Unix I – Einführung in das Unix Betriebssystem

Stephan Thesing

III. Vorlesung, 8.11.1999

- **Terminals**
- **Prozesse**
- **Utilities**

I/O Umgebung eines Programmes

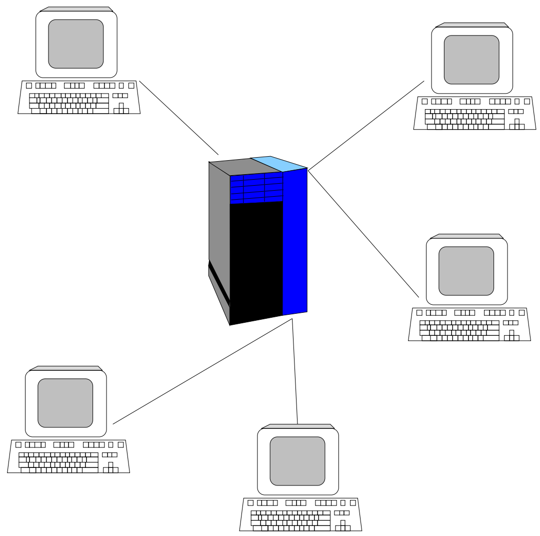
Jedes Programm hat nach dem Start drei I/O Kanäle geöffnet:

- `stdin` Standardeingabe
- `stdout` Standardausgabe
- `stderr` Fehlerausgabe

Diese sind normalerweise mit den Benutzerein-/ausgaben verbunden.

Für Benutzerein-/ausgaben gibt es in Unix das Konzept des *Terminals*.

Es stammt noch aus den Zeiten der Mainframes:



Terminals sind einfache Verbindungen für (nicht graphische) Ein-/Ausgabe.

Vielfältige Verbindungsmethoden:

- Bildschirm/Tastatur des Rechners selbst (Console: `/dev/console`)
- `xterm` Fenster
- Serielle Schnittstelle via Modem
- Netzwerkverbindung
- ...

Man kann auch heute noch einen Unix Rechner über die serielle Schnittstelle booten.

Terminals haben bestimmte Attribute

- Breite, Höhe
- Farbe
- Zeichensätze
- Geschwindigkeit, Verbindungsparameter

Terminals interpretieren bestimmte Zeichensequenzen als *Steuerzeichen*.

Diese dienen dazu, Ausgaben anzuhalten, etc.

Sind wichtig, da man z.B. mit der 'Interrupt' Sequenz Programme abschiessen kann.

Einige Sequenzen:

- Interrupt (CTRL-C)
- EOF (CTRL-D)
- Stop (CTRL-S)
- Start (CTRL-Q)
- Suspend (CTRL-Z)
- ... (Siehe `stty -a`)

Unix unterstützt so ziemlich alles an Terminals, was es auf der Welt gibt

- Fernschreiber
- XTerm
- Sun, PC, Amiga, … Tastaturen
- ASCII Terminals

Zählung unter `/usr/share/misc/termcap`: 1021 Terminals

Unterstützung so vieler Terminals geschieht durch zwei Teile.

1. System-intern: Konzept der Terminal-Devices.

   - Diese kann man durch Systemaufrufe konfigurieren: Steuerzeichen, Breite, Höhe, Anschlussparameter
   - Pseudoterminals für Netzwerkverbindungen
   - Kommando hierzu: *stty(1)* (Damit kann man auch CTRL-C ummappen)

2. Durch eine Programmbibliothek (`curses`)
    - Diese stellt eine einheitliche Schnittstelle bereit
    - Durch die Datenbank `/usr/share/misc/termcap`, in der für jedes Terminal bestimmte Control-Sequenzen definiert sind (etwa, 'Bildschirm löschen', 'Cursor nach unten')
    - Jedes Benutzerprogramm kann festlegen, welcher Terminaltyp zu benutzen ist. Geschieht über die Umgebungsvariable `TERM`.

Prozesse

Jedes Programm, das läuft ist ein *Prozess*. Es kann mehrere Prozesse gleichzeitig geben.

Prozesse können auch von anderen Prozessen kreiert werden.

Im System gibt es zwei Arten von Prozessen:

- Systemprozesse (Daemonen): Steuern wichtige Administrationsaufgaben
- Benutzerprozesse: Programme, z.B. `less`

Prozesse haben

- Eine Prozess-ID (PID)
- Einen Besitzer
- Eine Gruppe
- Genau einen Vater
- Priorität und *nice*-Level
- Ein akt. Directory
- Offene Files
- `umask`
- Session
- Exit-Status

Einen neuen Prozess kann man nur durch Kopieren eines laufenden Prozesses erzeugen (*fork(2)*).

