

# Betriebssysteme

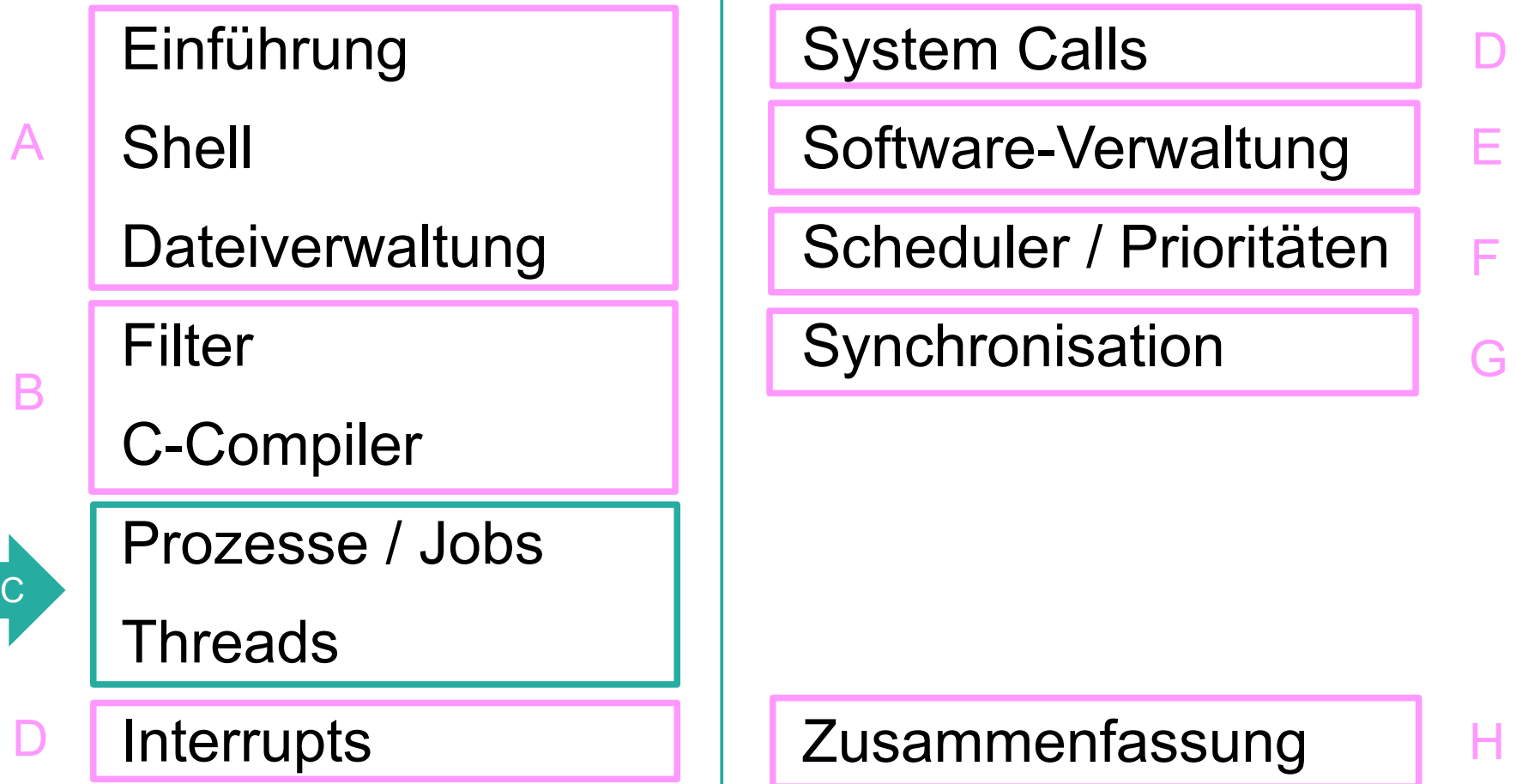
WS 2015/16

**Hans-Georg Eßer**

**Foliensatz C:**

v1.2, 2015/08/20

- Prozesse und Threads



Folien C

# Prozesse (Theorie)

## Vorlesung:

- Theorie / Grundlagen
- Prozesse & Threads im Linux-Kernel

## Praktikum:

- Prozesse auf der Linux-Shell
- Prozesse in C-Programmen
- Threads in C-Programmen

## Single-Tasking / Multitasking:

Wie viele Programme laufen „gleichzeitig“?

- MS-DOS, CP/M: 1 Programm
- Windows, Linux, ....: Viele Programme

## Single-Processing / Multi-Processing:

Hilft der Einsatz mehrerer CPUs?

