Die Welt von

UNIX

Es gibt eine Theorie, die besagt, wenn jemals irgendwer genau rausfindet, wozu das Universum da ist und warum es da ist, dann verschwindet es auf der Stelle und wird durch etwas noch Bizarreres und Unbegreiflicheres ersetzt.

Es gibt eine andere Theorie, nach der das schon passiert ist.

Zu diesen Unterlagen

Dieser Kurs behandelt einen kleinen Teil aller Befehle, Werkzeuge und Möglichkeiten von UNIX. Infolge der enormen Anzahl von Elementen und Versionen, die das UNIX System bilden, ist es unmöglich deren Anwendung und auch deren Zusammenwirken insgesamt aufzuzeichnen.

Für detailliertere Informationen steht Ihnen jeweils zu jedem UNIX System eine eigene Systemdokumentation zur Verfügung.

Ziel

Nach diesem Quartal können Sie sich in den meisten UNIX System bewegen und sich zurechtfinden. Sie kennen die Zusammenhänge eines UNIX Systems und können eigene Problemlösungen entwickeln.

Viel Spass

René Muggli

Benutzereingaben sind in `Courier fett`
Systemausgaben in `Courier`

Inhaltsverzeichnis

Zu diesen Unterlagen 1
Ziel 1
Inhaltsverzeichnis 3
Geschichtlicher Rückblick 7
  Hintergrund 7
  Entwicklungsrichtungen von UNIX Systemen 8
Ziele von UNIX 9
  Aufgaben eines Betriebssystems 10
  Die Komponenten von UNIX 11
    Der Kernel 12
    Die Shell 13
    Das Filesystem 14
Der Einstieg ins UNIX 16
  Der Login 16
    Regeln zum Login Namen: 16
    Regeln zum Password: 17
  Einige beteiligte System Komponenten zum Login 18
    Das System File /etc/passwd: 18
    Das System File /etc/shadow: 19
    Das System File /etc/group: 19
  Logout 20
Die Befehlssyntax 21
  Abbruch von Befehlen 21
Aktive Benutzer abfragen 22
UNIX email 23
  Mail senden 23
  Mail abrufen 23
Struktur des UNIX Systems 24
  Die wichtigsten Directories unter root: 25
  Directories 27
    Directory wechseln 27
  Pfadnamen 28
    Absolute Pfadnamen 28
    Relative Pfadnamen 28
  Filenamen 29
    Directory Inhalt anzeigen 29
  Zugriffsberechtigungen 31
    Konsequenzen von fehlenden Zugriffsberechtigungen 33
    Default für Zugriffsberechtigung ändern 34
  Sonderzeichen 35
    Neutralisieren von Sonderzeichen 36

Inhalt eines Files bestimmen ... 37
Wichtige UNIX Befehle ... 38
Directory erstellen ... 38
Directory löschen ... 39
Inhalt von Files anzeigen ... 40
Files löschen ... 42
Die Sache mit den i-nodes ... 44
Files kopieren ... 45
Files umbenennen / verschieben ... 47
vi : Der Full Screen Editor ... 49
Starten des vi ... 50
Die zwei verschiedenen Modi des vi Editors: ... 51
Austritt aus vi ... 52
Cursorbewegungen ... 53
Befehle zur Texteingabe ... 54
Löschen ... 54
Rückgängig machen ... 54
Zeichen mit Sonderfunktionen ... 55
Suchen von Zeichenfolgen ... 55
Ersetzen von Zeichenfolgen ... 56
Zeilen zusammenfügen ... 56
Zeilen trennen ... 56
vi Parameter ... 57
vi Parameter setzen ... 57
Korn Shell ... 59
Befehlswiederholung ... 59
Befehl anzeigen ... 59
Befehl zurückholen und editieren ... 60
Befehle zurückholen und modifizieren ... 61
Alias ... 62
Systemprompt anpassen ... 62
Files drucken ... 63
Drucker-Status abfragen ... 63
Druckaufträge annullieren ... 64
Drucker aktivieren und inaktivieren ... 64
Files suchen ... 66
Ein- und Ausgabe Umlenkung ... 69
Pipelines und Filters ... 71
Files vergleichen ... 72
Konvertieren und kopieren eines Files ... 75
Files sortieren ... 76
Zeichenfolgen in einem File suchen ... 78
Suchmuster Definitionen: ... 80
Doppelte Zeilen eliminieren ... 81
File Dump ... 82
Files kopieren mit Intervention ... 83

Zeilen, Worte, Zeichen zählen 84
Systemstatus 85
Datum, Zeit und Zeitzone abfragen 85
Name / Version des UNIX Systems 86
Befehlsdurchführungszeit feststellen 87
Anzahl belegter Blocks pro File oder Directory 88
Anzahl freier Blocks pro Filesystem 89
Prozesse / Prozess Status Liste 90
Beeinflussen von Prozessen 93
Automatische Programmausführung 94
Erstellen eines Crontab Jobs 95
Systemadministration 97
Verschiedene Superuser 97
Shellprogrammierung 100
Verarbeitung durch die Shell 100
Allgemeines über Shellskripts 101
Aufruf eines Shellskripts 101
Variablen 102
Variablenbezeichnung 102
Ausgabe der Variablen 102
Variablentypen 105
Bedingte Ausführungen 112
Einfache Bedingung 112
Mehrfachverzweigung 117
Schleifen 119
while Schleife 119
until Schleife 121
for Schleife 123
Schleifen Control Befehle 125
Weitere UNIX Befehle 127
Der Befehl eval 127
Der Befehl ulimit 128
Der Befehl type 128
Der Befehl expr 129
Der Befehl tput 131
Der Befehl trap 132
Bücherliste 133

Geschichtlicher Rückblick

Hintergrund

UNIX hat eine lange und faszinierende Geschichte und Entwicklung hinter sich. Die Geschichte von UNIX reicht zurück bis in die späten 60er Jahre. Damals entwickelte Ken Thompson das Betriebssystem MULTICS weiter zum Betriebssystem UNICS (Uniplexed Information and Computing Service). Trotz des Wortspiels über "EUNUCHS" hatte der Name Bestand, auch wenn der Name später zu UNIX geändert wurde.

Mitarbeiter von Thompson bei Bell Labs (eine Tochterfirma der AT&T) waren von diesem Betriebssystem so beeindruckt, dass Dennis Ritchie und seine Mitarbeiter halfen das Betriebssystem weiterzuentwickeln. Bei den damals verwendeten Maschinen (PDP-11/45 und PDP-11/70) war ein hardwaremässiger Speicherschutz möglich, was dem Betriebssystem erlaubte mehrere Benutzer zu unterstützen

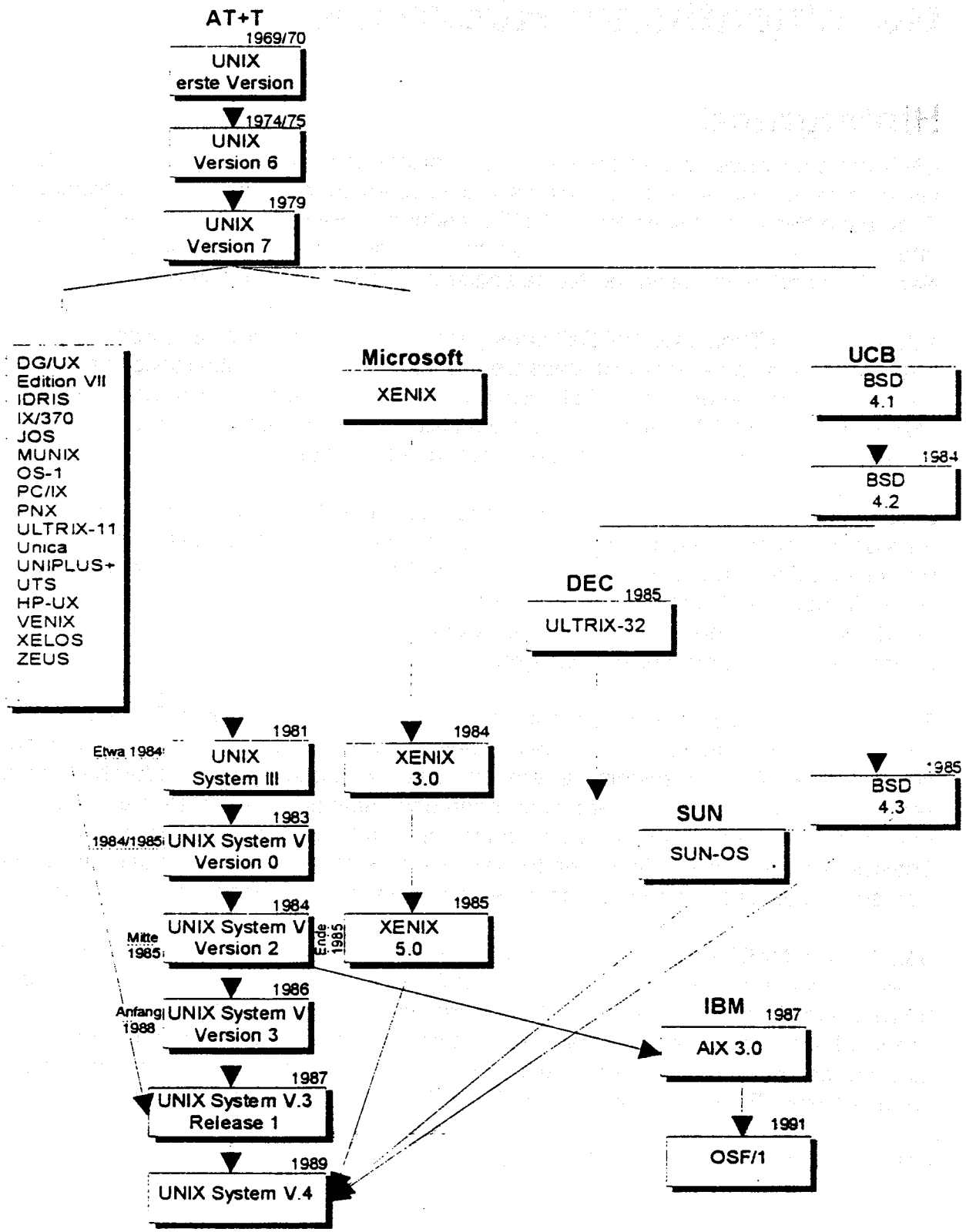
Eine zweite Entwicklung betraf die Sprache, mit der UNIX geschrieben wurde. Die erste Version war noch in Assembler geschrieben. Es war allen schnell klar, dass beim Portieren auf eine andere Maschine, dass nochmalige Schreiben der ganzen Software keine Freude war. Nach A wie Assembler, über die Sprache B, wurde C mit Strukturen in der Sprache und einem hervorragenden Compiler geschrieben.
C war die richtige Sprache zur richtigen Zeit.

Sehr bald interessierten sich verschiedenste Universitäten für dieses Betriebssystem, weil es zu einem minimalen Preis samt Quellcode abgegeben wurde. Somit konnten alle die wollten am Betriebssystem basteln, und das wurde auch getan. Dadurch entstanden sehr schnell und sehr viele Verbesserungen und damit auch sehr viele Versionen. Die Verwendung der verschiedenen Versionen von UNIX führte zu Problemen bei Entwicklern, die ihre Applikationen für verschiedene Plattformen unter UNIX entwickeln wollten. Um diese Probleme zu lösen, wurden verschiedene Standards entwickelt.

Seit Beginn der 90-er Jahre hat sich das modular aufgebaute Betriebssystem UNIX zu einem mächtigen, flexiblen und vielseitigen Betriebssystem entwickelt. Es dient als Betriebssystem für alle Arten von Computern, vom Einplatzsystem und Arbeitsplatzrechner über Mehrbenutzersysteme. Minicomputer und Mainframes bis hin zu Supercomputern. Die Anzahl der Computer, auf denen UNIX läuft, ist explosionsartig angewachsen. Es wird erwartet, dass dieses Wachstum weiterhin anhält.

UNIX System V.4 ist der aktuelle Standard.

Entwicklungsrichtungen von UNIX Systemen



Ziele von UNIX

UNIX ist ein interaktives Time-Sharing System. Es wurde von Programmieren für Programmierer entworfen, die in einer Umgebung arbeiteten, in der die Mehrheit der Benutzer schon relativ vertraut mit Softwareprojekten waren. Damit die vielen Programmierer gemeinsam an einem grösseren Projekt arbeiten konnten, müssen viele Hilfsmittel vorhanden sein, damit Informationen sinnvoll gemeinsam benutzt werden konnten.. Diese erfahrenen Programmierer, die sehr eng zusammenarbeiten, um komplexe Software zu erstellen, haben natürlich eine andere Vorstellung eines Betriebssystemes , als ein einzelner Anfänger, der alleine mit einem Textverarbeitungssystem übt.
Dieser wichtige Unterschied zieht sich von Anfang bis Ende in UNIX durch.

Was wünschen sich nun erfahrene Programmierer von einem System?
Das System sollte auf jeden Fall einfach, elegant und stabil sein. Eine Datei sollte auf jeden Fall nur eine Sammlung von Byte sein. Weiter sollte das System eine grosse Leistungsfähigkeit und Flexibilität aufweisen. Das ist nur zu lösen, wenn ein System aus wenigen Basiselementen besteht, diese aber unbeschränkt miteinander kombinierbar sind. Eine grundlegende Absicht von UNIX ist, dass ein Programm genau eine Aufgabe lösen soll, diese aber perfekt.

Ferner neigen Programmierer zu Minimalismus. Warum soll man z.B. bei einem Befehl copy eingeben, wenn `cp` genügt?

Sie können sich vorstellen, dass gerade UNIX all diese Bedürfnisse sehr gut unter einen Hut bringen kann, und darum auch heute noch die verbreitetste Plattform für Neuentwicklungen ist.

Heute (1996) wird UNIX bei kleineren Systemen (bis max. 4 Prozessoren) die keine grossen Ansprüche an die Sicherheit stellen, zunehmend von Windows NT bedrängt. Hingegen bei grösseren Systemen wird UNIX auch in Zukunft das Betriebssystem der ersten Wahl bleiben.

Aufgaben eines Betriebssystems

Ohne Software ist ein Computer nichts weiter als ein Klumpen Metall. Erst mit Software kann ein Computer Informationen verarbeiten, speichern und wieder finden, Tippfehler finden, Adventure Games spielen, und sich weiteren wertvollen Aktivitäten widmen.

Wie sie wissen, gibt es zwei Arten von Software:

- Systemprogramme, die der Verwaltung des Betriebs eines Computers selber dienen. und
- Anwendungsprogramme, die die Probleme ihres Benutzers lösen.

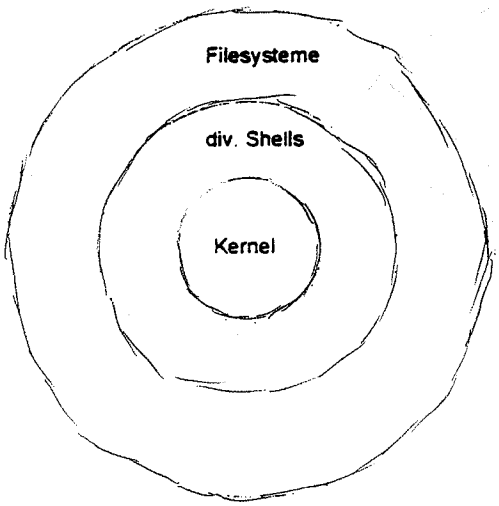
Wir werden uns hier vor allem auf die Systemprogramme beschränken.

Das Betriebssystem eines Computers verwaltet Ressourcen wie Speicher, Prozesse, Dateisysteme. Es besteht aus einem Systemkern (Kernel), einer Schnittstelle zum Bediener (Shell) und enthält die Verbindung zu Elementen, welche die Daten in geordneter Form so ablegen lassen, damit diese auch wieder gefunden werden können. Daneben befindet sich um das System herum eine Ansammlung von Dienst- und Systemprogrammen.

Die Komponenten von UNIX

Unter dem Begriff UNIX versteht man üblicherweise den gesamten Vorrat von Betriebssystemfunktionen, Systemprogrammen und manchmal auch Anwenderprogrammen wie Editoren, Compiler und Kommandointerpreter.

Die verschiedenen Ebenen von UNIX:



UNIX ist aus den folgenden 3 Schichten aufgebaut:

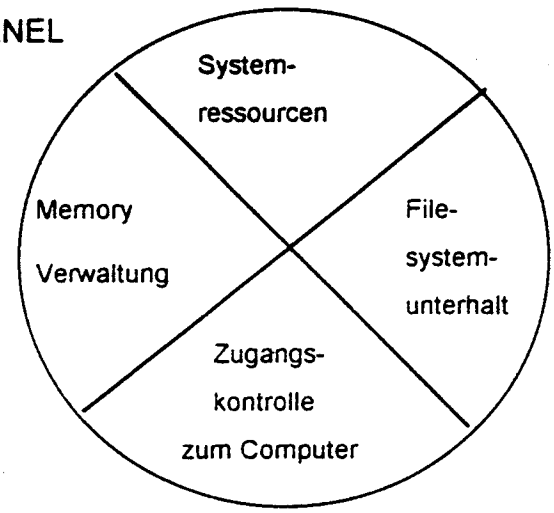
1. **Kernel** (Betriebssystemkern)
2. **Shell** (Kommandointerpreter, Programme, Tools, Utilities)
3. **Filesystem** (Verwaltung der Dateien)

Die einzelnen Teile werden nachfolgend genauer betrachtet.

Der Kernel

Das Herz von UNIX. Die folgende Abbildung zeigt die Hauptaufgaben des Kernels. Der Kernel ist eine Software, die den Zugang zum Computer kontrolliert, das Memory verwaltet und den einzelnen Benutzern Systemzeit zur Verfügung stellt.

Der KERNEL



Für den Normalbenutzer geschehen diese Funktionen alle automatisch. Einzelheiten. wie der Kernel intern im Detail funktioniert, bleiben den wirklichen Hackern vorbehalten.

Falls Sie sich diese Kenntnisse trotzdem erwerben möchten, sei das Buch von Maurice J. Bach, 'Wie funktioniert das Betriebssystem UNIX? empfohlen. In diesem Buch sind die Mechanismen und in die funktionellen Einheiten des Systems beschrieben. Um das Buch verstehen zu können, sind sehr gute Kenntnisse der C-Sprache nötig, da UNIX bis auf einen kleinen Teil (ca. 5%) vollständig in der Programmiersprache 'C' geschrieben ist.

Die Shell

Die Shell ist ein UNIX Systemprogramm von zentraler Bedeutung für die Eingabe von Befehlen und Ausgabe von Daten. Es ist das Bindeglied zwischen Benutzer und Kernel.

Wird ein Befehl über die Tastatur eingegeben, so übersetzt die Shell den Befehl in eine Sprache, die der Kernel versteht. Auf Grund der Aufgabe als Übersetzer, nennt man die Shell auch Befehlsinterpreter.

Zusätzlich zur Aufgabe als Befehlsinterpreter, stellt die Shell eine komplette Programmiersprache mit Variablen und Flusskontrollstrukturen zur Verfügung. Im Verlauf dieses Semesters werden Sie mit Hilfe der Shell verschiedene Programme schreiben.

Es sind im UNIX V.4 standardmässig vier Shells implementiert:

- **Bourne Shell**
- **C-Shell**
- **Korn Shell**
- **Job Control Shell**

Bourne Shell (`/sbin/sh`) - geschrieben von Steve Bourne bei den Bell Laboratories: auch bekannt als Bell Shell oder System V Shell. Dies ist die **Standardshell**. Programme die mit dieser Shell geschrieben sind, müssten auf sämtlichen UNIX Systemen lauffähig sein.

C Shell (`/usr/bin/csh`) - Kommandointerpreter aus der BSD (Berkeley Software Distribution) UNIX Version. Sie enthält Möglichkeiten, Kommandos zu repetieren, zu editieren und Alias-Funktionen zu setzen; zugeschnitten auf eine C-Programmierungs-umgebung.

Korn Shell (`/usr/bin/ksh`) - geschrieben von David Korn bei den Bell Laboratories. Sie vereint die Funktionalität der Bourne Shell und der C Shell, enthält Möglichkeiten, Kommandos zu repetieren, zu editieren und Alias-Funktionen zu setzen und enthält auch Job Control Funktionen. Dies ist die meistverwendete Shell.

Job Control Shell (`/sbin/jsh`) - gleich wie Bourne Shell; zusätzlich: POSIX (**P**ortable **O**perating **S**ystem, based on **UNIX**) konforme Implementierung für Jobsteuerung. Bietet die Möglichkeit, Jobs zwischen Background und Foreground zu verschieben, anzuhalten und diese wieder zu reaktivieren.

Das Filesystem

Unter einem Filesystem versteht man die Art und Weise, wie die Dateien verwaltet werden. Ähnlich wie bei DOS sind bei UNIX die Dateien in einem hierarchischen Filesystem verwaltet. UNIX hat aber im Gegensatz zu DOS die Möglichkeit, je nach Bedarf die entsprechenden Dateien auf verschiedene Arten zu verwalten.

