST
@cd $(dir); make distclean

endef

# declare the PHONYs
makesub: # calls make in subdirs
    $(foreach dir,$(dir $(SUBOBJS)),$(TMPMAKE))
```

Im obigen `foreach` wird das Kommando `dir` für alle Verzeichnisse in *$(SUBOBJS)* ausgeführt (das Verzeichnis wird extrahiert) und die Variable `$(dir)` mit dem jeweiligen Verzeichnis belegt. Dabei wird jedesmal die Variable *$(TMPMAKE)* (mit Inhalt „@cd $(dir); make “) aufgelöst und dadurch in das jeweilige Verzeichnis gewechselt und make ausgeführt.

```
lc: # local clean
    $(RM) $(CFILES:.c=.i) $(SFILES:.S=.i) # generated preprocs
    $(RM) $(CFILES:.c=.s) $(SFILES:.S=.s) # generated asms
    $(RM) $(CFILES:.c=.o) $(SFILES:.S=.o) # generated objects
```

```
    $(RM) $(CFILES:.c=.l) $(SFILES:.S=.l) # generated lists
    $(RM) $(PRJNAME).elf $(PRJNAME).m     # generated elf & map file

lmc: lc # local mostly clean
    $(RM) $(CFILES:.c=.d) $(SFILES:.S=.d) # generated dependecies

clean: lc
    $(foreach dir,$(dir $(SUBOBJS)),$(TMPCLEAN))

mostlyclean: lmc
    $(foreach dir,$(dir $(SUBOBJS)),$(TMPMOSTLY))

distclean: lmc
    $(RM) $(FLASH_FILE) $(EEP_FILE)       # generated hex & eep
    $(RM) *~ *.bak *.BAK                  # all (sic!) backups
    $(foreach dir,$(dir $(SUBOBJS)),$(TMPDIST))
```

Versionen von `make`

Die hier behandelte Version von make ist das sogenannte GNU make in der Version 3.80. Dieses Programm hat sich zur gängigsten Variante etabliert. Es gibt einen POSIX 2 Standard (IEEE Standard 1003.2-1992) in dem der Umfang von `make` spezifiziert ist. GNU make erfüllt diesen Standard und enthält zusätzlich noch einige Erweiterungen. Für viele Linux/Unix Derivate existieren eigene Versionen des `make` Kommandos, daher kann man nicht von einem allgemeinen `make` sprechen.

Eine detaillierte Beschreibung des GNU make Kommandos kann man im Web unter *http://www.gnu.org/software/make/* finden.

Das GNU Build-System

Bei der Verwendung von Softwarepacketen die mit dem GNU Build-System kompatibel sind, kann das entsprechende Projekt mit dem `make` Kommando kompiliert werden. Um dies zu ermöglichen wird mit dem Packet ein sogenanntes `configure` Shell Skript mitgeliefert. Bei der Ausführung erstellt dieses Skript die benötigten Makefile und passt diese an das jeweilige Betriebssystem an. Diese Makefile beinhalten, unter anderem, folgende Standard Targets (alle selbst geschriebenen Makefile sollte diese Targets ebenfalls enthalten):

install Kompiliert das Programm und kopiert alle ausführbaren Dateien, Bibliotheken usw. in die richtigen Programmverzeichnisse. Es sollten alle benötigten Verzeichnisse erstellt werden, wenn sie nicht bereits existieren. Kommandos innerhalb dieses Targets werden in drei Kategorien eingeteilt: normale, pre-Installation und post-Installation.

distclean Löscht alle Dateien die beim Build Prozess erzeugt wurden. Nach Ausführung des Targets sollten nur noch die original Packet-Dateien übrig bleiben.

uninstall Alle Dateien die von `install` erzeugt wurden werden gelöscht. Diese Regel sollte das Kompilierverzeichnis nicht verändern. Es gibt die gleichen Kategorien wie beim `install` Target.

Die übliche Reihenfolge beim Buildprozess sieht wie folgt aus:

1. Download des Packets.
2. Ausführen des `configure` Skripts mit „$./configure$“ (Makefile werden erstellt)
3. Ausführen von `make` (Target `all` wird implizit angenommen, Packet wird kompiliert)
4. Ausführen von `make install`

4.1.5 Das Debuggen von Programmen (`dbx`)

Zum Debuggen von Programmen steht in C der Source-Level Debugger `dbx` zur Verfügung. Dieser kann sowohl dazu benutzt werden, um Post-Mortem Debugging eines Programms durchzuführen, als auch zum interaktiven Debugging von Programmen. Der Aufruf des `dbx` als Post-Mortem Debugger sieht wie folgt aus:

```
dbx <Name des exekutierbaren Files> core
```

Dazu muss im momentanen Directory eine Datei namens `core` vorhanden sein, die nach dem Absturz eines Programms erzeugt wird. In dieser Datei ist der momentane Zustand des Programms zum Zeitpunkt des Absturzes gespeichert. Die wichtigsten Kommandos des Debuggers sind:

where Die Stelle des Absturzes und alle momentan aktiven Prozeduren werden angezeigt (‘*Stack-Backtrace*’).

w Einige Zeilen des Source-Codes, die der Absturzstelle vorangehen bzw. ihr folgen, werden angezeigt.

p <**expression**> Der Wert der Expression, der Variablen des abgestürzten Programms enthalten kann, wird angezeigt.

dump Information über die momentan aktive Prozedur (Aufrufparameter, lokale Variable) wird angezeigt.

quit oder

q Der Debugger wird verlassen.

Wird der Debugger zum interaktiven Debuggen eines Programms verwendet, so sieht die Aufrufsyntax folgendermaßen aus:

```
dbx <Name des exekutierbaren Files>
```

Einige wichtige Befehle zum interaktiven Debuggen sind (zusätzlich zu den oben erwähnten Befehlen):

run <**argument-list**> oder

r <**argument-list**> Das Programm wird mit den angegebenen Argumenten gestartet und läuft bis zum nächsten Breakpoint.

stop in <**procedure**> Am Anfang der Prozedur <*procedure*> wird ein Breakpoint gesetzt.

stop at <**line**> Ein Breakpoint wird in Zeile <*line*> im momentan aktiven Source-File gesetzt.

file <**file**> Das momentan aktive Source-File wird auf <*file*> gesetzt.

cont oder

c Die Programmausführung wird (nach einem Breakpoint) fortgesetzt.

step oder

s Die nächste Programmzeile wird ausgeführt. Bei Erreichen eines Prozeduraufrufs wird in diese Prozedur hinein verzweigt.

next oder

n Die nächste Programmzeile wird ausgeführt. Prozeduraufrufe werden als eine Programmzeile betrachtet (die ganze Prozedur wird mit einer `next` Anweisung exekutiert.

4.1.6 Der Aufruf von Programmen

Ein Programm muss immer mit Angabe des (absoluten oder relativen) Pfadnamens aufgerufen werden, sofern das Directory, in dem sich das Programm befindet, nicht in der Shell-Variable `PATH` enthalten ist. Ein Programm im momentanen Arbeitsdirectory kann mit ./<Programmname> aufgerufen werden.

4.2 Systemspezifische Funktionen in C-Programmen

Werden C-Programme unter UNIX oder Linux entwickelt, so steht dem Programmierer eine Funktionsbibliothek zur Verfügung, die weit über den Umfang der in ANSI-C definierten Funktionen hinausgeht. Bei der Programmierung sollte immer versucht werden, soweit als möglich ANSI-C konforme Funktionen zu verwenden. Es gibt jedoch einige Situationen, in denen davon abgegangen werden kann:

- Die Verwendung einer nicht ANSI-C konformen Funktion ist notwendig, um eine Aufgabe überhaupt lösen zu können.
- Bibliotheksfunktionen, die dazu dienen, das Verhalten eines Programmes an im verwendeten Betriebssystem übliche Konventionen anzupassen, können nicht nur, sondern sollen sogar verwendet werden. Das stellt nämlich sicher, dass bei einer eventuellen Änderung oder Erweiterung der Konvention das Programm durch erneutes Kompilieren automatisch an die neue Konvention angepasst wird. Ein Beispiel dafür ist die Bibliotheksfunktion `getopt(3)`.

- Es existiert eine Bibliotheksfunktion, die eine Funktionalität bereitstellt, die sonst händisch ausprogrammiert werden müsste. In diesem Fall kann die Bibliotheksfunktion verwendet werden, auch wenn sie nicht ANSI-C konform ist (unter der Voraussetzung, dass es keine ANSI-C Funktion vergleichbarer Funktionalität gibt).

Für Bibliotheksfunktionen, die nicht im ANSI-C Standard enthalten sind, ist nicht garantiert, dass Prototypen dafür deklariert sind. Für diese Funktionen muss der Prototyp selbst deklariert werden.

Die genaue Deklarationssyntax jeder einzelnen Bibliotheksfunktion kann der entsprechenden Manualseite (im On-Line Manual siehe 1.4.1) entnommen werden (Sie müssen die Deklarationen allerdings selbst in ANSI-C Prototypen umformen).

4.2.1 UNIX System-Calls

Da ein Hauptziel dieses Buches das Erlernen der Behandlung von Parallelität (Synchronisation, Interprozesskommunikation) ist, und es keine Standard-ANSI-C Funktionen dafür gibt, werden System-Calls zur Behandlung der Parallelität benötigt.

System-Calls in UNIX haben eine einheitliche Konvention zur Rückgabe von Werten: Der Rückgabewert jedes System-Calls ist vom Typ `int` (Ausnahme: `shmat(2)`[1]), und im Falle eines Fehlers wird der Wert `-1` geliefert und ein Fehlercode in die externe Variable `errno` geschrieben (die in der include-Datei `errno.h` deklariert ist). Die Funktion `strerror(3)` wandelt den Fehlercode in `errno` in einen String um, der die Fehlerursache beschreibt. Dieser String soll in der Fehlermeldung verwendet werden. *ACHTUNG!* Sie dürfen vor dem Aufruf von `strerror` keine Bibliotheksfunktion mehr aufrufen, da dadurch der Wert von `errno` geändert werden könnte. Falls es erforderlich ist `errno` Werte verschiedener Systemcalls *aufzuheben*, so müssen diese nach jedem in Frage kommenden Systemcall in einer entsprechenden Variable zwischengespeichert werden.

4.2.2 UNIX Bibliotheksfunktionen

Es gibt in UNIX sehr viele Bibliotheksfunktionen, die nicht in ANSI-C standardisiert sind. Diese Funktionen dienen einerseits dazu, den Entwickler bei der Erstellung von Programmen zu unterstützen, die betriebssystemspezifisches Verhalten zeigen (z.B. `crypt(3)` zum Verschlüsseln von Passwörtern gemäß der UNIX Konvention), andererseits dem Benutzer häufig gemachte Arbeit abzunehmen (z.B. `hsearch(3)` zur Verwaltung von Hash-Tabellen).

Allgemein sollten Sie versuchen, 'das Rad nicht zweimal zu erfinden', d.h., Sie sollten eine Funktion, die bereits als Bibliotheksfunktion vorhanden ist, nicht nochmals implementieren. Eine Ausnahme von dieser Regel ist dann gegeben, wenn Sie schon bei der Erstellung des Programms wissen, dass es auch auf einem anderen Betriebssystem installiert werden soll, auf dem diese Bibliotheksfunktion nicht vorhanden ist. Versuchen Sie allerdings auch in diesem Fall, sich bei der Implementierung der eigenen Funktion an die Syntax und Semantik der bereits vorhandenen Funktion zu halten (wer weiß, vielleicht wird diese Funktion irgendwann einmal standardisiert).

[1] Die Zahl in Klammern gibt bei Bibliotheksfunktionen – genauso wie bei UNIX-Kommandos – das Kapitel im Manual (siehe 1.4.1) an, in dem eine Beschreibung der Funktion gefunden werden kann

Argumentbehandlung (`getopt`)

Kommandos erlauben (verlangen) in der Regel Argumente und Optionen. Generell kann man sagen dass *Argumente* die Dinge sind auf die ein Kommando angewendet wird. Gemäß der Unix-Philosophie handelt es sich dabei meistens um Files. *Optionen* hingegen steuern normalerweise **wie** das Kommando angewendet wird.

Es gibt eine sehr genau definierte Konvention, wie Argumente und besonders Optionen von UNIX-Kommandos behandelt werden sollen. Da jedes unter UNIX erstellte Programm eigentlich ein UNIX-Kommando ist (auch in dem Fall, wenn man es nur selbst verwendet), hat sich *jedes* UNIX Programm an diese Konvention zu halten.

Diese Konvention sieht im Wesentlichen folgendermaßen aus:

i. Alle Optionen (und eventuelle Argumente zu diesen Optionen) müssen *vor* den restlichen Argumenten geschrieben werden.

ii. Jede Option beginnt mit einem waagrechten Strich (-) Ausnahme siehe v).

iii. Jede Option ist nur einen Buchstaben lang (diese Regel kann von einigen Programmen, die sehr viele Optionen kennen, nicht eingehalten werden).

iv. Eine Option kann ein Argument verlangen. Dieses ist dann allerdings obligat. Zwischen der Option und ihrem Argument können sich Leerzeichen befinden, müssen aber nicht.

v. Mehrere einbuchstabige Optionen (sofern sie kein Argument verlangen) können *einem* waagrechten Strich folgen (z.B. `ls -al`).

vi. Das Ende aller Optionen wird durch das erste Argument, das nicht mit einem waagrechten Strich beginnt, oder durch einen doppelten waagrechten Strich (--) gekennzeichnet.

Es gibt eine Bibliotheksfunktion, die die in UNIX übliche Art der Argumentbehandlung durchführt.

Die Funktion `getopt(3)` implementiert diese Konvention. Sie soll immer dann verwendet werden, wenn ein UNIX-Programm mindestens eine Option kennt. Sie wird folgendermaßen verwendet:

```
/* ********************************************************
includes **/

#include <stdio.h> #include <assert.h> #include <unistd.h>
#include <stdlib.h>

/* ********************************************************
globals **/

const char *szCommand = "<not yet set>";               /*
Programmname */

/* ********************************************************
functions **/
```

```
void Usage(void) {
 /*
  * Wenn die Optionen fehlerhaft sind, muss eine Usage-Meldung
  * ausgegeben werden, die den Namen, mit dem das Programm aufgerufen
  * wurde (in argv[0]) und die Aufrufsyntax beinhaltet.
  */
    (void) fprintf(
        stderr,
        "Usage: %s [-a] [-f filename] string1 [string2 [string3]]\n",
        szCommand
        );
    exit(EXIT_FAILURE);
}

int main(int argc, char **argv) {
    int c;
    int bError = 0;
    char * szInputFile = (char *) 0;
    int bOptionA = 0;

    szCommand = argv[0];              /* Kommandoname global speichern */

    /* Das dritte Argument zu getopt ist ein String, der die verwendeten
     * Optionen beschreibt. Ein Buchstabe ohne nachfolgenden Doppelpunkt
     * gibt an, dass die entsprechende Option kein Argument verlangt.
     * Ist ein Doppelpunkt vorhanden, verlangt die Option ein Argument.
     * Wenn alle Optionen behandelt sind, liefert getopt EOF zurueck.
     */
    while ((c = getopt(argc, argv, "af:")) != EOF)
    {
        switch (c)
        {
            case 'a':                     /* Behandlung der Option 'a' */
                if (bOptionA)                /* mehrmalige Verwendung? */
                {
                    bError = 1;
                    break;
                }
                bOptionA = 1;
            break;

            case 'f':                     /* Behandlung der Option 'f' */
                if (szInputFile != (char *) 0) /*mehrmalige Verwendung?*/
                {
                    bError = 1;
                    break;
```

```
                }
                szInputFile =optarg; /*optarg zeigt auf Optionsargument*/
            break;

            case '?':              /* falsches Argument wurde gefunden */
                bError = 1;
            break;

            default:               /* dieser Fall darf nicht auftreten */
                assert(0);
            break;
        }
    }

    if (bError)                                 /* Optionen fehlerhaft? */
    {
        Usage();
    }

    if (                              /* falsche Anzahl an Argumenten? */
        (optind + 1 > argc) ||
        (optind + 3 < argc)
        )
    {
        Usage();
    }

    /*
     * Die restlichen Argumente, die keine Optionen sind, sind in
     * argv[optind] bis argv[argc - 1] gespeichert.
     */
    while (optind < argc)
    {
        /*
         * Das momentan zu behandelnde Argument ist argv[optind]
         */
        ...
        optind ++;
    }
    ...
}
```

4.3 Fehlerbehandlung

Computerprogramme werden dazu erstellt, um bestimmte Aufgaben zu lösen (zum Beispiel Rechtschreibprüfung eines Textes). In der Spezifikation der Aufgabe ist jedoch leider meist nur implizit enthalten, unter welchen Rahmenbedingungen das Programm funktionieren soll. Was zum Beispiel soll passieren, wenn beim Schreiben des rechtschreibgeprüften Textes plötzlich die Festplatte voll ist?

In sicherheitskritischen Anwendungen (Nuklearreaktorsteuerung, "Fly-by-wire" Systemen etc.) ist ein genaue Spezifikation von allen möglichen Ausfallsarten und, wie sich das Computersystem in solchen Fällen zu verhalten hat, ein integraler Bestandteil des Designprozess[2]. Wie wenig Aufmerksamkeit darauf jedoch in normalen kommerziellen Anwendungen gelegt wird, offenbart sich ja tagtäglich dem Computerbenützer durch diverse Abstürze von Programmen und sogar Betriebssystemen.

Ein Programmierer sollte sich daher immer im Klaren sein über die

Annahmen über den Regelfall: Unter welchen Bedingungen soll das Programm richtig funktionieren? Wodurch können diese Bedingungen verletzt werden (= Fehlerfälle)?

Fehlerbehandlung: Welche Massnahmen sind zu setzen, wenn die getroffenen Annahmen verletzt werden? In welchem Zustand werden dabei Daten und System hinterlassen?

4.3.1 Ressourcenverwaltung

Ressourcen, die ein Programm verwenden kann, umfassen Dateien, Speicherbereiche oder die in den nächsten Kapiteln vorgestellten Interprozesskommunikationmechanismen (Message Queues, Named Pipes, Shared Memories etc). Diese Ressourcen können von mehreren Prozessen benützt werden (exklusiv oder auch gemeinsam). Im Allgemeinen sind Ressourcen nicht unbegrenzt verfügbar. Daher sollten Ressourcen, die nicht mehr benötigt werden, spätestens bei Programmende, aber unbedingt auch bei Programmabbruch wieder freigegeben werden!

Um sparsam mit Ressourcen umzugehen, werden sie nur für die Aktionen, für die sie wirklich gebraucht werden, belegt. Vor der Verwendung müssen Ressourcen angefordert, nach der Verwendung wieder freigegeben werden (Beispiel: Datei öffnen, Lesen oder Schreiben, Datei schließen).

Für kleinere Programme ist es technisch ratsam, das Anlegen aller Ressourcen in einer Funktion `void AllocateResources(void)` und das Löschen aller angelegten Ressourcen in einer Funktion `void FreeResources(void)` zusammenzufassen. Dies dient nicht nur der Übersichtlichkeit, sondern vereinfacht auch das Terminieren im Fehlerfall (siehe unten).

Sollen mehrere Prozesse miteinander kommunizieren, muss sowohl für den Regelfall als auch für den Fehlerfall festgelegt werden, welcher Prozess welche Ressourcen anlegt bzw. wieder löscht. Oft ist es sinnvoll, dass ein bestimmter Prozess für die gesamte Ressourcenverwaltung zuständig ist (in Client-Server Systemen zum Beispiel wird der Server die Ressourcenverwaltung durchführen und nicht die Clients).

[2] Interessierten sei dazu die VO "Fehlertolerante Systeme" empfohlen.

4.3.2 Schritte der Fehlerbehandlung

Im Folgenden werden die Schritte, die für eine korrekte Fehlerbehandlung nötig sind, erklärt:

Fehlerabfragen: Zu jedem Programmkonstrukt ist zu überlegen, welche Bedingungen erfüllt sein müssen, damit es richtig funktioniert bzw. wodurch diese Bedingungen verletzt sein können.

Mögliche Fehlerursachen zur Laufzeit des Programmes umfassen einerseits benutzerabhängige Eingaben (falsche Anzahl von Argumenten, falscher Eingabewert etc.) und andererseits systemabhängige Parameter (Festplatte voll etc.).

Ein Programm muss daher die Korrektheit der Benutzereingaben und die korrekte Ausführung von Systemcalls immer überprüfen (zB Returnvalue von `fopen()` abfragen!)

Tipp: Die "Returnvalue-" and "Error-" Kapitel in den Manualpages zu den System Calls geben sehr gute Anhaltungspunkte über mögliche Fehlerursachen!

Fehlerbekanntgabe: Im Falle eines Problems muss der Benutzer möglichst genau darüber informiert werden. Die Ausgabe hat auf `stderr` zu erfolgen. Relevante Informationen sind:

- Bei welchem Programm gibt es das Problem? (d.h. Programmname `argv[0]`)
- Was ist das Problem? (z.B.: "fopen failed")
- Was ist die Ursache? (Ausgabe der Ursache unter der Verwendung der Funktion `strerror(errno)`, siehe 4.2.1)

Eine den Richtlinien entsprechende Fehlerausgabe wird durch folgende Funktion erzielt:

