Vorlesungsscript FH Friedberg *(Hessen)*

Skript zu Betriebssysteme-Labor

Prof. Dr. Karim Roger Kremer

20. März 2005

Inhaltsverzeichnis

Kapitel 1

Einführung in UNIX

Im technisch-wissenschaftlichen Bereich haben heute sogenannte Workstations, die unter einer Version des Betriebssystems UNIX laufen, große Verbreitung gewonnen. Workstations sind Rechner, die in ihren äußeren Abmessungen mit einem PC für den Privatgebrauch vergleichbar sind, leistungsmäßig diesen Heim-PC's jedoch überlegen sind. Normalerweise gehören ein großer hochauflösender Monitor und der Anschluss an ein lokales und/oder Weitverkehrsnetz zur Grundausstattung von Workstations.

Aber auch wenn Sie einen leistungsstarken Rechner eines Rechenzentrums, einen sogenannten Mainframe, nutzen wollen, haben Sie es meist mit dem UNIX-Betriebssystem zu tun. Sowohl Workstations als auch Mainframes ist gemein, dass sie anders als PC's für mehrere Bentzer gleichzeitig zur Verfügung stehen. Man spricht von einem Multitasking-System, wenn mehrere verschiedene Aufgaben gleichzeitig bearbeitet werden; man spricht von einem Multiuser-System, wenn das System mit mehreren Benutzern umgehen kann. Ein Betriebssystem, das diese Anforderungen erfüllt, ist UNIX.

1.1 Historie und Versionen

Den Anfang in der UNIX-Geschichte machte ein Anwender. Ken Thompson benutzte am MIT einen Rechner des Typs DEC PDP 7. Da dieser Rechner mit einem Single User Betriebssystem ausgestattet war, war der Rechner häufig durch einen seiner Kollegen belegt, wenn Thompson daran arbeiten wollte. Von seiner Leistungsfähigkeit her war der Rechner jedoch in der Lage, mehrere Benutzer gleichzeitig zu bedienen.

Im Jahr 1969 entwickelte Thompson daher zunächst ein interaktives Mehrbenutzer-Betriebssystem mit dem Namen MULTICS (Multi User Computing System). Dieses Betriebssystem war ganz in Assembler-Sprache geschrieben.

Thompson erkannte, dass es besser wäre das Betriebssystem in einer höheren Programmiersprache zu entwickeln, um den Portierungsaufwand auf unterschiedliche Hardware zu verringern. Deshalb entwickelte er eine Sprache zur Entwicklung von Programmiersprachen mit dem Namen B. B enthielt keine Datentypen und war der Vorläufer der Programmiersprache C, die dann von Dennis Ritchie aus B entwickelt wurde. Im Jahr 1973 entstand mit 90% B-Kode und nur 10% Assembler-Kode das erste UNIX-Betriebssystem (Unified Conception

MULTICS).

Bis zum Anfang der 80'er Jahre wurde UNIX kostenlos auch in Quell-Kode an interessierte Hochschulen und Forschungseinrichtungen weitergegeben. Dies führte zu vielen Weiterentwicklungen und zufriedenen Benutzern. Der Name UNIX ist heute allerdings durch eine Lizens der Firma AT&T geschützt, d.h. man kann den Quellkode und vor allem den Namen UNIX von AT&T kaufen. Man unterscheidet deswegen:

UNIX-Look-Alikes: Betriebssysteme, die auf der Anwendungsschicht, wie UNIX funktionieren. Der Vorteil liegt in den ersparten Lizensgebühren. Außerdem ist der Funktionsumfang häufig erweitert z.B. Mehrprozessor-, Echtzeit-Unterstützung.

UNIX-Portierungen: Hier wurde der Quell-Kode von AT&T gekauft und an die Hardware angepasst.

Eine Hauptschiene neben AT&T UNIX bildete bald BSD UNIX (Berkley Software Distribution). BSD hat viel für die Benutzerfreundlichkeit von UNIX getan. Heute ist UNIX System V der Industriestandard für UNIX, der aus BSD und AT&T UNIX entstanden ist. Es gibt eine Definition für die Einhaltung dieses Standards die System V Interface Definition (SVID).

UNIX-Betriebssysteme gibt es für fast alle Hardware-Plattformen vom PC bis zum Großrechner (Mainframe). Es folgen einige namhafte Firmen, die UNIX-Betriebssysteme eingesetzt haben bzw. einsetzen:

Microsoft: PC-UNIX mit dem Namen XENIX

SUN: SunOS, Solaris für PC's und SUN-Workstations

IBM: AIX für PC's, RS/6000 Workstations und Mainframes

Seit Anfang der 90'er Jahre macht das von Linus Torvalds entwickelte Betriebssystem Linux Schlagzeilen: Linux ist ein UNIX-Betriebssystem, das es ermöglicht, auch PC's als Workstations zu benutzen. Die Idee von Linux, ein Betriebssystem und eine große Menge von Anwendungssoftware jedem frei zugänglich zu machen, wird von der Free Software Foundation unterstützt, die mit dem GNU-Projekt (Compiler für Ada, C, C++, FORTRAN, Java, Pascal, SmallTalk) schon seit längerer Zeit diese Ziele verfolgt.

Linux kann man kostenfrei aus dem Internet beziehen oder eine konfektionierte Version, wie z.B. die S.u.S.E.-Linux-Distribution, kaufen. Der Hauptvorteil der Distributionen ist die vereinfachte Installation und Administration des Betriebssystems und eine große Menge von Anwendungssoftware, die schon mitgeliefert wird.

Linux ist nicht kommerziell ausgerichtet sondern Freeware und in guter alter UNIX-Tradition auch als Quellcode erhältlich, so dass der Endanwender direkten Einfluss auf die Funktionalität von Linux nehmen kann. Freeware bedeutet hierbei, dass Linux keine Public Domain Software ist, sondern dass viele Linux-Komponenten das Copyright ihrer Entwickler tragen. Linux unterliegt den Bestimmungen der Free Software Foundation in der sogenannten GNU General Public License, die besagt, dass es erlaubt ist, die Software (kommerziell) zu nutzen und weiterzugeben, wenn sichergestellt ist, dass alle Modifikationen wieder frei verteilt werden dürfen.

Linux braucht in Bezug auf Leistungsfähigkeit und Stabilität den Vergleich mit kommerziellen UNIX-Varianten nicht zu scheuen. Es entspricht weitgehend dem System V Rel 4 - Standard und erlaubt die Nutzung X Window-basierter Anwendungen durch die Implementierung XFree86. Durch preemptives Multitasking und sehr gute Netzwerkmöglichkeiten ist Linux prädestiniert dafür, innerhalb eines Netzwerks Serverdienste zu übernehmen.

Ein Linux-PC kann z.B. Funktionen eines File- und Print-Servers übernehmen. Sie können aber auch über einen Linux-PC einen Internetzugang realisieren und so alle Internetdienste für das PC-Netzwerk verfügbar machen. Man kann natürlich auch die Dienste des Internet nur innerhalb des eigenen Netzwerks nutzen. Dies nennt man dann Intranet. Ein Web-Server gehört mittlerweile zum Lieferumfang der meisten Distributionen. Diesen Web-Server kann man z.B. zur Präsentation einer Firma im Internet benutzen. Auch Firewall-Funktionalitäten, die ein Netzwerk gegen unbefugte Zugriffe aus dem Internet schützen, lassen sich mit Linux realisieren.

Linux ist portabel, denn es werden mittlerweile viele Rechnerplattformen mit PowerPC-, Alpha-, MIPS- und SPARC-Prozessoren unterstützt. Die Frage, wie ein Linux-PC ausgestattet sein soll, lässt sich nicht pauschal beantworten. Folgende Eckdaten kann man jedoch für eine Minimalkonfiguration mit X Window nennen:

- Intel Pentium (oder kompatibler Prozessor) mit 133 MHz
- mindestens 32 MByte RAM
- VGA-Grafik mit 4 MByte RAM Grafikspeicher und 17 Zoll Monitor
- CD-ROM Laufwerk
- mehr als 500 MByte Festplattenplatz
- ISDN-Karte oder Modem

1.2 Erste Schritte in die UNIX-Praxis

Da bei UNIX-Systemen i.d.R. mehrere Anwender an einem Rechner oft auch gleichzeitig arbeiten, ist neben einem Benutzernamen ein Passwortschutz vorgesehen. Der Systemverwalter richtet Ihnen initial einen Benutzernamen und ein Passwort, dass meist schriftlich mitgeteilt wird, ein.

Nach dem Anmelden beim System, dem sogenannten login, wird i.d.R. die grafische Oberfläche X Window gestartet. Meist wird automatisch ein Fenster gestartet, in dem eine sogenannte Shell läuft. Die Shell ist die wichtigste interaktive Schnittstelle zur Ausführung von Kommandos für Benutzer und auch Programme. Die Shell interpretiert die Kommandos, d.h. sie übersetzt die Shell-Sprache in Aufrufe des Betriebssystemkerns und führt diese durch. In UNIX gibt es eine Vielzahl unterschiedlicher Shells. Auf einem System können mehrere Shells installiert sein und gleichzeitig benutzt werden. Wir werden uns später noch genauer mit der Programmierung der Korn Shell aus dem System V-Standard beschäftigen.

Nach dem login haben Sie die Möglichkeit das Passwort mit dem Befehl passwd zu ändern. Eine Sitzung oder session endet nach dem Befehl logout, mit dem man sich beim System wieder abmeldet.

Im Gegensatz zum Heim-PC schaltet man eine UNIX-Workstation nicht einfach aus. Das Ausschalten einer UNIX-Workstation geschieht normalerweise nur zu Wartungs- oder Installationszwecken nach der ordnungsgemäßen Beendigung aller Prozesse durch den Systemverwalter. Wird eine UNIX-Workstation im Betrieb einfach ausgeschaltet, so kann es zu ernsten Schäden im Dateisystem kommen ! Außerdem können ja noch andere Benutzer auf dem System angemeldet sein. Eine Liste aller angemeldeten Benutzer liefert der Befehl who.

Bei Linux-PC's im Heimgebrauch kann es sein, dass Sie die Rolle des Anwenders und des Systemverwalters in einer Person ausfüllen.

Das Passwort ist vergleichbar mit der Geheimzahl einer Scheckkarte. Weiß jemand Ihr Passwort, so kann er sich über Computernetze weltweit als Sie ausgeben. Dazu sollte man ein sicheres Passwort nehmen, dies nie verraten und es regelmäßig ändern. Regeln für ein sicheres Passwort sind u.a.:

- Keine Namen verwenden.
- nicht die Benutzerkennung als Passwort auch nicht in vertauschter Zeichenreihenfolge verwenden.
- Das Passwort sollte in keinem Lexikon oder Wörterbuch zu finden sein.
- Verwenden Sie Groß- und Kleinschreibung möglichst auch durcheinander.
- Verwenden Sie mindestens zwei Sonderzeichen wie $,%,/ und zwei Ziffern.
- Das Eintippen soll so rasch vor sich gehen, dass sich niemand das Passwort merken kann (schwierig).
- Das Passwort nie aufschreiben.

Ein sicheres Passwort ist z.B.: Ah1&7Fv!

So ein Passwort kann man aus einem Merksatz ableiten, den man gut behalten kann z.B.: Anna hat den ersten und siebten Freund verlassen!

Es ist heute üblich in UNIX mit graphischen Benutzeroberflächen (Graphical User Interface, kurz GUI) zu arbeiten. Bei UNIX wird heute i.d.R. das X Window-System, kurz X, benutzt. Da X in der 11. Version momentan vorliegt, findet man als Abkürzung in der Literatur oft X11.

Unter X gibt es noch kein einheitliches "look and feel", z.B. was die Bedienung mit der Maus angeht. Die Bedienung hängt von der Hardware und vom eingesetzten Window-Manager ab. (Dazu später mehr.)

Gewisse Grundelemente der Bedienung sind jedoch überall gleich. So sollte die Maus 3 Tasten haben. Die grundlegendsten Aktionen, wie Fenster verschieben, Menüpunkte auswählen usw. vollführt man mit der linken Maustaste.

Den Copy & Paste-Mechanismus löst man wie folgt aus: Man markiert einen Text, indem man den Mauszeiger mit gedrückter linker Maustaste über den Text bewegt. Größere Abschnitte kann man mit einem Mausklick der linken Taste am Anfang und der rechten Taste am Ende markieren. Die mittlere Maustaste wird benutzt um den Text an einer beliebigen anderen Stelle auch in einem anderen Fenster einzufügen.

Erwähnenswert sind noch die Hintergrund-Menüs, die mit einem Mausklick auf einer beliebigen Stelle des Bildschirmhintergrunds erzeugt werden. Da es drei Maustasten gibt, gibt es auch drei Hintergrund-Menüs. Ihr Aussehen kann völlig

frei konfiguriert werden. Meist hat der Systemverwalter schon etwas Sinnvolles eingetragen.

Es gibt ein aktives Fenster, in dem alle Eingaben landen, das farblich anders als die nicht aktiven Fenster dargestellt wird. Der Wechsel des aktiven Fensters geschieht je nach eingesetztem Window-Manager durch bloßes Verschieben des Mauszeigers in ein anderes Fenster oder durch zusätzlichen Mausklick zur Bestätigung.

In X trifft man meist auf mindestens ein Fenster, in dem genau wie im nicht graphischen UNIX ein Prompt auf Kommandoeingaben wartet. Solche Fenster nennt man auch virtuelle Terminals. Von einem solchen Fenster aus kann man nun Programme starten. Es ist in UNIX eher selten, dass man Programme über Menüs aktiviert.

Von einem virtuellen Terminal-Fenster lassen sich die Multitasking-Fähigkeiten von UNIX bequem nutzen. Dazu startet man für X geschriebene Programme, wie xterm, das ein virtuelles Terminal-Fenster erzeugt, im Hintergrund von einem virtuellen Terminal aus. Das bedeutet, dass nicht auf das Ende des Programms gewartet wird, bis der nächste Befehl eingegeben werden kann, sondern dass es möglich ist, Kommandos einzugeben während das Programm noch läuft. Alles was man dafür tun muss ist, ein Leerzeichen gefolgt von einem & an den Befehl anzuhängen. Ein X-Terminal wird z.B. durch den Befehl xterm & gestartet.

Ein exit oder logout auf einem zusätzlich gestarteten X-Terminal-Fenster bewirkt nicht die Abmeldung vom Rechner, sondern nur das Schließen des Fensters. Oft ist das erste virtuelle Terminal, das sogenannte login-Terminal, mit dem man sich beim System mit exit oder logout abmeldet. Auf manchen Systemen befindet sich ein Eintrag in einem Hintergrundmenü zur Abmeldung beim System.

Selbst die elementarsten UNIX-Befehle bieten eine Fülle von Optionen, die auch Profis nur selten auswendig können. Daher gibt es eine Online-Hilfe, die durch das Kommando man für manual (Kurzwort man-page) aufgerufen wird. Man gibt den Befehlsnamen hinter dem man-Befehl ein, z.B. man who. Mit den Tasten f und b kann man in der Hilfe blättern mit q verläßt man sie.

Unter X steht eine verbesserte Version der man-pages zur Verfügung. Diese kann mit dem Kommando xman & aufgerufen werden. Eine Einführung in die Bedienung von xman erhalten Sie durch einen Mausklick auf den Button Help. Zum Blättern im Text finden Sie am linken Rand des Fensters einen grauen Scrollbalken, den Sie mit der mittleren Maustaste verschieben können. Die Gliederung der Manual-Einträge ist stets ähnlich, so dass man sich trotz ihrer großen Menge, schnell zurechtfindet:

NAME: Name des Befehls und Funktion

SYNOPSIS: Angabe der Syntax aller Optionen und Argumente

DESCRIPTION: ausführliche Beschreibung der Funktionalität

OPTIONS: Beschreibung der einzelnen Optionen

SEE ALSO: Verweise auf verwandte Befehle

BUGS: Hinweise auf bekannte Fehler

Kapitel 2

UNIX-Korn-Shell-Programmierung

2.1 Einleitung

Auf der einen Seite dienen Shells in UNIX der interaktiven Steuerung des Rechners durch die Benutzer; auf der anderen Seite laufen Shell-Programme bei diversen Arbeiten des Betriebssystems für die Benutzer transparent ab. Z.B. werden beim Boot eines UNIX-Rechners verschiedene Hintergrundprozesse und Hardware-Treiber nach dem Laden des Betriebssystemkerns über Shell-Programme initialisiert. Genauso werden beim Shutdown Hintergrundprozesse und Hardware-Treiber über Skripte geordnet terminiert. Mit Shell-Programmen ist also auch die Steuerung komplexer Abläufe möglich.

Shell-Programme werden auch Skripte genannt. Skripte werden nicht compiliert sondern zur Laufzeit durch den Shell-Interpreter übersetzt - also interpretiert. Für die meisten UNIX-Systeme sind verschiedene Shells (Korn Shell, Bourne Shell, Borne Again Shell, TC-Shell usw.) verfügbar. Manchmal hängt es von den Vorlieben des Systemadministrators oder der Benutzer ab, welche Shells auf dem Rechner installiert sind. Allerdings ist im System V-Standard die Korn-Shell Pflicht, d.h. jedes System V-UNIX enthält die Korn Shell. Die Programmierung der meisten anderen Shells, wie z.B. Borne Again Shell und TC-Shell, unterscheiden sich kaum von der Programmierung der Korn-Shell. Aus diesem Grund beschränken wir uns auf die Beschreibung der Korn-Shell.

2.2 Aufbau und Aufruf eines Skripts

Ein Skript wird nur in Textform, d.h. als Quellkode, gespeichert. Befehle, die interaktiv eingegeben werden, können auch in einem Skript stehen. Nach dem Aufruf des Skripts interpretiert die Shell im einfachsten Fall sequentiell eine Zeile nach der anderen. Wie in anderen Programmiersprachen gibt es auch in Skripten Schleifen und Verzweigungen, die aber (wie wir noch sehen werden) eine andere Syntax und Semantik als dort haben.

Das folgende einfache Skript gibt eine sortierte Liste der angemeldeten Benutzer mit der Anzahl aus. Es soll als Textdatei unter dem Namen aktive_benutzer

gespeichert sein:

```
#!/bin/ksh
who | sort
echo "Anzahl angemeldeter Benutzer"
who | wc -l
```

An diesem Skript fällt zunächst die erste Zeile auf. Sie beginnt mit dem Zeichen #. Dies ist der Beginn eines Kommentars, d.h. was dahinter steht ist kein Befehl an die Shell. In diesem speziellen Fall folgt dem # noch ein !, was eine spezielle Bedeutung hat. Es ist nämlich die Anweisung an das Betriebssystem, das Skript mit der Shell, die in /bin/ksh gespeichert ist auszuführen. Auf diese Weise wird die Shell, die das Skript ausführen soll, also im Skript selber angegeben.

Ein solches Skript wird nun mit einem Texteditor erstellt. Es handelt sich bei einer Textdatei aber nicht um eine ausführbare Datei, weshalb das Ausführbarkeitsrecht noch gesetzt werden muss, bevor das Skript mit seinem Namen aufgerufen werden kann, z.B.:

```
chmod u+x aktive_benutzer
```

Der Aufruf des Skripts lautet nun:

```
aktive_benutzer
```

Die geschilderte Möglichkeit ist nicht die einzige ein Skript aufzurufen aber die gebräuchlichste. Eine weitere Möglichkeit des Aufrufs ist, eine nicht ausführbare Textdatei als Parameter an die Shell zu übergeben, z.B.:

```
ksh aktive_benutzer
```

Die Fehlersuche ist bei interpretierten Sprachen wie der Shell-Sprache wesentlich einfacher als bei compilierten Sprachen, da die Original-Quelldatei direkt ausgeführt wird und Fehler unmittelbar mit der betreffenden Zeile angezeigt werden. Wenn man ein detailliertes Ablaufprotokoll braucht kann man die Korn-Shell z.B. wie folgt aufrufen:

```
ksh -x aktive_benutzer
```

2.3 Übergabe von Argumenten

Skripte werden häufig benutzt, um kompliziertere Abläufe zu verkapseln. Z.B. gibt es eine Fülle von Optionen zu vielen UNIX-Befehle, die man sich schwer merken kann. Wir wollen ein Programm schreiben, das alle Dateien eines bestimmten Benutzers findet und ihren kompletten Pfadnamen ausgibt. Wir wissen, dass wir diese Aufgabe mit dem find-Befehl lösen können und finden z.B. nach dem Lesen der Manual-Page für find folgende Kommandozeile heraus:

```
find / -user kremer -print 2>/dev/null
```

Die Umleitung der Standardfehlerausgabe in /dev/null bewirkt hier nur, dass Fehlermeldungen unterdrückt werden, die z.B. durch unerlaubte Zugriffe des find-Kommandos entstehen können.

Da wir uns diese Zeile nicht merken wollen, schreiben wir sie in ein Skript mit dem Namen suche. Wir haben nun nur ein Problem. Was machen wir, wenn wir alle Dateien von root finden wollen? Entweder wir edieren das Skript und ändern kremer in root um, oder wir nutzen eine Möglichkeit, Argumente an das Skript zu übergeben.

Wir wählen natürlich den zweiten, flexibleren Weg. Beim Aufruf des Skriptes können wir nämlich Positionsparameter übergeben, die im Skript durch $0, $1, $2 usw. bezeichnet werden. $0 bezeichnet den Namen des Skripts, $1 den 1. Parameter, $2 den 2. Parameter usw. Unser flexibles Skript suche sieht z.B. wie folgt aus:

```
#!/bin/ksh
echo "Dateiliste von $1"
find / -user $1 -print 2>/dev/null
```

Es folgt der Aufruf, um alle Dateien von root zu finden:

```
suche root
```

Wenn wir das Skript in dieser Weise aufrufen, entstehen weitere Probleme: Das Shell-Fenster, in dem wir den Befehl aufrufen, ist nämlich für weitere Eingaben gesperrt. Außerdem passen die Ausgaben u.U. nicht in den Ausgabepuffer des Fensters, wodurch wir Dateien übersehen können. Wir lösen das erste Problem, indem wir das Skript als Hintergrundprozess starten:

```
suche root &
```

Das zweite Problem lösen wir, indem wir die Standardausgabe in eine Datei wie folgt umleiten:

```
#!/bin/ksh
echo "Dateiliste von $1" > liste_$1
find / -user $1 -print >> liste_$1 2>/dev/null
```

An diesem Skript sehen wir weiter, dass der Parameter $1 an verschiedenen Stellen eingesetzt werden kann, z.B. in der echo-Anweisung, die einen Text auf die Standardausgabe schreibt oder als Bestandteil des Dateinamens liste_$1.

Weiterhin sind in der Shell alle Aufrufparameter als eine Zeichenkette durch $* und alle Aufrufparameter als getrennte Zeichenketten durch $@ ansprechbar. $@ wird vor allem im Zusammenhang mit Schleifen verwendet. Darauf kommen wir noch zu sprechen. Mit $# kann die Anzahl der übergebenen Parameter überprüft werden.

Neben der Möglichkeit der Aufrufparameter, werden bei interaktiven Skripten Eingabeparameter benutzt, die während des Programmlaufs mit dem Befehl read eingelesen werden. Wir schreiben ein Skript, das eine Datei als Telefonverzeichnis füllt:

```
#!/bin/ksh
echo "Vor-,Nachnamen,Telefonnummer eingeben"
read vorname nachname telnr REST
echo $vorname $nachname $telnr >> $HOME/telnr.dat
```

Variablen brauchen also nicht (wie in compilierten Sprachen) vor ihrer Verwendung deklariert zu werden, sondern werden durch den read-Aufruf implizit vereinbart. Bei der ersten Verwendung oder bei einer Wertzuweisung (z.B. nachname="Meier") müssen Variablen ohne $-Zeichen verwendet werden. Soll der Inhalt einer Variablen verwendet werden, so muss das $-Zeichen dem Variablennamen vorangestellt werden (z.B. echo $nachname).

Alle an ein Shell-Skript übergebenen Parameter (Aufrufparameter, Eingabeparameter und Umgebungsparameter) sind vom Datentyp her Zeichenketten. Es gibt zwar Möglichkeiten, Zeichenketten in Zahlen umzuwandeln, mit denen dann auch gerechnet werden kann, aber dies spielt bei der Anwendung von Shells eine eher untergeordnete Rolle. Der Datentyp Zeichenkette dominiert also bei der Anwendung von Shells.

Im read-Aufruf des Beispiels fällt noch der Paramter REST auf. Er hat folgende Bedeutung: Nehmen wir an, nach der Aufforderung zur Eingabe, werden mehr als die drei Parameter eingegeben also z.B.:

```
Heinrich Meier 06031/123456 Wilhelm-Leuschner-Strasse 13
```

Leerzeichen sind also Trenner der verschiedenen Parameter. Soll ein Leerzeichen in einer Zeichenkette auftauchen, so muss diese mit " und "umschlossen werden, also z.B. "Jan Wilhelm". Nun enthält der Parameter REST im oberen Beispiel die letzten beiden Zeichenketten und die telnr enthält weiter das Gewünschte:

```
echo $REST
Wilhelm-Leuschner-Strasse 13
echo $telnr
06031/123456
```

Der letzte Parameter enthält also alle Zeichenketten, bis zum Schluss der Eingabe. Eine Eingabeprüfung auf gültige Parameter ist hiermit allerdings noch längst nicht realisiert. Für ein einführendes Beispiel ist dies auch zu aufwendig.

Sollte die Eingabe weniger Werte enthalten als von read gefordert, so bleiben die letzten Parameter des read-Aufrufs leer (leere Zeichenkette "").

Wenn Parameter für mehrere Skripten z.B. während einer gesamten Sitzung gelten sollen, ist es lästig, sie bei jedem Aufruf explizit übergeben zu müssen. Hierfür gibt es die implizite Übergabe von Parametern über die Umgebung (Environment). Das Environment ist nichts anderes, als eine Menge von Zeichenketten-Variablen, die aber nicht nur bei Shells sondern bei allen Prozessen verwendet werden. Das Environment ist also Teil der Prozessdefinition in UNIX. Hierüber werden vielerlei Informationen an verschiedenste Programme übergeben, z.B. der Suchpfad für Programme PATH, das aktuelle Verzeichnis PWD, die Login-Shell SHELL, der Benutzername LOGNAME, der Rechner, auf dem der X Window Server läuft, für X Window Clients im Netz DISPLAY usw. Die aktuelle Setzung aller Umgebungsvariablen liefert z.B. der Befehl env.

Grundsätzlich werden Umgebungsvariablen von einem Elternprozess an einen Kindprozess vererbt. Genauer gesagt erhält der Kindprozess eine Kopie der Umgebung des Elternprozesses in einem gesonderten Speicherbereich. Das bedeutet, dass Änderungen in der Umgebung des Kindprozesses, die Umgebung des Elternprozesses unberührt lassen. Natürlich haben auch Änderungen in der Umgebung des Elternprozesses nach dem Aufruf des Kindprozesses keine Auswirkungen auf dessen Umgebung.

In der Korn-Shell ist eine Variable zuächst lokal, d.h. sie wird nicht automatisch an einen aufgerufenen Prozess weitergegeben. Erst durch den Befehl export wird sie der Umgebung hinzugefügt. Nehmen wir an, dass wir Umgebungsvariablen verwenden möchten, um zwei Skripte zum Backup und Restore von Dateien ablaufen zu lassen:

```
Verzeichnis=$HOME/ksh-skripte
Medium=/dev/fd0
export Verzeichnis Medium
```

Die beiden Skripte können dann die Parameter Verzeichnis und Medium verwenden, ohne dass diese explizit übergeben werden. Das Skript backup sichert Dateien mit dem tar-Befehl (siehe Manual-Page):

```
#!/bin/ksh
# Skript backup
tar cvf $Medium $Verzeichnis > backup.prot
```

Das Skript restore kopiert die Dateien auf dem Archiv Medium in das Verzeichnis zurück:

```
#!/bin/ksh
# Skript restore
tar xvf $Medium $Verzeichnis > restore.prot
```

Empfehlenswert ist die Übergabe von Argumenten aus dem Environment also dann, wenn sie sich selten ändern. Wir fassen noch einmal zusammen:

1. Es gibt verschiedene Möglichkeiten, Argumente an ein Skript zu übergeben:
   (a) Aufrufparameter
   (b) Eingabeparameter
   (c) Umgebungsparameter
2. Argumente sind immer vom Datentyp Zeichenkette.

2.4 Substitution

Wie wir gesehen haben, können Parameter z.B. in einer echo-Anweisung verwendet werden. Was machen wir aber, wenn wir einen Text ausgeben wollen, der genauso heißt wie die Variable also z.B. $LOGNAME? Wir müssen der Shell mitteilen, dass es sich bei $LOGNAME nur um Text handelt, den sie nicht durch einen Wert ersetzen soll. Für diesen Zweck werden die einfachen Anführungszeichen ' verwendet:

```
echo '$LOGNAME hat den Wert' $LOGNAME
```

Als Ausgabe erscheint z.B.:

```
$LOGNAME hat den Wert kremer
```

Die einfachen Anführungszeichen schützen also das $-Zeichen vor der Interpretation durch die Shell. Neben dem $-Zeichen werden auch folgende Maskenzeichen nicht interpretiert, d.h. nur als Text aufgefasst:

```
$ & < > >> * ? [ ] { } \ ; : ‘
```

Die Bedeutung dieser Zeichen ist teilweise bekannt bzw. wird noch geklärt.

Doppelte Hochkommate “” schützen Maskenzeichen ebenfalls vor Interpretation durch die Shell, jedoch nicht folgende Zeichen:

```
$ \ ‘
```

Der Unterschied zwischen einem geraden Hochkomma und einem Hochkomma von links oben nach rechts unten ist allerdings in der Shell von großer Bedeutung:

```
gerades Hochkomma: ’
Hochkomma von links oben nach rechts unten: ‘
```

Das Hochkomma von links oben nach rechts unten wird nämlich zur Kommandosubstitution verwendet, d.h. die eingeschlossene Zeichenkette wird als Befehl ausgeführt und z.B. im Falle der echo-Anweisung der Ausgabetext des Befehls eingefügt:

```
echo “Aktuelles Verzeichnis von $LOGNAME: ‘pwd‘”
```

Dieser Befehl liefert als Ausgabe:

```
Aktuelles Verzeichnis von kremer: /home/kremer
```

Um den Unterschied zwischen einfachen und doppelten Hochkomma klar zu machen, wird die echo-Anweisung etwas verändert:

```
echo ’Aktuelles Verzeichnis von $LOGNAME: ‘pwd‘’
```

Als Ausgabe erhält man:

```
Aktuelles Verzeichnis von $LOGNAME: ‘pwd‘
```

2.5 Punktoperator

Wenn ein Skript von einer Shell aus aufgerufen wird, erzeugt UNIX in der bekannten Weise einen neuen Prozess für eine sogenannte Subshell. Werden nun lokale Variablen oder Umgebungsvariablen in diesem Skript in der Subshell verändert, so hat dies (wie erläutert) keine Auswirkungen auf die Variablen in der übergeordneten Shell. Für den Fall, dass die Variablen in der aktuellen Shell geändert werden sollen, wird der Punktoperator (ein dem Skriptnamen vorangestellter Punkt) verwendet. Der Punktoperator bewirkt also, dass keine Subshell gestartet wird, sondern dass das Skript in der aktuellen Shell ausgeführt wird. Man kann den Mechanismus leicht anhand eines Skripts hinauf, das das aktuelle Verzeichnis ändert, nachvollziehen:

```
#!/bin/ksh
cd ..
```

Ruft man das Skript ohne Punktoperator auf, so befindet man sich nach dessen Ende unverändert im gleichen Verzeichnis. Ruft man es jedoch mit Punktoperator auf, so hat sich das aktuelle Verzeichnis (die Umgebungsvariable PWD) geändert:

```
. hinauf
```

2.6 Fallunterscheidungen

Um verschiedene Fälle durch unterschiedliche Kontrollflüsse zu bearbeiten, verfügt die Shell über Ablaufstrukturen. Sie unterscheiden sich aber stark von den Fallunterscheidungen in Programmiersprachen wie C(++), Java usw.

