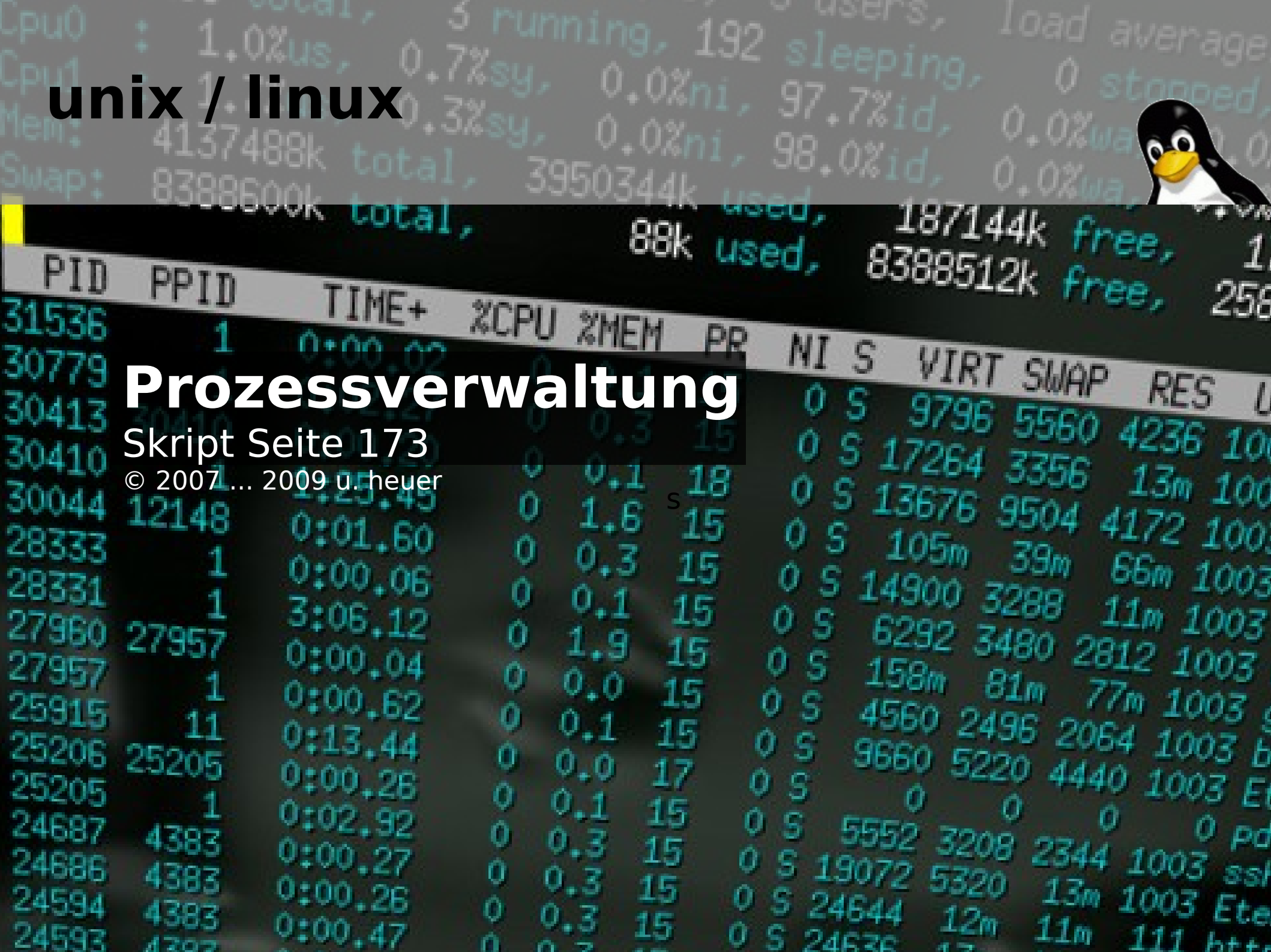
# unix / linux



# Prozessverwaltung

Skript Seite 173

# Ziele

- **Den Prozessbegriff von Linux verstehen**
- **Die wichtigsten Kommandos zum Abrufen von Prozessinformationen kennen**
- **Prozesse beeinflussen und beenden können**

# Prozess

- Ein Prozess ist im wesentlichen ein "laufendes Programm".

**Prozesse haben:**

- Code (das Programm), der ausgeführt wird
- Daten, auf denen der Code operiert.
- diverse Attribute für die interne Verwaltung des Betriebssystems.

# Prozess Attribute

Intern im Kernel werden die Attribute in der Datenstruktur `task_struct` repräsentiert.

Im aktuellen Kernel v2.6 verwaltet Linux eine task-Liste, hinter der sich ein doppelt verkette Liste von `task_struct` verbirgt.