Unter UNIX gibt es nur drei verschiedene Arten von Dateien:

- **Ordinary Files** (gewöhnliche Dateien)
- **Directory Files** (Ordner Dateien, Verzeichnis)
- **Special Files** (Spezielle Files, Geräte)

Ordinary Files

Diese gewöhnlichen Dateien enthalten oft die gespeicherten Informationen des Benutzers. Definitionsgemäss ist eine gewöhnliche Datei nichts anderes als eine Folge von Bytes, die zu einer Einheit zusammengefasst sind. Das Heisst somit auch, dass diese Dateien weder ein Startbit noch ein End of File Zeichen haben.

Directory Files

Sie dienen nur zur besseren Übersicht und zur Gliederung der Dateien.

Special Files

Jedes Gerät, wie z. B der Drucker, die Harddisk, das Memory, etc. wird im UNIX durch ein Special File angesprochen.

Um eine Datei auszudrucken, ins Memory zu laden oder auf einer Harddisk zu speichern sind genau die gleichen Vorgänge nötig. UNIX schreibt die Informationen einfach in ein File. Je nachdem werden die Informationen dann gespeichert oder eben ausgedruckt.

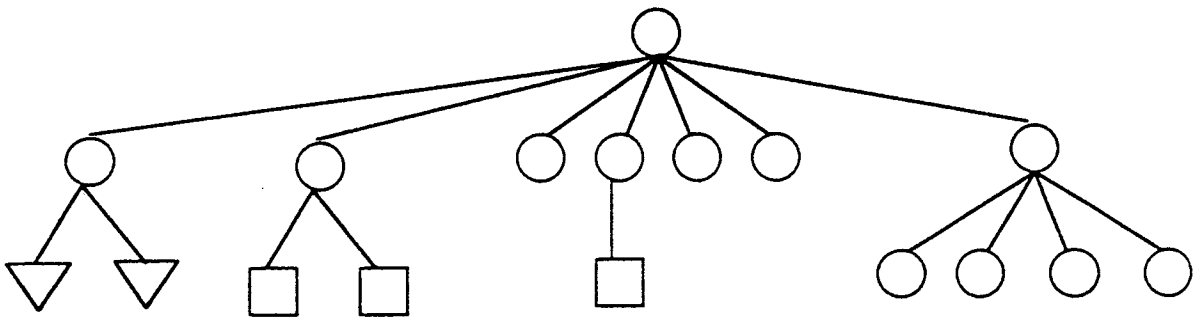
Filesystemhierarchie

Falls es möglich wäre das Filesystem anzusehen, würde es wie ein umgekehrter Baum oder wie ein Organigramm aussehen.

= Ordinary Files

= Directories

= Special Files



Im Unterschied zu DOS gibt es im UNIX nur ein einziges ROOT Verzeichnis, egal wieviele Disks ein System enthält.

Die einzelnen Pfade sind im Unterschied zu DOS mit einem einfachen Schrägstrich / abgetrennt.

Der Einstieg ins UNIX

Wie Sie bereits gesehen haben, ist UNIX ein Multiuser System. Dabei können bis zu mehreren hundert Benutzern gleichzeitig an einem UNIX System arbeiten. Der Zugang erfolgt entweder über das Netzwerk (oft TCP/IP, das UNIX Protokoll) oder über einen seriellen Zugang.

Damit die einzelnen Benutzer vom System unterschieden werden können, müssen sich alle Benutzer mit einem Loginnamen und einem Passwort am System anmelden.

Der Benutzer muss dem System bereits bekannt sein.

Der Login

Zutritt immer über Login und Passwort:

```
Login:      .....
Password:   .....
```

Regeln zum `Login Namen:`

1. Der Loginname muss in `kleinbuchstaben` eingegeben werden. UNIX unterscheidet GROSS und Kleinbuchstaben
2. Falls Grossbuchstaben eingegeben werden, wird UNIX nur in Grossbuchstaben antworten, bis sich der Benutzer vom System abmeldet und wieder neu einloggt hat.

Regeln zum Password:

1. 6 bis 8 Zeichen
2. Mindestens ein numerisches Zeichen
3. Mindestens zwei alphabetische Zeichen
4. Vorsicht bei Verwendung von Spezial Zeichen
5. bei einem Passwortwechsel muss sich das neue Passwort vom alten Passwort in mindestens 3 Zeichen unterscheiden
6. Nur der Superuser ist berechtigt, ein beliebiges Passwort oder keines zu setzen.

Der Superuser (root) ist ein Benutzer dem sämtliche Rechte auf dem System gewährt werden. Das Passwort des Superusers muss somit besonders sorgfältig verwaltet werden.

Beispiel eines erfolgreichen Logins:

```
Welcome to the Pentagon

login: rmuggli
Password:

UNIX System V/386/486 Release 4.0 Version 2.0
Copyright (C) 1984,1986, 1987, 1988, 1989, 1990 AT&T
Copyright (C) 1987, 1988 Microsoft Corp.
All Rights Reserved
Last Login: Thu Nov 12 10:34:17 from TBZ

/         : Disk space:329.04 MB of 474.08 MB avail (64.41%)
/proc     : Disk space: 0.00 MB of   0.00 MB avail ( 0.00%)
/dev/fd   : Disk space: 0.00 MB of   0.00 MB avail ( 0.00%)
/stand    : Disk space: 2.95 MB of   9.99 MB avail (29.25%)
/proc     : Disk space:387.13 MB of 611.89 MB avail (63.27%)

Total Disk Space : 719,13 MB of 1095,98 available (65,62%)

*************************************************************
Welcome to the most secure engine
*************************************************************

/home/rmuggli  $ _
```

Nach erfolgreichem Login erscheint das Systemprompt $. Falls Sie als Superuser eingeloggt sind, erhalten Sie #.

Einige beteiligte System Komponenten zum Login

Im Gegensatz zu vielen anderen Betriebssystemen hat UNIX den Vorteil, dass viele Dateien lesbar sind. Am Login Vorgang sind folgende Dateien beteiligt:

Das System File `/etc/passwd`:

more /etc/passwd

← bedeutet Administrator

```
root:x.0:1:000-Admin(0000):/:
rmuggli:x:103:120:Lehrer:/home/rmuggli:/usr/bin/ksh
```

1 2 3 4 5 6 7

1. Feld:	`rmuggli`	Loginname
2. Feld:	`x`	Platzhalter (Überbleibsel aus UNIX V.3 Version)
3. Feld:	`103`	UID = User Identifikation (für alle Superuser UID = 0)
4. Feld:	`120`	GID = Gruppen Identifikation
5. Feld:	`Lehrer`	Kommentar
6. Feld:	`/home/rmuggli`	HOME - Directory (nicht zu verwechseln mit `/home`)
7. Feld:	`/usr/bin/ksh`	Programm, welches für diesen Benutzer gestartet werden soll. Falls dieser Eintrag fehlt, wird die Bourne Shell gestartet.

Das System File /etc/shadow:

Diese File darf aus Sicherheitsgründen nur der Superuser (root) ansehen.

```
root:NsF5.ofHLRD.1:8514:0:168:7:::
rmuggli:ZFKL6.pkdfg.z:8623:0:168:7:  :  :
   1          2          3   4  5  6  7  8  9
```

1. Feld:	`rmuggli`	Loginname
2. Feld:	`ZFKL...`	Passwort, immer auf 13 Stellen verschlüsselt
3. Feld:	`9623`	letzte Änderung (Anzahl Tage seit 1.1.1970)
4. Feld:	`0`	Minimum Anzahl Tage, nach denen das Passwort geändert werden kann (0 = Passwort darf immer geändert werden)
5. Feld:	`168`	Maximum Anzahl Tage der Gültigkeit des Passwortes
6. Feld:	`7`	Anzahl Tage für die Warnung, dass das Passwort ungültig wird
7. Feld:	-	Anzahl Tage wie lange der Benutzer inaktiv sein darf
8. Feld:	-	Datum ab wann der Login nicht mehr benutzt werden kann
9. Feld:	-	Für die Zukunft reserviertes Flag, auf Null gesetzt

Das System File /etc/group:

Jeder Benutzer muss einer bestimmten Gruppe angehören.

```
root: :0:root
Tbz: :120:gast1,gast2
  1  2   3      4
```

1. Feld:	`Tbz`	Gruppenname
2. Feld:		leer
3. Feld:	`120`	Gruppennummer
4. Feld	`gast1,gast2`	zusätzliche Benutzer (nur zusätzliche Gruppenmitglieder werden einzeln aufgeführt)

Logout

Um sich vom System abzumelden bestehen 2 Möglichkeiten:

1. Direkt nach dem Systemprompt `^d` eintippen. (<Control> + d).

oder

2. Direkt nach dem Systemprompt das *Befehl* `exit` eintippen.

Die Befehlssyntax

Wie Sie sicher auch bald feststellen werden, gibt es im UNIX leider keine allgemeingültige Regeln zu Befehlen, da die einzelnen Programme (Befehle) von zu vielen verschiedenen Personen geschrieben wurden.
Trotzdem ein paar Anhaltspunkte:

1. Befehle werden in UNIX oft klein geschrieben.
2. Optionen beginnen mit einem Minuszeichen, direkt nach dem Befehl.
3. Der Dateiname wird, falls nötig, am Schluss geschrieben.

Beispiel:

```
$ ls -l brief
```

`ls`	=	Befehl
`-l`	=	Option
`d.brief`	=	Filename

Oft sind Befehle in Kleinbuchstaben geschrieben. UNIX berücksichtigt die Gross- und Kleinschreibung.

Abbruch von Befehlen

Befehle die unendlich lange Ausgaben erzeugen, können mit folgenden Tastenkombinationen abgebrochen werden:

^c (<Control> + c)

oder über die

Delete - Taste (je nach Tasten Definition)

man command ¦ more

Auf dem System kann zu jedem Befehl mit Hilfe des Befehls `man` ein Online-Hilfetext abgerufen werden. Diese Manual ist sehr umfangreich und beinhaltet alles nur Erdenkliche zu den einzelnen Befehlen. Das Manual ist in verschiedene Sektionen unterteilt. Die meistgebrauchten Benutzerbefehle sind in der Sektion 1enthalten.

Beispiele:

`$ man passwd | pg`	Hilfe zum Befehl `passwd`, seitenweise Anzeige
`$ man -s1 passwd | pg`	Hilfe zum Befehl `passwd` seitenweise Anzeige, beginnend mit der Sektion 1.
`$ man who | more`	Hilfe zum Befehl `who`, ebenfalls seitenweise Anzeige
`$ man date who | more`	Hilfe zu den Befehlen `date` und `who`, seitenweise Anzeige

Das Manual für die Benutzerbefehle liegt in Buchform unter folgenden Titel vor:

- UNIX SVR4 MP User's/System Administrator's Reference Manual (a-l)
- UNIX SVR4 MP User's/System Administrator's Reference Manual (m-z)

NAME

write - write to another user

SYNOPSIS

write *user* [*line*]

DESCRIPTION

write copies lines from your terminal to that of another user. When first called, it sends the message:

Message from *yourname* **(tty??)** [*date*]...

to the person you want to talk to. When it has successfully completed the connection, it also sends two bells to your own terminal to indicate that what you are typing is being sent.

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal, an interrupt is sent, or the recipient has executed "mesg n". At that point **write** writes EOT on the other terminal and exits.

If you want to write to a user who is logged in more than once, the **line** argument may be used to indicate which line or terminal to send to (e.g., **term/12**); otherwise, the first writable instance of the user found in **/var/adm/utmp** is assumed and the following message posted:

user is logged on more than one place.
You are connected to "*terminal***".**
Other locations are:
terminal

Permission to write may be denied or granted by use of the **mesg** command. Writing to others is normally allowed by default. Certain commands, such as the **pr** command, disallow messages in order to prevent interference with their output. However, if the user has super-user permissions, messages can be forced onto a write-inhibited terminal.

If the character ! is found at the beginning of a line, write calls the shell to execute the rest of the line as a command.

write runs **setgid()** [see **setuid(2)**] to the group ID **tty**, in order to have write permissions on other user's terminals.

write will detect non-printable characters before sending them to the user's terminal. Control characters will appear as a `^' followed by the appropriate ASCII character; characters with the high-order bit set will appear in metanotation. For example, `\003' is displayed as `^C' and `\372' as `M-z'.

The following protocol is suggested for using **write**: when you first **write** to another user, wait for them to **write** back before starting to send. Each person should end a message with a distinctive signal (i.e., (o) for ``over'') so that the other person knows when to reply. The signal (oo) (for ``over and out'') is suggested when conversation is to be terminated.

FILES

/var/adm/utmp	to find user
/usr/bin/sh	to execute !

SEE ALSO

mail(1), **mesg**(1), **pr**(1), **sh**(1), **who**(1), **setuid**(2).

DIAGNOSTICS

user is not logged on	if the person you are trying to **write** to is not logged on.
Permission denied	if the person you are trying to **write** to denies that permission (with **mesg**).
Warning: cannot respond, set mesg -y	if your terminal is set to **mesg n** and the recipient cannot respond to you.
Can no longer write to user	if the recipient has denied permission (**mesg n**) after you had started writing.

Aktive Benutzer abfragen

who [am i]

who -option

Funktion: Identifiziert die aktiven Benutzer auf dem System.

Optionen:

`am i`	Identifiziert nur den Benutzer, der das Terminal momentan benützt
`-H`	Versieht die verschiedenen Ausgabespalten mit einem Titel
`-u`	Zeigt zusätzliche Informationen an, z.B. die verstrichene Zeit seit der letzten Aktivität am Terminal und die Prozess Identifikations Nummer (PID) der Benutzershell

Beispiele:

`$ who -Hu` Wer ist eingeloggt?

```
NAME    LINE    TIME          IDLE     PID     COMMENTS
user01  s02     Nov 22  13:23          .       645    Buchhaltung
user02  s15     Nov 22  13:45          0:04    679    Demo Center
user03  s04     Nov 12  09:44          old     301
```

`PID` Prozess - Nummer.

`IDLE` Die verstrichene Zeit seit der letzten Aktivität auf dem Terminal.
Ein Punkt bedeutet in dieser Minute aktiv, 0:04 bedeutet seit 4 Minuten keine Aktivität des Benutzers, `old` bedeutet seit über 24 Stunden keine Aktivität des Benutzers.

`$ who am i` Als wer bin ich eingeloggt

```
user01  s04     Nov 22   12:32
```

UNIX email

Mail senden

`mail loginname1 loginname2 ...`

Funktion: Senden von Meldungen an andere Systembenutzer

Beispiel:

```
$ mail user01
Kommst Du zum Mittagessen um 12:00?
Gruss user02
<Ctrl> d                    mail - Meldung abschliessen auf einer leeren Zeile
```

Meldungen, die den Empfänger infolge Systemunterbruch nicht erreichen, sind im File `dead.letter` (im Home - Directory) gespeichert.

Mail abrufen

`mail [-option ...]`

Optionen:

`d`	Löschen der angezeigten Meldung und Ausgabe der nächsten Meldung
`q`	Austritt aus mail
`?`	Ausgabe aller Aktionsmöglichkeiten am Bildschirm

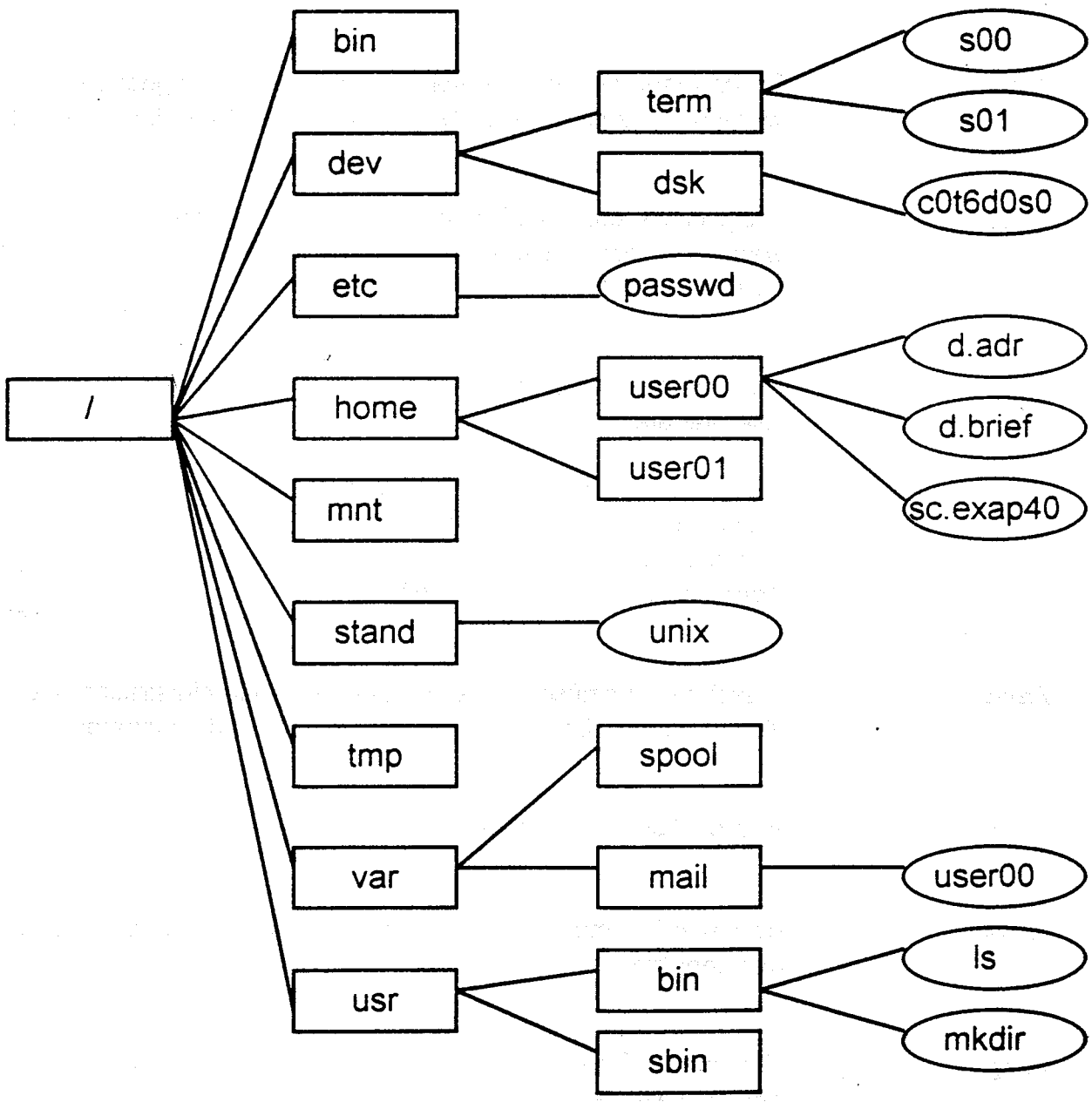
Beispiel:

```
$ mail
From root Thu Sep 30 09:24 GMT 1993
Content-Length:     85

Installation of Volume Manager on guru4 as package instance
<VolMgr>
was successful.
?                           Hier können die Optionen eingegeben werden.
```

Struktur des UNIX Systems

Der Aufbau der Dateistruktur kann bei einzelnen UNIX Varianten leicht anders sein.
Meist liegt aber folgender Grundaufbau vor:



Die wichtigsten Directories unter root:

Directoryname	Inhalt
/	Das root Directory, d.h. das Hauptdirectory für das ganze UNIX Filesystem. Für den Superuser mit dem Loginnamen root ist das Directory root auch das Home- oder Login - Directory. (Bei LINUX `/root`)
/stand	Programme, die beim Booten des UNIX Systems gebraucht werden. Default Eintritts - Directory für das Boot File System (bfs).
/sbin	Programme, die beim Booten und System - Recovery gebraucht werden. Einige dieser administrativen Programme sind auch unter `/usr/bin` oder `/usr/sbin` zu finden.
/dev	Verbindungsfiles zwischen UNIX und den peripheren Einheiten (Gerätedateien).
	`/dev/console` System Konsole `/dev/lp` Line Printer `/dev/term/*` Terminals `/dev/dsk/*` System Disks
/etc	Maschinenspezifische Files für die Systemadministration und Konfiguration. Befehle für den Superuser / Administrator.
/opt	Add-on Applikationspakete
/home	Benutzer Directories und Files. Meist das übergeordnete Directory der Benutzer.
/tmp	Temporäre Files, die beim Anhalten des Systems automatisch gelöscht werden.

`/var`	Enthält verschiedene Directories wie z.B.:	
	`/var/mail`	Mail-Files für Benutzer
	`/var/news`	Anschlagbrett für Neuigkeiten
	`/var/tmp`	grosse systemerstellte temporäre Files, die bei Aufnahme des Mehrbenutzer Betriebes gelöscht werden.
`/usr`	enthält maschinenunabhängige Files, die unveränderlich sind:	
	`/usr/bin`	UNIX Shell Kommandos und Utilities. `/bin` ist symbolisch gelinkt mit `/usr/bin`.
	`/usr/sbin`	Kommandos für Systemunterhalt und den Superuser
	`/usr/lib`	Bibliotheken für Programme und maschinenabhängige Datenfiles
	`/usr/share`	Maschinenunabhängige sharable Files, wie ASCII Datenfiles/-banken
	`/usr/catman`	Online Manual Seiten (Hilfetext zu Befehlen)
`/lost+found`		File-Waisen (z.B. Files ohne Namen nach einem Systemcrash)

Directories

2 Einträge, die immer vorhanden sind:

.	Bezieht sich auf das aktuelle Directory
..	Bezieht sich auf das Parent Directory, d.h. auf das übergeordnete Directory.