Die if-Anweisung hat folgende Syntax:

```
if <Kommando>
then
<Kommandos>
else
<Kommandos>
fi
```

Der else-Zweig kann ggfs. auch fehlen:

```
if <Kommando>
then
<Kommandos>
fi
```

Hinter dem if steht also keine Bedingung (wie vielleicht gewohnt), sondern ein Kommando. Was hat es damit auf sich? Hier wird ein (genial) einfaches Konzept verwendet. Das einzige, was ein Kindprozess seinem Elternprozess immer übergeben muss, ist nämlich der exit-Kode. Dieser exit-Kode ist ein ganzzahliger Wert, der den Erfolg oder Misserfolg und ggfs. einen Fehler des Kommandos anzeigt. Wenn man mit dem if-Kommando nun einen Vergleich durchführen will, ruft man ein anderes Programm in einem Kindprozess auf, das den Vergleich durchführt und einen entsprechenden exit-Kode zurückgibt. Ein solches Vergleichsprogramm ist test, das auf allen System V-UNIX Systemen verfügbar ist, aber eben nicht Teil der Korn-Shell (kein Shell built in) ist. (Sie sollten aus diesem Grund nie ein Programm mit dem Namen test schreiben, denn das kann zu "merkwürdigen" Effekten bei der Ausführung von Skripten führen.)

test ermöglicht:

- den Vergleich von Zeichenketten
- den Vergleich von ganzahligen Werten
- die Überprüfung von Dateizuständen

test liefert als exit-Kode:

- 0, wenn der Test erfolgreich war

- ungleich 0, wenn der Test nicht erfolgreich war.

Wir können z.B. wie folgt überprüfen, ob die Anzahl übergebener Parameter beim Aufruf unseres obigen Skripts suche stimmt:

```
#!/bin/ksh
if test $# -ne 1
then
   echo "Aufruf $0 <Benutzername>"
   exit 1
fi
echo "Dateiliste von $1" > liste_$1
find / -user $1 -print >> liste_$1 2>/dev/null
```

Dieses Skript enthält im Falle eines Fehlers beim Aufruf ein exit-Kommando. Mit dem exit-Kommando wird das Skript verlassen und der angegebene exit-Kode (hier 1) an den aufrufenden Prozess übergeben.

Es ist bei der if-Ablaufstruktur wichtig, dass die Teile if then else und fi in eigenen Zeilen untergebracht werden. Will man für then z.B. keine neue Zeile beginnen, so muss man das Trennsymbol ; einfügen:

```
if test $# -ne 1; then
   echo "Aufruf $0 <Benutzername>"
   exit 1
fi
```

Für Vergleiche beinhaltet test die folgenden in der Tabelle angegebenen Operatoren:

Ausdruck	exit-Kode 0
Zeichenkette1 = Zeichenkette2	Gleichheit
Zeichenkette1 != Zeichenkette2	Ungleichheit
Zeichenkette	nicht leer
-n Zeichenkette	nicht leer
-z Zeichenkette	leer
Zahl1 -eq Zahl2	=
Zahl1 -ne Zahl2	<>
Zahl1 -gt Zahl2	>
Zahl1 -lt Zahl2	<
Zahl1 -ge Zahl2	>=
Zahl1 -le Zahl2	<=

Zur Überprüfung von Dateizuständen stehen folgende Optionen zur Verfügung:

Option	exit-Kode 0
-a	existiert
-r	lesbar
-w	schreibbar
-x	ausführbar
-f	regulär
-d	Verzeichnis
-h	Link
-b	Block Device
-c	Character Device
-p	Named Pipe
-s	nicht leer

Als Beispiel wollen wir ein Skript ansehen, das das Heimatverzeichnis auf einen Sekundärspeicher sichert, der über die Umgebungsvariable MEDIUM übergeben wird.

```
#!/bin/ksh
if test ! -b $MEDIUM
then
    echo "$MEDIUM ist kein Block Device"
    echo "Die Sicherung wird abgebrochen"
    exit 1
fi
echo "Sicherung von $HOME"
echo "Bitte Medium einlegen"
echo "Taste j startet, sonst Abbruch"
read antwort
if test "$antwort" = "j"
then
    tar cvf $MEDIUM $HOME
fi
```

Zunächst wird überprüft ob die Umgebungsvariable ein Block Device kennzeichnet (z.B. Diskette /dev/fd0). Das ! bedeutet hierbei die Negation also kein Block Device. Anschließend wird dem Benutzer die Möglichkeit gegeben ein Medium in das Laufwerk einzulegen, da der read-Befehl das Skript anhält, bis eine Eingabe erfolgt ist. Die Variable antwort wird nun getestet.

Die Hochkommata um $antwort herum sind wichtig: Drückt der Benutzer nämlich nur die Eingabe-Taste, so bleibt $antwort leer. Ohne die Hochkommata würde test in folgender Weise aufgerufen.

```
test = "j"
```

Dies führt zu folgender Fehlermeldung:

```
test: =: unary operator expected
```

Mit den Hochkommata gelingt das test-Kommando, da es einen Vergleich mit einer leeren Zeichenkette macht:

```
test "" = "j"
```

Um mehrere test-Bedingungen miteinander zu verknüpfen gibt es das logische Und -a und das logische Oder -o. Durch folgende Anweisung liefert beispielsweise exit-Kode 0, wenn Zahl zwischen 2 und 9 liegt:

```
test "$Zahl" -gt 1 -a "$Zahl" -lt 10
```

Eine Alternative Schreibweise für den test-Befehl, sind die eckigen Klammern [ ]. Äquivalent zu der obigen test-Anweisung ist also:

```
[ "$Zahl" -gt 1 -a "$Zahl" -lt 10 ]
```

Das Leerzeichen hinter [und vor] ist wichtig, da ansonsten ein Fehler auftritt.

Die if-Anweisung kann außer dem test-Kommando jedes beliebige andere Kommando aufrufen. Es wird nur der exit-Kode dieses Kommandos ausgewertet. Folgendes Skript wertet aus, ob ein Benutzername auf dem System verzeichnet ist oder nicht. Dazu wird die Datei passwd mit dem grep-Kommando untersucht. grep liefert bei einem Treffer den exit-Kode 0 sonst 1 zurück:

```
if grep "$1" /etc/passwd >/dev/null
then
    echo "Benutzer $1 ist dem System bekannt"
    exit 0
else
    echo "Benutzer $1 ist dem System unbekannt"
    exit 1
fi
```

Der exit-Kode kann übrigens auch ausgewertet werden, er wird immer in dem Parameter $? zurückgegeben. Dies veranschaulichen die folgenden Aufrufe mit ihren Ausgaben:

Bedingung nicht erfüllt:

```
test "n" = "j"
echo $?
```

Die Ausgabe lautet:

```
1
```

Bedingung erfüllt:

```
test "j" = "j"
echo $?
```

Die Ausgabe lautet:

```
0
```

if-Anweisung können auch geschachtelt werden, was aber schnell zu unübersichtlichen Skripten führt:

```
if [ ! -a "$Datei" ]
then
    echo "$Datei ist nicht vorhanden"
else
    if [ -d "$Datei" ]
    then
        echo "$Datei ist ein Verzeichnis"
    else
        if [ -c "$Datei" ]
        then
            echo "$Datei ist ein Char Device"
        fi
    fi
fi
```

Übersichtlicher geht es bei gleicher Semantik mit dem Konstrukt elif:

```
 if [ ! -a "$Datei" ]
then
    echo "$Datei ist nicht vorhanden"
elif [ -d "$Datei" ]
then
   echo "$Datei ist ein Verzeichnis"
elif [ -c "$Datei" ]
then
   echo "$Datei ist ein Char Device"
fi
```

Die zweite Ablaufstruktur für Verzweigungen ist case, das dem case in C ähnlich ist. case prüft nicht den exit-Kode eines Kommandos, sondern es vergleicht einen Parameterwert mit einer Liste von Werten.

```
case Parameterwert in
     Liste1) Kommandos1 ;;
     Liste2) Kommandos2 ;;
     ...
     ListeN) Kommandos N;;
esac
```

Gibt es einen Wert in der Liste, der mit dem Parameterwert übereinstimmt, so werden die Kommandos hinter der Liste ausgeführt. Dabei werden nur die Kommandos hinter dem ersten zutreffenden Muster ausgeführt. Anschließend wird das Skript nach dem Ende der gesamten case-Anweisung fortgesetzt.

Listen in case-Anweisungen können aus konstanten Zeichenketten bestehen, die durch | voneinander getrennt werden. Der folgende Skript-Ausschnitt testet z.B., wie die Variable antwort gesetzt ist:

```
case "$antwort" in
Ja | ja | JA | jA) exit 0;;
Nein | nein | N | n) exit 1;;
esac
```

Listen können auch durch Muster (Wildcards) gebildet werden. So können wir den obigen Skript-Ausschnitt wie folgt abkürzen:

```
case "$antwort" in
[Jj][Aa]) exit 0;;
[Nn]ein | [Nn]) exit 1;;
esac
```

Hierbei stehen die eckigen Klammern für die Auswahl eines Zeichens aus der vorgegebenen Menge. Weitere Wildcards sind * für beliebig viele (auch kein) Zeichen und ? für genau ein beliebiges Zeichen. Der * wird häufig für den Default-Fall in der case-Anweisung benutzt. Als Beispiel dazu folgt ein Skript, das ein versiegeltes System für Benutzer realisiert, in dem sie nur bestimmte Befehle aufrufen dürfen:

```
#!/bin/ksh
echo "Bitte Kommando eingeben"
read cmd REST
case "$cmd" in
cal | date | find | ls | ps) ksh -c "$cmd $REST";;
*) echo "Kommando nicht erlaubt"
esac
```

Ein erlaubtes Kommando wird mit Parametern mit ksh -c ausgeführt; nicht erlaubte Kommandos werden abgewiesen.

Wenn man sich Skripten z.B. im Systembereich von Linux (Startverzeichnis /sbin/init.d) ansieht, fallen einem weitere Verzweigungsbefehle auf. Sie werden durch die Operatoren && - logisches Und - bzw. || - logisches Oder - dargestellt. Hierbei kann ein zweiter Befehl in Abhängigkeit vom Erfolg, d.h. exit-Kode 0, oder Mißerfolg, d.h. exit-Kode ungleich 0, eines ersten Befehls gestartet werden.

&& hat folgende Syntax:

```
Befehl1 && Befehl2
```

Dies hat die Bedeutung: Falls Befehl1 erfolgreich war, wird Befehl 2 ausgeführt, sonst nicht. Man kann sich das leicht merken, denn optimierende C-Compiler werten auf diese Weise logische Und-Verknüpfungen in Bedingungen aus. Wenn die erste Bedingung nicht zutrifft, braucht die zweite Bedingung erst gar nicht überprüft zu werden.

Es folgen zwei Beispiele für die Anwendung von &&:

```
ls adressen.dat >/dev/null 2>/dev/null && vi adressen.dat
[ -f adressen.dat ] && vi adressen.dat
```

|| hat folgende Syntax:

```
Befehl1 || Befehl2
```

Dies hat die Bedeutung: Falls Befehl1 fehlgeschlagen war, wird Befehl2 ausgeführt, sonst nicht. Dies hat die analoge Entsprechung bei der Bedingungsauswertung in optimierenden C-Compilern.

Es folgen zwei Beispiele für die Anwendung von ||:

```
grep Meier /etc/passwd || echo "Meier ist nicht verzeichnet"
[ -a tmpdir ] || mkdir tmpdir
```

Die Befehle && und || können auch kombiniert auftreten. Hierbei wird dann streng von links nach rechts ausgewertet.

```
Befehl1 || Befehl2 || Befehl3
```

Das Ergebnis von Befehl1 || Befehl2 gilt als nicht erfolgreich, wenn beide Befehle einen exit-Kode ungleich 0 zurücklieferten. Nur in diesem Fall wird Befehl3 ausgeführt.

```
Befehl1 && Befehl2 || Befehl3
```

Das Ergebnis von Befehl1 && Befehl2 gilt als nicht erfolgreich, wenn einer der beiden Befehle nicht erfolgreich war. Nur in diesem Fall wird Befehl3 ausgeführt.

```
Befehl1 || Befehl2 && Befehl3
```

Das Ergebnis von Befehl1 || Befehl2 gilt als erfolgreich, wenn einer der beiden Befehle erfolgreich war. Nur in diesem Fall wird Befehl3 ausgeführt.

```
Befehl1 && Befehl2 && Befehl3
```

Das Ergebnis von Befehl1 && Befehl2 gilt als erfolgreich, wenn beide Befehle erfolgreich waren. Nur in diesem Fall wird Befehl3 ausgeführt.

Hierzu noch ein paar Beispiele:

```
[ -a text.dat ] && cp text.dat text.bak || \
echo "text.dat nicht gefunden oder cp misslungen"
who | grep kremer >/dev/null && echo "kremer ist angemeldet" || \
echo "kremer ist nicht angemeldet"
```

Das \-Zeichen dient hier, einen Befehl in mehrere Zeilen schreiben zu können.

Manchmal müssen mehrere Befehle im Anschluss an && oder || ausgeführt werden. Für diesen Fall können Kommados mit Hilfe der Klammern { } gruppiert werden:

```
[ -a text.dat ] && { cp text.dat text.bak; vi text.dat } || \
echo "text.dat nicht gefunden oder Edieren misslungen"
```

2.7 Integer-Arithmetik

Um mit ganzzahligen Werten zu rechnen, gibt es zwei Möglichkeiten: den Befehl expr und den Befehl let. Während let ein built-in-Befehl der Korn-Shell ist, ist expr ein separates Programm. expr ist nicht zuletzt deswegen um einiges langsamer als let, aber es beinhaltet auch einige Zeichenkettenoperationen.

Operatoren von expr sind +, -, *, / und % für den ganzahligen Divisionsrest. expr wird angewendet, indem Rechenergebnisse durch Kommandosubstitution auf Variablen zugewiesen werden, z.B.:

```
i=2
Ergebnis = ‘expr $i + 10 \* 2‘
echo $Ergebnis
22
```

Da * ein Maskenzeichen darstellt, wird es durch den vorangestellten \ vor der Interpretation durch die Shell geschützt.

Die Operatoren von let entsprechen i.w. den ganzzahligen Operatoren in C, so gibt es neben +, -, *, / und % auch Operatoren wie +=, -=, *=, /= und %=. Der mit let auszuwertende Ausdruck muss entweder ohne Leerzeichen geschrieben werden, oder er muss in doppelte Hochkommata eingeschlossen werden. Neben dem Wort let ist eine alternative Schreibweise gebräuchlich, die aus zwei öffnenden und schließenden runden Klammern besteht. In dieser Schreibweise dürfen auch beliebig Leerzeichen vorkommen. Führende $-Zeichen dürfen bei der Bezeichnung von Variableninhalten weggelassen werden. Es folgen einige Beispiele für korrekte let-Ausdrücke:

```
let i=i+1
let “i = i + 1”
let i+=1
(( i += 1 ))
(( Zahl1 = 10 ))
(( Zahl2 = 15 ))
(( Ergebnis = ((15 + Zahl1) * Zahl2) *10 ))
echo $Ergebnis
3750
```

Neben Zeichenketten-Variablen können in der Korn-Shell auch ganzzahlige Variablen vom Typ integer vereinbart werden. Dies beschleunigt die let-Arithmetik, da dann Typumwandlungen von Zeichenketten in ganzzahlige Werte und umgekehrt entfallen:

```
integer i=2 j=5
(( i+=j*2 ))
echo $i
12
```

Ist die Variable auf die zugewiesen wird vom Typ integer, so kann die doppelte Klammer oder let auch fehlen. Auf der rechten Seite des Gleichheitszeichens dürfen integer-Variablen und Zeichenketten-Variablen, die in integer konvertierbar sind, stehen:

```
Menge=10; Preis=25; integer Gesamt=0
Gesamt=116*Menge*Preis/100
echo $Gesamt
290
```

In einer integer-Deklaration kann einer Variable auch ein Ausdruck zugewiesen werden:

```
integer a=b*3/4
```

Der Befehl integer ohne Argumente zeigt alle definierten integer-Variablen mit ihren Werten an.

Neben der Arithmetik wird let auch für Vergleiche von ganzen Zahlen verwendet. Es ist dann um einiges schneller als der test-Befehl:

```
(( Zahl1 == Zahl2 ))
(( Zahl1 != Zahl2 ))
(( Zahl1 <= Zahl2 ))
(( Zahl1 >= Zahl2 ))
(( Zahl1 < Zahl2 ))
(( Zahl1 > Zahl2 ))
(( Zahl1 == Zahl2 ))
```

Zahlenvergleiche können mit dem Und-Operator && und mit dem Oder-Operator || miteinander verknüpft werden:

```
i=10
j=30
(( i < 15 && j > 40 ))
echo $?
1
(( i < 15 || j > 40 ))
echo $?
0
```

Innerhalb eines Vergleichs können auch arithmetische Ausdrücke stehen:

```
(( i < 15 && j + 20 > 40 ))
echo $?
0
```

Die abkürzende Schreibweise von let mit den Vergleichsoperatoren ==, !=, <=, >=, < und > führt zu übersichtlicheren Bedingungen als beim test-Befehl:

```
if test "$anz" -gt "$max"
then
...
fi
if (( anz > max ))
then
...
fi
```

2.8 Schleifen

In der Korn-Shell gibt es zwei Arten von Schleifen:
Bedingungsschleifen mit der Ablaufstruktur while oder until,
Zählschleifen mit der Ablaufstruktur for.
Die while-Schleife hat eine ähnliche Syntax wie die if-Verzweigung:

```
while <Kommando>
do
<Kommandos>
done
```

Die Kommandos im Schleifenkörper werden solange ausgeführt, bis der exit-Kode des Kommandos hinter dem while ungleich 0 ist. Die Kommandos können einmal, mehrmals oder auch gar nicht durchlaufen werden. Letzteres ist der Fall, wenn das Kommando hinter dem while beim ersten Aufruf einen exit-Kode ungleich 0 zurückliefert.

Die Bedingung einer while-Schleife ist häufig ein test-Befehl; es kann aber jedes andere Kommando verwendet werden, z.B. auch let als Vergleichsbefehl. Die folgende Schleife zählt einen Index hoch:

```
i=1
while (( i <= 4 ))
do
echo $i
(( i += 1 ))
done
```

Das folgende Skript schreibt einen Text in eine Datei:

```
#!/bin/ksh
(( $# != 1 )) && { echo "Aufruf: $0 <Dateiname>; exit 1 }
Prompt = "> >"
Ausgabe = $1
>$Ausgabe #Ausgabedatei wird neu angelegt
echo "Geben Sie Text ein (e-Ende)"
echo "$Prompt\c"; read Eingabe
while [ "$Eingabe" != "e" ]
do
   echo "$Eingabe">> $Ausgabe
   echo "$Prompt\c"; read Eingabe
done
```

Das \c in der echo Anweisung dient dazu, den Zeilenvorschub nach der echo-Anweisung zu unterdrücken.

Die until-Schleife ist die logische Negation der while-Schleife. Hier werden die Kommandos im Schleifenkörper solange ausgeführt, bis das Bedingungskommando den exit-Kode 0 zurückliefert.

Gleichbedeutend sind z.B.:

```
while (( i <= 4 ))
do
...
done
until (( i > 4 ))
do
...
done
```

until- oder while-Schleifen werden häufig in Hintergrundprozessen verwendet. Das folgende Skript blogin stellt die Anmeldung eines Benutzers fest und sendet diesem Benutzer anschließend eine Nachricht mit dem Befehl write:

```
(( $# != 1 )) && { echo "Aufruf: $0 <Benutzername>"; exit 1 }
Schlafzeit=60
Prot=$HOME/.logprot # Protokolldatei
until who | grep $1 > /dev/null
do
   sleep $Schlafzeit
   # Schlafen, um den Prozessor nicht unnötig zu belasten
done
write $1 "Gotcha"
Zeile="Anmeldung von $1 um `date`"
echo $Zeile >> $Prot # Meldung in Protokolldatei
echo $Zeile # interaktive Meldung
```

Dieses Skript wird im Hintergrund gestartet, z.B.:

```
blogin root &
```

while und until werden in Hintergrundprozessen auch häufig als Endlosschleifen eingesetzt. Ein solcher Prozess wird dann i.d.R. erst durch ein Signal z.B. beim Herunterfahren des Rechners beendet:

```
while true
do
...
done
until false
do
...
done
```

Die for-Schleife arbeitet im Vergleich zur while- oder until-Schleife völlig unterschiedlich. Zunächst betrachten wir ihre Syntax:

```
for <Variable> in <Zeichenkette1> ... <ZeichenketteN>
do
   <Kommandos>
done
```

Bei den Schleifendurchläufen nimmt die Variable nacheinander die Werte der in einer Liste angegebenen Zeichenketten an. Dabei wird die Variable i.d.R. in den Kommandos des Schleifenkörpers verwendet. Hierzu ein Beispiel. Folgendes Skript sichert eine Folge von Dateien, die in der Liste der for-Anweisung explizit angegeben sind:

```
#!/bin/ksh
for Datei in passwd group services
do
```

```
    # Zugriffsrechte rw-------
    chmod u+w $Datei.bak.gz 2>/dev/null
    cp $Datei $Datei.bak
    gzip $Datei.bak
    # Zugriffsrechte r--------
    chmod 400 $Datei.bak.gz
done
```

Alternativ zu der expliziten Angabe der Liste im Skript kann sie auch mit Hilfe der Aufrufparameter übergeben werden. Wir sehen hier eine Anwendung der Variablen #@:

```
for Datei in #@
do
...
done
```

Der entsprechede Aufruf des Skriptes lautet:

```
backup passwd group services
```

Man kann sogar den Parameter #@ weglassen, da er den Default-Wert in der for-Liste darstellt:

```
for Datei
do
...
done
```

Durch Verwendung von Mustern für Dateinamen können ebenfalls Listen für for generiert werden:

```
# alle mit ls ohne Optionen sichtbaren Dateien
# des aktuellen Verzeichnisses
for Datei in *
do
...
done
# alle Punktdateien mit mindestens drei Stellen
# also nicht . und ..
for Datei in .??*
do
...
done
```

Die for-Liste kann durch Kommandosubstitution auch dynamisch generiert werden. Hierfür gibt es natürlich sehr viele Anwendungen. Das folgende Skript legt eine Sicherungskopie von allen Dateien mit der Erweiterung .txt unterhalb des Heimatverzeichnisses an:

```
#!/bin/ksh
for Datei in `find $HOME -name "*.txt" -print 2>/dev/null`
```

```
do
   if [ -f $Datei ]
   then
      cp $Datei $Datei.bak
   fi
done
```

2.9 continue und break

Wie in C gibt es auch in der Korn-Shell Befehle, um Schleifen zusätzlich zu steuern.

continue wird angewendet, wenn Befehle innerhalb eines Schleifenkörpers nicht ausgeführt werden sollen. Es wird dann, direkt mit dem Kommando im Schleifenkopf fortgefahren. continue kann bei jeder Schleifenart for, while oder until verwendet werden. Betrachten wir folgendes Beispiel:

```
#!/bin/ksh
i=0
while (( i < 5 ))
do
   ((i+=1))
   (( i == 3 )) && continue # Überspringe Rest
echo $i
done
```

Der Aufruf dieses Skriptes ergibt als Ausgabe:

```
1
2
4
5
```

Bei einer for-Schleife können mit der continue-Anweisung z.B. bestimmte Zeichenketten übergangen werden, für die eine Verarbeitung nicht sinnvoll ist. Folgendes Skript soll Informationen aus der Datei /etc/passwd für Benutzernamen formatiert ausgeben:

```
#!/bin/ksh
(( $# == 0 )) && { echo "Aufruf: $0 <Nutzer1>...<NutzerN>"; exit 1 }
for Benutzer
do
   info=`grep "^$Benutzer:" /etc/passwd`
   IFS=:
   set $info
   echo "Kennung: $1"
...
done
```

Falls der Benutzer verzeichnet ist, wird mit dem grep-Befehl eine Zeile aus der Datei /etc/passwd in die Zeichenketten-Variable info übergeben. Das ^-Zeichen

zeigt grep an, dass $Benutzer am Zeilenanfang gesucht werden muss. Danach wird die Zeile in der Variablen info mit Hilfe des set-Kommandos in einzelne Worte zerlegt. Das Trennzeichen zwischen zwei Worten wird durch die Anweisung IFS=: als Doppelpunkt festgelegt. set $info weist nun der Variablen $1 das erste Wort, der Variablen $2 das zweite Wort usw. automatisch zu.

Wenn der übergebene Benutzername nicht in /etc/passwd verzeichnet ist, macht dieses Skript jedoch einen Fehler. Die Variable info ist in diesem Fall leer. Es wird anschliesend das set-Kommando ohne Parameter aufgerufen. Dies führt zur Ausgabe aller Umgebungsvariablen. Also darf set und die restlichen Befehle des Schleifenkörpers nicht ausgeführt werden, wenn $info leer ist. Dies programmieren wir durch folgende Erweiterung:

```
...
info=‘grep “^$Benutzer:“ /etc/passwd‘
if [ -z “$info” ]
then
     echo “Benutzer: $Benutzer ist unbekannt”
     continue
fi
...
```

Im Gegensatz zum continue-Befehl verlässt der break-Befehl die Schleife (nicht das Skript), d.h. es wird mit dem Befehl, der dem Schleifenende folgt, fortgefahren.

2.10 Umlenkung von Standardein- und -ausgabe

Ausgaben mehrerer gruppierter Kommandos können in folgender Weise in eine Datei umgeleitet werden:

```
{ pwd; echo “Inhaltsverzeichnis”; ls -l } > Inhalt
```

Auch die Ausgaben von Kommandos, die in Schleifen ausgeführt werden, können umgeleitet werden:

```
while/until/for ...
do
   <Kommandos>
done > Ausgabe
```

Außer > sind auch die Operatoren >> und 2> erlaubt. 2> leitet die Standardfehlerausgabe um.

Als Beispiel betrachten wir das folgende Skript, das eine Liste der Dateien aller übergebener Benutzer in die Datei Suchliste abstellt:

```
#!/bin/ksh
for name
do
   echo “Dateien des Benutzers $name”
   find / -user $name -print
done > Suchliste 2>/dev/null
```

Die Ausgabe einer Schleife kann auch einem weiteren Kommando über den Pipe-Mechanismus übergeben werden:

```
while/until/for ...
do
   <Kommandos>
done | <Kommando>
```

Auch die Umlenkung der Eingabe lässt sich für alle Schleifenarten durchführen:

```
while/until/for ...
do
   <Kommandos>
done < Einabe
```

Folgendes Skript liest eine Eingabedatei zeilenweise ein und schreibt nummerierte Zeilen in eine Ausgabedatei:

```
#!/bin/ksh
Eingabe=$1
integer i=1
while read satz
do
   echo "$i: $satz"
   (( i+=1 ))
done < $Eingabe >$Eingabe.nummeriert
```

Es ist auch möglich, die Ausgabe eines Kommandos über eine Pipe als Eingabe an die Schleifenkommandos weiterzuleiten, z.B.:

```
sort /etc/passwd |
while read kennung ...
do
   echo ...
done
```

2.11 Signale und deren Behandlung

Die Tastenkombination Strg-c oder Ctrl-c beendet im Normalfall ein laufendes Skript und man befindet sich wieder in der Ausgangs-Shell. Der Betriebssystemkern informiert hierbei den laufenden Prozess durch ein Signal. Mit dem Befehl trap können Signale erkannt und durch Programmstücke behandelt werden. Man spricht davon, dass ein Signal abgefangen wurde. Das trap-Kommando hat folgende Syntax:

trap "<Kommando1>; ... <KommandoN>" <Signal1> ... <SignalM>

Bei Eintreffen eines der Signale, wird dann die Liste der in Hochkommata angegebenen Kommandos abgearbeitet. Anschließend fährt das Skript an der Stelle fort, an der es unterbrochen wurde. Lässt man die Kommandoliste leer, so werden die angegebenen Signale einfach ignoriert, d.h. sie führen nicht zum Abbruch.

Folgendes Skript zeigt die Anwendung des trap-Kommandos:

```
#!/bin/ksh
trap "so lasse ich mich nicht abbrechen..." 2 3 15
integer i=0
while (( i < 5 ))
do
   echo "Schleifendurchlauf $i
   (( i+=1 ))
   sleep 10
done
```

trap-Kommandos können wiederholt werden. Insbesondere kann die Behandlung eines Signals wieder in den Normalzustand zurückversetzt werden. Dies kann z.B. sinnvoll sein, wenn nur ein bestimmter Teil eines Skripts nicht unterbrochen werden darf:

```
...
trap " " 2 # Abbruch mit <del> ignoriert
format $1 || exit 1
trap 2 # Abbruch mit <del> wieder möglich
cp * $2
...
```

Kapitel 3

Programmentwicklung in UNIX

3.1 Übersetzen und Binden von Programmen

Charakteristisch ist unter UNIX die Bereitstellung von Dienstprogrammen, die als Rahmen- oder Treiberprogramme je nach Bedarf Übersetzer, Optimierer und Binder eigenständig aufrufen. Die notwendigen Informationen über die auszuführenden Aktionen beziehen die Treiber aus den Endungen der Dateinamen, die die Dateiinhalte kennzeichnen und deshalb einheitlich verwendet werden müssen:

Dateiendung	Dateityp
.a	Programmbibliothek
.c	C-Quelldatei
.f	Fortran-Quelldatei
.h	C-Header-Datei
.i	C-Quelldatei nach Präprozessor
.o	Objektdatei
.s	Assembler-Datei

Die Rahmenprogramme zur Verarbeitung von C-Quelldateien heißen cc oder gcc. gcc steht für den Treiber des GNU-Compilers und wird unter Linux hauptsächlich verwendet.

Beim Aufruf eines Compiler-Treibers können sowohl Übersetzer- als auch Binder-Optionen sowie ein oder mehrere Dateinamen mit unterschiedlichen Dateiendungen übergeben werden. Die am häufigsten verwendeten Übersetzeroptionen für C werden in der folgenden Tabelle erläutert:

Option	Beschreibung
-c	Quelle wird nur übersetzt nicht gebunden
-D	Vereinbarung eines #define
-U	Löschen eines #define
-g	Debug-Informationen einfügen
-I	Suchpfad für Includes ergänzen
-O	Objektkode optimieren
-E	nur Präprozessorlauf auf stdout ausgeben
-v	Arbeitsschritte des Treibers auf stderr ausgeben
-Wa,<Optionen>	Optionen für den Assembler

Die häufigsten Bindeoptionen zeigt die folgende Tabelle:

Option	Beschreibung
-lname	Bibliothek libname.a wird aufgesucht
-Ldir	dir in den Bibliotheks-Suchpfad aufnehmen
-o name	gebundenes Programm unter name speichern
-p	Programm zum Profiling für prof binden

Es folgen einige Beispiele zum Umgang mit gcc:

Erzeugen einer ausführbaren Datei a.out:

```
gcc prog.c
```

Erzeugen einer ausführbaren Datei prog:

```
gcc -o prog prog.c
```

Erzeugen des Objektkodes in der Datei test.o:

```
gcc -c prog.c
```

Erzeugen einer ausführbaren Datei a.out mit Debug-Informationen:

```
gcc -g prog.c
```

Erzeugen einer ausführbaren Datei prog mit Debug-Informtionen:

```
gcc -o prog -g prog.c
```

Erzeugen einer optimierten ausführbaren Datei prog ohne Debug-Informationen:

```
gcc -o prog -O test.c
```

Aktuelles Verzeichnis (.) beim Suchpfad für Include-Dateien ergänzen:

```
gcc -I. prog.c
```

Binden von Modulen aus der Mathematik-Bibliothek libm.a:

```
gcc -lm prog.c
```

Erzeugen eines Listing aus Assembler-Kode mit zugehörigen C-Quellzeilen in assembler.list. -alh,-L sind dabei Anweisungen an den Assembler

```
gcc -g -Wa,-alh,-L prog.c > assembler.list
```

3.2 Debugger

Programme sind während ihrer Entwicklung selten fehlerfrei. Syntaxfehler wie vergessene Klammern oder falsch formulierte Anweisungen werden vom Übersetzer entdeckt. Schwieriger sind Fehler zu entdecken, die erst zur Laufzeit auftreten und zu falschen Ergebnissen oder zu Programmabbrüchen führen. Man kann durch Ausgaben (printf() in C) aus dem Programm versuchen solchen Fehlern auf die Spur zu kommen. Für größere Programme oder kompliziertere Abläufe ist das aber sehr mühsam. Daher gibt es Dienstprogramme zum Auffinden von Laufzeitfehlern. Solche Programme heißen Debugger (wörtlich übersetzt Entwanzer).

