

# Das Prozesskonzept

In diesem Teil wird das Prozesskonzept von Unix beschrieben und elementare Kommandos zum Auflisten und Unterbrechen von Prozessen.

## Inhalt

1. Das Prozesssystem .....	2
2. Prozessgenerierung und Synchronisation .....	5
3. Prozesszustände.....	8
4. Systemprozesse .....	11
4.1. Der Startvorgang der Benutzerprozesse .....	12
5. Befehle für das Prozess-System.....	15
5.1. ps .....	15
5.2. top .....	16
5.3. kill .....	21

# 1. Das Prozesssystem

Der Begriff **Prozess** (engl. process) bezeichnet ein Programm, das gerade geladen ist und (meist) läuft.

Unix unterscheidet streng zwischen den Terminologien Prozess und Programm.

- Ein Programm ist das (auf der Platte gespeicherte) Objekt, das ausführbaren Code (meist Maschinensprache-Befehle) enthält.
- Zum Prozess wird ein Programm dadurch, dass es aufgerufen - und damit in den Arbeitsspeicher geladen wird. Notwendig wird diese Unterscheidung dadurch, dass ein Programm mehrmals gleichzeitig aufgerufen werden kann.

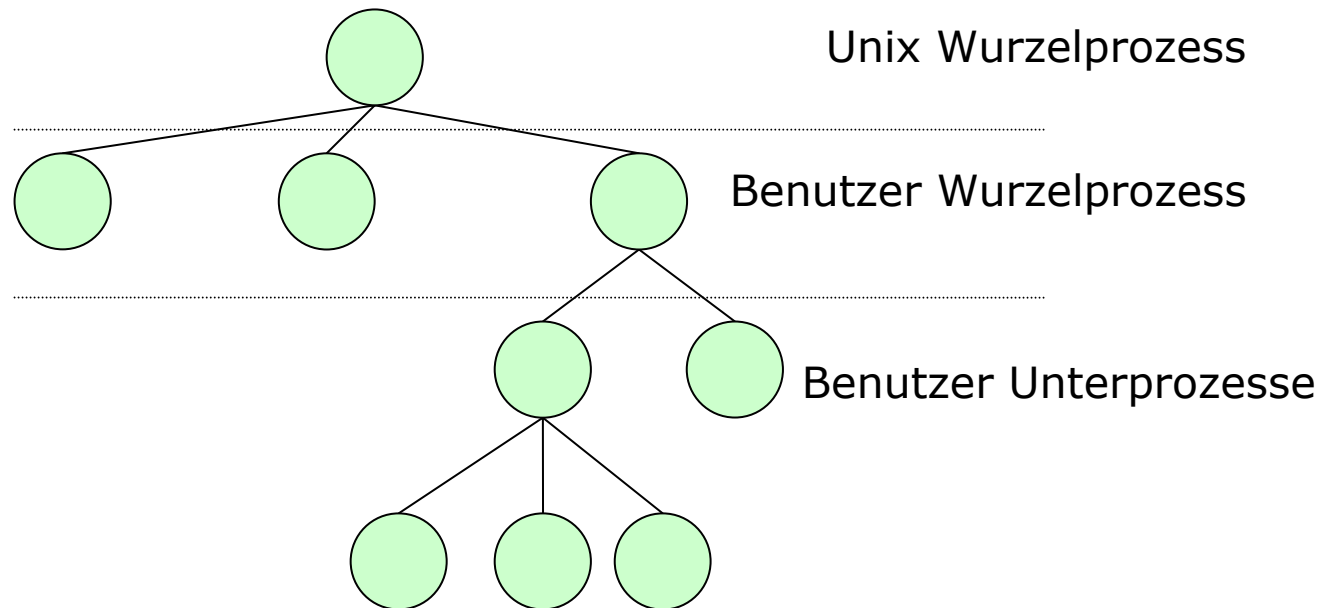
Das Programm existiert nur einmal (als Datei auf der Platte), jede Instanz des Programms, die sich im Speicher befindet ist ein Prozess.

Jeder Prozess hat unter Unix mehrere Eigenschaften, die wichtigste ist die Prozess-ID, eine eindeutige Nummer, die ihn ansprechbar macht. (**PID**)

Als Prozess-ID werden in der Regel 16 Bit Zahlen verwendet, die beim Systemstart von 0 ab hoch gezählt werden. Das bedeutet, dass Unix nicht mehr als 65536 Prozesse gleichzeitig laufen lassen kann, eine Grenze, die sicherlich weit genug gesteckt ist. Sobald die PID 65535 erreicht ist, werden die nachfolgenden Prozesse wieder mit den freigewordenen Prozessen ab PID 2 nummeriert.

