Unix I – Einführung in das Unix Betriebssystem

Stephan Thesing

IV. Vorlesung: Shells

- **Generelles**
- **Bourne Shell: `/bin/sh`**

Shells

Eine Shell ist der Hauptinteraktionspunkt mit dem System.

Sie liest Eingaben vom Terminal (oder einem File) und interpretiert diese.

Meistens führt sie dann ein Programm aus.

Sie bietet eine eigene Kommandosprache, mit der man auch Scripte schreiben kann.

Wird bei Einlogen als User-Prozess gestartet (Console oder `xterm`-Fenster)

Aufgaben der Shell:

- Auswerten der Kommandozeile (Pattern, Quotes)
- Verwaltung von *Umgebungsvariablen*
- Umlenkung von stdin, stdout, stderr, ...
- Zusammenschaltung von Kommandos (Pipelines)
- Starten von Kommandos im Vorder–/Hintergrund
- Verwaltung von Konfigurationsparametern

Es gibt mehrere Shells, z.B.

- Bourne-Shell `/bin/sh`
- `tcsh` `/usr/local/bin/tcsh`
- Korn-Shell `/bin/ksh`
- `zsh`

Diese unterscheiden sich in Syntax der Kommandosprache, interaktiven Features, etc.

Heute besprechen wir die Bourne-Shell.

Die Bourne-Shell ist die Standardshell auf allen Unix-Systemen.

Ist immer unter `/bin/sh` zu finden.

Bietet eine einheitliche Kommandosprache für Scripte.

Bietet allerdings wenig Unterstützung für interaktives Arbeiten (dafür `tcsh` besser).

Scripte sollten immer in Bourne-Shell Syntax geschrieben werden (Portabilität!).

Hier eingeführten Konzepte auch (abgewandelt) auf andere Shells übertragbar.

Kommandozeilenverarbeitung

Wenn eine Zeile eingegeben wird, interpretiert die Shell diese Zeile, teilt sie auf:

- In das Kommando
- die Argumente zu dem Kommando

Z.B.

```
ls -l /home/stud/thesing
```

Argumente werden durch Leerzeichen oder TABs getrennt.

Die Shell macht aber weitaus kompliziertere Bearbeitungen!

- Homedirectory-Expansion `ls ~/tmp`, `ls ~schellen`
- Variablensubstitution `ls $HOME`
- Kommandosubstitution `` ls `cat /tmp/xx` ``
- Feldaufteilung
- Pfadmatching `ls /tmp/x*`
- Entfernen von Quotes

Die Shell bietet Umgebungsvariablen.

Diese kann man setzen, etwa `FOOBAR=nix`

Den Inhalt einer Variablen kann man wieder einsetzen, indem man ein `$` vor den Namen der Variable stellt:

```
echo $FOOBAR
```

Es gibt einige vordefinierte Variablen:

- `HOME` Das Homedirectory des Benutzers
- `USER` Der eigene Username
- `PATH` Liste von Directories, in denen nach Kommandos gesucht wird
- `TERM` Der Typ des Terminals

Man kann die Ausgabe eines Programmes wieder auf der Kommandozeile einsetzen:

```
ls -l `cat /tmp/xx` ~thesing
```

Wenn `/tmp/xx` die Zeilen `/bin`, `/etc` enthält, würde obiges folgendermassen expandiert werden:

```
ls -l /bin /etc /home/stud/thesing
```


Die Eingabezeile wird nach obigen Expansionen in Felder aufgeteilt. Felder werden dabei durch Leerzeichen oder TABs getrennt.

(Genauer: in der Umgebungsvariablen `IFS` stehen die Zeichen, die Felder trennen.)

So würde also

```
ls -l /bin /etc /home/stud/thesing
```

in fünf Felder zerlegt:

1. `ls`
2. `-l`
3. `/bin`
4. `/etc`
5. `/home/stud/thesing`

Danach werden *Pfadpattern* expandiert:

Pattern beschreiben eine Menge von Files/Directories, z.B.

```
rm *.txt
```

Das Pattern `*.txt` wird expandiert:

Alle Files/Directories, die in `.txt` enden werden dafür eingesetzt.

