Unix I – Einführung in das Unix Betriebssystem

Stephan Thesing

V. Vorlesung: Shells (cntd.)

- **Bourne-Shell: Misc, Limits, Scripts**
- **`tcsh`**

Aliase

Wenn man ein Kommando oft braucht, kann man eine Abkürzung dafür definieren: einen *Alias*:

```
alias ll="ls -l"
alias cw="cd /home/stud/thesing/project/www"
alias cd='echo Altes Directory: $PWD; cd'
```

Alle Aliase kann man mit `alias` angezeigt bekommen.

Aliase löscht man mit `unalias` *Name* oder `unalias -a`.

Ein Alias wird nur an der Stelle eines Kommandos expandiert.

Limits

Man kann Grenzen für die benutzbaren Ressourcen angeben: *Limits*

Es gibt *soft* und *hard* Limits:

- Hard Limits können nicht überschritten werden. Jeder Prozess, der diese Grenzen überschreitet, wird beendet. Hard Limits können vom Benutzer nur *gesenkt* werden. Man kann sich also gerne weniger CPU Zeit erlauben, diese jedoch nicht vergrössern.
- Soft Limits können vom Benutzer beliebig geändert werden (im Rahmen der hard Limits!). Ein Prozess, der gegen sie verstösst bekommt nur ein Signal geschickt, dass er abfangen kann.

Zur Beeinflussung der Limits gibt es das `ulimit` Programm:

```
ulimit -H -a
time(seconds)         unlimited
file(blocks)          unlimited
data(kbytes)          1048576
stack(kbytes)         32768
coredump(blocks)      unlimited
memory(kbytes)        510468
locked memory(kbytes) 510468
process(processes)    532
nofiles(descriptors)  1772

ulimit -S -t 100

ulimit -S -t
100
```

Es gibt folgende Limits (mit `ulimit` Option):

- CPU Zeit in Sekunden (`-t`)
- File Grösse in Blocks (`-f`)
- Datensegmentgrösse in KByte (`-d`)
- Stackgrösse in KByte (`-s`)
- Coredumpgrösse in Blocks (`-c`)
- Physikalischer Speicher in KByte (`-m`)
- Max. Anzahl Prozesse des Benutzers (`-p`)
- Max. Anzahl offener Files (`-n`)

So setzt z.B. `ulimit -H -p 30` das hard Limit der Prozesse.

Scripte

Man kann auch Kommandos in ein File schreiben, statt sie am Terminal einzugeben.

Etwa:

```
>cat xx
echo Hallo!
FOO=nix
echo $FOO

>sh xx
Hallo!
nix
```

Das geht sogar noch automatischer!

Unter Unix kann man Textfiles schreiben, die vom System als Programm ausgeführt werden!

Dazu muß das File mit einer *Interpreterangabe* beginnen:

```
>cat yy
#! /bin/sh
echo Hallo!
FOO=nix
echo $FOO

>chmod a+x yy
>./xx
Hallo!
nix
```


Dies funktioniert auch mit anderen Interpretern, nicht nur der Shell!

Allgemein:

Wenn ein File f mit `#!`p beginnt, wird p als Interpreter aufgefasst und als $p\ f$ ausgeführt.

Man kann auch Argumente in der Interpreterzeile mitgeben:

```
>cat zz
#! /usr/bin/awk -f
BEGIN{FS=":";}
{printf("Hallo %s\n",$5);}

./zz /etc/passwd
Hallo root
Hallo bin
```

Obiger Aufruf wird ausgeführt als `/usr/bin/awk -f ./zz /etc/passwd`

Kommandostrukturen

Die Bourne-Shell kennt auch kompliziertere Kommandostrukturen für Schleifen, etc.

So kann man etwa Tests durchführen:

```
#! /bin/sh

dir=`pwd`

if [ $dir != $HOME  ]; then
  echo nicht im homedir
else
 echo Home sweet home!
fi
```


Die Bourne-Shell kennt folgende Konstrukte:

- If-Then-Else Abfragen
- While-Schleifen
- For-Schleifen
- Switch-Statement

Zu deren Benutzung: Siehe *sh(1)*!

tcsh

Die Bourne-Shell eignet sich nicht gut zum interaktiven Arbeiten.

Als Ersatz wurde die `tcsh` geschrieben:

- Editieren der Eingabezeile
  - Blättern in vorigen Kommandozeilen
  - Bewegen akt. Zeile mittels Cursortasten, etc.
  - Einfügen von Text
- History-Funktion: Speichern vorheriger Kommandozeilen
- Completion: Vervollständigen von Dateinamen, etc.
- Generell beliebig belegbare Tastenkombinationen

Die tcsh hat aber auch gravierende Nachteile:

- Andere Kommandosprache, als die Bourne-Shell
- Dateiumleitung rudimentär
- Alias Expansion verkrüppelt
- Generell nicht zum Schreiben von Scripten geeignet
- Nicht überall verfügbar

Wir beschränken uns heute auf die Unetrschiede zur Bourne-Shell :-)

Interaktive Features

Die tcsh hat eine Reihe eingebauter Funktionen, mit deren Hilfe man etwa ein Wort löschen kann, etc.