Neben der Prozess-ID hat jeder Prozess einen eindeutigen Elternprozess. Jeder Prozess wird von einem Elternprozess gestartet (mit Ausnahme des Urprozesses). Die Nummer des Prozesses, der den aktuellen Prozess gestartet hat, ist die **PPID** - die Parent-Process-ID.

Somit besteht ein laufendes Unix System aus einer baumartigen Struktur von Prozessen:



Folgende grundlegenden Arten von Prozessen existieren:

- Ein Prozess ist in der Lage, **Signale** von anderen Prozessen zu empfangen. Diese Signale können den Prozess dazu bringen, bestimmte Handlungen vorzunehmen, etwa seine Konfigurationsdatei neu einzulesen oder sich zu beenden.

- **Daemon**-Prozesse sind einfach Prozesse, die im Hintergrund *lauern* und warten, bis sie gebraucht werden. In der Regel sind das Serverprozesse, die darauf warten, dass ein Client ihre Dienste in Anspruch nimmt. Das Charakterisierende an solchen Prozessen ist, dass sie kein eigenes Ausgabeterminal haben.
- Zombies sind Prozesse, die eigentlich schon ihre Arbeit erledigt haben und damit gestorben sind, aber noch auf einen anderen Prozess warten müssen, der langsamer ist und ihre Daten braucht. Zombies verbrauchen keine Rechenzeit mehr, belegen aber noch Arbeitsspeicher.

## 2. Prozessgenerierung und Synchronisation

Ein **Image** ist die Ablaufumgebung eines Programms im Rechner:

- Speicherabbild (Programmcode und Daten)
- Werte des Register,
- Zustand der geöffneten Dateien,
- Name des aktuellen Verzeichnisses und
- Stand des Befehlszählers.

Zur Erzeugung und Synchronisation von Prozessen stehen im Wesentlichen vier Systemaufrufe bzw. Kommandos zur Verfügung:

1. Der Systemaufruf **fork()** erzeugt eine Kopie eines Prozesses (des Aufrufers) mit gleichem Image. Beide Prozesse (Vater und Sohn) laufen mit gleichem Stand des Befehlszählers und gleicher Umgebung weiter.

**Achtung:** gleiche Umgebung, **nicht** selbe Umgebung, d.h. **keine** gemeinsamen Daten.

Vater und Sohn identifizieren sich anhand des Wertes, den fork zurück liefert:

0 Sohnprozess

>0 Vater

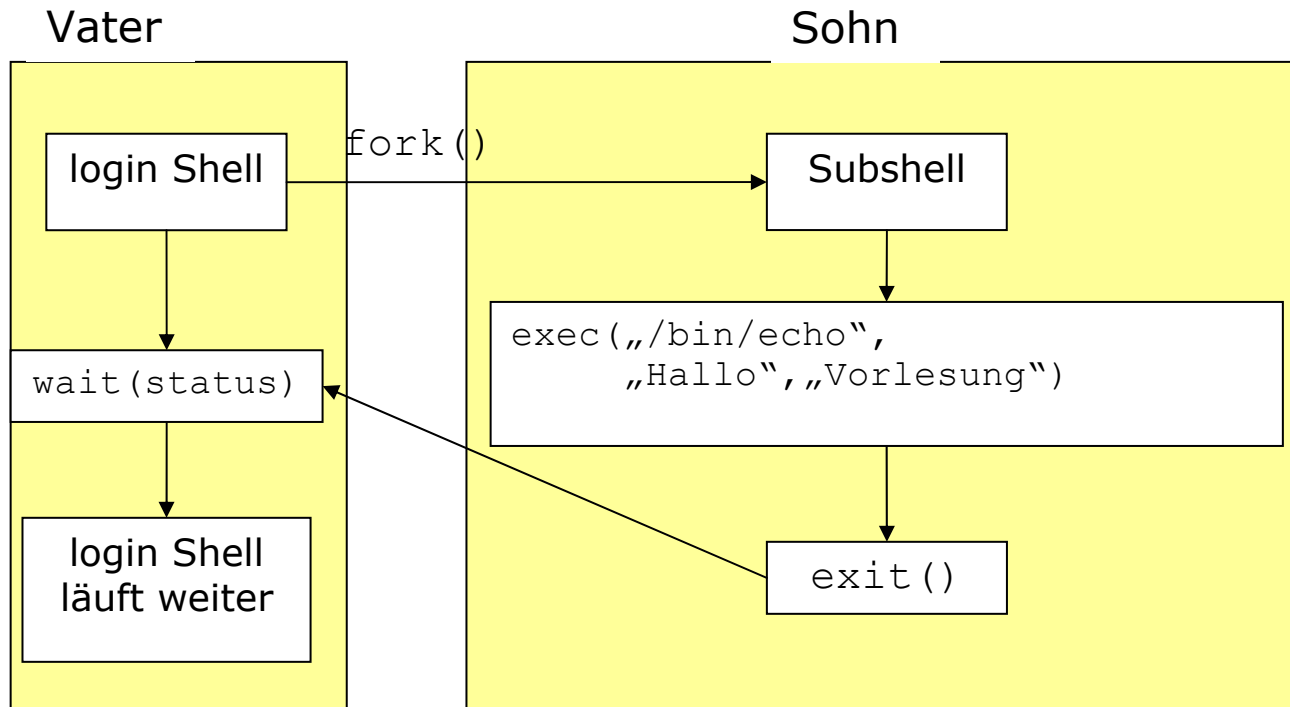
-1 Fehler: Vater kann keinen Prozess erzeugen