Directory wechseln

`cd [directory]`

Funktion: Wechseln von einem Directory in ein anderes. (change directory) Fehlt die Angabe des Argumentes, so wird in das Home - Directory gewechselt.

Beispiele:

```
$ pwd
/home/user01
```
(print working directory), zeigt den aktuellen Standort an

```
$ cd ..
$ pwd
/home
```
eine Directory Stufe Richtung root zurück

```
$ cd /
$ pwd
/
```
zum Root Directory

```
$ cd
$ pwd
/home/user01
```
zum Home - Directory zurück

```
$ cd -
```
wechselt ins vorherige Directory (nur mit der Korn Shell)

Pfadnamen

Alle Files und Directories im System können über einen Pfadnamen (engl. pathname) aufgefunden werden. Dieser Pfadname kann durch mehrere Directories führen.

```
/home/rmuggli/brief
```

Auch in UNIX werden 2 Arten von Pfadnamen unterschieden:

Absolute Pfadnamen

Der absolute Pfadname gibt den Ort des Files oder Directories immer von der root aus gesehen an. Absolute Pfadnamen beginnen also immer mit einem '/'.

Beispiele:

```
/home/rmuggli
/var/mail
```

Relative Pfadnamen

Diese Art der Angabe wird oft verwendet , wenn man das übergeordnete Directory ansprechen möchte. In solchen Fällen ist die Angabe des relativen Pfadnamens kürzer als die Angabe des absoluten Pfadnamens.

Beispiele:

```
../..
../rmuggli
```

Filenamen

- Bis max. 256 Zeichen (in einem `ufs` Filesystem)
- Klein und Grossbuchstaben werden unterschieden !
- Filenamen werden meist klein geschrieben
- Falls Leerschläge verwendet werden, muss der Filename in " " gesetzt werden.

Directory Inhalt anzeigen

```
ls [-option ...] [file ...] [directory ...]
```

Funktion: Listet Files und Directories in einer alphabetisch sortierten Reihenfolge. Mit entsprechenden Optionen werden auch diverse File- oder Directory-Attribute wie Zugriffsberechtigungen, Modifikationsdatum, Eigentümer, etc. angezeigt.

Optionen:

`-l`	Ausgabe im Langformat, d.h. Zugriffsberechtigungen, Anzahl Links. Eigentümer, Gruppe, Grösse in Bytes und letztes Modifikationsdatum.
`-a`	Alle Eintragungen, auch "." und "..", sowie die mit einem Punkt beginnenden Files
`-s`	Grösse in Anzahl Blocks zu 512 Bytes
`-d`	Für ein Directory, nur den Directorynamen ausgeben, nicht dessen Files
`-u`	Datum des letzten Zugriffs
`-R`	Ausgabe aller Subdirectories und der darin enthaltenen Files
`-i`	i-Node Nummer für jedes File.

Beispiel:

`$ ls -l` gibt eine lange Ausgabe der Files

```
-|rwx|rw-|r--|   1 rmuggli     tbz    58   Oct 17 15:23 file1
```

Art des Files:

- `-` gewöhnliches File
- `d` Directory
- `l` symbolisch gelinktes File oder Directory
- `b` Block orientiertes Special File
- `c` Charakter orientiertes Special File
- `p` Pipe File (fifo special file)

Zugriffsberechtigungen: (siehe nächste Seite)

read, write und execute Erlaubnis für	Eigentümer
read und write Erlaubnis für	Gruppenmitglieder
read Erlaubnis für	alle anderen

`1` Anzahl Links des Files. Bei Directories sind hier die Anzahl der direkt darunterliegenden Subdirectories gemeint.(Abzüglich die Einträge "." und "..")

`rmuggli` Eigentümer = Loginname des Erstellers

`tbz` Gruppe zu der der Ersteller gehört

`58` Grösse in Anzahl Bytes

`Oct 17 15:23` Datum der letzten Änderung. Liegt das Änderungsdatum um mehr als 6 Monate zurück, so wird anstelle von Tag und Zeit, das Jahr der letzten Änderung eingesetzt.

`file1` Filename (Dateiname)

Zugriffsberechtigungen

Im Mehrbenutzersystem UNIX werden die einzelnen Dateien auf folgende Weise vor unberechtigten Benutzern geschützt:

Es bestehen die drei Grundrechte

- **Leseerlaubnis** `r`
- **Schreiberlaubnis** `w`
- **Ausführerlaubnis** `x`

(oder bei Directories die Eintrittserlaubnis)

Damit die Berechtigungen einfacher vergeben werden können, sind den einzelnen Berechtigungen Zahlen zugeordnet:

`r=4`
`w=2`
`x=1`
Keine Erlaubnis =-

Die Berechtigungen sind bei jeder Datei vorhanden. Somit kann nur eine Datei mit Schreiberlaubnis geändert werden. Auch auf ein Drucker kann nur mit Schreiberlaubnis geschrieben werden.

Zudem werden die Berechtigungen noch in 3 Stufen vergeben:

- Dem Eigentümer
- Der Gruppe
- Allen Anderen

Je nach Eigentümer und Gruppenzugehörigkeit werden die Rechte anders vergeben. Mit dem Befehl `id` kann die eigene Zugehörigkeit überprüft werden:

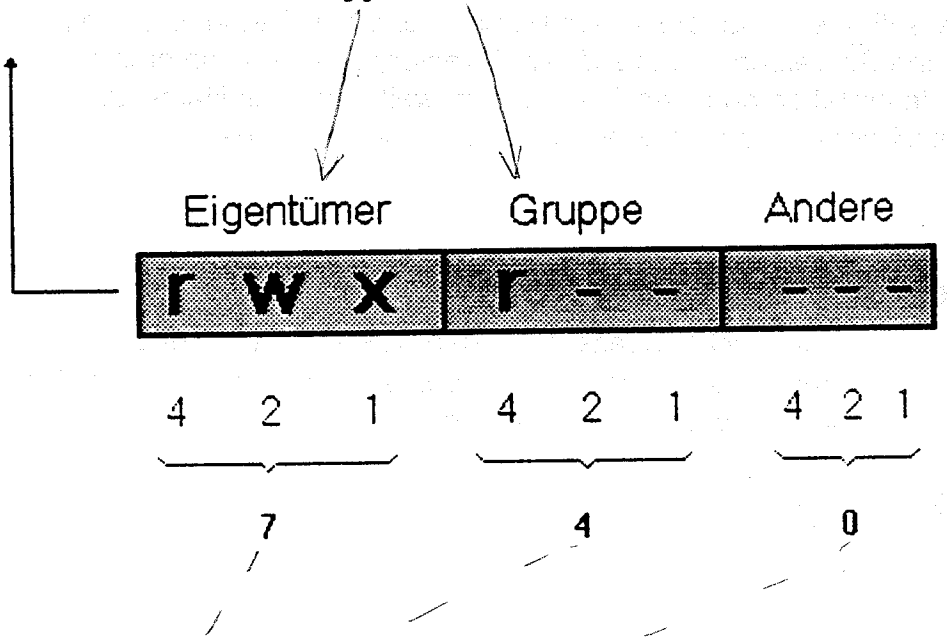
Beispiel:

```
$ id
uid=112(rmuggli), gid=200(tbz)
```

Die Berechtigungen werden am einfachsten mit ls -l überprüft.

Beispiel:

```
$ ls -l
-rwx  r--  ---  1   rmuggli tbz     58  Oct 17 15:23    file1
```



Die Berechtigungen können mit dem Befehl chmod geändert werden

```
$ chmod 0740 file1
```

Regeln:

- Der Eigentümer (Owner) kann seine Berechtigungen uneingeschränkt ändern
- Für den Superuser gibt es keine Beschränkung der Zugriffsberechtigungen!

Konsequenzen von fehlenden Zugriffsberechtigungen

Wichtig ist die Unterscheidung zwischen Zugriffsberechtigung für das File und der Zugriffsberechtigung für das Directory.

Im Directory selbst sind nur die i-node Nummer und der Filename eingetragen. Falls ein Benutzer keine Schreiberlaubnis auf dem Directory besitzt, kann er in diesem Directory weder ein File erstellen noch ein File löschen, weil damit der Name des Files ins Directory geschrieben bzw. im Directory gelöscht werden müsste.

Keine Lese Berechtigung (read)

für File: kann nicht angezeigt werden (z.B. mit `cat` oder `pg`)
kann nicht kopiert, jedoch gelinkt und umbenannt werden
kann nicht gedruckt werden

für Directory: kann Inhalt nicht ausgeben (mit `ls`, `ls -l`)

i-node Filename } ls -ia
i-node Filename

Keine Schreib Berechtigung (write)

für File: kann nicht abgespeichert werden (aus einem Editor)

für Directory: keine Files erstellen, kopieren, linken, umbenennen und löschen

Keine Ausführberechtigung Berechtigung (execute)

für File: ein Shellscript kann nicht ausgeführt werden

für Directory: keine lange Liste kann ausgegeben werden , z.B. mit `ls -l`.
(Aber: nur ls ohne Option geht)
keine Eintrittserlaubnis ins Directory

Default für Zugriffsberechtigung ändern

umask

Funktion: Setzt einen neuen Default (Grundwert) für den eingeloggten Benutzer bezüglich Zugriffsberechtigungen, die allen neu erstellten Files und Directories gegeben werden sollen.

Die bei **umask** angegebenen Werte werden von der Berechtigung des Defaultwertes für Files und Directories abgezogen.

Die Defaultwerte für Zugriffsberechtigungen nach der Systeminstallation sind:

Files:	666 = rw- rw- rw
Directories:	777 = rwx rwx rwx

Mit **umask** gesetzte, neue Werte, sind nur solange gültig, als der Benutzer eingeloggt ist. Für permanente Verfügbarkeit kann **umask** vom System Administrator im File `/etc/profile` systemweit gesetzt werden.

Beispiele:

$ umask 022 Neue Files werden mit den Zugriffsberechtigungen *644* erstellt

```
 666
-022
----------
=644          ⇨   rw-r--r--
```

Neue Directories werden mit den Zugriffsberechtigungen 755 erstellt

```
(777 - 022)        ⇨   (rwx r-x r-x)
```

$ **umask** umask Wert wird angezeigt.
022

Sonderzeichen

Zur Erstellung von Suchmustern stehen die Sonderzeichen zur Verfügung, die in der nachfolgenden Tabelle aufgeführt sind:

Sonderzeichen	Bedeutung
*	Steht für eine beliebig lange, auch leere, Zeichenkette; gilt auch für einen Punkt.
?	Steht für ein einzelnes Zeichen
[ ]	Für alle Zeichen in der Klammer
[!...]	Ohne die Zeichen in der Klammer

Beispiele:

`$ ls *c`	Alle Filenamen im Directory, die mit `.c` enden ausgegeben Auch z.B. `tetris.c` !
`$ ls ???.out`	Alle Filenamen im Directory, die mit `.out` enden und davor drei Zeichen haben
`$ ls [A-Z]*`	Alle Filenamen im Directory, die mit einem Grossbuchstaben von A bis Z beginnen und danach beliebig viele oder keine Zeichen enthalten, werden ausgegeben
`$ ls [!0-9]??[xy]`	Alle Filenmanen im Directory, die nicht mit einer Zahl beginnen, an 2. und 3. Position ein beliebiges Zeichen enthalten und an der 4. Position ein "x" oder "y" haben, werden ausgegeben

Neutralisieren von Sonderzeichen

Falls Filenamen mit Sonderzeichen vorkommen, müssen sie neutralisiert werden. Dazu existieren Sonderzeichen, mit deren Hilfe die spezielle Bedeutung anderer Sonderzeichen neutralisiert werden können.

Zu diesen Sonderzeichen gehören:

Backslash (\)	Einzelne Sonderzeichen neutralisieren
Anführungszeichen ("...")	Mehrere Sonderzeichen neutralisieren, wobei z.B. das Sonderzeichen "$" nicht berücksichtigt wird
Einfache Anführungszeichen (´...´)	Mehrere Sonderzeichen neutralisieren

Beispiele:

Inhalt eines Files bestimmen

`file [-option ...] file ...`

Funktion: Führt eine Reihe von Tests durch, um ein File inhaltlich zu klassifizieren.

Achtung: Für die diversen Tests werden nur die ersten 10 Zeilen einer Datei gelesen .

Beispiel:

```
$ file *          Welcher Art sind die Files im Current Directory
kapitel1:         commands text
game.c:           c program text
d.art:            ascii text
d.art.inx:        data
druck:            empty
exap:             directory
```

Typische Klassifizierungen:

`empty`	File existiert, ist aber leer
`directory`	File ist ein Directory
`english text`	Text mit Gross- und Kleinschreibung. Mehr als 20% der Punktation im File ist von einem Leerschlag oder Newline Zeichen gefolgt.
`ascii text`	Weniger als 20% der Punktation im File ist von einem Leerschlag oder Newline Zeichen gefolgt.
`commands text`	File ist ein Shell Programm
`ELF 32-bit LSB`	Eine ausführbare Code-Datei, executable 80386
`data`	Ein Datenfile (binär)
`c program text`	Text mit Klammerungen entsprechend der C-Syntax

Wichtige UNIX Befehle

Directory erstellen

mkdir [option ...] directory...

Funktion: Erstellt ein oder mehrere Directories.

Optionen:

`-p pathname` Die in *pathname* angegebenen, aber noch nicht existierenden Directories werden dazuerstellt.

Beispiele:

```
$ ls -l
-rwxr--r--   1    rmuggli tbz      36   Oct    4 10:51    a.fil
-rwxr--r--   1    rmuggli tbz      120  Nov  6 12:30     b.fil
```

```
$ mkdir proj1
$ ls -l
-rwxr--r--   1    rmuggli tbz      36   Oct    4 10:51    a.fil
-rwxr--r--   1    rmuggli tbz      120  Nov  6 12:30     b.fil
drwxrwxrwx   1    rmuggli tbz      32   Nov 12 10:45     proj1
```

Im neuen Directory `proj1` werden die Einträge "." und ".." automatisch gemacht.

```
$ pwd
/home/user01

$ mkdir -p tbz/shell
```

Die neuen Directories sind: `/home/rmuggli/tbz`
`/home/rmuggli/tbz/shell`

Directory löschen

`rmdir [-option ] directory ...`

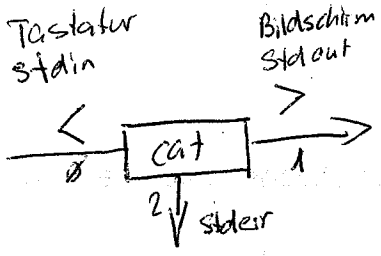
Funktion: Entfernt ein oder mehrere Directories aus einem Filesystem. Es werden nur leere Directories gelöscht.

Beispiele:

`$ rmdir john`	Aktuelles Directory ist: `/home/user01`
`$ rmdir /home/user01/john`	Aktuelles Directory ist z.B. `/home/user13`
`$ rm -r /home/user05`	Löscht im Directory `/home/user05` alle Files, alle Subdirectories inkl. Files, und das Directory `user05` selbst, falls es nicht das aktuelle Directory ist.

Weitere Details siehe unter dem Befehl `rm`.

Inhalt von Files anzeigen



`cat [-option ...] file ...`

Funktion: Gibt den Inhalt eines oder mehrerer Files auf dem Bildschirm aus.

```
$ cat d.adr
```

Bemerkung: Mit `^s` (Control + s) kann die Ausgabe unterbrochen, mit `^q` (Control + q) wieder fortgesetzt werden.
Auch die Bild Stop Taste kann verwendet werden.

`pg [-option .][+linenumber][+/pattern/][file ..]`

Funktion: Gibt den Inhalt eines Files seitenweise aus.

Optionen:

`+n`	Ausgabe ab Zeile `n`
`-p string`	Ändern des `pg`-Promptes von ":" auf `string`. Enthält `string` ein `%d`, so zeigt das neue Prompt die aktuelle Seitennummer an (ideal als Alias definieren)

Beispiele:

```
$ pg +20 file3
```
Das File `file3` wird ab Zeile 20 seitenweise am Bildschirm ausgegeben.

```
$ pg -p "Seite %d:" file[1-9].txt
```
Das `pg`- Prompt wird derart umdefiniert, dass die Seitennummer innerhalb des aktuellen Files angezeigt wird.

Interventionen nach dem Aufruf von pg auf dem Page-Prompt (Default = :):

h	Alle möglichen pg-Befehle auf Bildschirm ausgeben
<RETURN>	Vorwärts um eine Bildschirmseite
q	Beenden des pg-Befehles
-	Zurück um eine Bildschirmseite (ab aktueller)
$	Ausgabe der letzten Seite des Files
+/Zeichenfolge	Ausgabe ab der Zeile, welche die gesuchte Zeichenfolge enthält.
	Nächsten Suchbegriff mit // suchen.
s filename	Das aktuelle File wird unter "filename" abgelegt
! Shell-Kommando	Ausführen eines Shell-Kommandos
m	Zur *m*. Bildschirmseite
+m	*m* Bildschirmseiten vorwärts (ab aktueller)
-m	Zurück um *m* Bildschirmseiten (ab aktueller)

Daneben gibt es noch den Befehl

```
more
```

der sehr ähnlich wie pg funktioniert.

Files löschen

`rm [-option ...] filename ...`

Funktion: Löschen eines oder mehrerer Files.
Löschen von Directories und Subdirectories.

Optionen:

`-i` Für jedes einzelne File muss bestätigt werden, ob es wirklich gelöscht werden soll. Nur bei der Eingabe von `y` wird gelöscht. Bei `n` oder <NL> wird nicht gelöscht.

`-r` Sämtliche Files, Subdirectories und das Directory selbst werden gelöscht, falls das Directory selbst nicht das aktuelle Directory ist.

ACHTUNG !

Nur löschen, wenn man 100 %-ig sicher ist !

Gelöschte Files können nur vom Backup zurückgeholt werden !!

Beispiele:

`$ rm a.*` Löscht alle Files, deren Namen mit a beginnen, ohne Rückfrage!

`$ rm .*` Löscht alle Files, deren Filenamen mit einem Punkt beginnen.

`$ rm *` Löscht alle Files im aktuelle Directory, ausser denjenigen, die mit einem Punkt beginnen.

```
$ rm -r /home/user01/user011
```

Löscht alle Files, Subdirectories und dann das Directory selbst, falls `user011` nicht das aktuelle Directory ist.

```
$ rm -ri /home/user01/user011
```

Wie oben, aber mit Abfrage ob gelöscht werden soll.

Die Sache mit den i-nodes

Jedes normale File und jedes Directory besteht aus 3 verschiedenen Komponenten:

1. dem Eintrag im Directory (i-node Nummer und Filename)
2. dem i-node selbst
3. Datenblocks auf Disk oder Diskette

Der Inhalt eines i-nodes kann normalerweise nicht ersichtlich gemacht werden

Ausgabe der i-node Nummer:

```
$ ls  -i  d.brief                    i-node Nummer anzeigen
370       d.brief
```

i-node 370

Eigentümer
Gruppenzugehörigkeit
Dateityp
Zugriffsrechte
Dateigrösse
Adressen der Daten
Anzahl Links
Datum der letzten Modifikation
Datum des letzten Zugriffs
Datum der letzten I-Node Änderung

Files kopieren

```
cp [-i] [-p] file1 file2
cp [-i] [-p] file1 [ ... ] directory
cp [-i] [-p] -r directory1 directory2
```

Funktionen:

- Kopieren eines oder mehrerer Files in ein Directory.
- Kopieren eines ganzen Directorybaumes in ein anderes Directory.

Optionen:

`-i`	Falls auf ein bestehendes File kopiert wird, wird eine Bestätigung verlangt. Nur falls die Antwort ein "y" ist, wird das bestehende File überschrieben.
`-r`	Erlaubt das Kopieren eines Directories inkl. Subdirectories und Files (OHNE Special Files!) in ein anderes Directory, auf einen anderen Ast.
`-p`	Falls auf ein bestehendes File kopiert wird, werden die Zugriffsberechtigungen und die Modifikationszeit des Original-Files übernommen, nicht aber der Eigentümer.

Bemerkungen:

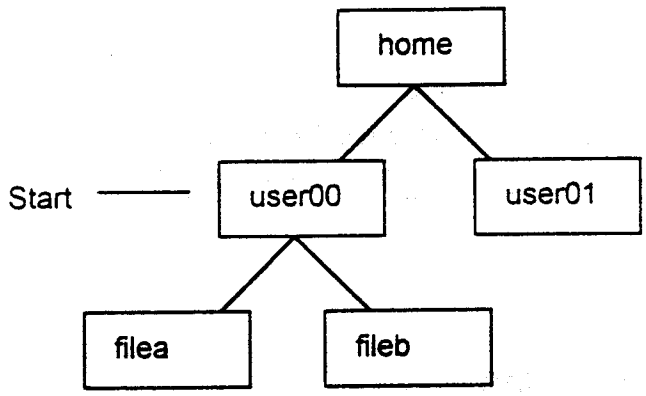
Kopie auf ein bestehendes File überschreibt dieses inhaltlich. Die Zugriffsberechtigungen, Eigentümer und Gruppe werden dabei nicht geändert, d.h. sie werden vom bestehenden File übernommen. Die Modifikationszeit der Kopie ist diejenige, zu der die Kopie gemacht wurde.