- Windows 95/98/Me: 1 CPU
- Windows 2000, XP,  
Vista, 7, Linux, Mac OS, ....: Mehrere CPUs

## **MS-DOS:**

- Betriebssystem startet, aktiviert Shell  
COMMAND.COM
- Anwender gibt Befehl ein
- Falls kein interner Befehl:  
Programm laden und aktivieren
- Nach Programmende: Rücksprung zu  
COMMAND.COM

Kein Wechsel zwischen mehreren Programmen

## Prozess:

- Konzept nötig, sobald  $>1$  Programm läuft
- Programm, das der Rechner ausführen soll
- Eigene Daten
- von anderen Prozessen abgeschottet
- Zusätzliche Verwaltungsdaten

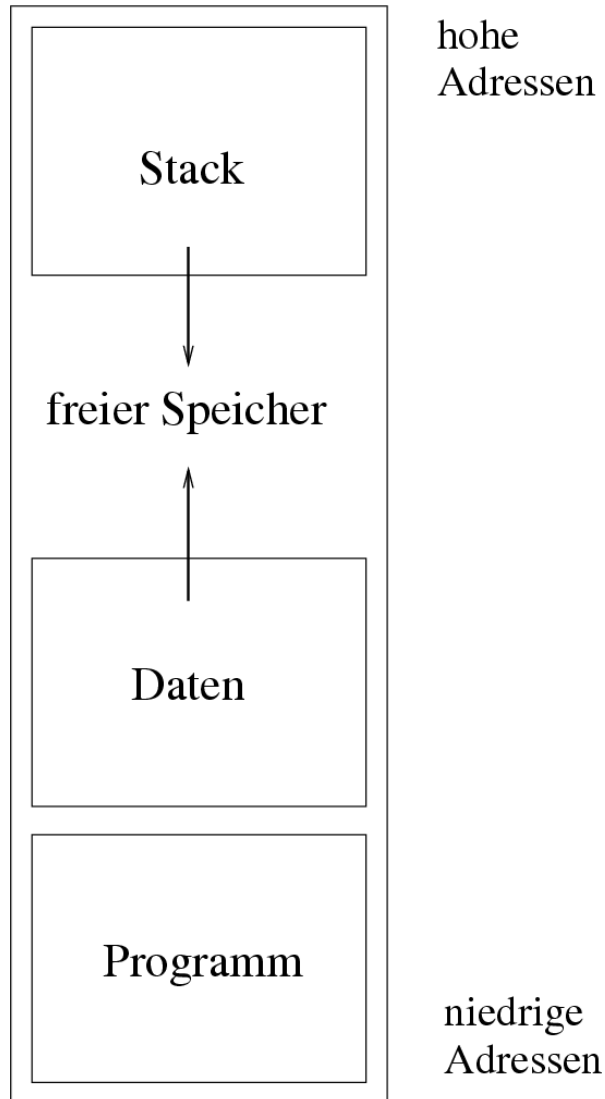
## Prozessliste:

- Informationen über alle Prozesse und ihre Zustände
- Jeder Prozess hat dort einen **Process Control Block (PCB)**:
  - Identifier (PID)
  - Registerwerte inkl. Befehlszähler
  - Speicherbereich des Prozess
  - Liste offener Dateien und Sockets
  - Informationen wie Vater-PID, letzte Aktivität, Gesamtlaufzeit, Priorität, ...



## Prozess im Detail:

- Eigener Adressraum
- Ausführbares Programm
- Aktuelle Daten (Variableninhalte)
- Befehlszähler (Program Counter, PC)
- Stack und Stack-Pointer
- Inhalt der Hardware-Register (Prozess-Kontext)