Moderne Debugger, wie ddd (Data Display Debugger) oder ups, erlauben die schrittweise Ausführung der Programme mit Ansicht des Quellkodes und der momentanen Belegung der Variablen. Dabei werden die Funktionen beim Debugging komfortabel mit der Maus in einer Fensteroberfläche gesteuert.

3.3 Make-Tool und Bibliotheken

Das Spektrum realer Programme reicht von einigen 100 bis zu einigen 100 000 Programmzeilen. Moderne Programmiersprachen unterstützen durch ihre Sprachdefinition die Zerlegung eines großen Programms in mehrere kleinere übersichtlichere Teile. In C wird ein Programm in mehrere Funktionen zerlegt. Ein Modul entspricht in C einer Datei, die eine oder mehrere inhaltlich zusammenhängende Funktionen beinhaltet.

Die Übersetzung einzelner Module von Hand ist bei größeren Programmsystemen fehleranfällig und dauert häufig länger als nötig. Probleme sind dabei:

- vergessene Übersetzungen
- falsche Übersetzeroptionen
- falsche Versionen dazugebundener Bibliotheken und Module
- unnötige Neuübersetzungen

Das Werkzeug make steuert die Übersetzung größerer Programmsysteme mit den Zielen Vereinfachung und Beschleunigung der Gesamtübersetzung. Bei richtiger Anwendung von make werden die Programme immer mit den richtigen Optionen übersetzt und mit den aktuellen Versionen anderer Module oder Bibliotheken gebunden.

Die Optimierungsidee von make ist dabei sehr einfach: Es werden nur Quelldateien neu bearbeitet, wenn sich die zu erzeugenden Zieldateien seit der letzten Bearbeitung voraussichtlich ändern werden. Kriterium dafür sind die Modifikationszeitpunkte der Quell- und Zieldateien. Ist der Modifikationszeitpunkt mindestens einer Quelldatei jünger als der Modifikationszeitpunkt der Zieldatei, so wird die Zieldatei meist mit Hilfe der Quelldatei neu bearbeitet.

Natürlich kann man eine solche Optimierung auch ohne make durch geeignete Compiler-Aufrufe selber vornehmen, aber die Fehlergefahr wird bei großen Systemen schnell zu einem Risiko.

Hierzu ein Beispiel: Nehen wir an, dass wir 3 Module main.c, swap.c und sum.c und eine Header-Datei globals.h erstellt haben. Initial können wir die

Module durch folgenden Aufruf übersetzen und zum ausführbaren Programm prog binden lassen.

```
gcc -o prog main.c swap.c sum.c
```

Wir stellen nun fest, dass wir im Modul main.c noch etwas ändern müssen. Anstatt nun swap.c und sum.c neu zu übersetzen, binden wir die bereits erzeugten Objekt-Dateien swap.o und sum.o mit main.o nach dessen Übersetzung. Dies können wir über den Compiler-Treiber wie folgt realisieren:

```
gcc -o prog main.c swap.o sum.o
```

Wir können den Übersetzungsvorgang mit make so automatisieren, dass solche Situationen erkannt und optimiert bearbeitet werden. Die Steuerdatei z.B. mit dem Namen makefile hat folgenden Inhalt:

```
# Compile and link main.c, swap.c, sum.c
prog: main.o swap.o sum.o
     gcc main.o swap.o sum.o -o prog
main.o: main.c globals.h
     gcc -c main.c
swap.o: swap.c
     gcc -c swap.c
sum.o: sum.c
     gcc -c sum.c
```

Der Aufruf make benutzt die Steuerdatei mit dem Namen makefile. Will man einen anderen Namen der Steuerdatei verwenden, z.B. makefile1, so ruft man make wie folgt auf:

```
make -f makefile1
```

Das #-Zeichen leitet einen Kommentar ein. Die Steuerdatei besteht aus Abhängigkeitszeilen und Kommandozeilen. Eine Abhängigkeitszeile enthält das Ziel und Dateien von denen das Ziel abhängt. Einer Abhängigkeitszeile folgen u.U. mehrere Kommandozeilen, die immer mit einem Tabulatorzeichen beginnen müssen. Die Kommandozeilen werden nun ausgeführt, wenn das Ziel älter als eine der Quellen ist oder wenn das Ziel nicht existiert. Kommandozeilen müssen nicht unbedingt etwas mit der Übersetzung von Programmen zu tun haben, obwohl make natürlich dazu in erster Linie konzipiert wurde.

Beim Aufruf

```
make
```

wird das Ziel der ersten Abhängigkeitszeile untersucht, also ist die Reihenfolge der Abhängigkeitszeilen wichtig.

Beim Aufruf

```
make prog
```

oder

```
make swap.o
```

wird das Ziel unabhängig von der Postition in der Steuerdatei erzeugt.

Nehmen wir an keine der Zieldateien existiert und wir rufen make das erste mal auf. Was passiert?

Zunächst wird die erste Abhängigkeitszeile untersucht und festgestellt, dass das Ziel nicht existiert aber auch keine der Quelldateien. Nun versucht make sukzessive die Quellen main.o, swap.o und sum.o zu erzeugen, d.h. für diese Quellen Abhängigkeitszeilen zu finden, in denen sie als Ziele vorkommen, z.B.:

```
main.o: main.c globals.h
```

Auf diese Weise werden dann main.o, swap.o und sum.o erzeugt und schließlich hieraus dann prog.

Für den oberen Fall, dass nun nur main.c verändert wurde und alle anderen Quellen unverändert blieben verfährt make wie folgt:

Zunächst wird wieder die erste Abhängigkeitszeile und dann die Abhängigkeitszeilen, in denen die Quelldateien dieser Zeile als Ziele auftauchen, untersucht. Dabei wird nur nach der Abhängigkeitszeile

```
main.o: main.c globals.h
```

ein Compile-Schritt gestartet:

```
gcc -c main.c
```

Schließlich wird noch der Bindevorgang nach der ersten Abhängigkeitszeile durchgeführt:

```
gcc -o prog main.c swap.o sum.o
```

Über sogenannte Regeln weiß make mit Hilfe fest vereinbarter Erweiterungen der Dateinamen (Extensions) wie aus C, C++ oder Fortran-Quellen Zieldateien gebildet werden, z.B. bedeutet .c nach .o Übersetzen einer C-Quelldatei ohne Binden. In der Steuerdatei können Makrovariablen angegeben werden, die von der jeweiligen eingebauten Regel benutzt werden, z.B. CFLAGS vom C-Compiler. Hierüber kann der Anwender die Regeln steuern. Makrodefinitionen können statt in der Steuerdatei auch in der Kommandozeile angegeben werden, z.B.:

```
make CFLAGS=-g
```

Das Kommando

```
make -p
```

zeigt die Liste aller vordefinierten Makrovariablen und Regeln.

Die Steuerdatei für die obigen 3 Module können wir durch die Anwendung der implizit bekannten Regeln wie folgt vereinfachen:

```
# Compile and link main.c, swap.c, sum.c
CFLAGS=-O # Makrovariable zur Übersetzung
prog: main.o swap.o sum.o
    gcc main.o swap.o sum.o -o prog
# Abhängigkeit von globals.h kennt make sonst nicht.
main.o: main.c globals.h
```

Die Abhängigkeits- und Kommandozeilen zur Übersetzung der Module swap.c und sum.c können entfallen, da es eine Regel gibt, mit der implizit festgelegt ist, wie aus .c Dateien .o Dateien entstehen. Die Abhängigkeitszeile zu main.o muss erhalten bleiben, da sonst eine Veränderung an globals.h auch zu einer Neuübersetzung von main.c führen muss. Schließlich kann make auch nicht wissen, wie aus den Modulen das Gesamtprogramm gebunden wird. Deswegen bleibt die erste Abhängigkeitszeile mit ihrer Kommandozeile erhalten.

Bei größeren Projekten werden Gruppen von Funktionen in unterschiedliche Bibliotheken abgestellt oder Funktionen aus existierenden Bibliotheken verwendet. Zur Verwaltung von Bibliotheken gibt es das Kommando ar mit folgender Syntax:

```
ar <Schalter> <Bibliotheksdatei> [Dateien]
```

Die wichtigsten ar-Schalter sind:

q: Datei(en) hinzufügen

r: Datei(en) ersetzen oder hinzufügen

u: in Kombination mit r nur neuere Datei(en) als im Archiv ersetzen

d: Datei(en) löschen

p: Datei(en) listen

x: Datei(en) extrahieren, Archiv bleibt unverändert

t: Inhaltsverzeichnis des Archivs listen

s: Symboltabelle regenerieren (alternativ ranlib-Befehl)

v: verbose alle Meldungen ausgeben

Durch die implementierte ar-Regel ist eine Steuerdatei zur Generierung und Verwendung einer Programmbibliothek sehr leicht herzustellen.

```
# Compile and link "main.c" with "libaux.a"
# Generate and update "libaux.a"
CC = gcc
ARFLAGS = rsv
CFLAGS = -O
prog: main.o libaux.a
     $(CC) main.o -L. -laux
libaux.a: libaux.a(swap.o) libaux.a(sum.o)
```

Über die Makrovariable ARFLAGS werden ar-Programm die Schalter mitgegeben. In der Zeile

```
$(CC) main.o -L. -laux
```

bedeutet der Parameter -L., dass der Suchpfad für Bibliotheken um das aktuelle Verzeichnis . erweitert wird. Der Parameter -laux besagt, dass die Bibliothek mit dem Namen libaux.a konsultiert werden soll.

Die letzte Abhängigkeitszeile hat eine spezielle Syntax, da die ar-Regel Bibliothekseinträge als Abhängigkeiten verwendet und nicht .o-Dateien.

Über vordefinierte Makrovariablen hinaus können auch benutzerdefinierte Variablen verwendet werden:

```
# Compile and link main.c, swap.c, sum.c
CC = gcc
CFLAGS = -O
OBJECTS = main.o swap.o sum.o # benutzerdefiniert
prog: $(OBJECTS)
$(CC) $(OBJECTS) -o prog
main.o: main.c globals.h
```

Die Variable OBJECTS ist keine vordefinierte sondern eine benutzerdefinierte Variable. Sollten sich die Namen der Module ändern, so muss die Steuerdatei nun nur noch an zentraler Stelle geändert werden.

Etwas komplizierter ist die Erzeugung benutzerdefinierter Regeln. Es wird aber i.d.R. auch nicht notwendig sein. Als Beispiel wird eine Regel in einer Steuerdatei definiert, mit der aus einer Fortran-Quelldatei mit der Endung .f eine Objektdatei mit der Endung .o erzeugt wird:

```
# Compile and link main.f, swap.f, sum.f
FC=f77
FFLAGS=-O
OBJECTS=main.o swap.o sum.o
.SUFFIXES: .f
# Regel, um .o-Objekt aus .f-Fortran-Quelle zu generieren
.f.o:
        $(FC) -c $(FFLAGS) $<
a.out: $(OBJECTS)
        $(FC) $(OBJECTS)
```

Mit der Pseudo-Abhängigkeitszeile

```
.SUFFIXES: .f
```

wird zunächst angezeigt für welche Namenserweiterung (hier .f) eine Regel definiert wird. Die beiden darauf folgenden Zeilen definieren die Regel, wobei $< eine Spezialvariable ist, die für den Namen der Datei steht (hier eine .f-Datei), die die Ausführung der Regel verursacht. Weitere Spezialvariablen zur Formulierung von Regeln sind:

$<: Name der Datei, die die Ausführung der Regel verursacht

$*: wie $< nur ohne Endung

$@: Name der Zieldatei

Eine Steuerdatei mit benutzerdefinierter Regel zur Verwaltung einer Bibliothek könnte wie folgt aussehen:

```
# Compile and link main.f with libaux.a
# Generate and update libaux.a
ARFLAGS=rsv
FC = f77
FFLAGS=-O
RM=rm
.SUFFIXES: .f
.f.a:
      $(FC) -c $(FFLAGS) $<
      $(AR) $(ARFLAGS) $@ $*.o
      $(RM) $*.o
.f.o:
      $(FC) -c $(FFLAGS) $<
a.out: main.o libaux.a
      $(FC) main.o -L. -laux
libaux.a: libaux.a(swap.o) libaux.a(sum.o)
```

Zur Abarbeitung einer Befehlssequenz aus einer Steuerdatei beim Aufruf dienen Pseudoziele, z.B. das Pseudoziel clean beim Aufruf make clean. Pseudoziele haben keine Vorfahren und werden nie generiert, sind also nie aktuell, d.h. die zugehörigen Kommandos werden auf jedem Fall ausgeführt. Das folgende Beispiel zeigt eine Steuerdatei mit 3 Pseudozielen:

```
# Compile and link main.c, swap.c, sum.c
CC = gcc
CFLAGS = -O
OBJECTS = main.o swap.o sum.o
info:
@echo Please type: make all, make prog or make clean
all:
     $(MAKE) clean
     $(MAKE) prog
clean:
     -rm $(OBJECTS) prog
prog: $(OBJECTS)
      $(CC) $(OBJECTS) -o prog
main.o: main.c globals.h
```

Der Aufruf make ohne Ziel oder make info bewirkt die Ausgabe des Hilfetexts.

Wie man beim Pseudoziel make all sieht, ist make auch in der Lage sich selbst (rekursiv) aufzurufen. Das Spezialzeichen - bei -rm $(OBJECTS) prog bewirkt, dass make bei einem rm-Fehler nicht abbricht. @ bei @echo ... bewirkt, dass das Kommando selbst nicht ausgegeben wird.

Wichtige Optionen für den Aufruf von make sind:

-f: angegebene Steuerdatei benutzen

-k: Bearbeitung trotz Fehler fortsetzen

-n: auszuführende Kommandos anzeigen jedoch nicht ausführen

-p: Makrodefinitionen anzeigen

-s: ausgeführte Kommandos nicht anzeigen

3.4 Versionsverwalter

Für größere Programmierprojekte, an denen mehrere Programmierer für längere Zeit arbeiten, ist es notwendig die Entwicklungsschritte zu protokollieren und verschiedene Versionen der entstehenden Software zu verwalten.

Zwei in UNIX gebräuchliche Systeme zur Verwaltung von Versionen sind RCS (Revision Control System aus BSD-UNIX) und SCCS (Source Code Control System aus AT&T-UNIX). Beide Systeme verwalten den Quellkode von Programmen in der Weise, dass von Version zu Version nur die Änderungen festgehalten werden. Somit lassen sich jederzeit alle Modifikationen und ihre Begründungen rekonstruieren. Die Versionsnummern werden automatisch erhöht, können aber auch bei Sprüngen manuell festgelegt werden.

Die Speicherung des Quellkodes erfolgt in spezifischen Dateien, die sich nicht mit einem herkömmlichen Editor bearbeiten lassen. In der folgenden Tabelle ist eine Auswahl von SCCS-Befehlen zusammengestellt:

SCCS-Befehl	Beschreibung
admin	erzeugt, initialisiert und verwaltet SCCS-Dateien
delta	schreibt geänderten Quellkode in eine SCCS-Datei
get	extrahiert eine Version aus einer SCCS-Datei
prs	liefert Entwicklungs-Informationen einer SCCS-Datei
rmdel	löscht eine Version aus einer SCCS-Datei
sccs	verwaltet alle SCCS-Kommandos
sccshelp	liefert Hilfe zu SCCS-Kommandos
what	identifiziert die Version eines Binärprogramms

Die wichtigsten Befehle von RCS können der folgenden Tabelle entnommen werden:

RCS-Befehl	Beschreibung
ci	schreibt geänderten Quellkode in RCS-Datei
co	extrahiert eine Version des Quellkodes aus RCS-Datei
ident	identifiziert die Version eines verwalteten Binärprogramms
rcs	generiert und verwaltet RCS-Dateien
rcsintro	liefert eine Einführung in RCS
rlog	liefert Informationen über die Entwicklung einer RCS-Datei

Die Arbeit mit RCS soll nun an einem Beispiel verdeutlicht werden. Ausgangspunkt ist das folgende C-Programm dreieck.c zur Berechnung des Umfangs eines Dreiecks:

```
#include <stdio.h>
char *rcsid = "$Id$";
int main(void) {
float a, b, c, p;
scanf("%f %f %f", &a, &b, &c);
p = a + b + c;
printf("%f\n",p);
}
```

Die globale Variable rcsid zeigt auf eine Zeichenkette, in die das RCS die aktuelle Versionsnummer einträgt. So kann später jederzeit mit ident die Version des Binärprogramms erfragt werden.

Zunächst wird ein Unterverzeichnis RCS für die RCS-Datei angelegt; die nachfolgenden Befehle verwenden dieses Unterverzeichnis automatisch. Anschließend kann mit dem Kommando ci eine RCS-Datei erzeugt und das C-Programm dreieck.c in diese übernommen werden.

```
mkdir RCS
ci dreieck.c
RCS/dreieck.c,v <-- dreieck.c enter description,
terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> berechnet den Umfang eines Dreiecks
>> .
initial revision: 1.1
done
```

Das ci-Kommando erfragt eine Beschreibung des Programms und weist als erste Versionsnummer 1.1 aus. Die RCS-Datei befindet sich im Unterverzeichnis RCS; der Dateiname endet auf v.

```
ls RCS
dreieck.c,v
```

Im nächsten Schritt soll eine neue Version der C-Quelle erstellt werden, die das Programm benutzerfreundlicher macht. Hierzu muss mit co der Quelltext extrahiert werden; die Option -l markiert in der RCS-Datei, dass eine neue Version erstellt wird und setzt bei dreieck.c das Schreibrecht.

co -l dreieck.c
RCS/dreieck.c,v –> dreieck.c
revision 1.1 (locked)
done

Nun wird die Quelldatei dreieck.c geändert, z.B. indem die printf()-Zeile wie folgt ersetzt wird:

printf("Umfang = %f", p);

Im Quelltext der extrahierten Quelle ist in der Zeile mit der Variablen rcsid die Eintragung der Version durch RCS erkennbar. ci erfragt nun die Beschreibung der Änderungen:

```
ci dreieck.c
RCS/dreieck.c,v <-- dreieck.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> verbesserte Benutzerschnittstelle
>> .
```

done

Um das Programm jetzt zu übersetzen, muss es mit co erneut extrahiert werden, wobei die Option -l nicht angegeben werden darf.

```
co dreieck.c
RCS/dreieck.c,v --> dreieck.c
revision 1.2
done
gcc dreieck.c
a.out
```

Mit dem Befehl ident kann nun auch die Version des Binärprogramms erfragt werden:

```
ident a.out
a.out:
$Id: dreieck.c,v 1.2 2001/02/18 19:46:58 kremer Exp $
```

Das rlog-Kommando zeigt die Beschreibung der RCS-Datei und der durchgeführten Änderungen:

```
rlog dreieck.c
RCS file: RCS/dreieck.c,v
Working file: dreieck.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2; selected revisions: 2
description:
berechnet den Umfang eines Dreiecks
----------------------------
revision 1.2
date: 2001/02/18 19:46:58; author: kremer; state: Exp; lines: +2 -2
verbesserte Benutzerschnittstelle
----------------------------
revision 1.1
date: 2001/02/18 19:36:27; author: kremer; state: Exp;
Initial revision
=============================================
```

Kapitel 4

Systemprogrammierung in UNIX

4.1 Einführung

Die Beziehung zwischen Anwendungsprogrammen und Betriebssystem ist auf unterschiedlichen Ebenen untergebracht. Die unterste Ebene bildet die Hardware: Prozessor, Arbeitsspeicher, Festplatten, Disketten, CD, DVD, Grafikkarten, Netzwerkkarten usw. Auf dieser Ebene werden nur einfache Kommandos ausgeführt, wie z.B. "Positioniere die Schreib-Leseköpfe der Festplatte auf Spur 81 und lese Sektor 0". Die Software, die auf dieser Ebene mit der Hardware zusammen arbeitet, ist i.d.R. nicht portabel, da sie Details der Hardware, wie z.B. das Registerlayout und den Befehlssatz eines Festplatten-Controllers, berücksichtigt. Programme auf dieser Ebene werden Gerätetreiber genannt.

Die nächst höhere Ebene bildet der Betriebssystemkern, der im Arbeitsspeicher residiert. Der Kern bedient sich der Dienste der Gerätetreiber und steuert und verwaltet den Rechner mit Hilfe von Dateisystem, Prozessverwaltung, Arbeitsspeicherverwaltung und Netzwerkdiensten.

Die Anwendungsprogramme greifen niemals direkt auf Gerätetreiber oder die Hardware zu, sondern sie verwenden die Funktionen des Betriebssystemkerns über eine Schnittstelle aus Systemaufrufen und Bibliotheksfunktionen. Der Unterschied in der Namensgebung zwischen Systemaufrufen und Bibliotheksfunktionen ist wie folgt begründet: Systemaufrufe werden ausschließlich im privilegierten Operationsmodus der CPU kontrolliert durch den Betriebssystemkern ausgeführt. Bibliotheksfunktionen werden zumindest teilweise im nicht privilegierten Benutzermodus der CPU ausgeführt. Bibliotheksfunktionen können ihrerseits Systemaufrufe benutzen.

Die Funtion fopen() ist beispielsweise eine Bibliotheksfunktion, die zum Öffnen einer Datei verwendet wird, um die Datei anschließend formatiert zu lesen und zu schreiben, z.B. mit fscanf() und fprintf()). fopen() liefert als Rückgabewert einen File Pointer: FILE *fopen(const char *path, const char *mode);

fopen() verwendet ihrerseits den Systemaufruf open() zum Öffnen einer Datei mit unformatiertem Lesen und Schreiben, z.B. mit read() und write(). open() liefert als Rückgabe einen Dateideskriptor: int open(const char *pathname, int flags);

Systemaufrufe und Bibliotheksfunktionen sind in unterschiedlichen Abschnitten der Manual Pages beschrieben: Systemaufrufe befinden sich im Abschnitt 2, Bibliotheksfunktionen in Abschnitt 3. Es kann sein, dass es z.B. die Funktion open() in unterschiedlichen Abschnitten gibt, z.B. zusätzlich als Tcl Built-In Kommando in Abschnitt n. In diesem Fall kann der Abschnitt durch den Aufruf man 2 open explizit angegeben werden.

Die Standardfunktionen und Systemaufrufe für C sind in der Bibliothek /usr/lib/libc.a untergebracht. Die Namen der Module kann man z.B. mit dem Bibliotheksprogramm ar anschauen: ar t libc.a | more. Will man zusätzliche Funktionen aufrufen, so muss man sie für den Bindevorgang spezifizieren z.B. beim Aufruf des Compiler-Treibers gcc:

```
gcc -o prog prog.c -lm
```

gcc beinhaltet den Aufruf des C-Compilers, des Assemblers und des Binders in einem. Deswegen kann man beim Aufruf von gcc Bindeoptionen angeben. Die Option -lm besagt, dass eine Bibliothek mit dem Namen libm.a, die mathematische Bibliothek für sin(), cos(), exp() usw. dazugebunden werden soll. Bei der Option -l gilt die Konvention, dass der vordere Bestandteil lib und der hintere .a zu dem angegebenen mittleren Teil hier m ergänzt wird.

Das Programm prog.c kann nun sämtliche mathematischen Funktionen aus libm.a verwenden. Es muss nur der Header math.h in die Quelle eingefügt werden, damit der Compiler über die Definition der Funktionen verfügt. Der Name des Maschinenprogramms wird hinter der -o Option angegeben. Wird die -o Option weggelassen, so erzeugt gcc standardmäßig die Datei a.out. Das Programm prog.c kann z.B. wie folgt aussehen:

```
#include <math.h>
int main(void) {
    printf("exp(%f) = %f", 1.0, exp(1.0));
}
```

4.2 Argumentübergabe und Basisfunktionen

Beim Aufruf können C-Programmen Argumente aus der Kommandozeile übergeben werden. Sie werden als Zeichenketten übergeben, d.h. wenn eine Zeichenkette z.B. als Zahl im Programm interpretiert werden soll, muss sie explizit z.B. mit atoi() oder atof() konvertiert werden. Technisch wird ein Feld von Zeigern auf die Zeichenketten aus der Kommandozeile in argv[] und die Anzahl der Zeichenketten in argc übergeben. Der Programmname mit Hilfe von argv[0] übergeben; argc ist also mindestens 1. Ein Programm, dass lediglich die Argumente aus der Kommandozeile ausgibt, sieht folgendermaßen aus:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i++)
      printf("argv[%d] = %s\n", i, argv[i]);
}
```

Neben den Argumenten aus der Kommandozeile werden bei jedem Prozess - also nicht nur bei C-Prozessen - Umgebungsvariablen vom Elternprozess vererbt (Environment). Diese Umgebungsvariablen sind auch als Zeichenketten in der Form Name=Wert gespeichert und können im C-Programm in ähnlicher Weise wie die Argumente gelesen werden. Das folgende Programm gibt lediglich alle Umgebungsvariablen mit ihren Werten aus:

```
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]) {
 while (*envp != NULL) printf("%s\n", *envp++);
}
```

Hierbei fällt auf, dass die Anzahl der Umgebungsvariablen nicht als Parameter zur Verfügung steht. Das Ende der Liste wird durch einen NULL-Zeiger symbolisiert. Der Zugriff auf die Umgebungsvariablen wird in dieser Art jedoch selten verwendet, da Variablename und Wert in einem Feldelement gemeinsam gespeichert sind, z.B. HOME=/home/kremer.

Typischer ist es, dass zu dem Namen einer Umgebungsvariablen, deren Wert gesucht wird. Natürlich kann man für diese Anwendung eine eigene Dekodierfunktion schreiben, was aber nicht notwendig ist, denn hierfür gibt es die Bibliotheksfunktion getenv():

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  char *path;
  char *vname = "PATH";
  path = getenv(vname);
  if (path != NULL) {
    printf("PATH=%s\n", path);
    exit(0);
  }
  else {
    fprintf(stderr, "PATH nicht im Environment.\n");
    exit(1);
  }
}
```

Natürlich gibt es auch Bibliotheksfunktionen, um Umgebungsvariablen hinzuzufügen und zu ändern putenv(), setenv() sowie zu löschen unsetenv().

Eine weitere Basisfunktion chdir() bewirkt das Ändern des aktuellen Verzeichnisses. Dies entspricht dem Kommando cd auf der Shell-Ebene. Das aktuelle Arbeitsverzeichnis wird in der Umgebungsvariablen PWD gespeichert und kann auf der Shell-Ebene mit dem Kommando pwd ausgegeben werden. Mit Hilfe der Funktion getcwd() kann eine Kopie dieser Variablen im C-Programm angelegt werden, um dann beispielsweise in ein Unterverzeichnis mit chdir() zu verzweigen:

```
#include <stdio.h>
#include <sys/param.h>
#include <unistd.h>
```

```
int main(int argc, char *argv[], char *envp[]) {

  char dir[MAXPATHLEN], subdir[MAXPATHLEN];

  if (getcwd(dir, MAXPATHLEN) == NULL) {
     perror("getcwd error");
     exit(1);
  }
  else
    printf("current working directory: %s\n", dir);

  printf("change directory to subdirectory: ");
  scanf("%s", subdir);
  strcat(dir, subdir);

  if (chdir(subdir) == -1) perror("bad directory");

  if (getcwd(dir, MAXPATHLEN) == NULL) {
     perror("getcwd error");
     exit(1);
  }
  else
    printf("current working directory: %s\n", dir);
}
```

Eine mächtige Bibliotheksfunktion ist system(), mit der jedes Kommando ausgeführt werden kann. Dazu ruft system() die Standardshell /bin/sh auf und übergibt die Zeichenkette des aufgerufenen Kommandos ggfs. mit Parametern an diese Shell. Die Shell führt das Kommando, falls es gefunden wurde und für den aktuellen Benutzer ausführbar ist, aus und gibt als Rückgabewert den Returncode des Kommandos zurück. Es folgt ein Programm zur Ausführung eines beliebigen eingelesenen Kommandos:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int rc;
  char command[1024];

  printf("Kommando eingeben: ");
  scanf("%s", command);
  rc=system(command);
  if (rc != 0)
    printf("\nKommando fehlgeschlaten rc = %d\n", rc);
}
```

4.3 Prozess-Handhabung

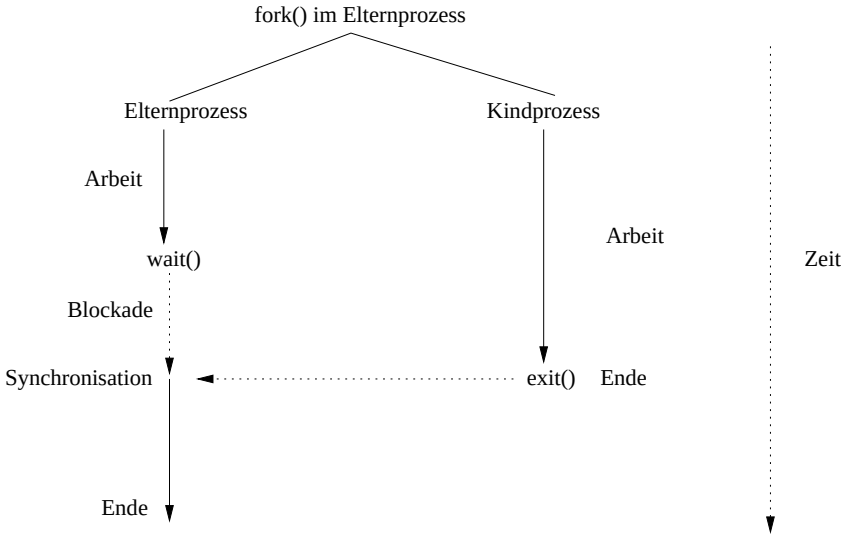
Die Einfachheit der Erzeugung eines neuen Prozesses ist eine der Stärken in der Unix-Welt: Jeder Prozess kann mit Hilfe des Systemaufrufs fork() einen neuen

Prozess erzeugen. Dabei sind Prozesse durch eine systemweit eindeutige Identifikationsnummer, die Prozess Id oder kurz PID, gekennzeichnet. Außerdem wird die PID des Elternprozesses, die Parent PID oder kurz PPID, vom Betriebssystem im Prozess-Kontrollblock vermerkt. Die Kommandos ps f oder ps l liefern z.B. eine Übersicht über die Prozesse des aktuellen Benutzers mit PID und PPID.

Beim Aufruf von fork() entsteht ein neuer Prozess, der ein identisches Abbild des erzeugenden Elternprozesses ist. Damit die Prozesse nun unterschiedliche Kontrollflüsse haben können, gibt fork() im Elternprozess die PID des Kindprozesses zurück und im Kindprozess immer 0. Um die PPID zu ermitteln, dient dem Kindprozess die Funktion getppid(). Der folgende Programmausschnitt zeigt dieses Verhalten:

```
int pid;
if ((pid = fork()) ! = 0) {
  printf("Elternprozess erzeugte PID %d\n", pid);
  fflush(stdout);
}
else {
  printf("Kindprozess von PPID %d erzeugt\n",
         getppid());
  fflush(stdout);
}
```

Das Prozessmodell von Unix sieht vor, dass der Elternprozess seine Kindprozesse überwachen kann. Dies heißt auch, dass der Kindprozess (im Gegensatz zur Natur) i.d.R. vor dem Elternprozess beendet werden muss. Der Return Code des Kindprozesses, der mit exit() bestimmt wird, muss (kann) mit dem Systemaufruf wait() vom Elternprozess ausgewertet werden. Das Schaubild zeigt dieses Verhalten:




```
#include <stdio.h>
#include <stdlib.h>
```

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

#define COUNT 10

int main(void) {

int i,pid,exit_status;

  if ((pid=fork())!=0) { /* Elternprozess */

    /* Arbeit des Elternteils */
    printf("Elternprozess erzeugte Kindprozess %d\n",
           pid);
    fflush(stdout);


    for (i=0; i < COUNT; i++)
       printf("Elternteil %d\n", i);

    /* Warten auf das Ende des Kindteils */
    if (wait(&exit_status)==pid)
      printf("Kindprozess mit %d beendet\n",
              WEXITSTATUS(exit_status));
  }
  else { /* Kindprozess */

    /* Arbeit des Kindteils */
    printf("Kindprozess von PPID %d erzeugt\n",
            getppid());
    fflush(stdout);
    for (i=0; i < COUNT; i++)
      printf("Kindteil %d\n", i);
    /* Ende des Kindteils */
    exit(0);
  }
}
```

Natürlich kann der oben gezeigte zeitliche Ablauf von Eltern- und Kindprozess nicht erzwungen werden, da es sich um zwei voneinander unabhängige Prozesse handelt. Es können zwei unerwünschte Situationen auftreten:

- Der Elternprozess endet vor dem Kindprozess.
- Der wait-Aufruf im Elternprozess erfolgt erst nach dem Ende des Kindprozesses oder der Elternprozess enthält keinen wait()-Aufruf, um den exit-Kode des Kindprozesses auszuwerten.