Besteht `file2` noch nicht, erhält es die Zugriffsberechtigungen von `file1`. Eigentümer und die Gruppe von `file2` sind diejenigen des Benutzers, der die Kopie macht.

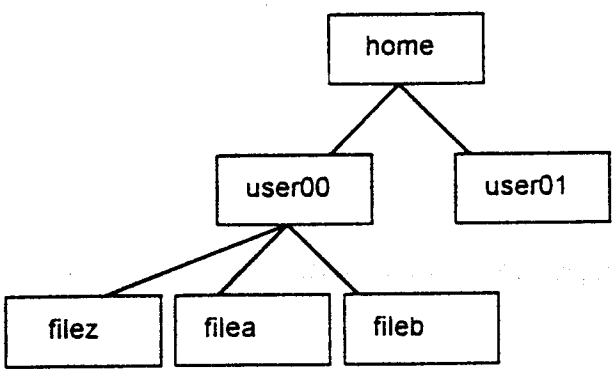
Wird ein `directory1` in ein `directory2` kopiert, welches bereits existiert, so wird das ganze `directory1` als Struktur darin erstellt.

Beispiele:

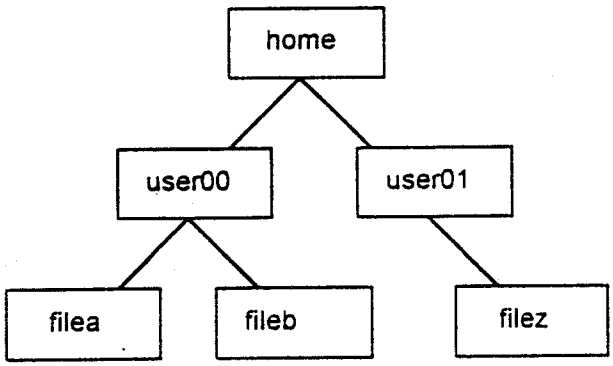
Die folgenden Beispiele beziehen sich auf diese Struktur:



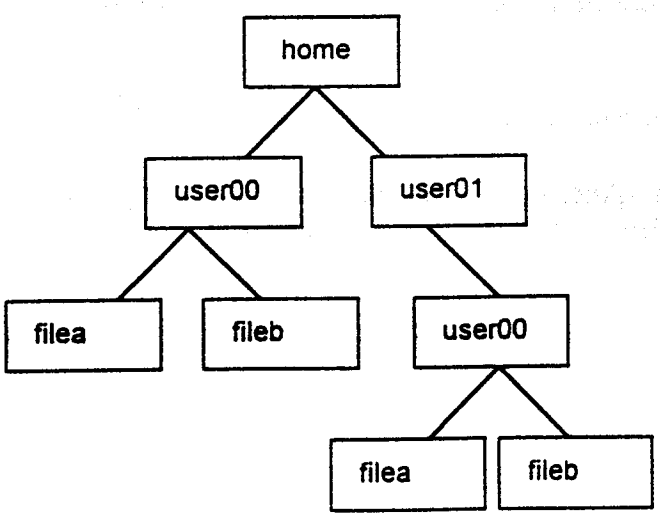

```
$ cp filea filez
```



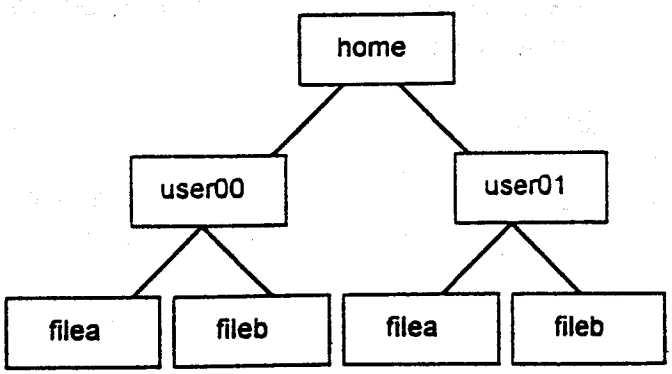

```
$ cp filea ../user01/filez
```




```
$ cd /home
$ cp -r user00 user01
```




```
$ cp filea fileb ../user01
```



Files umbenennen / verschieben

```
mv [-f] [-i] file1 file2
mv [-f] [-i] file1... directory1
mv [-f] [-i] directory1 directory2
```

Funktion:

Umbenennen eines Files im aktuellen Directory.

Verschieben eines Files von einem Directory in ein anderes.

Umbenennen eines Subdirectories im aktuellen Directory oder Verschieben in ein neues Directory.

Optionen:

`-i`	Verlangt eine Bestätigung, falls ein File oder Directory auf ein bereits bestehendes File bzw. Directory umbenannt wird.
`-f`	Allfällige (Fehler-)Meldungen werden unterdrückt.

Bemerkungen:

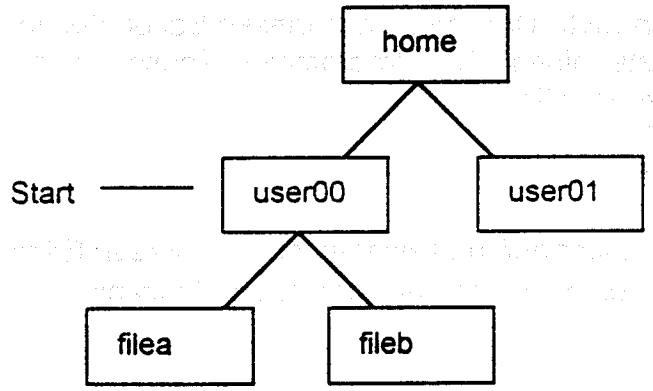
Wird ein File in ein anderes Directory verschoben, kann es umbenannt werden oder seinen Originalnamen behalten.

Existiert `directory2` nicht, so wird `directory1` zu `directory2` umbenannt.

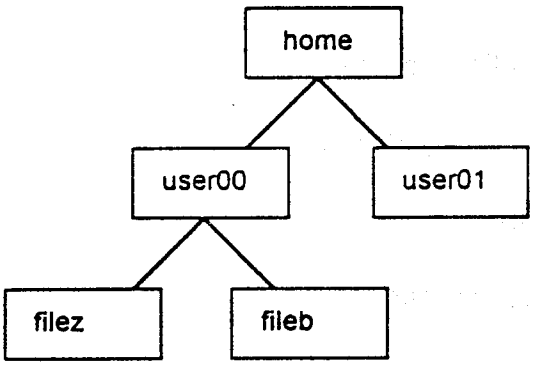
Befinden sich Input- und Outputfile nicht im gleichen Filesystem, so erfolgt das Verschieben wie ein Kopiervorgang mit anschliessendem Löschen des Inputfiles.

Beispiele:

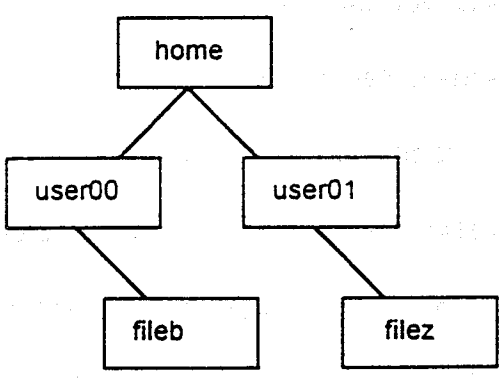
Die folgenden Beispiele beziehen sich auf diese Struktur:



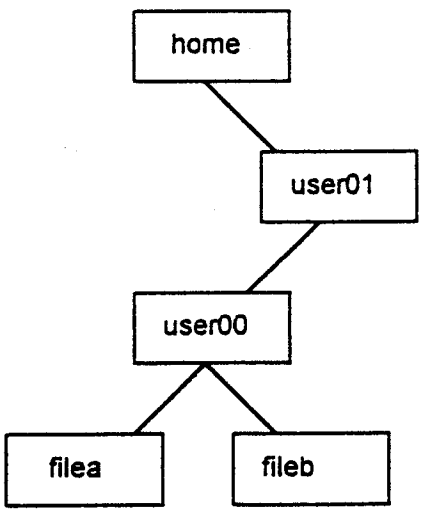

```
$ mv filea filez
```



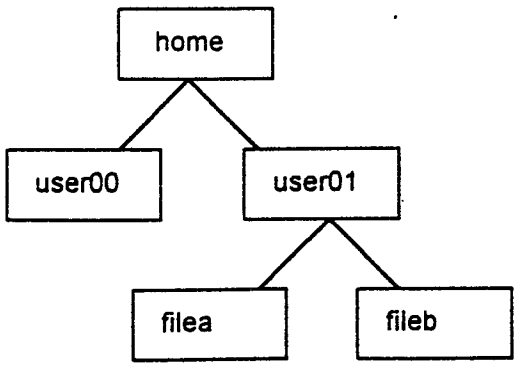

```
$ mv filea ../user01/filez
```




```
$ cd /home
$ mv user00 user01
```




```
$ mv filea fileb ../user01
```



vi : Der Full Screen Editor

Dies ist der Stolz von allen UNIX Hackern. Ohne diesen Editor läuft in UNIX gar nichts. Er ist der typischste Vertreter eines UNIX Programmes. Tausende von Möglichkeiten und ziemlich gewöhnungsbedürftig.

Darum hier das Wichtigste:

Der `vi` - Befehl ruft einen Editor auf, und zwar einen Full Screen Editor. Einige Eigenschaften des `vi` - Befehles werden nachfolgend aufgezählt:

- ca. 200 Anweisungen
- Davon ca. 10-20 für häufigsten Gebrauch
- Max. Zeilenlänge 1023 Zeichen und Newline Zeichen
- Arbeitsfile nicht transparent für Benutzer (`/tmp/Ex...`)
- Recoverymöglichkeit nach System-Crash
- Mehrere Files können gleichzeitig editiert werden
- Setzen von Optionen und Abkürzungen
- Temporärer Rücksprung in den UNIX-Shell

Starten des `vi`

Es bestehen folgende Möglichkeiten, den `vi` zu starten:

`$ vi filename`	Editieren eines existierenden Files oder ein neues File erstellen und gleichzeitig einen Namen vergeben
`$ vi file1 file2`	Editieren von mehreren Files
`$ vi file[0-9]`	Editieren von allen Files `file0,file1.., file9`
`$ vi -r filename`	Recover eines Files nach einem System-Crash

Wird ein neues File mit dem Befehl `vi` erstellt, so erhält man folgendes Bild:

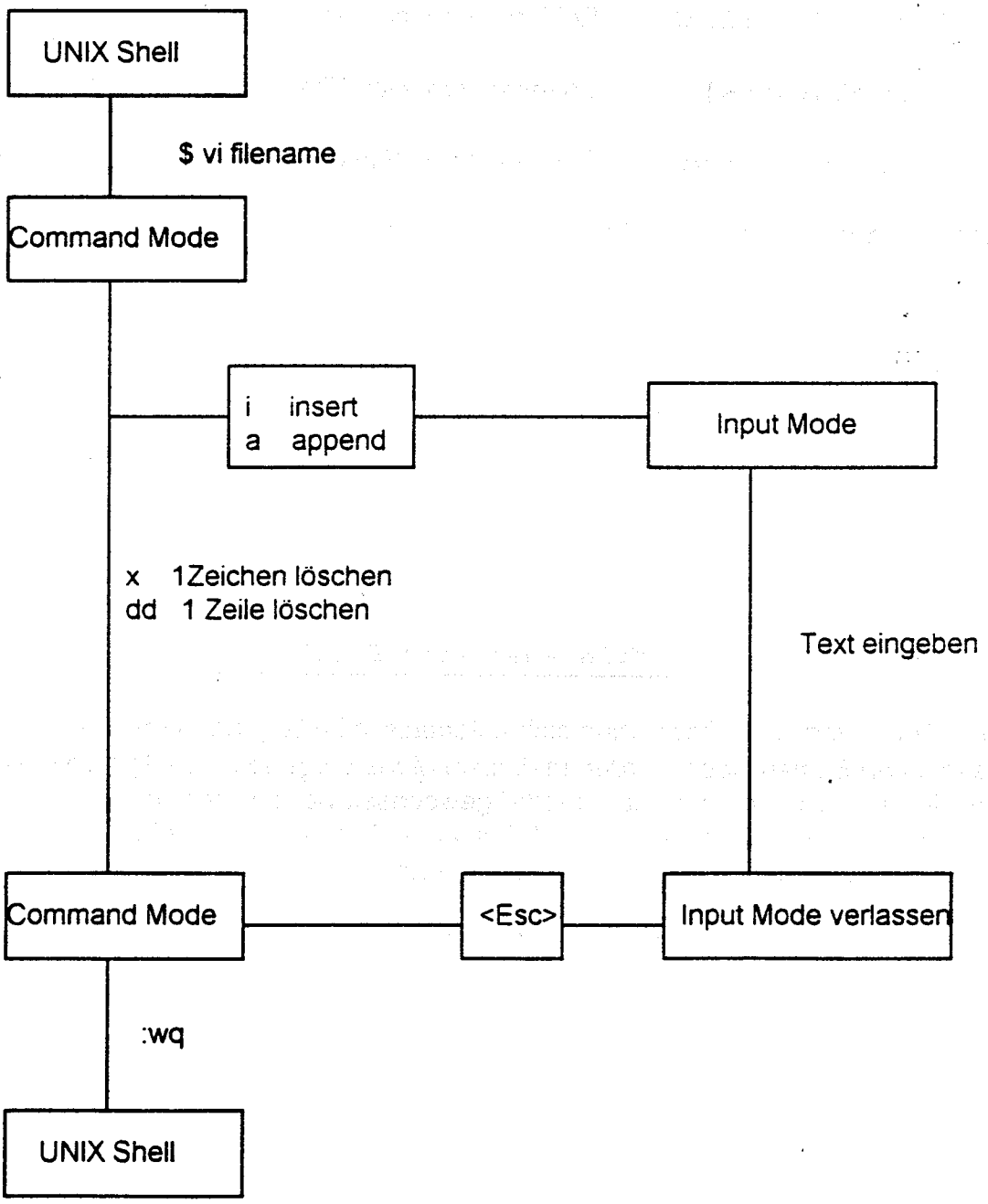
```
~
~
~
~
~
~
~
~
~
"filename" [new file]
```

Nach Aufruf von `vi` befindet man sich automatisch im sogenannten "command mode". Die meisten Editierungen werden in diesem Mode vorgenommen. Um Text einzufügen, muss in den sogenannten "input mode" gewechselt werden. Mit der Taste <ESC> springt man zurück in den "command mode". Folgende Darstellung soll die Aktionsmöglichkeiten in den zwei verschiedenen Modi von `vi` erläutern:

Die zwei verschiedenen Modi des vi Editors:

Der Command Mode: Hier kann gelöscht werden oder in den Input Modus gewechselt werden

Der Input Mode: Hier kann Text eingegeben werden.



$

Austritt aus `vi`

Verschiedene Möglichkeiten stehen im Command Modus zur Verfügung:

`:wq`	write / quit (Normalfall, um den `vi` Editor zu verlassen)
`ZZ`	gleich wie `:wq` (Hier ohne Doppelpunkt)
`:w`	write ohne quit (für Zwischenspeicherungen. Editor wird nicht verlassen)
`:w file`	File unter einem neuem Filenamen abspeichern, wobei auch ein ganzer Pfad angegeben werden kann. Achtung: Falls das File bereits besteht, wird es überschrieben!
`:wq!`	Abspeichern und verlassen unter Zwang (Wird bei verschiedenen Systemfiles gebraucht (`/etc/passwd`)
`:1,8 wq! file`	Nur die Zeilen 1-8 werden im File `filename` abgelegt.
`:q!`	Bei ungewollten Änderungen. Verlassen unter Zwang. ohne abspeichern.
`:e!`	(edit)Liest das File in Bearbeitung nochmals von der Disk in den Arbeitsspeicher, d.h. die letzten Änderungen, die noch nicht abgespeichert wurden, können so rückgängig gemacht werden ohne den Editor zu verlassen.
`:e! filename`	File `filename` editieren, ohne das aktuelle File zu speichern und ohne den Editor zu.

Cursorbewegungen

Der Cursor lässt sich auf verschiedene Arten bewegen. Oft wird zuerst <ESC> betätigt, um sicher im Command mode zu sein:

Pfeiltasten:	links	rechts	auf	ab
Buchstabe:	(h)	(l)	(k)	(j)

<NL>	1 Zeile abwärts
`1G`	File Anfang
`G`	File Ende
nG	Zeile *n*
`e`	(end) Wort Ende
`b`	(back) zurück an Wortanfang
`w`	(word) Anfang nächstes Wort
`H`	(home) Anfang des Bildschirms
`M`	(middle) Mitte des Bildschirmes
`L`	(last) Ende des Bildschirmes
^ oder 0	Zeilenanfang
$	Zeilenende

Achtung

Befindet sich `vi` im Inputmodus, so verursachen die 4 Pfeiltasten unsinnige, sonderbare Zeichen wie etwa `^A`,`^B`, etc..

Befehle zur Texteingabe

Wechseln vom Command mode in den Text Input mode.

`i`	Einschieben vor aktuellem Zeichen (insert)
`a`	Einschieben nach aktuellem Zeichen (append)
`A`	Anhängen an Zeilenende
`o`	Neue Zeile unterhalb Cursorposition (open new line)
`O`	Neue Zeile oberhalb Cursorposition
`r`	Ersetzen von 1 Zeichen (replace)
`R`	Überschreiben ab aktuellem Zeichen
`cw`	Ersetzen von 1 Wort, ab aktueller Position (change word)

Löschen

`x`	Löschen von 1 Zeichen auf aktueller Position
`dw`	Löschen von 1 Wort (ab aktueller Position)
`dd`	Löschen von 1 Zeile
`D`	Löschen ab aktueller Position bis Zeilenende
`dG`	Löschen ab aktueller Position bis Fileende

Rückgängig machen

`u`	Rückgängig machen der Änderungen bis zum letzten ESC-Tastendruck.
`:e!`	Alle Änderungen rückgängig machen (Datei neu einlesen)

Zeichen mit Sonderfunktionen

Gewisse Zeichen haben Sonderfunktionen. Ihre Verwendung ist mit grösster Vorsicht zu geniessen:

Punkt	.	1 beliebiges Zeichen
Stern	*	Repetition eines Zeichens
Eckige Klammer	[ ]	Ersatzzeichen
Backslash	\	Zeichen neutralisieren
Pfeil nach oben	^	Zeilenanfang
Dollar	$	Zeilenende

Suchen von Zeichenfolgen

Die Eingabe von / positioniert den Cursor auf die unterste Zeile am linken Bildschirmrand:

`/Zeichenfolge`	Suchen der `Zeichenfolge` Richtung Fileende von der aktuellen Position aus.
`/^Zeichenfolge`	Die gesuchte `Zeichenfolge` muss am Anfang der Zeile stehen.
`/Zeichenfolge$`	Die gesuchte `Zeichenfolge` muss am Ende der Zeile stehen.

Wird an Stelle von / ein ? eingegeben, so läuft die Suche der `Zeichenfolge` rückwärts.

Nach einer gefundenen Zeichenfolge kann mit dem Buchstaben `n` (next) die nächste Zeichenfolge gesucht werden. Anschliessende Substitution ist möglich (siehe Substitution).

Ersetzen von Zeichenfolgen

1. Erste gefundene Zeichenfolge auf aktueller Zeile ersetzen:

```
<Esc> :s/Zeichenfolge1/Zeichenfolge2
```

`s`	steht für substitute (ersetzen)
`Zeichenfolge1`	Suchbegriff
`Zeichenfolge2`	ersetzen durch `Zeichenfolge2`

2. Alle gefundenen Zeichenfolgen auf der aktuellen Zeile ersetzen:

```
<Esc> :s/Zeichenfolge1/Zeichenfolge2/gc
```

`g`	Steht für global; alle Vorkommen der
`Zeichenfolge`	auf der Zeile werden ersetzt.
`c`	(confirm)Bestätigung verlangen, ob ersetzt werden soll. nur bei Eingabe von `y` wird ersetzt.

3. Alle gefundenen Zeichenfolgen im ganzen File ersetzen:

```
<Esc> :1,$s/Zeichenfolge1/Zeichenfolge2/g
```

`1,$`	von der 1. bis zur letzten Zeile.

Zeilen zusammenfügen

Cursor auf Zeilenende positionieren:

`J`	(Join) aktuelle Zeile mit nachfolgender Zeile zusammenfügen
`nJ`	n Zeilen ab aktueller Zeile zusammenfügen

Zeilen trennen

Eine Zeile kann getrennt werden. Dazu muss an der zu trennenden Stelle ein "a" (append) gefolgt von <NL> eingegeben werden. Somit wird die Zeile getrennt!

vi Parameter

Der vi kennt 38 Parameter. Diese können die Arbeitsweise und den Ablauf des Editors sinnvoll ergänzen. Die Benützung dieser Parameter kann temporär oder permanent erfolgen.