```
void PrintError(const char *szMessage) {
    if (errno != 0)
    {
        (void) fprintf(
            stderr,
            "%s: %s - %s\n",
            szCommand,
            szMessage,
            strerror(errno)
            );
    }
    else
    {
        (void) fprintf(
            stderr,
            "%s: %s\n",
            szCommand,
            szMessage
            );
    }
}
```

In dieser Funktion ist `szCommand` eine globale Variable, die den Programmnamen (`argv[0]`) enthält. Diese muss im Hauptprogramm zu Beginn initialisiert werden.

"Recovery": Einfache Programme können im Fehlerfall immer durch eine Terminierung des Programmes erfolgen. Dabei ist auf ein "sauberes Terminieren" zu achten: Alle angelegten und nicht mehr benötigten Ressourcen sind zu löschen und die Daten sind in einem definierten Zustand zu hinterlassen.

Um im Fehlerfall schwerwiegende negative Auswirkungen (zum Beispiel den Verlust von Daten) zu vermeiden, ist aber meist eine Recoverystrategie festzulegen. Beim Versuch eine schreibgeschützte Datei zu beschreiben, könnte zum Beispiel ein Benutzerdialog gestartet werden und nach einem anderen Dateinamen fragen.

Das Löschen aller angelegten Ressourcen kann auch im Fehlerfall durch den Aufruf der vom Benutzer definierten Funktion `void FreeResources(void)` (siehe oben) erfolgen.

Beispiele für mögliche Implementierungen der beschriebenen Fehlerstrategien sind in den Programmbeispielen in den nachfolgenden Kapiteln zu finden. In diesen Beispielen wird davon ausgegangen, dass in einem Programmmodul `support.c` die Funktion `Print_Error()` implementiert ist und dass der Prototyp der Funktion `Print_Error()` im File `support.h` definiert ist.

4.4 Client-Server Prozesse, Implizite Synchronisation

Dieses Kapitel gibt eine kurze Einführung über das Prozesskonzept in UNIX. In der Folge werden parallele Prozesse der *Client-Server* Struktur betrachtet. Zur Koordination dieser parallelen Prozesse werden *implizite* Synchronisationsmechanismen ("Message Queue" und "Named Pipes") vorgestellt. Da Serverprozesse üblicherweise "endlos" laufen, erfolgt der Abbruch eines Servers meist von außen durch das Senden von "Signalen". Wie Prozesse auf Signale richtig reagieren, wird zum Abschluss dieses Kapitels erklärt.

4.4.1 Prozesse in UNIX

Wenn ein ausführbares Programm (Shell Kommando, Shell Skript, übersetztes C-Programm) aufgerufen (gestartet) wird, generiert und startet UNIX einen Prozess. Ein Prozess ist die Ausführung eines so genannten "Image". Ein "Image" umfasst Programm, Daten und Statusinformation. Der Adressbereich eines Prozesses gliedert sich in drei Segmente: Ein (schreibgeschütztes) Programmsegment, ein Datensegment und ein Stacksegment.

Die Statusinformationen (Attribute eines Prozesses) werden teilweise im Datensegment, teilweise in Datenstrukturen des Betriebssystemkerns gehalten. Zu ihnen gehören unter anderem:

- Prozessnummer (process identifier, PID)

  Jedem Prozess ist eine solche eindeutige Nummer zugeordnet. Sie wird dazu verwendet, um einen Prozess eindeutig identifizieren zu können.

- Prozessnummer des Elternprozesses (parent identifier, PPID)

  Wird ein Prozess durch einen anderen Prozess (Elternprozess) erzeugt (siehe `fork(2)`), dann bezeichnet PPID die Prozessnummer des Elternprozesses.

- Besitzerrechte: User ID, Group ID, ...

  Jedem Benutzer ist eine eindeutige Benutzernummer und eine eindeutige Gruppennummer zugeordnet. Da jeder Prozess einem Benutzer zugeordnet ist, wird nun jedem Prozess die entsprechende Benutzernummer als User ID und Gruppennummer als Group ID zugeordnet.

- Aufrufparameter (`argv`), Environment (z.B. Shell-Variablen)

- Kommunikationskanäle: Deskriptoren von Files, Message Queues, Shared Memorys, Semaphoren, und Signaldefinitionen

Während der Ausführung sind die Adressbereiche verschiedener Prozesse normalerweise völlig voneinander getrennt. Ein Austausch von Daten zwischen Prozessen kann daher in UNIX nur über zusätzliche Mechanismen realisiert werden. In diesem Kapitel werden *Message Queues* und *Named Pipes* betrachtet. Weitere Möglichkeiten zur Kommunikation werden später beschrieben.

Die einfachste Möglichkeit, parallele Prozesse zu erzeugen, ist die Ausführung von Programmen als Hintergrundprozesse. Es können natürlich auch mehrere parallele Prozesse dasselbe Programm ausführen.

4.4.2 Message Queues

Nachrichten (Messages) ermöglichen es einem Prozess, formatierte Daten an einen anderen Prozess zu übertragen. Dazu muss zuerst eine Message Queue (Warteschlange für Nachrichten) eingerichtet werden. Ein Prozess kann Nachrichten an eine bestimmte (oder mehrere) Message Queue(s) schicken, und andere Prozesse können diese Nachrichten von dort empfangen. Eine Message Queue ist u.a. charakterisiert durch

- einen vom Benutzer frei wählbaren nummerischen *Key*, der die Message Queue eindeutig identifiziert;
- User ID und Group ID des erzeugenden Prozesses;
- Read-Write Permissions für User, Group und alle anderen (genauso wie bei Files);
- Statusinformation (z.B. Zeitpunkt des letzten Zugriffs)

Folgende Funktionen stehen in der Library für die Manipulation von Message Queues zur Verfügung:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgget(key_t key, int msgflg);
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Das Anlegen einer Message Queue erfolgt durch den Systemaufruf `msgget()`. War der Aufruf erfolgreich, so liefert die Funktion einen Identifier zurück, der für alle weiteren Operationen mit dieser Message Queue (z.B. Senden einer Message) verwendet werden muss. Im Fehlerfall liefert die Funktion `-1`. Der Parameter `key` ist der oben beschriebene, eindeutige "Name" der Message Queue, ein nummerischer Schlüssel vom Typ `key_t`, ein ganzzahliger Typ. Der Parameter `msgflg` enthält eine Kombination aus den in Tabelle 4.2 beschriebenen Flags.

`PERMISSIONS`	Read-Write Permissions (z.B. 0660)
`IPC_CREAT`	Bewirkt, dass eine neue Message Queue mit dem vorgegebenen Key erzeugt wird. Existiert bereits eine Message Queue mit diesem Key, so wird der Identifier für diese Message Queue zurückgeliefert.
`IPC_EXCL`	Dieses Flag ist nur im Zusammenhang mit `IPC_CREAT` sinnvoll. Es bewirkt, dass im Falle der Existenz einer Message Queue mit dem spezifizierten Key nicht der Identifier sondern `-1` (Fehler) zurückgeliefert wird.

Tabelle 4.2: Flags für `msgget`

Mit Hilfe des System Call `msgsnd()` können nun Nachrichten gesendet (in die Queue gestellt) und mit Hilfe des System Call `msgrcv()` empfangen (aus der Queue gelesen) werden. Der erste Parameter bei beiden Funktionen identifiziert die Message Queue (Return-Wert von `msgget()`).

Mit `msgsnd()` wird eine zuvor im Programm vorbereitete Nachricht aus der Struktur `msgp` in die Queue mit Deskriptor `msqid` gestellt. Es ist zu beachten, dass beim Aufruf von `msgsnd()` die *Adresse* dieser Struktur zu übergeben ist. Die Struktur der Nachricht ist vom Benutzer zu definieren. Dabei gibt das erste Element den Nachrichtentyp an und muss vom Typ `long` sein. Abgesehen von dieser Einschränkung ist der Typ der Nachricht beliebig. Eine Nachrichtenstruktur könnte beispielsweise so aussehen:

```
struct msgbuf
{
    long mtype;          /* message type */
    char mtext[TXTLEN];  /* message text */
};
```

Die erforderliche Größe des Nachrichteninhalts wird in `msgsz` angegeben und kann mit `sizeof(msg) - sizeof(long)` berechnet werden. Der Parameter `msgflg` spezifiert die Aktionen, die auszuführen sind, wenn der interne Pufferbereich überläuft. Ist `IPC_NOWAIT` gesetzt, dann wird die Nachricht nicht gesendet, und `msgsnd()` terminiert sofort. Andernfalls wird der Prozess suspendiert, bis wieder genügend Platz in der Queue vorhanden ist, die Message Queue `msqid` aus dem System entfernt (gelöscht) wird oder `msgsnd()` durch ein Signal unterbrochen wird (Signalbehandlung siehe 4.4.5).

Der Systemaufruf `msgrcv()` liest aus der Message Queue mit Deskriptor `msqid` und schreibt das Ergebnis in die Struktur `msgp`, in der `msgsz` Bytes für den Nachrichteninhalt reserviert sind. Das Lesen aus einer Message Queue ist immer *konsumierend*, d.h., eine Nachricht wird durch das Lesen aus der Message Queue entfernt.

Durch Angabe des Message-Typs kann spezifiziert werden, welche Nachricht empfangen werden soll. Wenn `msgtyp 0` ist, dann wird die erste Nachricht der Queue empfangen; ist `msgtyp` größer als `0`, dann wird die erste Nachricht vom Typ `msgtyp` empfangen; ist `msgtyp` schließlich kleiner als `0`, dann wird die erste Nachricht mit dem niedrigsten Nachrichtentyp zwischen `1` und `-msgtyp` empfangen. Der Nachrichtentyp wird dabei durch den Sender im Feld `msgp.mtype` festgelegt. Aus dem Obigen ergibt sich auch, dass nur positive Zahlen als Nachrichtentypen verwendet werden dürfen.

Das Verhalten von `msgrcv()` für den Fall, dass keine Nachricht in der Queue vorhanden ist, die die geforderten Kriterien erfüllt, kann mit Hilfe von `msgflg` spezifiziert werden. Wird dabei das Flag `IPC_NOWAIT` gesetzt, so retourniert die Funktion `msgrcv()` sofort, anderfalls wird so lange gewartet, bis eine entsprechende Nachricht vom spezifizierten Typ vorhanden ist, bzw. eine der oben beschriebenen Ausnahmesituationen eintritt. Das Funktionsresultat von `msgrcv()` ist die Anzahl der Bytes, die tatsächlich gelesen wurden, bzw. `-1` im Fehlerfall.

Die beiden Arten des Lesens bzw. Schreibens werden auch als *blocking* bzw. *non-blocking read* bzw. *write* bezeichnet.

Für die implizite Synchronisation – wie wir sie in diesem Kapitel kennen lernen werden – ist das blockierende Lesen/Schreiben von Relevanz.

Mit Hilfe des System Calls `msgctl()` kann der Status der durch `msqid` identifizierten Message Queue gelesen oder gesetzt werden. Die Datenstruktur `buf` dient dabei zur Übernahme oder Übergabe von Statusinformation. Der Parameter `cmd` legt das auszuführende Kommando fest. Weiters kann mit Hilfe dieses System Calls eine Message Queue gelöscht (aus dem System entfernt) werden.

Dazu ist für `cmd` die Konstante `IPC_RMID` zu verwenden und `buf` kann auf `NULL` gesetzt werden.

Siehe auch: `msgget(2), msgop(2), msgctl(2), intro(2)`

Beispiel

Das folgende Beispiel zeigt die Implementierung eines vereinfachten Druckerspoolers als Client-Server System. Der Server (Druckerspooler) liest periodisch Filenamen aus einer Message Queue und druckt die entsprechenden Files auf dem Drucker aus. Die Clients werden vom Benutzer aufgerufen. Sie kopieren das auszudruckende File in ein spezieles Directory (das Spooldirectory) und teilen dem Server den Filenamen über die Message Queue mit.

Beachten Sie, dass in diesem Beispiel keine explizite Synchronisation notwendig ist, da sowohl der Client als auch der Server blockieren, wenn sie keine Nachricht senden bzw. empfangen können. Nicht ausprogrammiert sind die Funktionen zum Ausdrucken (PrintFile) bzw. Kopieren (CopyFile) der Dateien.

Der Server enthält auch eine Signalbehandlungsroutine (`handler()`) zum "sauberen Terminieren" des Programms. Eine detailierte Beschreibung von Signalbehandlungsroutinen finden Sie im Kapitel

4.4.5.

In beiden Programmen wird das gemeinsame Include File `msgtype.h` verwendet, in dem die Nachrichtenstruktur (`message_t`), der Schlüssel der Queue (`KEY`) und die Permissions (`PERM`) definiert sind.

Message-Typ Definition (Header File)

```
/*
 *  Module:        Message Queue Example
 *  File:          msgtype.h
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  8/26/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Message Type
 *
 *  FileID: msgtype.h 1.1
 */

/*
 * ********************************************************* includes ***
 */

#include <limits.h>           /* the constant PATH_MAX is defined here */

/*
 * ********************************************************** defines ***
 */

#define KEY 182251                            /* key for message queue */
#define PERM 0666                                  /* permission bits */
#define MESSAGE_TYPE 1                            /* type of messages */

/*
 * ********************************************************** typedefs ***
 */

typedef struct                                /* message buffer typedef */
{
    long nType;                                        /* message type */
    char szText[PATH_MAX];                                /* user data */
} message_t;

/*
 * ************************************************************* EOF ***
```

```
 */
```

Client Prozess

```
/*
 *  Module:        Message Queue Example
 *  File:          client.c
 *  Version:       1.2
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  9/16/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Client Process
 *
 *  FileID: client.c 1.2
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "support.h"
#include "msgtype.h"

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
static int nQueueID = -1;                            /* message queue ID */

/*
 * ******************************************************** functions ***
 */

void FreeResources(void)
{
                            /* nothing to do (queue removed by server ) */
}
```

```
void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((nQueueID = msgget(KEY, PERM)) == -1)
    {
        BailOut("Can't access message queue!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s <filename>\n", szCommand);

    BailOut((const char *) 0);
}


int main(int argc, char **argv)
{
    message_t sMessage;                              /* message buffer */

    szCommand = argv[0];                           /* store command name */

    if (argc != 2)                                    /* check arguments */
    {
        Usage();
    }

    if (strlen(argv[1]) >= PATH_MAX)       /* check length of argument */
    {
        BailOut("Filename too long!");
    }

    AllocateResources();
```

```
    CopyFile(argv[1]);     /* copy file to printer spooler's directory */

    (void) strcpy(sMessage.szText, argv[1]);         /* store filename */
    sMessage.nType = MESSAGE_TYPE;

    if (msgsnd(                                        /* send message */
        nQueueID,
        &sMessage,
        sizeof(sMessage) - sizeof(long),
        0
        ) == -1)
    {
        BailOut("Can't send message!");
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************ EOF ***
 */
```

Server Prozess

```
/*
 *  Module:        Message Queue Example
 *  File:          server.c
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  8/26/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Server Process
 *
 *  FileID: server.c 1.1
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

```
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "support.h"
#include "msgtype.h"

/*
 * ********************************************************* globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
volatile static int nQueueID = -1;                  /* message queue ID */

/*
 * ***************************************************** prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */

/*
 * ***************************************************** functions ***
 */

void FreeResources(void)
{
    if (nQueueID != -1)                      /* queue already created? */
    {
        if (msgctl(nQueueID, IPC_RMID, (struct msqid_ds *) 0) == -1)
        {
            nQueueID = -1;                  /* queue no longer present */

            BailOut("Can't remove message queue!");
        }

        nQueueID = -1;                      /* queue no longer present */
    }
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }
```

```
    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((nQueueID = msgget(KEY, PERM | IPC_CREAT | IPC_EXCL)) == -1)
    {
        BailOut("Can't create message queue!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s\n", szCommand);

    BailOut((const char *) 0);
}


void Handler(int nSignal)
{
    FreeResources();

    exit(EXIT_SUCCESS);
}


int main(int argc, char **argv)
{
    message_t sMessage;                              /* message buffer */

    szCommand = argv[0];                           /* store command name */

    if (argc != 1)                                    /* check arguments */
    {
        Usage();
    }

    (void) signal(SIGTERM, Handler);
    (void) signal(SIGINT, Handler);
    (void) signal(SIGQUIT, Handler);
```

```
    AllocateResources();              /* create all required resources */

    while (1)
    {
        if (msgrcv(                                 /* receive message */
            nQueueID,
            &sMessage,
            sizeof(sMessage) - sizeof(long),
            0,
            0
            ) == -1)
        {
            BailOut("Can't receive from message queue!");
        }

        PrintFile(sMessage.szText);
    }

    return 0;   /* not reached */
}

/*
 * ************************************************************ EOF ***
 */
```

Eine Erweiterung des oben angeführten Beispiels auf `N` Server, wobei jeder Server einen eigenen Drucker bedient, ist leicht möglich. Im Clientprozess wird der Nachrichtentyp zur Identifizierung des `i`-ten Servers verwendet, z.B.

```
   sMessage.nType = i;
```

im `i`-ten Serverprozess werden die Nachrichten mittels

```
   msgrcv(nQueueID, &sMessage, sizeof(sMessage) - sizeof(long), i, 0);
```

selektiv gelesen.