Gibt es z.B. die File `a.txt`, `nixda.txt` und `foobar/muell.txt`, dann expandiert obiges zu

```
rm a.txt nixda.txt
```

Pattern schliessen '`/`' nicht implizit ein. Dagegen:

```
rm */*.txt *.txt
```

expandiert zu

```
rm foobar/muell.txt a.txt nixda.txt
```


Die Shell kennt folgende Pattern

- `*` passt auf null–, ein–, zwei–, etc -maliges Auftreten eines Zeichens: `a*.txt` passt auf `a.txt`, `abdfdf.txt` aber nicht auf `b.txt`
- `?` passt auf genau ein Zeichen: `?.txt` passt auf `a.txt`, jedoch nicht auf `ab.txt`
- `[]` Klassen von Zeichen. So passt etwa `a[0-9].txt` auf `4.txt`, jedoch nicht auf `a.txt` oder `a09.txt`. Man kann auch negieren: `[!0-9].txt`

Man kann natürlich mehrere Pattern haben:

```
ls /etc/*/?.txt.[0-9].tmp
```

Wichtig: Pattern ignorieren Files, die mit einem `.` beginnen: `ls *` würde z.B. ein File `.profile` nicht anzeigen!

Dagegen würde `ls .??*` dies tun (Punkt explizit gematcht).

Problem: Wie kann ich Argumente angeben, die Leerzeichen enthalten?

Lösung: Man quotet sie, schliesst sie in Anführungszeichen ein:

```
mail -s 'nichts besonderes' thesing
```

Es gibt zwei Arten von Anführungszeichen:

- Die einfachen Anführungszeichen ('). Der Text zwischen den Anführungszeichen wird genauso übernommen, wie er da steht (inkl. `$`, doppelten Anführungszeichen, etc.)
- Die doppelten Anführungszeichen: Der Text wird übernommen, aber Variablenexpansion und Kommandoexpansion finden weiterhin statt.

Beispiele:

```
ls 'so ein unsinn'
echo "$HOME sweet $HOME"
echo "Dies ist `cat /tmp/msg`" schoen
ls '`pwd` $HOME'
ls "*.txt"
```

expandieren zu

```
ls so ein unsinn
echo /home/stud/thesing sweet /home/stud/thesing
echo Dies ist Meldung schoen
ls / /home/stud/thesing
ls *.txt
```

Umgebungsvariablen

Werden in der Shell gesetzt:

```
FOOBAR=nix
AB="so ein Unsinn"
klein=GROSS
```

Es gibt zwei Sorten von Variablen:

- Globale Variablen: werden an aufgerufene Programme vererbt
- Lokale Variablen: sind nur in der Shell verfügbar

Variablen sind per default lokal.

Werden mittels `export` global gemacht:

```
export FOOBAR
```

Siehe *sh(1)* für weitere Details der Expansion.

Redirection

Die Shell kann die Ein–/Ausgaben umleiten:

- stdin (normalerweise Tastatur)
- stdout (normalerweise Bildschirm)
- stderr (dito)

Will man etwa die Ausgabe eines Kommandos in einem File haben:

```
ls -l >/tmp/liste
```

Ähnlich kann man die Eingabe aus einem File nehmen, statt der Tastatur:

```
tr -d '\r' </tmp/msdosfile >/tmp/unixfile
```

Man kann auch an Ausgaben anhängen:

```
ls >>/tmp/liste
```

Die Fehlerausgaben kann man getrennt umleiten:

```
find / -name '*.txt' 2>/dev/null
```

Oder aber mit der Ausgabe zusammen in ein File schicken:

```
find / -name '*.txt' >/tmp/ausgabe 2>&1
```

Interessante Variante (für Scripte):

```
cat <<EOM
Dies ist ein Text
der von stdin gelesen wird.
Homedir: $HOME
EOM
```


Umleitungen sind nützlich, um etwa Ausgaben eines Programmes von einem anderen bearbeiten zu lassen.

```
ls /etc >/tmp/aus
grep passwd </tmp/aus
```

Da die meisten Unixprogramme Eingaben von stdin lesen und Ergebnisse auf stdout ausgeben, kann man das Anlegen einer Zwischendatei `/tmp/aus` umgehen.