Dabei werden Attribute des Vaterprozesses auf den Kindprozess vererbt:

- Virtual Memory
- Files
- umask
- cwd
- ...

Dadurch entsteht eine Baumstruktur aller aktiven Prozesse.

Prozesse laufen 'gleichzeitig': Sie werden immer ein Stück abgearbeitet, bis der nächste Prozess an die Reihe kommt.

Unix benutzt (fast) preemptives Scheduling hierfür.

Welcher Prozess laufen darf, wird anhand seiner Priorität und seines *nice*-Levels bestimmt.

Die Priorität wird durch eine Zahl angegeben (kleine Zahl ⇒ hohe Priorität!). Sie wird hochgezählt, solange der Prozess läuft.

Existiert ein Prozess mit höherer Priorität (kleinerer Prioritätszahl), wird dieser Prozess zur Ausführung gebracht.

Der *nice*-Level dient dazu, einen Offset einzuführen, also Prozesse zu benachteiligen/bevorzugen.

Ihre Shell ist z.B. ein Userprozess.

Er nimmt ihre Eingaben entgegen und führt dann Kommandos aus, indem er einen neuen Prozess kreiert und diesen das Programm ausführen lässt:

- Der neue Prozess sucht in einer Liste von Directories nach dem Programmfile.
- Wenn die Zugriffsrechte es erlauben, wird der neue Prozess durch das Programm *überlagert*.
- Dieses führt dazu, dass die Ausführung am Beginn des Programmfiles neu beginnt.

An das Kind werden (auch nach Überlagerung durch ein Programm) einige Attribute vererbt:

- Offene Files (`stdin`, `stdout`, `stderr`!)
- cwd
- umask
- UserID
- GroupID
- Umgebungsvariablen

Prozesse können auch im Vordergrund oder Hintergrund laufen!

Um Prozesse zu beeinflussen, kann man ihnen *Signale* schicken.

Diese veranlassen, je nach Typ des Signals, verschiedene Aktionen:

- `KILL` Sofortige Beendigung des Prozesses
- `INT` Abfangbare Beendigungsaufforderung
- `HUP` Hangup
- `STOP` Anhalten
- `CONT` Fortfahren
- ... (siehe `/usr/include/sys/signal.h`)

In Verbindung mit Terminals:

Bestimmte Controlsequenzen werden vom Terminaltreiber interpretiert, der daraufhin bestimmte Signale an die Prozesse dieses Terminals sendet:

- CTRL-C: `INT`
- CTRL-\: `QUIT`
- CTRL-Z: `STOP`

(Im Unterschied dazu: CTRL-Q, CTRL-S vom Terminaltreiber nur intern interpretiert!)

Utilities

Files:

Zum Suchen in Dateien ist `grep` sehr nützlich.

Es sucht nach Mustern in Dateien und gibt alle Zeilen einer Datei aus, die auf ein bestimmtes Muster passen.

Z.B.: `grep Unix /home/stud/thesing/Unix-Kurs/Testdatei`

Auch komplizierte Muster sind möglich:

`grep '^Unix' ...`

`grep 'Datei?.txt' /tmp/foobar`

`grep 'File*.txt' /tmp/baz`

`grep` kennt eine ganze Menge von Optionen, siehe *grep(1)*.

Das Programm `find` such nach Dateien/Directories im Filesystem.

Um z.B. alle Dateien zu finden, die mit `.txt` enden:

```
find / -name '*.txt' -print
```

Oder, um alle Directories zu finden, die mit `RCS` beginnen:

```
find . -name 'RCS*' -type d -print
```

Man kann auch für jeden Treffer ein Programm ausführen lassen:

```
find . -name foobar -exec mv \{\}   \{\}.old ';'
```


Prozesse:

Das Programm `ps` dient zur Ausgabe aller laufenden Prozesse.