Das bedeutet, dass es – theoretisch – keine maximale Anzahl Prozesse gibt.

# Prozess-Attribute PID

**PID** Process Identity

Die eindeutige Prozess-Nummer, kurz PID.

Die PID dient zur unmissverständlichen Identifizierung des Prozesses und kann immer nur einmal belegt werden.

Die PID ist in der Regel im Bereich von 1 ... 32768

# Prozess-Attribute PPID

**PPID** Parent Process Identity

Alle Prozesse kennen die Prozessnummer des Eltern-Prozesses, kurz PPID.

Jeder Prozess kann Ableger hervorbringen, die dann einen Verweis auf ihren Erzeuger enthalten.

# Prozess-Attribute PPID

Der einzige Prozess, der keinen solchen Eltern-Prozess hat ist der beim Systemstart erzeugte "Prozess" mit der PID 0. Aus diesem geht der "Init"-Prozess mit der PID 1 hervor, der dann "Ur-ahn" aller anderen Prozesse im System wird.

Neue Kernel (ab 2.6.22) hängen die Kernel-Threads Prozesse neuerdings an den Prozess mit der PID 0 an.

# Prozess-Attribute PPID

**# Kernel 2.6.19.1**

```
$ ps ax -o pid,ppid,cmd --forest
  PID  PPID CMD
    1     0 init [2]
    2     1 [migration/0]
    3     1 [ksoftirqd/0]
    6     1 [events/0]
    8     1 [khelper]
    9     1 [kthread]
   83     9  \_ [kblockd/0]
   85     9  \_ [kacpid]
  156     9  \_ [kseriod]
  177     9  \_ [pdflush]
  178     9  \_ [kswapd0]
  179     9  \_ [aio/0]
  813     9  \_ [scsi_eh_0]
  837     9  \_ [kpsmoused]
  839     9  \_ [exec-osm/0]
...
```

**# Kernel 2.6.22.1**

```
$ ps ax -o pid,ppid,cmd --forest
  PID  PPID CMD
    1     0 init [2]
    2     0 [kthreadd]
    3     2  \_ [migration/0]
    4     2  \_ [ksoftirqd/0]
    5     2  \_ [watchdog/0]
    9     2  \_ [events/0]
   11     2  \_ [khelper]
   94     2  \_ [kblockd/0]
   96     2  \_ [kacpid]
  228     2  \_ [ata/0]
  230     2  \_ [ata_aux]
  231     2  \_ [ksuspend_usbd]
  234     2  \_ [khubd]
  236     2  \_ [kseriod]
  263     2  \_ [kswapd0]
....
```

# Prozess-Attribute

**Benutzer**, **Gruppen**
Jeder Prozess ist **einem einzigen** Benutzer und **einer Reihe** von Gruppen zugeordnet.

Diese Attribute sind wichtig, um die Zugriffsrechte des Prozesses auf Dateien, Geräte usw. zu bestimmen. Ausserdem darf der Benutzer, dem der Prozess zugeordnet ist, den Prozess anhalten, beenden oder anderweitig beeinflussen.

Die Benutzer- und Gruppen-Zuordnungen werden an die Kindprozesse bei der Erzeugung vererbt.

# /proc/<PID>

Die Daten, die Linux verwendet um die Prozesse zu verwalten, sind im Pseudo-Filesystem /proc einsehbar.

Für jeden Prozess wird ein Verzeichnis mit der **<PID>** als Verzeichnisname erzeugt.
In diesem Verzeichnis befinden sie alle relevanten Daten, die das Betriebssystem verwendet, um den Prozess zu verwalten.

Ein Prozess kann mit dem "magic"-Link **`/proc/self`** seine eigenen Daten lesen.

# /proc/<PID>/

**cmdline** **Com**mand **Line**
in dieser Datei stehen die Optionen und Argumente, die dem Command mit gegeben wurden. Die einzelnen Optionen und Argumente sind durch ein Nullbyte (0x00, \0) getrennt.

**cwd** **C**urrent **W**orking **D**irectory.
Der symbolische Link **cwd** zeigt auf das Verzeichnis, wo sich der entsprechende Prozess gerade 'aufhält'

# /proc/<PID>/

**environ** in dieser Datei is das gesamte Environement vom Prozess enthalten.
Die einzelnen Variablen sind durch ein Nullbyte (0x00, \0) getrennt.