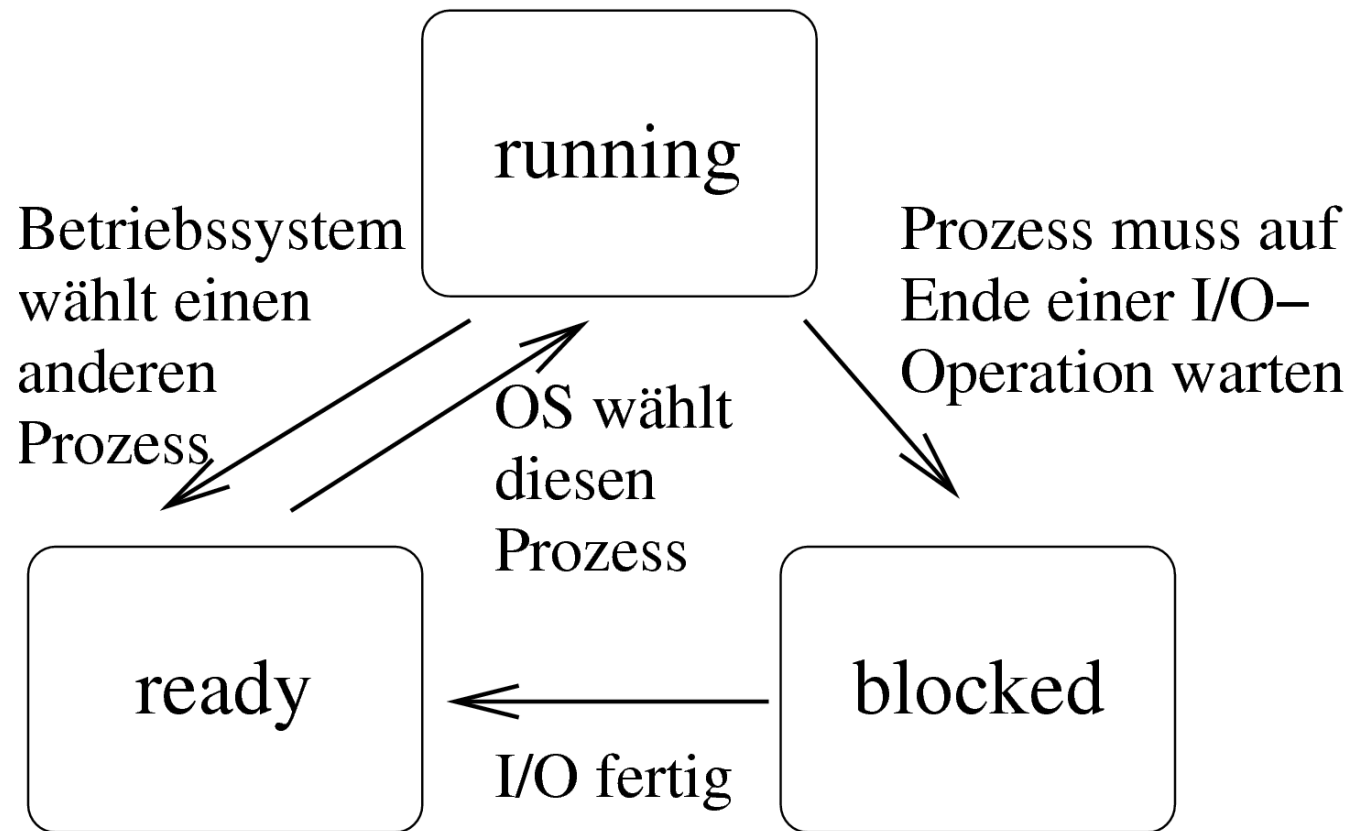


- Daten: dynamisch erzeugt
- Stack: Verwaltung der Funktionsaufrufe
- Details: siehe Kapitel Speicherverwaltung
- Stack und Daten „wachsen aufeinander zu“

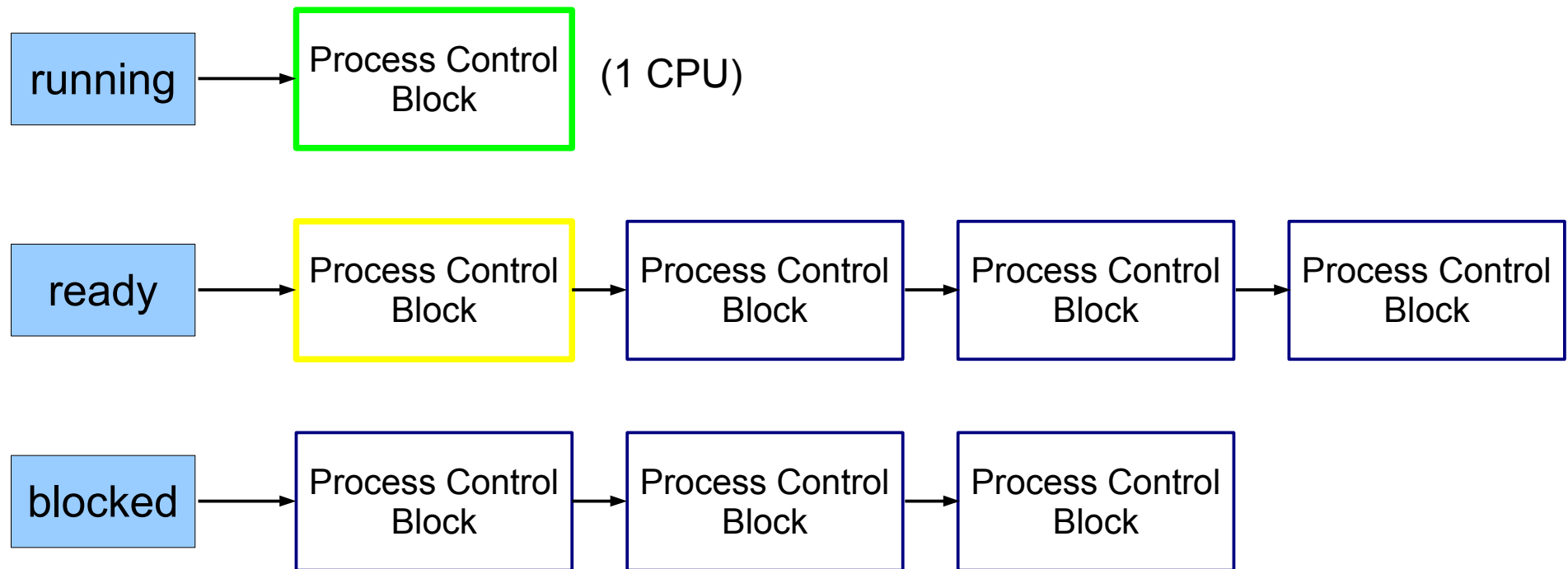
## Zustände

- **laufend / running:** gerade aktiv
- **bereit / ready:** würde gerne laufen
- **blockiert / blocked / waiting:** wartet auf I/O
  
- **suspendiert:** vom Anwender unterbrochen
- **schlafend / sleeping:** wartet auf Signal (IPC)
- **ausgelagert / swapped:** Daten nicht im RAM

