Einführung in

UNIX

+

CSH

Ruedi Strub
Version 2.2

[1]

[1] Last changes: 29.6.1999

Inhaltsverzeichnis

Befehl	Syntax	Beschreibung	Seite	Beispiel

switch	switch (string)
	case muster1:
	kommandos
	breaksw
	case muster2:
	kommandos
	breaksw
	default:
	kommandos
	breaksw
	endsw
		Mehrfachverzweigung. In den case - Mustern könne auch Joker (Wildcards) ? * [ ] vorkommen. Mit breaksw erfolgt ein Sprung nach endsw. Fehlt breaksw werden die folgenden case - Bedingungen ebenfalls abgearbeitet.	25	`switch (‘line‘)`
				`case [jJ]:`
				`echo "J"; japrogr`
				`breaksw`
				`case [nN]:`
				`echo N"; neinprogr`
				`breaksw`
				`default:`
				`echo 'Nur j oder N eingeben!'`
				`endsw`

time	time kommando	Nach Schluß von kommando wird dessen Ausführungszeit angezeigt, Ohne Parameter die Laufzeit der Shell + Childs		`time sort liste.dat` `time # Laufzeit Shell und Kindprozesse`
umask	umask nnn	XOR - Maske für die Zugriffsrechte auf Dateien, die der User neu anlegt. Gibt es auch als UNIX-Kommando		`umask 023 # erzeugt rw-r--r--`
unalias	unalias name	Hebt das Alias name wieder auf (siehe auch alias)	21	`unalias ll`
unhash	unhash	Löscht Hash-Tabelle. C-Shell sucht in den path-Directories	17	`unhash`
unlimit	unlimit [-h] [resource]	Hebt Beschränkung einer Resource auf (siehe auch limit) -h (hard) : nur root-User kann Einstellung wieder ändern		`unlimit filesize` `unlimit # alle limits werden aufgehoben`
unset	unset varnames	Entfernt eine oder mehrere lokale Variable (siehe auch set)	21	`unset a1 a2 a3`
unsetenv	unsetenv varname	Entfernt (globale) Environment-Variable (siehe auch setenv)	21	`unsetenv LD_LIBRARY_PATH`
wait	wait	Warten auf Abschluß aller Hintergrundprozesse oder Interrupt (CTRL-C)		`wait`

while	while (ausdruck)
	kommandos
	end
		Schleifenbefehl: Schleife wird solange durchlaufen, wie die Bedingung ausdruck erfüllt, bzw. ungleich 0 ist	26	`while ($a) # Zahl in a gibt Anzahl vor`
				`echo $a ; @ a --`
				`end # bei repeat nur 1 Kommando!`

@	@ varname = ausdruck @ varname[ ] = ausdruck	Berechnet arithmetischen ausdruck und weist ihn varname zu ohne Parameter : die Werte aller Variablen werden angezeigt (wie set)	25	`@ sum++ # Erhoeht sum um 1` `@ c = a*b # Berechnet Produkt`

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Banahan, Rutter: *UNIX*, Hanser Verlag 1987, ISBN 3-446-13975-3

[2] Boes, Reimann: *UNIX System V*, BHV Verlag 1994, ISBN 3-89360-029-9

[3] Gulbins: *UNIX V. 7 bis V.3*, Springer Verlag 1988, ISBN 3-540-19248-4

[4] Daniel Gilly: *UNIX in a Nutshell*, Reilly Associates Inc. Second Ed. 1992, ISBN 1-56592-001-5

[5] Hetze, Hondel u.a.: *LinuX Anwenderhandbuch*, LunetIX Softfair 1994, ISBN 3-929764-02-4

Befehl	Syntax	Beschreibung	Seite	Beispiel

onintr	onintr [label] [-]	Sprungziel in Script bei Interrupt (CTRL-C) - : Intr.signal wird ignoriert (ohne Param.) : Intr.signal wird wieder akzeptiert	27	`onintr intprgm # intprgm muß existieren`
				`onintr - # Intr. wird übergangen`
				`onintr # wird wieder akzeptiert`

popd	popd [+n]	Macht cd zum untersten Directory auf Stack und verschiebt diesen, bzw nimmt n-ten Eintrag und verschiebt um n		`popd` `popd +2`
pushd	pushd [name] [+n]	Wechselt ins Directory name und schiebt Namen des alten Directory auf Stack bzw. tauscht untersten mit n-tem Eintrag		`pushd /usr/bin` `pushd +4`
rehash	rehash	Baut die Hash-Tabelle mit Variablen path neu auf	17	`rehash`
repeat	repeat n kommando	Schleifenbefehl: das kommando wird n - mal ausgeführt	27	`repeat 5 cd ..`

set	set varname = wert set varname [n] = wert set	Weist lokalen Variablen varname den Wert wert zu Arrayzuweisung Anzeige aller lokaler Variablen	21	`set tag = (mo di mi do fr sa so)`
				`set tag [3] = Mittwoch`
				`set x = 1 y = 2 z = 5`
				`set flag # erzeugt leere Variable`

setenv	setenv [varname [wert]]	Weist der Environment-Variablen (global) varname den Wert wert zu. Werden automatish exportiert	21	`setenv LD_LIBRARY_PATH /usr/lib` `setenv # Anzeige aller Env.Variablen`

shift	shift varname	Die Parameter in der Kommandozeile werden eins nach links verschoben bzw. die Elemente des arrays varname		`shift # $1 -> $0; $2 -> $1 etc.`
				`shift tag # tag[2] -> tag[1]; tag[3]-> ..`
				`while ($#argv) # solange Argumente da`
				`echo $argv[1] ; shift`
				`end`

source	source [-h] scriptname	Startet Kommando innerhalb des Shellprozesses ohne eigene PID. -h : die Kommandos des Scripts werden in die history -liste übernommen aber nicht ausgeführt	17	`source /.cshrc` `source -h testcom`
stop	stop [jobID]	Hält einen Hintergrundprozeß an	18	`stop %1` `stop # hält den current job an`
suspend	suspend	Hält den Vordergrundprozeß an. Entspricht CTRL-Z		suspend

1 Die Bedeutung von UNIX

Das Betriebssystem UNIX ist ein Multiuser- Multitasking Betriebsystem. Dies bedeutet daß gleichzeitig mehrere Programme (s.g. Prozesse) bearbeitet werden und mehrere Nutzer gleichzeitig auf dem Rechner arbeiten können. *Gleichzeitig* bedeutet in diesem Fall, dass nach bestimmten Zeiträumen zyklisch zwischen den einzelnen Prozessen umgeschaltet wird und so eine pseudogleichzeitige Bearbeitung erreicht wird.

Seine große Verbreitung verdankt UNIX drei Umständen:

- Der größte Teil des Quellcodes ist in der Programmiersprache C geschrieben und nur ein kleiner Teil in Assembler. Die Portierung auf einen neuen Rechner (mit einem neuen Prozessor) ist daher relativ einfach, da lediglich der Assemblerteil neu geschrieben werden muß, und der C-Teil auf einem entsprechenden Cross-Compiler übersetzt werden kann. Es ist daher auf allen Plattformen vorhanden.
- Aufgrund günstiger Lizenzbedingungen verbreitete sich UNIX sehr schnell an Universitäten und Hochschulen, wo in großem Umfang s.g. Public-Domain-Software entstand. (X11, GNU....). Seit 1991 existiert mit Linux ein freies Unix.
- In praktisch allen UNIX-Systemen sind die Netzwerkkomponenten für das TCP/IP-Protokoll enthalten, welche die Grundlage für das Internet bilden.
- Die Stabilität von Unix übertrifft diejenige der meisten anderen Betriebssysteme.

Diesen Vorzügen stehen jedoch auch Nachteile gegenüber, die insbesondere in den 80er-Jahren seine Verbreitung einschränkten:

- Eingeschränktes Echtzeitsystem, daher nicht geeignet für schnelle Steuer- und Regelzwecke oder Meßwerterfaßung.
- Kein einheitlicher Prozessor auf verschiedenen UNIX-Rechnern, i.d.R muß ein Programm für jeden Rechner neu kompiliert werden.
- Wesentlich geringeres kommerzielles Softwareangebot mit etwa doppeltem bis dreifachem Preis, gegenüber MSDOS/Windows-System.

Der Einsatz von UNIX war bisher fast ausschließlich auf Workstations bzw. noch leistungsfähigere Maschinen beschränkt. Mit LINUX für PC's ab 386 setzte sich erstmals ein UNIX-System in beachtlichem Umfang auch auf PC-Ebene durch, ohne bisher jedoch die dominierende Rolle von DOS/Windows auf diesem Gebiet anzugreifen.

2 Der Aufbau von UNIX

Der Aufbau von UNIX ist streng schalenartig. Im Zentrum befindet sich die Hardware, umgeben von dem s.g. Kernel. Auf diesem bauen die Utilities aber auch die Anwendersoftware auf. Die Hardware und damit die Maschineneigenschaften des Rechners bleiben dem Nutzer damit verborgen.

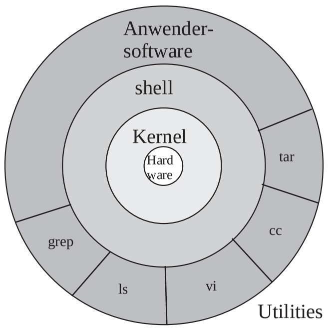

Abbildung 1: Aufbau des UNIX-System

Der Kernel kann in die folgenden Module zerlegt werden:

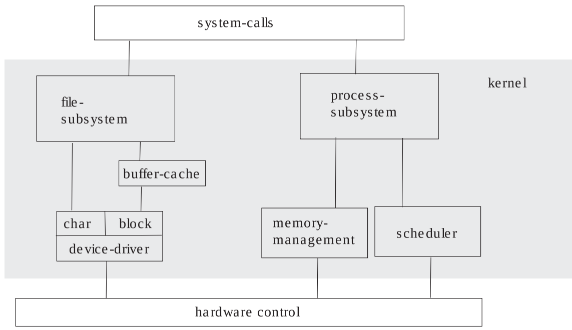

Abbildung 2: Struktur des UNIX-Kernels

Das **process-subsystem** verwaltet alle im System laufenden Prozesse (Programme). Dazu wird ein **memory-management**-Modul benötigt das die Speicherverwaltung (einschließlich des virtuellen Speichers im Swapbereich (siehe 4.8) der Festplatte) übernimmt und ein **scheduler** der die Umschaltung zwischen den Prozessen steuert.
Das **file-subsystem** hat die Aufgabe den oder die Massenspeicher (Festplatte(n)) zu verwalten. Dabei unterscheidet man block- und zeichenorientierte

Befehl	Syntax	Beschreibung	Seite	Beispiel

foreach	foreach varname (liste) kommandos end	Schleifenbefehl. Für jeden in liste stehenden Wert von varname wird die Schleife einmal durchlaufen	26	`foreach tag (samstag sonntag montag)`
				`echo $tag`
				`end`

glob	glob wortliste	Wie echo, jedoch keine Aufhebung der Metazeichen mit \	16	
goto	goto marke	Sprungbefehl in einem Script zu marke:	26	`goto ende`
hashstat	hashstat	liefert Statistisk der Wirkung der Hashtabelle	17	`hashstat --> 5 hits 0 misses 100%`
history	history [-h] [-r] [n]	Zeigt die History-Liste: -h ohne Nummern, -r umgekehrte Reihenfolge, n nur die letzten n Kommandozeilen	19	`history        # zeigt gesamte Liste` `history 5      # zeigt die letzten 5`

if	if (ausdruck) kommando; if (ausdruck) then kommandos [else [if (ausdruck) then] kommandos endif	Bedingte Kommandoausführung. Kann mehrfach verschachtelt werden. (Einzelnes Kommando kommt ohne then und endif aus)	25	`if ( $div == 0 ) goto error`
				
				`if ($name != $user) && ($UID != 0) then`
				`echo "$name hat keinen Zugriff !"`
				`exit 1`
				`endif`

jobs	jobs [-l]	zeigt alle Jobs des Users an. -l : mit PID	18	`jobs -l`
kill	kill [-l] [signal] [ID]	Beendet Prozeß : ID kann PID oder %jobnr sein Signal kann Nummer oder Name sein, -l listet Namen	10 11	`kill -l` `kill -9 %2`

limit	limit [-h] [resource[limit]]	Ohne Parameter werden die Resourcen des current process angezeigt. Die Limits können dann eingestellt werden. Die einstellbaren Resourcen sind systemabhängig	28	`limit            # zeigt alle Res.werte`
				`limit filesize   # zeigt max.  Filesize`
				`limit cputime 5m # cputime <= 5 Min.`

line	line	Nimmt eine Zeile von der Tastatur entgegen	25	`set eingabe = ‘line‘`
login	login [user] [-p]	logout und anschließendes login für user. –p rettet enviroment		`login mueller -p`
logout	logout	Beendet login-Shell. Wichtig falls ignoreeof gesetzt ist		`logout`
nice	nice [+/- n] kommando	Ändert die Prozeß-Priorität von kommando. n = -20 ... +20 Default-n = 4 (in Bourne-Shell existiert /usr/bin/nice renice) + n : Addiert n zum nice-Value (verringert Priorität) - n : Subtrahiert n vom nice-Value (erhöht die Priorität)	9	`nice -5 sort liste.dat` `nice     # default für aktuelle shell`
nohup	nohup [kommando]	Start von kommando, so daß es nach logout weiterläuft. SIGHUP-Signal (Seite 20) wird ignoriert	17	`nohup # aktuelle shell ignoriert SIGHUP` `nohup sort liste.dat`
notify	notify [jobID]	Meldung über Statusänderung (Ende) eines Hintergrundprozesses erfolgt sofort.		`notify %3` `notify     # current background is default`

Datenübergabe an den Massenspeicher.
Die Oberfläche zu den Utilities und den Anwenderprogrammen bilden die - bei allen Rechnern einheitlichen - **system-calls** (Systemaufrufe). Sie entsprechen den DOS- und BIOS-Interrupts unter MS-DOS.