Gesetzte Parameter anzeigen

```
$ vi
: set all
```

vi Parameter setzen

1. `/etc/profile`
   Dieses File wird von allen Benutzern beim Einloggen durchlaufen. Hier wird eine Grundumgebung für alle Benutzer gesetzt. Änderungen an dieser Datei kann nur der Superuser vornehmen.

   ```
   EXINIT="set option option . . . " ; export EXINIT
   ```

2. `$HOME/.profile`
   Jeder Benutzer am System hat ein eigenes `.profile`. Hier kann die persönliche Umgebung gesetzt werden.

   ```
   EXINIT="set option option . . . " ; export EXINIT
   ```

3. `./.exrc`
   Dieses File wird erst beim starten des vi gelesen. Jedes Directory kann ein eigenes `.exrc` beinhalten.

   ```
   set option option...
   ```

4. **im vi selber**
   Die gesetzten Parameter sind dann nur für den aktuellen vi Aufruf gültig.

   ```
   :set option option...
   ```

Beispiele von Parametern:

Name	Abkürzung	Parameter ausschalten
`exrc`		`(noexrc)`

Die `vi`-Optionen, die zusätzlich im File `.exrc` im Home Directory oder in einem beliebigen Subdirectory gesetzt werden können, werden beim Aufstarten von `vi` miteinbezogen. Ist `noexrc` gesetzt, so werden die im `.exrc` enthaltenen Optionen nicht berücksichtigt.

`flash, flash` `(noflash)`

Bei jeder falschen Eingabe ertönt ein nervender Piepston oder der Bildschirm blinkt.

`ignorecase, ic` `(noic)`

Gross-/Kleinbuchstaben werden bei Suchbefehlen identisch behandelt.

`number, nu` `(nonu)`

Numerieren der Zeilen. Die Nummern werden nur im vi angezeigt und sind nicht in der Datei abgespeichert.

`showmode, smd` `(nosmd)`

Angabe am Bildschirm (unterste Zeile) in welchem Modus man sich befindet.

`term`

Diese Option zeigt den Bildschirmtyp an.

Korn Shell

Der wichtigste Befehlsinterpreter unter UNIX. Die Korn Shell ist eine Erweiterung der Bourne - Shell. Zusätzliche Fähigkeiten der Korn Shell gegenüber der Bourne Shell sind z.B.:

- Befehlsgedächtnis (History - Mechanismus)
- Eingebauter Befehlszeilen Editor
- Alias - Mechanismus
- Ca. 10 Mal schneller als die Bourne Shell.

Befehlswiederholung

Jeder ausgeführte Befehl wird im File `.sh_history` im Home - Directory abgelegt. Dieses File wird auch bei wiederholten Logins weitergeführt.

Befehl anzeigen

`$ history`	Zeigt die 16 letzten Befehle
`$ history -n`	zeigt die n letzten Befehle an
`$ history n`	zeigt Befehle, startend mit dem Befehl n

Die Variable `HISTSIZE` definiert, wieviele Befehle zurückverfolgt werden können. (Default `HISTSIZE=127`). Überprüfen mit:

```
$ echo $HISTSIZE
127
```

Beispiel:

```
$ history          Zeigt die letzten 16 Befehle an
1   date
2   ls
3   chmod 777 a.fil
4   history
```

Befehl zurückholen und editieren

Folgende Variable muss definiert sein (File `.profile` oder `/etc/profile`):

```
EDITOR=vi; export EDITOR
```

Der Befehls Editor kennt *Input Modus* und *Befehls Modus* wie `vi`:

Befehl zurückholen

⇨ Taste **<ESC>** und:

`-`	holt den letzte Befehl zurück
`+`	ein Befehl vorwärts (falls zuweit zurück)
`/string`	sucht nach dem Befehl, der `string` enthält (beginnend beim aktuellen Befehl)
`n`	(next) sucht nächste Zeile, die `string` enthält

Befehle zurückholen und modifizieren

Letzten Befehl zurückholen	<Esc> -
Befehl suchen	<Esc> /Befehl
Nächsten Befehl suchen	n

Befehl editieren

Ein Zeichen nach rechts	l oder Leertaste
Ein Zeichen nach links	h
Anfang des Wortes	b
Ende des Wortes	w
Ende Zeile	$
Anfang Zeile	0
Springen zum Zeichen *x*	f*x*

Einfügen und ersetzen

Nach dem Cursor einfügen	a
Vor dem Cursor einfügen	i
Am Ende der Zeile einfügen	A
Beliebig viele Zeichen ersetzen	R
Einfügemodus verlassen	<Esc>

Löschen

Ein Zeichen löschen	x
Aktuelles Wort löschen	dw
Befehlszeile löschen	dd oder Ctrl C
Änderung rückgängig machen	u

Alias

alias name=wert

Die Aliasfunktion erlaubt eine Namensersetzung. Häufigster Gebrauch bei längeren Befehlen.

Beispiele:

```
$ alias dir="ls -lia"
```
Falls sie sich von DOS nicht trennen können

```
$ dir
Total 4
drwxrwxrwx   2   pat   actg  128   Aug  28   12:32    source
```
Mal sehen, ob es funktioniert

Aliasliste anschauen

```
$ alias
dir=ls -lia
```
Welche Alias sind gesetzt ?

Alias auflösen

```
$ unalias dir
```
UNIX Befehle sind doch besser als DOS.

Systemprompt anpassen

```
$ PS1="<\$PWD > $ "; export PS1
</home/user01 > $
```
PS1=Primary System Prompt

Files drucken

`lp [-option...] file...`

Funktion: Files auf Drucker ausgeben.

Optionen:

`-nx`	Druckt *x* Kopien (Default = 1)
`-d dest`	Drucker ist `dest`(ination)
`-t title`	Druckt einen Titel `title` auf die Banner Seite. Falls mehrere Worte im Titel vorkommen, so müssen sie in " " eingeschlossen werden.
`-m`	Schickt eine email, wenn Druck beendet ist.

Beispiel:

```
$ lp -n2 d.adr
Request-id is wenger-139
```

File `d.adr` mit 2 Kopien ausdrucken

Drucker-Status abfragen

`lpstat [-option...]`

Funktion: Gibt den Status von Benutzeraufträgen, von verfügbaren Druckern. von Druckerklassen und von Geräten aus.

Optionen:

`-t`	Zeigt die gesamte Statusinformation an.

Druckaufträge annullieren

`cancel request-id...`

Funktion: Jeder Benutzer kann einen oder mehrere seiner Druckeraufträge, die mit `lp` aufgegeben wurden, annullieren..

Beispiel:

`$ cancel wenger-139` `wenger-139 cancelled`	Druckauftrag löschen
`$ cancel -u rmuggli`	löscht alle Druckaufträge von Benutzer `rmuggli`

Drucker aktivieren und inaktivieren

`enable printer...`

Funktion: Aktiviert den oder die angegebenen Drucker, so dass Druckaufträge angenommen werden. Die Umkehrung hiervon ist `disable`.

Beispiel:

`$ enable hplaser` `hplaser now enabled`	Drucker aktivieren, so dass Druckaufträge entgegengenommen werden können

disable [-option...] printer...

Funktion: Deaktiviert (legt still) den oder die angegebenen Drucker. Begonnene Ausdrucke werden bei einer erneuten Aktivierung mit `enable` von vorne ausgedruckt. Druckaufträge werden trotzdem weiter angenommen, aber nicht mehr ausgedruckt.

Optionen:

`-r "reason"` Es kann ein Grund für das Anhalten des Druckers mitangegeben werden. Dieser wird einem Benutzer bei Verwendung von `lpstat` mitgeteilt.

`-w` Der Drucker wird nicht inaktiviert, bevor der aktuell druckende Auftrag fertig gedruckt ist.

Beispiel:

```
$ disable -r "Papierwechsel" hplaser
hplaser now disabled
```

Drucker `hplaser` inaktivieren, mit der Meldung Papierwechsel

Files suchen

`find pathname.. expression.. action`

Funktion: Durchsucht ganze Directory-Strukturen inkl. Subdirectories nach Files, welche bestimmte Kriterien erfüllen müssen.

`pathname:`

Es können ein oder mehrere Pfadnamen angegeben werden.

`expression:`

`-name` *`file`*	Nach File mit `file` suchen. Achtung auf Sonderzeichen: `?` und `*`; Falls solche Wildcards verwendet werden, muss der ganze Ausdruck in Anführungszeichen stehen.
`-perm [-]`*`onum`*	Exakte Zugriffserlaubnis in oktaler Form wie z.B. 766. Falls `onum` ein Minuszeichen vorangestellt wird, so müssen die gesuchten Files als Minimum die in `onum` enthaltenen Berechtigungen besitzen.
`-type` *`x`*	`x = f` Normale Datei `x = d` Directory
`-user` *`name`*	Eigentümer (Owner) der Datei ist `name`
`-size [±]`*`n`*	File hat weniger als (-)oder mehr als (+) `n` Blöcke zu 512 Bytes. Wird bloss `size` *`n`* angegeben, so muss das File genau *n* Blocks gross sein. Falls nach der Grösse *`n`* der Buchstabe `c` folgt, wird nach Anzahl Zeichen gesucht.
`-inum` *`n`*	File mit der i-node Nummer `n`

action:

`-print`	Ausgabe des Pfadnamens der gefundenen Files
`-exec` *`Befehl`* `{} \;`	Ausführen des Befehls falls File gefunden wird
`-ok` *`Befehl`* `{} \;`	Gleich wie `-exec` aber mit Bestätigung

Verneinung:

```
$ find . ! -user rmuggli -print
```

Alle Files im aktuellen Directory, die **nicht** dem Benützer `rmuggli` gehören, werden ausgegeben.

Bemerkungen:

`\(` *`Bedingung1 -o Bedingung2`* `\)`

Es kann auch nach mehreren Bedingungen gleichzeitig gesucht werden. Falls ich zwei verschiedene Files gleichzeitig suchen möchte, muss ich die oder-Verknüpfung gebrauchen. Ohne Angaben werden die Bedingungen und verknüpft, das heisst, die Bedingungen treffen für ein einzelnes File zu.

`-o`	ODER-Verknüpfung
`-a`	UND-Verknüpfung (default)

Anstelle eines ganzen Pfadnamens kann auch ein Punkt (.) angegeben werden, falls vom aktuellen Directory gesucht werden soll.

Beispiele:

```
$ find . -name d.adr -print 2>/dev/null
```

Die Pfadnamen von allen Files d.adr im aktuellen Directory werden auf den Bildschirm ausgegeben. Die Fehlermeldungen werden in den Bit-Bucket geworfen.

```
$ find . -size +10 -print
```

Die Pfadnamen aller Files im aktuellen Directory die grösser als 5 Kilobytes gross sind, werden auf den Bildschirm ausgegeben.

```
$ find . ! -user rmuggli -print
```

Gibt alle Pfadnamen im aktuellen Directory aus, die nicht rmuggli gehören.

```
$ find / -size 0 -ok rm {} \;
```

Sucht alle leeren Files auf dem ganzen System und fragt bei jedem File das gefunden wird, ob es gelöscht werden soll.

```
$ find /usr/bin /usr/sbin -type d -exec ls -lia {} \;
```

In den Directories `/usr/bin` und `/usr/sbin` werden alle Directories mit dem Befehl `ls -lia` ausgegeben.

```
$ find . \( -name "*[0-9]*" -o -name "*dd*" \) -mtime -7 -ok \ rm
{} \;
```

Alle Files im aktuellen Directory, die irgendwo im Namen eine Zahl haben und innerhalb sieben Tage modifiziert wurden, sowie alle Files die irgendwo im Namen die Zeichenfolge dd enthalten und innerhalb sieben Tage modifiziert wurden, werden mit der Frage, ob sie gelöscht werden sollen, angezeigt. Nur mit einer Antwort, die mit y beginnt, werden die einzelnen Files gelöscht.

Ein- und Ausgabe Umlenkung

Normalerweise ist der Bildschirm Ausgabeort der Daten, bzw. die Tastatur Eingabeort der Daten, die auf Veranlassung eines Shell Befehles bearbeitet werden. Diese Art von Ein-/ Ausgabe wird als Standard-Input und Standard-Output benannt, und können vom Benutzer umdefiniert oder umgelenkt werden. Die Fehlermeldungen oder der Standard-Error wird ebenfalls standardmässig auf den Bildschirm ausgegeben.

Die Fähigkeit, Input und Output umzulenken stellt einer der grössten Vorteile der **Shell** dar.
Alles was auf den Bildschirm ausgegeben werden kann, ist auch in ein File umlenkbar. Dabei kann das File auch ein Special-File sein.

- Die Umlenkung des **Standard-Inputs** erfolgt mit dem Zeichen **<**.
- Die Umlenkung des **Standard-Outputs** mit dem Zeichen **>**.
- Die Umlenkung des **Standard-Errors** mit den Zeichen **2>**.

Standardoutput an einem bestehenden File **anhängen** mit den Zeichen **>>**.

Standard-Fehlermeldung an Standard-Output anhängen **2>&1**

Wird der Standardoutput umgelenkt, so wird der Standard-Error nicht automatisch umgelenkt. Damit der Standard-Error auch umgelenkt wird, muss der File-Descriptor für Standard-Error, nämlich 2 ebenfalls angegeben werden.

Will man die Fehlermeldungen weder auf dem Bildschirm noch in einem File ablegen, so lässt man sie einfach in den **Bit Bucket** nach `/dev/null` verschwinden. Dieses File ist immer leer.

Beispiele:

```
$ ls -lia >dir.liste
```
Die Ausgabe von `ls -lia` wird ins File `dir.liste` geschrieben. Falls `dir.liste` bereits existiert wird es überschrieben. Falls es noch nicht existiert, wird es neu erstellt.

```
$ mail user00 < d.brief
```
Den Inhalt des Files d.brief dem Benutzer `user00` per e-mail schicken.

```
$ cat d.adr >>d.allekunden
```
Die Ausgabe von d.adr wird an das File `d.allekunden` angehängt. Falls `d.allekunden` noch nicht existiert wird es neu erstellt.

```
$ find /home/dir00 -name "d.*" -print >d.files 2>fehler
```
Es werden alle Pfadnamen im /home/dir00, die mit d. beginnen, ins File d.files geschrieben. Die Fehlermeldungen (z.B. keine Zutrittserlaubnis ins Directory) werden ins File fehler geschrieben.

```
$ find /home/rene -name "d.*" -print >d.files 2>/dev/null
```
Wie Beispiel oben. Fehlermeldungen werden aber ins File `/dev/null` umgelenkt. (Bit-Bucket)

```
$ cat a.file b.file >c.file 2>&1
```
`a.file` und `b.file` werden zusammengehängt (cat = concatenate) und in das File `c.file` umgelenkt. Die 1 vor dem > Zeichen ist fakultativ. Allfällige Fehlermeldungen (z.B. ein nicht existierendes File) werden dem File, in welches der Standard-Output umgelenkt wird, angehängt.

Pipelines und Filters

Die Standardausgabe eines Shell-Befehls kann mittels eines Pipe-Zeichens (|) zur Standardeingabe des nächsten Befehls gemacht werden, d.h. es wird eine Pipeline gebildet.

Eine Pipeline macht das Erstellen temporärer Files unnötig.

Beispiele:

`$ ls -l | lp`	Druckt eine Liste aller Files im aktuellen Directory im langen Format auf dem Drucker aus.
`$ ls -lR / | pg`	Listet alle Files im langen Format und zwar von der `root` aus, inkl. aller Files in allen Subdirectories und zeigt die Ausgabe seitenweise auf dem Bildschirm an.
`$ file * | pg`	Zeigt den Filetyp jedes Files im aktuellen Directory, und zeigt die Ausgabe seitenweise am Bildschirm an.
`$ pg a.fil b.fil c.fil | lp`	Der Standard-Output des `pg`-Befehls wird als Standard-Input für den Druckbefehl verwendet.

Pipelines können nur dann verwendet werden, wenn der Befehl links von der Pipeline eine Ausgabe ergibt, die normalerweise auf dem Standard-Output (Bildschirm) erscheint, und der Befehl rechts von der Pipeline den Standardinput entgegennehmen kann.

Files vergleichen

Es existieren zwei Tools, die Files vergleichen:

a) `cmp` Meldet den 1. Unterschied

b) `diff` Meldet alle Unterschiede

a) `cmp [-option] filea fileb`

Funktion: Vergleicht den Inhalt zweier Files, und meldet die erste Zeilennummer und Zeichenposition, in der Differenzen enthalten sind.

Option:

`-s` Unterdrückt Angabe der unterschiedlichen Zeilen/Zeichen.
Eine Möglichkeit in der Shellprogrammierung. wenn man nur am Return Code interessiert ist, den man weiter abfragen kann.

Return Code (Exit Status)

Jedes Kommandos liefert einen Return Code, der mit dem Kommando `echo $?` ausgegeben werden kann. Näheres siehe Shellprogrammierung.

0	Files sind identisch, o.k.
1	Files unterscheiden sich, nicht o.k.
2	Vergleich unmöglich

Beispiel:

```
$ cmp d.adr d.adrcopy
d.adr d.adrcopy differ: char 873, line13
$ echo $?          Return Code anzeigen lassen
1                  Files unterscheiden sich.
```

b) diff [-option ...] file1 file2

Funktion: Vergleicht die beiden angegebenen Files und gibt auf die Standardausgabe aus, welche Zeilen wie geändert werden müssen, um `file2` aus `file1` wieder herzustellen. Die Ausgabe hat etwa folgendes Format:

`n1 a n3,n4`	für einzufügende Zeilen
`n1,n2 d n3`	für zu löschende Zeilen
`n1,n2 c n3, n4`	für auszutauschende Zeilen

`n1, n2, n3` sind dabei Zeilenangaben

Optionen für Files:

`-b`	Hierbei werden Tabulator - und Leerzeichen am Ende der Zeile beim Vergleich ignoriert.
`-i`	Klein- und Grossbuchstaben werden für den Vergleich nicht unterschieden.

Optionen für Directories:

`-s`	Listet zusätzlich alle Files, die in beiden Directories identisch sind.
`-r`	Die Suche nach identischen Subdirectories wird rekursiv, d.h. von der untersten Directory Ebene her gestartet.

Beispiel:

```
$ diff  file1 file2
1,2c1
<yyyyyyyyyyyyyyyyyy                    < bedeutet: Das steht im File1
<yyyyyyyyyyyyyyyyyy
---------------
>01234567890012345678                  > bedeutet: Das steht im File2
4d2
<Diese Zeile gehört aber nicht hierher!
7a6
>111111111111111
$
```

Erklärung:

Die Änderungsanweisungen geben an, wie File `file1` editiert werden muss, damit es File `file2` angepasst werden kann.
Zeile 1 und 2 des ersten Files mit Zeile 1 des zweiten Files ersetzen.
Zeile 4 (die im zweiten File nach der Zeile 2 käme) des ersten Files löschen.
Nach Zeile 7 des ersten Files Zeile 6 des zweiten Files anfügen.

Konvertieren und kopieren eines Files

`dd [if=inputfile] [of=outputfile] [option=value ...]`

Funktion: Kopiert ein oder mehrere Files (oder ganze Filesysteme), wobei gleichzeitig gewisse Konvertierungen möglich sind. `dd` ist sehr schnell, wenn durch die Option `bs=n` eine hohe Blockgrösse benutzt wird.
Ist `if=inputfile` oder `of=outputfile` nicht angegeben, so wird die Standardeingabe bzw. Standardausgabe ersetzt.
Input- und Outputfile müssen verschieden sein, sonst wird ohne Warnung die Ausgangsdatei gelöscht!!!!

Optionen:

`if=`*`file`*	Input Filename
`of=`*`file`*	Output Filename
`ibs=`*`n`*	Input Block Grösse (Default = 512 Byte)
`obs=`*`n`*	Output Block Grösse (Default = 512 Byte)
`bs=`*`n`*	Buffer Grösse für Konversion
`skip=`*`n`*	Überspringen von *n* Blocks zu 512 Bytes vor dem Start der Konversion
`count=`*`n`*	n Input Blocks zu 512 Bytes kopieren
`conv=ascii`	Konvertiert EBCDIC zu ASCII
`=ebcdic`	Konvertiert ASCII zu EBCDIC
`=lcase`	Grossbuchstaben zu Kleinbuchstaben
`=ucase`	Kleinbuchstaben zu Grossbuchstaben
`=swab`	Tauscht jedes Byte Paar

Beispiele:

`$ dd if=d.adr of=d.adrdd conv=ucase`
Konvertieren von Kleinbuchstaben in Grossbuchstaben:

`$ dd if=d.adr skip=5 count=10 of=d.adrtxt`
Inputfile 'd.adr', 5 Blocks überspringen, dann 10 Blocks kopieren, Outputfile ist `d.adrtxt`.

Files sortieren

`sort [-option .] [+pos1 -pos2] [inputfile .] [-o outputfile]`

Funktion: Sortiert die Zeilen aller angegebenen Files und schreibt das Ergebnis auf die Standardausgabe.
Wird kein Sortierschlüssel angegeben, wird die ganze Zeile sortiert.
Es können ein oder mehrere Schlüssel angegeben werden.