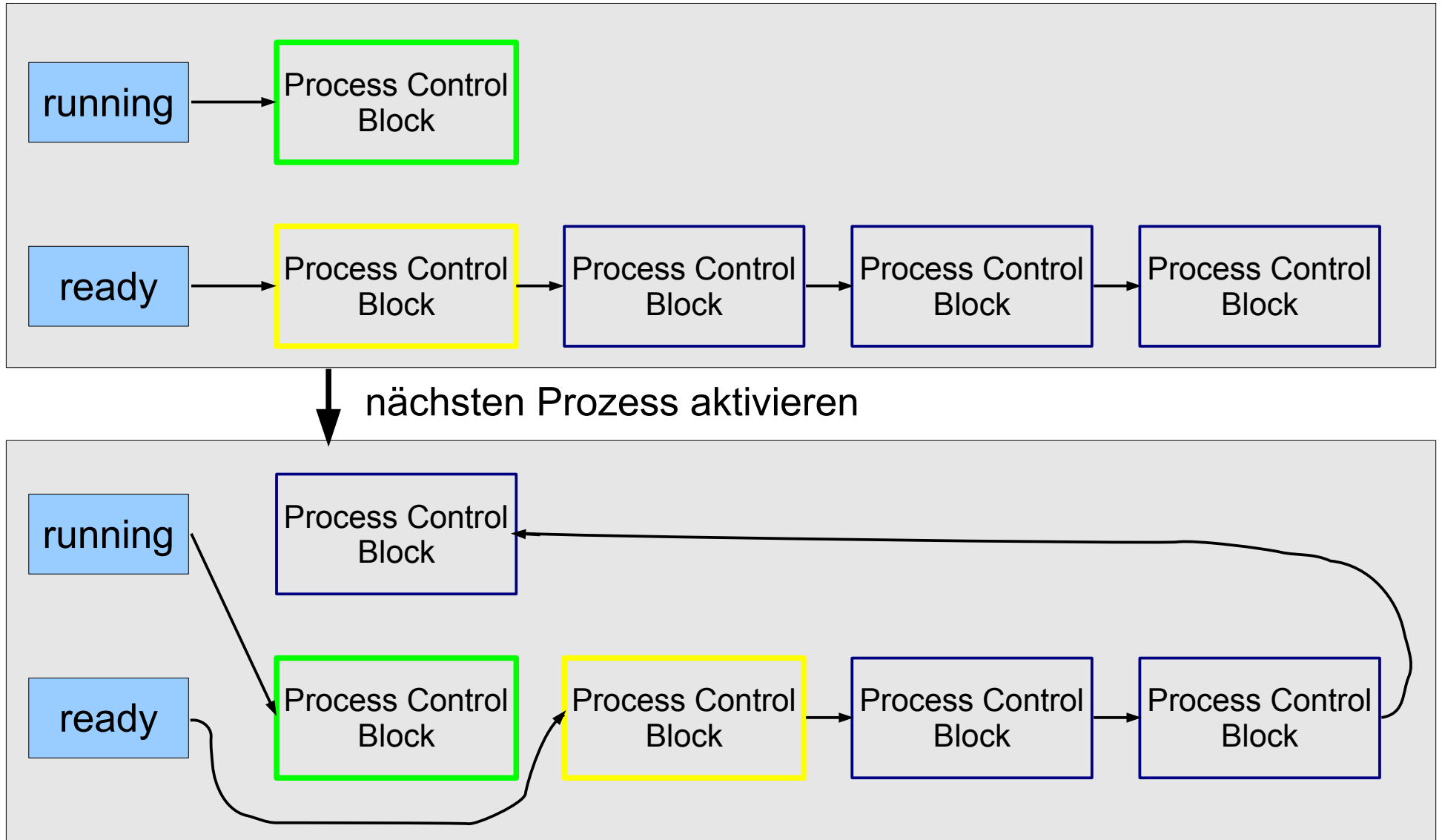
## Zustandsübergänge



## Prozesslisten



# Prozesse (6)



## Hierarchien

- Prozesse erzeugen einander
- Erzeuger heißt Vaterprozess (parent process), der andere Kindprozess (child process)
- Kinder sind selbständig (also: eigener Adressraum, etc.)
- Nach Prozess-Ende: Rückgabewert an Vaterprozess

# Prozesse (Praxis, Anwenderperspektive)



- In der Shell gestartete Anwendungen laufen standardmäßig im **Vordergrund** – d. h.,
  - die Shell ist blockiert, solange das Programm läuft,
  - und es nutzt das aktuelle Terminal (-Fenster) für Ein- und Ausgabe
- Alternativ kann ein Programm im **Hintergrund** laufen:
  - die Shell kann dann sofort weiter genutzt werden (weitere Kommando eingeben),
  - keine Eingabe möglich, aber Ausgabe (auch ins aktuelle Terminal; besser umleiten)

- Typische Vordergrund-Programme
  - Kommandos, die eine Anfrage sofort beantworten
  - Text-Editoren
  - Compiler
- Typische Hintergrund-Programme
  - manuell gestartete Server (Dienste)
  - unter X Window: grafische Anwendungen (die kein Terminal brauchen, sondern ein eigenes Fenster öffnen)

- Programm im Vordergrund starten:  
einfach den Namen eingeben  
Bsp.: `ls -l`
- Programm im Hintergrund starten:  
kaufmännisches Und (&, ampersand) anhängen  
Bsp.: `/usr/sbin/apache2 &`
- Wechsel von Vordergrund in Hintergrund:
  - Programm mit [Strg-Z] unterbrechen
  - Programm mit `bg` in den Hintergrund schicken
- Wechsel von Hinter- in Vordergrund: `fg`