3 Das UNIX-Filesystem

3.1 Dateien und Verzeichnisse

Das Filesystem von UNIX ist baumartig aufgebaut. Jeder Knoten kann Wurzel eines Teilbaumes sein. Es ist streng hierarchisch aufgebaut. Das Haupt- oder root-Verzeichnis (root-directory) wird mit '/' bezeichnet. Zur Bezeichnug von Pfaden werden die Namen der Subdirectories mit dem Slash '/' - getrennt (im Gegensatz zum Backslash '\' , bei MSDOS). Die Bezeichnug eines Directories kann absolut (ausgehend vom Rootverzeichnis) oder relativ (ausgehend vom aktuellen Verzeichnis) erfolgen. Das aktuelle Verzeichnis wird mit einem Punkt '.' bezeichnet, das darüberliegende (parentdirectory) mit zwei Punkten '..'. In jedem Verzeichnis können Dateien oder weitere Directories nebeneinander stehen.

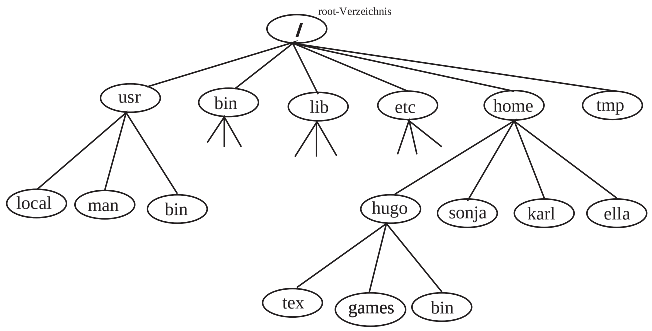

Abbildung 3: UNIX-Filesystem

Jedes Verzeichnis kann "mountpoint" für eine weiter Festplatte oder Partition einer Festplatte sein. Beim Zugriff auf ein solches Verzeichnis erscheint dann die Filestruktur dieser Platte bzw. Partition. Über ein Systen Namens NFS (Network File System) lassen sich auch Filesysteme anderer Rechner oder Teile davon übers (Inter-)Netz mounten und erscheinen dann als Teil des Filesystems des lokalen Rechners.

3.2 Zugriffsrechte

Der Umstand, daß auf einem UNIX-Rechner gleichzeitig mehrere Nutzer arbeiten, macht es erforderlich die Zugriffsrechte für jede Datei und jedes Programm festzulegen. Jede Datei hat daher einen Besitzer, der diese Zugriffsrechte bestimmt. Besitzer ist derjenige, der die Datei erzeugt hat. Er kann die Rechte lesen, schreiben oder ausführen für sich, die Nutzergruppe zu der er gehört und für den Rest der Welt getrennt vergeben. Diese Zugriffsrechte werden zusammen mit dem Inhaltsverzeichnis mit dem Befehl `ls -l` angezeigt:

Befehl	Syntax	Beschreibung	Seite	Beispiel
#	#	Kommentar, der Rest der Zeile nach # wird ignoriert		`# Kommentare sind in 2 Wochen nützlich!`
#!shell	#!shell	Gibt in der ersten Zeile des Scripts die verwendete Shell an, mit Pfad und ev. Optionen		`#!/bin/csh` `#!/bin/csh -f`
:	:	Leeres Kommando, liefert Exit-Status 0		
alias	alias [name [kommandos]]	Weist der Kurzform name die kommandos zu	21	`alias ll ls -l` `alias # zeigt alle aliase an`
bg	bg [jobid]	Schiebt Prozeß in den Hintergrund	18	`bg %1`
break	break	Bricht den laufenden Schleifenbefehl (while, foreach) ab	27	
breaksw		siehe switch	25	
case		siehe switch	25	
cd	cd [directory]	Wechsel des working directory		`cd /etc # führt ins Verzeichnis /etc` `cd # Wechsel ins home-Directory`
chdir	chdir [directory]	wie cd, nützlich wenn cd per alias neu definiert		
continue	continue	überspringt einen Schleifendurchlauf (while, foreach)	27	
default		siehe switch	25	
dirs	dirs [-l]	gibt den Directory-Stack aus, (siehe auch popd und pushd -l : statt ~ wird der absolute Pfad des Home-Dir gezeigt)		`dirs` `dirs -l`
echo	echo [-n] string	Schreibt string auf stdout. -n : nach string wird kein NEWLINE ausgegeben. (es gibt auch ein /bin/echo)		`echo -n "Guten Tag Herr $user, "` `echo wie geht es ihnen?`
else		siehe if	25	
end		siehe foreach oder while	26	
endif		siehe if	25	

eval	eval kommandozeile	Ersetzt in einem 1. Durchlauf in einer Kommandozeile alle Variablen bzw. Kommandos in ' ' -Zeichen durch ihre Inhalte, bevor sie im 2. Durchlauf abgearbeitet wird.	17	`set b = $a ; set a = "Guten Tag"`
				`eval echo $b`
				–> ergibt Ausgabe : `Guten Tag`

exec	exec kommando	Startet kommando mit der PID der aktuellen Shell. Es wird kein neuer Prozeß erzeugt.	17	`exec csh # nach Änderung der .cshrc`
exit	exit ausdruck	Beendet Script und übergibt Wert von ausdruck an Elternprozeß. Ohne ausdruck $status übergeben	28	`exit 0 # alles ok` `exit 2 # Abbruch mit Fehler`
fg	fg [%jobID]	Holt den curent job in den Vordergrund Holt Job mit Nr. jobID in den Vordergrund	18	`fg` `fg %2`

```
-rw-rw-rw-  1 hugo  gruppel     4927 Sept 23 17:28  buecherliste
-rwxr-xr-x  2 hugo  gruppel    23619 Sept 24 15:30  suchprogramm
drwxrwxrwx  1 hugo  gruppel     1024 Aug  10 13:25  alte-listen
```

Tabelle 1: Directoryausschnitt

Die Zugriffsrechte werden in der ersten Spalte der Tabelle angezeigt. Das erste Zeichen dient der Kennzeichnung der Dateiart: '-' bedeutet, daß es sich um eine normale Datei bzw. ein Programm handelt, "d" charakterisert ein Unterinhaltsverzeichnis (Zeile 3 in Tabelle 1 "alte-listen"). Die nächsten 9 Zeichen stellen die Zugriffsrechte für den Besitzer, die Gruppe und den Rest der Welt (von links nach rechts) dar.

```
r = read    : Berechtigung zum Lesen der entsprechenden Datei
w = write   : Berechtigung zum Schreiben in die Datei
x = execute : a) Es handelt sich um ein ausfuehrbares Programm
                 und darf ausgefuehrt werden.
              b) In das Directory darf verzweigt werden.
```

Die Zugriffsrechte können vom Eigentümer mit dem chmod-Befehl geändert werden. Shell-Scripts müssen die r und x - Berechtigung haben.
Vergeben Sie niemals Schreibrechte in Directories an andere!

3.3 Dateiarten

Das erste Zeichen in Tabelle 1 gibt die Art der Datei an. Folgende Zeichen sind möglich:

```
- normale Datei   (plain file)   normale Datei oder Programm
d Verzeichnis     (directory)    Unterverzeichnis
c Geraetedatei    (special file) Zugang zu Geraet
p Pipe            (named pipe)   Verbindung zwischen zwei Prozessen
l symbolischer Link              Zeiger auf eine andere Datei oder Directory
```

3.4 Aufbau des Dateisystems

Der Aufbau des Dateisystems unterscheidet sich bei verschiedenen UNIX-Systemen erheblich, so daß hier nur die Grundzüge dargestellt werden. Eine Festplatte (bzw. eine Festplattenpartition) besteht aus einer linearen Reihe von Blöcken, die typisch 1024 Byte groß sind, bei grossen Festplatten meist grösser. Alle Blöcke auf einem Massenspeicher sind durchnummeriert und haben damit eine eindeutige Adresse. Diese Kette kann in 6 Bereiche unterteilt werden (siehe Abb. 4).

3.4.1 I-Nodes

Der logische Kern des Dateisystems sind die I-Nodes (Informations-Knoten). Für jede Datei existiert genau ein I- Node. Der I-Node enthält Zeiger auf diejenigen Blöcke im Datenbereich in denen die Datei zu finden ist, d.h. deren

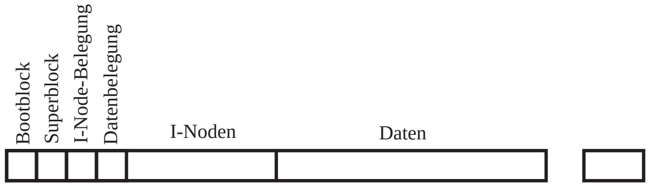

Abbildung 4: Aufbau des Dateisystems

Blocknummern. Daneben stehen in der I-Node der Dateityp, der Eigentümer, dessen Gruppe, die Anzahl der Links, die Größe der Datei, Zeiten der letzten Änderung, des letzten Zugriffs etc. **Der einzig mögliche Zugang zu einer Datei führt über den I-Node**. Directories werden hierbei ebenso behandelt wie normale Dateien und stellen Dateien dar, die sich lediglich im Typ unterscheiden. Bei einigen Dateisystemen stehen in der I-Node noch verschiedene Flags, die Dateiattribute darstellen.

3.4.2 Bootblock

Der Bootblock steht am Anfang des Datenträgers. Bei Festplatten unterscheidet man zwischen dem Masterbootrecord, der im ersten Sektor der Festplatte steht, und dem Bootsektor der im ersten Sektor jeder Partition steht. In Abb. 4 ist der Aufbau einer Partition dargestellt.
Der Bootsektor kann ein Programm zum Laden des Betriebssystems enthalten, den s.g. Bootloader. Der Bootblock wird beim Start des Rechners durch ein ROM-Programm gelesen und ausgewertet.
Sind mehrere Betriebsysteme im Rechner installiert, wird der Masterbootsektor (der 1. Festplatte) meist vom einem Bootmanager belegt.

3.4.3 Superblock

Der Superblock enthält die Angaben über Typ und Aufbau des Dateisystems (Größe des Datenträgers, Anzahl I-Nodes etc.). Die s.g. **magic number** im Superblock stellt eine Kennzahl für den Typ des Filesystems dar. Ein spezielles Byte im Superblock, das **Valid-Flag**, wird vom Kernel beim Aufsetzen des Filesystems gelöscht. Wird das Dateisystem bei einem regulären Systemhalt ordnungsgemäß abgesetzt, setzt der Kernel das Valid-Byte wieder. Bei einem Systemabsturz (oder unqualifiziertem Abschalten) bleibt dieses Flag natürlich ungesetzt, sodaß beim Hochfahren der Kernel den vorangegangenen Absturz erkennt, eine Warnmeldung abgibt und versucht das eventuell zerstörte Filesystem zu reparieren.