Optionen:

`-f`	Klein- und Grossbuchstaben werden für den Vergleich nicht unterschieden.
`-n`	Numerische Werte werden gemäss ihren arithmetischen Wertigkeiten sortiert.
`-tx`	Feldseparator ist in diesem Fall das Zeichen 'x' (Standard: <tab> oder Leerschlag)
`-b`	Führende Leerschläge werden für den Vergleich ignoriert.
`-o file`	Die Ausgabe soll anstatt auf die Standardausgabe auf `file` gehen: Dieses Outputfile darf den gleichen Namen wie das Inputfile haben.

Bemerkungen:

Ohne Optionen wird nach dem **ASCII-Zeichensatz** sortiert.
Es gilt folgende Reihenfolge:

- nicht druckbare Zeichen, (oktaler Code kleiner als 040)
- Sonderzeichen: <Leerzeichen> ! " # $ % ' ( ) * + - . / , 0-9, : ; < = >
- ? @
- alle Grossbuchstaben
- [\ ] ^ _ `
- alle Kleinbuchstaben
- { | } ~
- Zeichen mit einem Code grösser als oktal 0176

Sortierschlüssel werden mit der Notation `+pos1 -pos2` definiert, wobei `pos1` und `pos2` als Felder, nicht als absolute Zeichenpositionen zu verstehen sind.

Bei mehreren Sortierschlüsseln wird das Kommando von links nach rechts aufgelöst. Das erste Feld ist immer `+0`. Ein Feld ist eine Zeichenkette, die keine Leerschläge enthält.

Ein Feldseparator kann ein Leerschlag, Tabulator-Zeichen und ein anderes explizit angegebenes Zeichen sein.

`+pos1` und `-pos2` kann auch die Form `m.n` haben. In diesem Fall wird der Sortierschlüssel limitiert: Er beginnt mit dem Zeichen `n` im Feld Nummer `m` von `pos1` und endet vor dem Zeichen `n` des Feldes `m` in `pos2`. Fehlt ein definierter Feldseparator, wird `m` immer als *0* angegeben.

Beispiele:

```
$ sort +0.4 -0.14  +0.24 -0.39 d.adr -o d.adrsort
```

Es wird das Feld 0 (ganze Zeile) des Files `d.adr` sortiert, beginnend beim fünften bis und ohne 14. Zeichen. Als zweites wird ab 25. bis und ohne 39. Zeichen sortiert. Die Ausgabe wird ins File `d.adrsort` geschrieben.

```
$ sort +3n -4  +2n -3 -t: /etc/passwd
```

Das 4. Feld (Gruppennummer) wird zuerst sortiert. Falls gleiche Gruppennummern vorhanden sind, wird noch weiter das 3. Feld (Benutzernummer) des Files `/etc/passwd` sortiert.

Zeichenfolgen in einem File suchen

`grep [-option ...] expression [file]`

Funktion: Sucht Zeichenfolgen in einem oder mehreren Files und gibt diese auf die Standardausgabe aus.

Optionen:

`-i` (ignorecase) Klein- und Grossbuchstaben werden für die Übereinstimmung nicht unterschieden.

`-v` (verbose) Es werden alle Zeilen ausgegeben, auf die das Muster nicht zutrifft.

`-c` (count) Es wird nur die Anzahl der Zeilen gezählt, die dem Muster genügen.

`-l` (list) Die Filenamen aller Files, in denen die Bedingungen erfüllt sind, werden ausgegeben.

`-n` (number) Vor jeder zutreffenden Zeile wird der Filename und die Zeilennummer ausgegeben.

`-s` (status) Es wird keine Ausgabe produziert, sondern nur der entsprechende Return Code (Exit Status) zurückgeliefert:

0	gefunden
1	nicht gefunden
2	Vergleich unmöglich

Beispiele:

```
$ grep Gaston a.fil
```

Alle Zeilen im File `a.fil`, die `Gaston` enthalten werden ausgegeben.

```
$ grep -v Marsupilami a.fil
```

Alle Zeilen von `a.fil` die `Marsupilami` nicht enthalten

```
$ grep -i Superman *
```

Alle Zeilen von allen Files im aktuellen Directory, die `Superman` enthalten, werden mit Filenamen ausgegeben.

```
$ grep -ni 'Daily Planet' *
```

Alle Zeilen von allen Files im aktuellen Directory, die `Daily Planet` oder `daily planet` enthalten, werden mit der Zeilennummer ausgegeben.

```
$ find /home -name games -exec ls -l {} \;|grep July| lp
```

Es werden alle `games` im Format `ls -l` an `grep` als Eingabe übergeben. `grep` gibt alle Zeilen die `July` enthalten an `lp` (Drucker) zur Ausgabe weiter.

Suchmuster Definitionen:

grep enthält eine Fülle von Möglichkeiten, Zeichen(-folgen) in einem oder gleich mehreren Files zu finden. Dazu verwendet man folgende Sonderzeichen:

`^`	am Anfang einer Zeile
`$`	am Ende einer Zeile
`....`	ein beliebiges einzelnes Zeichen pro Punkt
`\.`	das Zeichen ist ein normaler Punkt
`[a-d]`	Eines der Zeichen im Bereich `a-d`
`[^a-d]`	Alle Zeichen ausser den Zeichen `a-d`
`ausdruck``*`	Der Ausdruck darf 0 bin n mal vorkommen
`ausdruck``+`	Der Ausdruck darf ein- oder mehrmals vorkommen
`ausdruck``?`	Der Ausdruck darf kein - oder einmal vorkommen

Beispiele:

`[Dd]uck`	Entweder `Duck` oder `duck`
`^Duck`	Zeilen, die mit `Duck` beginnen
`Zürich$`	Zeilen, die mit `Zürich` enden
`Tr.ck`	Zeilen, die z.B. `Track`, `Trick`, oder `Truck`, etc.
`^[abx]`	Entweder `a`, `b` oder `x`, jedoch am Zeilenanfang
`[ac-z]`	Entweder `a` oder `c`, `d`, ..., `x`
`[^a-z]`	Nicht ein Kleinbuchstabe an dieser Zeichenposition
`[-a-z]`	Entweder - oder alle Kleinbuchstaben

Doppelte Zeilen eliminieren

uniq [option...[+n][-n][inputfile][outputfile]]

Funktion: Eliminiert alle identischen Folgezeilen. d.h. File muss sortiert sein.

Optionen:

-u	Auf das Outputfile werden nur die nicht wiederholten Zeilen geschrieben.
-d	Auf das Outputfile wird nur je eine der repetierten Zeilen geschrieben.
-c	Übersteuert die Option -u und -d. Output ist wie `uniq` ohne Optionen, ausser dass das Outputfile zusätzlich mit Angabe der Anzahl wiederholten Zeilen versehen wird.
-n	Anzahl der zu überspringenden Felder pro Zeile (ein Feld ist eine Zeichenkette ohne Leerschläge, Tabulatoren oder Newline Zeichen).
+n	Anzahl der zu überspringenden Zeichen.

Beispiel:

```
$ cat textfile
Ein Gespenst geht um in Europa
Ein Gespenst geht um in Europa
Sein oder Nichtsein, das ist hier die Frage
Das Restaurant am Ende des Universums
Das Restaurant am Ende des Universums
Das Restaurant am Ende des Universums
$ uniq -c textfile
2  Ein Gespenst geht um in Europa
1  Sein oder Nichtsein das ist hier die Frage
3  Das Restaurant am Ende des Universums
```

File Dump

hd [filename]

Funktion: Erstellt einen Abzug (Hexadezimaldump) eines Files. Wird kein File angegeben, wird *Stdin* erwartet.

Beispiel:

```
$ hd d.adr | more
```

od [-option ...] [file]

Funktion: Erstellt einen oktalen Abzug eines Files. Wird kein File angegeben, wird der Standardinput von der Tastatur her erwartet.

Optionen:

`-c`	ASCII-Zeichen,
`-d`	16-Bit-Worte dezimal
`-b`	Octale Ausgabe

Beispiele:

`$ od -cx sc.exap40 >work`	Dump wird in File `work` geschrieben.
`$ echo '$IFS' | od -cx`	Da `od` auf *Stdin / Stdout* beruht, ist das Kommando voll Pipe-fähig.

Files kopieren mit Intervention

`tr [-option]string1 [string2][<inputfile] [>outputfile]`

Funktion: Ersetzen oder löschen von angegebenen Zeichenfolgen auf Basis ein Zeichen. Die Zeichen können auch in der Oktalform '\0xxx' angegeben werden.

Optionen:

`-d`	(**d**elete) Löscht die im `string1` angegebenen Zeichen
`-s`	(**s**queeze) Alle Wiederholungen von Zeichen erscheinen als ein einziges Zeichen.

Beispiele:

```
$ cat file1
TBZ
Tiger
$ tr 'T' 't' <d.adr
tBZ
tiger
```

Wandelt ein grosses `T` in ein kleines `t` um.

```
$ tr -d 't' < file1
BZ
iger
```

`t` wird gelöscht. Ursprüngliches File bleibt unverändert!.

Zeilen, Worte, Zeichen zählen

`wc [-option ...] [file ...]`

Funktion: Zählt die Anzahl Zeilen, Worte und Zeichen in einem oder mehreren Files und gibt das Resultat auf die Standardausgabe aus.

Eine Zeile ist durch ein Newline-Zeichen (hex 0a) abgegrenzt, ein Wort durch einen Leerschlag (hex 02), Tabulator - (hex 09) oder ein Newline-Zeichen.

Optionen:

`-l`	Zählt nur die Zeilen
`-w`	Zählt nur die Wörter
`-c`	Zählt nur die Zeichen

Default Ausgabe ist `wc -lwc`

Beispiele:

```
$ wc d.adr
32     125    960      d.adr
```
File hat 32 Zeilen, 125 Wörter und 960 Zeichen.

```
$ wc *
32     125    960      xyfile
32     112    715      oldfile
24     124    773      newfile
88     361    2448     total
```
Alle Files im Directory berücksichtigen

Das Gesamttotal im Directory

```
$ ls | wc -w
14
```
Files im Directory zählen

Systemstatus

Unter Status versteht man eine bestimmte Situation eines Systems zu einem bestimmten Zeitpunkt.

- Welches Datum ist im System definiert?
- Welche Prozesse laufen aktiv für einen bestimmten Benutzer?
- Ist ein Prozess aktiv?
- Wieviel Platz ist auf der Disk noch frei ?
- etc.

Datum, Zeit und Zeitzone abfragen

`date [mmddHHMM[cc]yy]`

Funktion: Gibt Datum und Uhrzeit des Systems aus. Das Systemdatum und die Systemzeit neu setzen kann nur der Superuser.

Die Ausgabe des `date` Befehls kann zusätzlich formatiert werden. Dazu wird das Pluszeichen (+) und eine Reihe von Feld-Deskriptoren verwendet, welchen immer ein %- Zeichen vorangestellt wird. Die Format Anweisungen müssen in Hochkommas eingeschlossen sein. Einige dieser Feld-Deskriptoren sind:

`%B`	voller Monatsname
`%d`	Tag des Monats, immer zweistellig (01-31)
`%e`	Tag des Monats (1-31)
`%m`	Monat des Jahres (1-12)
`%Y`	für die vierstellige Jahresangabe

Beispiele:

```
$ date
Fri  Nov 12 08:16:47 MES 1996
$ date '+%e. %B %Y'
12. November 1996
```

Name / Version des UNIX Systems

```
uname [-option ...]
```

Optionen:

`-s`	Systemname
`-n`	Zeigt den Nodenamen an (für Netzwerk-Kommunikation)
`-r`	Zeigt die Release Nummer des Betriebssystems an
`-v`	Zeigt die Version des Betriebssystemes an
`-m`	Zeigt den Hardwaretyp (Maschine) an
`-p`	Zeigt den Prozessor-Typ an
`-a`	Alle Angaben anzeigen lassen

Beispiel:

```
$ uname -a                          Zeigt, an welchem System gearbeitet wird
gurul gurul 4.0 2.0 i386/i486 386/486/MC
```

Befehlsdurchführungszeit feststellen

`time befehl`

Funktion: Der angegebene `befehl` wird ausgeführt und anschliessend 3 verschiedene Zeiten ausgegeben (Sekunden):

`real`	Zeit für Kommando vom Startpunkt bis zum Ende, d.h. bis zur Rückgabe des System Prompts
`user`	Zeit im Benutzer Modus, also die Zeit, welche der Prozess für memoryinterne Programm Befehle (Addition, Bereichsübertragungen, etc) aufwendet
`sys`	Zeit im Systemmodus, also für Prozess Aufbau und Aufrufe vom System (System Calls)

Beispiel:

```
$ time find /home -name core -exec rm {} \;
real        0m12.20s
user        0m0.25s
sys         0m2.87s
```

Anzahl belegter Blocks pro File oder Directory

du [-option] [name ...]

Funktion: Gibt die Grösse aller Subdirectories in Anzahl Blocks zu 512 Bytes aus.

Optionen:

-s	Nur Gesamttotal der Blocks aller Subdirectories wird ausgegeben.
-a	Ausgabe der Blockzahl für jedes einzelne File
-r	Falls **du** ein Directory nicht durchsuchen kann (wegen fehlender Entrittsberechtigung), wird eine entsprechende Fehlermeldung ausgegeben.

Beispiele:

```
$ du -a
6   ./prog.c
5   ./book/chap1
3   ./book/chap27
29  .
```

Alle Files im aktuellen Directory werden mit Anzahl Blocks ausgegeben.

Total des aktuellen Directories

```
$ du -s /etc
7105
```

Total Blocks im Directory **/etc**

```
$ du /home/user00
355         ./progs.dir
46          ./unix.dir
78          ./work
270         ./save.dir
```

Blocks pro File im Directory **/home/user00**

Anzahl freier Blocks pro Filesystem

`df [-option] [filesystem ...]`

Funktion: Ausgabe der freien Blöcke zu 512 Bytes auf jedem Filesystem

Optionen:

`-b`	Gibt nur die Anzahl freier Blocks zu 1024 Byte aus
`-e`	Gibt nur die Anzahl freier Files aus
`-g`	Gibt Informationen über das Filesystem selbst, z.B. Typ
`-t`	Gibt zusätzlich zur Anzahl freier Files und Anzahl freier Blocks auch die Anzahl total Blocks und Files an.
`-v`	Gibt die gebrauchten, sowie die freien Blocks in Prozenten an
`-n`	Filesystemtypname

`$ df` Gibt die benützten und freien Blocks in Prozenten an.

```
/          (/dev/root):          255712 blocks     24472 files
/proc      (/proc):                   0 blocks       129 files
/dev/fd    (/dev/fd):                 0 blocks         0 files
/stand     (/dev/dsk/c0t6d0sa): 15996 blocks       398 files
/nt        (/dev/dsk/c0t4d0s1): 668148 blocks    55506 files
```

`$ dfspace` Pendant zu `df`

```
/      :Disk space: 124.85 MB of 791.11 MB available (15.78%).
/stand :Disk space:   7.81 MB of  20.00 MB available (39.05%).
/nt    :Disk space: 326.24 MB of 489.84 MB available (66.60%).

Total Disk Space: 1012.43 MB of 2200.09 MB available (46.02%).
```

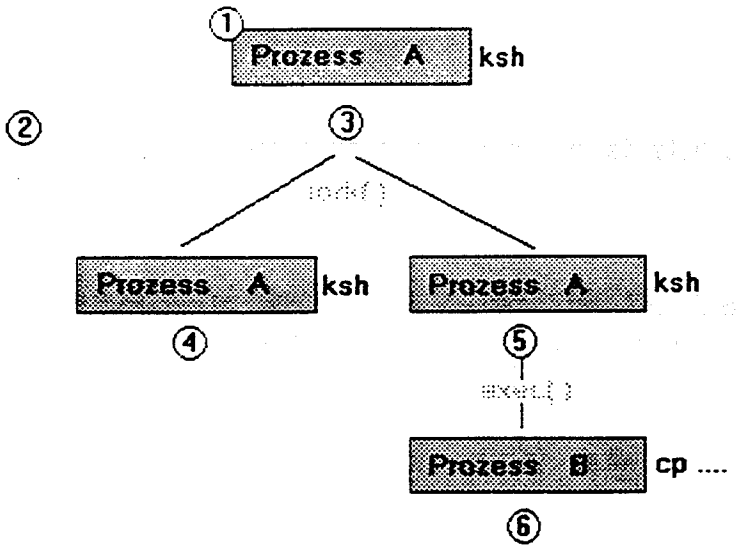
Prozesse / Prozess Status Liste

Wie kommt es zu einem Prozess?

- Ein Programm wird vom Disk in den Speicher kopiert (*exec*), eine Umgebung (*Environment*) aufgebaut, der Eintrag in die Prozess Tabelle des Kernels vorgenommen, usw.

Was geschieht, wenn über die Tastatur oder innerhalb eines Shell Skripts die Durchführung eines Programmes, z.B. `cp filex filey` verlangt wird?

- Prozesse in einem UNIX System entstehen durch Abspaltung eines bestehenden Prozesses. Dadurch entstehen für kurze Zeit zwei identische Prozesse. Diese Prozess Verdoppelung wird durch einen System Call (*fork*) vom Kernel auf Verlangen der Shell initialisiert. Aus dem abgespalteten Prozess entsteht sogleich ein neuer Prozess mit *exec*, gemäss obigem Beispiel also der `cp` Prozess.



`ps [-option ... ]` oder top (display top CPU processes)

Funktion: Gibt Informationen über Prozesse im System. Abhängig von den angegebenen Optionen, werden Informationen über alle Prozesse, die von einem, mehreren oder keinem Terminal ausgehen, in langer oder kurzer Beschreibung ausgegeben.

Optionen:

`-a`	(all) Informationen über alle eigenen Prozesse.
`-l`	(long) Eine ausführliche Informationsliste wird ausgegeben.
`-f`	(full) Eine vollständige Informationsliste aller eigenen Prozesse wird ausgegeben.
`-e`	(every) Informationen zu allen Prozessen auf dem System werden ausgegeben.
`-u` *`user`*	(user) Es werden Informationen zu den Prozessen der Benutzer gezeigt, die in *user* aufgeführt sind.

Beispiele:

`$ ps -f`	Prozessliste der eigenen Prozesse ausgeben

- Tippen Sie obigen Befehl ein und versuchen Sie, die Ausgabe anhand der Erklärungen auf der nächsten Seite zu erklären.

Erklärungen:

F	Flags bezüglich des Prozesses (siehe *man*)
S	Status des Prozesses (siehe *man*) S sleeping R runnable Z zombie O running
UID	User-ID Nummer (mit Option -f Loginname)
PID	Prozess-ID Nummer (wird benötigt, wenn ein Prozess via Signal in seinem Verhalten beeinflusst wird)
PPID	Parent Prozess-ID Nummer
C	Scheduling Parameter
CLS	Prioritätsklasse (Scheduling Class), nur in Verbindung mit der Option *c*
PRI	Prozess Priorität (je höher die Nummer desto tiefer die Priorität)
NI	Nice-Wert zur Prioritätsberechung (nur Time-Sharing Prozesse können einen *'nice'* Wert haben)
ADDR	Adresse im Memory (falls residenter Prozess, sonst Adresse auf Disk)
SZ	Grösse des Prozesses im Hauptspeicher in Anzahl 'klicks' (auf INTEL Computers = 4KB)
WCHAN	Grund des Wartezustandes, falls *waiting* oder *sleeping*. Fehlt, wenn der Prozess aktiv ist.
TTY	Terminal, welches mit diesem Prozess assoziiert ist
STIME	Startzeit des Prozesses
TIME	Kumulative Ausführungszeit
COMD	Befehlsname

Beeinflussen von Prozessen

`kill [-signal] processid [...]`

Funktion: Senden eines Signals an einen Prozess. Abhängig vom Signal, kann der Prozess unterschiedlich darauf reagieren.

Einige der wichtigen Signale:

SIGHUP	1	Hangup (z.B. single-user-process)
SIGINT	2	Interrupt (z.B. <CTRL>/c)
SIGKILL	9	Sofortiges Beenden
SIGTERM	15	Software Stopp - Signal
SIGSTOP	23	Prozess anhalten
SIGTSTP	24	Prozess anhalten u. in den Background schieben
SIGCONT	25	Angehaltenen Prozess weiterlaufen lassen

Beispiel:

```
$ kill -9 3142          Signal 9 der Prozessnummer 3142 schicken.
Prozess 3142 killed
```

Das völlige Verständnis für Signale und `kill` finden Sie in der C-Programmierung.

Damit ein Prozess gekillt werden kann, muss man Eigentümer des Prozesses sein, oder Superuser und die Prozess-ID Nummer des Prozesses kennen

Automatische Programmausführung

`crontab [file | -l | -r | -e]`

Funktion: Ausführung eines Programms, wiederkehrend, immer zu einem bestimmten Zeitpunkt.

Optionen:

`-l`	(list) Alle eigenen Crontab-Jobs anzeigen
`-r`	(remove) Alle eigenen Crontab-Jobs löschen
`-e`	(edit) Ein neues oder bestehendes Crontab-File editieren

Regeln und Voraussetzungen :

Benutzer können `crontab` nur dann verwenden, wenn ihr Name im File `/etc/cron.d/cron.allow` aufgeführt ist. Falls dieses File nicht existiert, wird das File `/etc/cron.d/cron.deny` überprüft.

Nur der Superuser kann diese Dateien verändern.

Existiert keines der beiden Files, ist nur der Superuser berechtigt.