2. Der Systemaufruf **exec**(file, arg1, arg2, ...) überlagert das Image (seinen Programmcode) eines Prozesses mit dem Programm, das in der Datei „file“ steht und führt es mit den angegebenen Argumenten aus.
3. Der Systemaufruf **wait**(status) lässt den Vaterprozess warten, bis der Sohn beendet ist; wait liefert dabei die Prozessnummer des terminierten Sohns.
4. Der Systemaufruf **exit**() beendet einen Prozess und aktiviert evtl. einen wartenden Vaterprozess.

exit und wait sind auch auf Ebene der Shell als Kommando realisiert.

Beispiel:

Nach dem Login ist die Shell der einzige Benutzerprozess (login shell). Soll ein Kommando, z.B. das Kommando „echo“ ausgeführt werden, dupliziert sich die Shell (durch fork) und überlagert der Kopie dann den Code des echo-Kommandos (exec) während die Originalshell auf das Terminieren des echo Programms wartet.



Wenn ein Prozess im Hintergrund abläuft, unterbleibt der wait-Aufruf des Sohns, bei der Subshell wird dann die Standardausgabe auf „/dev/null“ gelenkt.

### 3. Prozesszustände

Obgleich jeder Prozess eine unabhängige Einheit ist, muss er mit anderen Prozessen in Beziehung treten können.

Ein Prozess kann z.B. eine Ausgabe erzeugen, die von anderen Prozessen als Eingabe benötigt wird, wie z.B. durch das folgende Kommando:

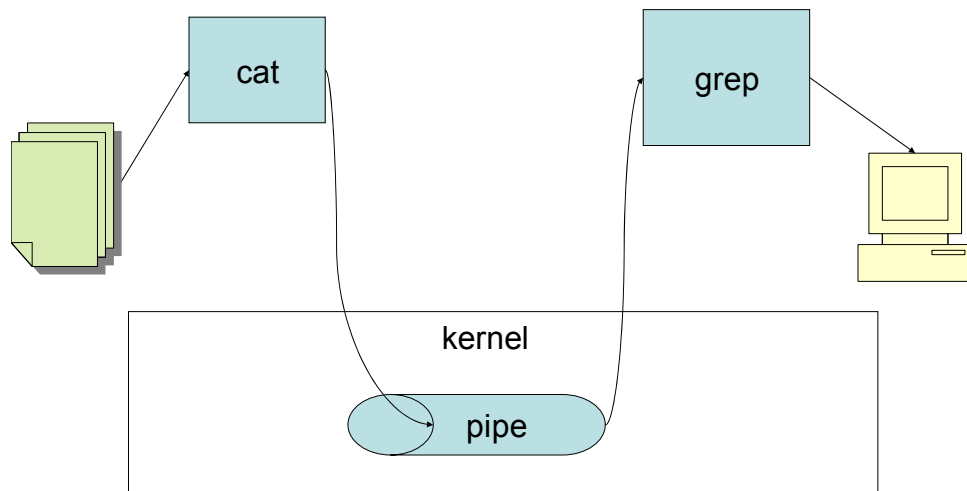
```
$ cat kapitel1 kapitel2 | grep Unix  
...  
$
```

Dabei werden zwei Prozesse gestartet:

- cat schreibt die beiden Dateien in eine Pipe,
- grep liest aus der Pipe.