3.4.4 Belegungstabellen

Um möglichst schnell eine freie I-Node bzw. freie Blöcke im Datenbereich zu finden, existieren s.g. Belegungstabellen, deren Größe ebenfalls im Superblock

6.11 Übersicht: Die eingebauten Kommandos der C-Shell

Resource	Beschreibung	Einheiten
cputime	Max. zur Verfügung stehende CPU-Zeit für den Prozeß (kann mit cpu abgekürzt werden)	nh für n - Stunden; nm für n - Minuten; mm:ss für Minuten und Sekunden
filesize	Max. Größe einer von dem Prozeß erzeugten Datei	nk für n Kilobytes
datasize	Max. Größe des Datenspeicher (inkl. Stack)	nm für n Megabytes
stacksize	Max. Größe des Prozeßstacks	
coredumpsize Max.	Größe des Coredumpfiles (Prozeßkopie bei Fehler)	

Tabelle 15: Prozeßresourcen

Option	Beschreibung
-b	die folgenden Optionen werden als Optionen für ein angegebenes Kommando betrachtet, und nicht als Optionen für die csh selbst
-c	liest die Kommandos aus der Datei in arguments. Die folgenden Optionen werden in argv geschrieben und damit dieser Datei übergeben
-e	die csh wird verlassen, wenn ein Programm eine Fehlermeldung macht (exitstatus)
-f	Schnellstart: die Dateien .login und .cshrc werden beim Start der Shell nicht gelesen (immer angeben in Shellscripts!)
-i	startet eine interaktive Shell mit einem Prompt (Eingabeaufforderung)
-n	interpretiert ein Shellscript ohne die Kommandos auszuführen. (Test auf Syntaxfehler!)
-s	Liest Kommandos nur vom Standardinput
-t	nach Abarbeitung eines Kommandos wird die Shell verlassen
-v	Setzt die verbose Variable (s. S. 36). Alle Kommandos vor ihrer Ausführung auf dem Standardoutput ausgegeben und zwar so wie sie da stehen, ohne Substitution
-V	wie -v, die Variable verbose wird jedoch vor dem Lesen von .cshrc gesetzt. Dadurch wird diese ebenfalls ausgegeben
-x	Setzt die Variable echo (s.S. 36) Ausgabe der Kommandozeile nach allen Substitutionen
-X	wie -x, setzt die Variable echo jedoch vor dem Lesen von .cshrc, dadurch wird diese mit ausgegeben

Tabelle 16: Aufrufparameter der C-Shell

notiert ist. Im einfachsten Fall handelt es sich um Bitmaps, d.h. für jeden I-Node bzw. jeden Block im Datenbereich existiert ein Bit, welches gesetzt wird, wenn der I-Node bzw. der Datenblock belegt ist. In anspruchsvolleren Systemen handelt es sich um Verkettungstabellen ähnlich der FAT unter MS-DOS.

3.4.5 Links

Einträge in einem Verzeichnis bestehen aus einem Dateinamen und einem Zeiger auf den zur Datei gehörenden I-Node. Auf diese Weise ist es möglich, mehrere Verzeichniseinträge (mit verschiedenen Namen) für ein und dieselbe Datei zu erzeugen. Die Verbindung zwischen einem Verzeichniseintrag und einem I-Node wird als Link bezeichnet. Mit dem Kommando `ln` kann ein solcher Link auf eine bestehende Datei erzeugt werden. Es erzeugt einen Eintrag in dem gewünschten Verzeichnis und benutzt den I-Node-Zeiger der bestehenden Datei. Man bezeichnet einen solchen Link als harten Link. Harte Links können nur innerhalb einer Partition erzeugt werden, da nur hier die I-Nodes eindeutig sind. Um Partitiongrenzen überschreiten zu können, gibt es symbolische Links. Sie werden mit dem Kommando `ln -s` erzeugt. Symbolische Links werden als spezieller Dateityp im Verzeichnis eingetragen. In diesem Eintrag ist der Zugriffsweg auf die Zieldatei gespeichert. Damit können symbolische Links über Festplatten- und Partitionsgrenzen hinaus zeigen und sogar Links über das NFS (Network File System) auf Festplatten anderer Rechner ausführen. In jedem Verzeichnis befinden sich zwei harte Links. Diese zeigen auf das darüber liegende Verzeichnis (..) und ein Link auf sich selbst (.). Über diese Einträge werden die Verbindungen des Verzeichnisbaumes realisiert.

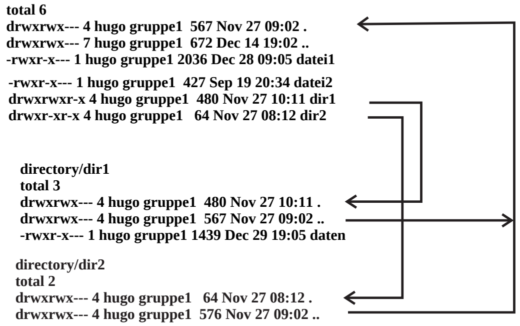

Abbildung 5: Links zwischen den Directories

4 Das UNIX-Prozeß-System

Unter einem Prozeß versteht man ein in Ausführung befindliches Programm. Jedesmal, wenn vom Nutzer ein Programm aufgerufen wird, entsteht ein Prozeß. Wird dasselbe Programm mehrmals aufgerufen, entsteht mit jedem Aufruf ein neuer Prozeß. Neben dem eigentlichen Programm gehört zu einem Prozeß die s.g. Programmumgebung. Dazu gehören u.a. Speicherbelegung, geöffnete Dateien, das aktuelle Inhaltsverzeichnis und die sichtbaren Umgebungsvariablen ($home, $path etc.).

4.1 Die Prozeßkenndaten

Als Adreßraum eines Prozesses bezeichnet man den zu einem Prozeß gehörenden logischen Speicherbereich. Die dort abgelegten Daten werden unterteilt in Benutzer- und Systemdaten. Zu den Systemdaten gehören:

- prozeßspezifische Systemdaten
- Systemstack (für jeden Prozeß).

Der Adreßraum für die Benutzerdaten wird in drei getrennte Bereiche unterteilt (Segmente)

- Textsegment: hier steht der Programmcode. Falls dieser sich nicht selbst modifiziert, kann er von mehreren Prozessen genutzt werden.
- Datensegment: hier liegen die Nutzerdaten des Prozesses. Man unterscheidet einen initialisierten und einen nicht initialisierten (bss-Segment) Bereich.
- Kellersegment: Stacksegment = Benutzerstack

Ein Prozeß kann die Größe seines Stack- und seines Datensegment (innerhalb systembedingter Grenzen) dynamisch ändern. Mit dem Kommando `size` kann die Anfangsgrösse dieser Segmente angesehen werden. Die wichtigsten Kenndaten eines Prozesses können mit dem Befehl `ps -efl` angezeigt werden:

```
 F S      UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN    STIME TTY      TIME CMD
 8 S  mueller   448     1  0  59 20 6002c020    239 6002c1f0   Apr 06 console  0:00 -tcsh
 8 S  mueller  2646  2644  0  50 20 6146c008    652 61621426 08:18:02 pts/8    0:06 nedit brief.tex
```

Das obige Beispiel entstand unter Solaris. Die angezeigten Parameter unterscheiden sich bei verschiedenen Unixsystemen und sind den Manualpages mit `man ps` zu entnehmen.

```
F     = Flag   (ohne Bedeutung, veraltet )
S     = Status (S = sleeping, O = running, R = Runnable, Z = Zombie, T = Stopped )
UID   =  effektive UserID unter dem der Prozess laeuft
PID   = Prozessnummer
PPID  = Prozessnummer des Elternprozesses
C     = Prozessornutzung (veraltet)
PRI   = momentane Prioritaet
```


jobs	zeigt alle aktiven Jobs und deren Zustand an
fg %jobnr	holt einen Job in den Vordergrund
bg %jobnr	schiebt einen suspendierten Job in den Hintergrund, wo er weiterläuft
CTRL-Z	unterbricht einen Prozeß, der im Vordergrund läuft (suspendiert ihn)
kill %jobnr	beendet einen Prozeß
stty tostop	unterbindet, daß Hintergrundprozesse unkontrolliert auf den Bildschirm schreiben. Damit wird erreicht, daß bei einer Ausgabe sich die Hintergrundprozesse unterbrechen, und warten bis sie in den Vordergrund geholt werden.

Tabelle 14: Befehle zur Jobverwaltung

> `bg`	der einzige Job der möglich ist
> `bg %`	current Job
> `bg %1`	der Job mit der Nr.1
> `bg %so`	der Job der mit “so“ anfängt
> `%&`	der einzige der möglich ist

6.9 Verwaltung der Prozeßresourcen

Jedem Prozeß werden durch das System bestimmte Resourcen zur Verfügung gestellt. Ihre Größe kann angezeigt und eingestellt werden. Dies geschieht in der C-Shell mit dem Kommando

`limit [-h] [resource [limit]]`

Die einstellbaren Resourcen sind nicht auf allen Systemen gleich. Mit `limit` (ohne Parameter) werden die Resourcen und die eingestellten Werte angezeigt. Mit dem Parameter -h werden die Werte hart festgeschrieben und können danach nur noch vom root-user vergrössert werden. Es existieren (häufig) die folgenden Resourcen:

6.10 Die Aufrufparameter der C-Shell

Wie die meisten Unix-Programme kann die C-Shell auch mit Parametern in der Kommandozeile aufgerufen werden, die ihre Eigenschaften modifizieren:

`csh [optionen] [argumente]`

Argumente sind ausführbare Programme (Scripts). Die möglichen Optionen stehen in der folgenden Tabelle:

Für die Fehlersuche (debugging) in Shell-Scripts sind die Optionen -n, -v und -x sehr nützlich.

6.6.10 break - *Abbruch der aktuellen Schleife*

Ähnlich der `continue`-Anweisung jedoch brutaler. `break` beendet die gesamte Schleife, d.h. das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.

6.6.11 exit - *Beendigung des Scriptprogramms*

Beendet den Scriptprozeß. Mit

```
exit ausdruck
```

kann der Wert von ausdruck an den Elternprozeß übergeben werden. Wird kein ausdruck angegeben, wird der Wert der Variablen status übergeben. (siehe 6.4.5 auf Seite 36)

6.7 Der Hash-Mechanismus

Beim Aufruf eines Kommandos werden die in der PATH-Variablen abgelegten Pfade abgesucht. Um diesen Vorgang zu beschleunigen, wird von der C-Shell (und tcsh) im Arbeitsspeicher eine Tabelle mit allen Dateien die sich in diesen Pfaden befinden mit den zugehörigen absoluten Pfaden angelegt. Die Tabelle wird beim Start der C-Shell angelegt. Tritt danach eine Änderung ein, sind die Tabellen mit dem Befehl `rehash` (oder Aufruf einer neuen Shell) zu aktualiseren

6.8 Das Job-Konzept

In der C-Shell ist (wie auch in der Korn-Shell) ein Job-Control-System eingebaut. Dieses System erlaubt es Prozesse, die im Hintergrund laufen in den Vordergrund zu holen, zu unterbrechen, weiterlaufen zu lassen und wieder in den Hintergrund zu schieben. Wird ein Prozeß im Hintergrund gestartet, wird die PID und die Jobnummer angezeigt. Der zuletzt in den Hintergrund gebrachte Job heisst **current job**. Wird hinter einem Job-Befehl keine Nummer angegeben, bezieht er sich automatisch auf den current job. Die Prozeßnummer wird durch ein vorangestelltes Prozentzeichen '%' gekennzeichnet: z.B.: bg %2
Folgende Befehle stehen zur Verfügung:

Mit diesen Befehlen ist es möglich Jobs im Hintergrund laufen zu lassen, die bei Ein- oder Ausgaben anhalten. Sie können bei Bedarf vom Nutzer in den Vordergrund geholt werden, wo die Ein-Ausgaben erfolgen, wonach man sie im Hintergrund wieder weiterlaufen lassen kann.

Beispiel: Ein zeitraubender Prozeß soll in den Hintergrund getan werden:

```
> sort text.dat
> CTRL-Z
```

der Job wird gestartet und suspendiert um ihn in den Hintergrund zu bringen

Nun kann der Prozeß mit einem der folgenden Befehle in den Hintergrund gebracht werden. Dort läuft er bis zu einer eventuellen Eingabe weiter:


```
NI   = Nice-Wert
ADDR = Memory-Adresse des Prozesses
sz   = Groesse des vom Prozess benutzten virtuellen Memory
WCHAN = Falls Prozess wartet: Adresse des Events, wenn Prozess running = 0
TTY  = kontrollierendes Terminal
TIME = Kummulierte CPU-Zeit des Prozesses
COMD = Name des Programms
```

4.2 Die Prozeßnummer (PID)

Jeder Prozeß ist durch eine eindeutige Prozeßnummer gekennzeichnet, die PN = Process Identification. Jeder Prozeß kann einen oder mehrere neue Prozesse erzeugen, s.g. Kindprozsse. Die Prozeßnummer des Elternprozesses wird mit PPID = Parent Process Identification bezeichnet. Dadurch entsteht über die Eltern-Kind-Beziehung eine hierarchische Prozeßstruktur. Die Wurzel dieses Baumes bildet der init-Prozeß mit der PID=1. Mit dem Kommando ps -ef kann man eine Prozeßliste erzeugen, in der man die Abstammungshirarchie der Prozesse über die PID's bzw. PPID's verfolgen kann. Mit der PID=0 läuft auf jedem Unixsystem der Prozeß swapper. Er ist für das Ein- und Auslagern von Programmen und Daten in den Hintergrundspeicher (Swapbereich der Festplatte) zuständig (siehe Abschnitt 4.8 auf Seite 18)

4.3 Erzeugung und Ende eines Prozesses

Die Erzeugung eines neuen Prozesses geschieht durch den Systemaufruf `fork`. Der neue Prozeß ist dabei eine genaue Kopie des aufrufenden Prozesses, mit Ausnahme der folgenden Parameter: PID, PPID, der Befehlszähler (programcounter) ist auf den Anfang des Programms gesetzt und der neue Prozeß hat eigene File-Deskriptoren.
`fork` ist ein Funktionsaufruf. Bei erfolgreichem Start des Kindprozesses liefert er dem Elternprozeß die Prozeßnummer PID des Kindprozesses und dem Kindprozeß eine Null zurück. Von nun an laufen beide Prozesse unabhängig voneinander weiter, wenn nicht durch einen wait-Aufruf der Elternprozeß auf das Ergebnis des Kindprozesses wartet. Soll kein identischer Kindprozeß gestartet werden sondern ein anderes Programm (der aufrufende Prozeß soll aber weiterhin bestehen), so geschieht dies in zwei Schritten:

1. Durch `fork` wird ein Kindprozeß als Kopie gestartet
2. Der Kindprozeß erkennt an der vom `fork`-Aufruf gelieferten Null seinen Kindstatus und überlagert sich durch exec mit dem neuen Programm.