**fd** Im Verzeichnis fd (**F**ile**D**escriptor) sind Links auf die vom Prozess geöffneten Files eingetragen.

**status** Diese Datei zeigt die verschiedensten Statusinformationen in einem für Menschen lesbaren Form dar.

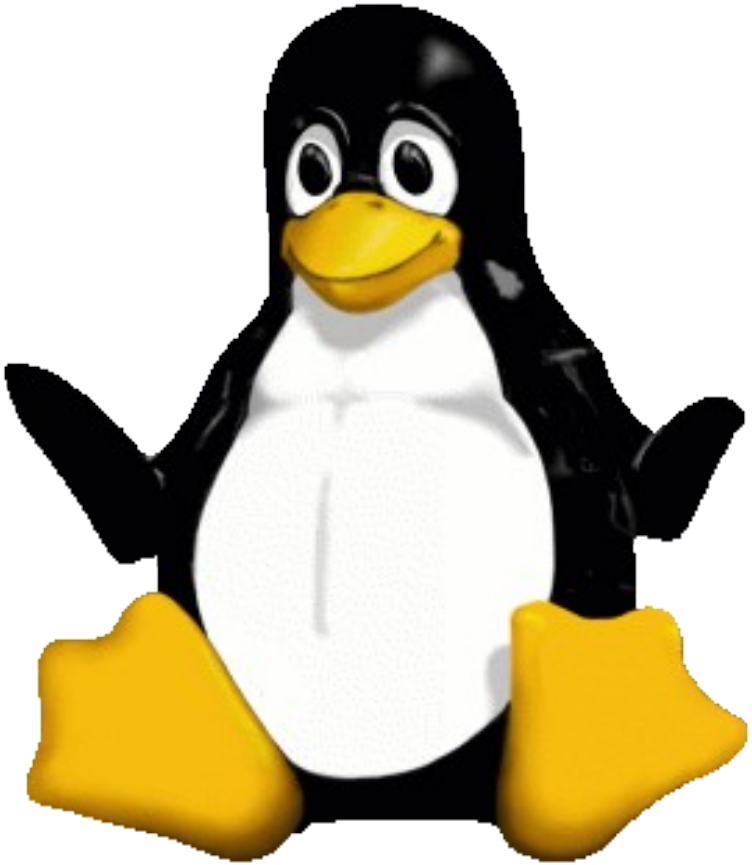
# /proc/<PID>/

Neben den aufgelisteten Einträgen befinden sich weitere Informationen zu den Prozessen in den jeweiligen Subverzeichnungen.

Die detaillierte Beschreibung der einzelnen Einträge können in der Manpage **`proc(5)`** und in der Dokumentation des Kernels nachgelesen werden.

/usr/src/linux/Documentation/filesystems/proc.txt

# Fragen ?



# Übungen

**9.1** Wie können Sie sich die Umgebungsvariablen eines beliebigen Ihrer Prozesse anschauen? (Tipp: /proc-Dateisystem.)

**9.2** Wie gross ist die maximal mögliche PID?
Was passiert, wenn diese Grenze erreicht wird?
(Tipp: Suchen Sie in den Dateien im /usr/include/linux nach der Zeichenkette »PID_MAX«.)