4.4.3 Named Pipes

Pipes ermöglichen die Datenübertragung zwischen Prozessen nach dem FIFO (First in First Out) Prinzip. In UNIX werden *Named Pipes* und *Unnamed Pipes* unterschieden. Während auf Named Pipes genauso wie auf gewöhnliche Files von beliebigen Prozessen zugegriffen werden kann, existieren Unnamed Pipes nur zwischen verwandten Prozessen. Dieses Kapitel befasst sich in der Folge mit Named Pipes, Unnamed Pipes werden in Kapitel 4.8 behandelt.

Folgende Funktionen stehen in der Library für die Manipulation von Pipes zur Verfügung:

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

```
int mkfifo(const char *path, mode_t mode);
int remove(const char *path);
```

Das Anlegen einer Named Pipe erfolgt mittels der Funktion `mkfifo()`. Der Parameter `path` ist der Pfadname der Named Pipe. War der Aufruf von `mkfifo()` erfolgreich, so gibt es danach im entsprechenden Directory einen Eintrag für die Named Pipe (vgl. Erzeugen eines Files). Der Parameter `mode` legt die Zugriffsberechtigung auf die Named Pipe fest.

Nach dem erfolgreichen Anlegen einer Named Pipe kann damit wie mit einem gewöhnlichen File operiert werden (`fopen(3s), fputs(3s), fgets(3s), fclose(3s)`).

Das Löschen einer Named Pipe erfolgt wie bei einem gewöhnlichen File mit Hilfe von dem System Call `remove()`.

Im Zusammenhang mit Named Pipes gilt:

- Ein Prozess, der eine Named Pipe zum Lesen öffnet, wird solange verzögert, bis ein anderer Prozess die Named Pipe zum Schreiben geöffnet hat (d.h. es existiert ein Schreiber) und umgekehrt.

- Ein Prozess, der von einer Named Pipe liest, die zumindest von einem anderen Prozess zum Schreiben geöffnet wurde, wird solange verzögert, bis der schreibende Prozess Daten geschrieben hat (blocking I/O). Terminiert der letzte schreibende Prozess, bzw. versucht ein Prozess von einer Named Pipe zu lesen, die von keinem anderen Prozess zum Schreiben geöffnet ist, so erhält der Leseprozess `EOF` sofern keine Daten zum Lesen mehr bereitstehen. In diesem Zusammenhang kann es auch leicht zu "Busy Waiting" kommen, wenn der Leseprozess die Named Pipe, nachdem alle schreibenden Prozesse die Named Pipe geschlossen haben, nicht schließt und wieder neu öffnet.

- Das Lesen aus einer Pipe ist konsumierend.

- Ein Prozess, der auf eine Named Pipe schreibt, die "voll ist", wird solange verzögert, bis durch das Auslesen der Pipe durch einen anderen Prozess wieder Platz vorhanden ist. Ebenso wird ein Prozess verzögert, der von einer Named Pipe liest, in die nichts geschrieben wurde (*implizite Synchronisation*).

- Ein Prozess, der auf eine Named Pipe schreibt, die von keinem anderen Prozess zum Lesen geöffnet ist, erhält das Signal `SIGPIPE` (siehe Kapitel 4.4.5).

- Die Anzahl der Lese- und Schreiberprozesse muss nicht notwendigerweise gleich sein. Gibt es mehr als einen Schreiber bzw. Leser, so müssen zusätzliche Mechanismen zur Koordination (z.B. Semaphore, siehe Kapitel 4.5.1) verwendet werden. Kritisch ist sowohl die Situation mit mehr als einem Schreiber, da bei unkoordiniertem Vorgehen ein (nicht vorhersehbarer) "Datensalat" entstehen kann, als auch die Situation mit mehreren Lesern, da im Vorhinein nicht bestimmt werden kann, welcher Leser welche Daten lesen wird (einmal gelesene Daten werden aus der Pipe entfernt).

Zur Veranschaulichung zeigen wir das Druckerspooler-Beispiel von vorhin, jetzt allerdings unter Verwendung einer Named Pipe. Es sei darauf hingewiesen, dass für diese Implementierung gewisse Einschränkungen gelten: Es wird vorausgesetzt, dass die Named Pipe beim Aufruf des Client-Prozess bereits existiert und dass immer nur ein Client "gleichzeitig" gestartet wird. Andernfalls könnte es sein, dass mehrere Clients "gleichzeitig" auf die Pipe schreiben und dadurch die Filenamen verstümmeln, da Schreiboperationen nur bis zu einer gewissen Maximalgröße (PIPE_BUF Bytes) garantiert *atomic* (nicht unterbrechbar) sind.

Der Serverprozess enthält eine Signalbehandlungsroutine; genauere Informationen dazu finden Sie im Kapitel 4.4.5.

Das gemeinsame Include File `namedpipe.h` definiert den Namen der Named Pipe (`PIPE_NAME`)und einen Datentyp (`pipedata_t`), für die Daten, die über die Pipe ausgetauscht werden.

Named Pipe Definition (Header File)

```
/*
 *  Module:        Named Pipe Example
 *  File:          namedpipe.h
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  8/26/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Named Pipe Definitions
 *
 *  FileID: namedpipe.h 1.1
 */

/*
 * ************************************************** includes ***
 */

#include <limits.h>           /* the constant PATH_MAX is defined here */

/*
 * ************************************************** defines ***
 */

#define PERM 0666                                  /* permission bits */
#define PIPE_NAME "spooler-pipe"                      /* name of pipe */

/*
 * ************************************************** typedefs ***
 */
```

```
typedef char pipedata_t[PATH_MAX];                /* pipe data typedef */

/*
 * ************************************************************ EOF ***
 */
```

Client Prozess

```
/*
 *  Module:        Named Pipe Example
 *  File:          client.c
 *  Version:       1.2
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  9/16/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Client Process
 *
 *  FileID: client.c 1.2
 */

/*
 * ********************************************************** includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include "support.h"
#include "namedpipe.h"

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
static FILE *pNamedPipe = (FILE *) 0;         /* pointer for named pipe */

/*
 * ******************************************************* prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */
```

```
/*
 * ********************************************************* functions ***
 */

void FreeResources(void)
{
    if (pNamedPipe != (FILE *) 0)        /* named pipe already opened? */
    {
        if (fclose((FILE *) pNamedPipe) == EOF)
        {
            pNamedPipe = (FILE *) 0;            /* pipe no longer open */

            BailOut("Can't close named pipe!");
        }

        pNamedPipe = (FILE *) 0;                /* pipe no longer open */
    }
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{                                                          /* open pipe */
    if ((pNamedPipe = fopen(PIPE_NAME, "w")) == (FILE *) 0)
    {
        BailOut("Can't open named pipe!");
    }

}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s <filename>\n", szCommand);
```

```
    BailOut((const char *) 0);
}


int main(int argc, char **argv)
{
    pipedata_t szFileName;

    szCommand = argv[0];                          /* store command name */

    if (argc != 2)                                   /* check arguments */
    {
        Usage();
    }

    if (strlen(argv[1]) >= PATH_MAX)       /* check length of argument */
    {
        BailOut("Filename too long!");
    }

    AllocateResources();

    CopyFile(argv[1]);     /* copy file to printer spooler's directory */

    (void) strcpy(szFileName, argv[1]);              /* store filename */

    if (fprintf(pNamedPipe, "%s\n", szFileName) < 0)
    {
        BailOut("Can't write to pipe!");
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************* EOF ***
 */
```

Server Prozess

```
/*
 *  Module:        Named Pipe Example
 *  File:          server.c
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  8/26/99
```

```
 *  Author:         Thomas M. Galla
 *                  tom@vmars.tuwien.ac.at
 *  Contents:       Server Process
 *
 *  FileID: server.c 1.1
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "support.h"
#include "namedpipe.h"

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
volatile static FILE *pNamedPipe = (FILE *) 0;   /* ptr for named pipe */
volatile static int bPipeCreated = 0; /*flag indicates creation status */

/*
 * ******************************************************* prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */

/*
 * ******************************************************** functions ***
 */

void FreeResources(void)
{
    if (pNamedPipe != (FILE *) 0)        /* named pipe already opened? */
    {
        if (fclose((FILE *) pNamedPipe) == EOF)
        {
            pNamedPipe = (FILE *) 0;            /* pipe no longer open */
```

```
            BailOut("Can't close named pipe!");
        }

        pNamedPipe = (FILE *) 0; /* pipe no longer open */
    }

    if (bPipeCreated)                   /* named pipe already created? */
    {
        if (remove(PIPE_NAME) != 0)
        {
            bPipeCreated = 0;               /* pipe no longer present */

            BailOut("Can't remove named pipe!");
        }

        bPipeCreated = 0;
    }
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if (mkfifo(PIPE_NAME, PERM) == -1)           /* created named pipe */
    {
        BailOut("Can't create name pipe!");
    }

    bPipeCreated = 1;
}


void Usage(void)
{
```

```
    (void) fprintf(stderr, "USAGE: %s\n", szCommand);

    BailOut((const char *) 0);
}


void Handler(int nSignal)
{
    FreeResources();

    exit(EXIT_SUCCESS);
}


int main(int argc, char **argv)
{
    pipedata_t szFileName;

    szCommand = argv[0];                          /* store command name */

    if (argc != 1) /* check arguments */
    {
        Usage();
    }

    (void) signal(SIGTERM, Handler);
    (void) signal(SIGINT, Handler);
    (void) signal(SIGQUIT, Handler);

    AllocateResources();              /* create all required resources */

    while (1)
    {
        if ((pNamedPipe = fopen(                          /* open pipe */
            PIPE_NAME,
            "r"
            )) == (FILE *) 0)
        {
            BailOut("Can't open named pipe!");
        }

        while (fgets(             /* read from pipe until EOF or error */
            szFileName,
            sizeof(szFileName),
            (FILE *) pNamedPipe
            ) != NULL)
        {
```

```
            szFileName[strlen(szFileName) - 1] = '\0';/* strip newline */

            PrintFile(szFileName);
        }

        if (ferror(pNamedPipe))                    /* really an error? */
        {
            BailOut("Can't read from pipe!");
        }

        if (fclose((FILE *) pNamedPipe) == EOF)
        {
            pNamedPipe = (FILE *) 0;            /* pipe no longer open */

            BailOut("Can't close named pipe!");
        }

        pNamedPipe = (FILE *) 0;                /* pipe no longer open */
    }

    return 0;                                          /* not reached */
}

/*
 * ************************************************************* EOF ***
 */
```

4.4.4 Unterschiede zwischen Message Queues und Named Pipes

Die wesentlichen Unterschiede zwischen Message Queues und Named Pipes sind hier noch einmal zusammengefasst:

- Message Queues eignen sich für die paketweise Übertragung von strukturierten Daten Named Pipes für unstrukturierte Datensequenzen.
- Message Queues werden durch einen nummerischen Key identifiziert, Named Pipes durch einen Pfadnamen.
- Daten aus einer Named Pipe können nur sequenziell (FIFO) gelesen werden, Nachrichten aus einer Message Queue können über den Nachrichtentyp selektiv gelesen werden.
- Für die Lese- und Schreiboperationen gibt es bei Message Queues spezielle Systemaufrufe, Named Pipes können hingegen wie normale Files behandelt werden.
- Für Named Pipes gibt es einen eindeutigen Eintrag innerhalb des UNIX Filessystems (d.h. der Status kann über das `ls(1)` Kommando abgefragt werden, Message Queues werden vom Betriebssystem besonders verwaltet, der Status kann mittels `ipcs(1)` festgestellt werden.

- Wenn mehrere Prozesse gleichzeitig Nachrichten in eine Message Queue schreiben, sorgt das Betriebssystem für den ordnungsgemäßen Ablauf, bei Named Pipes hat der Programmierer für die Koordination der parallelen Schreiboperationen zu sorgen (siehe auch Seite 240) .

- Bei Pipes muss es zum Zeitpunkt des Schreibens zumindest einen Leser geben. Ist dies nicht der Fall, so erhält der schreibende Prozess das Signal `SIGPIPE` (siehe Kapitel 4.4.5). In eine Message Queue kann auch geschrieben werden, wenn vorläufig kein Leseprozess aktiv ist.

- Bei `open` auf eine Named pipe findet eine zusätzliche Synchronisation statt. Das `open` blockiert, bis das jeweils andere Ende der Named Pipe geöffnet wurde. Diese Synchronisation ist nicht mit der Synchronisation beim Lesen/Schreiben der Named Pipe zu verwechseln.

4.4.5 Ausnahmebehandlung in UNIX, Signale

Wie wir in den vorigen Kapiteln gesehen habe, laufen Server praktisch "endlos". Will man einen Server aus irgend einem Grund doch stoppen, so gibt es prinzipiell zwei Möglichkeiten:

- Ein spezieller Client Prozess generiert eine "ENDE" Nachricht

- man sendet dem Server ein *Signal.*

Ein Signal ist ein Ereignis, das einem Prozess mitgeteilt wird. Es entspricht einer "leeren Nachricht", von der zu einem bestimmten Zeitpunkt nur festgestellt werden kann, ob sie vorhanden ist oder nicht. Tabelle 4.3 gibt einen Überblick über die wichtigsten Signale.

Folgende Funktionen stehen in der Library in Bezug auf Signale zur Verfügung:

```
#include <signal.h>

int kill (pid_t pid, int sig);
void (*signal (int sig, void (* func)(int)))(int);
```

Erzeugen von Signalen

Die meisten in der Tabelle 4.3 angeführten Signale werden durch die Hardware oder die Systemsoftware beim Eintreffen der jeweiligen Bedingung (z.B. Illegal Instruction, Segmentation Violation etc.) erzeugt und sollten nicht für eigene (Anwender-) Zwecke gebraucht werden.

Einige Signale können vom Terminal aus erzeugt werden, so z.B. SIGINT in der Regel durch Eingabe von `<Ctrl>-C` und SIGQUIT durch Eingabe von `<Ctrl>-\`.

C-Programme können Signale mit Hilfe des Systemaufrufs `kill()` erzeugen. Die Funktion `kill()` sendet das Signal `sig`, das eines der oben angeführten Signale sein muss (symbolische Namen sind im Include-File `signal.h` definiert), an den Prozess mit der Identifikationsnummer `pid`.

Symbol	Wert	Beschreibung
SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGFPE	8*	Floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGSTOP	17+	Stop (cannot be caught or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19•	Continue process after stop
SIGTTIN	21+	Background read from terminal
SIGTTOU	22+	Background write to terminal
SIGIO	23•	I/O is possible
SIGXCPU	24	CPU time limit exceeded
SIGXFSZ	25	File size limit exceeded
SIGUSR1	30	User defined signal 1
SIGUSR2	31	User defined signal 2

Tabelle 4.3: Signale in UNIX

Behandlung von Signalen

Werden keine speziellen Maßnahmen getroffen, dann werden nach dem Empfang eines Signals folgende Default-Aktionen ausgeführt:

Nach dem Empfang der meisten Signale terminiert ein Prozess. Beim Empfang eines der Signale, die in Tabelle 4.3 mit * markiert sind, wird zusätzlich ein "Core Dump" angelegt. Das kann z.B. dazu verwendet werden, um einen Prozess, der in einer Endlosschleife läuft, mit dem Signal SIGQUIT (`<Ctrl>-\`) abzubrechen und danach mit Hilfe eines Debuggers die Fehlersuche durchzuführen.

Eine weitere Ausnahme bilden die in Tabelle 4.3 mit • und + gekennzeichneten Signale: Alle Signale, die mit • markiert sind, werden ignoriert, d.h. ihr Empfang bewirkt also nichts. Der Empfang eines mit + markierten Signals hat zur Folge, dass der betroffene Prozess gestoppt wird.

Die Default-Behandlungen können mittels der Funktion `signal()` geändert werden. Diese Funktion dient lediglich zur *Vorbereitung* der zukünftigen Signalbehandlung; durch ihren Aufruf stellt sie eine Beziehung zwischen einem Signal `sig` und einer Funktion `func` her, sodass sobald das Signal `sig` eintrifft, die Funktion `func` aufgerufen wird. Das Funktionsergebnis von `signal()` ist die Adresse derjenigen Funktion, die zuvor mit dem Signal `sig` verbunden war, bzw. `SIG_ERR` im Fehlerfall. Jede einmal getroffene Zuordnung ist solange gültig, bis sie explizit geändert wird.

Für `func` sind folgende Werte möglich:

`SIG_DFL`: Die Default-Aktion (siehe oben) wird durchgeführt.

`SIG_IGN`: Das Signal wird ignoriert.

`func` (eigene Funktion): Das Signal wird durch Aufruf dieser Funktion behandelt.

Die meisten *System Calls* können durch Signale nicht unterbrochen werden; es gibt allerdings einige (wenige) Ausnahmen, z.B. Lesen und Schreiben von/auf ein langsames Peripheriegerät (Terminal), Operationen (Öffnen) auf Named Pipes und Message Queues. In Zweifelsfällen ist das Verhalten eines Systemaufrufs der jeweiligen Manualpage zu entnehmen.

Die oben erwähnten System Calls können bis zu einem gewissen Stadium abgebrochen werden und müssen nach der Signalbehandlung eventuell wiederholt werden. Dieser Fall wird durch den Return-Wert `-1` und durch den Fehlercode `EINTR` in der globalen Variablen `errno` angezeigt (siehe dazu `intro(2)` bzw. Kapitel 4.8).

Bei der vorausgegangenen Diskussion gilt es noch zu beachten, dass die Signale SIGKILL und SIGSTOP weder ignoriert noch durch eine eigene Funktion abgefangen werden können. Jeder solche Versuch produziert eine Fehlermeldung der Funktion `signal()` und bleibt sonst wirkungslos.

Eigenschaften der Signalbehandlungsroutine

Bei der Programmierung einer Signalbehandlungsroutine sind im Vergleich zu normalen Funktionen einige Einschränkungen zu beachten:

Erstens hat eine Signalbehandlungsroutine einen Parameter vom Typ `int`, in dem das aktuelle Signal übergeben wird. Dadurch ist es möglich, in einer Signalbehandlungsroutine, die für mehrere Signale zuständig ist, festzustellen, welches Signal tatsächlich eingetroffen ist.