Im ersten Fall muss der Kindprozess einen anderen kontrollierenden Elternprozess bekommen. Er wird sozusagen adoptiert. Der Adoptiv-Elternprozess ist der Prozessverwalter des Betriebssystems mit dem Namen init. Mit dem Befehl ps

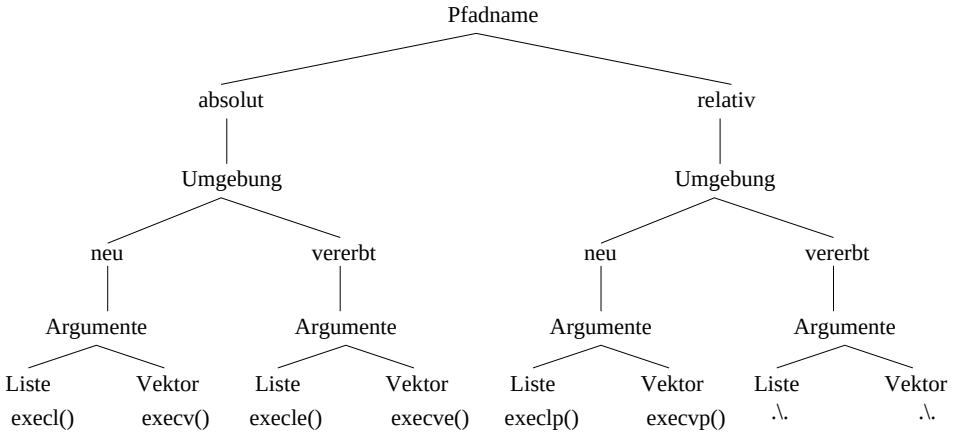
alx kann man sehen, dass der init-Prozess die PID 1 hat und der Kindprozess als Elternprozess die PPID 1 trägt. Wenn der Kindprozess nun endet führt der init-Prozess einen wait()-Aufruf ihn aus.

Im zweiten Fall könnte der Kindprozess eigentlich seine Betriebsmittel freigeben. Bis auf den Eintrag in der Prozesstabelle geschieht das auch, denn die Prozesstabelle enthält den exit-Kode des Kindprozesses, der ja noch an den Elternprozess übermittelt werden soll. Aus dem Kindprozess ist ein "untoter" Zombie-Prozess geworden. Der ps-Befehl liefert tatsächlich einen Zombie-Zustand für den Prozess. Erst wenn der Elternprozess den wait()-Aufruf macht oder wenn der Elternprozess endet (dann ist der Kindprozess wieder verwaist s.o.), verschwindet der Zombie-Prozess aus der Prozesstabelle.

Während der erste Fall unproblematisch für das Betriebssystem ist, kann der zweite Fall die Arbeit auf dem Rechner stark behindern: Die Prozesstabelle hat eine feste Höchstzahl von Einträgen für Prozesse. Wenn diese Höchstzahl erreicht ist, können keine weiteren Prozesse mehr erzeugt werden. Die Prozesstabelle kann also mit Zombie-Einträgen verstopft werden. Dies kann z.B. bei einem schlecht programmierten Serverprozess passieren, der für jede Client-Anfrage einen Kindprozess erzeugt aber keinen wait()-Aufruf für den Kindprozess enthält.

In dieser Situation bleibt nichts anderes übrig als den Serverprozess mit dem kill-Kommando zu beenden. Für das kill-Kommando wird kein neuer Prozess benötigt, da es Teil des Betriebssystemkerns ist.

Häufig dient der Kindprozess dazu ein anderes Programm auszuführen als der Elternprozess, das aber nur als ausführbare Datei und nicht im Quellkode vorhanden ist. In diesem Fall kann der erzeugte Kindprozess mit Hilfe einer Bibliotheksfunktion aus der exec-Gruppe durch ein neues Prozessabbild überlagert werden. Die Funktionen der exec-Gruppe unterscheiden sich in der Behandlung des Programmpfads in der Übergabeart von Parametern und in der Behandlung der Umgebungsvariablen: Der Pfad kann absolut oder relativ zum Suchpfad PATH angegeben werden. Die Parameter können als Liste (execl(), execle(), execlp()) oder als Vektor (execv(), execve(), execvp()) übergeben werden. Die Umgebung kann als Kopie vom Elternprozess vererbt werden (execl(), execlp(), execv(), execvp()) oder sie kann vom Elternprozess explizit neu erzeugt und übergeben werden (execle(), execve()). Es ergibt sich also folgende Unterteilung:



Die Syntax der Funktionen der exec-Gruppe lautet:

```
int execl(const char *path, const char *arg0, ...,
          NULL)
int execv(const char *path, const char *argv[])
int execle(const char *path, const char *arg0, ...,
           NULL, char * const envp[])
int execve(const char *path, const char *argv[],
           char * const envp[])
int execlp(const char *file, const char *arg0, ...,
           NULL)
int execvp(const char *file, const char *argv[])
```

Die letzten Elemente von argv[] und envp[] müssen beim Aufruf als Endekennzeichen jeweils den NULL-Zeiger enthalten.

4.4 Signale und deren Behandlung

Ein Signal ist die Mitteilung an einen Prozess, dass ein bestimmtes externes Ereignis eingetreten ist. Ein solches Ereignis tritt asynchron zum Programmlauf auf, z.B. wenn der Anwender control c drückt oder wenn ein Kindprozess endet. Wenn ein Signal auftritt, muss der betroffene Prozess sofort darauf reagieren, d.h. seine Arbeit wird unterbrochen. Deswegen spricht man auch von einem Interrupt. Der Prozess kann nun auf ein aufgetretenes Signal verschieden reagieren:

- Der Prozess kann das Signal einfach ignorieren und weiterarbeiten (außer SIGKILL s.u.).
- Der Prozess kann eine Signal-Behandlungsfunktion (Signal-Handler, Interrupt-Handler) ausführen und anschließend dort fortfahren, wo er unterbrochen wurde.
- Der Prozess kann auf das Signal mit jeder voreingestellten Aktion (default action) reagieren, dies bewirkt häufig den Abbruch des Prozesses.

Signale werden zwischen dem Betriebssystemkern und Prozessen sowie zwischen verschiedenen Anwenderprozessen übermittelt. Damit Signale zwischen Prozessen ausgetauscht werden können, muss der sendende Prozess dieselbe effektive User-ID wie der empfangende Prozess besitzen oder der sendende Prozess muss die effektive User-ID 0 des Superusers besitzen.

Auf der Shell-Ebene hat das Kommando zur Übermittlung von Signalen die Syntax:

```
kill -<Signal> <PID>
```

Als Systemaufruf von C aus ist die Syntax:

```
int kill(<PID>, <Signal>)
```

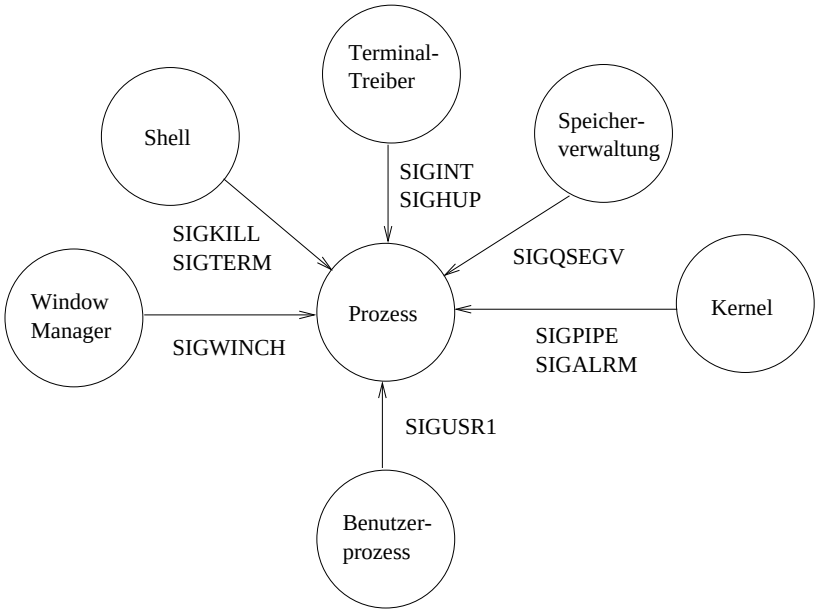
Gibt kill 0 zurück, so ist der Aufruf erfolgreich gewesen, ansonsten wird -1 zurückgegeben. In errno steht dann der Fehlerkode.

Es gibt etwas über 30 verschiedene Signale, die über die Signalnummer oder den zugeordneten Signalnamen bezeichnet werden können. Mit dem Befehl kill -l kann von der Shell aus die Zuordnung zwischen Signalnummer und Signalname eingesehen werden. (Alternativ kann die Header-Datei signal.h konsultiert werden.)

Einige wichtige Signale mit ihren Ursachen und ihren Default-Aktionen sind in der folgenden Tabelle zusammengestellt:

Nr	Signalname	Default-Aktion	Ursache
1	SIGHUP	Abbruch	Fenster schließen
2	SIGINT	Abbruch	Unterbrechung control c
3	SIGQUIT	Abbruch	Unterbrechung ctrl \
9	SIGKILL	unbedingter Abbruch	kill -9 <PID>
11	SIGSEGV	Speicherabzug und Abbruch	unberechtigter Speicherzugriff
13	SIGPIPE	Abbruch	Schreiben auf nicht gelesene Pipe
14	SIGALRM	Abbruch	Timer Interrupt (alarm())
15	SIGTERM	Abbruch	kill <PID>
17	SIGCHLD	Ignorieren	Ende eines Kindprozesses
28	SIGWINCH	Ignorieren	Fenstergröße geändert

Es können zwar beliebige Signale zwischen Prozessen ausgetauscht werden, jedoch senden bestimmte Prozesse, wie der Betriebssystemkern (Kernel), die Shell, der Window Manager u.s.w. meist bestimmte festgelegte Signale, wie das folgende Bild veranschaulicht:



Um die Reaktion auf ein Signal zu definieren gibt es den Systemaufruf signal(). Hierbei kann zu einer vorgegebenen Signalnummer ein Signal-Handler installiert werden:

void signal(int signum, void *handler)

Der Parameter handler ist die Funktion, die beim Interrupt ausgeführt wird oder eine der Konstanten SIG_IGN oder SIG_DFL. Bei Setzung von SIG_IGN wird das Signal ignoriert, bei Setzung von SIG_DFL wird die Default-Aktion zum Signal wieder eingestellt. Man kann pro zu behandelndem Signal einen Handler definieren oder auch einen Handler für mehrere Signale. Dies ist möglich, da die Signalnummer dem Handler implizit übergeben wird. Das folgende Programmbeispiel zeigt einen Prozess, der die Signale SIGINT und SIGQUIT behandelt. Hierbei stellt das Senden von SIGQUIT die Default-Aktion für SIGQUIT wieder her; das Senden von SIGINT bewirkt das nachfolgende SIGQUIT-Aufrufe ignoriert werden:

```
#include <stdio.h>
#include <sys/signal.h>

void catcher(int sigtype) {
        printf("Signal %d abgefangen\n", sigtype);
        if (sigtype == SIGQUIT)
                signal(SIGQUIT, SIG_DFL);
        else if (sigtype == SIGINT)
                signal(SIGQUIT, SIG_IGN);
}

int main(void) {

        int i=0;

        signal(SIGQUIT, catcher);
        signal(SIGINT, catcher);
        while (1) {
                printf("working ... %d\n", i++);
                sleep(1);
        }

}
```

Bei der Signalbehandlung hat das Signal SIGKILL (9) eine Sonderrolle: Es kann nämlich nicht von einem Signal-Handler abgefangen werden und führt immer zum Abbruch des betreffenden Prozesses. Dies hat seinen guten Grund, denn durch das Ignorieren aller Signale oder das Einführen von Signal-Handlern für alle Signale wäre ein systemschädlicher Prozess nicht mehr zu beenden. Damit das nicht passiert, kann der Systemadministrator oder der Benutzer des Prozesses ihn immer mit kill -SIGKILL <PID> beenden.

Vor dem leichtfertigen Umgang mit kill -9 oder kill -SIGKILL möchte ich aber hier ausdrücklich warnen. Die meisten Prozesse, wie Editoren, Internet-Dämone Datenbankserver-Prozesse beinhalten eine Endebehandlung, wenn sie mit kill <PID> beendet werden. Bei dieser Endebehandlung wird z.B. die Datenbank auf einen konsistenten Zustand gebracht. Beendet man hingegen z.B. den Datenbankserver mit kill -SIGKILL, so wird er sofort, ohne Aufsuchen eines Handlers beendet. Dabei können also inkonsistente Zustände der Datenbank

entstehen. Versuchen Sie deshalb immer erst einen Prozess mit kill <PID> oder anderen Signalen zu beenden, bevor Sie ihn gewaltsam mit kill -SIGKILL beenden.

4.5 Pipe-Kommunikation

Bisher haben wir kennengelernt, wie man neue Prozesse erzeugen kann und wie Signale auf sie wirken. Wir lernen nun kennen, wie Prozesse innerhalb eines Rechners miteinander kommunizieren können. Eine andere Angelegenheit ist die Kommunikation von Prozessen zwischen verschiedenen Rechnern (Rechnernetzwerke).

Ein sehr häufig angewendeter Mechanismus ist die Kommunikation von Prozessen über eine Pipe. Einem Prozess stehen als Kommunikationswege standardmäßig die Standardeingabe (Dateideskriptor 0), die Standardausgabe (Dateideskriptor 1) und die Standardfehlerausgabe (Dateideskriptor 2) zur Verfügung. In der Shell können diese Deskriptoren auch in oder von Dateien umgeleitet werden. Will man beispielsweise eine sortierte Liste aller angemeldeten Benutzer eines Systems erhalten, so könnte man das über eine Zwischendatei wie folgt realisieren:

```
who > temp
sort < temp
```

Schneller und einfacher geht es aber von der Shell aus mit einer Pipe. Hierbei wird die Standardausgabe von who mit Standardeingabe von sort verbunden. Die Pipe wird intern nicht in einer Datei sondern im Arbeitsspeicher realisiert:

```
who | sort
```

Allerdings wissen wir bisher nicht, wie wir eine Pipe-Kommunikation ohne Zuhilfenehmen der Shell programmieren können. Das wollen wir nun besprechen.

Pipes sind unidirektionale Kommunikationsmechanismen, die über einen Puffer im Arbeitsspeicher realisiert sind. Sie haben ein Ende auf das geschrieben und ein anderes Ende von dem gelesen werden kann. Die beteiligten lesenden und schreibenden Prozesse sind gleichzeitig aktiv und kommunizieren über die Pipe miteinander. Zwei Kindprozesse eines gemeinsamen Elternprozesses können nach folgendem Schema miteinander kommunizieren:

- Prozess A legt eine Pipe an und erhält für beide Enden der Pipe Dateideskriptoren durch den Systemaufruf pipe() zurück:

  ```
  int fds[2];
  if (pipe(fds) == -1) perror("Anlegen der Pipe");
  ```

- Prozess A erzeugt Kindprozess B. B erbt beide Dateideskriptoren der Pipe und schließt den Schreibdeskriptor fds[1]:

  ```
  if (fork() == 0) { /* lesender Kindprozess sort */
    dup2(fds[0], 0);
    close(fds[1]);
    execlp("sort","sort", 0);
  }
  ```

- Der Aufruf dup2(fds[0], 0) bewirkt hierbei, dass die Standardeingabe mit dem Dateideskriptor 0 auf fds[0] also auf das lesende Ende der Pipe umgeleitet wird. Prozess B bezieht seine Eingaben also aus der Pipe und nicht von der Tastatur.

- Prozess A erzeugt Kindprozess C. C erbt ebenfalls beide Dateideskriptoren der Pipe und schließt den Lesedeskriptor fds[0]:

```
else /* nur in Prozess A nicht bei B fork() */
if (fork() == 0) {
  dup2(fds[1],1);
  close(fds[0]);
  execlp("who","who", 0);
}
```

- Auch hier wird dup2() eingesetzt, diesmal, um die Standardausgabe in das schreibende Ende der Pipe umzuleiten.

- Der Elternprozess A schließt beide Enden der Pipe und wartet auf das Ende seiner Kindprozesse B und C:

```
else { /* nur in Prozess A ausgeführt */
  close(fd[0]);
  close(fd[1]);
  wait(0);
  wait(0);
}
```

Anmerkung 1: Es ist notwendig den Schreibdeskriptor der Pipe im lesenden Prozess und im Elternprozess zu schließen, sonst erhält der lesende Prozess nie eine EOF (End-of-File) Meldung der Pipe. Es wird nämlich erst EOF gesendet, wenn alle Prozesse, die an die Pipe angeschlossen sind, ihren Schreibdeskriptor geschlossen haben. Den Lesedeskriptor im schreibenden Prozess und im Elternprozess zu schließen ist hingegen nur guter Stil.

Anmerkung 2: Die wait()-Aufrufe stellen sicher, dass der Elternprozess A nicht beendet wird, bevor seine Kinder B und C enden. Sonst könnte die Shell ihren Prompt ausgeben, bevor die Ausgabe von Prozess B beendet ist.

Die beiden Prozesse B und C mussten in unserem Beispiel keine Maßnahmen ergreifen, um sich gegenseitig zu synchronisieren. Der Pipe-Mechanismus beinhaltet eine implizite Synchronisation, die in diesem Beispiel schlecht zur Geltung kommt. Deswegen schauen wir uns ein anderes Beispiel an:

Nehmen wir an, der schreibende Prozess hat 230 Byte in die Pipe gestellt. Der lesende Prozess enthält aber nur einen Puffer für 100 Byte und arbeitet daher in folgender Schleife:

```
while ((count = read(fd, buffer, 100)) > 0) {
      /* Daten in buffer bearbeiten */
}
```

In den ersten beiden Schleifendurchläufen werden 100 Byte eingelesen und bearbeitet. Im dritten Schleifendurchlauf werden 30 Byte eingelesen und bearbeitet. Der vierte Durchlauf blockiert den Leseprozess beim read()-Aufruf, bis neue Daten gesendet werden oder bis der Schreibdeskriptor der Pipe geschlossen wird.

Hier wird wieder klar, warum der Schreibdeskriptor der Pipe im Elternprozess und im lesenden Kindprozess unbedingt geschlossen werden muss: Solange irgendein Prozess einen offenen Schreibdeskriptor der Pipe besitzt, wird keine EOF-Meldung an den Lesedeskriptor übermittelt. read() liefert dann nicht 0 zurück, sondern blockiert den Leseprozess (u.U. für immer).

Die Synchronisation der über eine Pipe kommunizierenden Prozesse verläuft also nach folgendem Schema:

1. Da Pipes nur eine begrenzte Speicherkapazität haben (einige KByte), blockiert der schreibende Prozess, wenn er eine volle Pipe vorfindet, bis der lesende Prozess Daten abgerufen hat.

2. Der lesende Prozess blockiert, wenn keine Daten mehr in der Pipe anstehen, bis der schreibende Prozess neue Daten in die Pipe stellt.

4.6 FIFO-Kommunikation

FIFO's werden auch häufig Named Pipes genannt. Die Bezeichnung FIFO (First In First Out) beschreibt die Tatsache, dass die Reihenfolge der Eingaben in der Ausgabe der Named Pipe erhalten bleibt.

FIFO's stehen unter System V Unix immer zur Verfügung und sind als Dateien realisiert. Über den Namen einer FIFO können beliebige Prozesse eines Rechners - nicht nur Kindprozesse eines gemeinsamen Vorfahren - miteinander kommunizieren.

Eine FIFO wird mit dem Systemaufruf mknod() erzeugt. mknod() dient dem Systemverwalter auch dazu Geräteeinträge im Verzeichnis /dev zu machen. FIFO's können aber mit mknod() auch mit Benutzerrechten angelegt werden. Die Syntax von mknod() ist:

```
int mknod(const char *pathname, mode_t mode,
          dev_t dev);
```

pathname: Pfad und Name der FIFO

mode: Zugriffsrechte, die mit der Konstanten S_IFIFO aus <sys/stat.h> durch logisches oder verknüpft werden. Hierdurch weiß das Betriebssystem, dass eine FIFO angelegt werden soll. Hierbei werden die mit umask vereinbarten Rechte berücksichtigt.

dev: dev gibt die Kennzahl der Device an (Device Number), dev wird bei FIFO's auf 0 gesetzt.

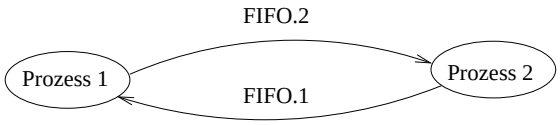
Als Beispiel folgt ein Aufruf zum Erzeugen einer FIFO:

```
if (mknod("/tmp/FIFO.1", 0644|S_IFIFO, 0) == -1)
   perror("cannot create FIFO /tmp/FIFO.1");
```

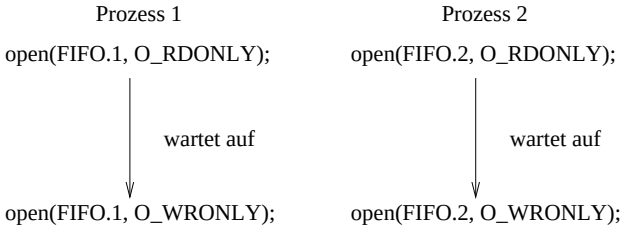
Geöffnet wird eine FIFO mit dem Systemaufruf open(). Dabei öffnet der Schreiber die FIFO mit dem Flag O_WRONLY und der Leser mit dem Flag O_RDONLY.

Schreibt man zwei Programme, die über eine FIFO miteinander verbunden sind, so blockiert der open()-Aufruf des Schreibers bis der Leser ebenfalls ein open() durchgeführt hat und auch umgekehrt. Das Öffnen der FIFO ist also ein Synchronisationspunkt beider Prozesse.

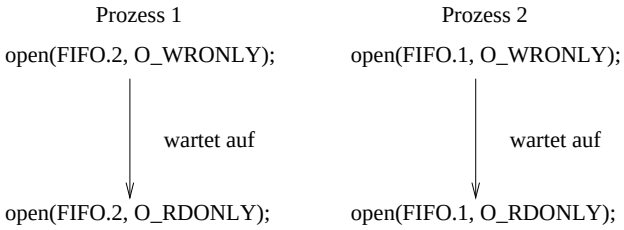
Wenn man eine bidirektionale Kommunikation zwischen zwei Prozessen erreichen will, muss man zwei FIFO's zwischen den beteiligten Prozessen einrichten. Hierbei kann es bei einer ungünstigen Reihenfolge der open()-Aufrufe zu einer Verklemmung (Deadlock) der beteiligten Prozesse kommen. Das folgende Bild beschreibt diese Situation:



1. Deadlock-Situation

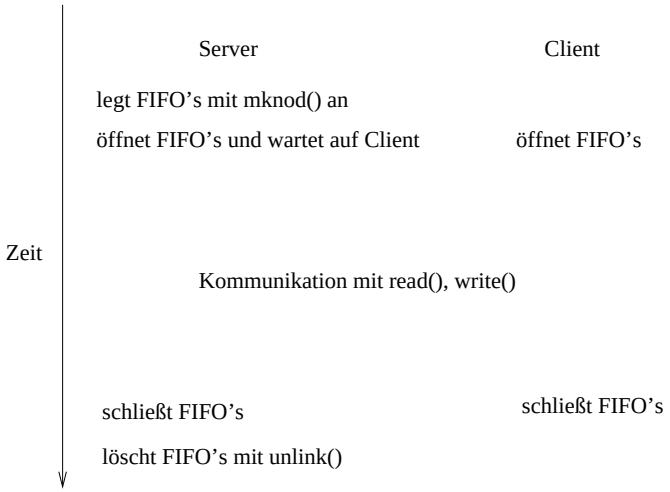


2. Deadlock-Situation



Eine solche Verklemmung kann nur verhindert werden, wenn die open()-Operationen für Lesen und Schreiben einer FIFO in beiden Prozessen jeweils zueinander passend aufgerufen werden. Z.B. erfolgt zunächst das open() von FIFO.1 und dann von FIFO.2 in beiden Prozessen.

Mit FIFO's lässt sich auf einfache Weise ein Client-Prozess mit einem Server-Prozess auf einem Rechner verbinden:



4.7 Gemeinsamer Speicher und gemeinsame Objekte

Die Speicherverwaltungs-Software des Betriebssystemkerns vermittelt jedem Prozess über das Demand-Paging-Verfahren virtuellen Speicher. Gemeinsam benutzter Speicher ist über den virtuellen Speicher realisiert und stellt eine effektive Form der Kommunikation zwischen einer Gruppe von Prozessen her. Der Rechner kann hierbei auch als Mehrprozessor-Rechner ausgelegt sein, wodurch eine echte Parallelarbeit mit Kommunikation über den gemeinsamen Speicher möglich ist (SMP-Systeme: Symmetrischer Multiprozessor).

Das Verfahren zur Verwaltung des gemeinsamen Speichers verläuft folgendermaßen:

- Irgendein Prozess fordert anfangs ein gemeinsam benutztes Speichersegment an. Dieser Prozess bestimmt die Größe und bekommt für das Segment einen numerischen Schlüssel vom Betriebssystem zugeteilt.
- Irgendwann später kann sich ein weiterer Prozess entscheiden, das Segment in seinen Adressraum einzuhängen. Dazu muss er nur den numerischen Schlüssel des Segments kennen.
- Ein gemeinsam genutztes Speichersegment ist analog zu Dateien unabhängig von Prozessen: Das Segment bleibt auch im System, wenn es an keinen Prozess mehr angehängt ist. Ferner haben gemeinsam genutzte Speichersegmente Zugangsberechtigungen und Modifikationszeit-Einträge wie Dateien.
- Mit dem Kommando ipcs -m kann die Liste der gemeinsam benutzter Speichersegmente abgerufen werden.

Im Einzelnen sehen die Systemaufrufe zur Verwaltung des gemeinsamen Speichers wie folgt aus:

```
int shmget(key_t key, int size, int shmflg)
```

shmget() legt ein neues Speichersegment an oder ergreift Besitz von einem bestehenden Segment. Hierbei haben die Parameter folgende Bedeutungen:

key: Schlüssel zur Identifikation des Segments

size: Anzahl der Bytes des Segments

shmflg: IPC_CREAT | 0644 beim Anlegen eines Segments oder 0 beim Zugriff auf ein bereits existierendes Segment

Rückgabewert von shmget() ist die Kennung des angelegten oder existierenden Segments oder -1 bei Misserfolg.

```
void *shmat (int shmid, const void *shmaddr,
             int shmflg)
```

shmat() hängt ein gemeinsam genutztes Speichersegment in den Prozess ein. Die Parameter haben folgende Bedeutung:

shmid: Kennung des Segments die von shmget() zurückgeliefert wurde

shmaddr: Adresse für die Einblendung des Speichersegments; normalerweise 0, wenn das System selber wählen soll

shmflg: z.B. SHM_RDONLY zum lesenden Einblenden; normalerweise 0, wenn mit den Rechten des Speichersegments eingeblendet werden soll

Der Rückgabewert von shmat() ist ein Zeiger auf das gemeinsam benutzte Speichersegment.

Ein einfaches Beispiel für gemeinsam genutzten Speicher liefern zwei Programme mit folgender Spezifikation:

Das Programm pline ist als Hintergrundprozess gedacht, der alle 4 Sekunden eine Zeichenkette ausgibt, die in einem gemeinsam genutzten Speichersegment untergebracht ist.

Das Programm cline ist als Vordergrundprozess gedacht, der die Informationen über die auszugebende Zeichenkette in dem gemeinsam genutzten Speichersegment ändert.