Mit dem exec-Aufruf werden die Segmente des aufrufenden Prozesses durch die des neu Generierten ersetzt. Dem neuen Prozeß kann dabei die aktuelle Systemumgebung des aufrufenden Prozesses mit übergeben werden. Die Prozeßnummer bleibt erhalten.

Ein Prozeß wird über einen exit-Aufruf beendet. Dabei kann ein Statuswert (der s.g. Exit-Status) an den eventuell wartenden Elternprozeß, zusammen mit der Prozeßnummer des beendeten Kindprozesses, weitergereicht werden. Ein Exit-Status wird auch dann übergeben, wenn sich der Prozeß nicht selbst

beendete, sondern durch den Benutzer oder einen Programmfehler abgebrochen wurde. Es liefert also jeder Prozeß bei seiner Beendigung einen Statuswert zurück. Per Konvention ist dies der Wert Null, falls der Prozeß seine Aufgabe erfolgreich abgeschlossen hat und ein Wert ungleich Null in allen anderen Fällen. Auf diese Weise ist eine wenn auch sehr beschränkte Prozeßsynchronisation möglich. Stirbt ein Elternprozeß bevor alle seine Kindprozesse beendet sind, so erbt der Prozeß init (mit der PID 1) die verbleibenden Kindprozesse und wird damit deren Elternprozeß. Wird ein Kindprozeß beendet, so kann sein Prozeßkontrollblock solange nicht aus dem Hauptspeicher entfernt werden, bis dem Elternprozeß diese Terminierung mitgeteilt werden konnte, d.h. bis dieser mit `wait` auf die Beendigung des Kindprozesses wartetende Elternprozeß diese Meldung erhalten hat. Dieser Zustand (Kind ist beendet, kann aber noch nicht ausgeräumt werden) wird als **Zombie-Zustand** bezeichnet. Der Prozeß `init` wartet daher ständig auf die Beendigung eines Kindprozesses. Ein Prozeß kann mit `wait` nicht auf die Beendigung eines bestimmten Kindprozesses warten. Er bekommt als Funktionsergebnis neben dem Exit-Status jedoch auch die PID des beendeten Kindprozesses mitgeteilt.

4.4 User- und Gruppennummer eines Prozesses

Jeder Prozeß hat User- und Gruppennummern. Diese dienen der Regelung der Zugriffsrechte. Ein Prozeß besitzt stets zwei User und zwei zugehörige Gruppennumern:

- die **effektive** User- und Gruppennummer wird für die Überprüfung der Dateizugriffsrechte verwendet.
- die **reale** User- und Gruppennummer sind die Nummern des Nutzers, der das Programm aufgerufen hat.

Beim Programmstart (der Prozeßgenerierung) wird sowohl für die effektiven wie die realen Nummern diejenigen des aufrufenden Nutzerns eingetragen. Programme können mit dem **setuid-flag** versehen werden. Ist dieses Flag gesetzt, erscheint bei den Zugriffsrechten statt des "x" ein "S". In diesem Fall wird die User-ID des Programmdateibesitzers als effektive Usernummer des Prozeßes eingetragen und die Usernummer des aufrufenden Nutzers als reale Benutzernummer. Auf diese Art kann auf Dateien zugegriffen werden, die dem Programmdateibesitzer gehören, und auf die der aufrufende Nutzer eigentlich keinen Zugriff hat. Steht bei den Gruppenrechten statt des x ein s, geschieht dasselbe mit den Gruppennummern, als effektive Gruppennummer des Prozesses wird die Gruppennummer des Programmdateibesitzers eingetragen. Ein anschauliches Beispiel für den Sinn dieser Einrichtung, ist die Änderung des eigenen Paßwortes. Die Datei

`/etc/passwd`

enthält (falls kein NIS installiert ist) die Daten der einzelnen User und das (verschlüsselte) Passwort. Sie gehört aus Sicherheitsgründen dem root-User, und darf durch andere Nutzer nicht beschrieben werden. Das Programm

`/bin/passwd`

gehört ebenfalls dem root-User. Da jedoch beim User das Ausführungsrecht "s" gesetzt ist, kann jeder, natürlich kontrolliert durch das Programm, sein eigenes Passwort (und nur dieses) in der Datei ändern.

Beispiel:

```
...
goto markel
...
markel:  ...
```

6.6.8 onintr - *Sprunganweisung zu Interuptroutine*

Ein Shellscript kann ein von außen (einem anderen Prozeß) kommendes Signal verarbeiten. Dies wird meist als Abbruch im Fehlerfall eingesetzt. Es wird von der DEL-Taste erzeugt. Im Script kann mit dem Befehl

`onintr` <*label*>

festgelegt werden, wohin das Programm beim Eintreffen des Interruptssignal verzweigen soll. Mit dem Befehl onintr - können Programmteile, die nicht unterbrochen werden dürfen, z.B. Filekopiervorgänge, geschützt werden. Mit onintr (ohne Parameter) werden Unterbrechungen wieder zugelassen. Wegen den unterschiedlichen Tastaturen und den durch andere Prozesse abgefangenen Tastaturcodes (Xli, mwm, twm usw) muß häufig einiger Aufwand betrieben werden, um die Taste zu finden, die den hier beschrieben Interrupt auslöst.

Beispiel:

```
#!/bin/csh -f
onintr errintr
...
...
errintr:
echo 'Externer Abbruch des Programms'
rm -f tempfile.$$
exit 1
```

6.6.9 continue - *Fortsetzung mit nächstem Schleifendurchlauf*

Sowohl `continue` als auch `break` kommen nur innerhalb von Schleifenkonstrukten vor. Mit continue wird der laufende Schleifendurchlauf abgebrochen und der nächste (mit dem nächsten Wert der Laufvariablen) begonnen.

Beispiel

```
#!/bin/csh -f
set n = -11
while ( $n < 10 )
@ n++
if ( $n == 0 ) then
continue
endif
@ c = $a / $n
echo $n $c
end
```

6.6.4 foreach - Schleife *Schleife für einen Satz Parameter*

Schleifenbefehl (entspricht `for..in` der Bourne-Shell). Der Kern der Schleife (der aus mehreren Kommandos bestehen kann) wird für jeden Wert der Laufvariablen einmal durchlaufen. Beginnt mit `foreach` und endet mit `end`. Dazwischen steht der Schleifenkern. Die Schlüsselwörter `foreach` und `end` müssen am Zeilenanfang stehen.

Beispiel:

```
#!/bin/csh -f
foreach laufvar (Samstag Sonnabend Sonntag)
echo ''----------------------- $laufvar:''
grep $laufvar *
end
# sucht im aktuellen Directory alle Dateien nach Wochenenden ab
```

6.6.5 while - Schleife *Schleife die läuft bis Abbruchbed. erfüllt*

Die Schleife beginnt mit dem Schlüsselwort `while` und endet mit dem Schlüsselwort `end`. Vor jedem Durchlauf wird die Bedingung geprüft. Ist sie erfüllt, wird der Schleifenkern ausgeführt, wenn nicht wird die Schleife abgebrochen.

Beispiel:

```
#!/bin/csh -f
#Fakultaet
set n = 1
set prod = 1
while ( $n <= $1 )
@ prod = $prod * $n
@ n++
end
echo "$1 Fakultaet ist $prod"
```

6.6.6 repeat - *Mehrmalige Ausführung eines Kommandos*

Die repeat - Anweisung ermöglicht es eine Kommando n-mal ausführen zu lassen, wobei n vorgegeben wird. Dabei darf es sich nur um ein einzelnes Kommando handeln

Beispiel:

```
#!/bin/csh -f
# 10 Zeilen eingeben
repeat 10 line > brief.txt
```

6.6.7 goto - *Sprunganweisung zu Marke*

Sprunganweisung zu einer Marke innerhalb eines Shellscripts. Das Ziel ist durch ein Label zu kennzeichnen (Name gefolgt von Doppelpunkt).

4.5 Der Zustand eines Prozesses

Ein Prozeß kann sich in einem der drei Grundzustände befinden:

- er kann aktiv sein, d.h. er rechnet (**running**)
- er kann rechenwillig sein, aber die CPU nicht besitzen (**suspended**)
- er kann auf ein Ereignis warten (**waiting**)

Ein Prozeß verliert die CPU, wenn er auf ein Ereignis warten muß (Übergang zu waiting) oder wenn ein Prozeß mit höherer Priorität rechenbereit geworden ist. Das letztere kann zwei Ursachen haben. Entweder ist ein Ereignis eingetreten, auf das der höher priorisierte Prozeß gewartet hat, oder die Priorität des Prozeßes wurde erhöht. In beiden Fällen geht der unterbrochene Prozeß in den Zustand suspended. Das `ps`-Kommando zeigt weitere Zwischenzustände an, z.B. beendeter Prozeß, der noch im Speicher steht.

4.6 Die Prozeßpriorität und der Scheduler

In einem System konkurrieren stets mehrere Prozesse um die Zuteilung der CPU. Im System muß daher eine Steuerung vorhanden sein, nach der eine Prozeßauswahl getroffen wird. Diese Steuerung heißt Scheduling-Algorithmus. UNIX benutzt für das Scheduling (d.h. Umschalten zwischen den Prozessen) einen prioritätsgesteuerten Algorithmus. Die CPU wird immer dem rechenbereiten Prozeß mit der höchsten Priorität zugeteilt. Um eine einseitige Vergabe der CPU an die Prozesse mit hoher Priorität zu verhindern, wird die Priorität jedes Prozesses in bestimmten Zeitintervallen (ca. 1 Sekunde) neu berechnet. In diese Berechnung gehen die Grundpriorität, die im letzten Zeitintervall erhaltene CPU-Zeit und die Größe des Prozesses ein. Das `ps`-Kommando zeigt zwei Prioritätswerte an:

- Die **aktuelle Priorität** (PRI), die der Prozeß momentan besitzt, und die für die CPU-Vergabe durch den Scheduler maßgebend ist.
- Die **nice Priorität** (NI), welche die Grundpriorität des Prozesses darstellt. Sie wird als Steigerungswert für die Berechnung der aktuellen Priorität verwendet.

Bei den Zahlwerten für die Priorität bedeuten kleine Zahlen eine hohe Priorität. Grundsätzlich besitzen Prozesse im Systemmodus eine höhere Grundpriorität. Im Systemmodus befindet sich ein Prozeß, wenn er eine Systemfunktion (system call) aufgerufen hat. Soll ein Prozeß mit einem anderen als dem Standard-nice-Wert gestartet werden, kann dies mit dem Kommando `nice` geschehen. Nachträgliche Änderungen sind mit `renice` möglich.

4.7 Prioritätsklassen

Mit der Version V.4 wurde das Prozeßkonzept erweitert, sodaß nun der Kernel (beinahe) realzeitfähig wurde. Es wurden drei Prioritätsklassen eingeführt:

TIMESHARING Zu dieser Klasse gehören die normalen Benutzerprozesse, die dem in Abschnitt 4.6 auf der vorherigen Seite beschriebenen Scheduler-Algorithmus unterliegen.

REAL-TIME Jeder Prozeß dieser Klasse erhält eine feste Priorität, die während der Laufzeit nicht verändert wird. Nur der root-Nutzer kann sie mit priocntl ändern. Da diese Prozesse nicht unter das Time-Sharing fallen, kann vom Anwender eine feste Ablaufreihenfolge der Prozesse festgelegt werden. Dies hat wesentliche Auswirkungen auf das Gesamtsystem.

SYSTEM In dieser Klasse liegen die Systemaufrufe, die für den Betrieb unbedingt erforderlich sind. Die Prozesse dieser Klasse haben die höchste Priorität.