Erstellen eines Crontab Jobs

Beispiel:

Jeden Montag um 20.00 Uhr soll `/home/user01/filea` nach `/backup/filesave` kopiert werden.

```
$ crontab  -e
0 20 * * 1 /usr/bin/cp /home/user01/filea /backup/filesave
```

```
1  2 3 4 5                    6
```

1	Minuten, 0-59
2	Stunde, 0-23
3	Tag, 1-31
4	Monat, 1-12
5	Wochentag, Sonntag (0) bis Samstag (6)
6	Programm mit der Angabe von absoluten Pfaden !

Mögliche Einträge pro Position:

*	Jedes Mal
1-5	1 bis 5
1,5	1 und 5
1,5,10	1,5 und 10

Überprüfen der Crontab-Jobs

```
$ crontab -l
0  20 *  * 1   /usr/bin/cp /home/user01/filea /backup/filesave
```

Ein Benutzer kann mehrere Crontab-Jobs starten lassen. Diese werden aber immer in einem einzigen Sammelfile (Crontab-File) dem System übergeben.

at [-option [job] time [day] [+increment]

Funktion: Einmaliger Start von Jobs zu einer bestimmten Zeit.

Optionen:

`-l`	Listet alle mit `at` aufgegebenen Jobs des Benutzers auf
`-r n`	Löscht den Job Nummer *n*
`-m`	Schickt eine mail Meldung wenn der Job fertig ist
`-f filename`	Liest die zuführenden Befehle vom File `filename`

Beispiele:

```
$ at 16:00 <NL>                                  (ohne Tagesangabe = heute)
/home/user01/sh.copyjob                          (mehrere Aufträge möglich)
<Ctrl> d
job 671454000.a at Fri Apr 12 13:00:00 1996      Bestätigung des Systems
```

```
$ at -l                                          Liste der at- Jobs ausgeben
job 671454000.a at Fri Apr 12 13:00:00 1996
```

```
$ at -r 671454000.a                              Job löschen
```

```
$ at now +1 day                                  Morgen um diese Zeit
Shell Befehl
<Ctrl> d
```

```
$ batch                                          irgendwann.
/usr/bin/sort /home/file1 > /home/output 2>&1
<Ctrl> d
```

Systemadministration

Wie auf jeder grösseren Computeranlage gibt es auch auf UNIX Systemen einen Systemverantwortlichen, den System Administrator. Er ist verantwortlich für die täglichen Unterhaltsarbeiten und für die saubere Funktion des Systems.

Diese Arbeiten beinhalten:

- Systemkonfiguration (Hardware und Software)
- Installation von Applikationen
- Benutzer, Disks, Bildschirme und Drucker verwalten
- Dateien auf externe Medien sichern
- Zurückholen der gesicherten Dateien

Verschiedene Superuser

Es bestehen 2 verschiedene Möglichkeiten um als Systemadministrator arbeiten zu können:

1. login als `root`
2. `su` Befehl
3. Starten eines Systemverwaltungsprogrammes

Login als `root`

Das `root` Login erlaubt den Zugang zum System vom root Prompt (#) aus. Es können alle Befehle ohne irgendwelche Einschränkungen ausgeführt werden. Da das `root` login vollständige Kontrolle über das System bringt, sollte das `root` Passwort ausser dem Systemverantwortlichen möglichst niemandem bekannt sein.

Einige Probleme können jedoch nur durch den Benutzer `root` gelöst werden. Darum sollte immer jemand erreichbar sein, der genügend Kenntnisse über das System hat, und das `root` Passwort kennt.

An vielen Systemen ist aus Sicherheitsgründen der `root` -Login nur von der Konsole aus möglich.

Beispiel:

```
Welcome to the Pentagon UNIX System
System name:    Top Gun

login: root
Password:
UNIX System V/386/486 Release 4.0 Version 3.0
gurul
Copyright  (C) 1984, 1986, 1987, 1988, 1989, 1990 AT&T
Copyright  (C) 1987, 1988 Microsoft Corp.
All Rights Reserved
Last login: Thu Nov 25 18:00:05 from 135.135.14.110
#
```

Der Befehl `su`

Der `su` Befehl erlaubt einem bereits eingeloggten Benutzer vorübergehend die `root` Erlaubnis zu erhalten. Um von einem gewöhnlichen Benutzer zu `root` zu wechseln, wird am Systemprompt ($) der Befehl `su` eingetippt. Es wird nach dem Superuser Passwort verlangt.

Beispiel:

```
$ su
password:
# id                                    Überprüfen der Identität
uid=0(root)  gid=1(other)
#
```

Durch Eingabe von <Ctrl> `d` kann zum ursprünglichen Benutzer zurückgekehrt werden. (Die Control Taste gedrückt halten und gleichzeitig die Taste `d` drücken). Falls ein weiteres mal <Ctrl> `d` gedrückt wird, wird UNIX verlassen.

Runlevel

who -r (runlevel)

less /etc/inittab

cd /etc/rc.d

cd rc3.d

ls ← alle Skripts für runlevel 3

Shellprogrammierung

Verarbeitung durch die Shell

Neben der Analyse der Kommandozeile und der Ausführung des ausgewählten Programms hat die Shell noch weitere Aufgaben:

- Programmausführung
- Programmiersprache
- Verwaltung des Environments
- Pipelineaufbau
- Variablen Dateinamensubstitution
- Ein- und Ausgabe Umleitung

Die Shell Programmiersprache ist eine Interpreter Programmiersprache, d.h. die Shell analysiert jede Anweisung und führt sie danach aus. Dadurch wird die Programmausführungszeit länger als z.B. bei einem kompilierten C Programm, d.h. bei dem die Anweisungen zuerst in maschinengerechter Form gebracht werden. bevor sie ausgeführt werden.

Während die Standard Bourne Shell die wichtigsten Merkmale einer Programmiersprache wie Schleifenkonstruktionen und Entscheidungsanweisungen aufweist, fehlen andere wichtige Merkmale wie Arrays, Datentypen und eingebaute arithmetische Operationen. Die Korn Shell verfügt über einige Datentypen, einschliesslich Integers und Arrays, sowie über eine eingebaute Integer-Arithmetik.

Vorsicht:
Kann ein Kommando innerhalb eines Shell Skripts nicht ausgeführt werden, so wird die Ausführungskontrolle an den nächsten Befehl weitergegeben. Der Shell Skript wird also nicht abgebrochen. Dieser Umstand ist besonders bei Remoteverbindungen und Datensicherungsabläufen zu beachten.

Allgemeines über Shellskripts

Ein Shell Skript ist ein mit einem Editor erstelltes File, das Anweisungen enthält. Darunter versteht man:

- UNIX Shell Befehle
- Variablen
- Kontrollstrukturen
- Parameterübergabe (Variablenübergabe)
- Start von Applikationsprogrammen

Die Möglichkeit der Shell Programmierung erlaubt dem Anwender das Schreiben seiner eigenen Tools, wie z.B. der Aufbau eines Benutzermenüs oder allgemeiner Werkzeuge für den Applikations- und Systemunterhalt. Viele Abläufe im UNIX werden aus Shellscripts gestartet

Ein Shell Skript kann mit einem der UNIX-Editoren erstellt werden. Normalerweise wird nach dem Erfassen des Skripts die execute Erlaubnis gesetzt (mit `chmod`).

Aufruf eines Shellskripts

Es gibt vier Möglichkeiten ein Shellscript aufzurufen:

	Beschreibung	**Beispiel**	**x-Erlaubnis nötig ?**	**Zusätzliche Shell ?**	**Anwendung**
1.	Shellskriptname als Befehl	`$ sh.game`	ja	ja	Normalfall
2.	Mit zusätzlicher Shell aufrufen	`$ sh sh.game`	nein	ja	Shellscript ohne x-Erlaubnis aufrufen
3.	Mit Punkt aufrufen	`$ . sh.game`	nein	nein	Wie über Tastatur
4.	Debug Modus	`$ sh -x sh.game`	nein	ja	Shellscripts testen

Variablen

Variablenbezeichnung

Variablennamen setzen sich aus folgenden Zeichen zusammen:

a-z	A-Z	0-9	_

Andere Zeichen für Variablennamen sind nicht gestattet. Der Variablenname kann maximal 255 Zeichen lang sein.

Ausgabe der Variablen

`$ echo $variable`	Der Inhalt **einer** Variablen wird durch ein vorangestelltes Dollarzeichen Zeichen angesprochen.
`$ set`	Zeigt eine alphabetische Liste **aller** im Environment existierenden lokalen und exportierten Variablen an.
`$ env`	Alle **exportierten** Variablen. Auch diejenigen die in einer Parentshell exportiert wurden.
`$ export`	Variablen die in der **aktuellen** Shell **exportiert** wurden

echo $LOGNAME → gibt Loginnamen aus

Der Befehl echo

echo gibt Text oder Variableninhalt auf den Standard Output (Bildschirm) aus.
Dieser Befehl dient vor allem der:

1) Eingabeaufforderung

z.B.: echo "Name eingeben"

2) Text- und Variablenausgabe

z.B.: echo "$PWD"

3) Erweiterung von Files via Umlenkung

z.B.: echo "this is the end" >> neu.file

Backslash Zeichen mit Sonderbedeutung (im Linux echo -e)

\b	ein Zeichen zurück (Backspace)
\c	Zeilenausgabe ohne anschliessendes Newline Zeichen
\f	neue Seite (Formfeed)
\n	neue Zeile (Newline)
\t	Tabulator (Tabzeichen)

Zeichen, deren Sonderbedeutung hinfällig ist, werden mit vorangestelltem '\' markiert:

```
$ echo "Das \$- Zeichen ist ein solcher Fall".
```

Sonderzeichen oder Text mit Sonderzeichen müssen in oder ' ' stehen!

Beispiele:

```
$ echo "\nName: .....\b\b\b\b\b\c"
Name: _.....
```

Der Cursor steht auf der Position _.

```
$ echo "\neins\nzwei\ndrei\n"
eins
zwei
drei
```

Gebrauch von Anführungszeichen und dem Backslash - Zeichen

Zeichen	Bedeutung
" . . . "	(doppelte Anführungszeichen) Die Zeichen **$** ` und \ werden interpretiert. Alle anderen Zeichen werden ignoriert.
' . . . '	(einfache Anführungszeichen) **Keine** Zeichen werden interpretiert
`befehl`	(umgekehrte einfache Anführungszeichen) Führt `befehl` aus und fügt an dessen Stelle das **Befehlsresultat** ein.
\c	(umgekehrter Schrägstrich) Nimmt die Spezialbedeutung des Zeichens *c* weg. (z.B. das Newline Zeichen am Ende der Zeile)

Was ergeben also folgende Beispiele?

```
$ echo "*"  _____
$ echo "?"  _____
$ echo *    _____
$ echo ?    _____
```

Variablentypen

Die Shell kennt vier verschiedene Shell Variablentypen, auch Parameter genannt:

1.) Benutzerdefinierte Variablen
2.) Systemdefinierte Variablen (oder Schlüsselwort Parameter)
3.) Positional Parameter
4.) Spezielle Shell Variablen.

Allgemein können diese Variablen durch den Benutzer nach Belieben geändert werden, soweit dies Sinn macht.

Benutzerdefinierte Variablen

Die Variablen werden allgemein wie folgt gesetzt:

```
variable=Wert
```

Der Inhalt einer Variable kann aus nur einem einzigen Wort oder auch aus mehreren Wörtern bestehen, falls diese dann in Anführungs- und Schlusszeichen (" ") eingepackt werden.

Beispiele:

```
$ filename=d.adr
$ echo $filename
d.adr
$ lp $filename

$ gruss="Guten Tag"
$ echo $gruss
Guten Tag

$ dir="ls -lia"
$ $dir

$ a=`date`
$ echo $a
Tue Jul 8 16:29:02 MET 1991
```

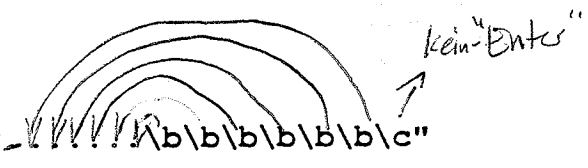
Der Befehl `read`

Als Alternative zum Konstantenübertrag in eine Variable dient `read`. Dieser, in der Shell eingebaute Befehl liest eine Eingabe bis zum <NL> vom Standard Input ein, und speichert diese in der entsprechenden Variablen ab. Die Variable kann, muss aber nicht vordefiniert sein:

```
$ read variable
```

Dies ist die einzige Möglichkeit der Shell, Eingaben in Variablen über die Tastatur vorzunehmen. Doppelte Anführungszeichen " " haben keine Sonderbedeutung beim `read` - Befehl.

Beispiel:




```
$ echo "Name:\b\b\b\b\b\c"
$ read name
$ echo $name
```

Der Befehl export

Sollen Variablen allen folgenden Prozessen oder Subshells zugänglich gemacht werden, müssen diese exportiert werden:

```
$ name=rene
$ export name
```

Systemdefinierte Variablen

Diese sind namentlich von der Shell vorgegeben und sind GROSS geschrieben. Sie können vom Benutzer inhaltlich neu gesetzt und verändert werden. Diese Veränderungen gelten aber nur bis zur nächsten Änderung oder bis zum Logout. Eine permanente Zuteilung kann über das `.profile` oder `/etc/profile` erfolgen, das ja ein spezielles Skript darstellt.

Beispiele:

PATH Pfadnamen der Befehldirectories. Dies sind die Directories, in denen die Befehle gesucht werden. Die einzelnen Directories sind in der Suchreihenfolge aufgereiht und durch einen Doppelpunkt (:) getrennt.

```
$ echo $PATH
/usr/ucb:/bin:/usr/bin
```

HOME Pfadname des Home - Directories, normalerweise das Directory nach dem Einloggen oder das Directory nach 'cd' ohne Angabe.

```
$ cp $HOME/d.adr /tmp/d.${LOGNAME}
```

PS1 System Prompt (Primary System Prompt), normalerweise das Dollarzeichen: $

```
$ PS1="What's next, Boss? "
What's next, Boss ? _
```

PS2 Secondary System Prompt, normalerweise >. Nach diesem Zeichen verlangt die Shell weitere Eingaben.

```
$ PS2="weiter:> "
$ d\
weiter:>ate"
Tue Nov 12 15:28:39 MEZ 1992
```

IFS Internal Field Separator. Separatorzeichen der Shell, um Wörter in der Befehlsliste zu trennen, normalerweise" ", <TAB> und <NL>. Quizfrage: Wie können Sie den Inhalt der Variable IFS überprüfen?

```
$IFS=:
$echo:$HOME
/home/rene
```

LOGNAME Login Name des Benutzers

MAIL Briefkastenname des Benutzers, normalerweise /var/mail/$LOGNAME

SHELL Name des Befehlsinterpreters, der bei einem Shellsprung aufgerufen wird.

Positional Parameter (Argumente der Befehlszeile)

Die Eingabe eines Shell Befehles bewirkt die automatische Ablage der einzelnen Worte in Variablen mit vorgegebenem Namen. Das erlaubt dem Benutzer, diese Variablen innerhalb eines Shell Skripts als sog. Positional Parameter über ihre $- Bezeichnung anzusprechen.

Positional Parameters sind nicht exportierbar! Man muss sie zuerst in normale Variablen ablegen. Variablennamen der Positional Parameter sind auf die Ziffern 0 - 9 begrenzt. Die Zuordnung der Argumente erfolgt sequentiell.

Die Variable $0 enthält immer den Namen des aufgerufenen Shell Skripts. Dies ist nützlich für Fehlermeldungen. Ab der Variablen $1 sind dann die Argumente der Befehlszeile zugeordnet:

	`sh.script`	`Meier`	`/home`	(fehlt hier)	(fehlt hier)		Arg*n*
Positional parameter:	**0**	**1**	**2**	**3**	**4**	**...**	**n**

Die ersten 10 Positional Parameter können über die entsprechenden Variablennamen angesprochen werden.

Beispiel:

```
$ set `ls`
$ echo $1 $2 $3
d.adr d.brief zins.c
```

Der Befehl `shift`

Mit **`shift`** kann man die Positional Parameter um n Positionen verschieben. Die Originalwerte werden durch die neuen Werte ersetzt. Wird 'shift' ohne Wert n eingegeben, so wird automatisch n=1 gesetzt. Shift kann auch einen grösseren Wert als neun annehmen.

Beispiel:

```
$ set `date`                    (Befehl set siehe anschliessend)
$ echo      $1      $2          $3      $4          $5      $6
            Tue     Nov         3       14:29:38    MET     1996
$ shift 2
$ echo      $1      $2          $3      $4          $5      $6
            3       14:29:38    MET     1996
```

Der Positional Parameter `$0` kann durch `shift` nicht verändert werden.

Ist der Wert n grösser als die Anzahl der Positional Parameter, so erscheint die Fehlermeldung `cannot shift` und alle Positional Parameter werden gelöscht. Der exit - Status des `shift` - Befehles ist dann 1 (falsch).

Dateien auf Diskette speichern

① Diskette formatieren:

fd format /dev/fd0u1440

Spezielle Shell Variablen

Diese vordefinierten Spezial Variablen sind Variablen, die die Shell innerhalb des Benutzer Prozesses setzt. Diese können vom Benutzer nur abgefragt, jedoch nicht modifiziert werden.

$? Exit Status (auch Return Code genannt) des zuletzt abgelaufenen Vordergrundprozesses (und nur von diesem!)

Fehlerfrei = 0
Mit Fehler ≠ 0

```
$ echo "Der Return Code war $?"
0
```

$# Anzahl der Positional Parameter (ohne $0)

```
$ echo "Anzahl Pos. Parameter. war $#"
37
```

$$ Prozessnummer des ablaufenden Prozesses. Verwendung als Bestandteil von Filenamen möglich:

```
$ last > /tmp/last$$
$ ls /tmp/last*
/tmp/last3465
```

$* Alle Parameter als eine einzige Zeichenkette

```
$ echo "$*"
$1 $2 ... $9
```

Variablen erweitern / ergänzen

An den folgenden Beispielen sei der Zusammenhang bei der Substituierung von Variablennamen erläutert. Bei Nichtbefolgung gewisser Regeln können sonderbare Dinge geschehen:

```
$ o=Orangen
$ b=baum
$ echo $o $b
Orangen baum
$

$ p=$o-Plantage
$ echo $p
Orangen-Plantage
```

Achtung: Falls die Variable nicht eindeutig erkannt werden kann, muss die Variable in geschweiften Klammern {}stehen.

```
$ p=$oplantage
$ echo $p              #Variable $oplantage existiert nicht

$ p=${o}plantage       #Variable ist so eindeutig gekennzeichnet
$ echo $p
Orangenplantage
```

Praktische Anwendung:

```
$ cat $HOME/filea > ${LOGNAME}.01
```

Die Shell löst die Variablen auf und führt folgenden Befehl aus:

```
$ cat /home/user00/filea > user00.01
```

Pfadvariable ergänzen mit `/etc`:

```
$ PATH=${PATH}:/etc
```

Bedingte Ausführungen

Einfache Bedingung

Es ist in Shellscripts oft der Fall, dass nur etwas ausgeführt werden soll, wenn auch bestimmte Bedingungen zutreffen. Dies kann mit folgenden Befehlen kontrolliert werden.

```
if Bedingung
 then Befehle
  [[ elif Bedingung
   then Befehle ]
  [ else Befehle ]]
fi
```

Die Bedingung in der `if . then fi` Konstruktion wird mit dem anschliessend erklärten Befehl `test` benutzt.

Der Befehl `test`

`test` kann numerische Werte (Integers), Strings und Fileattribute überprüfen. Als Resultat erhält man immer den Wert 0 (wahr) oder 1 (falsch).

Dieser Zusatz zu `if` kann auf zwei Arten angegeben werden:

1) `if test ausdruck` (explizit)

2) `if [ ausdruck ]` (implizit)

Bei der Eingabe mit den eckigen Klammern muss zwischen dem Ausdruck und den Klammern ein Leerschlag vorhanden sein!

Operatoren für den Vergleich von numerischen Werten

`-eq`	gleich (equal)
`-ne`	ungleich (not equal)
`-gt`	grösser (greater than)
`-lt`	kleiner (lower than)
`-ge`	grösser gleich (greater equal)
`-le`	kleiner gleich (lower equal)

Zur Beachtung: Folgende Werte sind identisch!

```
a="5"
a="005"
a="      5"
```

Es wird also der effektive Zahlenwert getestet.

Beispiele:

```
$ cat testfile1
if test "$?" -eq 0
 then echo "Kein Fehler aufgetaucht"
 else echo "Fehler aufgetaucht"
fi
$ testfile1
Kein Fehler aufgetaucht
$
```

```
$ a=5; export a
$ cat testfile2
if test "$a" -eq 10
 then echo "Zahl ist 10"
 elif test "$a" -lt 10
 then echo "Zahl ist kleiner als 10"
 else echo "Zahl ist grösser als 10"
fi
$ testfile2
Zahl ist kleiner als 10
$
```

Operatoren für den Vergleich von alphanumerischen Werten

Falls Zeichenketten überprüft werden sollen, muss mit den anschliessend beschriebenen `test` Operatoren verglichen werden. Die Bedingung ist erfüllt und übergibt den Exit Status 0 falls:

`string1 = string2`	Zeichenkette1 identisch ist mit Zeichenkette2
`string1 != string2`	Zeichenkette1 nicht identisch mit Zeichenkette2
`string`	Inhalt der Zeichenkette grösser Null ist und `string` von `test` gesehen wird. z.B. Variable hat einen Inhalt. (Ist gleichbedeutend mit `'test -n'`)
`-n string`	Inhalt von `string` grösser als Null ist
`-z string`	Inhalt von der Zeichenkette Null ist (und `string` von `test` gesehen wird) z.B. Ist die Variable leer ?