Die Datenstruktur für den gemeinsam genutzten Speicher wird in der Header-Datei line.h untergebracht:

```
struct info {
        char c;
        int length;
};

#define KEY             ((key_t) (1234))
#define SEGSIZE         sizeof(struct info)

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

Das Programm pline ist in pline.c wie folgt implementiert:

```
/* Beispiel fuer gemeinsamen Speicher mehrerer
   Prozesse:
   pline gibt alle 4 Sekunden eine Zeile aus.
   Die Zeilenlaenge und der Zeileninhalt wird von
   anderen Prozessen gefuellt.
*/

#include "line.h"

main() {

int i, id;
struct info *ctrl;
struct shmid_ds shmbuf;

id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
if (id <0) {
        perror("pline: shmget failed");
        exit(1);
}
printf("shmid %d\n", id);

/* Shared Memory atach */

ctrl = (struct info *) shmat(id,0,0);

if (ctrl <= (struct info *)(0)) {
        perror("pline: shmat failed:");
        exit(2);
}

/* set default parameters */

ctrl->c      = 'a';
ctrl->length = 10;

/* main loop */
while (ctrl->length > 0) {
        for (i = 0; i < ctrl->length; i++)
           putchar(ctrl->c);
        putchar('\n');
        sleep(4);
}

/* Shared Memory detach */

if (shmdt(ctrl) < 0) {
        perror("pline: shmdt failed");
```

```
        exit(3);
}

/* Shared Memory zurueckgeben (loeschen) */

if (shmctl(id, IPC_RMID, &shmbuf) < 0) {
        perror("pline: shmctl IPC_RMID failed");
        exit(4);
}
exit (0);
}
```

Es muss noch erklärt werden, was die Systemaufrufe shmdt(ctrl) und shmctl(id, IPC_RMID, &shmbuf) bewirken:

- shmdt() entfernt den gemeinsam genutzten Speicher aus dem Adressraum des aufrufenden Prozesses.
- Mit shmctl() kann das Speichersegment von berechtigten Benutzern verwaltet werden, z.B. IPC_RMID zum Löschen des Segments oder SHM_LOCK um Swapping eines Segments zu verhindern.

Das Programm cline zum Ändern der Zeichenkette ist in cline.c wie folgt implementiert:

```
/* Beispiel fuer gemeinsamen Speicher mehrerer
   Prozesse:
   cline arbeitet zusammen mit pline.
   cline erlaubt die Einstellung der Werte fuer
   Zeilenlaenge und -inhalt im
   gemeinsamem Speicherelement.
*/

#include "line.h"

main(int argc, char *argv[]) {

int i, id;
struct info *ctrl;

if (argc != 3) {
   fprintf(stderr, "usage: cline <char> <length>\n");
   exit(1);
}
id = shmget(KEY, SEGSIZE, 0);
if (id <0) {
        perror("cline: shmget failed");
        exit(2);
}

ctrl = (struct info *) shmat(id,0,0);
```

```
if (ctrl <= (struct info *)(0)) {
        perror("cline: shmat failed:");
        exit(3);
}

/* copy command line data to shared memory */

ctrl->c      = argv[1][0];
ctrl->length = atoi(argv[2]);

/* Shared Memory detach */

if (shmdt(ctrl) < 0) {
        perror("cline: shmdt failed");
        exit(4);
}

exit (0);
}
```

Eine von BSD-Unix her stammende Alternative für gemeinsam genutzten Speicher sind die Memory Mapped Objects, die mit dem Systemaufruf mmap() verwaltet werden. Mit mmap() kann ein Objekt, z.B. eine Datei oder auch ein Gerätetreiber, in mehrere Prozesse eingebunden werden kann. Die Syntax von mmap() lautet wie folgt:

```
void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
```

Hierbei bedeuten die Parameter:

start: Adresse für die Einblendung des Objekts; normalerweise 0, wenn das System wählen soll

length: Größe der Regin in Bytes

prot: Zugriffsbedingungen, die der Prozess einhalten muss, z.B. PROT_READ, PROT_READ | PROT_WRITE, PROT_EXEC, PROT_NONE

flags: Bestimmung des Objekts für den privaten Zugriff durch einen Prozess mit MAP_PRIVATE oder den gemeinsamen Zugriff durch mehrere Prozesse mit MAP_SHARED

fd: Dateideskriptor des einzublendenen Objekts

offset: Stelle innerhalb des Objekts vom Anfang an, ab der eingeblendet werden soll

Ein typischer Aufruf von mmap() lautet z.B.:

ptr = mmap(0, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)

Die Unterschiede zwischen gemeinsamen Speichersegmenten (shmget()) und gemeinsamen Objekten (mmap()) sind durch ihre Realisierung als Arbeitsspeicher bzw. als Dateien begründet:

mmap() benutzt einen Pfadnamen und Systemaufrufe, die auch bei Dateien verwendet werden, wie z.B. open(), unlink(), chown() und chmod(). Gemeinsame Objekte sind bei mmap() nicht flüchtig, d.h. sie sind i.d.R. nach einem Systemabsturz (z.B. durch Stromausfall) erhalten. Zur schnelleren Arbeitsweise werden Operationen zunächst in einem Puffer im Arbeitsspeicher durchgeführt, bevor sie explizit mit dem Systemaufruf msync() auf die Datei zurückgeschrieben werden. Einige Implementationen von Unix erlauben mmap()-Objekte über verteilte Dateisysteme, wie NFS. Hiermit werden solche gemeinsamen Objekte zu einem Kommunikations-Mechanismus, der aber gegenüber anderen gebräuchlichen Mechanismen, wie TCP/IP-Sockets und RPC's langsam ist.

shmget() reserviert Arbeitsspeicher und nutzt andere Systemaufrufe zur Verwaltung des gemeinsamen Speichers z.B. shmctl(). Gemeinsamer Speicher ist flüchtig, da er ausschließlich im Arbeitsspeicher gehalten wird.

4.8 Kritische Programmabschnitte und Semaphoren

Die Programme cline und pline synchronisieren ihre Zugriffe auf den gemeinsam genutzten Speicher nicht. Dies kann zu Fehlern durch Inkonsistenzen des Speichers führen:

Normalerweise wird cline in der Schafphase von pline den gemeinsamen Speicher ändern. Dann läuft alles richtig. Nehmen wir aber an pline ist gestartet und durchläuft gerade die Schleife, um z.B. 10 a's auszugeben. Nun wird cline t 50 aufgerufen. cline führt den Befehl ctrl->c=argv[1][0] aus. Zu diesem Zeitpunkt ist ctrl->length = 10. Es werden von pline dann 10 t's anstatt der gewünschten 50 t's ausgegeben. Das System ist inkonsistent.

Diese Inkonsistenz rührt daher, dass beide Prozesse nicht synchronisiert auf den gemeinsamen Speicher zugreifen. Folgende Bedingungen müssen erfüllt sein, um Konsistenz zu wahren:

Der pline-Prozess darf die Daten im gemeinsamen Speicher solange nicht lesen, wie der cline-Prozess die Daten ändert. Anders ausgedrückt dürfen folgende Befehle in cline nicht durch pline unterbrochen werden:

```
ctrl->c=argv[1][0];
ctrl->length=atoi(argv[2]);
```

Umgekehrt darf der cline-Prozess Daten solange nicht ändern, wie der pline-Prozess Daten ausgibt. In pline dürfen also folgende Befehle nicht durch cline unterbrochen werden:

```
for (i=0;i < ctrl->length;i++) putchar(ctrl->c);
```

Solche Programmteile, die nicht durch andere Prozesse unterbrochen werden dürfen, nennt man kritische Abschnitte.

Um kritische Abschnitte zu realisieren, gibt es in Betriebssystemen verschiedene Mechanismen, wie Schlossvariablen, Semaphore, Bedingungsvariablen und

Monitore. Häufig werden Semaphore verwendet, die z.B. in Unix implementiert sind.

Den Zugriff auf einen gemeinsamen Speicher kann man z.B. mit Hilfe eines Semaphors als kritischen Abschnitt realisieren. Es wird hierbei eine Schaltervariable für den Semaphor verwendet, die beim Wert 1 den gemeinsamen Speicher als benutzt und bei 0 als frei kennzeichnet. Eine einfache Schaltervariable reicht allerdings für die Realisation dieses binären Semaphors nicht aus, denn die Überprüfung des Semaphors auf Freiheit und das Belegen des Semaphors muss als nicht unterbrechbar realisiert sein.

```
/* Warten bis Schaltervariable frei ist */
while (locked);
/* Schaltervariable belegen */
locked = 1;
```

Wären diese Operationen unterbrechbar, so könnten zwei Prozesse die Schaltervariable als frei ansehen und beide belegen. Semaphore sind also nicht normale Benutzervariablen, sondern sind im Betriebssystemkern gemeinsam mit ihren Operationen realisierte Variablen (abstrakte Datentypen).

Unix-Semaphoren sind mehrwertig, d.h. sie können auch mehr als zwei Werte annehmen. Dabei enthält der Maximalwert des Semaphors die Anzahl gleichzeitig aktiver Prozesse, die sich im kritischen Abschnitt befinden. Ist der Maximalwert 1, so handelt es sich um einen binären Semaphor.

Mit mehrwertigen Semaphoren kann z.B. die Pufferverwaltung für Daten erzeugende und konsumierende Prozesse realisiert werden. Der Semaphor gibt dabei die Anzahl der im Puffer befindlichen Datenelemente an. Erzeugende Prozesse erhöhen und konsumierende Prozesse verringern den Semaphor jeweils um 1. Weitere Eigenschaften von Unix-Semaphoren sind:

- Es können Gruppen von Semaphoren erzeugt und manipuliert werden.
- Semaphoren werden i.d.R. automatisch freigegeben, wenn ein Prozess endet.
- Semaphoren haben einen Eigentümer, Zugriffsrechte und Modifikationszeitpunkte.

Der Aufruf semget() legt einen Satz von Semaphoren an oder übernimmt die Kontrolle über einen bestehenden Satz:

```
id = semget(key, nsems, flags)
```

key ist der numerische Schlüssel, der den Satz von Semaphoren identifiziert. nsems ist die Anzahl der Semaphoren. flag ist z.B. IPC_CREAT | 0644 zum Anlegen eines Satzes mit den Zugriffsrechten rw-r–r– und 0 für den Zugriff auf einen existierenden Satz von Semaphoren. In id wird die Kennung des Semaphorensatz vom Betriebssystem zurückgeliefert.

Mit dem Aufruf semop() können die Werte eines Semaphorensatzes geändert werden:

```
semop(id, sops, nsops)
```

id ist die Kennung des Semaphorensatzes, die semget() zurückgeliefert hat. sops ist eine Folge von Operationen, die mit den Semaphoren durchgeführt werden sollen, wobei jede einzelne Operation mit drei ganzzahligen Werten in folgender Form definiert wird:

- sem_num: Nummer der zu bearbeitenden Semaphore
- sem_op: Operation auf der Semaphore
- sem_flg: Optionen für die Operation

nsops gibt die Anzahl von Operationen in sops an.

Der Wert von sem_op bestimmt die Operation mit dem Semaphor:

sem_op = 0: Der aufrufende Prozess wartet, bis der Wert des Semaphors 0 ist, es sei denn im sem_flg wurde IPC_NOWAIT angegeben s.u.

sem_op > 0: Der aufrufende Prozess wartet nicht, sondern er erhöht den Wert des Semaphors um sem_op.

sem_op < 0: Der aufrufende Prozess wartet, bis der Wert des Semaphors im Betrag größer oder gleich dem Betrag von sem_op ist, es sei denn im sem_flg wurde IPC_NOWAIT angegeben s.u. Anschließend wird der Wert des Semaphors um sem_op verringert.

Es folgt ein Beispiel für einen binären Semaphor, mit dem das obige Problem gelöst werden kann oder z.B. der exklusive Zugriff auf eine Datei für genau einen Prozess gesichert werden kann. Die Header-Datei sem.h sieht wie folgt aus:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 123456L
#define PERMS  0666
```

Das Programm zum Belegen des Semaphors lock.c sieht folgendermaßen aus:

```
#include "sem.h"

static struct sembuf op_lock[2] = {
        /* wait for sem#0 to become 0 */
        0, 0, IPC_NOWAIT,
        /* then increment sem#0 by 1 */
        0, 1, 0
};

int my_lock(void) {
        int semid;
        if ((semid = semget(SEMKEY, 1, 0)) == -1) {
                if ((semid = semget(SEMKEY, 1,
                                IPC_CREAT | PERMS)) < 0)
```

```
                        perror("semget error");
        }
        if (semop(semid, &op_lock[0], 2) < 0) {
                perror("semop lock error");
                return(1);
        } else return(0);
}

int main() {
        if (my_lock() == 1) {
          fprintf(stderr,
                  "Die Datei wird schon benutzt.\n");
          exit(1);
        }
        else {
          fprintf(stderr,
                  "Die Datei ist nun exklusiv.\n");
          exit(0);
        }
}
```

Das Freigeben des Semaphors ist in unlock.c wie folgt programmiert:

```
#include "sem.h"

static struct sembuf op_unlock[1] = {
        /* decrement sem#0 by 1
           set's it to 0 even if it is 0 */
        0, -1, IPC_NOWAIT
};


void my_unlock(void) {
        int semid;
        if ((semid = semget(SEMKEY, 1, 0)) == -1) {
                if ((semid = semget(SEMKEY, 1,
                     IPC_CREAT | PERMS)) < 0)
                        perror("semget error");
        }
        if (semop(semid, &op_unlock[0], 1) < 0)
                perror("semop unlock error");
}

int main() {
        printf("unlocking\n");
        my_unlock();
}
```

Mit folgendem Programm rmlock.c wird der Semaphor wieder gelöscht:

```
#include "sem.h"
```

```
int main() {

        int semid;

        if ((semid = semget(SEMKEY, 1, 0)) < 0) {
                perror("semaphore not there");
                exit(1);
        }
        if (semctl(semid, 0, IPC_RMID, 0) < 0) {
                perror("can't remove semaphore");
                exit(2);
        }
}
```

4.9 Nachrichten-Warteschlangen

Eine weitere Art von Mechanismen zur Interprozesskommunikation sind die Nachrichten-Warteschlangen (Message Queues). Eine Message Queue ist eine geordnete Liste von Nachrichten, die vom Betriebssystemkern verwaltet wird. Mit einem gemeinsam genutzten Speicher hat sie einiges gemeinsam:

- Message Queues werden über einen numerischen Schlüssel identifiziert.
- Message Queues haben Eigentümer.
- Message Queues sind Zugriffsrechte zugeordnet.
- Jeder zugangsberechtigte Prozess, der den Schlüssel der Message Queue kennt, kann Nachrichten dorthin senden und von dort empfangen.
- Message Queues nach der Erzeugung ohne zugeordnete Prozesse weiter existieren.
- Message Queues wirken gepuffert:
  - Zum Zeitpunkt des Sendens einer Nachricht muss nicht notwendigerweise ein Empfangsprozess existieren.
  - Zum Zeitpunkt des Empfangs einer Nachricht muss nicht notwendigerweise ein Sendeprozess existieren.

Jede Nachricht in der Message Queue enthält Anwendungsdaten und einen zugeordneten Typ. Dieser Typ ist ein positiver ganzzahliger Wert, der eingesetzt wird, um die Reihenfolge zu kontrollieren, mit der Nachrichten aus der Warteschlange entnommen werden. Durch den Systemaufruf msgget() kann eine neue Message Queue erzeugt werden oder auf eine existierende Message Queue zugegriffen werden.

```
int msgget(key_t key, int msgflag)
```

Hierbei bedeuten die Parameter:

key: Schlüssel der Message Queue.

msgflag: Zugriffsrechte beim Erzeugen der Message Queue z.B. 0644|IPC_CREAT oder 0 beim Zugriff auf eine existierende Message Queue.

Der Rückgabe wert von msgget() ist der Deskriptor zum Ansprechen der Message Queue für Sende- und Empfangsoperationen. Die Sendeoperation ist msgsnd() mit folgender Signatur:

```
int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag)
```

Hierbei bedeuten die Parameter:

msgqid: Deskriptor der Message Queue.

ptr: Zeiger auf eine Struktur der Form:

```
struct msgbuf {
long mtype, /* Typ der Nachricht */
char *data /*Anwendungsdaten */
}
```

length: Länge der Anwendungsdaten in Byte

flag: 0 oder IPC_NOWAIT, wobei 0 bedeutet, dass beim Senden bei einer vollen Message Queue gewartet wird, bis wieder Platz verfügbar geworden ist und IPC_NOWAIT bedeutet, dass bei einer vollen Message Queue nicht gewartet wird und der Rückgabewert der Funktion -1 ist.

Bei erfolgreichem Senden ist der Rückgabewert der Funktion 0, ansonsten -1. Der Betriebssystemkern interpretiert den Aufbau einer gesendeten Struktur nicht, d. h. ptr muss nur auf eine long-Variable zeigen, die den Typ der Nachricht angibt. Als Beispiel wollen wir folgende Struktur in eine Message Queue senden:

```
typedef struct my_msg {
  int msglen; /* nicht mitgesendet */
  long mtype;
  short s;
  char c[80];
} Message;

Message m;
```

Folgender Aufruf versendet den relevanten Teil der Struktur m:

```
msgsnd(msqid, (struct msgbuf *)&m.mtype,
 sizeof(Message) - sizeof(int) - sizeof(long), 0)
```

Um eine Nachricht aus einer Message Queue zu empfangen, wird der Systemaufruf msgrecv() verwendet:

```
int msgrcv(int msqid, struct msgbuf *ptr, int length, long msgtype, int flag)
```

msqid: Deskriptor der Message Queue.

ptr: Zeiger auf den Empfangspuffer.

length: Anzahl Byte des Empfangspuffers.

flag: Z.B. 0, IPC_NOWAIT oder MSG_NOERROR, wobei MSG_NOERROR bedeutet, dass length Bytes ohne Fehler eingelesen werden, wenn mehr als length Byte bereitstehen.

msgtype: Auswahl der zu entnehmenden Daten aus der Message Queue.

- **msgtype=0:** Unabhängig vom gesendeten Typ wird der älteste Eintrag entnommen (0 als Sendetyp ist also nicht gestattet).
- **msgtype>0:** Die älteste Nachricht des Sendetyps msgtype wird entnommen.
- **msgtype<0:** Die älteste Nachricht des kleinsten Sendetyps, der kleiner oder gleich dem Betrag von msgtyp ist wird entnommen (Typ 1 wird vor Typ 2 usw. entnommen).

msg_recv() blockiert, wenn keine passende Nachricht in der Message Queue verfügbar ist, es sei denn IPC_NOWAIT wird in flag gesetzt oder eines der folgenden Ereignisse tritt ein:

- Die Message Queue wird geschlossen.
- Der empfangende Prozess erhält bei der Ausführung von msg_recv() ein Signal.

Bei erfolgreichem Empfang gibt der Rückgabewert die Anzahl der empfangenen Bytes ohne die long-Variable für den Typ an.

Zur Ausführung von Steueroperationen auf einer Message Queue dient die Funktion msgctl():

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

msqid: Deskriptor der Message Queue.

cmd: Kode für das auszuführende Kommando, z.B. IPC_RMID zum Schliessen der Message Queue.

buff: I.d.R. 0.

Bei Erfolg liefert msgctl den Rückgabewert 0 sonst -1 zurück.

Ein Beispiel für die Anwendung einer Message Queue ist die Kommunikation eines Server-Prozesses mit mehreren Client-Prozessen. Alle Clients senden z.B. Nachrichten an den Server mit dem Nachrichtentyp 1. Damit der Client vom Server identifiziert werden kann, könnte er z.B. seine PID in den Anwendungsdaten senden. Der Server weiss dann von wem die Nachricht kommt und sendet seine Antwort an die Message Queue, wobei der gesendete Nachrichtentyp gleich der PID des Client ist. Der Client empfängt die Nachricht mit dem zu seiner PID passenden Typ.

Bezüglich ihrer Funktionalität liegen Message Queues also zwischen Named Pipes und gemeinsamem Speicher. Der Zugang ist flexibler als bei den streng nach dem Schema first-in, first-out (FIFO) arbeitenden Pipes; sie lassen jedoch nicht wahlfreien Zugriff wie gemeinsamer Speicher zu.

Kapitel 5

Systemadministration am Beispiel Linux

Bei der Installation einer Linux-Distribution wird von den halbautomatischen Installations-Programmen, wie YaST bei der S.u.S.E-Distribution, i.a. ein gebrauchsfertiges System erstellt. Sogar die Netzwerkkonfiguration ist i.d.R. so weit komplett, dass eine Verbindung über Modem oder ISDN-Karte zu einem Internet-Provider oder über eine Ethernet-Karte zu einem lokalen Netzwerk aufgenommen werden kann.

Trotzdem ist eine solche halbautomatische Installation meistens nicht fertig. Eventuell muss zusätzliche Software installiert werden, weitere Benutzer-Accounts müssen eingerichtet oder neue Hardware muss angeschlossen und integriert werden. In vielen Fällen sind Anpassungen an die individuellen Wünsche der Benutzer erforderlich. Weiterhin müssen gelegentlich Fehler behoben und Wartungsarbeiten, wie Datensicherungen oder Einspielen neuerer Software-Versionen, durchgeführt werden.

Diese Aufgaben umreißen den Begriff Systemadministration, wobei i.d.R. der Benutzer eines Linux-Systems auch gleichzeitig in die Rolle des Systemadministrators schlüpfen muss.

Der Systemverwalter hat den Account mit dem Namen root und wird auch als Superuser bezeichnet. Der Superuser hat mehr Rechte als ein Benutzer, d.h. aber auch, dass alle sinnvollen und hilfreichen Schutz- und Sicherheitsmechanismen normaler Benutzer für den Superuser außer Kraft sind. Arbeiten sollten daher nur unter dem Superuser ausgeführt werden, wenn sie nicht mit Benutzer-Privilegien durchgeführt werden können.

5.1 Systeminitialisierung und Systemterminierung

5.1.1 Laden des Kernels - Bootkonzepte

Im einfachsten Fall lädt das BIOS einen Linux-spezifischen Bootloader vom ersten Sektor einer Diskette. Dieser Loader schreibt den Kernel an die richtige Stelle im Arbeitsspeicher und übergibt dann die Kontrolle an den Kernel. Der erste Block der Kerneldatei enthält also diesen Bootloader.

Zur Erzeugung einer solchen Bootdiskette wird die Kerneldatei z.B. mit dem Kommando dd auf die Diskette geschrieben:

```
dd if=<kernel_auf_festplatte> of=/dev/fd0 bs=18k
```

Bei dieser Methode hat man allerdings keine Möglichkeit, Bootparameter an den Kernel zu übergeben, d.h. alle im Kernel compilierten Einstellungen müssen zum System passen.

Normalerweise soll Linux auf der Festplatte installiert und von dort gestartet werden. Festplatten werden anders als Disketten in getrennte Verwaltungsbereiche, sogenannte Partitionen, aufgeteilt. Eine Festplatte kann in maximal 4 primäre Partitionen eingeteilt werden. Eine der primären Partitionen kann als erweiterte Partition zusätzlich in logische Partitionen unterteilt werden. Jede der Partitionen kann ein eigenes Dateisystem tragen und von einem anderen Betriebssystem besiedelt werden.

Der Boot-Vorgang von der Festplatte verläuft wie folgt: Das BIOS lädt zunächst den ersten Sektor der Festplatte, den sogenannten Master Boot Record, in den Arbeitsspeicher und führt den darin enthaltenen Bootloader aus. Der Bootloader der Festplatte ist austauschbar, ist aber im Prinzip Teil des bevorzugten Betriebssystems auf der Festplatte.

Dieser Boot-Vorgang birgt Konfliktpotential, wenn weitere Betriebssysteme auf der gleichen Festplatte installiert werden. So kann es passieren, dass der Bootloader nur bestimmte Betriebssysteme unterstützt oder dass bei der Neuinstallation eines zweiten Betriebssystems der Bootloader im MBR einfach überschrieben wird. Ein Ausweg aus dieser Situation ist ein zweistufiges Bootkonzept, bei dem beispielsweise der Windows-Bootloader im MBR aus dem ersten Sektor der als aktiv markierten Windows-Partition einen zweiten Bootloader in den Arbeitsspeicher lädt und diesem Secondary Bootloader das Laden des eigentlichen Betriebssystems überlässt. Ein bewährter häufig eingesetzter Bootloader für Linux ist LILO. Für die wichtigsten Betriebssystem-Kombinationen auf einer Festplatte werden die folgenden Bootkonzepte empfohlen:

Windows 95/98 und Linux: Wenn beide Betriebssysteme in je einer primären Partition untergebracht sind, kann LILO im MBR installiert werden. Bei einer Nachinstallation von Windows kann es aber passieren, dass der MBR mit dem Windows-Bootloader überschrieben wird. Deswegen sollte man sich eine Möglichkeit schaffen, Linux zur Not z.B. von Diskette oder CD zu booten, um LILO neu installieren zu können.

Windows NT und Linux: Hier empfiehlt es sich den Bootloader von NT zu benutzen. Dieser Bootloader kann neben dem Orginal-Bootsektor auch Kopien von Bootsektoren starten. Auf der Windows NT-Partition wird nun eine Kopie des LILO-Bootsektors aus der Linux-Partition angelegt.

Soll nur Linux auf der Festplatte betrieben werden, so vereinfacht sich die Boot-Situation erheblich, z.B. kann LILO direkt im MBR installiert werden oder es wird als Secondary Bootloader von einem anderen Bootloader aus gestartet.

5.1.2 Initialisierung des Kernels

Wenn der Bootloader die Kerneldatei in den Arbeitsspeicher geladen hat, wird sie zunächst dekomprimiert und an die richtige Stelle gebracht. Dann beginnt

die zweite Phase der Initialisierung, in der das Betriebssystem startet und die Kontrolle über die Hardware übernimmt. Zuerst werden einige interne Funktionen und Tabellen des Kernels initialisiert. Dazu gehören die Seitentabellen für das Demand-Paging, die Belegung der Interrupts, die Einrichtung der Prozesstabelle und der Start des Prozess-Schedulers. Wenn der Arbeitsspeicher ohne Fehler ist, verläuft diese Phase i.d.R. reibungslos. Es findet noch keine Bildschirmausgabe statt. Als nächstes wird die Kommandozeile des Kernels ausgewertet und die Argumente an Setup-Funktionen übergeben, die später zur Initialisierung von Gerätetreibern verwendet werden. Die Systemkonsole wird nun für Ausgaben auf den Bildschirm eingerichtet. Bei Geräten mit PCI-Bus wird das PCI-Subsystem aktiviert und dabei das PCI-BIOS aktiviert, das den Bus nach installierten Geräten absucht. Danach wird die Geschwindigkeit des Rechners ermittelt und der Timer kalibriert. Das virtuelle Dateisystem wird initialisiert und der verfügbare und aktuell belegte Arbeitsspeicher auf der Konsole ausgegeben. Nun werden die geräteunabhängigen Netzwerkebenen initialisiert. Dabei werden u.a. die unterstützten IP-Protokolle angezeigt. Nun wird der Prozessor auf bekannte Fehler untersucht, d.h. je nach Prozessortyp können hier die Ergebnisse verschiedener Tests auf der Konsole erscheinen. Als letzter Schritt wird in der zweiten Phase der Linux-Banner mit der Versionsnummer des Kernel ausgegeben.

Die dritte Phase beginnt, indem der Kernel einen Kernel-Thread mit dem Namen init erzeugt. Ein Kernel-Thread ist eine Kernelfunktion, die wie ein Prozess bereits vom Scheduler verwaltet wird, sich aber den Speicherbereich mit anderen Kernel-Threads teilt (gemeinsamer Speicher). init erzeugt zwei weitere Kernel-Threads, nämlich kflushd und kswapd, die den Kernel bei der Verwaltung des virtuellen Speichers unterstützen. Danach wird die Kernelfunktion setup aufgerufen, die für die Initialisierung der Gerätetreiber, des Programmladers und der Dateisysteme verantwortlich ist. setup ruft einen Gerätetreiber nach dem anderen und veranlasst ihn, nach dem zugehörigen Gerät zu suchen, es zu initialisieren und sich nach einer erfolgreichen Initialisierung im Kernel zu registrieren. Die meisten Gerätetreiber geben genaue Statusinformationen auf der Konsole aus. Danach werden die Partitionen geprüft, eventuell vorgesehene RAM-Disks in den Arbeitsspeicher gelesen und die Kernelfunktion zum Laden verschiedener Binärformate vorbereitet. Die letzten Schritte der Kernelinitialisierung bilden die Registrierung der unterstützten Dateisysteme und der Mount der Dateisysteme. Ab jetzt hat der Kernel die Kontrolle über alle Komponenten und Geräte des Rechners und ist bereit, Prozesse zu starten.

5.1.3 Initialisierung der Systemprozesse - Konzept der Runlevel

Die vierte und letzte Phase der Systeminitialisierung beginnt, wenn der Kernel-Thread init die Programmdatei /sbin/init in den Arbeitsspeicher lädt und ausführt. Durch diesen Vorgang wird der erste Prozess erzeugt. Ein Prozess unterscheidet sich vom Kernel-Thread dadurch, dass er einen eigenen virtuellen Adressraum benutzt, der vom Kernel verwaltet wird. Der init-Prozess ist verantwortlich für die Erzeugung aller weiterer Prozesse. Er endet nicht, denn er sorgt z.B. dafür, dass jederzeit Benutzersitzungen neu gestartet werden können. init führt beim Systemstart ein oder mehrere Shell-Skripten aus. Zunächst werden jedoch die in der Datei /etc/inittab angegebenen virtuellen Terminals und

die seriellen Schnittstellen mit getty-Prozessen belegt.

Es gibt verschiedenene Versionen des init-Programms, aber ein System V-kompatibles init hat sich überwiegend, z.B. auch bei S.u.S.E.-Linux, etabliert. Neben Multiuser-Betrieb ist es z.B. möglich das System im Singleuser-Betrieb ohne Netzwerk hochzufahren, um Systemwartungen durchführen zu können. Weiterhin wird das Konzept der sogenannten Runlevel benutzt, mit dem über den Ablauf von Shell-Skripten definierte Zustände des Rechners hergestellt werden.

Jeder Runlevel zeichnet sich also durch eine Laufzeitumgebung des Systems aus. Beim Wechsel zwischen unterschiedlichen Runleveln werden bestimmte Komponenten des Betriebssystems hinzugefügt und andere Komponenten gelöscht. Die Namen der Shell-Skripte und Programme zur Herstellung der Runlevel sind in der Datei /etc/inittab gespeichert. Sie werden vom init-Prozess gestartet. Dabei ist der Default-Runlevel, der beim automatischen Hochfahren des Systems erreicht wird, durch den Eintrag initdefault festgelegt. Normalerweise ist dies Runlevel 2 oder 3. Die Runlevel von S.u.S.E-Linux und ihre Bedeutung zeigt die folgende Tabelle:

Runlevel	Bedeutung
0	Systemhalt
S	Singleuser-Betrieb
1	Mulituser-Betrieb ohne Netzwerk
2	Multiuser-Betrieb mit Netzwerk
3	Mulituser-Betrieb mit Netzwerk und X Window
4	frei
5	frei
6	Neustart

Durch init S kann der Systemadministrator den Rechner beispielsweise von irgendeinem der Runlevel 1, 2 oder 3 in den Singleuser-Betrieb zur Systemwartung bringen. Nach der Systemwartung bringt der Administrator den Rechner normalerweise durch init 2 oder init 3 wieder in den Multiuser-Betrieb mit Netzwerk. Der Aufruf init 0 hält das System an; äquivalent ist der Befehl shutdown -h now. Bei init 6 wird ein Neustart des Systems veranlasst; äquivalent ist hierfür der Befehl shutdown -r now.

Realisiert wird das Runlevel-System mit einer Reihe von hochspezialisierten kleinen Shell-Skripten, die jeweils mit den Argumenten start oder stop aufgerufen werden können. Diese Skripten stehen bei S.u.S.E.-Linux im Verzeichnis /sbin/init.d (bei anderen Linux-Systemen häufig unter /etc/rc.d/init.d). Für jeden Runlevel gibt es nun ein eigenes Verzeichnis von Namensverweisen (symbolische Links) auf diese Shell-Skripten, beispielswiese /sbin/init.d/rc2.d (bzw. /etc/rc.d/rc2.d) für den Runlevel 2.

Die Datei /etc/inittab enthält für jeden Runlevel eine Zeile, in der Form:

```
ID:Runlevel:Aktion:Programmaufruf
```

Die Zeilen zum Start der verschiedenen Runlevel heißen beispielsweise:

```
l0:0:wait:/sbin/init.d/rc 0
```

```
l1:1:wait:/sbin/init.d/rc 1
l2:2:wait:/sbin/init.d/rc 2
l3:3:wait:/sbin/init.d/rc 3
l6:6:wait:/sbin/init.d/rc 6
```

Es wird hier also generell das Skript rc (run command) mit dem Runlevel als Parameter aufgerufen. Weiter ist ID eine Kennung, mit der Fehlermeldungen aus diesem Level beginnen. Aktion ist hier wait, d.h. es wird mit der Bearbeitung weiterer Zeilen aus der inittab gewartet bis das aufgerufene rc-Skript beendet ist.