- Programme, die aus einer laufenden Shell heraus gestartet wurden, heißen **Jobs** dieser Shell
- Anzeige mit: `jobs`

```
[esser@macbookpro:~]$ jobs
[esser@macbookpro:~]$ nedit &
[1] 77787
[esser@macbookpro:~]$ vi /tmp/test.txt
^Z
[2]+  Stopped                  vi /tmp/test.txt
[esser@macbookpro:~]$ find / > /tmp/ergebnisse.txt &
[3] 77792
[esser@macbookpro:~]$ jobs
[1]  Running                   nedit &
[2]+  Stopped                  vi /tmp/test.txt
[3]-  Running                   find / > /tmp/ergebnisse.txt &
[esser@macbookpro:~]$
```

# Job-Verwaltung (2)

Zustand des Jobs:

- running: aktiv / bereit
- stopped: mit ^Z oder kill -STOP angehalten
- terminated: beendet, wird nur 1x angezeigt

```
[esser@macbookpro:~]$ jobs
[1]   Running          nedit &
[2]+  Stopped          vi /tmp/test.txt
[3]-  Running          find / > ...&
```

Kommandos

1,2,3, ...: Job-Nummer

- + : „current job“ = letzter Job, der
  - im Vordergrund gestartet und dann unterbrochen
  - oder im Hintergrund gestartet wurde
- viele Kommandos (fg, bg, ...) ohne Parameter beziehen sich auf den current job
- : „previous job“ = vorletzter Job mit obiger Eigenschaft

- Jobs gezielt ansprechen: %n (mit n = Job-Nummer)

```
[esser@macbookpro:~]$ jobs
[1]    Running                    nedit /tmp/1 &
[2]    Running                    nedit /tmp/2 &
[3]    Running                    nedit /tmp/3 &
[4]-   Running                    nedit /tmp/4 &
[5]+   Running                    nedit /tmp/5 &
[esser@macbookpro:~]$ kill %3
[esser@macbookpro:~]$ jobs
[1]    Running                    nedit /tmp/1 &
[2]    Running                    nedit /tmp/2 &
[3]    Terminated                nedit /tmp/3
[4]-   Running                    nedit /tmp/4 &
[5]+   Running                    nedit /tmp/5 &
[esser@macbookpro:~]$ jobs
[1]    Running                    nedit /tmp/1 &
[2]    Running                    nedit /tmp/2 &
[4]-   Running                    nedit /tmp/4 &
[5]+   Running                    nedit /tmp/5 &
```

## Kommandos zur Job-Verwaltung

- `bg %n`: in den Hintergrund bringen
- `fg %n`: in den Vordergrund bringen
- `kill %n`: beenden
- `kill -SIGNALNAME %n`: Signal schicken, siehe nächste Folie
- `disown %n`: Verbindung mit der Shell lösen;  
`disown -a`: für alle Jobs
- `wait %n`: Warten, bis Job beendet ist

## Signale (mit Signalnummer)

- `TERM`, 15: terminieren, beenden (mit „Aufräumen“); Standardsignal
- `KILL`, 9: sofort abbrechen (ohne Aufräumen)
- `STOP`, 19: unterbrechen (entspricht `^Z`)
- `CONT`, 18: continue, fortsetzen; hebt `STOP` auf
- `HUP`, 1: hang-up, bei vielen Server-Programmen: Konfiguration neu einlesen (traditionell: Verbindung zum Terminal unterbrochen)
- Liste aller Signale: `kill -l`



```
$ kill -l
```

```

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       16) SIGSTKFLT
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        31) SIGSYS        34) SIGRTMIN
35) SIGRTMIN+1    36) SIGRTMIN+2    37) SIGRTMIN+3    38) SIGRTMIN+4
39) SIGRTMIN+5    40) SIGRTMIN+6    41) SIGRTMIN+7    42) SIGRTMIN+8
43) SIGRTMIN+9    44) SIGRTMIN+10  45) SIGRTMIN+11  46)
SIGRTMIN+12
47) SIGRTMIN+13  48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-
14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-
10
55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX

```

- Die Bezeichnung **Job** bezieht sich immer auf die aktuelle Shell-Sitzung
- Jobs, die Sie in verschiedenen Shells starten, haben nichts miteinander zu tun
- Allgemeinerer Begriff: **Prozess**
- Tool für die Prozessanzeige: `ps`
- Die (Gesamt-) Prozessliste (`ps auxw`) enthält alle Prozesse auf dem Linux-System

- `ps` (ohne Optionen) zeigt alle Prozesse an, die zur aktuellen Shell-Sitzung gehören – das sind dieselben wie in der Ausgabe von `jobs`:

```
[esser@quadamd:~]$ jobs
[1]+  Angehalten   vi /tmp/test4
```

```
[esser@quadamd:~]$ ps
  PID TTY          TIME CMD
 27967 pts/0        00:00:00 bash
 28160 pts/0        00:00:00 vi
 28168 pts/0        00:00:00 ps
```

- über Optionen (ohne „–“) lässt sich die Ausgabe von `ps` anpassen, z. B. `ps auxw`:
  - `a`: alle Prozesse (die ein Terminal haben)
  - `u`: „user oriented format“
  - `x`: auch Prozesse ohne Terminal
  - `w`: „wide“: Befehlszeilen nicht abschneiden