Eine Pipe ist ein FIFO Speicher, der als Speicherbereich im Hauptspeicher realisiert ist:

## Pipe mit cat und grep

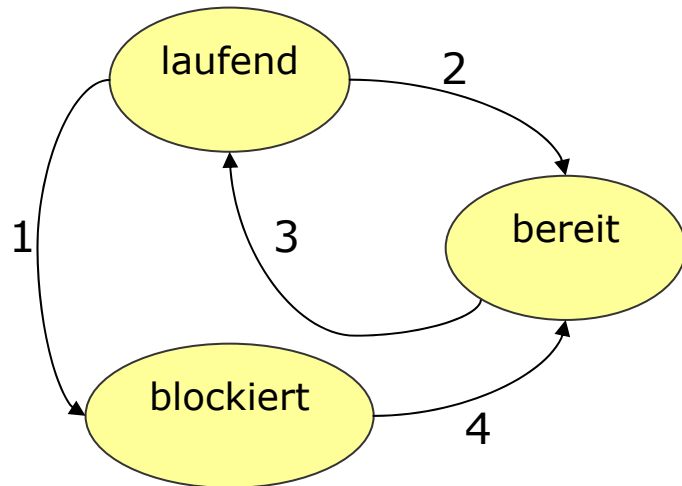


Abhängig von der relativen Geschwindigkeit der beide Prozesse kann es sein, dass grep bereit ist, nach Mustern zu suchen, aber warten muss, weil noch keine Daten in der Pipe stehen. Auch ist es möglich, dass ein Prozess bereit zum Laufen ist, aber das Betriebssystem gerade einem anderen Prozess die CPU zugewiesen hat.

Insgesamt kann ein Prozess P im Zustand laufend, bereit oder blockiert sein:

- P laufend: im Augenblick wird die CPU von P benutzt
- P blockiert: P wartet auf ein externes Ereignis
- P bereit: P ist lauffähig, aber zeitweise gestoppt, da ein anderer Prozess die CPU belegt

Dabei gibt es folgende Zustandsübergänge:



1. Prozess blockiert bei Ereignis
2. Scheduler wählt anderen Prozess
3. Scheduler wählt diesen Prozess
4. Ereignis ist eingetreten

## 4. Systemprozesse

Unix bietet ein echtes präemptives Multitasking, das jedem Prozess, der gerade läuft, eine bestimmte Menge Rechenzeit zuweist. Der Prozeß, der für diese Zuweisung zuständig ist, heißt **Scheduler** und hat grundsätzlich die **Prozeß-ID 0**. Das bedeutet, dass er grundsätzlich der erste Prozess ist, der gestartet wird. In den meisten Unixen bleibt dieser Prozess unsichtbar.

Der Scheduler startet als erstes den Prozess, der zwangsläufig immer die **PID 1** bekommt, den **Init-Prozess**. Dieser Prozess ist der Vater aller weiteren Prozesse des Systems. Er ist zuständig für die **Initialisierung des Systems** und damit auch für die gesamten weiteren Prozesse, die im Verlauf noch gestartet werden.

Der Init-Prozess startet nun zwei verschiedene Arten von Prozessen, die wir uns im Folgenden etwas genauer betrachten wollen: Dämon-Prozesse und Benutzer-Prozesse.

- Dämon-Prozesse (Service Prozesse)

Als Dämonen werden diejenigen Prozesse bezeichnet, die selbst kein Terminal besitzen und benutzen. Meist handelt es sich dabei um Serverprozesse, also Dienstprogramme, die im Hintergrund ablaufen und zu bestimmten Anlässen bestimmte Aktionen durchführen.

Der Init-Prozess startet, je nachdem, wie er konfiguriert ist, verschiedene solcher Prozesse direkt. Das bedeutet, dass diese Dämon-Prozesse als Eltern-Prozess direkt den Init-Prozess haben.

- Benutzer-Prozesse

Benutzer-Prozesse sind die Prozesse, die einem bestimmten Terminal zugeordnet sind. Ein Terminal ist im einfachsten Fall ein Bildschirm und eine Tastatur, die über eine serielle Lei-

tung mit dem Computer verbunden ist. Diese Terminalleitungen werden noch heute als TTY-Leitungen bezeichnet, von TeleTYpe (Fernschreiber) - tatsächlich waren es Fernschreiber, die die ersten Unix-Maschinen angesteuert hatten.

Um an einem Terminal zu arbeiten, muss sich ein/e Benutzer/in zunächst einloggen, d.h., sich ausweisen. Das geschieht üblicherweise durch die Nennung eines Usernamens und eines dazu passenden Passworts. Ist dieser Vorgang abgeschlossen, so wird ein Kommandointerpreterprogramm (Shell) gestartet, und der/die Benutzer/in kann mit dem System arbeiten.

#### 4.1. Der Startvorgang der Benutzerprozesse

Zunächst startet der Init-Prozess für jedes Terminal, das aktiv sein soll, einen so genannten **GETTY-Prozess**. Dieser Prozess schreibt eine Begrüßungsmeldung (aus der Datei /etc/issue) auf den Bildschirm und fordert dann zur Eingabe des Benutzernamens auf.

```
Rechnername login:
```

Diese Aufforderung steht solange auf dem Bildschirm, bis jemand tatsächlich daran arbeiten will und folglich seinen Benutzernamen eingibt. Der GETTY-Prozess nimmt diesen Namen entgegen und startet danach ein Programm namens login, dem er den gelesenen Benutzernamen als Parameter mitgibt.