Das Skript rc arbeitet nun wie folgt: Nehmen wir an, dass der aktuelle Runlevel 2 ist und dass wir durch den Aufruf init 1 in den Runlevel 1 wechseln wollen. Was passiert? init liest /etc/inittab und ruft /sbin/init.d rc 1 auf. Nun übernimmt rc die Kontrolle. rc führt zunächst die Skripte aus dem aktuellen Runlevel 2 (Verzeichnis /sbin/init.d/rc2.d), deren symbolischer Namen mit K beginnt, mit dem Parameter stop aus. Dabei kann die Reihenfolge der Aufrufe von Bedeutung sein. Sie wird durch zwei Zahlen hinter dem Buchstaben K (Kill) angegeben, z.B. wird mit K20apache zunächst der Apache-Webserver heruntergefahren bevor später mit K40network das Netzwerk heruntergefahren wird. Einige der Kill-Skripten des Runlevels 2 heißen in S.u.S.E-Linux beispielsweise:

```
K10mysql
K19cron
K19nscd
K20adabas
K20apache
K20at
K20gpm
K20inetd
K20lpd
...
K40network
K42pcmcia
K44i4l_hardware
K45dummy
K50serial
K81svgatext
K99kerneld
```

Als nächstes ruft rc alle Start-Skripten deren symbolische Namen mit S beginnen für den Runlevel 1. Die zweistellige Zahl hinter dem S ist hier auch wieder für die Reihenfolge des Starts zuständig, z.B. wird mit S01kerneld der Kernel-Dämon gestartet, der z.B. dafür zuständig ist, weitere Kernelmodule zur Laufzeit nachzuladen, bevor die seriellen Schnittstellen mit S02serial initialisiert werden. Es folgen einige der symbolischen Namen für Start-Skripten von Runlevel 1 bei S.u.S.E.-Linux:

```
S01kerneld
S02serial
S03pcmcia
S04dummy
```

```
S09syslog
S10boot.setup
...
S20kbd
```

Um eine zentrale Steuerung der Skripten zu ermöglichen, wird bei S.u.S.E-Linux die Datei /etc/rc.config zum Setzen von Umgebungsvariablen benutzt. Diese Umgebungsvariablen werden bei den Skripten generell durch die Zeile

```
. /etc/rc.config
```

importiert.

Es ist natürlich möglich auch eigene Skripten zur Systeminitialisierung einzufügen. Sie werden dann in das Verzeichnis /sbin/init.d geschrieben und weiterhin werden symbolische Links in den entsprechenden Runlevel-Verzeichnissen angelegt.

5.1.4 Shutdown

Bei einem vernetzten Mehrbenutzer-System wie Linux sind vor dem Abschalten des Rechners einige Terminierungs-Aktionen notwendig. Das Betriebssystem muss heruntergefahren werden, bevor der Rechner abgeschaltet wird. Hierbei werden alle Benutzerprozesse und Dämonen beendet, der Festplatten-Cache zurückgeschrieben und schließlich alle Dateisysteme mit umount abgekoppelt.

Zum geordneten Herunterfahren gibt es das Programm shutdown. Es sorgt dafür, dass alle angemeldeten Benutzer über den Systemhalt informiert werden, verhindert neue Anmeldungen und ruft dann init auf, um in den Runlevel 6 oder 0 zu wechseln. Wenn hier alle Prozesse außer init und der letzten Shell beendet sind, werden alle Dateisysteme außer dem Root-Dateisystem mit umount -a abgekoppelt. Nach der Synchronisation des Root-Dateisystems wird das Schreibrecht darauf weggenommen und dem Kernel mit einem der Befehle halt, poweroff oder reboot von init mitgeteilt, wie der Folgezustand des Systems nach dem Shutdown sein soll. In diesem Zustand kann der Rechner ohne Probleme ausgeschaltet werden.

5.2 Kernel-Konfiguration und Kernel-Module

Der mit der Distribution installierte Kernel ist ein generischer Kernel, der für viele unterschiedliche Hardware-Konfigurationen passt. Oft besteht jedoch der Wunsch oder die Notwendigkeit, einen speziellen Kernel zu erzeugen, der für ein bestimmtes Hardware-System optimiert ist. Dazu kann der Administrator die Kernel-Quellen angepasst zu Prozessortyp, Bus, Erweiterungskarten usw. übersetzen.

Insbesondere zur Nutzung bestimmter Erweiterungskarten ist eine Neuübersetzung des Kernels oft unerlässlich. Einige Erweiterungskarten besitzen zwar ein eigenes BIOS, das aber nur in Single-Tasking-Umgebungen ausreicht. Deshalb muss der Quelltext vor dem Übersetzen zur Treiberauswahl konfiguriert werden. Dies ist aber weniger kompliziert, als es sich anhört.

Die meisten Treiber sind als sogenannte Module übersetzbar. Im laufenden Betrieb können zum Kernel über Module Gerätetreiber oder Funktionen hinzugeladen werden. Lediglich Gerätetreiber, die beim Systemstart benötigt werden

müssen in den Kernel fest eingebaut werden. Dies sind i.w. das Dateisystem, mit dem die Root-Partition formatiert wurde, und der Festplatten-Treiber, der die Platte steuert auf dem das Root-Dateisystem liegt. Wird das System als Diskless Workstation ohne eigene Platte über das Netzwerk betrieben, so müssen die Netzwerktreiber und das NFS-Dateisystem fest in den Kernel eingebunden werden.

Die Arbeit mit Kernel-Modulen hat viele Vorteile. Distributoren können z.B. mit einem modularisierten Kernel kleinere Installationssysteme herstellen, die besser an die Zielsysteme angepasst sind als überladene generischen Kernel. Benutzer profitieren von dem geringeren Speicherbedarf des Betriebssystems, da Module nur bei Bedarf in den Arbeitsspeicher geladen werden. Außerdem sind die Fälle, in denen ein der Kernel speziell übersetzt werden muss, durch die Modularisierung deutlich weniger geworden.

Der Quellkode von Linux liegt normalerweise im Verzeichnis /usr/src/linux. Zur Konfiguration des Kernels gibt es unterschiedliche Möglichkeiten. Alle werden vom Quelltext-Verzeichnis aus über ein bestimmtes make-Target gestartet und erstellen die Konfigurationsdatei .config, aus der zur Übersetzungszeit die eingestellten Parameter gelesen werden.

Konfiguration über Kommandzeilen-Modus:

```
make config
```

Traditionell kann im Kommandozeilen-Modus konfiguriert werden. Es werden dabei Fragen gestellt, bei denen man die Antwort aus 2 oder 3 verschiedenen Alternativen auswählen kann. Bei 2 Alternativen y oder n, wobei y das Programm fest in den Kernel einbindet und n den Kernel ohne das Programm bindet. Bei 3 Alternativen kommt noch die Möglichkeit m hinzu, mit der man das Programm als Modul konfigurieren kann. Wenn eine andere Eingabe als die angebotenen Alternativen erfolgt, wird automatisch eine Hilfestellung ausgegeben.

Konfiguration über ein Menüsystem im Text-Modus:

```
make menuconfig
```

Diese Konfiguration arbeitet auf der Basis der ncurses-Bibliothek zur Terminal-Ansteuerung. Die Arbeit hiermit ist i.w. selbsterklärend.

Konfiguration über ein Menüsystem X Window-Modus:

```
make xconfig
```

Benutzer die das X Window-System benutzen und Tcl/Tk installiert haben, benutzen i.d.R. diese komfortabelste Möglichkeit der Kernel-Konfiguration. Sie ist genau wie die Konfiguration im Text-Menüsystem selbsterklärend.

Bei der Auswahl der Gerätetreiber als Modul müssen Abhängigkeiten der Module untereinander beachtet werden. Viele der kritischen Abhängigkeiten werden aber bei der Konfiguration falls notwendig automatisch korrigiert. Beispielsweise darf man nicht den Dateisystem-Typ msdos in den Kernel fest einbinden und den Basistyp fat als Modul davon trennen.

Nach Verlassen der Konfigurations-Umgebung erfolgt die Übersetzung des Kernels und der Module getrennt voneinander. Der Kernel kann mit folgenden Schritten von einem Shell-Fenster aus erzeugt werden.

- Löschen alter Objekt-Dateien:

```
make clean
```

- Erstellen der Makefiles (Dependencies) aus der Konfigurationsdatei .config:

```
make dep
```

- Übersetzen der Quellen in das komprimierte Kernel-Image:

```
make bzImage
```

Alle Schritte geben ihre Meldungen auf dem Shell-Fenster aus. Speziell im Übersetzungsschritt, der länger dauert, werden sehr viele Meldungen ausgegeben, die man zusätzlich in eine Datei umlenken sollte, um sie später noch einmal ansehen zu können. In der bash oder ksh kann man mit dem tee-Filterprogramm Meldungen und Fehlermeldungen z.B. in eine Datei (hier outanderr) leiten und gleichzeitig am Bildschirm betrachten:

```
make bzImage 2>&1 | tee outanderr
```

Nachdem der Kernel erfolgreich übersetzt wurde, werden als nächstes die Module übersetzt. Dies geschieht durch folgenden make-Befehl:

```
make modules
```

Wenn Kernel und Kernel-Module erfolgreich übersetzt wurden, können sie z.B. auf die Festplatte installiert werden. Da sich das System bei einer falschen Kernel-Konfiguration aber u.U. nicht mehr hochfahren lässt, bietet es sich an den alten Kernel zu retten, so dass z.B. über LILO auch der alte Kernel zur Not geladen werden kann. Um den alten Zustand komplett wieder herstellen zu können, sollte man auch ein Backup von den Kernel-Modulen in /lib/modules machen. Im einzelnen sehen die notwendigen Schritte wie folgt aus:

Kopieren des alten komprimierten Kernel-Images in eine Backup-Datei:

```
cp /boot/vmlinuz /boot/vmlinuz.old
```

Kopieren des neuen komprimierten Images in das /boot-Verzeichnis:

```
cp /usr/linux/src/arch/i386/boot/bzImage /boot/vmlinuz
```

Beim Boot mit LILO muss nach jeder Installation eines neuen Kernels LILO ebenfalls neu installiert werden. Außerdem muss die LILO-Konfigurationsdatei /etc/lilo.conf angepasst werden, damit der Kernel-Backup zur Not auch mit LILO gestartet werden kann. Die Datei /etc/lilo.conf kann bei einer Installation des LILO im MBR z.B. wie folgt aussehen:

```
# LILO Konfigurations-Datei
# Start LILO global Section
boot=/dev/sda
vga=normal
message=/boot/greetings
```

```
read-only
prompt
timeout=100
# End LILO global Section
#
image = /boot/vmlinuz
  root = /dev/sda3
  label = linux
  password=???

#
image = /boot/vmlinuz.old
  root = /dev/sda3
  label = linux.old
  password=???
```

Hierbei wird LILO im MBR der Platte ersten SCSI-Platte sda untergebracht. Es können alternativ die Kernel-Images /boot/vmlinuz und /boot/vmlinuz.old über die Eingabe von linux oder linux.old am Boot-Prompt von LILO gestartet werden.

Nach diesen ganzen Aktionen darf man auf keinen Fall vergessen, LILO neu zu installieren, denn sonst lässt sich das System trotzdem nicht starten. Die LILO-Installation erfolgt einfach durch den Aufruf:

```
lilo
```

Als letztes werden nun noch die Module (evtl. nach vorherigem Backup der alten Module) mit folgendem Kommando vom Verzeichnis /usr/src/linux aus installiert:

```
make modules_install
```

5.3 Dateisysteme, Dateiverwaltung und Datensicherung

Dateisysteme werden während des Betriebs dauernd verändert. Benutzer speichern ihre Daten und Programme ab, Programme wie Mail und News benötigen häufig viel Plattenplatz. Irgendwann ist es so weit, dass der Festplattenplatz aufgebraucht ist. Auf diese Situation kann der Administrator mit unterschiedlichen Maßnahmen reagieren:

- Überflüssige Daten können gelöscht und selten benutzte Daten können komprimiert und/oder auf CD oder Bandarchiv geschrieben werden.
- Es können Disk-Quotas vergeben werden, die den nutzbaren Plattenplatz durch die Benutzer beschränken.
- Durch Einrichtung eines NFS-Dateiservers in einem lokalen Netz kann Plattenplatz eingespart werden. Der Dateiserver hält gemeinsam genutzte Daten und Programme für mehrere Rechner des Netzwerks.

- Das Dateisystem kann durch Installation einer zusätzlichen Festplatte vergrößert werden.

Weiterhin müssen Maßnahmen getroffen werden, die die Auswirkungen von Hardware-Defekte lindern oder es den Benutzern erlauben versehentlich gelöschte Daten wiederherzustellen. Dazu dienen RAID- und Backup-Systeme. Für Administratoren ist es also unerlässlich, sich mit der Verwaltung von Dateisystemen auszukennen.

5.3.1 Partitionierung der Festplatte

Beim PC wird eine Festplatte in Partitionen aufgeteilt. Dies hat verschiedene Vorteile:

- Mehrere Betriebssysteme können auf einer Festplatte residieren.
- Die Datensicherheit wird erhöht, da ein Plattenfehler häufig nur eine Partition betrifft.
- Der Datenbestand lässt sich auf der Festplatte besser strukturieren.

Die Partitionierung einer leeren Festplatte ist i.a. unkritisch. Wenn eine Platte, die Daten enthält, umpartitioniert werden soll, z.B. um eine Partition zu vergrößern, müssen alle Daten unbedingt auf ein anderes Medium zwischengespeichert werden. Sonst gehen Daten mit hoher Wahrscheinlichkeit verloren.

Im ersten Sektor einer Festplatte liegt die Partitionstabelle. Hier können maximal vier primäre Partitionen eingerichtet werden. Anstelle einer primären Partition kann auch eine erweiterte Partition eingerichtet werden, die den gesamten restlichen Speicherplatz umfasst, der keiner primären Partition zugeordnet ist. In der erweiterten Partition können nun weitere logische Partitionen angelegt werden. Der Hauptunterschied zwischen einer primären und einer logischen Partition ist, dass das Betriebssystem direkt nur von der primären Partiton geladen werden kann. Allerdings kann man Betriebssysteme auch indirekt mit einem Bootloader, wie LILO, der auf einer primären Partition installiert ist, von einer logischen Partion starten. Bei der Nutzung einer Festplatte durch mehrere Betriebssysteme sollten die Partitionsgrenzen immer auf Zylindergrenzen der Festplatte liegen.

Unter Linux dient das Programm fdisk zu Erstellung von Partitionen. fdisk ist eigentlich ohne Probleme bedienbar, wenn man die Bedeutung von Boot-Flags und Partitionstypen kennt. Das Boot-Flag ist ein einzelnes Bit für jede Partition. Ist es gesetzt, so kann von dieser Partition ein Betriebssystem geladen werden. Das Windows-Bootprogramm erwartet genau ein gesetztes Boot-Flag auf einer Festplatte. LILO ignoriert die Boot-Flags. Andere Bootloader erlauben auch mehrere gesetzte Boot-Flags. Um zu erkennen um welchen Dateisystemtyp es sich handelt, ordnet jeder Betriebssystem-Hersteller seinen Dateisystemen eindeutige Kennungen (1 Byte) zu. Diese Kennung nennt man Partitionstyp. Mit fdisk ist es nun möglich den Partitionstyp für eine Partition zu setzen. Dies ist vor allem für andere Betriebssysteme als Linux wichtig. Linux erkennt nämlich den Typ des Dateisystems zusätzlich an dessen Aufbau. Gängige Partitionstypen listet fdisk wie folgt auf:

```
0  Empty           17  Hidden HPFS/NTF 5c  Priam Edisk     a6  OpenBSD
1  FAT12           18  AST Windows swa 61  SpeedStor       a7  NeXTSTEP
2  XENIX root      1b  Hidden Win95 FA 63  GNU HURD or Sys b7  BSDI fs
3  XENIX usr       1c  Hidden Win95 FA 64  Novell Netware  b8  BSDI swap
4  FAT16 <32M      1e  Hidden Win95 FA 65  Novell Netware  c1  DRDOS/sec (FAT-
5  Extended        24  NEC DOS         70  DiskSecure Mult c4  DRDOS/sec (FAT-
6  FAT16           3c  PartitionMagic  75  PC/IX           c6  DRDOS/sec (FAT-
7  HPFS/NTFS       40  Venix 80286     80  Old Minix       c7  Syrinx
8  AIX             41  PPC PReP Boot   81  Minix / old Lin db  CP/M / CTOS / .
9  AIX bootable    42  SFS             82  Linux swap      e1  DOS access
a  OS/2 Boot Manag 4d  QNX4.x          83  Linux           e3  DOS R/O
b  Win95 FAT32     4e  QNX4.x 2nd part 84  OS/2 hidden C:  e4  SpeedStor
c  Win95 FAT32 (LB 4f  QNX4.x 3rd part 85  Linux extended  eb  BeOS fs
e  Win95 FAT16 (LB 50  OnTrack DM      86  NTFS volume set f1  SpeedStor
f  Win95 Ext'd (LB 51  OnTrack DM6 Aux 87  NTFS volume set f4  SpeedStor
10  OPUS           52  CP/M            93  Amoeba          f2  DOS secondary
11  Hidden FAT12   53  OnTrack DM6 Aux 94  Amoeba BBT      fd  Linux raid auto
12  Compaq diagnost 54 OnTrackDM6      a0  IBM Thinkpad hi fe  LANstep
14  Hidden FAT16 <3 55 EZ-Drive        a5  BSD/386         ff  BBT
16  Hidden FAT16    56 Golden Bow
```

5.3.2 Einrichtung der physikalischen Dateisysteme

Dem Benutzer zeigt sich das Dateisystem als ein Verzeichnisbaum mit einer Menge verschiedener Dateien. Um die Daten in dieser Form zu organisieren, benötigt das Betriebssystem eine Grundstruktur auf dem Datenträger, in der Verzeichnisse und Dateien angelegt werden können. Im Unterschied zum logischen Dateisystem aus Benutzersicht haben wir es in diesem Fall also mit einem physikalischen Dateisystem zu tun. Jede Partition kann nun ein anderes physikalisches Dateisystem tragen.

In Linux gibt es nun hauptsächlich Programme, die ein Linux-Dateisystem auf einer Partition anlegen. Nehmen wir an, dass wir auf der ersten SCSI-Platte unseres Systems eine DOS-Partition (FAT16) und 3 Linux-Partitionen haben. fdisk zeigt z.B. folgende Ausgabe:

```
   Device Boot    Start       End    Blocks   Id  System
/dev/sda1   *         1       100    803218+   6  FAT16
/dev/sda2           262       278    136552+  82  Linux swap
/dev/sda3           279       527   2000092+  83  Linux
/dev/sda4           101       261   1293232+  83  Linux
```

Der am meisten verwendete Typ für Linux-Dateisysteme ist Extended 2 (EXT2). Mit dem Systemprogramm mke2fs kann das EXT2-Dateisystem nun auf einer Partition angelegt werden, z.B. durch folgenden Aufruf:

```
mke2fs /dev/sda3
```

Doch Vorsicht bei der Anwendung, denn es gehen sicher alle Daten, die dort gespeichert sind, verloren. Für auszulagernde Dateien kennt Linux das Swap-Format, das mit dem Kommando mkswap angelegt werden kann, z.B.:

```
mkswap /dev/sda2
```

Bei der automatischen (Erst-)installation, z.B. mit S.u.S.E.-Linux werden diese Schritte automatisch durchgeführt.

5.3.3 Mount der physikalischen Dateisysteme

Durch das logische Dateisystem verwaltet Linux unterschiedliche physikalische Dateisysteme in einem einzigen Verzeichnisbaum des Systems. Für den Benutzer ist also prinzipiell gleichgültig, ob sich die Daten auf Festplatte, CD, Diskette oder sogar auf einem NFS-Dateiserver im Netz befinden.

Um dieses homogene Bild zu erhalten, werden Unterbäume des Verzeichnisbaums auf die verschiedenen Medien, d.h. bei Festplatten die Partitionen gelegt. Dazu gibt es den Befehl mount, der einem Verzeichnis einen Platz auf einem Medium zuweist. mount kann mit Optionen von der Shell-Kommandozeile aufgerufen werden, oder die Eingaben können von der Datei /etc/fstab durch mount -a gewählt werden. Für unser Beispielsystem könnte /etc/fstab folgendes Aussehen haben:

```
# Festplatte
/dev/sda1 /msdos  msdos    defaults    1  0
/dev/sda2  swap   swap     defaults    0  0
/dev/sda3  /      ext2     defaults    1  1
/dev/sda4  /usr   ext2     defaults    1  2
none       /proc  proc     defaults    0  0
# Restliche Devices
/dev/sr0   /cdrom iso9660  ro,nodev,nosuid,noauto,user    0  0
/dev/fd0   /floppy  auto   noauto,nodev,nosuid,user   0  0
```

In der ersten Spalte der Tabelle erscheint das physikalische Gerät, also z.B. im Falle einer Festplatte die Partition. Bei dem proc-System wird kein Gerät zugeordnet, denn es wird als Pseudo-Dateisystem im Arbeitsspeicher gehalten. Die zweite Spalte kennzeichnet den sogenannten Mount-Point, d.h. das Verzeichnis an dem das Medium angekoppelt wird. Die Swap-Partition hat keinen Mount-Point, denn nur das Betriebssystem soll hierauf zugreifen können. Die dritte Spalte bildet den erwarteten Typ des Dateisystems. Die Option auto beim Floppy-Laufwerk überlässt Linux komplett die Erkennung des Dateisystem-Typs. Hierdurch können verschiedene Dateisysteme abwechselnd auf Floppy angekoppelt werden. Die vierte Spalte zeigt an, mit welchen Optionen der Mount erfolgen soll. Wichtige Optionen zeigt die folgende Tabelle:

Option	Beschreibung
ro oder rw	Nur-Lese- oder Schreib-Lese-Zugriff
auto oder noauto	Erlaubnis oder Verbot des automatischen mount mit mount -a
exec oder noexec	Erlaubnis oder Verbot der Ausführung von Binärdateien
suid oder nosuid	Erlaubnis oder Verbot von Set-ID-Programmen
dev oder nodev	zeichen-, blockorientierte Geräte sind ansprechbar oder nicht
user oder nouser	Erlaubnis oder Verbot des mount durch Nicht-root-Benutzer
sync oder async	synchroner oder asynchroner I/O der Verzeichnisdaten

Der Wert defaults steht für die Kombination rw, suid, dev, exec, auto, nouser und async. Speziell bei Geräten, für die ein mount durch nicht-root-Benutzer möglich ist, stellt der Import von Set-ID-Programme ein Sicherheitsproblem dar. Deshalb wurde für die Floppy und das CD-ROM-Laufwerk die Option nosuid gewählt. Ein weiteres Sicherheitsproblem stellt sich, wenn das Gerät für das der mount erlaubt ist, einen weiteren Geräteeintrag enthält. Dieser darf nicht als Gerät interpretiert werden, sonst sind nicht-root-Benutzer in der Lage weitere Geräte zu importieren. Aus diesem Grund sollte die Option nodev gesetzt werden.

Mit der fünften Spalte wird bestimmt welche Partitionen mit dem dump-Kommando gesichert werden. Hier bedeutet 0 keine Sicherung; 1 heißt Sicherung. Die letzte Spalte wird vom Programm fsck beim Boot benutzt, um anzuzeigen ob und - wenn ja - in welcher Reihenfolge die Dateisysteme überprüft werden sollen. Das root-Dateisystem sollte hier eine 1 haben, damit es als erstes überprüft wird, die anderen EXT2-Dateisysteme sollten hier eine 2 haben, damit sie danach überprüft werden. Dateisysteme, die nicht vom fsck-Programm untersucht werden sollen, wie das swap- und proc-Dateisystem, werden hier mit einer 0 gekennzeichnet.

5.3.4 Datensicherung

Die regelmäßige Datensicherung ist eine der wichtigsten Aufgaben der Systemverwaltung. Bei professionell betriebenen Systemen ist wenigstens eine wöchentliche, wenn nicht eine tägliche Datensicherung erforderlich.

Die Datensicherung kann auf unterschiedlichen Medien erfolgen, z.B. auf CD's, ZIP's oder auch auf einer zweiten Festplatte. Das meist verwendete Medium ist jedoch das Magnetband. Heute werden meistens sogenannte Streamer verwendet, z.B. DAT-Streamer, bei denen sich das Magnetband in einer Cartridge (Gehäuse) befindet.

Zur Verwaltung eines solchen Streamers dient der Befehl mt (magnetic tape). Mehrere Dateiarchive können auf einem Magnetband untergebracht werden; sie sind dann jeweils durch ein End of File-Zeichen (EOF) voneinander getrennt. Es gibt zwei Betriebsarten für Magnetbänder Rewind on Close und No Rewind on Close. Rewind on Close heißt, das nach der Lese- oder Schreiboperation das Band automatisch zurückgespult wird; bei No Rewind on Close wird das Band angehalten und bleibt stehen, bis die nächste Operation ausgeführt wird. Diese Betriebsarten werden über die Geräteschnittstelle spezifiziert. Z.B. bezeichnet nst0 den ersten SCSI-Streamer im No Rewind on Close-Modus, st0 ist derselbe Streamer aber im Rewind on Close-Modus. Nachfolgend werden beispielhaft die wichtigsten mt-Kommandos für einen SCSI-Streamer angegeben. Mit dem

folgenden Befehl wird das Band wird zurückgespult und, falls das Laufwerk es erlaubt, automatisch ausgeworfen:

```
mt rewoffl
```

Das Kommando wirkt auf die Device, die unter dem Namen /dev/tape ansprechbar ist. Dies ist i.d.R. ein symbolischer Link, z.B. auf /dev/st0, der mit folgendem Kommando angelegt werden kann:

```
ln -s /dev/st0 /dev/tape
```

Das folgende Kommando führt dazu, dass das Band zurückgespult wird.

mt rewind

Wenn beispielsweise am Anfang des Bandes ein Archiv geschrieben wurde und ein zweites Archiv auf das Band geschrieben werden soll, kann mit folgendem Befehl der Schreib-/Lesekopf hinter das Ende des ersten Archiv positioniert werden:

```
mt -f /dev/nst0 asf 1
```

Die Option asf steht für absolute skip file. Diese Operation kann durch Zurückspulen und relatives Vorspulen (Option fsf, forward skip file) um eine EOF-Markierung ebenfalls realisiert werden:

```
mt rewind
mt -f /dev/nst0 fsf 1
```

Wenn mehrere Archive auf dem Band gespeichert sind, darf erst wieder hinter dem letzten gespeicherten Archiv ein weiteres Archiv geschrieben werden, sonst werden die Teile hinter dem neuen Archiv unbrauchbar. Der folgende Befehl spult an das Ende aller Archive und kann deshalb benutzt werden, um ein neues Archiv an die bereits bestehenden anzuhängen:

```
mt -f /dev/nst0 eom
```

Die Erstellung der eigentlichen Archive wird nicht mit dem Befehl mt, sondern mit tar (tape archive), cpio, afio oder dump/restore vorgenommen. Neben diesen Archivierungs-Programmen gibt es eine Reihe von High-Level-Programmen zur benutzergeführten Verwaltung der Archive, wie z.B. amanda, tob oder backup. Diese Programme nutzen die Low-Level-Programme zur Archivierung.

Bei einer höheren Backupfrequenz ist es nicht wünschenswert, bei jedem Backup alle Daten zu sichern. Es reicht aus, Daten zu sichern, die sich seit dem letzten Backup verändert haben. Diese Methode nennt sich inkrementeller Backup. Einem vollständigen Backup wird der Level 0 zugeordnet. In den Backups mit Level größer als 0 werden nur die Daten gespeichert, die seit der letzten Sicherung mit Level kleiner oder gleich dem aktuellen Level verändert worden sind. Ein Level 1-Backup enthält also alle Daten, die seit dem letzten Voll-Backup verändert wurden und nicht in einem anderen Level 1-Backup bereits gespeichert sind. Dieses Prinzip findet z.B. beim tar-Befehl Verwendung (s.u.).

Alternativ können Level auch so definiert werden, dass immer alle Veränderungen seit dem letzten Backup eines tieferen Levels gespeichert werden. Dies

führt dann dazu, dass der Umfang der höheren Backup-Level bei Wiederholung steigt.

Selbstverständlich stehen Kommandos zur Erstellung inkrementeller Backups zur Verfügung. Das find-Kommando ist zur Erstellung von Dateilisten hervorragend geeignet. Die Programme tar und dump bieten Optionen mit denen inkrementelle Backups angelegt werden. Es wird nun an einem Beispiel die Vorgehensweise bei einem inkrementellen Backup mit tar vorgestellt. Mit folgendem tar-Befehl wird ein Level 0-Backup des gesamten Dateisystems angelegt:

```
tar -c -M -f /dev/nst0 -V "Level 0-Backup von /"
-g /var/adm/backup/dir /
```

Die Option -c steht für create, d.h. Erstellen eines Archivs. -M bedeutet multivolume, d.h. der Backup kann sich über mehrere Bänder erstrecken. Mit -f wird das Ziel für das Archiv angegeben, d.h. der SCSI-Streamer. Mit der Option -V wird dem Archiv ein Name gegeben, d.h. hiermit kann das Archiv identifiziert werden. Mit der Option -g wird eine Datei bestimmt, die gleichzeitig Zeitmarke des Backup und Inhaltsverzeichnis des Archivs ist. Mit Hilfe dieser Datei ist es möglich, inkrementelle Backups zu machen. Der folgende Befehl schreibt alle Veränderungen relativ zu dem Voll-Backup, also einen Level 1-Backup, heraus.

```
tar -c -z -f /dev/nst0 -V "Level 1-Backup vom ..."
-g /var/adm/backup/dir /
```

Hiermit werden alle Dateien gespeichert, deren Änderungszeit neuer als im letzten Backup ist. Die Option -z bewirkt, dass die Daten mit dem Programm gzip vor dem Sichern komprimiert werden. Bei unkomprimierten Dateien spart das i.d.R. 50 % Platz ein. Wenn dieser Befehl später wiederholt wird, werden nur Dateien gesichert, deren Änderungszeit neuer ist als in allen vorangehenden Backups. Mit folgendem Befehl kann man sich z.B. den Inhalt eines Archivs anzeigen lassen:

```
tar -t -f /dev/nst0
```

Das Zurückspeichern aller Dateien eines Archivs erreicht man z.B. mit folgendem Befehl:

```
tar -x -f /dev/nst0
```

Soll nur die Datei bs.lyx zurückgespeichert werden, so funktioniert das wie folgt:

```
tar -x bs.lyx -f /dev/nst0
```

5.4 Benutzer- und Gruppenverwaltung

Nachdem die Installation abgeschlossen ist, gibt es zunächst nur einen Account für root. Alle weiteren Benutzerbereiche müssen von root eingerichtet werden. Bei vielen Linux-Distributionen gibt es Dienstprogramme zur Unterstützung der Benutzerverwaltung, z.B. YaST bei S.u.S.E.-Linux. Der Vorgang bei der Einrichtung eines neuen Benutzer-Accounts ist bei allen Programmen im Prinzip gleich:

- Es wird ein Eintrag in der Passwortdatei /etc/passwd angelegt.
- Falls nötig wird ein Eintrag in die Gruppendatei /etc/group erzeugt.
- Das Heimatverzeichnis wird erzeugt und eine gewisse Grundausstattung z.B. zur Initialisierung der Shell und anderen Programmen (Skeleton-Files) hineinkopiert und dem neu eingerichteten Benutzer übereignet.
- Falls nötig wird der Benutzer noch in weitere Dateien eingetragen, z.B. für Disk-Quotas oder Zugriffsberechtigungen auf bestimmte Programme, z.B. Datenbanken.

Die Datei /etc/passwd ist auf Linux-Systemen i.d.R. für alle Benutzer lesbar. Dies führt zu einem Sicherheitsproblem, wenn das verschlüsselte Passwort direkt in dieser Datei eingetragen ist. Da der Verschlüsselungs-Algorithmus bekannt ist, können z.B. potentielle Passworte mit Hilfe eines Programms generiert werden, um das Passwort zu erraten. Um einen solchen Angriff zu verhindern, benutzen die meisten Linux-Systeme ein sogenanntes Shadow-Passwort-System. Das verschlüsselte Passwort wird dabei nicht in der weltlesbaren /etc/passwd sondern in einer nur für root und die Gruppe shadow lesbaren Datei /etc/shadow gespeichert.