4.8 Prozeßauslagerung

In einem Unixsystem steht in der Regel ein virtuelles Speichersystem zur Verfügung. Man versteht darunter einen meist festen Teil der Festplatte (**Swapbereich**), der als Erweiterung des Arbeitsspeichers dient. Natürlich muß ein Programm, welches abgearbeitet werden soll, im Arbeitsspeicher stehen. Ein ausgefeilter Algorithmus sorgt daher dafür, daß durch Umlagerung stets die benötigten Programme bzw. die benötigten Teile davon im Arbeitsspeicher stehen. Damit wird nicht nur eine Beschleunigung des gesamten Rechners erreicht, sondern auch die Möglichkeit geschaffen, daß auch Programme die grösser sind als der Arbeitsspeicher abgearbeitet werden können.
Das Aus- und Einlagern von Teilen des Hauptspeichers in den Swapbereich geschieht durch einen eigenen Prozeß mit dem Namen **swapper** und der Prozeßnummer 0. Dieser Prozeß wird beim Systemstart erzeugt und bleibt dann ständig aktiv.
Wieviel Speicher (in pages, siehe man pagesize) ein Prozeß im Swapbereich belegt, ist in der Spalte SZ des Kommandos `ps -l` zu ersehen.

4.9 Vorder- und Hintergrundprozesse

Startet ein Anwender ein Kommando, so erzeugt die Shell einen Kindprozeß. Dieser wird abgearbeitet und liefert am Ende ein Signal an den Elternprozeß zurück.
Unter einem Vordergrundprozeß verstehen wir einen Prozeß, der an ein Terminal (Ein/Ausgabegerät) gekoppelt ist und von diesem aus kontrolliert werden kann. Dies bedeutet, daß die Shell, aus der heraus das Kommando aufgerufen wurde, in den Wait-Status geht, bis der Kindprozeß abgeschlossen ist, d.h. in diesem Fall laufen Eltern- und Kindprozeß **synchron**.
Soll ein Hintergrundprozeß gestartet werden, ist nach dem Kommandonamen das Zeichen "&" einzugeben. In diesem Fall ist der Prozeß an kein Terminal gebunden. Eltern- und Kindprozeß laufen unabhängig voneinander, sie laufen **asynchron**. Liefert ein Hintergrundprozeß Daten an die Standardausgabe kann es zu Kollisionen mit Vordergrundprozessen kommen. Eine solche Ausgabe sollte daher mit Redirect oder Pipe umgelenkt werden (siehe 5.2.1 und 5.2.2). Nach dem Start eines Hintergrundprozesses wird die Job-nummer (in eckigen Klammer) sowie PID ausgegeben, und die aufrufende Shell meldet sich wieder mit dem Prompt.

3.:
`if -x $pgmname > eval $pgmname`

6.6.2 Switch-Anweisung - *bedingte Mehrfachverzweigung*

Gegenüber if..then..else-Konstrukten übersichtlicher. Beginnt mit dem Schlüsselwort `switch` und endet mit dem Schlüsselwort `endsw`. Dazwischen beliebig viele Zweige mit `case` gefolgt von einem Muster. Steht Ende der case-Verzweigung ein `breaksw`, wird von dort nach `endsw` gesprungen. Fehlt `breaksw` werden die folgenden case-Bedingungen ebenfalls getestet und ev. ausgeführt. In den case-Bedingungen können die Wildcards '*' und '?' verwendet werden. Die Bedingung `default` entspricht `case *` und wird daher stets erfüllt. Achten Sie darauf, daß die Kommandos in der der case-Anweisung folgenden Zeile beginnen müssen.

Beispiel:

```
#!/bin/csh -f
set input = `line` # line in backticks
switch ($input)
case [aA] :
programA
breaksw
case [bB] :
programB
breaksw
default
echo 'Bitte nur a oder b eingeben.'
breaksw
endsw
```

6.6.3 line - *interaktive Eingabe von Tastatur*

Die Abarbeitung des Scripts wird unterbrochen und von der Tastatur kann eine Zeile eingegeben werden. Abschluß mit CR:

Beispiel:

```
#!/bin/csh -f
echo 'Bitte Namen eingeben:'
set name = `line` # line in backticks
echo 'Guten Tag, $name!'
echo 'Bitte nochmal eingeben:'
if (`line` == $name) then
echo 'Gut gemacht!'
else
echo 'Das waren zwei verschiedene Namen!'
endif
```

Op.	Status	Beisspiel
-r	ist Lesezugriff erlaubt	`if (-r liste.dat)then`...
-w	ist Schreibzugriff erlaubt	`if (-w brief. text)then` ...
-x	ist es eine ausführbare Datei	`if (-x sorti) then`...
-e	existiert Datei mit diesem Namen	`if (-e /usr/local/bin/tcsh)then`...
-o	ist Nutzer Eigentümer (owner) der Datei	`if (-o /home/pub/buch.txt)then`...
-z	existiert Datei und hat Länge Null	`if (!-z pgm.log)then`... # Negation
-f	ist es normale Datei	`if (-f norm. txt)then`...
-d	ist es ein Directory	`if (-d /texte/tex)the` ...

Tabelle 13: Dateiattribute

werden sie jedoch hier getrennt dargestellt, und in Übersicht auf die hier folgenden Stellen verwiesen. Kontrollstrukturen kommen vor allem in Shell-Scripts zur Anwendung. Neben bedingten Programmverzweigungen, Schleifenkonstrukten umfassen sie auch die Möglichkeit von Kommentaren, Sprungbefehlen, der interaktiven Eingabe und einige Hilfsbefehle zum Testen von Schleifen. Die Wirkungsweise wird hier nur sehr kurz dargestellt, da sie aus anderen Programmiersprachen bekannt sein dürfte. Sie kann auch aus den Man-Pages (Aufruf mit

`man` <*kommandoname*>

entnommen werden.

6.6.1 if ... then ... else-Anweisung - *bedingte Programmausführung*

Bedingte Ausführung von Programmabschnitten. Die Anweisung beginnt stets mit einem `if` und endet mit `endif` Dazwischen können mehrere `else if` und ein `else` - Zweig stehen. `if`, `else` und `endif` sollten immer an einem Zeilenanfang stehen. Die möglichen Operatoren in der Bedingung sind in der Tabelle in Abschnitt 6.5 auf der vorherigen Seite aufgeführt

Mehrere Bedingungen können durch die logischen Operationen (siehe 6.5 auf der vorherigen Seite) miteinander verknüpft werden. Darüberhinaus gibt es spezielle Operatoren, die es gestatten den Status von Dateien auf einfache Weise zu testen (siehe Tabelle 13)

Beispiele:

1.:
```
#!/bin/csh -f
if ( $n != 0 ) then @ c = $a / $n
```

2.:
```
#!/bin/csh -f
if ( $a == ' ' ) then
echo 'Da ist ein leerer String!'
else if ($a != $b) then
echo "$a ist nicht gleich $b!"
else
echo 'Die sind beide gleich!'
endif
```


4.10 Prozeßkommunikation und Prozeßsynchronisation

Jeder Prozeß unter Unix kann weitere Prozesse anlegen, die dann asynchron vom Elternprozeß abgearbeitet werden. In der Regel soll jedoch ein solcher neu angelegter Prozeß eine bestimmte Funktion ausführen auf deren Beendigung (Ergebniss) der Elternprozeß wartet. Dies erfordert eine Interprozeßkommunikation, die im Unix-Kernel implementiert ist. Die einfachste Form dieser Kommunikation der **`exit`**-Aufruf wurde oben schon beschrieben.

4.10.1 Signale

Eine weitere Möglichkeit der Prozeßsynchronisation stellen die **Signale** dar. Ein Signal ist ein asynchrones Ereignis und bewirkt eine Unterbrechnung auf der Prozeßebene. Signale können entweder von außen durch den Benutzer (z.B. durch Betätigen der Unterbrechungstaste) oder durch das Auftreten von Programmfehlern (Adreßfehler, ungültige Instruktion, Division durch Null etc.) erzeugt oder aber durch externe Unterbrechnugen (wie das SIGKILL-Signal oder der Ablauf einer mit `alarm` gesetzten Zeitscheibe) hervorgerufen werden. Auch ein anderer Prozeß kann, mittels des Systemaufrufs `kill` (*pid, signalnr*), ein Signal senden; nur der root-User darf jedoch Signale an fremde Prozesse schicken. Fängt ein Programm ein Signal nicht explizit ab, um es dann zu ignorieren, so führt das Signal zum Programmabbruch.

Der Systemaufruf `tsignal (signalnr, funktion)` erlaubt es einem Programm vorzuschreiben, daß beim Auftreten des Signals `signalnr` die Funktion `funktion` angesprungen werden soll. Ist funktion = SIG_IGN, so wird keine Funktion angesprungen und das Signal ignoriert. Wird ein Signal an den Prozeß mit der PID 0 gesendet, so wird es an alle Prozesse der gleichen Prozeßfamilie gegeben. Das Signal SIGKILL (9) kann nicht abgefangen oder ignoriert werden und führt in jedem Fall zum Programmabbruch. Somit kann durch `kill -9 pid` ein Benutzer seine eigenen Prozesse abbrechen. Der root-User ist auch in der Lage fremde Prozesse zu beenden. Die Signale SIGUSR1 und SIGUSR2, welche keine feste Bedeutung haben, können zur Interprozeßkommunikation benutzt werden. Mit der Einführung von Unix-V.3 wurde der Signalmechanismus erweitert. Man kann nun in einem Programm mit `sighold (signal)` einen kritischen Abschnitt beginnen und mit `sigrelse(signal)` beenden. Tritt bei der Abarbeitung dieses kritischen Abschnitts das Signal `signal` auf wird es bis zur Beendigung des Abschnitts zurückgehalten.

Die möglichen Signale sind dargestellt in:

Tabelle 2: Prozeßsignale

siehe Seite 20

SIGHUP	1	Abbruch einer Dialogstationsleitung (Analog zum Aufhängen des Telefons)
SIGINT	2	UNTERBRECHUNG von der Dialogstation
SIGQUIT	3*	ABBRUCH von der Dialogstation
SIGILL	4*	Ausführung einer ungültigen Instruktion
SIGTRAP	5*	**trace-trap**-Unterbrechnung (Einzelschrittaus-führung)
SIGIOT	6*+	IOT-Instruktion
SIGEMT	7*+	EMT-Instruktion
SIGFPE	8*	Ausnahmesituation bei einer Gleitkommaoperation (floating point exception)
SIGKILL	9	KILL-Signal
SIGBUS	10*+	Fehler auf dem Systembus
SIGSEOV	11*+	Speicherzugriff auf unerlaubtes Segment
SIGSYS	12*	Ungültiges Argument bei einem Systemaufruf
SIGPIPE	13	Es wurde auf eine Pipe oder Verbindung geschrieben, von der keiner liest
SIGALARM	14	Ein Zeitintervall ist abgelaufen
SIGTERM	15	Signal zur Programmbeendigung
SIGUSR1	16	frei für Benutzer
SIGUSR2	17	frei für Benutzer
SIGCLD	18	Signalisiert Beendigung eines Kindprozesses
SIGPWR	19+	Signalisiert einen Spannungsausfall
SIOPOLL	20	Signalisiert das Anstehen eines Ereignisses

Tabelle 2

Die mit + markierten Siganle sind maschinenspezifisch und müssen nicht bei allen Implementierungen vorhanden sein

Die mit * gekennzeichneten Signale erzeugen einen Speicherabzug (engl. **core dump**) des Programms, sofern sie nicht explizit ignoriert oder abgefangen werden. Dabei wird eine Kopie des Prozesses, so wie er im Hauptspeicher steht in die Datei core im aktuellen Verzeichnis geschrieben. Dieser Speicherabzug kann dann z.B. mit **gdb** analysiert werden.

6.5 Operatoren

Innerhalb eines Ausdrucks sind folgende Operatoren erlaubt. Die Liste entspricht den Vorrangregeln bei ihrer Abarbeitung:

	Oerator	Bedeutung	Beispiel
1.	()	Explizite Vorrangangabe (Klammern)	
2.	+ -	Vorzeichen	@ c = - $a
3.	~	Einer-Komplement	@ c = ~ $a
4.	!	Logische Negation	if !($a == 3) then. . .
5	* / %	Multiplikation, Division, Modulo	@ c = $a * $b
6.	<< >>	Shift (bitweise) nach links bzw. rechts	@ c = ($a << 2)

7.	== !=	Zeichenketten vergleichen (Joker in der Zeichenkette nicht erlaubt)	if ($a == $b) then
			
			echo "die sind gleich"
			
			endif

8.	=~ !~	Zeichenketten vergleichen (Joker *, ? und [...] erlaubt)	if ($a =~ *.text) then
			
			echo "da ist eine Textdatei"
			
			endif

9.	< <=	Relationale Operatoren kleiner, kleiner gleich	if ($a $b) then ....(in a und b Zahlen)
10.	> >=	Relationale Operatoren grösser, grösser gleich	if ($a >= 5) then... (a muß Zahl sein)
11.	&	bitweises UND	@ c = ($a & 7)
12.	|	bitweises ODER	@ c = ($a $mask)
13.	∧	bitweises EXKLUSIV ODER	@ c = ($a A $mask)
14.	&&	logisches UND	if (($a != $b) && ($a 5)) . . .
15.	||	logisches ODER	if (($a == $b) \|\| ($a =0))then . . .
16.	command	liefert 1 wenn command erfolgreich, sonst null	if (command == 1) then . . .