Dazu wird stdout des ersten Programmes an stdin des zweiten gekoppelt:

```
ls /etc | grep passwd
```

Das geht mit einer beliebigen Anzahl von Programmen:

$$p_1 \mid p_2 \mid p_3 \mid \cdots \mid p_N$$

Eine solche Konstruktion nennt man *Pipeline*.

Die Shell startet jedes der angegebenen Programme und lenkt stdout von p_i nach stdin von p_{i+1} um.

Eine solche Pipeline wird von der Shell in eine eigene Prozessgruppe getan, so dass ein CTRL-C alle diese Prozesse zusammen abbricht.

Jedes Programm kann sein stderr in eine andere Datei umleiten:

```
ls 2>/tmp/ls-err /etc | \
 grep -i unix 2>/tmp/grep-err >/tmp/grep-res
```


Kommandos starten

Nachdem die Shell die Eingabezeile in Felder aufgeteilt hat, wertet sie diese aus.

Dazu wird das erste Feld der Eingabe angesehen und mehrere Fälle unterschieden:

- Variablenzuweisung
- Shell-internes Kommando
- Externes Kommando (Programm)

Einige Kommandos muss die Shell intern implementieren:

- `cd`
- `umask`
- ...

Wenn das erste Feld kein internes Kommando darstellt, wird es als Name eines externen Programmes aufgefasst, welches auszuführen ist.

Dazu muss das Programmfile gesucht werden.

Enthält das erste Feld einen Schrägstrich `/`, dann wird nach dem File gesucht, das durch diesen Pfad beschrieben wird:

```
/bin/ls -l
bin/ls -l
```

Ansonsten wird in einer Liste von Directories nach einem File des gegebenen Namens gesucht.

Diese Liste ist veränderbar und steht in der Umgebungsvariablen `PATH`:

```
echo $PATH
/home/stud/thesing/bin:/usr/local/bin:/usr/sbin:/bin:/usr/bin:
/sbin:/usr/bin/X11:/shared/bin:/home/stud/thesing:.
```


Für jedes Directory D in `PATH` wird versucht, das File *D/Name* als Programm auszuführen.

Beim ersten Treffer bricht die Suche ab: Directories weiter vorne in `PATH` haben Priorität gegenüber Directories weiter hinten!

Um herauszufinden, welche Programm nun ausgeführt werden würde kann man `which` benutzen:

```
>which ls
/bin/ls
```

Man kann Programme auch im Hintergrund laufen lassen.

Dann wartet die Shell nicht auf deren Beendigung, sondern liest sofort wieder die nächste Eingabezeile.

Die Ausführung im Hintergrund wird durch das Ampersand `&` notiert:

```
>ls -R / >/tmp/aus &
>ps -l
  UID   PID  PPID CPU PRI NI VSZ  RSS  STAT TT  COMMAND
19124 16019 15558   1  10  0 424  136  S    p1  sh
19124 16184 16019   6   2  0 412  192  R    p1  ls -R /
19124 16185 16019   3  31  0 324  120  R+   p1  ps -l
>jobs
[1] 16184                     ls -R / >/dev/null
> fg %1
> [CTRL-C]
```


Wenn man CTRL-Z drückt, wird das Programm im Vordergrund angehalten.

Man kann es mit `bg` dann in den Hintergrund schicken:

```
>ls -R / >/dev/null 2>&1
^Z%1 Suspended
> jobs
[1] 16194 Suspended           ls -R / >/dev/null 2>&1
> bg
> fg %1
^C>
```