Zweitens liefern Signalbehandlungsroutinen keinen Funktionswert; alle Datenübergaben an andere Funktionen müssen daher über globale Variable implementiert werden.

Diese Einschränkungen sind durch die Art und Weise der Verwendung von Signalbehandlungsroutinen begründet: Sie werden asynchron aufgerufen (d.h. bei Empfang des betreffenden Signals), unterbrechen dabei den normalen Programmablauf und kehren nach Beendigung an die Stelle im Programm zurück, an der unterbrochen wurde. Der Programmierer hat daher keine Kontrolle über den Zeitpunkt des Aufrufes und kann daher keine aktuellen Parameter und keine Variable zur Speicherung des Funktionsergebnisses bereitstellen.

Während der Ausführung der Signalbehandlungsroutine ist das Auftreten eines weiteren Signals derselben Art blockiert; andere Signale können dagegen empfangen werden.

Variablen, die in der Signalbehandlungsroutine gelesen oder modifiziert werden, sollten als `volatile` deklariert werden. Damit ist sichergestellt, dass der Compiler nicht eine dieser Variablen in ein Register legt, sodass die Änderung der Variablen durch die Signalbehandlungsroutine nicht wirksam wird.

Zusätzlich definiert der C-Standard, dass Variablen die in einer Interruptroutine verwendet werden, als atomic action (nicht unterbrechbar) bearbeitet werden können müssen. Die Variable muss dafür eine bestimmte Größe haben, um in einer atomaren Aktion gelesen oder geschrieben werden zu können. Ansi-C definiert daher einen neuen Typ, der diese Eigenschaft hat: `sig_atomic_t` (Siehe

auch `signal.h`).

Verwendung von Signalen

Beispiel 1

Im Folgenden zeigen wir, wie Signale dazu verwendet werden können, in unseren zuvor beschriebenen Serverprozessen ein "korrektes Terminieren" zu implementieren.

Zuerst definieren wir eine Funktion `clean_up()`, die die "Aufräumarbeiten" in unserem ersten Server erledigen soll. Da dieser zukünftigen Signalbehandlungsroutine mit Ausnahme der Signalnummer keine weiteren Parameter übergeben werden können, müssen sämtliche benötigten Variablen (z.B. Identifier der Message Queue) global vereinbart werden. Außerdem ist das File `signal.h` zu inkludieren.

```
void clean_up(int sig)
{
    if (msgctl((int) msgid, IPC_RMID, (struct msqid_ds *) 0) == -1)
    {
        BailOut("cannot remove message queue");
    }
    exit(EXIT_SUCCESS);
}
```

Im Server müssen an geeigneter Stelle die gewünschten Signale mit der Signalbehandlungsroutine verbunden werden. Bei uns ist dies unmittelbar nach dem Erzeugen der Message Queues der Fall. Zu beachten ist, dass es in diesem Beispiel nicht notwendig ist, den Return-Wert von `signal()` abzufragen (siehe dazu `signal(3)`, Section *Errors*), da nur bei nicht existierenden Signalnummern oder beim Versuch für `SIGKILL` oder `SIGSTOP` eine Signalbehandlungsroutine zu installieren, ein Fehler auftreten kann. Die erste Fehlermöglichkeit wird durch die Verwendung vordefinierter Konstanten ausgeschlossen, die Zweite, indem nicht versucht wird, die nicht verwendbaren Signale zu behandeln.

```
(void) signal(SIGHUP, clean_up);
(void) signal(SIGINT, clean_up);
(void) signal(SIGQUIT, clean_up);
...
```

Beispiel 2

Wenden wir uns nun dem zweiten Beispiel zu (Printer Spooler mit *Named Pipes*). Nehmen wir an, der Serverprozess wird aus irgendeinem Grund mittels SIGKILL abgebrochen. Da dieses Signal nicht "abgefangen werden kann", wird die `clean_up()` Routine nicht ausgeführt, d.h. die Named Pipe existiert weiter. Nehmen wir weiters an, dass das Abbrechen des Server-Prozesses genau in dem Augenblick erfolgt ist, in dem ein Client Prozess die Named Pipe bereits zum Schreiben geöffnet hat.

Was geschieht mit einem Prozess, der versucht, auf eine Named Pipe zu schreiben, für die es keinen lesenden Prozess gibt? Das UNIX Betriebssystem generiert in diesem Fall das Signal SIGPIPE.

Im Client Prozess möchten wir in diesem Fall nicht "wortlos" terminieren, sondern dem Benutzer eine entsprechende Fehlermeldung zukommen lassen. Die benötigte Routine sieht folgendermaßen aus:

```
   void no_pipe (int a)
   {
       BailOut("Cannot communicate with Server");
   }
```

Im Clientprozess erfolgt das Setzen der Routine wieder einfach mittels

```
   (void) signal(SIGPIPE, no_pipe);
```

Beispiel 3

Es folgt ein weiteres Beispiel, in dem eine Signalbehandlungsroutine zur Behandlung mehrerer Signale verwendet wird.

Ein Programm soll das Auftreten der Signale `SIGINT` und `SIGQUIT` innerhalb eines bestimmten Zeitintervalls zählen. Nach Ablauf des Zeitintervalls soll die Anzahl der Vorkommnisse der beiden Signale auf *stdout* ausgegeben werden. Das Zeitintervall wird als Argument übergeben, die Überwachung des Timeouts erfolgt mittels `alarm(3)`.

```
/*
 *  Module:        Signal Example
 *  File:          sigcount.c
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  10/8/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Signal Counter
 *
 *  FileID: sigcount.c 1.1
 */

/*
 * ******************************************************** includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
```

```
#include "support.h"

/*
 * ********************************************************* globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
static volatile int nSigInt = 0;                      /* SIGINT counter */
static volatile int nSigQuit = 0;                    /* SIGQUIT counter */

/*
 * ***************************************************** prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */

/*
 * ****************************************************** functions ***
 */

void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    exit(EXIT_FAILURE);
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s <timeout>\n", szCommand);

    BailOut((const char *) 0);
}


void Handler(int nSignal)
{
    switch (nSignal)
    {
        case SIGINT:
            nSigInt++;
        break;
```

```
        case SIGQUIT:
            nSigQuit++;
        break;

        case SIGALRM:
            if (printf(
                "Anzahl <ctrl>-c: %d\n"
                "Anzahl <ctrl>-\\: %d\n",
                (int) nSigInt,
                (int) nSigQuit
                ) < 0)
            {
                BailOut("Can't print to stdout!");
            }
            if (fflush(stdout) == EOF)
            {
                BailOut("Can't flush stdout!");
            }

            exit (EXIT_SUCCESS);
        break;

        default:
            BailOut("Unexpected signal caught!");
        break;
    }
}


int main(int argc, char **argv)
{
    long nTimeout = 0;
    char *pError = (char *) 0;

    szCommand = argv[0];                          /* store command name */

    if (argc != 2)                                   /* check arguments */
    {
        Usage();
    }

    errno = 0;                                            /* reset errno */
    nTimeout = strtol(argv[1], &pError, 0); /*convert argv[1] to number*/

    if (                      /* timeout negative or conversion error? */
        (nTimeout <= 0) ||
```

```
        (*pError != '\0') ||
        ((errno != 0) && (nTimeout == LONG_MAX))
        )
    {
        BailOut("Can't convert timeout value!");
    }

    (void) signal(SIGINT, Handler);         /* install signal handlers */
    (void) signal(SIGQUIT, Handler);
    (void) signal(SIGALRM, Handler);

    (void) alarm(nTimeout);                          /* install timer */

    while (1)
    {
        (void) pause();                     /* wait for timer to elaps */
    }

    exit(EXIT_SUCCESS);                              /* never reached */
}

/*
 * ************************************************************* EOF ***
 */
```

4.5 Explizite Synchronisation

4.5.1 Semaphore und Semaphorfelder

Das am allgemeinsten anwendbare Hilfsmittel zur expliziten Lösung von Synchronisationsproblemen sind Semaphore.

Unter UNIX sind Semaphore globale Objekte (vergleichbar mit Files, Message Queues und Shared Memorys), deren eigene Lebensdauer von der Lebensdauer der sie verwendenden Prozesse unabhängig ist. Das bedeutet z.B., dass eine Semaphorvariable auch nach Terminierung aller Prozesse, die sie verwenden, noch existiert, soferne sie nicht explizit gelöscht worden ist. Die unterstützten Operationen sind teilweise sehr mächtig und gehen weit über die Funktionalität der ursprünglichen `P(S)` und `V(S)` Operationen hinaus. So können z.B. Felder von Semaphoren definiert werden, von denen dann ein oder mehrere Elemente um verschiedene, beliebig große Werte inkrementiert oder dekrementiert werden können. Diese gesamte Operation wird als *Atomic Action* durchgeführt.

Semaphor Basis Operationen

Die Semaphor Basis Operationen umfassen die folgenden Funktionen:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);   /* key_t ganzzahlig */
int semop(int semid, struct sembuf *sops, int nsops);
int semctl(int semid, int semnum, int cmd, union semun arg);
```

Mit dem System-Call `semget()` wird ein Feld von `nsems` Semaphoren vom System angefordert; dies ist Voraussetzung für die Benutzung dieses Feldes durch den aufrufenden Prozess. Damit kann entweder ein neues Feld angelegt werden oder eine Zugriffsmöglichkeit auf ein bereits bestehendes (das z.B. von einem anderen Prozess angelegt wurde) geschaffen werden. Der Parameter `key` ist dabei der Name des Semaphor-Feldes, ein numerischer Schlüssel vom Typ `key_t`. Wird der Schlüssel `IPC_PRIVATE` verwendet, dann wird ein Semaphor-Feld neu angelegt, auf das andere, zum erzeugenden Prozess nicht verwandte Prozesse **nicht** zugreifen können.

Der Parameter `semflg` enthält die Zugriffsberechtigungen für das durch `key` angesprochene Semaphor-Feld. Ähnlich den Filezugriffsberechtigungen bestehen die Zugriffsberechtigungen aus jeweils 3 Bit für den Eigentümer des Semaphor-Feldes, die Benutzergruppe des Eigentümers und alle anderen Benutzer. Das erste Bit jeder Dreiergruppe gibt die Lese-, das Zweite die Schreib- oder Änderungsberechtigung an; das dritte Bit hat keine Bedeutung und wird immer ignoriert.

Ist zusätzlich das Flag `IPC_CREAT` gesetzt, dann wird ein neues Semaphor-Feld mit dem angegebenen `key` angelegt, wenn ein solches noch nicht existiert. Ist `IPC_CREAT` nicht gesetzt, dann ist es ein Fehler, wenn ein Semaphor-Feld mit dem Namen `key` nicht bereits existiert. Ist das Flag `IPC_EXCL` gesetzt, dann wird es als Fehler angesehen, wenn das Semaphor-Feld mit dem Schlüssel `key` bereits existiert (ist nur in Verbindung mit `IPC_CREAT` sinnvoll).

Der Funktionswert von `semget()` ist ein gültiger Semaphor-Deskriptor, falls die Funktion fehlerfrei ausgeführt werden konnte, `-1` anderenfalls. Dieser Semaphor-Deskriptor muss nun für alle weiteren Operationen mit diesem Semaphor-Feld verwendet werden. Außer den oben angeführten Fällen tritt ein Fehler auch dann auf, wenn ein Prozess ein Semaphor-Feld zu verwenden versucht, für das er nicht die erforderlichen Zugriffsberechtigungen besitzt, oder wenn die maximale Anzahl von Semaphor-Deskriptoren überschritten wird (systemweit).

Zieht man eine Analogie mit dem Filesystem, dann entspricht die Funktion `semget()` der Funktionen `fopen()`, `key` hat eine ähnliche Bedeutung wie der Filename und der Semaphor-Deskriptor entspricht einem Filepointer.

Für Operationen mit Semaphoren steht der System-Call `semop()` zur Verfügung. Der Parameter `semid` ist der Deskriptor, den man von `semget()` erhalten hat, `sops` ein Zeiger auf das Feld von Operationsbeschreibungen (d.h. jedes Element von `sops` definiert eine bestimmte Operation mit einer bestimmten Semaphor-Variablen) und `nsops` gibt die Größe dieses Feldes an. Der Funktionswert von `semop()` ist `0` für erfolgreiche Durchführung und `-1` im Fehlerfall.

Die Struktur `sembuf` sieht folgendermaßen aus:

```
struct sembuf
{
    unsigned short sem_num;  /* semaphore # */
    short sem_op;            /* semaphore operation */
```

```
    short sem_flg;          /* operation flags */
};
```

Die Struktur `sembuf` enthält die Information, auf welche Semphor-Variable des Feldes welche Operation anzuwenden ist. `sem_num` steht dabei für den Index der Semaphor-Variable im Semaphor-Feld, auf die diese Operation anzuwenden ist (`semid` und `sem_num` identifizieren daher eindeutig eine einzelne Semaphor-Variable).

`sem_op` gibt einen Operationscode an. Die Operation selbst wird nun folgendermaßen durchgeführt: Falls `sem_op` positiv ist, wird der Semaphorwert um `sem_op` erhöht. Ist `sem_op` gleich `0`, so wird der aktuelle Semaphorwert überprüft: Falls er `0` ist, setzt der Prozess normal fort, ist er größer als `0`, muss der Prozess warten, bis entweder der Semaphorwert `0` wird, das Semaphor-Feld `semid` aus dem System entfernt (gelöscht) wird oder `semop` durch ein Signal unterbrochen wird. In den letzten beiden Fällen terminiert `semop` natürlich mit einem Fehlercode.

Ist `sem_op` negativ und der Absolutwert kleiner oder gleich dem Wert der Semaphor-Variablen, dann wird die Semaphor-Variable um den Absolutbetrag von `sem_op` vermindert. Ist der Wert der Semaphor-Variablen kleiner als der Absolutwert von `sem_op`, muss der Prozess warten, bis der Wert der Semaphor-Variablen durch eine andere Operation erhöht wurde, das Semaphor-Feld `semid` aus dem System entfernt wird oder `semop` durch ein Signal unterbrochen wird. In den letzten beiden Fällen terminiert `semop` wieder mit einem entsprechenden Fehlercode.

Durch entsprechendes Setzen von `sem_flg` kann man das oben beschriebene Verhalten von `semop` modifizieren. Ist das Flag `IPC_NOWAIT` gesetzt, so wird auf keinen Fall auf das Eintreten einer Bedingung gewartet.

Ist für mindestens eine Operation die entsprechende Bedingung nicht erfüllt, so wird **keine** Operation ausgeführt und `semop` terminiert sofort (wenn `IPC_NOWAIT` gesetzt ist).

Mit dem System-Call `semctl()` können Semaphorkontrolloperationen durchgeführt werden. Die Union `semun` sieht folgendermaßen aus:

```
union semun
{
    int val;
    struct semid_ds *buf;
    ushort *array;
};
```

Die Parameter `semid` und `semnum` identifizieren dabei eindeutig eine einzelne Semaphor-Variable des Feldes `semid`. Der Parameter `cmd` bezeichnet das auszuführende Kommando, und `arg` ist ein Argument für `cmd`. Abhängig von `cmd` kann `arg` ein Integer-Wert, ein Zeiger auf eine Struktur vom Typ `semid_ds` oder ein Zeiger auf Feld von Short Integers sein.

Die wichtigsten Kommandos sind: `GETVAL` liefert den Wert der durch `semid` und `semnum` identifizierten Semaphor-Variablen als Funktionswert von `semctl`. `SETVAL` setzt den Wert der betreffenden Semaphor-Variablen auf `arg.val`. `GETALL` schreibt die Werte aller Semaphor-Variablen des Feldes `semid` in das Feld `arg.array`. `SETALL` ist die Umkehrung von `GETALL`. `IPC_RMID` löscht das Semaphor-Feld `semid`.

Siehe auch: semget(2), semop(2), semctl(2), intro(2)

Das folgende Beispiel zeigt die Verwendung von Semaphoren, um zu gewährleisten, dass mehrere Prozesse einen kritischen Abschnitt nicht gleichzeitig ausführen.

Gemeinsames Includefile

```
/*
 * Module:        Semaphore Example
 * File:          sm_gem.h
 * Version:       1.4
 * Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 * Last Changes:  9/16/99
 * Author:        Wilfried Elmenreich
 *                wilfried@vmars.tuwien.ac.at
 * Contents:      Common Include File
 *
 * FileID: sm_gem.h 1.4
 */

/*
 * ********************************************************* defines ***
 */

#define KEY 18225                                  /* key for semaphore */
#define SEM_ANZ 1                                 /* number of semaphores */
#define SEM_NR  0                                /* 1st semaphor in field */
#define PERM    0666                                   /* permission bits */

/*
 * ************************************************************** EOF ***
 */
```

Beispiel mit einem Semaphor zur Synchronisation

```
/*
 * Module:        Semaphore Example
 * File:          sm_bsp.c
 * Version:       1.4
 * Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 * Last Changes:  9/16/99
 * Author:        Wilfried Elmenreich
 *                wilfried@vmars.tuwien.ac.at
 * Contents:      Semaphore Example
 *
 * FileID: sm_bsp.c 1.4
 */

/*
```

```
 * ********************************************************** includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include "support.h"
#include "sm_gem.h"

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";                /* command name */
static int nSemID = -1;                                 /* semaphore ID */

/*
 * ********************************************************* functions ***
 */

int V(int semid)                              /* Dijkstra's V(s) [signal] */
{
    struct sembuf semp;

    semp.sem_num = 0;
    semp.sem_op = 1;
    semp.sem_flg = 0;

    /* increment semaphore by one */
    return semop(semid, &semp, 1);
}

int P(int semid)                                /* Dijkstra's P(s) [wait] */
{
    struct sembuf semp;

    semp.sem_num = 0;
    semp.sem_op = -1;
    semp.sem_flg = 0;

    /* decrement semaphore by one, but wait if value is less than one   */
    return semop(semid, &semp, 1);
}
```

```
void FreeResources(void)
{
    /* semaphore is not deleted, because some other instance might      */
    /* still use it                                                     */

}

void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}

void AllocateResources(void)
{
    if ((nSemID = semget(KEY,SEM_ANZ, PERM | IPC_CREAT )) == -1)
    {
        BailOut("Cannot create semaphore!");
    }
    else
    {
        if ((semctl(nSemID, SEM_NR, SETVAL, 1)) == -1)
        {
            BailOut("Cannot initialize semaphore!");
        }
    }
}

void Usage(void)
{
    (void) fprintf(stderr,"USAGE: %s\n",szCommand);

    BailOut((const char *) 0);
}

int main(int argc, char **argv)
{
    szCommand = argv[0];

    if (argc != 1)
```

```
    {
        Usage();
    }

    AllocateResources();

    if (P(nSemID) == -1)
    {
        BailOut("Cannot P semaphore!");
    }

    Critical();                             /* Critical section is here */

    if (V(nSemID) == -1)
    {
        BailOut("Cannot V semaphore!");
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************** EOF ***
 */
```

Programm zum Löschen des Semaphors