Der **login-Prozess** gibt jetzt die Aufforderung auf den Bildschirm aus, das Passwort einzugeben:

```
password:
```

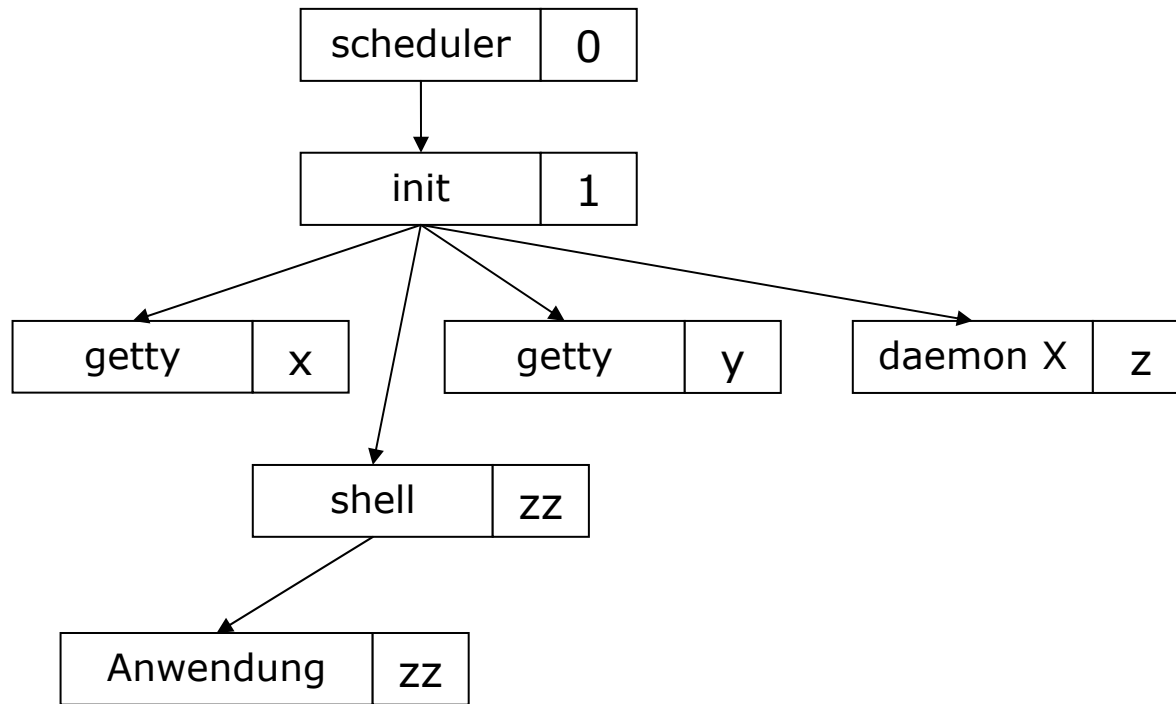
Das eingegebene Passwort wird jetzt vom login-Prozess verschlüsselt und dann mit dem ver-

schlüsselt abgespeicherten Passwort des Users verglichen. Sollte dieser Vergleich scheitern, so fragt der Prozess erneut nach dem Passwort, insgesamt dreimal, so dass nach dem dritten falschen Passwort erneut der GETTY-Prozess startet und wieder nach dem Benutzernamen fragt.

Wenn das Passwort korrekt war, so startet der login-Prozess jetzt einen Kommandointerpreter (oder in seltenen Fällen andere Programme). Dieser Kommandointerpreter, der in etwa funktioniert wie die DOS-Eingabeaufforderung wird unter Unix *Shell* genannt. Damit die beiden Prozesse GETTY und LOGIN nicht weiter im Speicher verbleiben (und dort unnötig Platz verschwenden), begehen diese Prozesse sozusagen Selbstmord und der Init-Prozess adoptiert die aufgerufene Shell. Die Shell hat jetzt also als Eltern-Prozess-ID (PPID) die Nummer 1.

Von diesem Zeitpunkt ab, können die Anwender jetzt ihrerseits Prozesse starten, indem sie Programme aufrufen.

Das folgende Bild demonstriert den Systemstart und ein einen an einem Terminal arbeitenden Benutzer:



## 5. Befehle für das Prozess-System

Unix stellt mehrere Kommandos zur Verfügung, mit der das Prozess-System angesehen werden kann.