Tabelle 12: Operatoren

Es ist darauf zu achten, daß die arithmetischen Operationen (auch , <, <=, > und >=) nur auf Variable mit numerischem Inhalt angewendet werden, da sonst eine Fehlermeldung erfolgt.

6.6 Kontrollstrukturen

Diese Befehlsgruppe gehört zu den eingebauten Befehlen der C-Shell, die im nächsten Abschnitt 6.11 auf Seite 45 behandelt werden. Wegen ihrer Bedeutung

Variable	Typ	Bedeutung
argv	val	Array welches die Kommandozeilen-Argumente enthält (siehe 6.4.2)
cdpath	val	Enthält Directory-Liste. Bei cd, popd und chdir wird dort nach dem Subdir gesucht
cwd	val	enthält den vollständigen Pfad des aktuellen Working-Directory
echo	flag	Wenn gesetzt, wird jede Kommandozeile ge-echot; entspricht Aufruf csh -x
gid	val	enthält Gruppen-ID des Nutzers
history	val	Mit ihr wird die Anzahl der Zeilen der History-Liste vorgegeben (im .cshrc zu setzen)
home	val	enthält Pfad des Heimatverzeichnises, Kurzform: ~
ignoreeof	flag	Wenn gesetzt, wird CTRL+D als EOF (End_of_File)-Zeichen von der Tastatur ignoriert, und führt nicht zum Verlassen der Shell
mail	val	mail (n file) enthält Fileliste, die alle n-Sekunden auf Änderung kontrolliert wird. Falls eine Änderung eingetreten ist, wird Meldung "You have new Mail" ausgegeben
noclobber	flag	Verhindert Löschen bzw Neuanlegen von Dateien bei Redirection (siehe 5.2.1 auf Seite 25)
noglob	flag	Wenn gesetzt, werden Variable in File- und Directorynamen nicht expandiert
nonomatch	flag	Wenn gesetzt, erscheint statt 'ls-F: No match.' das Suchmuster und 'No such file or directory'
notify	flag	Wenn gesetzt, wird Ende eines Hintergundprozesses sofort angezeigt, sonst erst bei Prompt
path	val	Directory-Liste, die durchsucht wird bei Aufruf eines Kommando, identisch PATH
prompt	val	enthält String, der als prompt erscheint
shell	val	enthält Pfad zur aktuellen Shell
status	val	Enthält Exit-Status des letzten Kommandos: 0 = OK, 1 = fehlerhaft
term	val	Enthält Typ der aktuellen Terminalemulation geliefert von `/etc/ttytype` identisch TERM
time	val	time (n f): von Prozessen die länger als n sek. dauern wird Laufzeit ausgegeben
tty	val	Enthält die Bezeichnung des Ports, über den der User am System hängt
uid	val	Enthält die User-ID des aktuellen Nutzers (von /usr/bin/id)
user	val	Enthält den Login-Namen des Users
verbose	flag	Jede Zeile eines Scripts oder aus History-Liste (nach Substitution) wird angezeigt

Tabelle 11: Von der Shell belegte Variable

4.10.2 Pipes

Diese Art der Kommunikation zwischen zwei Prozessen ist nur dann möglich, wenn der Elternprozeß die Pipe durch einen entsprechenden Systemaufruf aufbaut und dann zwei Kindprozesse erzeugt, die von ihm die Pipedateien erben. Eine Pipe ist stets unidirektional. Für den Prozeß ist die Pipe eine Datei, auf die er schreiben oder lesen kann wie in andere Dateien auch (nur positionieren ist nicht möglich). Die Datenmenge pro Operation ist auf die Pipe-Buffer-Grösse beschränkt, die von der Implementierung abhängt (in der Regel 4KByte).

4.10.3 Named Pipes

Die named pipes stellen eine Erweiterung des Pipemechanismus dar. Eine named pipe besitzt einen mit `/etc/mknod` angelegten Geräteeintrag vom Typ FIFO (First In First Out) und hat damit einen entsprechenden externen Namen, unter dem sie angesprochen werden kann (beim `ls`-Kommando ist sie vom Typ p). Mittels dieses Namens können nun mehrere Prozesse miteinander kommunizieren, ohne einen gemeinsamen Elternprozeß zu haben. Dies wird in der Regel dazu benutzt ein Serverkonzept zu realisieren. Der Dienstprozeß (Client) liest dabei seine Aufträge aus der `named pipe` während die Auftragsprozesse (Server) ihre Aufträge in die named pipe schreiben. Dabei muß der Datenumfang der einzelnen Aufträge festgelegt sein, um beim Lesen diese trennen zu können. Die Einträge von named pipes liegen entgegen der üblichen Konvention zumeist nicht im Verzeichnis /dev sondern in dem Katalog des Serverprozesses. Die Serverprozesse `lpsched` und `cron` sind typische Beispiele für Dienstprozesse, die so arbeiten.

4.10.4 Weitere Mechanismen zur Interprozeßkommunikation

Mit dem System V wurden drei weitere Mechanismen zur Kommunikation zwischen verschiedenen Prozessen eingeführt. Mit System V.3 kam noch ein vierter dazu:

- Nachrichten (message queues)
- Semaphore (Systemaufruf semop)
- Speicher auf den mehrere Prozesse zugreifen können (shared memory)
- Streams (Teil des NSS = Network Support Service) (V.3)

Auf diese Mechanismen kann hier jedoch nicht näher eingegangen werden und es wird auf die Literatur verwiesen.

4.11 Daemonen

In einem laufenden UNIX-System gibt es eine Reihe von Prozessen, die systemweite Aufgaben ausführen. Solche Prozesse werden meist beim Hochfahren des Systems gestartet und laufen während der gesamten Betriebszeit. Solche Prozesse werden Daemonen genannt. Sie sind keinem realen Nutzer oder einem Terminal zugeordnet. Beispiele dafür sind der Druckerdaemon (der das Druckersystem

verwaltet), der cron-Prozeß (der die Aufträge verwaltet, die zu einem bestimmten Zeitpunkt ausgeführt werden sollen) oder der xdm (X-Display-Manager). Natürlich können auch vom Nutzer solche Daemon-Prozesse gestartet werden, die u.U. bis zum Ausloggen des Nutzers laufen.

6.4.4 Arrays

In der C-Shell sind außer den o.g. skalaren Variablen auch Felder (Arrays) möglich. Die Anzahl der Elemente wird bei der Initialisierung angegeben. Die Werte für die Elemente werden durch runde Klammern zusammen- gefasst:

`set farbe = ( rot gruen blau grau gelb schwarz weiss)`

Damit ist `farbe` ein Array, welches bis zu einer neuen Initialisierung 8 Elemente enthält. Auf die einzelnen Elemente kann mit ihrer Ordnungsnummer in eckigen Klammern zugriffen werden.

`echo farbe[1]` ergibt rot
`echo farbe[2]` ergibt gruen
`echo farbe[3]` ergibt blau
`echo farbe[4]` ergibt grau
etc.

6.4.5 Vordefinierte Shell-Variable

Die C-Shell kennt eine Reihe von Variablen, die sie selbst setzt und/oder interpretiert. Bei einigen wertet die Shell den Inhalt der Variablen aus, bei anderen wird lediglich ihre Existenz geprüft (Flag-Funktion). Entsprechend gibt es zwei Arten der Initialisierung:

`set var = wert` (val)
`set var` (flag)

Einige Variable werden nie explizit belegt wie:

`argv, cwd` und `status`

Automatisch durch die Shell belegt werden:

`argv, cwd, gid, home, path, prompt, shell, status, term, tty, uid` und `user`.

Die folgenden Variablen werden zwar, wenn sie belegt sind, von der C-Shell ausgewertet, sind aber durch den Nutzer zu belegen. Am besten in der Datei `~/.cshrc` (`~/.tcshrc`)

`cdpath, echo, history, ignoreeof, mall, noclobber, noglob,`
`nonomatch, notify, path, prompt, term, time, verbose`

Bedauerlicherweise werden von den unterschiedlichen Implementationen der C-ShelI auf verschiedenen System teilweise unterschiedliche Shell-Variable unterstützt. Hier sind deshalb nur diejenigen aufgeführt, welche von (fast) allen Systemen unterstützt werden. Die Genaue Liste der verfügbaren Variablen erhält man durch
`man csh`.

csh	tcsh	Bedeutung
set		definiert eine lokale Variable
setenv		definiert eine globale Variable
@		Veranlasst die Berechnung des nachfolgenden Ausdrucks und Zuordnung des Ergebnisses zu der unmittelbar folgenden Variablen (ersetzt das Kommando expr der Bourne-Shell)
$ {var}		Inhaltsoperator: Variable wird durch ihren Wert ersetzt
$ {var[i]}		Selektiert das i-te Wort in var. i kann eine Zahl, ein Bereich m - n, - n, m- oder * sein (* = alle Worte). Es kann auch eine Variable sein: $a
${#var }		liefert die Anzahl Wörter in var
${#argv }	$#	Anzahl der Argumente in der Kommandozeile
$0		Name des Programms
${argv[n]}	$n	Einzelne Argumente aus der Kommandozeile. Positionsparameter n ...9
$n		wie $argv[n] , Kurzschreibweise
${argv[*]}	$*	Alle Argumente aus der Kommandozeile, außer $0, identisch mit ${ argv }
$*		wie $argv[*] , Kurzschreibweise
$argv[$#argv]		letztes Argument in der Kommandozeile
${?var}		liefert 1 wenn var existiert, sonst 0
$$		Prozeßnummer PID der laufenden Shell (z.B.: als Name für temporäre Files)
$<		Eingabe einer Kommandozeile von der Konsole
$!		Prozeßnummer PID des letzten Hintergrundkommandos

Tabelle 10: C-Shell: Variable aus der Kommandozeile

des Zuweisungsoperators numerisch berechnet wird, und das Ergebnis in die Variable geschrieben wird:

@ varname zuweisungsoperator ausdruck

(Beachte das Leerzeeichen zwischen @-Operator und varname) Zuweisungsoperatoren sind die in der Sprache C üblichen: =, +=, -= , *= und /=.

Ebenso sind die Operatoren ++ und – zugelassen.

Beispiele : `@ zaehler = $zaehler + 3`

ist gleichbedeutend mit

`@ zaehler += 3`

oder
`set inkrement = “3“`
`@ zaehler = $zaehler + $inkrement`

`@ zaehler++` entspricht `@ zaehler = $zaehler + 1`
`@ zaehler--` entspricht `@ zaehler = $zaehler - 1`

5 Die Shell

Im UNIX-System nimmt die Shell eine zentrale Bedeutung ein. Sie stellt die Schnittstelle zwischen Maschine und Mensch her. Dabei erfüllt sie zwei fundamentale Funktionen:

- Als Kommandointerpreter analysiert sie die Benutzereingaben und startet Kommandos bzw. gibt die Fehlermeldungen aus.
- Als Programmierspracheninterpreter erlaubt sie die Abarbeitung von Kommandoprozeduren (Programmen), auch Shell-Scripts genannt.

Aus den verschiedenen Entwicklungslinien von UNIX resultieren unterschiedliche Shells, mit unterschiedlichen Leistungsmerkmalen und Syntax. Auf den meisten Rechnern sind mehrere Shells vorhanden. Die Standardshell kann für jeden Nutzer einzeln in der Datei /etc/passwd eingetragen werden. Mit dem Befehl `chsh` kann jeder Nutzer seine Standardshell ändern (nicht in allen Systemen). Die vorhandenen Shells sind in der Datei `/etc/shells` eingetragen.

Aufruf	**Name**	**Eigenschaften**
sh	Bourne-Shell	Standard-Shell in den UNIX-V-Systemen
csh	C-Shell	Aus der Berkley- (BSD)-Linie. Ihre Syntax ist sehr stark an die Programmiersprache C angelegt.
tcsh	tc-Shell	Erweiterung der C-Shell
ksh	Korn-Shell	Aus der AT&T-Linie. Seit UNIX V Rel.2.O. Aufwärtskompatibel zur Bourne Shell. Da sie die Vorteile der sh und der csh vereinigt hat, schien sie gute Chancen zu haben, die zukünftige Standardshell zu werden.
rsh	Restricted Shell	(Systemabhängig!)Stellt nur einen eingeschränkten Befehlsvorrat zur Verfügung, der oft fremden Nutzern aus Sicherheitsgründen angeboten wird.
rsh (oder remsh)	Remote Shell	Shell die auf einem fremden Rechner (remote) aufgemacht wird, um dort Kommandos aufzurufen
ssh	Secure Shell	Ebenfalls eine Remote-Shell, die aber zusätzlich die zu übertragenden Daten verschlüsselt und komprimiert.