Falls der Inhalt der Variablen angesprochen wird, so empfiehlt es sich, die ganzen Variablen in " "-- Zeichen zu schreiben, damit auch die Leerschläge erkannt werden.

Operatoren für den Vergleich von Files

`-s file`	File ist vorhanden und die Grösse ist grösser 0
`-f file`	File ist ein gewöhnliches File
`-d file`	File ist ein Directory
`-c file`	File ist ein charakterspezifisches File
`-b file`	File ist ein blockspezifisches File
`-p file`	File ist eine Pipe
`-r file`	File ist vorhanden und hat Leseerlaubnis
`-w file`	File ist vorhanden und hat Schreiberlaubnis
`-x file`	File ist vorhanden und ist ausführbar
`-u file`	File besitzt ein gesetztes set-user-id (SUID) Bit
`-g file`	File besitzt ein gesetztes set-group-id (SGID) Bit
`-k file`	File besitzt ein gesetztes sticky Bit

Beispiel:

```
$ cat fehlertestfile
if [ -s errorfile ]
then  echo "Folgende Fehler sind aufgetaucht:"
    cat errorfile
fi
$
```

Diese Datei wird nur angezeigt falls Fehler aufgetaucht sind!

Verneinung der File Operatoren

Mittels '!'-- Zeichen und einem folgenden Leerschlag können alle Vergleichsausdrücke verneint werden.

Beispiel:

```
$ cat chx
if test ! -x "$1"
then chmod +x "$1"
fi
$ chx sh.testfile
$
```

`sh.testfile` erhält x-Erlaubnis, falls nicht vorhanden

Mehrfachbedingung

Eine Mehrfachbedingung entsteht durch eine Anzahl einfacher Bedingungen, die durch die logischen Operatoren -a (and) und/oder -o (or) verbunden sind.

-a	ausdruck1 **und** ausdruck2
-o	ausdruck1 **oder** ausdruck2

'-a' hat eine höhere Priorität als '-o'!

Bei Bedarf können Klammern verwendet werden, um die Reihenfolge der Ausdrücke festzulegen. Die Anzahl der linken und rechten Klammern muss gleich sein. Die Klammern können auch dort gesetzt werden, wo aufgrund der logischen Regeln keine nötig wären.

Da Klammern für die Shell eine Sonderbedeutung haben, müssen diese mit dem '\'-Zeichen neutralisiert werden.

Beispiel:

```
$ cat file3
if [ \( -f file1 -o -f file2 \) -a -f file3 ]
then echo "file1 und file3 sind normale Dateien oder file2 und
file3 sind normale Dateien "
fi
$
```

Mehrfachverzweigung

case erlaubt die Möglichkeit, Mehrfachbedingungen zu erstellen. Echte Alternative zu schwerfälligen Konstruktionen von `'if .. elif .. elif ..fi'`.

```
case $variable in
 muster1)       befehl1
                befehl2
                befehl3;;
 muster2)       befehl4;;
 .
 .
 muster3)       befehl5;;
esac
```

Trifft ein Muster zu, dann werden der entsprechende Befehl oder die Befehlsfolgen ausgeführt. Die Ausführung der **case - esac** Verzweigung ist damit beendet. Jedes Muster muss mit einer Klammer ')' enden, damit das Muster von der Kommandoliste unterschieden werden kann. Am Ende der Kommandoliste müssen ein Paar ';;' stehen.

Das Muster kann alle Sonderzeichen enthalten. Muster, die für die gleiche Kommandoliste gelten, können mittels '|' getrennt werden. Wird als Muster nur '*' angegeben, wird diese Kommandoliste ausgeführt, falls keines der anderen Muster zutrifft.

Beispiel:

```
$ cat sh.case
echo "Geben Sie eine Zahl zwischen 1-3 an: . \b\c"
read zahl
case $zahl in
  1) echo "Sie haben die Zahl 1 gewählt" ;;
  2) echo "Hier sind das Datum und die Benutzer:"
     date
     who ;;
  3) echo "Der aktuelle Pfad ist:"
     pwd ;;
  *) echo "Eingabe ungültig" ;; ← für alle anderen Eingaben
esac
$
```

Der Befehl `exit`

`exit` verlässt eine Shell frühzeitig.

`exit [n]` wobei n=1-256

Mit n kann ein numerischer Exit Status gesetzt und mitgegeben werden. Wird kein Wert n angegeben, so wird der Exit Status des letzten Befehles übernommen.

Beispiel:

```
$ cat exitfile
if test "$antwort" = q
 then exit
fi
$
```

Schleifen

Bis zu diesem Zeitpunkt konnten erst falsche Eingaben abgefangen werden. Der Benutzer konnte bei falscher Eingabe nicht gezwungen werden, diese so lange zu wiederholen, bis sie richtig ist. Oft ist es wünschenswert, wenn Abläufe wiederholt werden können. Hierzu dienen die Schleifen.
UNIX kennt drei Arten von Schleifen:

1. **while** Schleife
2. **until** Schleife
3. **for** Schleife

`while` Schleife

`while` lässt Befehlssequenzen wiederholen und **solange** ausführen, als die `while` Bedingung erfüllt ist, deren der exit Status 0 ist.

```
while Bedingung
do
 Befehlsliste
done
```

Die Bedingung wird jedesmal überprüft, wenn die Schleife neu gestartet wird. Solange diese Bedingung erfüllt ist, läuft diese Schleife weiter ab. Die Befehlsliste ist die Liste der Befehle, die ausgeführt werden, solange die Bedingung der `while` Schleife erfüllt ist.

Der exit Status der `while` Schleife ist der exit Wert des letzten Befehles, der in der `while` Schleife ausgeführt wurde, oder Null, wenn kein Befehl ausgeführt wurde.

Beispiel:

```
$ cat frage
antwort=nein
while test "$antwort" != "ja"
do
 echo "Schleife abbrechen? .\b\c"
 read antwort
done
$
```

Die Schleife wird solange ausgeführt bis ja eingegeben wird.

```
$ cat sh.logout
# Einem Benutzer soll eine Meldung solange zugeschickt werden,
# bis er sich ausloggt
who
echo "Wer soll ausgeloggt werden?. . . . \b\b\b\b\c"
read user
if test -z "$user"
 then echo "abbruch"
 exit 1
fi
set -- `who | grep "$user"`
while who | grep $user >/dev/null
do
 banner PLEASE \ LOGOUT \ NOW! > /dev/"$2"
 sleep 10                          #10 Sekunden warten
done
echo "Benutzer hat sich ausgeloggt"
$
```

`until` Schleife

`until` verhält sich gleich wie `while` . Die Befehlsliste wird jedoch solange ausgeführt, bis die Bedingung eintrifft.

```
until Bedingung
do
 Befehlsliste
done
```

Beispiele:

```
$ cat testuntil
antwort =nein
until test "$antwort" = "ja"
do
 echo "Schleife abbrechen?.\b\c"
 read antwort
done
$
```

```
$ cat sh.monitor
# Warten bis ein bestimmter Benutzer eingeloggt ist
if test "$#" -ne 1
then  echo "Befehlsangabe: sh.monitor Benutzername"
    exit 1
fi
user="$1"
grep "^$user" /etc/passwd >/dev/null
if test $? -ne 0
then  echo "Diesen Benutzer gibt's gar nicht!"
    exit 2
fi
until who | grep "^$user" >/dev/null
do  echo "$user hat sich noch nicht eingeloggt..."
    sleep 10
done
echo "$user ist eingeloggt" > `tty`
$
```

Zusammenfassend:
`while` lässt eine Schleife solange ablaufen, **solange** die Bedingung erfüllt ist.
`until` lässt eine Schleife solange ablaufen, **bis** die Bedingung erfüllt ist.

Der Befehl `true`

Funktioniert gleich wie das leere Kommando ':'. Liefert immer den exit - Status 0. Wird häufig für unendliche Schleifen gebraucht aus denen oft erst beim Eintreffen einer bestimmten Bedingung explizit ausgebrochen werden soll.

Beispiel:

```
$ cat eingabefile
#Solange Namen in Liste schreiben bis STOP eingegeben wird
while true
do echo "\nBitte Namen eingeben:......\b\b\b\b\b\b\c"
   read name
   echo "(Eingabe mit 'stop' beenden)"
   case $name in    # mit 'stop' aus dem Loop brechen
   [Ss][Tt][Oo][Pp]) exit ;;
   esac
   echo $name >>maillist
done
$
```

Der Befehl `Null`

Das ':'- Zeichen ist ein leeres Kommando, welches nichts ausführt, aber immer den exit - Status 0 setzt.

Der Befehl `false`

Dieses Kommando macht nichts anderes, als dass es immer den exit - Status 1 setzt. Wird wie `true` häufig im Schleifenkopf verwendet.

for Schleife

Es gibt 2 ähnliche Arten der for Schleife:

1. Worte fix in der Wortliste
2. Worte als Positionalparameter

Worte fix in der Wortliste

for lässt eine do ... done Befehlsreihenfolge so lange ablaufen, wie Worte in der Wortliste vorhanden sind.

```
for Variable in Wortliste
do
 Befehlsliste
done
```

Die Wortliste ist eine Sequenz von Wörtern, die durch Leerschläge getrennt sind. Diese Werte werden pro do ... done Durchlauf nacheinander der Variablen in der for Schleife zugewiesen.

Beispiel:

```
$ cat lpfile
for name in sc.adrmut sc.adrlist sc.adrsave
do
 lp $name
done
$ lpfile            #Alle angegebenen Files werden ausgedruckt
Request-id is wenger-1024
Request-id is wenger-1025
Request-id is wenger-1026
$
```

Positional Parameter in der Wortliste

Fehlt bei der **for** Schleife die Konstruktion in Wortliste, so werden die Positional Parameter für die for Schleife gebraucht. Vorteil: Die Kontrolle über das Ende der Positional Parameter wird durch 'for' gewährleistet.

```
for Variable
do
 Befehlsliste
done
```

Beispiel:

```
$ cat for.lp
#Alle Shellscripts als Positionalparameter setzen
set `ls sh.*`
for script
do
 #Alle Shellscripts ausdrucken
 lp $script
done
$
```

oder das gleiche mit einer **while** Schlaufe:

```
$ cat while.lp
set `ls sh.*`
#Test ob noch Positionalparameter da sind
while test "$#" -gt 1
do
 lp $1
 shift 1
done
$
```

Schleifen Control Befehle

Für das Kontrollieren der Schleifen werden folgende Befehle verwendet:

```
break [n]
continue [n]
```

Sie können nur bei Schleifen eingesetzt werden, die eine `do .. done` Befehlssequenz ablaufen lassen, also bei `until`, `while` und `for`. `break` und `continue` ohne Angabe kontrollieren die aktuelle Schleife. Nach `break` wird am Ende der Schleife(n) weitergefahren. nach `continue` wird die Kontrolle an den Schleifenkopf zurückgegeben.

continue

```
while - until - for
do
:
continue
:
done
```

Soll mehr als nur eine Schleife übersprungen werden, wird dies mit dem Wert n angegeben. Dieser Wert gibt an, wieviele Subschleifen übersprungen werden sollen. `continue 2` würde also 2 Schleifen zurück gehen.

break

```
while - until - for
do
:
break
:
done
```

Der Befehl `break` springt immer zur Zeile nach dem `done`. Sollen mehrere `done` übersprungen werden, wird dies mit dem Wert n angegeben. `break 2` würde also 2 `done` überspringen.

Beispiele:

```
$ cat sh.name
while true
do
 echo "Name: ..........\b\b\b\b\b\b\b\b\b\b\c"
 read name                          # Eingabe
 if test -z "$name"                 # Ist die Variable leer?
   then continue                    # Zurück zum Schleifenkopf
   else break                       # Austritt aus  'do ... done'
 fi
done
echo "Name ist $name"
$
```

```
$ cat sh.del
# Shellscript, um Dateien zu löschen,
# damit sie nachher wieder geholt werden können.
if test "$#" -eq 0
then echo "Aufruf Filename(n)" ;
exit 1                              #Shellscript wird verlassen
fi
echo "Sollen folgende Files gelöscht werden?"
echo "`ls -x "$*"`\n"
while true
do  echo "Antworten mit j/n: .\b\c"
    read antwort
    case $antwort in
    j|J|y|Y ) break ;;              #Springt auf Zeile nach 'done'
    n|N)    exit ;;                 #Verlässt Shellscript
    *)      echo "Eingabe ungültig" ;;
#Springt  zu while true nach oben
    esac
done
#Falls Directory /rubbish nicht vorhanden
if test ! -d "/rubbish"
then        mkdir /rubbish 2>/dev/null
    if test $? -ne 0
    then echo "Directory /rubbish kann nicht erstellt werden"
; exit 2
    fi
fi
for file in `ls "$*"`
do  cp $file /rubbish
    /usr/bin/rm $file
done
echo "Gelöschte Files können aus /rubbish zurückgeholt werden"
$
```

Weitere UNIX Befehle

In diesem Kapitel werden weitere Shell Befehle behandelt, die sehr gut für die Shellprogrammierung eingesetzt werden können.

Der Befehl `eval`

`eval` kann vor eine beliebige Befehlssequenz, bestehend aus ein oder mehreren Befehlen gestellt werden. `eval` wird dann eingesetzt, wenn die Kommandozeile eine oder mehrere Variablen enthält. Vor allem wenn die Variablen Zeichen wie ; , | < > enthalten, die die Shell in unverpackter Form in der Kommandozeile erwartet, ist `eval` unabdingbar.

```
eval kommandozeile
```

Die ganze Kommandozeile wird mit einem vorgesetzten `eval` zweimal von der Shell bearbeitet, bevor sie ausgeführt wird.

Beispiele:

```
$ pipe="|"
$ ls $pipe wc -l
|: No such file or directory
wc: No such file or directory
-l: No such file or directory
$
$ eval ls $pipe wc -l
16
```

Beim ersten Durchlauf ersetzt die Shell in der Kommandozeile 'pipe' durch seinen Wert "|". Durch `eval` wird die Shell veranlasst, die Kommandozeile ein zweites Mal zu durchlaufen. Dabei interpretiert sie das Pipe - Zeichen.

```
$ sl='eval cd /home/rene ; pwd'
$ $sl
/home/rene
```

Diese Befehlsvariablen können im `.profile` angelegt werden. Exportieren nicht vergessen!

Der Befehl `ulimit`

Mit diesem Befehl kann die maximale Grösse eines einzelnen Files beschränkt werden.

```
ulimit size
```

`size` gibt die Grösse in Anzahl physischer Blöcke (512 Bytes)an.

Beispiel:

```
$ ulimit 1000
```

Jetzt können nur noch Files mit einer maximalen Grösse von 512 KBytes erstellt werden.

```
$ ulimit
1000
```

Wird `ulimit` ohne Angabe einer Grösse angegeben, so wird die aktuelle Einstellung angezeigt. Default Wert für `ulimit` ist: 65536.

Der Befehl `type`

Dieser Befehl ist sehr nützlich, wenn man wissen will, welcher Art ein Befehl ist. Ein Befehl kann als Funktion, als Shell Programm, als Shell eingebauter Befehl oder als UNIX Standard Befehl definiert sein.

```
$ type pwd
pwd is a shell built-in

$ type cat
cat is /bin/cat
```

Der Befehl `expr`

Mit `expr` können arithmetische und logische Operationen ausgeführt werden. Der Befehl `expr` liefert den Exit Status 1 falls die Bedingung wahr ist, ansonsten den exit Status 0, wenn die Bedingung falsch ist. (Genau umgekehrt wie bei `test`!)

Folgende Vergleichsoperationen sind möglich:

\<	kleiner
\<=	kleiner gleich
=	gleich
!=	ungleich
\>=	grösser oder gleich
\>	grösser

Werte vergleichen:

Sind die beiden Ausdrücke Zahlen, die miteinander verglichen werden, dann wird ein numerischer Vergleich ausgeführt, ansonsten ein Vergleich mit Strings.

Beispiele:

numerisch:

```
$ x=7
$ y=9
$ expr $x \< $y              # Ist $x kleiner als $y?
1                            # Ja!   wahr = 1,  falsch = 0 !!!
```

alphanumerisch:

```
$ x=0
$ export x
$ x=`expr $input = $output` # beide Namen gleich: 1
                            # beide Namen ungleich: 0
```

Arithmetische Operationen

`ausdruck1 + ausdruck2`	Addition von ausdruck1 und ausdruck2
`ausdruck1 - ausdruck2`	Subtraktion von ausdruck1 und ausdruck2
`ausdruck1 \* ausdruck2`	Multiplikation von ausdruck1 mit ausdruck2
`ausdruck1 / ausdruck2`	Division von ausdruck1 durch ausdruck2

Beispiele:

```
$ laenge=100
$ breite=200
$ flaeche= `expr $breite \* $laenge`
$ echo $flaeche
20000
```

```
$ cat countdown
n=10
while test $n -gt 0
do  echo "$n"
    n=`expr $n - 1`     # Nach jedem Durchgang 1 abziehen
    sleep 1
done
echo "START"
$
```

Der Befehl `tput`

`tput` kann Informationen aus dem 'terminfo' - File (unter dem Directory `/usr/lib`) herausholen. So können spezifische Bildschirmsteuerungszeichen auch für die Shell benutzt werden:

```
tput [-Ttermtype] attribut
```

`Ttermtype`	Angabe des Bildschirmtypes. Fehlt diese Angabe, so wird der Wert der Variablen TERM entnommen (z.B. TERM=vt220). Wichtig, wenn Prozesse im Hintergrund laufen sollen!
`attribut`	Variablennamen der entsprechenden Bildschirmsteuer-variablen:
`blink`	Blinken
`smso`	Reverse video
`rmso`	Reverse video ausschalten
`smul`	Underline
`bold`	Doppelte Intensität
`sgr0`	Alle Attribute ausschalten
`clear`	Bildschirm löschen
`cuu1`	Cursor up
`cud1`	Cursor down
`cuf1`	Cursor forward
`cup row column`	Cursor auf Zeile `row` und Spalte `column` positionieren

Beispiel:

```
$ tput blink
$ echo "Fehler in Eingabe"
$ tput sgr0
```

Normalerweise werden diese Attribute am Anfang des Skripts in Variablen abgelegt, die dann in einen Text eingebunden werden können:

```
$ smso=`tput smso`
$ rmso=`tput rmso`
$ echo "${smso}Fehler in Eingabe${rmso}"
```

Der Befehl `trap`

Mit dem `trap` Befehl kann man spezielle Signale von der Shell abfangen. Es empfiehlt sich die `traps` erst ganz am Schluss in den Shellscript einzubauen, wenn man sicher ist, ob er auch wirklich fehlerfrei läuft.

Signal	Grund
0	Shell wird verlassen
1	Unterbruch der Verbindung
2	Interrupt (^C)
9	Kill. Wirkt immer tödlich. (Kann als einziges Signal nicht abgefangen werden.
24	Shellscript anhalten (^Z)

Normalerweise wird die Shell durch die Signale 1 oder 2 unterbrochen. Mit `trap` können solche Signale abgefangen werden.

```
trap ´befehlsliste´ signal(e)
```

Traps werden vor allem in kritischen Abläufen in Shellscripts gesetzt. Wenn irgendeine Aktion gestartet wird, dann muss diese Aktion oft auch ganz zu Ende geführt werden. Dass in solchen Situationen nicht mit ^C abgebrochen werden kann steht der trap Befehl zur Verfügung.

`trap "" signal(e)`	Signale werden abgefangen
`trap signal(e)`	Shell reagiert wieder auf die Signale
`trap`	listet alle gesetzten Traps

Beispiel:

```
$ cat sh.trap
trap ´echo "Kein Abbruch möglich mit ^C !"´ 2
while true
do
 echo $i
done
trap 2
$
```

Beim eintreffen des Signals 2 wird die Meldung `Kein Abbruch möglich mit ^C am Bildschirm` erscheinen.

Bücherliste

Buchtitel	Autor	Verlag	Buchnummer (ISBN)
UNIX Schnellübersicht	W. von Thienen	Markt & Technik	3-87791-421-7
UNIX Grundlagen	H. Herold	Addison-Wesley	3-89319-306-5
UNIX-Leitfaden	P. Norton	Markt & Technik	3-87791-024-6
Der UNIX Werkzeugkasten: Progr. mit UNIX	Kerningham & Pike	Hanser Verlag	3-446-14273-8
UNIX Systemverwaltung	H. Richter	Addison-Wesley	3-89319-297-2
UNIX Shell Programmierung	S. Kochan & P. Wood	te-wi Verlag	3-89362-084-2
UNIX-Shells	H. Herold	Addison-Wesley	3-89319-318-2
UNIX - Wie funktioniert das Betriebssystem?	M.J. Bach	Hanser	3-446-15693-3