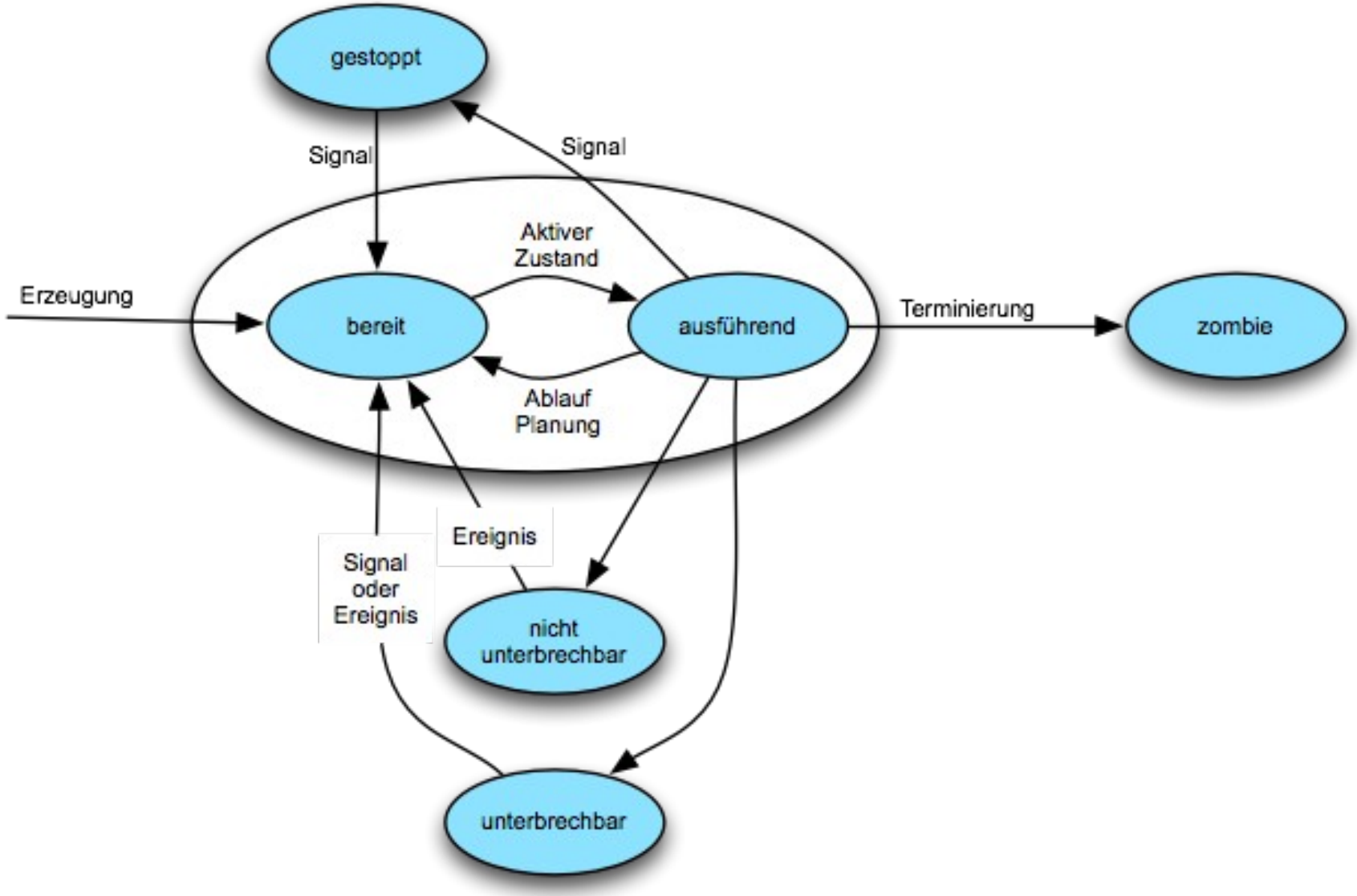
# Prozesszustände

Linux verwendet preemptives Multitasking, das bedeutet, dass ein Scheduler die CPU-Zeit in Form von Zeitscheiben an die wartenden Prozesse verteilt.

In der letzten Kernel Versionen (2.6.23, 2.6.24, ) wurde ein neuer Scheduler eingebaut, der die Prozesse fairer behandelt, als bisher.

Linux verwendet ein Prozess-Modell mit 6 Zuständen

# Prozesszustände



# Prozesszustände

**aktiv**:
Dieser Zustand entspricht eigentlich zwei Zuständen. Ein aktiver Prozess wird entweder ausgeführt oder er ist bereit, ausgeführt zu werden.

**unterbrechbar**:
Dieser Zustand ist ein blockierter Zustand, in dem ein Prozess auf ein Ereignis, wie z.B. die Beendigung einer I/O-Operation, die Verfügbarkeit einer Ressource oder ein Signal von einem anderen Prozess wartet.

# Prozesszustände

**nicht unterbrechbar**:
Es handelt sich hierbei um einen weiteren blockierten Zustand. Der Unterschied zwischen diesem Zustand und dem unterbrechbaren Zustand ist der, dass ein Prozess im nicht unterbrechbaren Zustand direkt auf eine Hardware-Bedingung wartet und daher kein Signal annehmen darf (und kann).

**gestoppt**:
Der Prozess wurde angehalten und kann nur durch eine Aktion seitens eines anderen Prozesses wieder aufgenommen werden.

# Prozesszustände

**zombie**:
Der Prozess wurde terminiert, aber aus irgend einem Grund muss seine Task-Struktur in der Prozesstabelle noch erhalten bleiben.

Meistens wird darauf gewartet, dass der Eltern-Prozess das Ende vom Kind-Prozess entgegen genommen hat [1]. Bis das erfolgt, muss die Task-Struktur behalten werden.

[1] Dies erfolgt durch den Aufruf der Funktion wait(2) im Eltern Process.

# Prozess Exitcode

Wird ein Prozess beendet, so gibt er seinem Elternprozess einen Rückgabewert – den Exitcode – retour.

Dieser Wert ist eine Zahl zwischen 0 und 255.

In der Regel gilt, wenn alles wie geplant abgelaufen ist, wird der Exitcode 0 zurückgegeben.

Ist ein Ereignis / Fehler aufgetreten, so ist der Exitcode > 0.

# Prozess Exitcode

In der Shell ist der Exitcode vom letzten ausgeführten Befehl in der Variable **$?** hinterlegt.