Tabelle 3: Shells

Hier soll die C-ShelI näher beschrieben werden. Vorerst jedoch Eigenschaften, die in allen Shells vorhanden sind.

5.1 Die Kommandozeile

Jede Kommandozeile wird nach ihrer Übergabe (Drücken der RETURN-Taste) von der Shell zerlegt und interpretiert. Der Aufbau einer Kommandozeile muß daher den folgenden Regeln genügen:

1. Aufbau: *Kommandoname Optionen Argumente*
   Optionen und Argumente werden von der Shell an das Kommando weitergegeben.
   Optionen modifizieren das Kommando (sind optional). Argumente sind die Grössen auf die das Kommando anzuwenden ist (z.B. Dateinamen). Die Anzahl der Optionen und der Argumente hängt vom Kommando ab (manchmal auch die Reihenfolge).
   Beispiel: `ls -l /etc`
2. Optionen beginnen (fast) immer mit einem - (minus) -Zeichen.
   Beispiel: `ps -u root`
3. Mehrere Optionen können hinter einem Minuszeichen zusammengefasst werden.
   Beispiel: `ps -eal`
4. Optionen können auch eigene Argumente (Optionsargumente) verlangen.
   Beispiel `ps -u root` `find /users -name hugo`
5. Die einzelnen Elemente eines Kommandos sind durch Leer- oder Tab-Zeichen zu trennen.
6. Eine Kommandozeile wird durch einen Zeilenvorschub CR abgeschlossen, und damit der Shell übergeben.
7. Es können mehrere Kommandos in einer Zeile übergeben werden. Sie sind durch Semikolon zu trennen.
   BeispieL `pwd; ls -l`
8. Ein Kommando kann sich über mehrere Zeilen erstrecken. Die oberen Zeilen müssen dann mit Backlash ('\') enden.
   Beispiel:
   `cp quelle1` \
   ? `quelle2` \
   ? `zieldirectory CR`
   (Das ?-Zeichen als Prompt in den Folgezeilen ist nicht einheitlich)

5.2 Das Ein- und Ausgabekonzept

Die Standardaufgabe eines Programms ist es Daten einzulesen, sie zu verarbeiten und wieder auszugeben. Zum Datentransfer wurden unter UNIX s.g. Kanäle eingerichtet. Unabhängig von der Art (Gerät, Datei etc) einer Datensenke bzw. Datenquelle sind alle Senken und alle Quellen einander formal gleichwertig. Beim Start eines Programms werden stets die folgenden drei Standarddateien (-kanäle) geöffnet:

Name	Abkürzung	Gerät	Deskriptor
Standardeingabe	stdin	Tastatur	0
Standardausgabe	stdout	Bildschirm	1
Standardfehlerausgabe	stderr	Bildschirm	2

Tabelle 4: Standard-IO-Channels

Sollen einer Variable mehrere Wörter zugeordnet werden, sind sie mit ( ) einzuklammern.

Beispiel: set a = (fuenf sechs sieben)

6.4.1 Globale und lokale Variable

Eine globale Variable wird (im Gegensatz zu einer lokalen Variablen) an Kindprozesse weitergegeben (siehe 5.4 auf Seite 27).
Globale Variable werden meist mit Großbuchstaben bezeichnet, lokale mit Kleinbuchstaben.

Hinweis : Die wichtigen und häufig benutzten Environment-Variablen USER, TERM, PATH und CDPATH werden durch die C-Shell automatisch bei jeder Veränderung mit den Werten der lokalen Variablen user, term path und cdpath belegt.

Zuweisung:	lokale Variable:	`set varname = wert`
	globale Variable:	`setenv varname wert`
Anzeigen:	aller lokalen Variablen:	`set oder set varname`
	aller globalen Variablen:	`env oder env varname`

Beipiele:

`set seinname = Hugo`
`set datum = ‘date‘` (*(Bedeutung von ‘ ‘ ,backticks, siehe 5.3 auf Seite 27*)
`echo ''$seinname kommt am $datum''` *(hier müssen ” ” stehen!*)

`Set prompt =` “! > “ *(Setzt prompt auf Historynummer (event) plus '>')*

6.4.2 Variable aus der Kommandozeile

Beim Aufruf eines Kommandos wird von der Shell automatisch der Inhalt der Kommandozeile in die Variable argv geschrieben. Mit **$argv**[0... 9] kann dann auf die einzelnen Argumente zurückgegriffen werden, wobei **$argv**[0] den Kommandonamen enthält. Statt **$argv**[n] kann auch $n verwendet werden. $0 enthält dann den Kommandonamen und $1, $2, ... enthalten die Argumente entsprechend ihrer Position in der Kommandozeile. Enhält die Kommandozeile mehr als 10 Argumente, kann die Reihe mit shift um ein Argument nach links verschoben werden. Das Argument mit der niedrigsten Nummer geht verloren, dafür rückt eines auf die oberste Stelle nach.
Die geschweiften Klammern in der folgenden Tabelle sind optional, es sei denn, sie trennen einen Variablennamen von den nachfolgenden Zeichen, die sonst Teil des Namens würden.

In der Spalte tcsh sind nur die Abweichungen von der csh-Shell angegeben.

6.4.3 Numerische Variable

Die C-Shell enthält eine Arithmetik mit ganzen Zahlen. Durch Voranstellen des Operators @ vor eine Variable wird veranlaßt, daß der Ausdruck rechts

anwenden.
Beispiel:
`set m = ‚Alles ist schoen!'`
`echo $m:s/les/kohol/`
ergibt: -> `Alkohol ist schoen!`

6.3 Der Alias-Mechanismus

Unter einem Alias versteht man eine (meist) abgekürzte Bezeichnung für ein komplexeres Kommando. Ein Alias wird mit dem folgenden Kommando erzeugt:

`alias` *neu_kommando alt_kommando(zeile)*

Damit wird in der s.g. Alias-Tabelle ein Befehl *neu_kommando* eingetragen. Wird ein Kommando aufgerufen, sucht die C-Shell zuerst in der Alias-Tabelle. Wird dort das Kommando nicht gefunden, wird in allen Verzeichnissen gesucht die in der Variablen `PATH` (identisch mit `path`) stehen. Vor der Ausführung der Kommandozeile wird der Alias-Namen durch *alt_kommando(zeile)* ersetzt. Dieser Austausch findet nur statt, wenn das Alias als Kommandoname verwendet wird, nicht wenn der Alias-Name als Argument in einer Kommandozeile steht. Durch ein Alias kann ein bestehendes Kommando auch 'überschrieben' werden (siehe Beispiel unten). Soll in einem solchen Fall der ursprüngliche Befehl aufgerufen werden, kann durch ein Voranstellen eines Backlashs vor das Kommando die Shell gezwungen werden die Alias-Tabelle zu übergehen.

An ein Alias können auch Parameter übergeben werden. Sie werden im Alias-Kommando mit einem Ausrufezeichen und laufender Nummer verwendet. (!1, !2 etc., !* bedeutet alle Parameter)

Beispiele:

`alias ll ls -la` *erzeugt Ausgabe eines ausführlichen Directory*
`alias cd cd !*;set prompt = \$cwd >\` *aktuelles Directory wird Promptzeichen*
`alias rm rm -i` *beim Löschen wird stehts nachgefragt*
(`\rm` *ruft wieder den ursprünglichen rm-Befehl auf)*
`alias m more l*` *kürzt more ab*

6.4 Variable

In der C-Shell ist die Verwendung von Variablen möglich. Sie haben den Typ string. Enthalten sie nur Ziffernzeichen, können sie als Operanden in arithmetischen Ausdrücken verwendet werden. Die Zuweisung von Werten geschieht mit den Kommandos in der folgenden Tabelle.

Befehl	Bedeutung
set	definiert eine lokale Variable
setenv	definiert eine globale Variable
@	Veranlasst die Berechnung des nachfolgenden Ausdrucks und Zuordnung des Ergebnisses zu der unmittelbar folgenden Variablen (ersetzt das Kommando expr der Bourne-Shell)Beispiel siehe 6.4.3

Tabelle 9: Variablenbelegung

Neben diesen Standarddateien öffnen die meisten Programme noch weitere Dateien. Programmtechnisch werden alle Dateien, mit denen ein Programm Daten austauscht, durchnummeriert, s.g. Dateideskriptoren. Die Standarddateien erhalten automatisch die Nummern 0 ...2 die weiteren Dateien 3...

5.2.1 Redirection

Jeder Datentransfer zu einer Standarddatei (-gerät) kann auf eine andere Datei umgeleitet werden. Dies geschieht mittels des Redirection-Mechanismus. Die Ausgabe wird mittels “>“ umgeleitet:

Beispiel: `ls > direc.dat` *(schreibt das Inhaltsverzeichnis in die Datei direc.dat)*

Sollte bei dem obigen Beispiel eine Datei direc.dat schon existieren, wird sie überschrieben. Durch die Schreibweise “>>“ wird erreicht, daß die Daten an die bestehende Datei angehängt werden, falls diese bereits existiert. Falls sie nicht existiert, wird sie erzeugt.

Beispiel: `ls >> direc.dat` *(hängt Inhaltsverzeichnis an Datei direc.dat an)*

Verlangt ein Programm die Eingabe von der Standardeingabe stdin, kann diese ebenfalls umgeleitet werden. Dies geschieht mit “<“. Als Beispiel dient `sort`, welches eine stdin-Eingabe fordert, wenn kein Argument angegeben wird:

Beispiel: `sort < liste.dat` *(statt von stdin wird von liste.dat gelesen)*

Natürlich können beide Kanäle auch gleichzeitig umgelenkt werden:

Beispiel: `sort < list.dat > resultat.dat` *(das Ergebnis wird in eine Datei geschrieben)*

Achtung: Eine Umlenkung von Datenströmen per Redirection ist nur möglich, wenn diese zu den Standarddateien gerichtet sind.

Eine Umleitung von stderr ist ebenfalls möglich. Sie unterscheidet sich allerdings bei den verschiedenen Shells. Unter der C-Shell (und tcsh) kann stderr nur gemeinsam mit stdout umgeleitet werden.

Nur csh und tcsh:

Beispiel: `kommando >& error.dat` *(leitet stdout* **und** *stderr in error.datei um)*

Eine Trennung beider Datenströme ist mit folgendem Trick möglich:

Beispiel: `(kommando > ausgabe.dat) >& fehler.dat` *(leitet die Ausgabe und die Fehler getrennt um)*

Durch die Klammer wird das Kommando in einer Subshell ausgeführt, wodurch eine getrennte Umleitung möglich wird.
(Unter der Bourne-Shell ist eine Umleitung mit `kommando 2 > fehler.dat` möglich.)

Um ein versehentliches Überschreiben einer Datei durch Redirection zu erschweren, kann mit

`set noclobber`

die Variable noclobber gesetzt werden. Existiert die Ausgabedatei, erscheint eine Fehlermeldung und das Löschen der Datei unterbleibt. Dies kann durch

Anhängen eines Ausrufezeichens “!“ an das Redirectionzeichen wieder aufgehoben werden.

```
set noclobber
ls >! direc.dat
```

Auf die “>>“-Umleitung hat die Variable noclobber die Auswirkung, daß die Daten nur an eine bestehende Datei angehängt werden, und bei Nichtvorhandensein der angegebenen Datei eine Fehlermeldung erfolgt, anstatt diese neu zu erzeugen. Die Aufhebung dieser Wirkung durch anhängen von “!“ ist ebenso möglich.
Falls eine Ausgabe lediglich unterdrückt werden soll, kann dies mit dem Gerät `/dev/null` erreicht werden, auf das die Ausgabe umgelenkt wird (Papierkorb).

5.2.2 Pipeline

Häufig tritt das Problem auf, daß die Ausgabe eines Kommandos die Eingabe eines anderen sein sollte. Dies könnte dadurch erreicht werden, daß die Ausgabe in eine Datei umgelenkt wird, die dann als Eingabe für das zweite Kommando dient. Eleganter ist jedoch das Pipe-Konzept.