Mit dem int. Kommando `bindkey` kann man nun beliebige Tastenkombinationen an solche Funktionen binden!

Default (u.a.):

- Mit Cursor Links/Rechts: Bewegen in der akt. Eingabezeile
- Cursor Hoch/Runter: Blättern in vorigen Eingabezeilen
- CTRL-A: Anfang der Zeile
- CTRL-E: Ende der Zeile

Alle Bindungen kann man sich mit `bindkey` ansehen.

Eine kurze Beschreibung der Funktionen gibt `bindkey -l`

Die tcsh kann automatisch angefangene Kommandonamen, Filenamen, Direcorynamen vervollständigen.

Dazu gibt man einen Präfix ein und drückt dann die TAB oder CTRL-D Taste.

Gibt es im akt. Directory etwa ein File `foobar.txt`, dann vervollständigt TAB nach der Eingabe

`>ls foo`*[TAB]*

zu

`>ls foobar.txt`

Existieren mehrere mögliche Vervollständigungen, werden alle aufgelistet.

Die tcsh kann automatisch Kommandonamen korrigieren.

Dazu setzt man die lokale Variable `correct`:

```
>set correct=cmd
```

Danach werden alle fehlerhaften Kommandonamen auf ähnlich aussehende Kommandos überprüft:

```
>pk -al

CORRECT>ps -al (y|n|e|a)? y
```


Die tcsh sammelt alle Eingabezeilen in einer Liste (*History*).

Diese wird etwa beim Blättern mit den Cursortasten verwendet.

In der lokalen Variablen `history` steht die Anzahl der max. gespeicherten Einträge:
`set history=100`

Man kann sich die History mit dem `history` Kommando anzeigen lassen:

```
>history
     1  15:29   ls
     2  15:29   less .cshrc
     3  15:31   bindkey -l
     5  15:32   bindkey
     6  15:40   echo $correct
     7  15:40   set correct=cmd
     8  15:40   ps -al
     9  15:48   history
```

Vorherige History-Einträge erhält man über die Nummer:

```
>!6
echo $correct
cmd
```

Die letzte Zeile ist als `!!` erreichbar:

```
>ls -l /etc
...
>!!
ls -l /etc
...
```

Die n-te vorherige Zeile als `!-`n.

Variablen

Die tcsh kennt, wie die Bourne-Shell, lokale und globale Variablen.

Allerdings ist die Syntax beim Setzen anders:

- Lokale Variablen werden durch `set` gesetzt:

  ```
  > set correct=cmd
  > set ignoreof=1
  ```

- Globale Variablen werden durch `setenv` gesetzt:

  ```
  >setenv PATH /usr/local/bin:/usr/bin:/bin:.
  ```

Variablen werden durch die Kommandos `unset` und `unsetenv` gelöscht.

Alle gesetzten Variablen kann man sich durch einfache Eingabe von `set` bzw. `setenv` ausgeben.

Die Expansion von Variablen geschieht auch durch Vorstellen des `$` Zeichens:

```
>echo $version
tcsh 6.08.04 (Astron) 1999-04-19 (sparc-sun-NetBSD)
>echo $PATH
/home/stud/thesing/bin:/usr/local/bin:/usr/sbin:
```


Im Unterschied zur Bourne-Shell werden ungesetzte Variablen nicht zu ’nichts’ expandiert.

Die tcsh bricht in diesem Fall mit einem Fehler ab!

Vorsicht: Taucht so etwas in einem Startup-File auf (`.tcshrc`), wird der Rest des Files nicht ausgeführt.

Beispiel:

```
> echo $NICHTGESETZT
NICHTGESETZT: Undefined variable.
```

Dateiumleitungen

Die tcsh kennt nur die folgenden Umleitungen:

- stdout, z.B. `ls -l / >/tmp/xx`
- stdin, z.B. `cat </tmp/xx`
- stderr und stdout *zusammen*: `ls >&/tmp/xx`
- stdout anhängen: `ls >>/tmp/xx`
- stderr und stdout anhängen: `ls >>&/tmp/xx`

Die tcsh kann keine Filedescriptoren einzeln umleiten!

Limits

Die Befehle zur Steuerung der Limits heissen bei der tcsh anders:

```
>limit
cputime         4:00:00
filesize        unlimited
datasize        65536 kbytes
stacksize       1536 kbytes
coredumpsize    0 kbytes
memoryuse       119344 kbytes
descriptors     64
memorylocked    39781 kbytes
maxproc         80

>unlimit coredumpsize
```

Per Default nimmt die tcsh die Soft Limits. Mit `limit -h` bekommnt man etwa die Hard Limits.