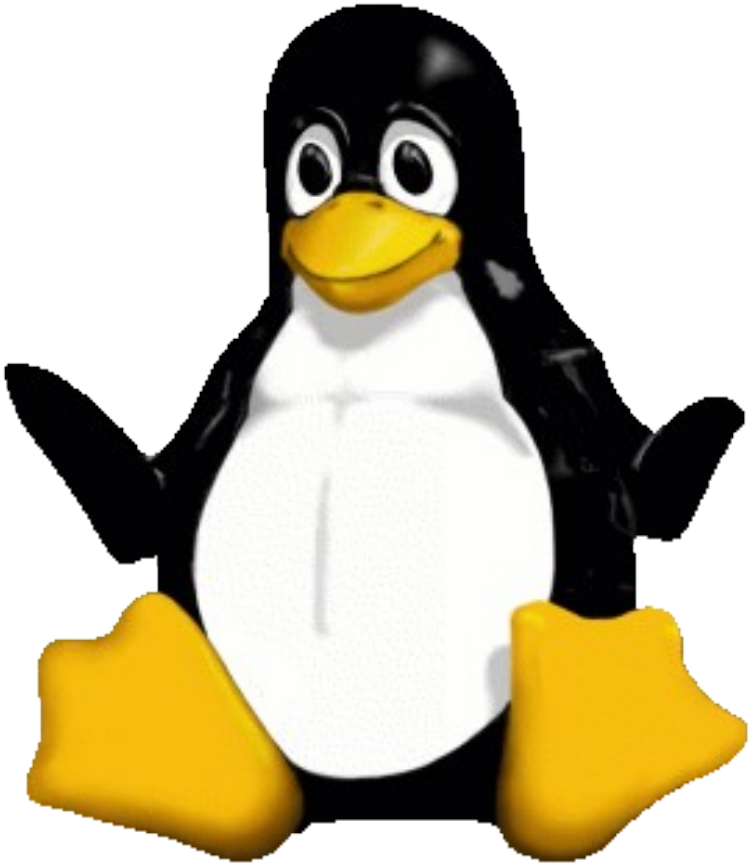
```
$ grep root /etc/passwd > /dev/null; echo $?
0
```

→ der String 'root' war im File vorhanden.

```
$ grep r00t /etc/passwd > /dev/null; echo $?
1
```

→ der String 'r00t' konnte nicht gefunden werden. s

# Fragen ?



# Übungen

**9.3** Starten Sie einen `xclock`-Prozess im Hintergrund.
In der Shellvariablen `$!` finden Sie die PID dieses Prozesses. Prüfen Sie den Zustand des Prozesses mit dem Befehl `` `grep ^State: /proc/$!/status` ``.
Stoppen Sie die `xclock` anschliessend, indem Sie sie in den Vordergrund holen und mit [ctrl]-z anhalten.
Wie ist nun der Prozesszustand?

**9.4** Beschreiben sie wie sie einen Zombie Prozess erzeugen können.
**Tipp**: Ein Prozess wird dann zum Zombie wenn er stirbt, sein Eltern-Prozess dies - aus welchem Grunde auch - noch nicht bemerkt hat.

# ps

Um Prozesse  aufzulisten wird der Befehl **ps** (process status) verwendet.

```
$ ps t
  PID TTY      STAT   TIME COMMAND
 3753 pts/2    Ss     0:00 -bash
 4140 pts/2    T      0:00 bash
 4156 pts/2    Z      0:00 [xclock] <defunct>
 4178 pts/2    R+     0:00 ps t
```

# ps

Weil der Befehl **ps** aus verschiedenen UNIX Varianten kombiniert wurde, ist die Syntax von **ps** ist sehr verwirrend. Die wichtigsten Optionen sind

**a** (all) zeigt alle Prozesse mit Terminal

**l** (long) gibt Zusatzinformationen aus, z. B. die Priorität

**r** (running) zeigt nur laufende Prozesse

**T** (Terminal) zeigt alle Prozesse im aktuellen Terminal

**U *name*** (User) gibt Prozesse von Benutzer name aus.

**x** auch Prozesse ohne Terminal an

# ps

Je nach Unix-System sind die Default-Optionen von **`ps`** unterschiedlich und dadurch unterscheiden sich die Ausgabe entsprechend, was störend bei Shell-skripten ist, die auf verschiedenen System laufen müssen.

Abhilfe schafft da die Optionen immer anzugeben und zu testen.

# ps

**ACHTUNG:** Beachten Sie, dass bei **ps** die Optionen unterschiedlich Bedeutungen haben können, je nachdem ob sie mit einem "-" angegeben werden oder nicht!

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0  2006 ?        00:00:27 init [2]
root         2     1  0  2006 ?        00:00:00 [migration/0]
...