```
/*
 * Module:         Semaphore Example
 * File:           sm_clr.c
 * Version:        1.4
 * Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 * Last Changes:  9/16/99
 * Author:         Wilfried Elmenreich
 *                 wilfried@vmars.tuwien.ac.at
 * Contents:       Clear Semaphore Program
 *
 * FileID: sm_clr.c 1.4
 */

 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include "support.h"
#include "sm_gem.h"

/*
 * ********************************************************* defines ***
 */

/*
 * ********************************************************* globals ***
 */

const char *szCommand = "<not yet set>";    /* command name */
static int nSemID = -1;         /* semaphore ID */

/*
 * ****************************************************** prototypes ***
 */

void BailOut(const char *szMessage);

/*
 * ******************************************************* functions ***
 */

int P(int semid)                            /* Dijkstra's P(s) [wait] */
{
    struct sembuf semp;

    semp.sem_num = 0;
    semp.sem_op = -1;
    semp.sem_flg = 0;

    /* decrement semaphore by one, but wait if value is less than one  */
    return semop(semid, &semp, 1);
}

void FreeResources(void)
{
    if (nSemID != -1)
    {
    if (semctl(nSemID, SEM_ANZ, IPC_RMID, 0) == -1)
        {
            nSemID = -1;
```

```
            BailOut("Cannot remove semaphore!");
        }
        nSemID=-1;
    }
}

void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}

void AllocateResources(void)
{
    if ((nSemID = semget(KEY, SEM_ANZ, PERM)) == -1)
    {
        BailOut("No semaphore to remove!");
    }
}

void Usage(void)
{
    (void) fprintf(stderr,"USAGE: %s\n",szCommand);

    BailOut((const char *) 0);
}

int main(int argc, char **argv)
{
    szCommand = argv[0];

    if (argc != 1)
    {
        Usage();
    }

    AllocateResources();

    if (P(nSemID) == -1)
    {
        BailOut("Cannot P semaphore!");
```

```
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************* EOF ***
 */
```

Semaphor Library

Um nicht immer, wenn Semaphore verwendet werden, die Funktion `semop()` verwenden zu müssen, steht die Library *sem182* [3] zur Verfügung. Diese bietet folgende Funktionen:

```
#include <sem182.h>

int seminit(key_t key, int semperm, int initval)
int semrm(int semid)
int semgrab(key_t key)
int P(int semid)
int V(int semid)
```

Die Funktion `seminit()` erzeugt und initialisiert einen Semaphor. Der Returnwert dieser Funktion ist die `semid`, welche für die nachstehend beschriebenen Funktionen benötigt wird. Tritt jedoch ein Fehler auf (z.B. Semaphor existiert bereits), so wird `-1` zurückgeliefert.

Mit Hilfe des Parameters `semperm` werden die Zugriffsrechte für den Semaphor bestimmt. Der Semaphor wird mit dem Wert `initval` initialisiert. Der Aufruf

```
semid = seminit(9525000, 0600, 1);
```

erzeugt, zum Beispiel, einen Semaphor mit `key` 9525000 und initialisiert ihn mit 1. Die Zugriffsrechte werden entsprechend der Oktalzahl 0600 (analog zu chmod) gesetzt.

Mit der Funktion `semrm()` wird ein nicht mehr benötigter Semaphor gelöscht. `Semid` gibt hier an, welcher Semaphor zu entfernen ist.

Die Funktion `semgrab()` liefert die `semid` eines bereits existierenden Semaphors. Diese Funktion benötigt ein Prozess, der einen Semaphor benutzen möchte, den ein anderer Prozess erzeugt hat. Die beiden Funktionen `P()` und `V()` entsprechen in ihrer Funktion den gleichnamigen von Dijkstra definierten Funktionen. Sie benötigen als Parameter die `semid` des zu modifizierenden Semaphors.

Alle Funktionen liefern im Fehlerfall -1 zurück.

Grundsätzlich gilt, dass Semaphore mit `seminit()` initialisiert werden müssen, bevor sie mit `P()` und `V()` verwendet werden können. Abschließend müssen sie mit `semrm()` gelöscht werden.

[3] `http://www.vmars.tuwien.ac.at/download/sem182.tgz`

Die Prototypen der zur Verfügung gestellten Funktionen stehen in der Datei `<sem182.h>`. Durch Verwenden der Option `-lsem182` des `gcc` werden die Funktionen der Semaphorlibrary (`sem182.a`) in das Programm eingebunden. Wenn Sie das File `eindhoven.c` übersetzen wollen und dafür die oben genannten Semaphoroperationen benötigen, so muss dies durch Aufruf von

```
gcc -ansi -pedantic -Wall -g -c eindhoven.c
```

und

```
gcc -o eindhoven eindhoven.o -lsem182
```

erfolgen.

Semaphorfelder

Die Semaphorlibrary enthält auch Funktionen, die das Arbeiten mit Semaphorfeldern (Semaphorarrays) leichter machen. Folgende Funktionen werden zur Verfügung gestellt:

```
#include <msem182.h>

int mseminit(key_t key, int semperm, int nsems, ...)
int msemgrab(key_t key, int nsems)
int mP(int semid, int nsems, ...)
int mV(int semid, int nsems, ...)
int msemrm(int semid)
```

Die Funktion `mseminit()` legt ein Semaphorarray an. Die Parameter `key` und `semperm` geben (wie bei der Funktion `seminit()`) den zu verwendenden Key und die Zugriffsberechtigungen an. Der Parameter `nsems` ist die gewünschte Anzahl der Elemente des Arrays. Nach `nsems` *müssen* genau so viele Parameter folgen, wie das Array Elemente hat: Das erste Element des Arrays wird mit dem ersten Parameter nach `nsems` initialisiert, das zweite Element mit dem zweiten Parameter nach `nsems`, und so weiter.

Dazu ein Beispiel:

```
semarray = mseminit(8525659, 0600, 3, 0, 1, 0);
```

legt ein Semaphorfeld mit dem Key 8525659 an, auf das nur der Erzeuger Zugriff hat. Das Feld enthält drei Elemente, wobei das Erste und Dritte mit 0 initialisiert werden, und das Zweite mit 1.

Die beiden Funktionen `msemgrab()` und `msemrm()` entsprechen in ihrer Funktion den beiden Funktionen `semgrab()` und `semrm()`.

Die Funktionen `mP()` und `mV()` führen auf mehreren Elementen eines Semaphorfeldes die entsprechende Semaphoroperation durch. Alle Semaphoroperationen, die von *einem* Aufruf von `mP()` (bzw. von `mV()`) ausgelöst werden, werden *atomic* (nicht unterbrechbar) durchgeführt. Der erste Parameter ist wieder der Deskriptor, der von `mseminit()` oder `msemgrab()` zurückgeliefert wurde. Der zweite Parameter gibt die Anzahl der Elemente an, auf denen die Operation durchgeführt werden

soll. Die folgenden Parameter (genau so viele, wie der zweite Parameter angibt) geben die *Indizes* der Elemente an, auf die die Operation wirken soll.

Hierzu ein Beispiel: Der Aufruf

```
   status = mP(semarray, 2, 0, 1);
```

bewirkt, dass im Semaphorfeld `semarray` die Elemente mit den Indizes 0 und 1 einer unteilbaren P-Operation unterworfen werden.

Bei Verwendung dieser Funktionen muss beim Linken der Object-Files auch die Option `-lsem182` angegeben werden.

Beispiel

Das folgende Beispiel zeigt eine Synchronisation von einem Serverprozess mit beliebig vielen Clientprozessen.

Gemeinsames Includefile

```
/*
 *  Module:        Semaphore Library Example
 *  File:          sml_gem.h
 *  Version:       1.5
 *  Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 *  Last Changes:  9/16/99
 *  Author:        Wilfried Elmenreich
 *                 wilfried@vmars.tuwien.ac.at
 *  Contents:      Common Include File
 *
 *  FileID: sml_gem.h 1.5
 */

/*
 * ********************************************************** defines ***
 */

#define KEY1    182251                          /* key for semaphore 1 */
#define KEY2    182252                          /* key for semaphore 2 */
#define PERM    0666            /* permission bits for both semaphores */

/*
 * ************************************************************** EOF ***
 */
```

Serverprozess

```
/*
 *  Module:        Semaphore Library Example
 *  File:          sml_svr.c
 *  Version:       1.4
 *  Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 *  Last Changes:  9/16/99
 *  Author:        Wilfried Elmenreich
 *                 wilfried@vmars.tuwien.ac.at
 *  Contents:      Server Process
 *
 *  FileID: sml_svr.c 1.4
 */

/*
 * ********************************************************** includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sem182.h>

#include "support.h"
#include "sml_gem.h"

/*
 * ********************************************************** defines ***
 */

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
static int nSem1ID = -1;                              /* semaphore 1 ID */
static int nSem2ID = -1;                              /* semaphore 2 ID */

/*
 * ******************************************************** prototypes ***
 */

void BailOut(const char *szMessage);
```

```
/*
 * ********************************************************* functions ***
 */

void FreeResources(void)
{
    if (nSem1ID != -1)
    {
        if (semrm(nSem1ID) == -1)
        {
            nSem1ID = -1;
            BailOut("Cannot remove semaphore");
        }
        nSem1ID = -1;
    }
    if (nSem2ID != -1)
    {
        if (semrm(nSem2ID) == -1)
        {
            nSem2ID = -1;
            BailOut("Cannot remove semaphore");
        }
        nSem2ID = -1;
    }
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((nSem1ID = seminit(KEY1, PERM, 0)) == -1)
    {
        BailOut("Cannot create semaphore!");
    }
    if ((nSem2ID = seminit(KEY2, PERM, 1)) == -1)
```

```
    {
        BailOut("Cannot create semaphore!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s\n", szCommand);

    BailOut((const char *) 0);
}


void Handler(int nSignal)
{
    FreeResources();

    exit(EXIT_SUCCESS);
}


int main(int argc, char **argv)
{
    szCommand = argv[0];

    if (argc != 1)
    {
        Usage();
    }

    (void) signal(SIGTERM, Handler);
    (void) signal(SIGINT, Handler);
    (void) signal(SIGQUIT, Handler);

    AllocateResources();

    while(1)
    {
        if (P(nSem1ID) == -1)
        {
            BailOut("Cannot P semaphore!");
        }

        ResponseFromServer();

        if (V(nSem2ID) == -1)
```

```
        {
            BailOut("Cannot V semaphore!");
        }
    }                                                       /* forever */
}

/*
 * ************************************************************ EOF ***
 */
```

Clientprozess

```
/*
 *  Module:         Semaphore Library Example
 *  File:           sml_clnt.c
 *  Version:        1.4
 *  Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 *  Last Changes:  9/16/99
 *  Author:         Wilfried Elmenreich
 *                  wilfried@vmars.tuwien.ac.at
 *  Contents:       Client Process
 *
 *  FileID: sml_clnt.c 1.4
 */

/*
 * ********************************************************** includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sem182.h>

#include "support.h"
#include "sml_gem.h"

/*
 * ********************************************************** defines ***
 */

/*
 * ********************************************************** globals ***
 */
```

```
const char *szCommand = "<not yet set>";                /* command name */
static int nSem1ID = -1;                                /* semaphore 1 ID */
static int nSem2ID = -1;                                /* semaphore 2 ID */

/*
 * ********************************************************* functions ***
 */

void FreeResources(void)
{
                                                    /* Nothing to remove */
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((nSem1ID = semgrab(KEY1)) == -1)
    {
        BailOut("Cannot grab semaphore!");
    }
    if ((nSem2ID = semgrab(KEY2)) == -1)
    {
        BailOut("Cannot grab semaphore!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s\n", szCommand);

    BailOut((const char *) 0);
}
```

```
int main(int argc, char **argv)
{
    szCommand = argv[0];

    if (argc != 1)
    {
        Usage();
    }

    AllocateResources();

    if (P(nSem2ID) == -1)
    {
        BailOut("Cannot P semaphore!");
    }

    RequestFromClient();

    if (V(nSem1ID) == -1)
    {
        BailOut("Cannot V semaphore!");
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************* EOF ***
 */
```

4.5.2 Sequencer und Eventcounts

Sequencer und Eventcounts können ebenso wie Semaphore zur Synchronisation paralleler, unter Umständen nicht verwandter, Prozesse verwendet werden. Sequencer und Eventcounts sind Mechanismen zur direkten Steuerung der Reihenfolge von Aktionen. Sie bestehen konzeptionell aus folgenden Komponenten:

`Eventcount E`: ganzzahlige Variable zum Zählen der Vorkommnisse von Ereignissen. Der Anfangswert von E ist 0.

- `await (E, v)`: blockiert Prozess bis E $\geq$ v
- `advance (E)`: erhöht E um 1
- `read (E)`: liefert den aktuellen Wert von E

`Sequencer S`: ganzzahlige Variable mit Anfangswert 0, die verwendet wird, um die Abfolge von Ereignisse zu entscheiden

`ticket (S)`: atomare Aktion, liefert den Wert des Sequencers S und erhöht anschließend den Wert von S um 1

Eventcounts können mit Sequencer kombiniert verwendet werden:

- **Eventcounts ohne Sequencer** bei fixer Menge von parallelen Prozessen mit klar definiertem Synchronisationsmuster.
  Beispiel: Fließbandarbeit (konstanter periodischer Ablauf)
- **Eventcounts mit Sequencern** wenn sich neu ankommende Prozesse mit den laufenden Prozessen aufsynchronisieren müssen (z.B. Server mit mehreren Clients).
  Beispiel: Ausgabe von Nummern und Warten auf Aufruf an einem Schalter (Behörde, Arzt, etc.)

Für die Programmierung mit Sequencern und Eventcounts steht eine Library (`seqev.a`[4]) zur Verfügung, die Sequencer und Eventcounts unter Verwendung von Semaphoren und einem Shared Memory implementiert.

Sequencer

Folgende Funktionen stehen in der Library `seqev.a` für die Manipulation von Sequencer zur Verfügung:

```
#include <seqev.h>

sequencer_t *create_sequencer(key_t key);
long sticket(sequencer_t *sequencer);
int rm_sequencer(sequencer_t *sequencer);
```

Die Funktion `create_sequencer()` dient dazu, einen Sequencer anzulegen, bzw. auf einen existierenden Sequencer zuzugreifen. Der erste Aufruf der Funktion `create_sequencer()` mit einem bestimmten `key` erzeugt einen Sequencer mit dem angegebenen `key` und initialisiert ihn. Dieser Key *muss eindeutig sein*, d.h., es darf kein Semaphor oder Shared Memory-Bereich mit diesem Key existieren. Der Returnwert ist ein Zeiger vom Typ `sequencer_t`, der (vergleichbar mit einem `FILE *`) zum Zugriff auf den Sequencer verwendet wird. Wenn beim Anlegen des Sequencers ein Fehler auftritt, dann liefert die Funktion `NULL` zurück und `errno` wird auf einen Wert gesetzt, aus dem die Fehlerursache ersichtlich ist. Wird `create_sequencer()` mit demselben `key` weitere Male aufgerufen, dann wird eine Referenz auf den bereits existierenden Sequencer zurückgeliefert. Damit erhält man in verschiedenen Prozessen Zugriff auf den gleichen Sequencer.

Mit Hilfe der Funktion `sticket()` kann dem entsprechenden Sequencer ein 'Ticket' entnommen werden. Sie liefert die Anzahl der Aufrufe von `sticket` mit diesem Sequencer *vor* dem jetztigen

[4] `http://www.vmars.tuwien.ac.at/download/SE.tgz`

Aufruf (also der Reihe nach folgende Werte 0, 1, 2,...), d.h., dass das erste Ticket, das gezogen wird, den Wert `0` hat! Im Fehlerfall, liefert sie `-1` und setzt `errno`.

Die Funktion `rm_sequencer()` dient zum Löschen des entsprechenden Sequencers. Sequencer werden nicht automatisch beim Terminieren des Prozesses gelöscht, sondern müssen mit Hilfe der Funktion `rm_sequencer()` gelöscht werden, sobald sie nicht mehr gebraucht werden. Die Funktion liefert den Wert `-1` zurück, wenn beim Löschen des Sequencers ein Fehler auftritt, ansonsten den Wert `0`.

Im Fehlerfall wird außerdem die Variable `errno` auf einen Wert gesetzt, der die Fehlerursache angibt.

Eventcounts

Für Eventcounts stehen in der Library `seqev.a` folgende Funktionen zur Verfügung:

```
#include <seqev.h>

eventcounter_t *create_eventcounter(key_t key);
int eawait(eventcounter_t *eventcounter, long value);
long eread(eventcounter_t *eventcounter);
int eadvance(eventcounter_t *eventcounter);
int rm_eventcounter(eventcounter_t *eventcounter);
```

Die Funktion `create_eventcounter()` dient dazu, einen Eventcounter anzulegen, bzw. auf einen existierenden Eventcounter zuzugreifen. Diese Funktion erzeugt einen Eventcount mit dem angegebenen Key. Der Eventcounter wird auf den Wert `0` initialisiert. Das bei `create_sequencer` zu den Themen Return-Wert und Fehlerbehandlung Gesagte gilt sinngemäß.

Die Funktion `eawait()` verzögert die Abarbeitung des Prozesses so lange, bis der Wert des Eventcounts `eventcounter` größer oder gleich dem Wert `value` ist. Die Funktion liefert `0`, wenn sie erfolgreich ausgeführt wurde, und `-1` im Fehlerfall.

Die Funktion `eread()` liefert den momentanen Wert des Eventcounts zurück, oder `-1`, wenn ein Fehler aufgetreten ist. Wird der Wert des Eventcounts während der Ausführung von `eread()` verändert, so ist für den Returnwert von `eread()` nur garantiert, dass er zwischen dem alten und dem neuen Wert des Eventcounts liegt. Diese Funktion wird (ebenso wie das Auslesen eines Semaphor) nur sehr selten benötigt. Falls Sie bei der Lösung ihrer Übungsaufgabe `eread()` verwendet haben, so ist diese Lösung mit großer Wahrscheinlichkeit falsch, oder unnötig aufwändig gelöst.

Die Funktion `eadvance()` erhöht den Wert des Eventcounts um 1. Die Funktion liefert `0`, wenn sie erfolgreich war, und `-1` im Fehlerfall.

Die Funktion `rm_eventcounter()` dient zum Löschen des entsprechenden Eventcounts. Eventcounts werden ebenfalls nicht automatisch beim Terminieren des Prozesses gelöscht, sondern müssen mit Hilfe der Funktion `rm_eventcounter()` gelöscht werden, sobald sie nicht mehr gebraucht werden. Diese Funktion liefert `-1` zurück, wenn beim Löschen des Eventcounts ein Fehler auftritt, `0` sonst.

Im Fehlerfall setzen alle Funktionen die Variable `errno` auf einen Wert, der die Fehlerursache angibt.

Hinweise

Wenn durch einen Programmierfehler Sequencer oder Eventcounts nicht richtig gelöscht wurden, dann löschen Sie bitte die dazugehörigen Semaphore und Shared-Memory Segmente mittels `ipcrm`.

Um die angegebenen Funktionen in einem Programm verwenden zu können, ist es notwendig, die entsprechende Library (`seqev.a`) einzubinden. Das geschieht, indem beim Linken hinter den Objekt-Files die Option `-lseqev` angegeben wird.

Beispiel

Sequencer Keys