### 5.1.ps

Um alle Prozesse aufzulisten, die gerade laufen, gibt es das Kommando ps (Process-Status), das zunächst nur die eigenen Prozesse auflistet. ps stellt immer nur eine Momentaufnahme dar, d.h., es werden genau die Prozesse aufgelistet, die im Augenblick laufen. Dynamische Veränderungen sind nicht darstellbar.

ps kennt eine ganze Menge verschiedener Optionen die das Verhalten bestimmen. Die wichtigsten sind (Achtung – diese Optionen sind in unterschiedlichen Unix Derivaten verschieden):

- l** Langes Format
- u** User Format (Mit User und Startzeit)
- a** Alle Prozesse, auch die anderer User
- x** Auch Daemon-Prozesse (Ohne eigene TTY-Leitung)
- f** Forest (Wald) Format - Der Prozessbaum wird dargestellt
- w** Wide (breite) Ausgabe - Zeilen werden nicht abgeschnitten

So kann also mit dem Befehl

```
$ ps
  PID TTY          TIME CMD
 3087 pts/1        00:00:00 bash
 3098 pts/1        00:00:00 ps
$
```

eine Liste aller Prozesse des Benutzers ausgegeben werden.

Alle gerade ablaufenden Prozesse sind mittels folgendem Kommando abrufbar:

```
$ ps -e
...
$
```

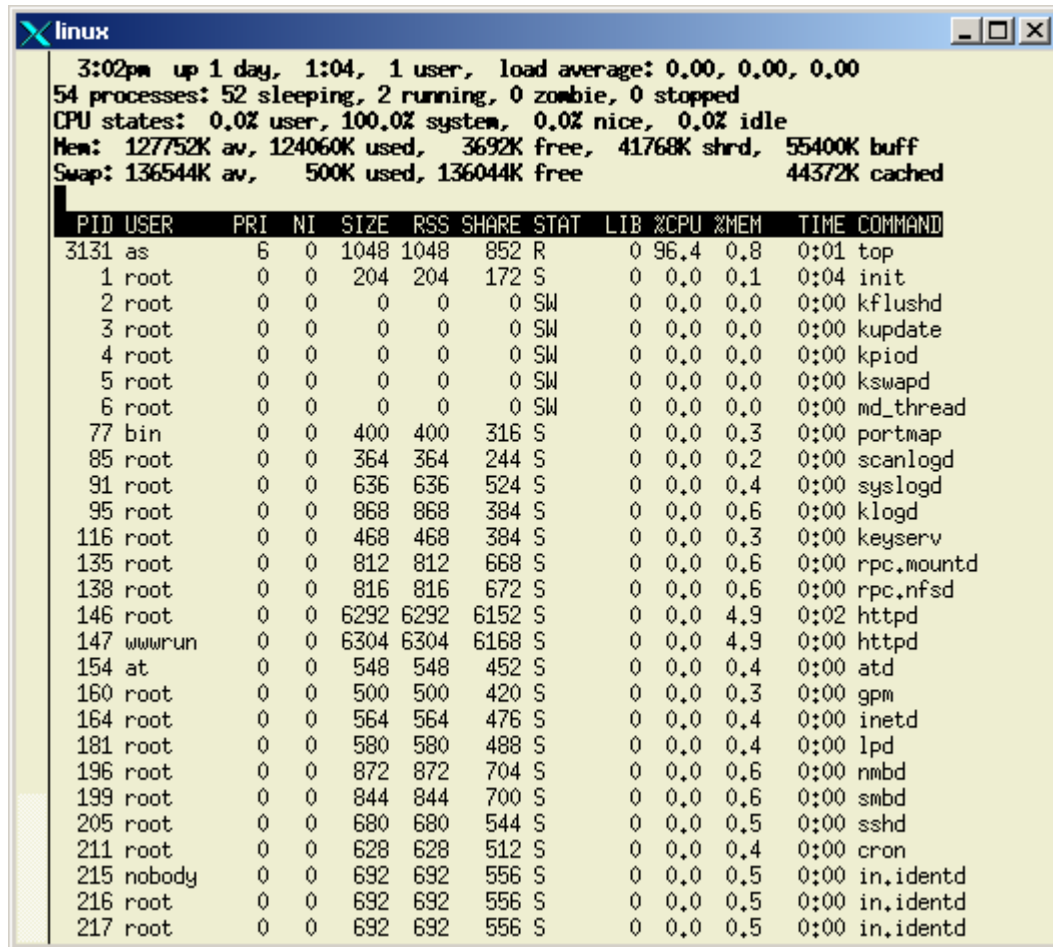
Die Abhängigkeiten der eigenen Prozesse sieht man durch:

```
$ ps -f
UID          PID  PPID  C  STIME TTY          TIME CMD
as           3087   3086  0  14:49 pts/1        00:00:00 -bash
as           3106   3087  0  14:57 pts/1        00:00:00 ps -f
$
```