$ ps ef
  PID TTY      STAT   TIME COMMAND
19722 pts/1    S      0:00 bash TERM=Eterm SHELL=/bin/bash
30310 pts/1    R+     0:00  \_ ps ef SHELL=/bin/bash TERM=Eterm
...
```

# top

**ps** zeigt den Zustand der Prozesse statisch an.

Der Befehl **top** updated die Liste auf dem Terminal in periodischen Abständen.
Verschiedene Optionen können behilflich sein um die 'grossen' (viel CPUZeit [P], viel Memory [M]) Prozesse zu finden.

Neben den Prozessen zeigt **top** viele systemrelevante Werte an (Uptime, Load, CPU, Memory).

# top

```
top - 00:04:48 up 197 days,  8:33,  2 users,  load average: 0.75, 0.48, 0.40
Tasks:  64 total,   1 running,  63 sleeping,   0 stopped,   0 zombie
Cpu0  : 37.3%us,  3.7%sy,  0.0%ni, 58.7%id,  0.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  : 20.3%us,  4.0%sy,  0.0%ni, 75.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3993292k total,  3813272k used,   180020k free,   178060k buffers
Swap:  4194296k total,        0k used,  4194296k free,  3096760k cached
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6056 | root | 16 | 0 | 77368 | 68m | 2476 | S | 35 | 1.8 | 598:56.70 | dvg_snmpd.pl |
| 5551 | mysql | 14 | -1 | 446m | 109m | 5344 | S | 21 | 2.8 | 1023:56 | mysqld |
| 177 | root | 15 | 0 | 0 | 0 | 0 | D | 1 | 0.0 | 111:35.21 | pdflush |
| 1 | root | 15 | 0 | 2072 | 640 | 552 | S | 0 | 0.0 | 0:01.85 | init |
| 2 | root | RT | 0 | 0 | 0 | 0 | S | 0 | 0.0 | 0:01.40 | migration/0 |
| 3 | root | 34 | 19 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.04 | ksoftirqd/0 |
| 4 | root | RT | 0 | 0 | 0 | 0 | S | 0 | 0.0 | 0:01.63 | migration/1 |
| 5 | root | 39 | 19 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.61 | ksoftirqd/1 |
| 6 | root | 10 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.01 | events/0 |
| 7 | root | 10 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.01 | events/1 |
| 8 | root | 20 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.00 | khelper |
| 9 | root | 10 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.00 | kthread |
| 83 | root | 10 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 22:55.34 | kblockd/0 |
| 84 | root | 10 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.70 | kblockd/1 |
| 85 | root | 16 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.00 | kacpid |
| 156 | root | 10 | -5 | 0 | 0 | 0 | S | 0 | 0.0 | 0:00.00 | kseriod |

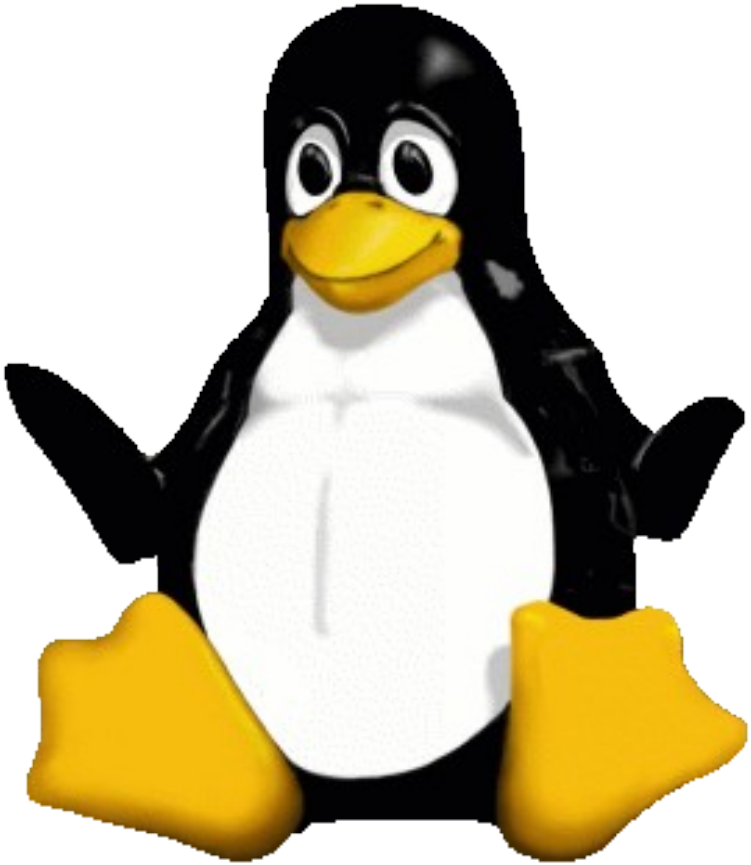
# pstree

Jeder Prozess hat einen Verweis auf seinen Eltern-prozess. Damit kann eine baumförmige  Prozess-Hierarchie aufgezeichnet werden:

```
$ pstree
init-+-/usr/lib/gnome-
     |-Eterm---bash-+-acroread
     |              +-gnome-terminal-+-bash-+-pstree
     |                               +-gnome-pty-helpe
     |                               +-{gnome-terminal}
     +-2*[Eterm---ssh]
     +-acpid
```