Beispiel: `ls -l | more` *(Die Ausgabe von ls ist Eingabe vom more)*

Die Ausgabedaten von `ls` wurden in einen Zwischenspeicher geschrieben und von dort an `more` übergeben. Die linke Seite muß stets eine Quelle sein, die auf stdout schreibt und die Rechte eine Datensenke die von stdin liest. Beide Kommandos werden gleichzeitig gestartet, und arbeiten die Daten wie von einem Band ab. Ihre Zusammenarbeit wird von UNIX synchronisiert. Natürlich sind auch mehr als 2 Kommandos verknüpfbar:

Beispiel: `ls -l | sort | grep .ps`
Gibt alle Files die .ps enthalten alphabetisch sortiert aus

Ebenso ist eine Kombination mit der Redirection möglich:

Beispiel: `sort < liste.dat | grep .txt > result.dat`

5.2.3 Hintergrundprozesse

Ein Hintergrundprozeß wird gestartet, indem man die Kommandozeile mit “&“ abschließt. Ein so erzeugter Prozeß gibt die ihn erzeugende Shell sofort wieder frei, d.h. die Shell wartet nicht auf den Abschluß des Kindprozesses (siehe 4.9 auf Seite 18).
Von besonderer Bedeutung ist die Umleitung der Ein- und Ausgabe bei Hintergrundprozessen , deren Ausgabe von der Konsole auf eine Datei umgeleitet werden muß. Soll ein Programm nach dem Ausloggen weiterlaufen, ist es mit

```
nohup (Kommando > ausgabe.dat) >& fehler.dat &
```

zu starten. Wird keine Ausgabeumleitung vorgenommen, wird automatisch eine Datei nohup.out angelegt, in die stdout und stderr umgeleitet werden.

5.2.4 Here Document

Mit dem Zeichen “<<“ können einfache Schleifenkonstruktionen erstellt werden.

Spezifikation

Die meisten Kommandozeilen bestehen aus mehreren Argumenten. Mit der Spezifikation können einzelne oder mehrere dieser Argumente aus der Kommandozeile herausgepickt und einer neuen Kommandozeile übergeben werden. Dies geschieht durch Anfügen eines Doppelpunktes an die Zeile:

!*Kommandozeile* : spez

wobei spez einen der folgenden Werte annehmen kann:

~	erstes Argument
n	n-tes Argument
$	letztes Argument
	alle Argumente
m-n	Argumente m bis n
-m	0-tes bis m-tes Argument
m -*	m-tes bis letztes Argument

Tabelle 7: Spezifikationen in der Kommandozeile

Beispiel: !Kommandozeile:`2-4` liefert das 2. bis 4. Argument der Kommandozeile.

Substition

Durch Anhängen eines zweiten Doppelpunktes ist die zusätzliche Angabe von Modifikationen möglich. Dadurch ist es möglich beliebige Teile der Kommandozeile herauszuschneiden oder zu verändern. Die Syntax lautet

!*Kommandozeile*:`spez`:`mod`

es können auch mehrere angegeben werden:

!*Kommandozeile*:`spez`:`mod1`:`mod12`:`mod3`...

Für `:mod..` kann stehen

:e	liefert Suffix eines Dateinamens (letzter Teil nach einem Punkt)
:g	steht vor e, h, r, 5, t: diese Änderung global durchführen
:h	streicht die letzte Komponente eines Pfadnamens
:p	zeigt die neue Zeile an ohne sie auszuführen
:g	markiert die ersetzten Wörter, für spätere Substitutionen
:r	liefert den Stamm des Dateinamens, läßt Suffix weg
:s/alt/neu/	ersetzt 'alt' durch 'neu' (auf das '/' am Ende achten!)
:t	liefert den letzten Teil des Pfadnamens
:x	wie q, zerlegt jedoch in Worte, Trennzeichen: TAB, CR, SP
:&	wiederholt die letzte Substitution

Tabelle 8: Substitutionen in der Kommandozeile

Hinweis: Mit den Möglichkeiten der **tc-Shell** ist die praktische Bedeutung dieser Art der Substitution in Kommandozeilen in den Hintergrund getreten. Daselbe Werkzeug läßt sich jedoch auch auf die Bearbeitung von Umgebungvariablen

die Login-Prozedur der Login-Shell übergibt, ist natürlich von der konkreten Konfiguration des Systems abhängig.

Als Beispiel:

HOME = Home-Directory des Users
PATH = /bin/usr/bin
SHELL = Name der Start-Shell aus /etc/passwd
MAIL = /usr/mail/loginname
TZ = Zeitzonenspezifikation

Diese können u.a. in der ~/.login-Datei und ~/.cshrc ergänzt werden, insbesondere durch Erweiterung von PATH und weiteren globalen Variablen die von den Anwenderprogrammen ausgewertet werden.

6.2 Der History-Mechanismus

Der History-Mechanismus speichert die eingegebenen Kommandozeilen und bietet die Möglichkeit, diese zurückzurufen um sie erneut eventuell geändert oder korrigiert ausführen zu lassen. Um den Machanismus zu aktivieren, muß die Variable history (.cshrc) gesetzt werden:

`set history = 30`

Damit werden 30 Kommandozeilen in der History-Liste gespeichert. Während einer UNIX-Sitzung werden alle Kommandozeilen automatisch durchnummeriert. Mit dem Befehl

`set prompt = ' ! > '`

kann die laufende Nummer der Eingabezeile als Prompt eingestellt werden.
Das Zurückholen einer Kommandozeile kann auf verschiedene Art erfolgen, stets ist als erstes ein Ausrufezeichen (!) einzugeben:

1.	!n	holt die n-te Kommandozeile zurück	
2.	!string	holt die letzte Kommandozeile mit string zurück	z.B.: !b letzte die mit „b" beginnt
3.	!?string?	holte die letzte Kommandozeile die string enthält	z.B.: !?grep? letzte mit „grep"
4.	!!	wiederholt das letzte Kommando in der Liste	

Tabelle 6: Arbeit mit der History-Liste

Anmerkung: Die tcsh - Shell erlaubt es zusätzlich mit den Kursortasten durch die Historyliste zu laufen, und so wesentlich anschaulicher frühere Komandozeilen zurückzuholen, deren letzter Aufruf möglicherweise bereits vom Bildschirm verschwunden ist.

6.2.1 Das Editieren einer Komandozeile

Eine mit den o.g. Befehlen zurückgeholte Kommandozeile kann nun mittels Kursor und Löschtaste editiert werden, darüberhinaus stehen jedoch auch komfortablere Editierbefehle zur Verfügung.

Beispiel:

```
kommando << ctrlstring
?  eingabel
?  eingabe2
.
.
?  ctrlstring
```

Mit diesem Konstrukt werden alle Eingabezeilen in ein temporäres File geschrieben (Variable und Kommandos werden aufgelöst) bis die Eingabezeile aus dem Wort ctrlstring besteht. Dieses File wird dann dem `Kommando` übergeben.

Beispiel: `sort << ENDE > sortiert.dat`

5.3 Metazeichen

Einige Zeichen haben für die Shell eine Sonderbedeutung:

symbol	Bedeutung
|	Pipesymbol
<	stdin - Umleitung
>	stdout - Umleitung
;	Trennzeichen zwischen Befehlen
&	Aufruf als Hintergrundprozeß
!	Zugriff auf History-Liste
>&	stderr - Umleitung (csh und tcsh)
.	aktuelles Directory
..	über dem aktuellen liegendes Directory
~	Home-Directory
*	“joker“ für bel. Zeichenkette
?	“joker“ für ein bel. einzelnes Zeichen
[ ]	“joker“ - für eine Liste von Zeichen
\	hebt Metabedeutung des folgenden Zeichens auf (ausser !)
$	Variablen-Inhalt
’text’	Metabedeutungen in text aufgehoben
”text”	Nur $ ’ und / haben in *text* Metabedeutung
‘kommando‘	Resultat von Kommando wird eingesetzt
-	Standardeingabe (als Argument) (programmabhängig)

Tabelle 5: Metazeichen

Soll ein Metazeichen als normales druckbares Zeichen verwendet werden, muß seine Metabedeutung aufgehoben werden. Dies geschieht durch Voranstellen des Backlashs “\“.

5.4 Das Environment

Wird ein Prozeß gestartet, wird ihm vom Elternprozeß die Umgebung mitgegeben (siehe Kapitel 4.3). Ein wichtiger Teil dieser Umgebung sind die globalen Variablen. Globale Variable werden in den C-artigen Shells mit

`setenv name Inhalt`

erzeugt. Beachten sie, daß die lokalen Variablen, die mit

`set name = wert`

erzeugt werden, nicht an Kindprozesse weitergegeben werden. Mit

`unsetenv name`

können globale Variable wieder gelöscht werden.

In den bourneartigen Shells werden globale Variable mit

`name = wert ; export name`

erzeugt.

Es hat sich eingebürgert, globale Variable mit GROSSBUCHSTABEN zu bezeichnen.
Einige globale Variable sind von der Shell vorbelegt:

PATH	enthält die Suchpfade
USER	enthält den Nutzernamen
HOME	enthält den Pfad zum Home-Verzeichnis

die folgenden Variablen müssen vom Nutzer belegt werden und werden von der Shell bzw. bestimmten Prozessen entsprechend ausgewertet:

LPDEST	kann vom Nutzer zum Festlegen des Defaultdruckers genutzt werden
MANPATH	enthält die Pfade zu den Dateien des Manuals
TERM	enthält die Bezeichnung der aktuellen Terminalemulation
LINES	Anzahl der Zeilen eines X-Windows-Fenster
COLOUMS	Anzahl der Zeichen pro Zeile im X-Windows-Fenster die beiden letzten werden durch die Shell gesetzt wenn sie eine Window-Umgebung entdeckt, sie werden z.B. von vi ausgewertet (s.a. resize)
PAGER	enthält den Namen des Ausgabeprogramms beim Man-Kommando
DISPLAY	enthält den Namen und die Displaynummer unter X-Windows
LD_LIBRARY_PATH	Pfad auf dem der Linker Bibliotheksdateien sucht

5.5 Die Kommandoausführung

Es gibt verschiedene Methoden ein Kommando zu starten:

Start mit Dateiname Die aktuelle Shell startet einen entsprechenden Kindprozeß mit eigener PID. Der Execute-Status (x-Flag) muß gesetzt sein. Falls der Pfad nicht angegeben wird, muß er in der Variablen PATH enthalten sein.

Shellausführung Aufruf über `sh` *scriptdatei*. Die Shell liest die Scriptdatei und führt ein Kommando nach dem anderen aus. Die aktuelle Shell ruft dabei eine neue Shell für die Abarbeitung der Kommandos auf, die ebenfalls eigene Prozesse mit eigener PID darstellen.

source *kommando* Es wird kein neuer Prozeß erzeugt. Damit sind auch alle lokalen Variablen im *kommando* gültig. Nach Abschluß wird in das aufrufende Programm zurückgekehrt.

exec *kommando* Nach Beendigung wird nicht in das aufrufende Programm zurückgekehrt, sondern der Prozeß wird beendet. Wird aus einer loginshell mit exec aufgerufen, findet nach Schluß des kommandos ein logout statt.

eval *kommando(s)* Bei `eval` vor einer Kommandozeile, werden in einem ersten Durchlauf alle Variablen durch ihre Inhalte ersetzt, die geänderte Zeile wird dann abgearbeitet. Dies gibt die Möglichkeit Metazeichen in Variable zu schreiben. z.B: `eval ls $grz liste` liefert `ls > liste` (falls grZ = '>') Ohne `eval` würde '>' als Parameter an `ls` übergeben. `eval` wird ebenfalls innerhalb der aktuellen Shell ohne eigene PID ausgeführt.

nohup *kommando* Startet *kommando* so, daß es auch nach Ausloggen des Users weiter läuft.

Kommandosubstitution Soll das Ergebnis eines Kommandos weiter verarbeitet werden, wird das Kommando in Apostrophs eingefasst, z.B.:
`set akt_dir = ‘pwd‘`.

6 Die C-Shell

Die C-Shell bringt gegenüber der Bourne-Shell (sh) einige zusätzliche Eigenschaften (Alias-, Job- und History- Mechanismus). Sie ist auf (fast) allen Systemen verfügbar. Die tcsh-shell stellt eine Erweiterung dar, die alle Eigenschaften der C-Shell enthält.

6.1 Login

Nach einem erfolgreichen login (Name und Passwort) verzweigt die login-Prozedur ins Home-Directory und ruft die Standardshell des Users auf (beide Angaben stehen in der Datei `/etc/passwd`). Diese Startshell (sie erkennt sich selbst als solche aufgrund ihrer PPID die auf einen login-Prozeß zeigen muß) wird auch **login-shell** genannt. Die csh liest nun nacheinander die Dateien

```
/etc/.cshrc
~/.cshrc
/etc/.login
~/.login
```

wobei ~ für das Home-Directory steht. Die in diesen Dateien stehenden Kommandos werden ausgeführt. Die Dateien `/etc/.login` und ~/.login werden nur einmalig nach dem Einloggen gelesen, während die Datei `~/.cshrc` beim Start jeder neuen Shell (also auch beim Start eines Shell-Scripts) gelesen und ausgeführt wird. Danach meldet sich die Shell mit dem Prompt (d.h. mit dem Zeichen % oder dem Inhalt der Variablen $prompt (siehe Abschnitt 6.4.5 auf Seite 35). Diese Dateien dienen dazu, systemweite oder benutzerspezifische Einstellungen des Environments vorzunehmen. Welche globalen Variablen