```
/*
 *  Module:        Sequencer & Eventcount Example
 *  File:          seqevkeys.h
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  8/26/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Keys for Sequencer and Eventcounts
 *
 *  FileID: seqevkeys.h 1.1
 */

/*
 * ********************************************************* defines ***
 */

#define SEQ_KEY 182251                              /* sequencer's key */
#define EVC_KEY 182252                             /* eventcount's key */

/*
 * ************************************************************* EOF ***
 */
```

Schreibender Prozess

```
/*
 *  Module:        Sequencer & Eventcount Example
 *  File:          writer.c
 *  Version:       1.2
```

```
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  9/16/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Writer Process
 *
 *  FileID: writer.c 1.2
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <seqev.h>

#include "support.h"
#include "seqevkeys.h"

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
static sequencer_t *pSequencer = (sequencer_t *) 0;       /* sequencer */
static eventcounter_t *pEventcount = (eventcounter_t *) 0;      /* evc */

/*
 * ******************************************************* prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */

/*
 * ******************************************************** functions ***
 */

void FreeResources(void)
{
    if (pEventcount!= (eventcounter_t*)0) /*eventcount already created?*/
    {
        if (rm_eventcounter(pEventcount) == -1)
        {
            pEventcount = (eventcounter_t *) 0;/*eventcount not present*/
```

```
            BailOut("Can't remove eventcount!");
        }

        pEventcount = (eventcounter_t *) 0;  /* eventcount not present */
    }

    if (pSequencer != (sequencer_t *) 0) /* sequencer already created? */
    {
        if (rm_sequencer(pSequencer) == -1)
        {
            pSequencer = (sequencer_t *) 0;   /* sequencer not present */

            BailOut("Can't remove sequencer!");
        }

        pSequencer = (sequencer_t *) 0;       /* sequencer not present */
    }
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((pSequencer = create_sequencer(          /* create a sequencer */
        SEQ_KEY
        )) == (sequencer_t *) 0)
    {
        BailOut("Can't create sequencer!");
    }

    if ((pEventcount = create_eventcounter(    /* create an eventcount */
        EVC_KEY
        )) == (eventcounter_t *) 0)
    {
```

```
        BailOut("Can't create eventcounter!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s\n", szCommand);

    BailOut((const char *) 0);
}


int main(int argc, char **argv)
{
    long nTicket = -1;

    szCommand = argv[0];                          /* store command name */

    if (argc != 1)                                   /* check arguments */
    {
        Usage();
    }

    AllocateResources();

    if ((nTicket = sticket(               /* draw ticket from sequencer*/
        pSequencer)
        ) == -1)
    {
        BailOut("Can't obtain ticket!");
    }

    if (eawait(                          /* wait for till it's our turn */
        pEventcount,
        nTicket
        ) == -1)
    {
        BailOut("Can't wait for turn!");
    }

    CriticalSection();                            /* do critical stuff */

    if (eadvance(           /* advance eventcounter => next one's turn */
        pEventcount
        ) == -1)
    {
```

```
        BailOut("Can't advance eventcount!");
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************* EOF ***
 */
```

4.6 Shared Memory

Parallele Prozesse können direkt über Shared Memory-Bereiche (Speicherbereich mit Zugriffsmöglichkeit durch mehrere Prozesse) miteinander kommunizieren. Für das Arbeiten mit Shared Memory-Bereiche stehen die folgenden Funktionen zur Verfügung.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, char *shmaddr, int shmflg);
int shmdt(char *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Ein Prozess legt einen Shared Memory-Bereich an, den dann andere Prozesse anfordern und mitverwenden können. Dazu muss jeder Prozess zunächst einen gemeinsamen Speicherbereich mit dem System-Call `shmget()` vom System anfordern: Dabei haben `key` und `shmflg` dieselbe Bedeutung wie beim System-Call `semget()` (siehe Abschnitt 4.5.1, Seite 256). Der spezielle Schlüssel `IPC_PRIVATE`, sowie die speziellen Flags `IPC_CREAT` und `IPC_EXCL` werden ebenfalls wie bei den Semaphoren verwendet.

Der Parameter `size` legt die Größe des angeforderten Speicherbereichs fest. Falls der durch `key` identifizierte Speicherbereich bereits existiert, kann `size` auch `0` sein, da in diesem Fall die Größe bereits feststeht.

Der Funktionswert von `shmget()` ist ein gültiger Shared Memory-Deskriptor, falls `shmget()` fehlerfrei ausgeführt werden konnte, `-1` anderenfalls. Dieser Shared Memory-Deskriptor muss nun für alle weiteren Operationen mit diesem Speicherbereich verwendet werden. Die möglichen Fehlerfälle entsprechen den Fehlerfällen für Semaphore. Außerdem gibt es noch (systemabhängige) untere und obere Grenzen von `size`. Weiters ist es ein Fehler, wenn bei der Anforderung eines existierenden Speicherbereiches `size` größer als die Größe dieses existierenden Segments ist.

Um das mit Hilfe von `shmget()` angeforderte Speichersegment verwenden zu können, muss es in das Datensegment des Prozesses eingehängt werden. Dazu dient der System-Call `shmat()`. Hierbei ist `shmid` ein Shared Memory-Deskriptor, der als Ergebnis eines vorausgegangenen Aufrufes von

`shmget()` erhalten wurde. Für den so identifizierten Speicherbereich wird nun Speicherplatz reserviert und an der Adresse `shmaddr` eingetragen. Falls `shmaddr` gleich `0` ist (das wird der häufigste Fall sein!!), so wird die Adresse vom System vergeben. Wird in `shmflg` `SHM_RDONLY` verwendet, dann kann das Shared Memory-Segment in der Folge nur gelesen werden, sonst ist prinzipiell Lesen und Schreiben erlaubt.

Das Ergebnis von `shmat()` ist die Startadresse des neuen Shared Memory-Segmentes, bzw. `-1` falls irgendein Fehler aufgetreten ist. Mit Hilfe dieser Startadresse kann das Shared Memory-Segment nun benutzt werden.

Benötigt ein Prozess ein Shared Memory-Segment nicht mehr, dann kann er es durch Aufruf des System-Calls `shmdt()` aus seinem Adressbereich entfernen. Das zu entfernende Shared Memory-Segment wird dabei durch `shmaddr` identifiziert, wobei `shmaddr` das Ergebnis eines vorherigen Aufrufes von `shmat()` ist. Das Ergebnis von `shmdt()` ist `-1` im Fehlerfall, `0` sonst.

Der Inhalt eines Shared Memory-Segmentes bzw. das Segment selbst werden durch `shmdt()` *nicht* zerstört oder gelöscht, ein entferntes Segment könnte z.B. durch neuerlichen Aufruf von `shmat()` wieder in den Adressraum des Prozesses eingefügt werden. Der Inhalt des Segmentes wäre dann durch die Schreiboperationen aller anderen Prozesse, die dieses Segment in der Zwischenzeit verwendet haben, bestimmt.

Ähnlich wie Semaphor-Felder müssen Shared Memory-Segmente explizit gelöscht werden. Dazu verwendet man den System-Call `shmctl()`: Der Parameter `shmid` identifiziert dabei eindeutig ein Shared Memory-Segment, `cmd` legt das auszuführende Kommando fest, und `buf` ist eine Datenstruktur zur Übernahme oder Übergabe von Statusinformationen. Das Kommando `IPC_RMID` löscht das betreffende Shared Memory-Segment.

Siehe auch: `shmget(2)`, `shmop(2)`, `shmctl(2)`, `intro(2)`

4.6.1 Beispiel

Das folgende Beispiel zeigt zwei Prozesse, die über ein Shared Memory-Segment Daten austauschen, wobei ein Prozess schreibt und der andere liest. Die Synchronisation der beiden Prozesse erfolgt in diesem Beispiel mittels Sequencer und Eventcounts.

Synchronisiert werden muss (in den meisten Fällen) insbesondere auch das Löschen des Shared Memory-Segmentes: Der schreibende Prozess darf es nicht löschen, bevor der lesende Prozess alle gesendeten Daten verarbeiten konnte, der Leser darf es nicht löschen, bevor der Schreiber alles gesendet hat. Eine Möglichkeit ist die Festlegung eines speziellen Datenwertes, den der Schreiber zum Schluss in das Shared Memory-Segment schreibt. Sobald der Leser diesen speziellen Wert liest, weiß er, dass der Schreiber fertig ist, und kann nun das Shared Memory-Segment löschen.

Das Include File `shmtype.h`, welches sowohl vom Client- als auch vom Serverprozess verwendet wird, definiert eine Struktur für das Shared Memory-Segment und legt Konstanten für die Schlüssel der einzelnen Synchronisations- und Kommunikationskonstrukte (`SHM_KEY`, `SEQ_KEY`, `EVC_KEY`) un deren Permissions (`PERM`) fest.

Shared Memory Definition (Header File)

```
/*
 *  Module:        Shared Memory Example
 *  File:          shmtype.h
 *  Version:       1.1
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  8/26/99
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Shared Memory Type
 *
 *  FileID: shmtype.h 1.1
 */

/*
 * ********************************************************* includes ***
 */

#include <limits.h>           /* the constant PATH_MAX is defined here */

/*
 * ********************************************************** defines ***
 */

#define SHM_KEY 182251                        /* key for shared memory */
#define SEQ_KEY 182252                            /* key for sequencer */
#define EVC_KEY 182253                           /* key for eventcount */
#define PERM 0666                                   /* permission bits */

/*
 * ********************************************************** typedefs ***
 */

typedef struct                                /* shared memory typedef */
{
    char szFileName[PATH_MAX];                                 /* filename */
    unsigned int nCount;                          /* number of copies */
} sharedmem_t;

/*
 * ************************************************************* EOF ***
 */
```

Client Prozess

```
/*
 *  Module:        Shared Memory Example
 *  File:          client.c
 *  Version:       1.3
```

```
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  1/20/00
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
 *  Contents:      Client Process
 *
 *  FileID: client.c 1.3
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <seqev.h>

#include "support.h"
#include "shmtype.h"

/*
 * ********************************************************** defines ***
 */

#define NUMBER_OF_COPIES 3

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
static int nSharedMemID = -1;                         /* shared mem ID */
static sharedmem_t *pSharedMem = (sharedmem_t *) -1;     /* shared mem */
static sequencer_t *pSequencer = (sequencer_t *) 0;       /* sequencer */
static eventcounter_t *pEventcount = (eventcounter_t *) 0;      /* evc */

/*
 * ******************************************************* prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */
```

```
/*
 * ********************************************************* functions ***
 */

void FreeResources(void)
{
    if (pSharedMem != (sharedmem_t *) -1)   /* shared memory attached? */
    {
        if (shmdt((void *) pSharedMem) == -1)
        {
            pSharedMem= (sharedmem_t *) -1;/*shared memory not attached*/

            BailOut("Can't detach shared memory!");
        }

        pSharedMem = (sharedmem_t *) -1; /* shared memory not attached */
    }
}


void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((nSharedMemID = shmget(         /* obtain shared memory handle */
        SHM_KEY,
        sizeof(sharedmem_t),
        PERM
        )) == -1)
    {
        BailOut("Can't obtain handle to shared memory!");
    }

    if ((pSharedMem = (sharedmem_t *) shmat(   /* attach shared memory */
        nSharedMemID,
```

```
        (const void *) 0,
        0
        )) == (sharedmem_t *) -1)
    {
        BailOut("Can't attach shared memory!");
    }

    if ((pSequencer = create_sequencer(          /* create a sequencer */
        SEQ_KEY
        )) == (sequencer_t *) 0)
    {
        BailOut("Can't create sequencer!");
    }

    if ((pEventcount = create_eventcounter(       /* create eventcount */
        EVC_KEY
        )) == (eventcounter_t *) 0)
    {
        BailOut("Can't create eventcounter!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s <filename>\n", szCommand);

    BailOut((const char *) 0);
}


int main(int argc, char **argv)
{
    long nTicket;

    szCommand = argv[0];                          /* store command name */

    if (argc != 2)                                   /* check arguments */
    {
        Usage();
    }

    if (strlen(argv[1]) >= PATH_MAX)       /* check length of argument */
    {
        PrintError("Warning: filename too long (will be truncated)!");
    }
```

```
    AllocateResources();

    if ((nTicket = sticket(               /* draw ticket from sequencer*/
        pSequencer)
        ) == -1)
    {
        BailOut("Can't obtain ticket!");
    }

    if (eawait(                          /* wait for till it's our turn */
        pEventcount,
        nTicket * 2           /* server's turn between any two clients */
        ) == -1)
    {
        BailOut("Can't wait for turn!");
    }

    (void) strncpy(pSharedMem->szFileName, argv[1], PATH_MAX);
    pSharedMem->szFileName[PATH_MAX - 1] = '\0';   /* terminate string */

    pSharedMem->nCount = NUMBER_OF_COPIES;

    if (eadvance(           /* advance eventcounter => next one's turn */
        pEventcount
        ) == -1)
    {
        BailOut("Can't advance eventcount!");
    }

    FreeResources();

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************* EOF ***
 */
```

Server Prozess

```
/*
 *  Module:        Shared Memory Example
 *  File:          server.c
 *  Version:       1.3
 *  Creation date: Fri Aug 20 15:18:51 MET DST 1999
 *  Last changes:  1/20/00
 *  Author:        Thomas M. Galla
 *                 tom@vmars.tuwien.ac.at
```

```
 *  Contents:      Server Process
 *
 *  FileID: server.c 1.3
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <seqev.h>

#include "support.h"
#include "shmtype.h"

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";               /* command name */
volatile static int nSharedMemID = -1;                /* shared mem ID */
volatile static sharedmem_t *pSharedMem = (sharedmem_t *) -1;
volatile static sequencer_t *pSequencer = (sequencer_t *) 0;
volatile static eventcounter_t *pEventcount = (eventcounter_t *) 0;

/*
 * ******************************************************** prototypes ***
 */

void BailOut(const char *szMessage);            /* forward declaration */

/*
 * ********************************************************* functions ***
 */

void FreeResources(void)
{
    if (pEventcount != (eventcounter_t *) 0)       /* already created? */
    {
        if (rm_eventcounter((eventcounter_t *) pEventcount) == -1)
```

```
        {
            pEventcount = (eventcounter_t *) 0;          /* not present*/

            BailOut("Can't remove eventcount!");
        }

        pEventcount = (eventcounter_t *) 0;    /* remember not present */
    }

    if (pSequencer != (sequencer_t *) 0) /* sequencer already created? */
    {
        if (rm_sequencer((sequencer_t *) pSequencer) == -1)
        {
            pSequencer = (Sequencer_t *) 0;   /* sequencer not present */

            BailOut("Can't remove sequencer!");
        }

        pSequencer = (Sequencer_t *) 0;       /* sequencer not present */
    }

    if (pSharedMem != (sharedmem_t *) -1)   /* shared memory attached? */
    {
        if (shmdt((void *) pSharedMem) == -1)
        {
            pSharedMem = (sharedmem_t *) -1;       /* shm not attached */

            BailOut("Can't detach shared memory!");
        }

        pSharedMem = (sharedmem_t *) -1; /* shared memory not attached */
    }

    if (nSharedMemID != -1)             /* shared mem already created? */
    {
        if (shmctl(nSharedMemID, IPC_RMID, (struct shmid_ds *) 0) == -1)
        {
            nSharedMemID = -1;            /* shared memory not created */

            BailOut("Can't remove shared memory!");
        }

        nSharedMemID = -1;                /* shared memory not created */
    }
}
```

```
void BailOut(const char *szMessage)
{
    if (szMessage != (const char *) 0)         /* print error message? */
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if ((nSharedMemID = shmget(    /* create shared memory exclusively */
        SHM_KEY,
        sizeof(sharedmem_t),
        PERM | IPC_CREAT | IPC_EXCL
        )) == -1)
    {
        BailOut("Can't create shared memory!");
    }

    if ((pSharedMem = (sharedmem_t *) shmat(   /* attach shared memory */
        nSharedMemID,
        (const void *) 0,
        0
        )) == (sharedmem_t *) -1)
    {
        BailOut("Can't attach shared memory!");
    }

    if ((pSequencer = create_sequencer(          /* create a sequencer */
        SEQ_KEY
        )) == (sequencer_t *) 0)
    {
        BailOut("Can't create sequencer!");
    }

    if ((pEventcount = create_eventcounter(       /* create eventcount */
        EVC_KEY
        )) == (eventcounter_t *) 0)
    {
        BailOut("Can't create eventcounter!");
    }
```

```
}


void Usage(void)
{
    (void) fprintf(stderr, "USAGE: %s\n", szCommand);

    BailOut((const char *) 0);
}


void Handler(int nSignal)
{
    FreeResources();

    exit(EXIT_SUCCESS);
}


int main(int argc, char **argv)
{
    long nNextTurn = 1;      /* client has to fill shared memory first */
    int i = 0;

    szCommand = argv[0];                          /* store command name */

    if (argc != 1)                                   /* check arguments */
    {
        Usage();
    }

    (void) signal(SIGTERM, Handler);          /* install signal handler */
    (void) signal(SIGINT, Handler);
    (void) signal(SIGQUIT, Handler);

    AllocateResources();

    while (1)
    {
        if (eawait( /* wait till it's our turn */
            (eventcounter_t *) pEventcount,
            nNextTurn
            ) == -1)
        {
            BailOut("Can't wait for turn!");
        }
```

```
        for (i = 0; i < pSharedMem->nCount; i++)
        {
            PrintFile((const char *) pSharedMem->szFileName);
        }

        if (eadvance(       /* advance eventcounter => next one's turn */
            (eventcounter_t *) pEventcount
            ) == -1)
        {
            BailOut("Can't advance eventcount!");
        }

        nNextTurn += 2;              /* next turn is after next client */
    }

    return 0;                                          /* not reached */
}

/*
 * ************************************************************ EOF ***
 */
```

4.7 Synchronisationsbeispiele

Es folgen einige Beispiele zu expliziter Synchronisation im Allgemeinen.

4.7.1 Reader Writer

Aufgabenstellung Der Zugriff auf ein File wird synchronisiert. Zu einem beliebigen Zeitpukt sollen beliebig viele Reader und Writer Zugriff auf das File wünschen. Mit Semaphoren ist sicherzustellen, dass entweder

- genau ein Writer Zugriff auf das File hat (d.h. kein anderer Reader oder Writer kann zur selben Zeit vom File lesen oder es beschreiben) oder
- 1 bis maximal N Reader gleichzeitig vom File lesen (es darf aber kein Writer Zugriff haben) oder
- kein Prozess auf das File zugreift.

Hierbei ist N eine Integer Zahl größer gleich 0.

Das Problem kann mit **Semaphoren** wie folgt gelöst werden:

Initialisierungen:

```
init(R, N);
init(W, 1);
```

Reader	Writer
	`P(W);`
`P(R);`	`for (i = 0; i < N; i++)` `  P(R);`
`/* critical section */`	`/* critical section */`
`V(R);`	`for (i = 0; i < N; i++)` `  V(R);`
	`V(W);`

Semaphor W stellt sicher, dass nur ein Writer "Read Permissions" sammelt (der Writer sammelt Read Permissions indem er den Wert von Semaphor R erniedrigt). Hat ein Writer alle Read Permissions eingezogen (d.h. der Wert von R ist 0) ist sichergestellt, dass sich kein Reader mehr in der Critical Section befindet.

Anmerkung Es sei hier nocheinmal betont, dass, was nicht aus dem Programmtext sondern aus der Aufgabenstellung hervorgeht, sowohl Reader als auch Writer beliebig oft aufgerufen werden können.

Anmerkung Diese Lösung mit Semaphoren garantiert nur die Erfüllung oben angeführter Anforderungen. Sie garantiert nicht, dass ein Writer jemals zum Schreiben kommt (*Fairness*). Es hängt von der jeweiligen Implementierung der Semaphore ab, in welcher Reihenfolge `P(Sem)` Operationen bedient werden. Stets neu ankommende Reader könnten in diesem Beispiel theoretisch verhindern, dass ein Writer jemals zum Schreiben kommt (*Starvation*). Gewährleistet die Implementierung der Semaphore jedoch, dass `P(Sem)` Operationen stets vor einem später auftretenden `P(Sem)` Aufruf bedient werden, so kann bei dieser Beispiellösung keine Starvation auftreten. (Im Allgemeinen sind Semaphore mittels einer Queue versehen, die nach dem FIFO-Prinzip (First In - First Out) arbeitet, d.h. `P(Sem)` Operationen werden in der Reihenfolge ihres Auftretens bedient)

4.7.2 Client Server

Aufgabenstellung Ein in einer Endlosschleife laufender Server liest aus einem Shared Memory, verarbeitet die gelesene Information und schreibt wieder in das Shared Memory. Clients schreiben Anforderungen in das Shared Memory, lassen diese durch den Server bearbeiten und lesen die Antwort wieder aus dem Shared Memory aus. Die Synchronisation soll genau einen Server und beliebig

viele Clients berücksichtigen, d.h. es gibt nur einen Serverprozess aber beliebig viele Clientprozesse. Es sollen nur sinnvolle Abarbeitungen zugelassen werden. Sinnvoll bedeutet hier, dass der Server sobald ein Client auf das Shared Memory geschrieben hat die Daten genau einmal liest, sie verarbeitet, das Shared Memory mit dem Ergebnis beschreibt und derselbe Client, der zuletzt die Anforderung geschrieben hat, die Daten wieder ausliest.

Das Beispiel kann mit **Sequencer und Eventcounts** wie folgt gelöst werden:

Initialisierungen:

```
create_evc(ec);
create_seq(seq);
```

Client

```
t = ticket(seq);
await(ec, 3 * t);
/* write shm */
advance(ec);

await(ec, 3 * t + 2);
/* read shm */
advance(ec);
```

Server

```
t = 1;

while (1)
{
    await(ec, t);
    /* read and write shm */
    advance(ec);
   t = t + 3;
}
```

Anmerkung Der Sequencer dient dazu, die Reihenfolge, in der mehrere Clients Requests an den Server senden dürfen, festzulegen. Für den Server ist die Abarbeitungsreihenfolge von vornherein bekannt. Es ist daher kein Sequencer erforderlich. Ein Wert gleich 1 modulo 3 des Eventcounts bedeutet, dass ein Client eine Anforderung in das Shared Memory geschrieben hat. Der Server bearbeitet die Anfrage und erhöht den Eventcount um anzuzeigen, dass der Client nun das Ergebnis auslesen darf.

Es ist zu beachten, dass die Variable `t` eine für den jeweiligen Prozess *lokale* Variable ist.

4.7.3 Busy Waiting

Beim **Busy Waiting** verbringt ein Prozess die Zeit bis eine Bedingung erfüllt ist mit *ständigem Abfragen* dieser Bedingung. Ein Beispiel ist etwa das wiederholte Auslesen der Uhrzeit um eine Aktion zu einem bestimmten Zeitpunkt ausführen zu können. Da Busy Waiting unnötig CPU Zeit konsumiert, ist es im Allgemeinen als *fehlerhafte* Implementierung des Wartens anzusehen.

Ein weiteres Beispiel ist der Versuch *Bedingungssynchronisation* mit nur einem Semaphor zu realisieren.

Aufgabenstellung Zwei Prozesse sollen abwechselnd auf ein Shared Memory zugreifen (kein Prozess darf zweimal hintereinander auf das Shared Memory zugreifen).

Häufig findet sich folgender fehlerhafte Lösungsansatz (Busy Waiting):

Initialisierungen:

```
init(S, 1);
a = 0;
```

Prozess 1

```
while(1)
{
    P(S);
    if (a == 1)
    {
        a = 0;
        /* do something */
    }

    V(S);
}
```

Prozess 2

```
while(1)
{
    P(S);
    if (a == 0)
    {
        a = 1;
        /* do something */
    }

    V(S);
}
```

Achtung, diese Lösung ist falsch, da sie Busy Waiting implementiert. Die Variable `a` ist hier als im Shared Memory liegend anzunehmen. `/* do something */` steht für die eigentlichen kritischen Operationen (auf dem Shared Memory).

Lösung

Initialisierungen:

```
init(S1, 1);
init(S2, 0);
```

Prozess 1

```
while(1)
{
    P(S1)

    /* do something */

    V(S2)
}
```

Prozess 2

```
while(1)
{
    P(S2)

    /* do something */

    V(S1)
}
```

Anmerkung Im Gegensatz zum Beispiel *Reader Writer*, bei dem beide Programme beliebig oft aufgerufen werden können (d.h. dass beliebig viele Prozesse gleichzeitig aktiv sein können) und zum Clientprogramm vom *Client Server* Beispiel, welches auch beliebig oft aufgerufen werden kann (also beliebig viele Clientprozesse können gleichzeitig aktiv sein), dürfen hier beide Programme nur genau einmal aufgerufen werden.

Die mögliche Anzahl von Prozessen, die das selbe Programm abarbeiten dürfen, folgt aus der Aufgabenstellung.

4.8 Verwaltung paralleler Prozesse

Dieses Kapitel zeigt wie in UNIX parallele Prozesse *dynamisch* erzeugt werden können. Zunächst wird dieser Mechanismus am Beispiel der Kommandoausführung der Shell beschrieben.

Wie bereits im Kapitel über UNIX erwähnt, ist die Shell ein Programm, das die vom Benutzer eingegebenen Kommandos liest und die entsprechenden Programme ausführt. Einige Kommandos sind direkt in der Shell selbst realisiert, die meisten jedoch sind eigene Programme. Wenn nun die Shell ein Kommando liest, das durch ein eigenes Programm realisiert ist, passiert Folgendes:

Der Shellprozess selbst veranlasst das Betriebssystem, eine nahezu identische Kopie des Shellprozesses zu erstellen. Der dazu verwendete Systemaufruf ist `fork()`. Unterscheidbar sind die Kopien nur am Returnwert von `fork()`. Der ursprüngliche Prozess wird im weiteren *Vater-Prozess*, der neue Prozess *Kind-Prozess* genannt. Beide Prozesse werden als *verwandte* Prozesse bezeichnet. Jeder der beiden Prozesse führt nach dem Systemaufruf `fork()` noch immer das gleiche Programm (d.h. das Shell Programm) aus.

Damit nun das Programm, das das auszuführende Kommando realisiert, gestartet werden kann, gibt es die Möglichkeit, einen Prozess mit einem anderen “image” zu überlagern, d.h. ein anderes Programm ersetzt das ursprüngliche. Dies wird durch den Systemaufruf `exec()` bewerkstelligt.

Nach dem Aufruf von `fork()` führt der Kindprozess der beiden Shellprozesse das dem auszuführenden Kommando entsprechende Programm aus und terminiert mit diesem.

Der Vater-Prozess macht nach dem Aufruf von `fork()` nichts anderes, als auf die Beendigung des Kind-Prozesses zu warten. Dieses Warten wird durch den Systemaufruf `wait()` realisiert. Nachdem der Kind-Prozess beendet ist, wartet der Vater-Prozess wieder auf die Eingabe eines neuen Kommandos. Soll ein Programm als Hintergrundprozess ausgeführt werden, dann wird dieser Prozess von der Shell wie oben beschrieben erzeugt, mit der Ausnahme, dass der Vater-Prozess nicht auf die Beendigung des Kind-Prozesses wartet.

Im Folgenden werden die Systemaufrufe `fork()`, `exec()` und `wait()` genauer beschrieben. Weiters wird ein Mechanismus zur Interprozesskommunikation zwischen “verwandten” Prozessen vorgestellt.

Erzeugung paralleler Prozesse

Mit Hilfe des Systemaufrufs `fork()` kann ein Prozess dynamisch erzeugt werden.

```
#include <unistd.h>

pid_t fork();
```

Der Aufruf von `fork()` generiert einen neuen Prozess, der bis auf seine Prozessnummer (PID) mit dem ursprünglichen Prozess (d.h. mit jenem Prozess, der den Aufruf durchgeführt hat) identisch ist. Diese beiden Prozesse sind miteinander "verwandt", der ursprüngliche Prozess ist der Vater-Prozess, der neue Prozess wird Kind-Prozess genannt. Nach dem Aufruf von `fork()` arbeiten Vater- und Kind-Prozess *parallel* und führen dabei *dasselbe Programm* aus. Auch der Kind-Prozess beginnt seine Programmausführung so, als ob er gerade den Aufruf von `fork()` ausgeführt hätte. Das Funktionsresultat von `fork()` ist im Kind-Proess `0`, und es ist im Vater-Prozess die PID des Kind-Prozesses. Im Falle eines Fehlers ist das Resultat `-1`. In diesem Fall konnte kein Kind-Prozess generiert werden.

Der Kind-Prozess "erbt" die gesamte Umgebung des Vater-Prozesses, einschließlich aller offenen Dateien und Signaldefinitionen. Aber er hat sein eigenes Daten- und sein eigenes Stack-Segment. Das bedeutet, dass der Kind-Prozess die zum Zeitpunkt des Aufrufes von `fork()` aktuellen Werte aller Variablen "übernimmt". Nach dem Aufruf von `fork()` manipulieren jedoch beide Prozesse ihre Variablen, ohne dass dies irgendwelche Auswirkungen auf den anderen Prozess hätte. Änderungen in den Signalbehandlungsroutinen nach `fork()` sind sowohl für den Vater- als auch für den Kindprozess lokal, d.h. sie beeinflussen den jeweils anderen Prozess nicht.

Ausgenommen davon sind Operationen mit denen globale Objekte wie Files, Semaphore, Shared Memory oder Message Queues bearbeitet werden. Mit solchen Operationen können Effekte erzielt werden, die von anderen (sowohl "verwandten" als auch "nicht verwandten") Prozessen wahrgenommen werden können. Zum Beispiel: Obwohl alle Filedeskriptoren prozess-lokale Variable sind, beziehen sich die Filedeskriptoren eines Vater- und eines Kind-Prozesses doch auf dieselben dahinter stehenden Files. Wenn also ein Kind-Prozess ein File durch eine Schreiboperation verändert, bemerkt (bzw. kann bemerken) auch der Vater-Prozess diese Veränderung. Dies kann zu Fehlern und unerwünschten Ergebnissen führen, wenn die beiden Prozesse z.B. gleichzeitig auf ein File schreiben wollen. Parallele Prozesse müssen daher synchronisiert werden, was ausschließlich in der Verantwortung des Programmmierers liegt.

Aus allem bisher Gesagtem ergibt sich folgendes typische Muster für die Verwendung von `fork()`:

```
pid_t pid;                                                  /* Prozess-Id */

switch (pid = fork())
{
    case -1:                    /* Fehlerfall: fork() nicht ausgefuehrt */
        BailOut("Can't fork child process!");
    break;

    case 0:          /* Hier folgt alles, was das Kind ausfuehren soll */
        ChildProcess();
    break;                            /* oder: exit(...) bzw. exec(...) */

    default:        /* Hier folgt alles, was der Vater ausfuehren soll */
        ParentProcess();
```

```
    break;
}                                       /* Ende des Switch-Statement */

Common();          /* Dieser Code wird von beiden Prozessen ausgefuehrt */
```

Siehe auch: `fork(2)`

Überlagerung eines Prozesses mit einem anderen Image

Dass Vater- und Kind-Prozess das gleiche Programm ausführen, ist natürlich nicht immer praktisch. Das würde bedeuten, dass die Source-Codes für alle Prozesse immer in einem Programm stehen müssten, und dass existierende Programme nicht wiederverwendet werden könnten. Der Systemaufruf `exec()` erlaubt es einem Prozess, ein anderes Programm auszuführen. Im Unterschied zu einem Prozeduraufruf, der sich vollständig in der Ebene der Programmiersprache abspielt, begibt sich `exec()` auf Systemebene und startet ein selbstständiges Programm. Im Unterschied zu einem Prozeduraufruf gibt es keine Rückkehr der Ablaufkontrolle an die Stelle des Aufrufs.

vorher	nachher
SIG_DFL	SIG_DFL
SIG_IGN	SIG_IGN
func	SIG_DFL

Tabelle 4.4: Signalzuordnungen vor und nach `exec`

Für Signaldefinitionen vor und nach dem Aufruf von `exec()` gelten Zuordnungen wie in Tabelle 4.4 aufgelistet. Alle Zuordnungen von Signalen zu eigenen Funktionen werden durch die Zuordnung zur Default-Aktion ersetzt; alle anderen Zuordnungen bleiben erhalten. Diese Vorgangsweise ist sinnvoll, da durch `exec()` ein *neues*, selbständiges Programm ausgeführt wird. Eine Signalbehandlungsroutine könnte nach einem `exec()` z.B. niemals ein offenes File schließen oder sonst auf im neuen Programm definierte Objekte einwirken.

Es gibt verschiedene Library Varianten von `exec()`:

```
#include <unistd.h>

int execv(const char *filename, char * const argv[]);
int execl(const char *filename, const char *arg0, const char *arg1,...,
            const char *argn, (const char *) 0);
int execvp(const char *filename, char * const argv[]);
int execlp(const char *filename, const char *arg0, const char *arg1,...,
            const char *argn, (const char *) 0);
```

Die Funktion `execv()` exekutiert das unter `filename` abgespeicherte Image; die Argumente des Aufrufs sind in `argv` enthalten (wie `argv` in `main`). Dabei ist das erste Argument (`argv[0]`) stets der Name des Kommandos. Das letzte Argument in `argv` muss `(const char *) 0` sein. Das mit dem Kommandonamen `filename` assoziierte Programm wird aufgerufen, die Argumente werden übergeben und das Programm wird dann ausgeführt.

Eine andere Variante von `exec()` ist `execl()`, bei dem die Argumente explizit und der Reihe nach angeführt werden. Man beachte, dass das letzte Argument wieder `(const char *) 0` sein muss. Das erste Argument `arg0` ist wieder der Name des Kommandos.

Die beiden Varianten `execvp()` und `execlp()` verhalten sich genauso wie `execv()` bzw. `execl()`, suchen aber zusätzlich — genauso wie die Shell — nach dem ausführbaren File `filename` in bestimmten Directorys. Diese Directorys werden der Environment-Variablen `$PATH` entnommen.

Zu beachten ist noch, dass nach einem Aufruf von `exec()` immer eine Fehlerbehandlung durchzuführen ist. Da das "Image" des aufrufenden Prozesses verloren geht, ist immer ein Fehler aufgetreten (d.h. `exec()` konnte nicht ausgeführt werden), wenn der Kontrollfluss eines Programmes an eine Stelle *nach* dem Aufruf von `exec()` gelangt (`assert(3)`).

Siehe auch: `execve(2)`, `execl(3)`

Warten auf die Beendigung der Kind-Prozesse

Der Systemaufruf

```
#include <sys/wait.h>

pid_t wait(int *status);
```

bewirkt, dass der aufrufende Prozess solange angehalten wird, bis eines seiner Kinder die Ausführung beendet, oder bis `wait()` durch ein eintreffendes Signal unterbrochen wird. `wait()` liefert als Funktionsresultat die PID des beendeten Kind-Prozesses, bzw. `-1` falls kein Kind-Prozess existiert, `wait()` durch ein Signal unterbrochen wurde oder ein sonstiger Fehler aufgetreten ist. Falls `status` auf eine gültige Adresse zeigt (d.h. ungleich `NULL` ist), dann liefert `wait()` den Exitcode (siehe `exit(3)`) des beendeten Kind-Prozesses als Ergebnisparameter.

In diesem Zusammenhang soll auf die Funktion `exit()` hingewiesen werden. Der Aufruf `exit(status)` beendet den Prozess und liefert den Integer-Wert `status` als Mitteilung (Exitcode) an die Umgebung, die den Prozess aufgerufen hat (Shell, Vater-Prozess). Dabei ist `status` stets `0` für erfolgreiches Beenden, während ein Wert größer `0` auf einen Fehler hinweist.

Die Nachteile von `wait()` sind, dass nur eine eher primitive Art der Synchronisation durchgeführt werden kann, und dass die Anwendung von `wait()` auf "verwandte" Prozesse beschränkt ist. Ein Vater-Prozess kann auf seine eigenen Kind-Prozesse warten.

Siehe auch: `wait(2)`

Das folgende Beispiel zeigt, wie die Library-Funktion `system(3)` mit Hilfe von `fork()`, `exec()` und `wait()` implementiert werden kann. Die Funktion `system()` erlaubt es, von einem C-Programm Shell-Kommandos aufzurufen und auszuführen. Das betreffende Shell-Kommando wird dabei der Funktion `system()` als Parameter übergeben.