## 5.2.top

Wenn statt der Auflistung der Momentaufnahme eine ständig aktualisierte Liste gewünscht wird, so ist das Programm top das benötigte Werkzeug. Dieses Programm gibt eine Liste der Prozesse aus und aktualisiert diese nach einer bestimmten Wartezeit (voreingestellt sind 5 Sek). Der

Nachteil ist, dass nur so viele Prozesse aufgelistet werden, wie auf den entsprechenden Bildschirm passen. Eine vernünftige Anwendung ist somit nur in einem entsprechend großem xterm-Fenster möglich.



```
linux
3:02pm up 1 day, 1:04, 1 user, load average: 0.00, 0.00, 0.00
54 processes: 52 sleeping, 2 running, 0 zombie, 0 stopped
CPU states: 0.0% user, 100.0% system, 0.0% nice, 0.0% idle
Mem: 127752K av, 124060K used, 3692K free, 41768K shrd, 55400K buff
Swap: 136544K av, 500K used, 136044K free 44372K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT   LIB  %CPU  %MEM  TIME COMMAND
 3131 as         6   0  1048 1048  852 R       0 96.4  0.8   0:01 top
   1 root         0   0   204  204  172 S       0  0.0  0.1   0:04 init
   2 root         0   0     0   0   0 SW      0  0.0  0.0   0:00 kflushd
   3 root         0   0     0   0   0 SW      0  0.0  0.0   0:00 kupdate
   4 root         0   0     0   0   0 SW      0  0.0  0.0   0:00 kpiod
   5 root         0   0     0   0   0 SW      0  0.0  0.0   0:00 kswapd
   6 root         0   0     0   0   0 SW      0  0.0  0.0   0:00 md_thread
   77 bin         0   0   400  400  316 S       0  0.0  0.3   0:00 portmap
   85 root         0   0   364  364  244 S       0  0.0  0.2   0:00 scanlogd
   91 root         0   0   636  636  524 S       0  0.0  0.4   0:00 syslogd
   95 root         0   0   868  868  384 S       0  0.0  0.6   0:00 klogd
  116 root         0   0   468  468  384 S       0  0.0  0.3   0:00 keyserv
  135 root         0   0   812  812  668 S       0  0.0  0.6   0:00 rpc.mountd
  138 root         0   0   816  816  672 S       0  0.0  0.6   0:00 rpc.nfsd
  146 root         0   0  6292 6292 6152 S       0  0.0  4.9   0:02 httpd
  147 wwwrun      0   0  6304 6304 6168 S       0  0.0  4.9   0:00 httpd
  154 at          0   0   548  548  452 S       0  0.0  0.4   0:00 atd
  160 root         0   0   500  500  420 S       0  0.0  0.3   0:00 gpm
  164 root         0   0   564  564  476 S       0  0.0  0.4   0:00 inetd
  181 root         0   0   580  580  488 S       0  0.0  0.4   0:00 lpd
  196 root         0   0   872  872  704 S       0  0.0  0.6   0:00 nmbd
  199 root         0   0   844  844  700 S       0  0.0  0.6   0:00 smbd
  205 root         0   0   680  680  544 S       0  0.0  0.5   0:00 sshd
  211 root         0   0   628  628  512 S       0  0.0  0.4   0:00 cron
  215 nobody     0   0   692  692  556 S       0  0.0  0.5   0:00 in.identd
  216 root         0   0   692  692  556 S       0  0.0  0.5   0:00 in.identd
  217 root         0   0   692  692  556 S       0  0.0  0.5   0:00 in.identd
```

Als Programm, das interaktiv benutzbar ist, hat top natürlich auch Befehlstasten, die den Ablauf verändern. Die folgenden Tasten sind die wichtigsten Befehle:

- Leertaste**    Sofortiges Update der Prozesse
- Strg-L**        Bildschirm neu aufbauen
- h oder ?**      Darstellung einer Hilfeseite
- i**                Ignoriere schlafende und Zombie-Prozesse (i ist ein Wechselschalter, erneutes Drücken bewirkt, daß diese Prozesse wieder angezeigt werden).
- r**                Renice - Damit kann einem Prozeß ein neuer Nice-Wert gegeben werden (sofern der User das Recht dazu hat). Das Programm fragt nach PID und Nice-Wert.
- k**                Kill - Entspricht dem Programm kill - siehe weiter unten. Damit können Signale an Prozesse geschickt werden.
- s**                Damit kann die Zeit verändert werden, die zwischen dem Auffrischen des Bildschirms gewartet wird. Eingabe in Sekunden. Vorsicht, eine zu kurze Zeit bringt den Rechner schnell in die Knie. Eine 0 steht für dauernde Neudarstellung ohne Wartezeit, Voreingestellt sind meist 5 Sekunden.
- w**                Schreibt die aktuelle Konfiguration in die Datei ~/.toprc - Damit wird nach dem Neustart von top diese Konfiguration wieder geladen.
- q**                Quit - Beendet das Programm

top kann verschiedene Felder darstellen, die mit dem f-Befehl ausgewählt werden können. Die einzelnen Felder haben folgende Bedeutung:

<b>PID</b>	Die Process-ID des Prozesses
<b>PPID</b>	Die Parent Process ID des Prozesses
<b>UID</b>	Die User ID des Users, dem der Prozeß gehört
<b>USER</b>	Der Username des Users, dem der Prozeß gehört
<b>PRI</b>	Die Priorität des Prozesses. Höhere Werte bedeuten höhere Priorität.
<b>NI</b>	Der Nice-Wert des Prozesses. Höhere Werte bedeuten geringere Priorität.
<b>SIZE</b>	Die Größe des Codes plus Daten plus Stack in KiloByte
<b>TSIZE</b>	Die Größe des Codes in KiloByte. ELF Prozesse werden nicht korrekt dargestellt
<b>DSIZE</b>	Die Größe der Daten und Stack in Kilobyte. ELF Prozesse werden nicht korrekt dargestellt
<b>TRS</b>	Text Resident Size - Die Größe des residenten Code-Blocks in KiloByte
<b>SWAP</b>	Größe des ausgelagerten Bereichs des Tasks
<b>D</b>	Größe der als Dirty markierten Speicherseiten

<b>LIB</b>	Größe der Library-Speicherseiten - Funktioniert nicht bei ELF-Prozessen.
<b>RSS</b>	Die Größe des physikalische Speichers, den das Programm benutzt. Für ELF-Format werden hier auch die Libraries mitgezählt, bei a.out Format nicht.
<b>SHARE</b>	Die Größe der benutzten Shared-Libraries des Prozesses.
<b>STAT</b>	Der Status des Prozesses. Das kann entweder ein <b>S</b> für schlafend, <b>D</b> für ununterbrechbar schlafend (dead), <b>R</b> für laufend (running) oder <b>T</b> für angehalten (traced). Dieser Angabe kann noch ein <b>&lt;</b> für einen negativen Nice-Wert, ein <b>N</b> für einen positiven Nice-Wert oder ein <b>W</b> für einen ausgelagerten Prozeß folgen. (Das W funktioniert nicht richtig für Kernel-Prozesse)
<b>WCHAN</b>	Die Kernelfunktion, die der Task gerade nutzt.
<b>TIME</b>	Die gesamte CPU-Zeit, die der Prozeß verbraucht hat, seit er gestartet wurde.
<b>%CPU</b>	Die CPU-Zeit, die der Prozeß seit dem letzten Bildschirm-Update verbraucht hat, dargestellt als Prozentsatz der gesamten CPU-Zeit.
<b>%MEM</b>	Der Anteil des Speichers, den der Task nutzt.
<b>COMMAND</b>	Das Kommando, mit dem der Prozeß gestartet wurde.
<b>TTY</b>	Die Terminalleitung des Prozesses.

## 5.3.kill

Mit dem kill-Befehl kann man einem oder mehreren Prozessen Signale senden. Diese Signale können den Prozess dann dazu bewegen, bestimmte Aktionen vorzunehmen. Der Befehl kill erwartet als Parameter zuerst das zu sendende Signal (mit vorgestelltem Bindestrich), entweder als Zahl oder als symbolischer Name, dann die PID des Prozesses, dem es geschickt werden soll. Es können auch mehrere ProzeßIDs angegeben werden.

Mit kill -l erhält man eine Liste der gültigen Signale, bei Linux sind das in der Regel die Folgenden:

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR			

Diese Signale haben, bis auf das Signal Nr. 9 (SIGKILL) und 15 (SIGTERM), keine festgelegte Bedeutung. Der Programmierer eines Programms kann festlegen, wie das Programm auf ein bestimmtes Signal reagieren soll. Das Signal 15 ist das voreingestellte Signal, das geschickt wird, wenn dem kill-Befehl kein Signal angegeben wird und fordert ein Programm auf, sich zu beenden. Ein Prozess muss aber nicht zwangsläufig abbrechen, wenn es dieses Signal empfängt. Das Signal 9 hingegen ist der "Todesschuss", diesem Signal kann sich kein Prozess entziehen.

```
$ sleep 20 &
[1] 3135
$ ps
  PID TTY          TIME CMD
 3087 pts/1        00:00:00 bash
 3135 pts/1        00:00:00 sleep
 3136 pts/1        00:00:00 ps
$ kill -9 3135
[1]+  killed                  sleep 20
$ ps
  PID TTY          TIME CMD
 3087 pts/1        00:00:00 bash
 3137 pts/1        00:00:00 ps
$
```