## Prozess-IDs in der Job-Liste

```
esser@sony:Folien> jobs
[1]- Running          xpdf -remote sk bs02.pdf &
[2]+ Running          nedit kap02/index.tex &
```

```
esser@sony:Folien> jobs -l
[1]- 8103 Running      xpdf -remote sk bs02.pdf &
[2]+ 20568 Running    nedit kap02/index.tex &
```

```
esser@sony:Folien> ps w|grep 8103|grep -v grep
 8103 pts/15 S          5:27 xpdf -remote sk bs02.pdf
```

```
[esser@quadamd:~]$ ps auw
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1122  0.0  0.0   1872    580 tty4      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty4
root      1127  0.0  0.0   1872    580 tty5      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty5
root      1150  0.0  0.0   1872    576 tty2      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty2
root      1151  0.0  0.0   1872    576 tty3      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty3
root      1153  0.0  0.0   1872    580 tty6      Ss+  Apr17   0:00 /sbin/getty -8 38400 tty6
root      1776  0.0  0.0   5248   2156 tty1      Ss   Apr17   0:00 /bin/login --
esser     1941  0.0  0.1   9272   4816 tty1      S    Apr17   0:01 -bash
esser     2956  0.0  0.0   1912    540 tty1      S+   Apr17   0:00 /bin/sh /usr/bin/startx
esser     2973  0.0  0.0   3128    708 tty1      S+   Apr17   0:00 xinit /home/esser/.xinitrc -- /etc/X11/xini
root      2974  0.0  1.8  85616  75712 tty8      Ss+  Apr17  14:56 /usr/bin/X -nolisten tcp :0 -auth /tmp/ser
esser     2977  0.0  0.0   1912    564 tty1      S    Apr17   0:00 /bin/sh /home/esser/.xinitrc
esser     3037  0.0  0.0   3456    560 tty1      S    Apr17   0:00 dbus-launch --sh-syntax --exit-with-session
esser     3045  0.0  0.0   1700     64 tty1      S    Apr17   0:00 /usr/lib/kde4/libexec/start_kdeinit +kcmn
esser     3174  0.0  0.0   1832    244 tty1      S    Apr17   0:00 kwrapper4 kmsserver
esser     3454  0.0  0.0   6552   2040 pts/2     Ss   Apr17   0:00 /bin/bash
root      3775  0.0  0.0   8560   2112 pts/2     S    Apr17   0:00 sudo su
root      3776  0.0  0.0   8316   1764 pts/2     S    Apr17   0:00 su
root      3784  0.0  0.0   6656   2172 pts/2     S+   Apr17   0:00 bash
esser     10674  1.4  7.0  443588  289940 pts/6     Sl   Apr30  131:17 /usr/lib/opera/opera
esser     10694  0.0  0.1   21764   7552 pts/6     S    Apr30   0:08 /usr/lib/opera//operapluginwrapper 101 110
esser     10695  0.0  0.0    3040    548 pts/6     S    Apr30   0:02 /usr/lib/opera//operaplugincleaner 10674
esser     10699  1.3  0.0     0         0 pts/6     Z    Apr30  119:12 [gtk-gnash] <defunct>
esser     12198  0.0  0.0   6552   1828 pts/4     Ss   Apr17   0:00 /bin/bash
root      12583  0.0  0.0   8560   2116 pts/4     S    Apr17   0:00 sudo su
root      13077  0.0  0.0   8316   1768 pts/4     S    Apr17   0:00 su
root      13097  0.0  0.0   6612   2052 pts/4     S+   Apr17   0:00 bash
esser     13653  0.0  0.0   6768   2140 pts/3     Ss   Apr17   0:00 /bin/bash
esser     18905  0.0  0.1   9128   4712 pts/6     Ss+  Apr22   0:02 /bin/bash
esser     27587  0.0  0.0   5256   2144 pts/3     S+   19:40   0:00 ssh backup
esser     28613  0.0  0.1  11744   7264 pts/5     Ss+  22:45   0:00 -bash
esser     29091  0.0  0.1  12156   7704 pts/0     Ss   22:58   0:00 -bash
esser     29307  0.0  0.0   5856   1172 pts/0     R+   23:42   0:00 ps auw
```

- Spalten in der Ausgabe von `ps auw`:
  - `USER`: Benutzer, dem der Prozess gehört
  - `PID`: Prozess-ID
  - `%CPU`: CPU-Nutzung in Prozent (Verhältnis Rechenzeit / Lebenszeit)
  - `%MEM`: RSS / RAM-Größe in Prozent
  - `VSZ`: Größe des virtuellen Speichers (in KByte)
  - `RSS`: Resident Set Size, aktuell genutzter Speicher (KByte)
  - `TTY`: Terminal
  - `STAT`: Prozess-Status
  - `START`: Startzeit des Prozesses (ggf. Datum)
  - `TIME`: bisherige Gesamtlaufzeit
  - `COMMAND`: Kommando (Aufruf)

- Signale an beliebige Prozesse schicken
  - wie vorher: Kommando `kill`
  - aber: nicht `kill %n` ( $n$ =Job-ID), sondern `kill p` ( $p$  = PID)
  - auch hier Angabe eines Signals möglich
- `killall Name`: alle Prozesse beenden, deren ausführbares Programm *Name* heißt
- mit `killall` auch (wie bei `kill`) andere Signale an alle Prozesse mit passendem Namen schicken