Beachten sie auch die "f"-Option von Befehl ps

# Fragen ?



# Übungen

**9.5** Das ps-Kommando erlaubt es Ihnen über die Option `-o`, die ausgegebenen Felder selbst zu bestimmen. Studieren Sie die Manpage `ps(1)` und geben Sie eine ps-Kommandozeile an, mit der Sie die PID, PPID, den Prozesszustand und das Kommando ausgeben können.

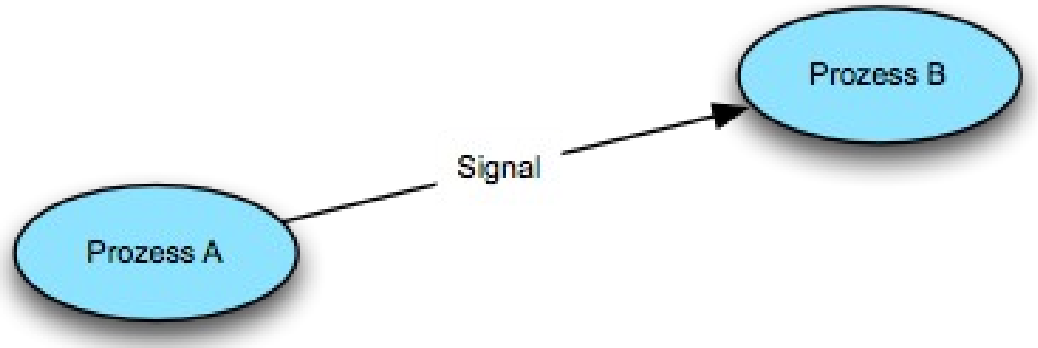
# Prozesse beeinflussen

Prozesse können auf verschiedene Art beeinflusst werden.

- Signale können an Prozesse gesendet werden
- Der Scheduler, der die Prozesse der CPU zuteilt, kann einen Prozess bevorzugen oder ignorieren.

# Signale

Prozesse können Signale von anderen, beliebigen Prozessen empfangen. Die Prozesse müssen dazu in keinem verwandten Zustand zu einander sein.




Da Signale zu jederzeit eintreffen können, sind diese asynchron.

Signale werden manchmal als "Software Interrupts" bezeichnet.

# Signale

Der sendende und empfangende Prozess muss dem gleichen User zugeordnet sein.

Der SuperUser kann Signale an jeden Prozess senden.

Der Prozess kann ein Signal ignorieren oder irgend eine Aktion auslösen.

Gewisse Signale kann ein Prozess nicht ignorieren (SIGSTOP und SIGKILL)

# Signale

**SIGHUP** (1, Hang Up) wird vom Eltern-Prozess bei dessen Ende an alle Kinder gesendet.
SIGHUP wird oft verwendet, um einem Daemon mitzuteilen, dass er seine Konfiguration neu einlesen soll.

**SIGINT** (2, Interrupt) unterbricht den Prozess; entspricht der Tastenkombination [ctrl]-c

**SIGKILL** (9, Kill) bricht den Prozess brutal ab. SIGKILL kann nicht ignoriert werden [ctrl]-\

# Signale

**SIGTERM** (15, Terminate) beendet den Prozess

**SIGCONT** (18, Continue) setzt einen mit SIGSTOP angehaltenen Prozess fort

**SIGSTOP** (19, Stop) hält einen Prozess an; entspricht der Tastenkombination [ctrl]-z.

☞ Alle Signale sind in der ManPage **signal(7)** beschrieben.