```
int system (const char *command)
{
```

```
    pid_t pid;
    pid_t wpid;
    int status;

    if (command == NULL)
    {
        return 0;
    }

    switch (pid = fork())
    {
        case -1:
            return -1;
        break;

        case 0:
            (void) execl ("/bin/sh",  "sh", "-c", command, (char *) 0);
            return -1;             /* signal error to calling function */
        break;

        default:
            while ((wpid = wait (&status)) != pid)
            {
                if (wpid != -1)             /* other child terminated? */
                {
                    continue;                    /* restart wait call */
                }
                if (errno == EINTR)          /* interrupted by signal? */
                {
                    continue;                    /* restart wait call */
                }

                return -1;         /* signal error to calling function */
            }
        break;
     }

     return WEXITSTATUS(status);  /*return exit status of child process*/
}
```

Beim Aufruf von `wait()` in einer Schleife im Beispiel handelt es sich *nicht* um busy waiting. Die Funktion `wait()` stopt die Prozessausführung bis entweder ein Kindprozess terminiert oder ein Fehler auftritt. Wurde `wait()` durch ein Signal unterbrochen so wird `-1` als Returnwert zurückgeliefert und `errno` auf `EINTR` gesetzt. In diesem Fall ist es notwending, `wait()` nocheinmal auszuführen. Ein Returnwert ungleich `pid` und ungleich `-1` kann auftreten, wenn ein Kindprozess mit einer anderen Prozessnummer als `pid` terminiert (und `wait()` muss neu aufgerufen werden). Da das Lesen der Statusinformation des Kindprozesses konsumierend ist, geht in diesem Fall die Statusinformation

dieses Prozesses verloren (um dies zu vermeiden dann, kann `waitpid(2)` verwendet werden, das nur nach Terminierung eines bestimmten Kindprozesses oder im Fehlerfall einen Wert retourniert).

Der Grund also, warum in diesem Beispiel `wait()` in einer Schleife aufgerufen wird, ist, dass allfällige Unterbrechungen von `wait()` durch Signale ignoriert werden sollen und stattdessen auf die tatsächliche Beendigung des Kind-Prozesses gewartet werden soll.

Zombies, SIGCHLD und wait3

Die oben beschriebene Art und Weise die Termination eines Kindprozesses zu überwachen gehört zum Regelfall, soweit es die Übungsbeispiele betrifft. Wird jedoch das Prinzip *forkender Server* angewendet, dann ist wie folgt vorzugehen:

Ein "forkender Server" ist ein Prozess, der an ihn gestellte Anforderungen dadurch erledigt, dass er den Start eines Kindprozesses initiiert und diesem die Aufgabe überträgt. Dabei ist es nicht notwendig auf die Terminierung dieses Kindprozesses zu warten. Vielmehr ist der Server im Stande, die nächste Anfrage zu erledigen[5]. Nachdem unter UNIX jedoch jedes Kommando einen Return-Wert liefert, besetzen auch bereits terminierte Prozesse einen Eintrag in der Prozesstabelle, solange bis ihr Return-Wert (vom Vater) konsumiert wird. Wird dieser Wert nicht *abgeholt*, so ist ein so genannter Zombie entstanden. Durch die Akkumulierung der Prozesseinträge kann es dann vorkommen, dass in der (begrenzten) Prozesstabelle kein Platz mehr frei ist und kein neuer Prozess mehr gestartet werden kann.

Jedem Prozess wird bei der Terminierung eines seiner Kindprozesse das Signal `SIGCHLD` geschickt. Ein *forkender Server* behandelt also korrekterweise dieses Signal, das normalerweise ignoriert wird. In der zugehörigen Signalroutine werden durch entsprechende Aufrufe von `wait3(2)` derartige Return-Werte konsumiert und die Tabelleneinträge somit gelöscht. `wait3(2)` funktioniert dabei ähnlich wie `wait(2)`, verzögert den aufrufenden Prozess allerdings nicht, wenn kein Return-Wert eines bereits *verstorbenen* Kindes vorhanden ist.

Weiters ist zu beachten, dass einige System-Calls mit Fehlerstatus terminieren, wenn während ihrer Ausführung Signale eintreffen. Erst eine genauere Betrachtung der Variable `errno` gibt Aufschluss darüber, ob ein wirklicher Fehler vorliegt oder lediglich ein eintreffendes Signal (`errno == EINTR`) den System-Call abgebrochen hat. In diesem Fall müsste, wie im obigen Beispiel, der Aufruf von neuem erfolgen (wenn möglich).

waitpid und wait3

Verschiedene Systeme bieten unterschiedliche wait Funktionen an. `wait3()` ist rein BSD spezifisch während `waitpid()` dem POSIX Standard [POS88] entspricht.

```
#include <sys/wait.h>

pid_t waitpid(
```

[5]Wenn etwa in der Shell ein Hintergrundprozess gestartet wird, so wird lediglich die PID des im Hintergrund ausgeführten Kommandos ausgegeben. Die Shell selbst jedoch ist sofort zur Ausführung des nächsten Kommandos bereit.

```
    pid_t process_id,
    int *status_location,
    int options);

pid_t wait3(
    int *status_location,
    int options,
    struct rusage *resource_usage);
```

Zu beachten ist, dass `wait3()` und `waitpid()` von der jeweils anderen Funktion nicht abgedeckte Funktionalität bietet.

Interprozesskommunikation zwischen verwandten Prozessen (Pipes)

Mit Hilfe von Files, Message Queues, Shared Memory und Named Pipes können Daten zwischen Prozessen ausgetauscht werden. Diese Mechanismen zur Interprozesskommunikation, die den Datenaustausch sowohl zwischen "verwandten" als auch "nicht verwandten" Prozessen ermöglichen, stellen relativ allgemeine Konzepte dar, die auch in vielen anderen Betriebssystemen vorhanden sind. Zusätzlich dazu gibt es in UNIX die Möglichkeit, zwischen zwei "verwandten" Prozessen Daten über eine so genannte *Pipe* zu übertragen.

Eine Pipe ist eine Verbindung zwischen zwei Prozessen mit einem "Leseende" für ankommende Daten und einem "Schreibende" zum Senden von Daten. Daraus ergibt sich, dass mit einer Pipe Daten nur in eine Richtung übertragen werden können. Will man also sowohl Daten senden als auch empfangen, dann benötigt man zwei Pipes, eine pro Übertragungsrichtung. Weiters ist zu beachten, dass Pipes nur zwischen "verwandten" Prozessen eingerichtet werden können.

Eine Pipe wird mittels eines Feldes von zwei Integer-Elementen deklariert. Diese beiden Elemente sind Filedeskriptoren, und zwar das erste Element (mit dem Index 0) für das Leseende und das zweite Element (mit dem Index 1) für das Schreibende der Pipe. Die Deklaration einer Pipe sieht also z.B. folgendermaßen aus:

```
int npipe[2];
    /* npipe[0] ... file descriptor for read  */
    /* npipe[1] ... file descriptor for write */
```

Das Öffnen einer Pipe wird durch den Systemaufruf `pipe()` bewerkstelligt, der als einzigen Parameter das oben erwähnte Feld von Filedeskriptoren benötigt. Falls die Pipe erzeugt werden kann, werden die beiden Filedeskriptoren entsprechend gesetzt und `pipe()` liefert den Funktionswert `0`. Im Fehlerfall ist der Funktionswert von `pipe()` `-1`.

```
#include <unistd.h>
#include <limits.h>  /* definition of PIPE_MAX */

int pipe(int npipe[2]);
```

Nach dem Aufruf von `pipe()` stehen beide Enden der Pipe im selben Prozess zur Verfügung. Die einzige Möglichkeit, die Pipe zur Kommunikation mit einem anderen Prozess zu verwenden, besteht

darin, *nach* dem Aufruf von `pipe()` mit `fork()` einen Kindprozess zu erzeugen, der dann durch "Vererbung" auch beide Enden der Pipe zur Verfügung hat.

Nach Aufruf von `fork()` muss einer der Prozesse das Leseende und der andere Prozess das Schreibende der Pipe schließen. Der schreibende Prozess schließt das Leseende, der lesende Prozess das Schreibende. Erst jetzt ist das Verhalten der Pipe definiert. Werden die Enden nicht richtig geschlossen, kann es zu unerwarteten Effekten kommen.

Nach all diesen Vorbereitungen kann nun sowohl das Lese- als auch das Schreibende der Pipe wie ein File[6] benutzt werden. Für das Lesen und Schreiben einer Pipe können daher die normalen Funktionen der System-I/O (`read()` und `write()`) verwendet werden. Mit `fdopen(3)` kann ein Stream (`FILE *` in einem C-Programm) zu einem Filedeskriptor assoziiert werden, sodass es dann möglich ist, Funktionen wie `fopen()`, `fclose()`, die auf Streams operieren, zu verwenden.

```
#include <stdio.h>

FILE *fdopen(int filedes, const char *mode);
```

Bei Zugriff auf die Pipe findet eine primitive Art der Synchronisation statt: Wird durch eine Schreibaktion ein bestimmtes Maximum an Daten in der Pipe überschritten, so wird der Sender blockiert. Sind keine Daten in der Pipe vorhanden, wartet ein Lesebefehl so lange, bis wieder Daten vorhanden sind. Wird auch das letzte Schreibende der Pipe geschlossen, entdeckt der Leser das Ende des Files (`EOF`) (natürlich erst nachdem er alles vorher Geschriebene gelesen hat). Terminiert der Leser vorzeitig oder wird das Leseende der Pipe geschlossen, erhält der Schreiber beim nächsten Schreibversuch das Signal `SIGPIPE`.

Wenn der Kindprozess ein existierendes Programm ausführen soll, sind die von `pipe()` gelieferten Filedeskriptoren meist nicht direkt verwendbar, wenn das Programm Eingaben auf Filedeskriptor `0` erwartet und seine Ausgaben auf Filedeskriptor `1` schreibt. In diesem Fall muss vor dem `exec()` des Programms dafür gesorgt werden, dass Filedeskriptor `0` ein Duplikat des Leseendes, bzw. Filedeskriptor `1` ein Duplikat des Schreibendes der Pipe ist. Dazu dient der Systemaufruf `dup()` bzw. `dup2()`.

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

int dup(int oldd);
int dup2(int oldd, int newd);
```

`dup()` liefert einen Filedeskriptor, der ein Duplikat von `oldd` darstellt, d.h. die Verwendung des neuen Filedeskriptors hat denselben Effekt wie die Verwendung von `oldd`. Der neue Filedeskriptor wie `oldd` bewegen beide denselben Positionszeiger im File weiter. Der von `dup()` zurückgelieferte Filedeskriptor ist stets der *kleinste*, zum Zeitpunkt des Aufrufes von `dup()` nicht in Verwendung befindliche.

Um also `stdin` (Filedeskriptor `0`) eines Prozesses, auf eine Pipe, oder ganz allgemein ein File, mit Filedeskriptor `pd` umzulenken, ruft man

[6] mit Einschränkungen. So kann z.B. `seek()` nicht verwendet werden

```
   (void) close(0);

   if (dup(pd) == -1)
   {
       /* Fehlerbehandlung */
   }

   (void) close(pd);
```

auf. Mit `dup2()` kann man die Nummer des neuen Filedeskriptors explizit angeben (wenn dieser bereits in Verwendung steht, wird er zuerst geschlossen). Dasselbe Resultat wie oben mit `dup()` und `close()` kann mit `dup2()` so erzielt werden

```
   if (dup2(pd, 0) == -1)
   {
      /* Fehlerbehandlung */
   }

   (void) close(pd);
```

Im Fehlerfall liefern beide Funktionen `-1` zurück.

Das folgende Beispiel zeigt, wie eine Pipe vom Vater- zum Kindprozess erzeugt werden kann. Außerdem demonstriert es die Verwendung der Library-Funktion `fdopen()`, mittels welcher ein Filedeskriptor in einen File-Pointer umgewandelt werden kann, mit dem bereits angesprochenen Vorteil, fast alle Funktionen der Standard-I/O verwenden zu können.

Beispiel

```
/*
 *  Module:        Fork Wait Pipes
 *  File:          %M%
 *  Version:       %I%
 *  Creation Date: Mon Aug 30 19:29:19 MET DST 1999
 *  Last Changes:  %G%
 *  Author:        Wilfried Elmenreich
 *                 wilfried@vmars.tuwien.ac.at
 *  Contents:      Program Example with fork, pipe & wait
 *
 *  FileID: %M% %I%
 */

/*
 * ********************************************************* includes ***
 */

#include <stdio.h>
#include <stdlib.h>
```

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

#include "support.h"

/*
 * ********************************************************* defines ***
 */

#define MAXCHAR 80

/*
 * ********************************************************** globals ***
 */

const char *szCommand = "<not yet set>";              /* command name */
static int pPipe[2];           /* file descriptors for read and write */
static pid_t pid;                       /* process ID of child process */
static FILE *pStream = (FILE *) 0;          /* pointer for pipe stream */

/*
 * ******************************************************** prototypes ***
 */

void BailOut(const char *szMessage);

/*
 * ********************************************************* functions ***
 */

void FreeResources(void)
{
    if (pStream != (FILE *) 0)
    {
        if (fclose((FILE *) pStream) == EOF)
        {
            pStream = (FILE *) 0;
            BailOut("Cannot close pipe stream!");
        }
        pStream = (FILE *) 0;                   /* pipe no longer open */
    }
}


void BailOut(const char *szMessage)
{
```

```
    if (szMessage != (const char *) 0)
    {
        PrintError(szMessage);
    }

    FreeResources();

    exit(EXIT_FAILURE);
}


void AllocateResources(void)
{
    if (pipe(pPipe) == -1)
    {
        BailOut("Cannot create pipe!");
    }
}


void Usage(void)
{
    (void) fprintf(stderr,"USAGE: %s\n",szCommand);

    BailOut((const char *) 0);
}


void FatherProcess(void)
{
    if (close(pPipe[0]) == -1)
    {
        BailOut("Error closing the read descriptor!");
    }

    if ((pStream = fdopen(pPipe[1], "w")) == (FILE *) NULL)
    {
        BailOut("Cannot open pipe for writing!");
    }

    if (fprintf(pStream, "Message from father\n") < 0) /* write to pipe*/
    {
        BailOut("Can't write to pipe!");
    }

    FreeResources();
}
```

```
void ChildProcess(void)
{
    char buffer[MAXCHAR + 1];

    if (close(pPipe[1]) == -1)
    {
        BailOut("Error closing the write descriptor!");
    }

    if ((pStream = fdopen(pPipe[0], "r")) == (FILE *) NULL)
    {
        BailOut("Cannot open pipe for reading!");
    }

    while(fgets(buffer, MAXCHAR, pStream) != NULL)
    {
        if (printf("%s",buffer) < 0)
        {
            BailOut("Cannot print to stdout!");
        }
        if (fflush(stdout) == EOF)
        {
            BailOut("Cannot flush stdout!");
        }
    }

    if (ferror(pStream))                          /* really an error? */
    {
        BailOut("Cannot read from pipe!");
    }

    FreeResources();
}


int main(int argc, char **argv)
{
    pid_t wpid;
    int status;

    szCommand = argv[0];

    if (argc != 1)
    {
        Usage();
```

```
    }

    AllocateResources();

    switch (pid = fork())
    {
        case -1:                                                /* error */
            BailOut("Fork failed!!");
        break;

        case 0:                                                 /* child */
            ChildProcess();         /* child process reading from pipe */
        break;

        default:                                               /* father */
            FatherProcess();         /* father process writing to pipe */

            while((wpid = wait(&status)) != pid)
            {
                if (wpid != -1)
                {
                    continue;
                }
                if (errno == EINTR)
                {
                    continue;
                }

                BailOut("Error waiting for child process!");
            }
        break;
    }

    exit(EXIT_SUCCESS);
}

/*
 * ************************************************************* EOF ***
 */
```

popen und pclose

In vielen Programmen will man die Ausgabe eines speziellen UNIX-Programms zur Verfügung haben bzw. eigene Daten als Eingabe an ein solches schicken. Dafür stehen die Library-Funktionen `popen()` und `pclose()` zur Verfügung.

```
#include <stdio.h>
```

```
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

`popen()` hat eigentlich die Funktion der Systemaufrufe `pipe()`, gefolgt von `fork()` und `exec()`. `popen()` öffnet eine Pipe, erzeugt einen Kindprozess und führt in diesem Kindprozess das UNIX-Kommando `command` aus. Das Funktionsergebnis ist ein gültiger File-Pointer, über den nun Daten an den Kindprozess geschickt bzw. vom Kindprozess empfangen werden können. Die Übertragungsrichtung wird dabei durch `type` festgelegt: Ist `type` gleich `"r"`, dann kann über den von `popen()` gelieferten File-Pointer die Standard-Ausgabe von `command` gelesen werden; ist `type` gleich `"w"`, dann können über den von `popen()` gelieferten File-Pointer Eingabedaten an die Standard-Eingabe von `command` geschickt werden. Im Fehlerfall liefert `popen()` den Pointerwert `NULL`.

Ein mit `popen()` geöffneter File-Pointer muss mit `pclose()` geschlossen werden. `pclose()` wartet auf die Terminierung des Kindprozesses und liefert den Exitstatus von `command` als Funktionswert, bzw. `-1` falls `fp` nicht mit der Funktion `popen()` geöffnet wurde.

Das folgende Beispiel zeigt die Verwendung von `popen()`. Es wird das Kommando `ls` ausgeführt. Die Ausgaben von `ls` werden im Programm gelesen, um sie anschließend zu verarbeiten.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    FILE *fp;

    CheckArguments(argc, argv);                  /* check arguments */

    if ((fp = popen("ls -rt", "r")) != NULL))
    {
        ProcessOutput(fp);                  /* process output of ls */

        (void) pclose(fp);
    }

    exit(EXIT_SUCCESS);
}
```

Siehe auch: `pipe(2)`, `popen(3)`

Literaturverzeichnis

[C89] ANSI Accredited Standards Committee, X3 Information Processing Systems. *American National Standard for Information Systems — Programming Language C*, Dezember 1989.

[C99] *International Standard ISO/IEC 9899 Programming languages – C*, Dezember 1999. approved by ANSI Accredited Standards Committee.

[Har91] S. Harbison and G. Steele, Editors. *C, A Reference Manual*. Prentice Hall, 91.

[Ker78] B. W. Kernighan and D. M. Ritchie, Editors. *The C Programming Language*. Prentice-Hall, 78.

[Ker88] B. W. Kernighan and D. M. Ritchie, Editors. *The C Programming Language, ANSI C*. Prentice Hall, 88.

[Koe89] A. Koenig. *C Traps and Pitfalls*. Addison-Wesley, Jänner 1989.

[Pos81a] J. Postel. Internet Protocol. Technical Report RFC 791, SRI International, Menlo Park, CA, USA, September 1981.

[Pos81b] J. Postel. Transmission Control Protocol. Technical Report RFC 793, SRI International, Menlo Park, CA, USA, September 1981.

[POS88] *IEEE Standard 1003.1, POSIX, Portable Operating System Interface for Computer Environments*, 1988.

[Rus98] D. A. Rusling. *The Linux Kernel*. New Riders Pub, Wokingham, Berkshire, UK, März 1998.

[Spe88] Henry Spencer. How to Steal Code or Inventing the Wheel Only Once. In *Proc. Winter Usenix Conf. Dallas 1988*, pages 335–345, Jänner 1988.

Index

The Ten Commandments for C Programmers

Henry Spencer

I Thou shalt run *lint* frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.

II Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.

III Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.

IV If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.

V Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".

VI If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.

VII Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.

VIII Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.

IX Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.

X Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX", and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.