- Wenn Sie sich in der Konsole abmelden (`exit`) oder unter X Window ein Terminalfenster schließen, erhalten alle in der Shell laufenden Jobs das HUP-Signal (Hang-up).
- Die Standardreaktion auf HUP ist: beenden
- Abmelden / Fenster schließen beendet also alle darin gestarteten Programme
- Auswege:
  - Programme mit `nohup` starten oder
  - Prozess mit `disown` von der Shell lösen

- nohup hat zwei Funktionen:
  - der gestartete Prozess ignoriert HUP-Signale
  - Ausgaben des Prozesses (auf die Standardausgabe) erscheinen nicht im Terminal, sondern werden in die Datei nohup.out geschrieben

```
[esser@macbookpro:~]$ nedit /tmp/1 &  
[1] 79142  
[esser@macbookpro:~]$ nohup nedit /tmp/2 &  
[2] 79144  
appending output to nohup.out
```

- Prozesse nach CPU-Auslastung sortiert anzeigen: top
- Anzeige wird regelmäßig aktualisiert

```
top - 00:07:30 up 19 days, 6:15, 7 users, load average: 0.00, 0.02, 0.05
Tasks: 194 total, 2 running, 191 sleeping, 0 stopped, 1 zombie
Cpu(s): 1.2%us, 0.7%sy, 0.0%ni, 98.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 4120180k total, 2353392k used, 1766788k free, 560756k buffers
Swap: 4191936k total, 0k used, 4191936k free, 566868k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3177	esser	20	0	302m	45m	20m	S	2	1.1	260:37.29	knotify4
10674	esser	20	0	433m	283m	27m	S	2	7.0	131:47.89	opera
3432	esser	20	0	893m	155m	33m	S	1	3.9	253:02.73	seamonkey-bin
1093	messageb	20	0	4524	2664	840	R	0	0.1	16:55.65	dbus-daemon
1576	root	20	0	12072	7272	3408	S	0	0.2	8:50.93	python
2076	root	20	0	5560	972	692	S	0	0.0	5:07.52	udisks-daemon
3238	esser	20	0	367m	64m	35m	S	0	1.6	1:13.69	krunner
29348	esser	20	0	2632	1180	852	R	0	0.0	0:00.03	top
1	root	20	0	3028	1892	1236	S	0	0.0	0:02.06	init
2	root	20	0	0	0	0	S	0	0.0	0:00.73	kthreadd
3	root	20	0	0	0	0	S	0	0.0	1:36.46	ksoftirqd/0
6	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
17	root	0	-20	0	0	0	S	0	0.0	0:00.00	cpuset
18	root	0	-20	0	0	0	S	0	0.0	0:00.00	khelper
19	root	0	-20	0	0	0	S	0	0.0	0:00.00	netns
21	root	20	0	0	0	0	S	0	0.0	0:02.28	sync_supers
22	root	20	0	0	0	0	S	0	0.0	0:00.05	bdi-default
23	root	0	-20	0	0	0	S	0	0.0	0:00.00	kintegrityd
24	root	0	-20	0	0	0	S	0	0.0	0:00.00	kblockd
25	root	0	-20	0	0	0	S	0	0.0	0:00.00	kacpid
26	root	0	-20	0	0	0	S	0	0.0	0:00.00	kacpi_notify
27	root	0	-20	0	0	0	S	0	0.0	0:00.00	kacpi_hotplug

- Sortierung in `top` anpassbar (Sortierspalte ändern mit `<` und `>`)
- Über der Prozessliste: Informationen zur Gesamtauslastung des Systems
- umschaltbar auf Anzeige/CPU bzw. /Kern: 1

```
top - 00:14:22 up 19 days, 6:22, 7 users, load average: 0.05, 0.03, 0.05
Tasks: 194 total, 2 running, 191 sleeping, 0 stopped, 1 zombie
Cpu0  : 0.7%us, 0.3%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1  : 1.7%us, 1.0%sy, 0.0%ni, 97.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2  : 0.0%us, 0.3%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3  : 3.6%us, 0.7%sy, 0.0%ni, 95.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem:   4120180k total, 2353400k used, 1766780k free, 560948k buffers
Swap:  4191936k total,          0k used, 4191936k free, 566868k cached
```

- Weitere Systeminformationen:
  - `free` (freien Speicher anzeigen)

```
[esser@quadamd:~]$ free
              total          used          free      shared    buffers     cached
Mem:          4120180      2347264      1772916           0       561488      566896
-/+ buffers/cache:      1218880      2901300
Swap:          4191936              0       4191936
```

- `uptime` (wie lange läuft das System schon?)

```
[esser@quadamd:~]$ uptime
00:34:08 up 19 days, 6:42, 6 users, load average: 0.06, 0.07, 0.05
```

- Jeder Linux-Prozess hat eine **Priorität**. Diese bestimmt, welchen Anteil an Rechenzeit der Prozess erhält.
- Priorität ist ein Wert zwischen -20 und 19.
- Konvention: hohe Priorität = kleiner Wert (also: -20 = maximale Prior., 19 = minimale Prior.)
- unter Linux/Unix auch als **nice value** („Nettigkeit“) bezeichnet: 19 = extrem nett, -20 = gar nicht nett
- Bei Programmstart Priorität mit `nice` setzen

- `nice` mit Priorität (Option) und auszuführendem Kommando (folgende Argumente) aufrufen, z. B.