# kill(1)

Der Befehl **kill** versendet Signale an ausgewählte Prozesse.

```
kill [signal] PIDs
```

```
# sendet das default signal (SIGTERM) an die 4
# aufgelisteten Prozesse
kill 123 543 2341 3453
```

```
# Killt alle vom Aufrufer kill-baren Prozesse
kill -9 -1
```

```
# Sendet dem Prozess mit der PID 2891 das
# Signal USR1
kill -USR1 2891
```

# killall(1)

Der Befehl **killall** versendet Signale an ausgewählte Prozessnamen

```
killall [signal] name ...
```

# sendet das default signal (SIGTERM) an die httpd Prozesse

```
killall /usr/bin/httpd
```

# killall(1)

Optionen von killall:

`-i,--interactive`

`-l,--list` Listet alle möglichen Signale

`-w,--wait` **`killall`** wartet und kontrolliert periodisch ob alle Prozesse beendet ist

`-u,--user user` sendet das Signal an alle Prozesse vom User `user`

`-r,--regexp` killall bearbeitet den Prozessnamen als Regulären Ausdruck

# pgrep(1)

Der Befehl **kill** hat den Nachteil, dass man zuerst die PID vom Prozess suchen muss.

```
$ ps -e | grep xclock
24637 ?        00:03:19 xclock
$ kill -KILL 24637
```

Der Befehl **pgrep** durchsucht die Prozesstabelle und gibt die PIDs der passenden Prozesse retour:

```
$ pgrep xclock
24637
```

Nutzt man nun die Commansubstitution der Shell so kann man das ganze in einer Zeile erledigen:

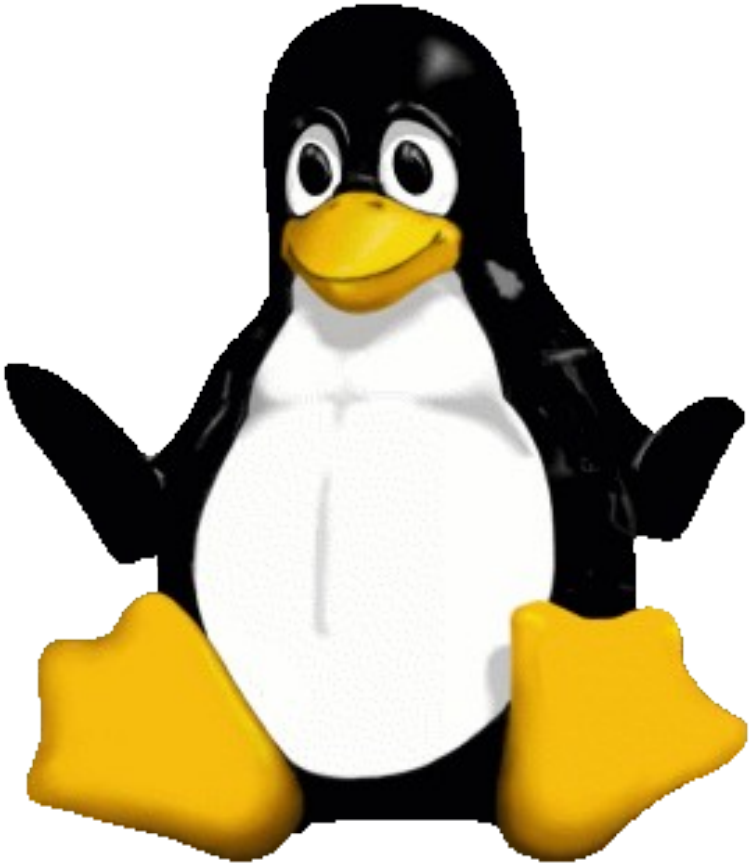
```
$ kill -KILL $(pgrep xclock)
```

# pkill(1)

Der Befehl **pkill** entspricht den Befehl pgrep, ausser, dass er gleich ein Signal - entsprechend den angegebenen Optionen - an die passenden Prozesse sendet:

```
$ pkill -KILL xclock
```

# Fragen ?



# Übungen

**9.6** Welche Signale werden standardmässig ignoriert?
Tipp: signal(7)

# nice

Neben Signale an einen Prozess zu senden kann der Scheduler beeinflusst werden wie er die CPU einen Prozess zuteilt.

Der nice-Wert eines Prozesse beeinflusst den Scheduler wie viel CPU-Zeit er einem Prozess zuordnet.

Der nice Wert kann von +19 (tiefste Priorität) bis -20 (höchste Priorität) betragen. Der default Wert ist 0

# nice(1)

Der Nice-Wert kann beim Starten eines Programms mit dem Befehl **nice** gesetzt werden:

nice [OPTION] [COMMAND [ARG]...]

# startet den Befehl make mit dem nice wert 15
$ nice -n 15 make

☞ Die Shell kann auch einen internen Befehl nice implementieren!

# renice(1)

Wenn der nice-Wert eines Prozess im nach hinein modifiziert werden soll, hilft der Befehl **renice** weiter:

```
renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]
```

User können den nice-Wert ihrer eigenen Prozesse nur erhöhen (was die  Priorität erniedrigt!)
Der SuperUser kann den nice-Wert aller Prozesse beliebig modifizieren!

# Fragen ?