Neben interaktiven Tools zur Verwaltung von Benutzer-Accounts, wie YaST, bietet das shadow System die Dienstprogramme useradd, usermod, userdel, groupadd, groupmod und groupdel zum Anlegen, Ändern und Löschen von Benutzer- und Gruppenprofilen. Diese Programme können z.B. sinnvoll eingesetzt werden, wenn eine ganze Reihe von Benutzern automatisch angelegt werden soll. Die useradd-Aufrufe können dann z.B. aus einem Skript oder einem C-Programm in einer Schleife erfolgen. Ein useradd-Kommando zur Einrichtung des Benutzers prakt1 mit der User-ID 1001 und Heimatverzeichnis /home/prakt1 in der Gruppe users mit dem verschlüsselten Passwort CqlW5t3tnDQXE sieht wie folgt aus:

```
useradd -u 1001 -g users -d /home/prakt1 -m
-p CqlW5t3tnDQXE prakt1
```

Die Option -m bewirkt, dass das Heimatverzeichnis angelegt wird und alle Skeleton-Files aus dem Verzeichnis /etc/skel dorthin kopiert werden. Anschließend wird das Heimatverzeichnis mit Inhalt dem Benutzer übereignet.

Soll der Benutzer prakt1 und sein Heimatverzeichnis (Option -r) gelöscht werden, so geht dies z.B. mit folgenden Befehl:

```
userdel -r prakt1
```

Die Zuordnung zwischen Gruppenname und Gruppen-ID ist in der Datei /etc/group verzeichnet. Mit dem Befehl groupadd kann eine neue Gruppe angelegt werden. Z.B. wird die Gruppe adabas mit der Gruppen-ID 101 wie folgt angelegt:

```
groupadd -g 101 adabas
```

Der Benutzer prakt1 kann nun nachträglich mit dem usermod Befehl in die sekundäre Gruppe durch Eintrag in /etc/group wie folgt aufgenommen werden:

```
usermod -G adabas prakt1
```

Um den Benutzer prakt1 von vornherein neben seiner primären Gruppe users auch in die sekundäre Gruppe adabas einzutragen, ändert sich der useradd-Befehl um die -G-Option wie folgt:

```
useradd -u 1001 -g users -G adabas -d /home/prakt1
-p CqlW5t3tnDQXE prakt1
```

Während in der Datei /etc/passwd der Benutzer nur mit seiner primären Gruppe, hier z.B. users, verzeichnet ist, können sekundäre Gruppen des Benutzers in der Datei /etc/group angegeben werden.

5.5 Drucker-Installation

Linux verwaltet Druckaufträge in Warteschlangen mit Hilfe eines Hintergrundprozesses mit dem Namen lpd (Line Printer Dämon). lpd nimmt die Druckaufträge der Anwender entgegen und leitet sie auf einen geeigneten Drucker (evtl. auch über das Netzwerk auf einen anderen Rechner) weiter. Der Betrieb des Druckers über lpd hat folgende Vorteile:

- lpd kann auch Druckaufträge annehmen, wenn gerade kein Drucker im System frei ist.
- lpd kann als zentraler Druckerserver für ein lokales Netz fungieren.
- lpd kann mit sogenannten Druckerfiltern sehr unterschiedliche Druckvorlagen für fast jeden Drucker aufbereiten.

Gerade der letzte Punkt macht das Drucken unter Linux flexibel, denn das Postscript-Format kann für fast jeden Drucker in sein spezifisches Format mit Hilfe des Filters Ghostscript umgewandelt werden.

lpd tritt für den Anwender normalerweise nicht in Erscheinung. Er arbeitet mit den Kommandos lpr, lpq und lprm mit dem Druckerdämon lpd zusammen. Mit lpr kann der Anwender sein zu druckendes Dokument an lpd übergeben. Dabei können unterschiedliche Warteschlangen z.B. bzgl. Druckqualität, Farb oder Schwarz-Weiß-Druck eingerichtet sein und mit lpr angesprochen werden. Es gibt eine Default-Warteschlange, in die ein Druckjob ohne Spezifikation der Warteschlange eingestellt wird. Das lpq-Kommando zeigt den Zustand einer Warteschlange an. Mit lprm kann ein Druckjob gelöscht werden.

Für den Administrator gibt es das Programm lpc, mit dem der Druckerdämon lpd verwaltet werden kann, z.B. zum Sperren bestimmter Warteschlangen.

Beim Boot wird lpd i.d.R. automatisch hochgefahren. Die Informationen über seine Konfiguration bezieht er i.w. aus der Datei /etc/printcap. Allerdings sind mit Ausnahme des Papierformats in /etc/printcap direkt keine Informationen über den Drucker gespeichert. Die Ansteuerung des Druckers findet nur mit Hilfe des Druckerfilters statt, der zu der jeweiligen Warteschlange in /etc/printcap spezifiziert wurde.

Ein häufig angewendeter Druckerfilter ist apsfilter, der für verschiedene Drucker leicht konfiguriert werden kann. Die Arbeitsweise von apsfilter lässt sich grob wie folgt beschreiben. Zunächst versucht es den Datentyp der Druckdatei, z.B. ASCII, Postscript, JPEG, GIF, zu erkennen. Hat die Datei nicht den Datentyp Postscript, so wird sie mit Hilfe spezieller Konvertierungsprogramme, z.B.

a2ps für ASCII, in Postscript umgewandelt. Schließlich wird das Ghostscript-Programm aufgerufen, das den eigentlichen Druckertreiber enthält und die Druckausgabe vornimmt.

Bei einem Netwerkdrucker kann die Filterung, d.h. die Umwandlung in Drucker-abhängig Daten, wahlweise beim Drucker-Client oder beim Drucker-Server geschehen. Damit die Client- Rechner berechtigt sind, mit dem Drucker-Dämon lpd des Drucker-Servers zusammen zu arbeiten, können ihre Namen auf dem Server-Rechner in der Datei /etc/hosts.lpd eingetragen werden.

5.6 Netzwerk-Basisinstallation

Die Administration von Netzwerken oder Netzwerk-Anwendungen ist nicht Thema dieses Skripts. Jedoch soll die Installation des Netzwerks besprochen werden, die notwendig ist, um einen Rechner in ein lokales Netz zu integrieren bzw. um ihn z.B. über eine ISDN-Karte mit einem Internet-Provider zu verbinden.

Zunächst muss die Hardware, d.h. meistens eine Ethernet- oder ISDN-Karte oder ein Modem vom Kernel aus gesehen konfliktfrei zur übrigen Hardware betrieben werden können. Für exotischere Hardware muss dazu der Kernel oder Kernel-Module neu generiert werden. Außerdem kann es bei Konflikten mit anderer Hardware sein, dass IO-Port und die Nummer des zugehörigen Interrupt-Request (IRQ) angepasst werden muss. An dieser Stelle möchte ich auf die Handbücher und Hilfestellungen der Distributoren bzw. die Hardware-Handbücher verweisen.

Wenn die Hardware korrekt installiert und der Gerätetreiber geladen ist, kann das TCP/IP-Netzwerk konfiguriert werden. Um verschiedene Netz-Hardware (Netzwerkkarten, Übertragungsmedien) einheitlich zu bearbeiten, definiert TCP/IP ein abstraktes Interface, über das die Netz-Hardware angesprochen werden kann. Dieses Interface bietet eine Menge von einheitlichen Operationen vor allem zum Senden und Empfangen von Datenpaketen.

Über ISDN- oder Modem-Verbindung zu einem Internet-Provider wird heute meistens das sogenannte Point to Point Protocol (PPP) eingesetzt. PPP erlaubt es, TCP/IP über eine serielle (Telefon-)Leitung zu betreiben. Beim Verbindungsaufbau verständigen sich Client und Server über diverse Verbindungsparameter. U.a. kann der Server dem Client eine sogenannte dynamische IP-Adresse zuweisen. Dies bedeutet grob gesagt, dass diese IP-Adresse dem Client nur für eine Sitzung zugeordnet wird. Es ist also für Client-Rechner ein Ausweg aus der Krise, die durch die Knappheit der IP-Adressen entstanden ist. Server-Dienste können allerdings nur mit einer statischen IP-Adresse angeboten werden.

Eine ISDN-Verbindung wird wie folgt konfiguriert. Wenn die ISDN-Karte über den Hardware-Treiber (meistens HISAX-Treiber) steuerbar ist, können im nächsten Schritt die Verbindungsparameter konfiguriert werden. Dazu wird das Programm isdnctrl verwendet, das aber meistens durch eine komfortablere Oberfläche, wie YaST mit dem Skript i4l bei S.u.S.E.-Linux, bedient wird. Parameter sind u.a. die eigene Telefonnummer (MSN), die anzurufende Telefonnummer und der Benutzername und das Passwort für den Provider. Schließlich werden mit isdnctrl noch die ISDN-Netzwerkprotokolle verschiedener Ebenen vereinbart. Das Layer-2-Protokoll ist meistens hdlc. Das Layer-3-Protokoll ist immer trans. Die Encapsulation ist meistens syncppp.

Im nächsten Schritt wird das ISDN-Interface, z.B. ippp0, in die TCP/IP-

Ebene eingebunden. Damit das Interface für TCP/IP nutzbar wird, muss es mit einer IP-Adresse versehen werden. Dazu dient der Befehl ifconfig (interface configure). ifconfig aktiviert das Interface und macht es für das Senden und Empfangen von IP-Datagrammen durch die Netzwerkebene des Kernels bereit. Der einfachste Aufruf ist für ifconfig ist:

```
ifconfig <Interface> <IP-Adresse>
```

In unserem Fall ist der Aufruf etwas komplizierter, da wir es mit einer Punkt-zu-Punkt-Verbindung zwischen zwei TCP/IP-Rechnern zu tun haben:

```
ifconfig ippp0 192.168.7.1 pointopoint 212.201.24.1
```

Hierbei ist IP-Adresse des eigenen Rechners 192.168.7.1 und die des Kommunikationspartners 212.201.24.1. Die eigene IP-Adresse hat weiter keine Bedeutung, denn bei syncppp wird dem Client-Rechner eine dynamische IP-Adresse vom Provider-Rechner zugewiesen.

Den letzten Schritt der Konfiguration bildet das Setzen der Routen. Mit dem folgenden route-Befehl teilen wir dem Betriebssystem mit, dass der Rechner 212.201.24.1 über das Interface ippp0, also die ISDN-Karte erreichbar ist:

```
route add -host 212.201.24.1 dev ippp0
```

Nun muss noch die Default-Route gesetzt werden; sie hat folgende Bedeutung: Wenn von dem eigenen Rechner eine Verbindung zu irgend einem Rechner im Internet aufgenommen werden soll, so kann das nur über den Rechner des Providers gelingen. Einen solchen Rechner nennt man einen Default-Gateway, denn von ihm aus sind weitere Rechner im Internet erreichbar. Mit dem folgenden Befehl teilen wir dem Betriebssystem mit, dass der Default-Gateway 212.201.24.1 über das Interface ippp0, also die ISDN-Karte, erreichbar ist:

```
route add default gw 212.201.24.1 dev ippp0
```

Die route-Befehle können beim Systemstart auch automatisch ausgeführt werden, dafür werden ihre Parameter dann in die Datei /etc/route.conf geschrieben.

Damit anstatt IP-Adressen in der Form 212.201.24.1 auch Rechnernamen wie monet.fh-friedberg.de verwendet werden können, müssen noch sogenannte Nameserver des Domain Name Service (DNS) angegeben werden. Das sind Rechner, die aus dem Rechnernamen die IP-Adresse bestimmen, mit der das IP-Protokoll intern kommuniziert. Um die Auswirkungen von Ausfällen zu reduzieren, werden immer mindestens zwei Nameserver angegeben. Die Datei, in der die Nameserver angegeben werden können, ist /etc/resolv.conf. Bei S.u.S.E.-Linux können die Nameserver auch zentral in der Datei /etc/rc.config angegeben werden.

Damit ist die Konfiguration der ISDN-Verbindung abgeschlossen. An dieser Stelle sollen noch wichtige Dateien zur Konfiguration einer ISDN-Verbindung mit ihren Inhalten beschrieben werden:

/etc/ppp/options.ippp0 enthält z.B. Benutzernamen zur Anmeldung beim Server und Erlaubnis der Vergabe einer dynamischen IP-Adresse durch den Server.

/etc/ppp/pap-secrets enthält Benutzernamen und Passwort zur Anmeldung beim Server.

/etc/rc.config.d/i4l_option.rc.config enthält bei S.u.S.E.-Linux die Verbindungsparameter, u.a. die eigene Telefon-Nummer und die des Providers, das verwendete ISDN-Protokoll und die Leerlaufzeit vor einem automatischen Verbindungsende. Mit diesen Werten wird die ISDN-Hardware mit dem Programm isdnctrl konfiguriert.

Ethernet-Interfaces werden über die Schnittstellen eth0, eth1 usw. angesprochen. Wenn der zugehörige Hardware-Treiber korrekt installiert ist, kann die Netzwerkkarte in das TCP/IP-Protokoll mit dem Befehl ifconfig eingebunden werden:

```
ifconfig eth0 212.201.24.181
```

Wenn der IP-Nummer in der Datei /etc/hosts ein Name zugeordnet wurde, z.B. berlin, ist folgender Aufruf äquivalent zum vorherigen:

```
ifconfig eth0 berlin
```

Als nächstes muss dem Interface eth0 eine Route zugeordnet werden, d.h. in unserem Fall ist dies eine Route zum Netzwerk 212.201.24.0. Der entsprechende route-Befehl lautet:

```
route add -net 212.201.24.0 dev eth0
```

Der Default-Gateway wird zusätzlich eingetragen, um fremde Netzwerke zu erreichen:

```
route add default gw 212.201.24.1 dev eth0
```

Auch hier müssen noch die Nameserver für DNS in /etc/resolv.conf angegeben werden. Damit ist die Basisinstallation der Ethernet-Verbindung abgeschlossen.

5.7 RPM-Softwarepakete

Zur Installation, Deinstallation, Update von Software wird in Linux heute überwiegend der sogenannte Redhat Package Manager (RPM) verwendet. RPM besteht aus einer Reihe von Dienstprogrammen und einer Datenbank für die installierten und zu installierenden Softwarepakete. Die zu verwaltende Software wird in RPM-Archiven zusammen mit Installationsanweisungen für RPM gehalten. RPM-Archive sind Dateien mit der Endung .rpm.

Abhängigkeiten die zwischen verschiedenen Softwarepaketen erfüllt sein müssen werden von RPM mit Hilfe der Datenbank analysiert. Eine solche Abhängigkeit kann z.B. sein, dass bestimmte Pakete installiert sein müssen, damit andere installiert werden können. Z.B. muss das X Window-Basissystem XFree86 mit dem X Server installiert sein, damit X Window-Applikationen installiert werden können. Eine andere Abhängigkeit ist z.B., dass zwei Softwarepakete nicht gleichzeitig installiert sein dürfen. Zum Aufruf des Package Managers dient das Programm rpm, das mit folgender Syntax aufgerufen werden kann:

Befehl	Bedeutung
rpm -i <paket>.rpm	Installieren des Pakets in <paket>.rpm
rpm -e <paket>.rpm	Deinstallieren des Pakets in <paket>.rpm
rpm -U <paket>.rpm	Upgrade des Pakets in <paket>.rpm

Die Deinstallation eines Pakets erfolgt nur dann, wenn es keine andere mit RPM installierte Software gibt, die dieses Paket benötigt. Auch dies wird mit Hilfe der RPM-Datenbank überwacht. Das Upgrade ist nicht gleichzusetzen mit der Deinstallation des alten Pakets und der Installation des neuen Pakets. Upgrade untersucht vor der Deinstallation welche Dateien des betroffenen Pakets seit der Installation oder dem letzten Upgrade verändert wurden, z.B. um das Paket an das System anzupassen. Solche Dateien werden i.d.R. nicht gelöscht, sondern mit der Erweiterung .rpmorig oder .rpmsave umbenannt, bevor die neuen Dateien installiert werden. Es kann auch sein, dass die alten Dateien unter ihrem ursprünglichen Namen weiterbestehen und nur Dateien mit der Endung .rpmnew neu installiert werden.

Mit rpm -q können auch Auskünfte aus den RPM-Archiven oder aus der RPM-Datenbank eingeholt werden. Die folgende Tabelle zeigt eine Auswahl der Optionen des Befehls rpm -q:

Option	Bedeutung
-i	Informationen über das Paket
-l	Dateiliste des Pakets
-f	Paket, das in der Datei gespeichert ist
-d	Dokumentation anzeigen
-c	Konfiguration anzeigen
-R	Paket-Abhängigkeiten anzeigen
–scripts	Skripten zur (De-)Installation anzeigen

Es folgen einige rpm-Beispielbefehle:

```
rpm -q -i gcc
```

Hier werden Paketinformationen über den gcc-Compiler ausgegeben:

```
Name        : gcc                           Relocations: (not relocateable)
Version     : 2.7.2.3                       Vendor: SuSE GmbH, Nuernberg, Germany
Release     : 38                            Build Date: Fre 23 Jul 1999 00:41:47 CEST
Install date: Fre 26 Mai 2000 19:48:34 CEST Build Host: fatou.suse.de
Group       : unsorted                      Source RPM: gcc-2.7.2.3-38.src.rpm
Size        : 6927072
License     : 1987-95 Free Software Foundation, Inc.
Packager    : feedback@suse.de
Summary     : The GNU C compiler and support files
Description :
Compiled with 486 optimization, to maximize performance on 486 processors,
but works fine with 386.
```

```
NOTE: Be sure to install at least the following packages besides this
one, or you won't be able to compile:  binutils, include, libc, and
the third part of the kernel source.  (the include files)
Original Version from FSF. Compiled with support for g77 and gnat.

Authors:
--------
...
```

Der folgende Befehl soll den Namen des Pakets ausgeben, in dem der Befehl /bin/ls gespeichert ist.

```
rpm -q -f /bin/ls
```

Dieser Befehl liefert z.B. die Ausgabe:

```
fileutil-3.16-73
```

Bisher wurde RPM nur zur Verwaltung von fertigen Binärpaketen verwendet. Man kann jedoch auch die Quellen eines Software-Pakets, das Source-RPM, installieren und diese ggfs. nach eigenen Modifikationen mit dem Dienstprogramm rpm in ein Binary-RPM übersetzen. Das Binary-RPM lässt sich nun installieren. Aus den entpackten und modifizierten Quellen kann nun noch ein neues Source-RPM erstellt werden. Mit der Manual-Page von rpm kann man sich mit diesen Möglichkeiten genauer vertraut machen.

Es gibt eine Reihe von Werkzeugen mit denen RPM-Archive verwaltet werden können, z.B. der Midnight Commander (mc), xrpm (in Python geschrieben), krpm (von KDE), gnorpm (von GNOME), YaST (von S.u.S.E.) und alien (in Perl geschrieben). alien kann u.a. alte TGZ-Archive in RPM-Pakete umwandeln.

Kapitel 6

Netzwerkfähige Unix-Anwendungen

6.1 Kommandos zum Zugriff auf entfernte Rechner

Es gibt unter Unix (Linux) eine Reihe von Funktionen, mit denen Benutzer Sitzungen auf entfernten Rechnern öffnen, Dateitransfer von oder zu entfernten Rechnern initiieren und Informationen über entfernte Rechner einholen können.

Eine wesentliche Funktion eines Netzwerks ist es, eine Sitzung an einem entfernten Rechner eröffnen und dort arbeiten zu können. Dem Benutzer stehen dann alle Funktionen des entfernten Rechners zur Verfügung. Grundsätzlich gibt es zwei Möglichkeiten zur Verwaltung von Sitzungen auf entfernten Rechnern.

rlogin: Mit dem Kommando rlogin kann man eine Sitzung auf einem entfernten Unix-Rechner starten.

telnet: Mit dem Kommando telnet kann man zusätzlich Sitzungen auf Nicht-Unix-Rechnern (auch auf Unix-Rechnern) starten, die das telnet-Protokoll unterstützen.

Die Benutzung von rlogin ist sehr einfach: Hat der Benutzer die gleiche Kennung, wie auf dem lokalen Rechner, so wählt er sich mit folgendem Kommando in den entfernten Rechner ein:

```
rlogin <Rechnername>
```

Will der Benutzer sich unter einem anderen Benutzernamen als dem momentan aktuellen auf dem entfernten System einwählen, so gib er folgendes Kommando ein:

```
rlogin <Rechnername> -l <Benutzername>
```

Normalerweise wird danach das Passwort des Benutzers verlangt, bevor er die Sitzung beginnen kann. Es gibt Situationen, wo diese Passwort-Abfrage störend ist. Z.B. wenn ein Programm sich automatisch, d.h. ohne Eingabe des Passwort

durch den Benutzer, auf dem entfernten Rechner einwählen will. Für diesen Fall ist es für den Inhaber der entfernten Benutzerkennung auch möglich, anderen Benutzern Zugriff auf seine Benutzerkennung ohne Passwort zu gewähren. Natürlich birgt dies ein gewisses Sicherheitsrisiko in sich. verzeichnet, Alles was der Benutzer dazu machen muss, ist eine Datei mit dem Namen .rhosts in seinem Heimatverzeichnis anlegen, in der der entfernte Benutzer mit dem Rechner steht. Die Einträge in .rhosts haben also folgende Form:

```
<Rechnername> <Benutzername>
```

Der .rhosts-Mechanismus funktioniert nicht nur bei rlogin, sondern bei allen r-Kommandos, also z.B. rsh und rcp.

Das Kommando rsh (remote shell) führt ein Kommando auf einem entfernten Rechner aus und startet dazu dort die Login-Shell. rsh verbindet bei der Kommandoausführung die Standardeingabe, -ausgabe und -fehlerausgabe des lokalen Rechners mit dem entfernten Rechner. Die Sitzung auf dem entfernten Rechner wird geschlossen, wenn das Kommando beendet ist. Die Syntax des rsh-Befehls ist wie folgt:

```
rsh <Rechnername> [-l Benutzername] <Kommando> <Argumente>
```

Die eckigen Klammern bedeuten, dass der Benutzername wie bei rlogin optional angegeben werden kann. Es folgt ein Beispiel für rsh:

```
rsh berlin ls -al
```

Mit dem Komando rcp (remote copy) können Dateien zwischen Unix-Rechnern ausgetauscht werden. Auf dem lokalen Rechner arbeitet rcp wie cp. Normalerweise befindet sich die Quell- oder die Zieldatei nicht am lokalen Rechner. Dann muss der Rechnername mit der Datei und ggfs. dem Pfad zusammen angegeben werden. Folgender Befehl kopiert Datei1, die unter dem Pfad1 auf dem entfernten Rechner gefunden wird, auf den lokalen Rechner als Datei2 im Pfad2.

```
rcp <Rechnername>:<Pfad1/Datei1> <Pfad2/Datei2>
```

Umgekehrt ist es auch möglich eine lokale Datei zum entfernten Rechner zu kopieren:

```
rcp <Pfad1/Datei1> <Rechnername>:<Pfad2/Datei2>
```

Mit der Zusatzoption -r ist es auch möglich, ganze Verzeichnisse mit Inhalt rekursiv mit rcp zu kopieren.

Das Pendant zu rlogin aus der TCP/IP-Familie ist telnet. Der einfachste Aufruf, um eine Sitzung auf einem entfernten Rechner zu eröffnen, lautet:

```
telnet <Rechnername>
```

Anschließend erfragt telnet den Benutzernamen und das Passwort. Man kann telnet auch auch als interaktives Programm aufrufen. Der Aufruf lautet dann einfach:

```
telnet
```

Mit ? können die implementierten Kommandos erfragt werden. Zu einzelnen Kommandos kann dann noch eine weitere Hilfestellungen angezeigt werden. Das Kommando open öffnet eine Sitzung, close schließt eine Sitzung und quit beendet telnet und die evtl. noch aktive Sitzung.

Das File Transfer Protocol (FTP) ist das Pendant des TCP/IP zum Unix-Programm rcp. ftp kann z.B. mit Rechnernamen aufgerufen werden:

```
ftp <Rechnername>
```

Das Programm ftp ist ein TCP/IP-Client-Prozess, der sich mit dem FTP-Server-Prozess auf dem angegebenen entfernten Rechner verbindet. Es werden dann vom Server Benutzername und Passwort erfragt. Nach erfolgreichem Login können Dateien vom Client zum Server mit put, oder vom Server zum Client mit get übertragen werden. Vorsicht: Je nachdem, ob man auf dem lokalen Rechner oder dem entfernten Rechner angemeldet ist, hat put und get natürlich eine andere Bedeutung. Dateien können im ascii-Modus oder im binary-Modus übertragen werden. Der ascii Modus dient zur Übertragung von Textdateien. Textdateien die in einem anderen Zeichenkode z.B. EBCDIC dargestellt sind, werden vor der Übertragung in ASCII-Kode umgewandelt. Für die Übertragung von Binärdateien ist dieser Modus ungeeignet, da er evtl. Daten verfälscht. Im binary-Modus werden die Daten daher in ihrer Originalversion übertragen.

Das ftp-Programm lässt sich mit Hilfe von Kommandos relativ komfortabel steuern. Die Hilfestellung zu ftp-Kommandos kann man mit ? oder help einsehen. Die wichtigsten Kommandos sind open zum Öffnen einer Sitzung, close zum Schließen einer Sitzung, put und get zur Dateiübertragung und quit zum Verlassen von ftp. Wird ftp ohne Rechnernamen aufgerufen, so muss der FTP-Server mit open <Rechnername> kontaktiert werden.

Zur Information über Rechner und über Benutzer gibt es unter Unix und in der TCP/IP-Protokollfamilie eine ganze Reihe von Kommandos, die ebenfalls im Client-/Server-Modus arbeiten. Die folgende Tabelle zeigt einige wichtige Informations-Kommandos mit ihrer Beschreibung:

Kommando	Beschreibung
rusers	aktive Benutzer entfernter Rechner (Unix)
rup	Status entfernter Rechner: Name, Uptime, Last (Unix)
finger	Informationen über Benutzer auf Rechnern mit Finger-Server (TCP/IP)

6.2 X Window-System

Das X Window-System realisiert eine grafische Benutzungsoberfläche für Unix. X Window wurde am MIT (Massachusetts Institute of Technology) entwickelt. Der Begriff grafische Benutzungsoberfläche besagt, dass alle Bildschirmausgaben im Grafikmodus erfolgen. Dies ist gegenüber dem Textmodus eine wesentliche Verbesserung, da nicht nur Texte sondern Grafiken und Bewegtbilder in Farbe ausgegeben werden können. Moderne Benutzungsoberflächen haben eine von den Anwendungen getrennte Funktionalität und ersetzen die Befehlseingabe durch direkte Manipulation z.B. mit der Maus.

Das X Window-System ist weitestgehend hardwareunabhängig konzipiert und auf alle ANSI-C/POSIX-konformen Systeme portierbar. Es unterstützt sich überlappende, hierarchisch angeordnete Fenster sowie Grafik- und Textoperationen. Das besondere am X Window-System ist seine Netzwerkfähigkeit, d.h. das X Window-Client-Prozesse auf entfernten Rechnern laufen können und mit dem X Window-Server-Prozess auf dem lokalen Rechner Ein- und Ausgaben über das sogenannte X Window-Protokoll austauschen. Der X Window-Server ist also neben der Kommunikation mit den X Window-Clients für die Ausgabe der Grafik als Bitmap auf dem Bildschirm und für die Eingabe per Tastatur und Maus zuständig.

Der X Window-Server-Prozess arbeitet also i.d.R. mit mehreren X Window-Client-Prozessen zusammen. Es ist aber auch möglich, dass ein X Window-Client-Prozess gleichzeitig mit mehreren X Window-Server-Prozessen zusammenarbeitet. Natürlich können sich X Window-Client-Prozesse und der X Window-Server-Prozess auch auf demselben Rechner befinden.

Die Netzwerkfähigkeit von X Window hat enorme Vorteile: So kann ein Benutzer, der an einemeinem grafikfähigen Rechner arbeitet, eine X Window-Anwendung auf einem Spezialrechner (z.B. Datenbank-Server) laufen lassen, der nicht grafikfähig ist und sich online die Ergebnisse dieser Anwendung ansehen. Die Kommunikation zwischen Client- und Server-Prozessen verläuft asynchron, d.h. eine Client-Anwendung kann eine Anforderung zu jeder Zeit an den Server absetzen, auch wenn eine vorangegangene Anforderung beispielsweise noch nicht vollständig bearbeitet ist. Deshalb ist die X Window-Kommunikation i.d.R. schnell.

Die X Window-Clients bestimmen normalerweise nur den Inhalt der Fenster. Eine Ausnahme bildet hier der sogenannte Window Manager. Hierbei handelt es sich um einen speziellen Client-Prozess, der die Verwaltung der Fenster für den X Window-Server übernimmt. Es gibt kann also genau ein Window Manager für einen X Window-Server gleichzeitig aktiv sein. Die Window Manager bieten u.a.:

- Dekorationen von Fenstern durch Titelleisten und Fensterrahmen
- Icon-Verwaltung
- Verwaltung des Eingabeschwerpunkts (input focus)
- angepasste Tastatur und Maustasten-Anwendung, z.B. Hilfemenüs, Pop Up Menüs usw.

Der Window Manager ist zuständig für das Verschieben von Fenstern, das Ändern der Fenstergröße, das Hervorholen verdeckter Fenster und die Wahl des input focus. Für die Implementierung XFree86 unter Linux stehen unterschiedliche Window Manager zur Auswahl. Am meisten verwendet werden momentan wohl der fvwm2 und der Window Manager von KDE. KDE ist allerdings mehr als ein Window Manager, denn es gibt eine ganze Reihen von nützlichen KDE-Anwendungen, die natürlich auch unter anderen Window Managern arbeiten.

Weiterhin gibt es eine Vielzahl von unterschiedlichen X Window-Client-Programmen, von der Terminal-Emulation angefangen über Editoren und Textverarbeitungsprogrammen bis hin zu Spielen. Viele der Anwendungen der sogenannten Core-Distribution beginnen mit dem kleinen Buchstaben x, z.B.:

X Window-Client-Programm	Kurzbeschreibung
xterm	Terminal-Emualtion mit Shellaufruf
xconsole	Konsolen-Emulation
xman	Manual-Pages
xload	Auslastungsanzeige
xmore	more-Version
xedit	ASCII-Editor
xclock	Uhr
xeyes	Cursor-gesteuerte X Augen

Ohne weiteren Eingriff wird eine X Window-Anwendung normalerweise ihre Ausgaben auf den Rechner leiten, von dem sie auch gestartet wurde. Wenn der X Window-Client also auf einem entfernten Rechner läuft, muss ihm mitgeteilt werden, an welchen anderen Rechner, d.h. an welchen X Window-Server, er seine Ausgaben senden soll. Dies geschieht über die Umgebungsvariable DISPLAY, z.B. mit folgendem Aufruf:

```
export DISPLAY=linux35:0
```

Hiermit wird allen X Client-Programmen, die nach dieser Setzung gestartet werden, mitgeteilt, dass sie ihre Ausgaben an den X Window-Server auf dem Rechner mit dem Namen linux35 schicken sollen. Wenn kein Kurzname vereinbart worden ist, muss der voll qualifizierte DNS-Name, z.B. linux35.fh-friedberg.de, angegeben werden. Hinter dem Rechnernamen steht noch mit einem Doppelpunkt getrennt eine 0. Dies ist notwendig, da es X Window-Rechner mit mehreren Bildschirmen gibt. Die erste 0 teilt dem Server mit, dass die Ausgabe an den ersten angeschlossenen Bildschirm gehen soll.