Z.B.:

```
>ps
  PID TT  STAT      TIME COMMAND
  314 p1  Ss+    0:00.11 -csh (tcsh)
  313 p2  Ss     0:00.20 -csh (tcsh)
 4512 p2  S      0:24.49 emacs vorlesung_3.tex
 4526 p2  S      0:09.01 gv vorlesung_3.ps
 5147 p2  I      0:01.57 gs -dNOPLATFONTS -sDEVICE=x11alpha -dNO
 5152 p2  R+     0:00.00 ps
  312 p3  Is+    0:00.07 -csh (tcsh)
```

```
gargoyle:/tmp>ps -l
  UID   PID  PPID CPU PRI NI   VSZ  RSS WCHAN  STAT TT      TIME COMMAND
10016   314   298   0  18  0   572  952 pause  Ss   p1   0:00.11 -csh (tcsh)
10016  5190   314   0  28  0   284  180 -      R+   p1   0:00.00 ps -l
10016   313   295   6   3  0   580  960 ttyin  Is+  p2   0:00.20 -csh (tcsh)
10016  4512   313   0   2  0  4476 5368 select S    p2   0:24.53 emacs vorle
10016  4526   313   0   2  0   608 1884 select S    p2   0:10.42 gv vorlesun
10016  5155  4526   6   2  0  2616 2864 select I    p2   0:01.06 gs -dNOPLA
10016   312   297   4   3  0   556  888 ttyin  Is+  p3   0:00.07 -csh (tcsh)
```

Universität des Saarlandes


```
gargoyle:/tmp>ps -u
USER       PID %CPU %MEM   VSZ  RSS TT  STAT STARTED      TIME COMMAND
thesing   5191  0.0  0.0   284  180 p1  R+   12:22PM   0:00.00 ps -u
thesing    312  0.0  0.2   556  888 p3  Is+  Fri07PM   0:00.07 -csh (tcsh)
thesing    313  0.0  0.2   580  960 p2  Is+  Fri07PM   0:00.20 -csh (tcsh)
thesing    314  0.0  0.2   572  952 p1  Ss   Fri07PM   0:00.11 -csh (tcsh)
thesing   4512  0.0  1.0  4476 5368 p2  S     8:30AM   0:24.53 emacs vorle
thesing   4526  0.0  0.4   608 1884 p2  S     8:32AM   0:10.43 gv vorlesun
thesing   5155  0.0  0.5  2616 2864 p2  I    11:47AM   0:01.06 gs -dNOPLAT
```

Universität des Saarlandes

Das Programm `kill` dient dazu, Signale an andere Prozesse zu schicken:

`kill` *[-Signal] Prozess-ID*

Etwa:

```
kill -STOP 1234
kill -KILL 8754
kill 5353
kill -HUP 131
```

`kill -l` listet alle bekannten Signale:-)

Das Programm `top` eignet sich als Monitor, welche Prozesse wirklich laufen.

```
load averages: 0.48,  0.61,  0.77                              12:32:06
46 processes:  1 running, 45 sleeping
CPU states: 0.0% user, 0.0% nice,  0.0% system, 0.0% interrupt, 100% idle
Memory: 53M Act,2908K Inact,392K Wired,407M Free,4K Swap,1024M Swap free

  PID USERNAME PRI NICE   SIZE   RES STATE   TIME   WCPU    CPU COMMAND
  247 root       2  -10  8716K   26M sleep 793:57  0.78%  0.78% XF86_SVGA
 5223 thesing    2    0  2628K 2900K sleep   0:01  0.60%  0.54% gs
  300 thesing    2    0  1972K 2396K sleep   6:32  0.00%  0.00% xfmail
 4512 thesing    2    0  4476K 5368K sleep   0:25  0.00%  0.00% emacs
  299 thesing    2    0    18M   15M sleep   0:21  0.00%  0.00% netscape
 4526 thesing    2    0   608K 1884K sleep   0:11  0.00%  0.00% gv
  213 root      18  -12   340K  608K sleep   0:08  0.00%  0.00% xntpd
  133 root       2    0    92K  444K sleep   0:07  0.00%  0.00% ypbind
  209 root      18    0    32K  308K sleep   0:04  0.00%  0.00% update
  295 thesing    2    0   572K 1764K sleep   0:02  0.00%  0.00% xterm
```

Die Programme `nice` und `renice` dienen dazu, den *Nice*-Level eines Prozesses zu beeinflussen.

Normale Benutzer dürfen den Nice-Level nur anheben (Prozess hat weniger Priorität).

Nur der Super-User (`root`) darf den Nice-Level herabsetzen!

Das Programm `nice` startet ein anderes Programm mit einem bestimmten Nice-Level:

```
/usr/bin/nice -n 10 ls -R / >&/dev/null
```

(Vorsicht! In der tcsh gibt es ein internes `nice`, das anders funktioniert!)

Das Programm `renice` dient dazu, bereits laufende Prozesse mit einem anderen Nice-Level zu versehen:

```
renice 10 5297
```