Auf dem X Window-Serverrechner wird die Ausgabe eines X Window-Client nur dann empfangen und dargestellt, wenn dies ausdrücklich für den Client-Rechner erlaubt wird. Dies ist eine Sicherheitsmaßnahme, denn sonst könnte der X Window-Server weltweit von irgendwelchen X Window-Clients bombardiert werden. Die Kommandos zur Zugriffskontrolle auf den X Window-Server sind xauth und xhost. Mit folgendem Befehl, der auf dem lokalen Rechner linux35 ausgeführt wird, wird X Window-Clients auf dem Rechner linux01 der Zugriff auf den X Window-Server auf Rechner linux35 erlaubt:

```
xhost +linux01
```

Mit folgendem Befehl gestattet man weltweit allen X-Window-Clients Zugriff auf den X Window-Server:

```
xhost +
```

6.3 Network File System (NFS)

Für die Arbeit mit Rechnern in einem lokalen Netz ohne ein verteiltes Dateisystem gibt es grundsätzlich zwei Möglichkeiten:

- Die Daten werden auf die Rechner übertragen, auf denen sie mit den Programmen verarbeitet werden können.

- Die Programme werden auf die Rechner übertragen, auf denen die Daten liegen.

Da der Transfer von Daten oder Programmen manuell z.B. mit Hilfe von rcp oder ftp geschehen muss, beinhaltet dies folgende Probleme:

- Die Handhabung der Dateiübertragung ist kompliziert und fehleranfällig.
- Die Netzbelastung ist relativ hoch.
- Es werden Dateien redundant im Netz gespeichert.
- Es können inkonsistente Kopien von Dateien entstehen.

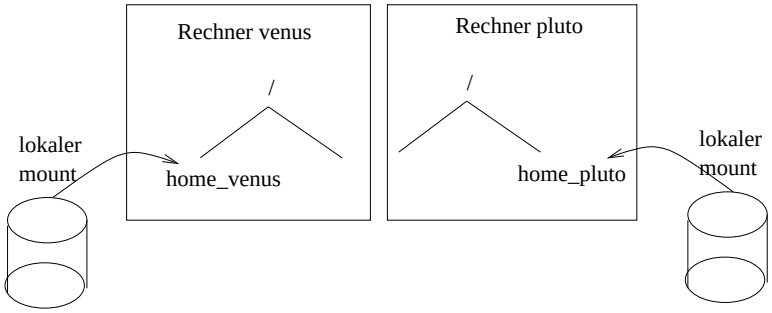
Um diesen Problemen zu begegnen, wurden verteilte Dateisysteme geschaffen. Hierbei wird für die Benutzer (Client) ein ortstransparenter Zugriff auf Dateien entfernter Rechner (Server) ermöglicht. Von verschiedenen Client-Rechnern aus können auch die selben Dateien auf einem oder auch mehreren Server-Rechnern angesprochen werden. Man spricht bei solchen Server-Rechnern auch kurz von Datei-Servern (File-Server). Die verbreitetsten verteilten Dateisysteme im Unix-Bereich sind Network File System (NFS), Remote File Sharing (RFS) und das Andrew File System (AFS). Wir werden im folgenden NFS, das in Linux-Systemen hauptsächlich betrieben wird, beschreiben.

Anwendungsbeispiele für ein verteiltes Dateisystem, wie NFS, sind z.B. zentral auf dem Datei-Server liegende Heimatverzeichnisse der Benutzer, die einmalige Installation von Software auf dem Datei-Server, z.B. von Programmen in /usr, und die gemeinsame Nutzung von Daten im CD-Laufwerk auf dem Datei-Server.

Wie wir schon besprochen haben, besteht der lokale Verzeichnisbaum eines Rechners i.d.R. aus mehreren Partitionen, die sich auch auf unterschiedlichen Festplatten befinden können. Dies hat Vorteile bei Plattenausfällen und bei der Datensicherung. Die Partitionen tragen ein bestimmtes Dateisystem, z.B. EXT2 oder msdos, und werden mit dem Befehl mount in den Verzeichnisbaum eingehangen. Dieses Konzept des lokalen-mount wurde nun für NFS erweitert.

NFS-Server können Unterbäume (nicht Partitionen), die auf ihren lokalen Festplatten gespeichert sind, an NFS-Clients exportieren. NFS-Clients können nun diese Unterbäume auf einen Mount-Point in ihrem lokalen Verzeichnisbaum importieren. Dabei ist die Beziehung zwischen einem NFS-Client und einem NFS-Server immer direkt, d.h. es gibt keine Ketten aus Client-Server/Client-Server. Zur Bearbeitung einer Server-Datei auf dem Client werden Teile der Datei über das Netz vom Datei-Server zum Client transferiert. Dateien werden allerdings niemals physikalisch verlagert, d.h. sie bleiben auf dem Rechner resident, auf dem sie erzeugt wurden.

Was müssen nun die Systemverwalter zum Betrieb des NFS-Client-Rechners und des NFS-Server-Rechners tun? Betrachten wir dazu folgendes Beispiel. Wir administrieren zwei Rechner lokal wie folgt:

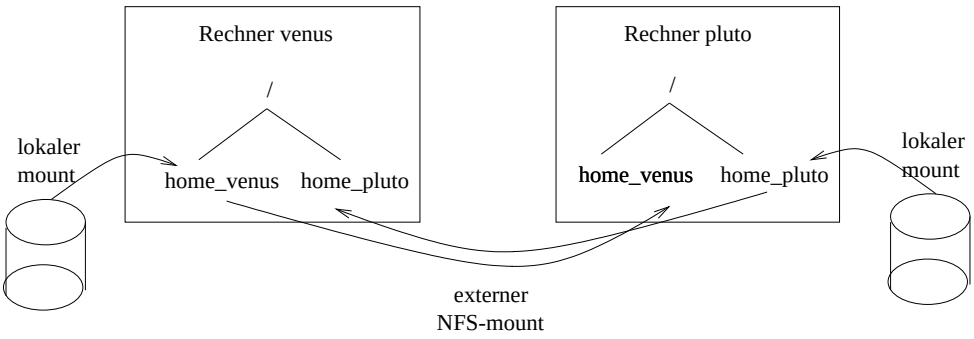


Nehmen wir an, dass auf dem Rechner venus im Verzeichnis home_venus das Heimatverzeichnis des Benutzers Kienzle liegt und auf dem Rechner pluto im Verzeichnis home_pluto das Heimatverzeichnis von Benutzer Hauser liegt. D.h. Kienzle arbeitet i.d.R. auf venus, Hauser arbeitet i.d.R. auf pluto. Nun braucht Hauser auch einen Account auf venus und Kienzle einen Account auf pluto und beide wollen die Daten ihrer Heimatverzeichnisse auf dem jeweils anderen Rechner weiterbenutzen. Was müssen wir dafür tun?

Wir müssen beide Rechner sowohl als NFS-Client als auch als NFS-Server einrichten. Wir sehen also, dass ein Rechner sowohl NFS-Client als auch NFS-Server sein kann. (Die Eigenschaft Client oder Server zu sein bezieht sich ja streng genommen auch nicht auf Rechner sondern auf Prozesse.)

- venus ist NFS-Server und exportiert home_venus an pluto; pluto ist NFS-Client und importiert home_venus von venus
- pluto ist NFS-Server und exportiert home_pluto an venus; venus ist NFS-Client und importiert home_pluto von pluto

Es ergibt sich folgendes Bild:



Der Systemverwalter des jeweiligen NFS-Server-Rechners gibt den Unterbaum zum Export an den jeweiligen NFS-Client-Rechner durch einen Eintrag in der Datei /etc/exports frei:

```
auf venus: /home_venus pluto.fh-friedberg.de(rw, root_squash)
auf pluto: /home_pluto venus.fh-friedberg.de(rw, root_squash)
```

Nach der Änderung der Datei /etc/exports muss der NFS-Server-Prozess neu gestartet werden durch:

```
/sbin/init.d/nfsserver restart
```

Nun kann der Systemverwalter des jeweiligen NFS-Client-Rechner einen Mount-Point anlegen und die Daten mit mount importieren:

```
auf venus: mount -t nfs pluto.fh-friedberg.de:/home_pluto /home_pluto
auf pluto: mount -t nfs venus.fh-friedberg.de:/home_venus /home_venus
```

Diese externen NFS-mounts können natürlich auch durch einen Eintrag in /etc/fstab automatisch beim Hochfahren des Rechners gemacht werden.

Für den aufmerksamen Leser stellt sich die Frage: Wie ist die Zugriffsüberwachung bei importierten NFS-Dateien realisiert? Die Antwortet lautet genauso wie im lokalen System, d.h. über die User-ID und Group-ID des zugreifenden Prozesses. D.h. der Zugriffsschutz arbeitet nur dann zuverlässig, wenn jeder Benutzer auf dem NFS-Client-Rechner und auf dem NFS-Server-Rechner dieselbe User-ID und Group-ID hat. Die konsistente Verwaltung der Accounts wird bei mehreren Unix-Systemen und Benutzern manuell schnell zu einem Problem. Auch aus diesem Grund gibt es eine Verwaltungs-Software für Dateien wie /etc/passwd für Netzwerke. Diese Software heißt Network Information Service (NIS) und wurde früher auch Yellow Pages genannt. Sie wird im folgenden Kapitel besprochen.

Die folgende Tabelle zeigt einige wichtige mount-Optionen:

NFS-mount-Option	Beschreibung
tcp, udp	TCP oder UDP Protokoll zum Server
rsize, wsize	Anzahl Bytes beim Lesen/Schreiben
fg, bg	Abbrechen/Wiederholen von mount-Versuchen nach Timeout
soft	nach Major-Timeout I/O-Fehler melden
hard	Wiederholen einer fehlerhaften Operation mit Konsolmeldung
intr, nointr	Unterbrechen hängender Dateioperation (hard) möglich, unmöglich
timeo	Minor-Timout-Wert in Zehntel Sekunden (Default 7)
retrans	Höchstzahl von Minor-Timeouts vor einem Major-Timout

Bei der mount-Operation ist das Timeout-Verfahren in NFS wichtig, wenn der NFS-Server oder die Verbindung zum NFS-Server gestört ist.Dabei werden zunächst in kurzen Abständen mount-Versuche und bei weiteren Misserfolgen in längeren Abständen weitere mount-Versuche gemacht. Das Timeout-Verfahren arbeitet wie folgt: Begonnen wird mit dem Minor-Timout. Nach jedem Misserfolg wird der bisherige Timeout verdoppelt, bis 60 Sekunden erreicht sind oder retrans-Timeouts gewartet wurde. Danach beginnt ein neuer Major-Timeout, der sich dadurch auszeichnet, dass mit dem doppelten Minor-Timeout als Startwert begonnen wird. Nach weiteren Major-Timeouts sind Major- und Minor-Timout beide 60 Sekunden lang. D.h. es wird nur jede Minute ein neuer mount-Versuch unternommen.

Die folgende Tabelle zeigt noch wichtige export-Optionen:

NFS-export-Option	Beschreibung
ro (default)	schreibgeschützt exportieren
rw	schreibbar exportieren
root_squash (default)	Client-User-ID 0 wird auf anonyme Server-ID abgebildet
no_root_squash	Client-User-ID 0 ist auch im Server die User-ID 0
squash_uids, squash_gids	angegebene ID's auf anonyme im Server abbilden
anonuid, anongid	anonyme ID's im Server setzen (z.B. -2 oder 65534)
noaccess	Zugriff auf Unterbaum des exportierten Unterbaums verbieten
insecure	Erlaubnis für NFS-Clients, die keinen reservierten Port benutzen

Zur Sicherheit sollten Optionen wie insecure und no_root_squash möglichst nicht benutzt werden. Außerdem sollte ein Unterbaum nur wenn es wirklich notwendig ist, als schreibbar (rw) exportiert wenn. Natürlich darf nicht an alle Rechner, sondern nur an bekannte vertrauenswürdige Rechner exportiert werden.

Bei importierten Dateien können weiterhin Probleme entstehen, wenn die Uhren von NFS-Server- und NFS-Client-Rechner nicht synchronisiert sind. Insbesondere Programme wie make, die mit dem Änderungszeitpunkt von Dateien arbeiten, bekommen dann Probleme. Es gibt Programme, wie xntp, netdate oder rdate, mit denen die Uhrzeit auf Rechnern mit einem Rechner mit Referenzzeit synchronisiert werden kann.

6.4 Network Information System (NIS)

Wie bei der Diskussion von NFS erwähnt, können Dateien zur Administration für Rechner im lokalen Netzwerk durch das Network Information System (NIS) verwaltet werden. Das Ziel ist dabei die Übereinstimmung und Synchronisation von Administrations-Dateien auf allen Rechnern des lokalen Netzes. Beispiele für solche Administrations-Dateien sind /etc/passwd, /etc/group und /etc/hosts.

NIS basiert wie NFS auch auf einer Client-/Server-Architektur. Es gibt einen Master-Server und u.U. mehrere Slave-Server, die auf einer Datenbank sogenannte Maps an die Clients exportieren. Maps sind die Datenbank-Repräsentation der Administrations-Dateien. Die folgende Tabelle zeigt die Namen der Administrations-Dateien mit ihren Maps:

Administrations-Datei	Maps
/etc/hosts	host.byname, host.byaddr
/etc/networks	networks.byname, networks.byaddr
/etc/passwd	passwd.byname, passwd.byuid
/etc/group	group.byname, group.bygid
/etc/services	services.byname, services.bynumber
/etc/rpc	rpc.byname, rpc.bynumber
/etc/protocols	protocols.byname, protocols.bynumber

Es gibt unterschiedliche Maps für verschiedene Suchschlüssel, z.B. host.byname und host.byaddr. Maps werden nur im Master-Server angelegt und verändert.

Der Master-Server exportiert die Maps automatisch auf die Slave-Server. Die Slave-Server machen den Zugriff schneller und sicherer.

Zum Erzeugen der Maps aus den Administrations-Dateien dient eine spezielle make-Steuerdatei mit dem Namen ypMakefile. Der NIS-Server wird durch das Programm ypserv gestartet. Für einen NIS-Client werden in der Datei /etc/yp.conf die Namen der NIS-Server-Rechner eingetragen und die Client-Programme, z.B. ypcat, ypmatch und ypwhich in /usr/sbin installiert.

Kapitel 7

Überblick über Windows-Betriebssysteme

Bereits Mitte der 80er-Jahre zeichnete sich ein Trend zu grafischen Nutzungsoberflächen (Graphical User Interface, GUI) ab. Diesem Trend folgte die Firma Microsoft durch Entwicklung von MS-Windows, das ein GUI für MS-DOS ist. Erst mit Windows 95 wurde ein eigenständiges Betriebssystem ohne MS-DOS-Basis entwickelt.

7.1 Windows 95/98

Windows 95 ist ein echtes 32-Bit Betriebssystem. In Windows 95-Fenstern können neben neuen Anwendungen auch MS-DOS Anwendungen betrieben werden. Außerdem ist es möglich Windows 3.11-Anwendungen, wenn auch nur im 16-Bit-Modus, zu betreiben. Wesentliche Merkmale und Vorteile von Windows 95 sind:

- neues Dateisystem mit 32-Bit-Festplatten- und Dateizugriff
- lange Dateinamen bis zu 255 Zeichen
- neue Dienstprogramme, z.B. zum Wiederherstellen gelöschter Dateien, zur Defragmentierung der Festplatten oder zur Erhöhung der Festplattenkapazität durch Komprimierung
- gleichzeitiger Ablauf mehrerer Anwendungen durch Multitasking

Etwa zur gleichen Zeit wie das Betriebssystem erschienen auch neue Versionen der bekannten Anwendungen Word, Excel und Access unter dem Paketnamen Office 95, die ebenfalls die Vorteile des neuen 32-Bit-Betriebssystems nutzen.

Windows 95 ist im Unterschied zu Windows NT ein reines Client-Betriebssystem, d.h. es ist nicht dazu geeignet, in einem Netzwerk Serverdienste anzubieten.

Windows 98 ist der Nachfolger von Windows 95. Es ist vor allem für Rechner interessant, die über neue Hardwaremöglichkeiten verfügen, wie z.B. ein Hochgeschwindigkeitsbus für Grafikausgaben AGP (Advanced Graphics Port), USB Schnittstellen (Universal Serial Bus) ein DVD-Laufwerk oder die Stromsparfunktion ACPI (Advanced Configuration Power Interface). Bietet die Hardware

diese Möglichkeiten, so sollte Windows 98 statt 95 eingesetzt werden, vor allem im Bereich von Computerspielen, -simulationen und -animationen. Gegenüber Windows 95 bringt Windows 98 vor allem folgende Verbesserungen:

- verbesserte Stabilität und Performance
- verbessertes Plug & Play
- Unterstützung neuer Hardware, wie z.B. DVD, AGP, ACPI und USB
- Windows NT-kompatible Treiberarchitektur
- verbessertes Power Management
- gleichzeitiger Betrieb mehrerer Grafikkarten und Monitore (Multimonitoring)
- Fixieren des Endzustands des Arbeitsspeichers auf der Festplatte und direktes Starten ohne erneutes Booten beim nächsten Einschalten (OnNow)

Windows 98 ist somit eine interessante Alternative zu Windows 95 vor allem für neuere Rechner im Büro und Heimbereich und eine sinnvolle Ergänzung zu Windows NT.

7.2 Windows CE

Die Hardware-Plattform für Windows CE ist der Handheld-PC, der über einen meheren MByte großen ROM und RAM-Speicher verfügt, aber kein integriertes Festplatten-, Disketten- oder CD-ROM-Laufwerk verfügt. Die Dateneingabe wird meist mittels einer kleinen Tastatur und einem Stift auf dem Bildschirm ausgeführt (Touch Screen). Die Daten werden auf RAM oder PCMCIA-Karten gespeichert und entweder über Kabel oder eine Infrarotschnittstelle mit dem PC ausgetauscht. Über diese Schnittstellen können natürlich auch Programme einschließlich neuer Updates übertragen werden.

Interessant ist Windows CE für Handhelds, weil hier Windows-PC-Anwendungen wie Word, Excel, Schedule und Internet Explorer verfügbar sind. Weiterhin besteht die Aussicht, dass Windows CE in Zukunft auf einer Vielzahl von Alltagsgeräten, wie Autoradios und Handys eingesetzt wird.

7.3 Windows NT

Windows NT (New Technology) wurde als 32-Bit-Server-Betriebssystem konzipiert. Dabei wird Multi Tasking- und Multi User-Betrieb unterstützt. Damit ist es im PC-Bereich der Hauptkonkurrent für die verschiedenen Unix-Versionen. Windows NT 4.0 gibt es in zwei Varianten Workstation und Server, die Unterschiede sind in der folgenden Tabelle ersichtlich:

	Workstation	Server
Verbindungen zu anderen Clients	10	unbegrenzt
Verbindung zu anderen Netzen	unbegrenzt	unbegrenzt
Multi-Prozessorbetrieb	2 Prozessoren	4 Prozessoren
Remote Access Service	1 Verbindung	255 Verbindungen
Directory Replikation	Import	Import und Export
Macintosh-Dienste	Nein	Ja
Netzwerktyp	Peer to Peer	Server

Bei der Entwicklung von Windows NT wurden folgende Ziele verwirklicht:

Portabilität: Beim Wechsel zu anderen Prozessor-Architekturen sollte der Aufwand möglichst gering sein. Deshalb wurde Windows NT so konzipiert, dass nur ein kleiner, prozessorabhängiger Teil des Betriebssystemkerns, der Hardware Abstraction Layer (HAL) ausgetauscht werden muss.

Multiprozessorbetrieb: Moderne Rechner werden zunehmend mit mehreren Prozessoren (Symmetrische Multiprozessoren, SMP) ausgestattet. Dies wird von Windows NT unterstützt.

Netzintegration: Die Integration verschiedener Netzwerkprotokolle wird durch Netzhardware (Network Driver Interface Specifikation, NDIS) und zur Anwendungssoftware (Transport Driver Interface, TDI) erreicht.

Verteilte Verarbeitung: Die Unterstützung für verteilte Datenverarbeitung in Form von Client-Server-Lösungen wird durch DCE-RPC (Distributed Computing Environment, Remote Procedure Call) und DCOM realisiert.

POSIX-Kompatibilität: Die POSIX-Kompatibilität wurde zeitweise in den USA bei öffentlichen Aufträgen verlangt. In Windows NT kann sie mit Hilfe eines Subsystems erreicht werden. Damit können auch POSIX-konforme Unix-Programme nach Neuübersetzung in Windows NT laufen.

C2-Sicherheit: Windows NT erfüllt die Sicherheitsstufe C2 nach dem Orange Book.

Windows NT verwendet objektorientierte Konzepte. Dabei ist Windows NT kein objektorientiertes Betriebssystem, d.h. es ist nicht in einer objektorientierten Sprache implementiert. Der objektorientierte Ansatz vereinfacht die gemeinsame Nutzung von Betriebsmitteln durch mehrere Prozesse und den Schutz vor unberechtigten Zugriffen. Folgende objektorientierte Konzepte kommen bei Windows NT zum Einsatz:

Kapselung: Ein Objekt besteht aus Daten und Funktionen. Die Daten werden in der objektorientierten Sprechweise Attribute genannt; die Funktionen werden als Methoden oder Dienste bezeichnet. Auf die Daten eines Objektes kann nun nur über seine Methoden indirekt zugegriffen werden. Dies schützt vor unzulässigen Zugriffen auf die Daten.

Klassen: Eine Klasse bestimmt die Attribute und Methoden gleichartiger Objekte. Die Objekte werden auch als Instanzen der Klasse bezeichnet.

Vererbung: Klassen können Attribute und Methoden an Unterklassen weitergeben oder vererben. Zusätzlich können Unterklassen weitere oder veränderte Attribute und Methoden enthalten. Somit werden Klassen durch Vererbung in spezialisierte Klassen überführt.Die Klasse PKW kann als Unterklassen, z.B. Limousine, Kombi, Coupe, Cabriolet oder Roadster haben.

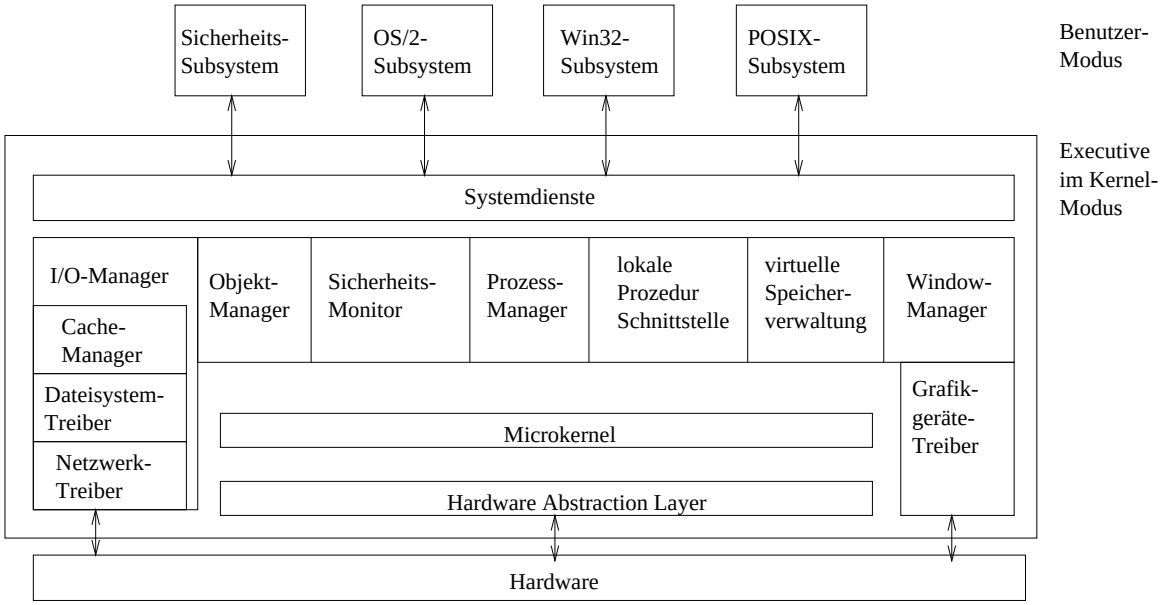
Polymorphismus: Dieser Begriff kann wörtlich als Vielgestaltigkeit übersetzt werden und meint, dass ein Methodenaufruf in Abhängigkeit von der Klasse des angesprochenen Objekts unterschiedliche Verarbeitungsschritte bewirkt. Beispielsweise führt die Funktion + bei Zahlen zu einer arithmetischen Addition, bei Zeichenketten hingegen zur Konkatenation, d.h. dem Hintereinanderhängen, von zwei Zeichenketten.

Windows NT betrachtet Betriebsmittel wie Dateien, Prozesse, Thread, Semaphore und Fenster als Objekte und erzeugt und verwaltet sie in einheitlicher Weise mit dem Objekt-Manager. Das Bild zeigt den grundsätzlichen Aufbau von Windows NT. Der Executive läuft im Kernel-Modus des Prozessors und hat Zugang zu den Systemdaten und zur Hardware. Die diversen Subsysteme laufen im Benutzer-Modus des Prozessors und können nicht direkt sondern nur indirekt über den Executive auf die Hardware zugreifen. Die Anwendungen greifen auf die Subsysteme über ein sogenanntes API (Application Program Interface) zu.

Die Modularität beruht i.w. auf dem HAL der dafür sorgt, dass die meisten Systemmodule die selbe abstrakte Sicht der darunter liegenden Hardware haben. Dazu werden allgemein gültige (generische) Hardware-Befehle und -Meldungen in konkrete Äquivalente der Prozessor-Plattform umgesetzt. Außerdem unterstützt HAL das Symmetrische Multiprocessing (SMP).

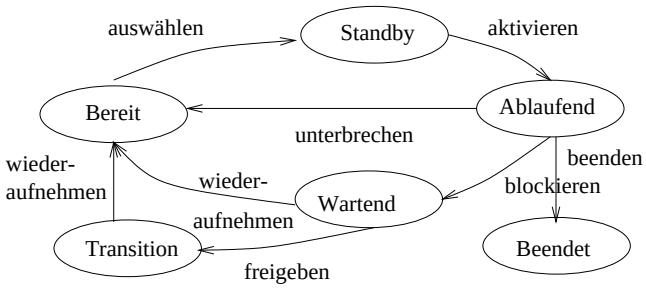
Der Microkernel ist die zentrale Komponente des Executive. Er steuert die Ausführung von Threads, Prozessen, Interrupts und Multiprozessor-Synchronisation. Der Microkernel läuft nicht in Threads ab und nimmt nicht am Paging teil.

Die Executive Services stellen eine Vielzahl an Funktionen für die Subsysteme über die Schnittstelle Systemdienste bereit. Mit Ausnahme des I/O-Managers und des Window/Graphic-Moduls benutzen alle Executive Services den HAL. Der I/O-Manager verarbeitet Anforderungen aus einer Warteschlange von I/O-Aufträgen. Der Objekt-Manager dient als Verwalter für alle anderen Objekte, d.h. Prozesse, Threads, Dateien, Speicherausschnitte usw. Der Zugriff auf die Objekte wird mit Hilfe von Sicherheits-ID's kontrolliert. Quotas geben an wieviele Objekte eines Typs benutzt werden dürfen.



Was ist nun eine Sicherheits-ID? Es handelt sich hierbei um eine global eindeutige Zahl, die einen Benutzer eindeutig identifiziert. Dazu werden in Windows NT Zahlen der Länge 128 Bit mit einem Hilfsprogramm erzeugt. Wenn man alle Zahlen gleichmäßig auf der Erde verteilen würde, würden immer noch $6 * 10^{23}$ Werte pro Quadratmeter zugeteilt werden. D.h. die Wahrscheinlichkeit, dass eine Zahl doppelt vorkommt ist hinreichend gering. Der Sicherheits-Monitor verhindert unerlaubte Zugriffe. Dazu besitzt jedes Objekt eine Kontrollliste für Zugriffe (Access Control List), die eine Anzahl von Sicherheits-ID's enthält. Bei jeder Sicherheits-ID ist vermerkt, welche Zugriffe erlaubt und verboten sind. Der Benutzer des Systems erhält bei seiner Anmeldung ein Sicherheitstoken, mit dem er sich bei allen Operationen gegenüber dem Betriebssystem ausweist. Dieses Token enthält eine Sicherheits-ID für den Benutzer und eine für alle Gruppen, denen er angehört. Vor einem Zugriff vergleicht der Sicherheits-Monitor nun die Sicherheits-ID's des Objektes mit dem zugreifenden Prozess des Benutzers und erlaubt oder verweigert den Zugriff.

Prozesse heißen in Windows NT Thread Objects. Zuerst werden Threads erzeugt und initialisiert, wodurch sie ablaufbereit werden. Die im Bild gezeigten Übergänge zwischen den Thread-Zuständen werden durch den Scheduler (Zeitpunkt bestimmen) und den Dispatcher (Prozess aktivieren) ausgelöst.



Die Zustände lassen sich wie folgt beschreiben:

Bereit (Ready): Der Thread kann ausgeführt werden.

Standby: Der Thread ist zur Ausführung ausgewählt, muss aber noch waren, bis ein Prozessor verfügbar wird. Wenn die Priorität des Threads hoch genug ist, kann ein laufender Thread unterbrochen werden. Ansonsten muss gewartet werden, bis der gerade laufende Prozess blockiert (wird) oder endet.

Ablaufend (Running): Ein laufender Thread belegt den Prozessor, bis er unterbrochen wird z.B. weil seine Zeitscheibe abgelaufen ist oder er sich selbst blockiert oder beendet.

Wartend (Waiting): Dieser Zustand tritt in den folgenden Fällen ein: Der Thread wartet auf ein Ereignis z.B. das Ende einer Ein-/Ausgabe und ist dadurch blockiert oder er wartet um eine Synchronisation mit anderen Threads auszuführen.

Transition: Dieser Zustand folgt auf den Wartezustand, sofern nach dessen Ablauf nicht alle Ressourcen verfügbar sind.

Beendet (Terminated): Ein Thread kann sich selbst beenden oder er wird durch einen anderen Thread beendet.

Die Häufigkeit, mit der Prozesse eine Zeitscheibe zum Ablauf zugeteilt bekommen, richtet sich nach deren Priorität. Prioritäten werden zwischen 0 und 31 vergeben, wobei 31 die höchste Priorität ist. Die Werte 16 bis 31 gehören zur Echtzeitklasse, während 0 bis 15 zur Klasse der variablen Prioritäten gehören. In der Echtzeitklasse ändern sich Prioritäten der Threads nicht. In der Klasse variabler Prioritäten wird ein Startwert zugewiesen, der sich ändern kann.

Der Prozess-Manager bietet Dienste zur Erzeugung, Nutzung und zum Löschen von Threads. Wenn beispielsweise eine Win32-Anwendung den Systemaufruf CreateProcess absetzt, wird eine Nachricht an das Win32-Subsystem gesendet. Dieses Subsystem ruft den Prozess-Manager auf, der den Prozess erzeugt. Der Prozess-Manager ruft seinerseits den Objekt-Manager auf, der ein Prozess-Objekt erzeugt und ein Objekt-Handle an das Win32-Subsystem zurückgibt.

Die lokale Prozedurschnittstelle (Local Procedure Call Facility) ermöglicht eine Client-Server-Beziehung zwischen den Subsystemen des Benutzermodus und den Subsystemen des Executive. Client- und Server-Teile befinden sich hierbei auf dem selben System. Dieses Konzept führt zu einer Reihe von Vorteilen:

Anwendungen können in einheitlicher Weise mit dem Executive kommunizieren

Es wird eine Grundlage für die verteilte Verarbeitung geboten.

Der Executive ist gegen andere Module geschützt, da sie nicht direkt auf Speicherbereiche des Executive zugreifen können.

Die Schnittstelle zum Executive wird vereinfacht, wodurch neue API's jederzeit ergänzt werden können.

Die virtuelle Speicherverwaltung (Virtual Memory Manager) bildet logische Programmadressen in physikalische Speicheradressen ab. Jedem Prozess steht dabei ein linearer Adressraum mit 32 Bit-Adressen zur Verfügung.

Das Window/Graphic-Modul ist für die grafische Oberfläche zuständig. Die Bedienung von Windows NT unterscheidet sich nicht grundlegend von anderen Windows-Versionen.