```
[esser@quadamd:~]$ nice -5 program &
```

```
[esser@quadamd:~]$ ps -eo user,pid,ni,cmd
```

```
USER          PID  NI  CMD
...
root          28299  0  [kworker/2:0]
root          28300  0  [kworker/0:1]
esser        28301  5  program
esser        28303  0  ps -eo user,pid,ni,cmd
```

- negative Nice-Werte kann nur Administrator *root* setzen:

```
[esser@quadamd:~]$ nice --10 vi
```

```
nice: kann Priorität nicht setzen: Keine Berechtigung
```

- Alternative Syntax für bessere Lesbarkeit:  
`nice -n Wert` (statt `nice -Wert`)
- vor allem für negative Werte intuitiver:  
`nice -n -10` (statt `nice --10`)

```
[esser@quadamd:~]$ su
Passwort:
root@quadamd:~# nice -n -10 program &
[1] 28373
root@quadamd:~# ps -eo user,pid,ni,cmd
USER          PID    NI  CMD
[... ]
root          28311    0  su
root          28319    0  bash
root          28373  -10  program
root          28375    0  ps -eo user,pid,ni,cmd
```



- Genauer: Nice-Wert in `nice`-Aufruf ist relativ zum „aktuellen Nice-Level“ (Standard: 0)
- angegebener Wert wird zum Nice-Wert addiert:

```
[esser@quadamd:~]$ nice
0
[esser@quadamd:~]$ nice -n 5 bash
[esser@quadamd:~]$ nice
5
[esser@quadamd:~]$ nice -n 10 bash
[esser@quadamd:~]$ nice
15
[esser@quadamd:~]$ _
```

- Nice-Wert für laufendes Programm ändern: `renice`
- Wert  $<0$  setzen darf nur *root*
- in alten Linux-Versionen galt auch: aktuellen Wert verringern darf nur *root*)

```
[esser@quadamd:~]$ program &
[5] 28937
[esser@quadamd:~]$ ps -eo user,pid,ni,cmd
USER      PID  NI  CMD
esser    28937   0  program
[esser@quadamd:~]$ renice 5 28937
28937: Alte Priorität: 0, neue Priorität: 5
[esser@quadamd:~]$ ps -eo user,pid,ni,cmd
USER      PID  NI  CMD
esser    28937   5  program
[esser@quadamd:~]$ renice 0 28937
28937: Alte Priorität: 5, neue Priorität: 0
[esser@quadamd:~]$ renice -10 28937
renice: 28937: setpriority: Keine Berechtigung
```

# Threads (Theorie)

## Was ist ein Thread?

- Aktivitätsstrang in einem Prozess
- einer von mehreren
- Gemeinsamer Zugriff auf Daten des Prozess
- aber: Stack, Befehlszähler, Stack Pointer, Hardware-Register separat pro Thread
- Prozess-Scheduler verwaltet Threads – oder nicht (Kernel- oder User-level-Threads)

## Warum Threads?

- Multi-Prozessor-System: Mehrere Threads echt gleichzeitig aktiv
- Ist ein Thread durch I/O blockiert, arbeiten die anderen weiter
- Besteht Programm logisch aus parallelen Abläufen, ist die Programmierung mit Threads einfacher

## Zwei unterschiedliche Aktivitätsstränge: Komplexe Berechnung mit Benutzeranfragen

Ohne Threads:

```
while (1) {  
    rechne_ein_bisschen ();  
    if benutzereingabe (x) {  
        bearbeite_eingabe (x);  
    }  
}
```

## Komplexe Berechnung mit Benutzeranfragen

Mit Threads:

T1:

```
while (1) {  
    rechne_alles ();  
}
```

T2:

```
while(1) {  
    if benutzereingabe (x) {  
        bearbeite_eingabe (x);  
    }  
}
```

## Server-Prozess, der viele Anfragen bearbeitet

- Prozess öffnet Port
- Für jede eingehende Verbindung: Neuen Thread erzeugen, der diese Anfrage bearbeitet
- Nach Verbindungsabbruch Thread beenden
- Vorteil: Keine Prozess-Erzeugung (Betriebssystem!) nötig



## Ein Prozess, neun Threads:

```
[esser:~]$ ps -eLf | grep mysql
```

UID	PID	PPID	LWP	C	NLWP	STIME	TTY	TIME	CMD
root	27833	1	27833	0	1	Jan04	?	00:00:00	/bin/sh /usr/bin/mysqld_safe
<b>mysql</b>	<b>27870</b>	<b>27833</b>	<b>27870</b>	<b>0</b>	<b>9</b>	<b>Jan04</b>	<b>?</b>	<b>00:00:00</b>	<b>/usr/sbin/mysqld --basedir=/usr</b>
mysql	27870	27833	27872	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27873	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27874	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27875	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27876	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27877	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27878	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr
mysql	27870	27833	27879	0	9	Jan04	?	00:00:00	/usr/sbin/mysqld --basedir=/usr

```
[esser:~]$
```

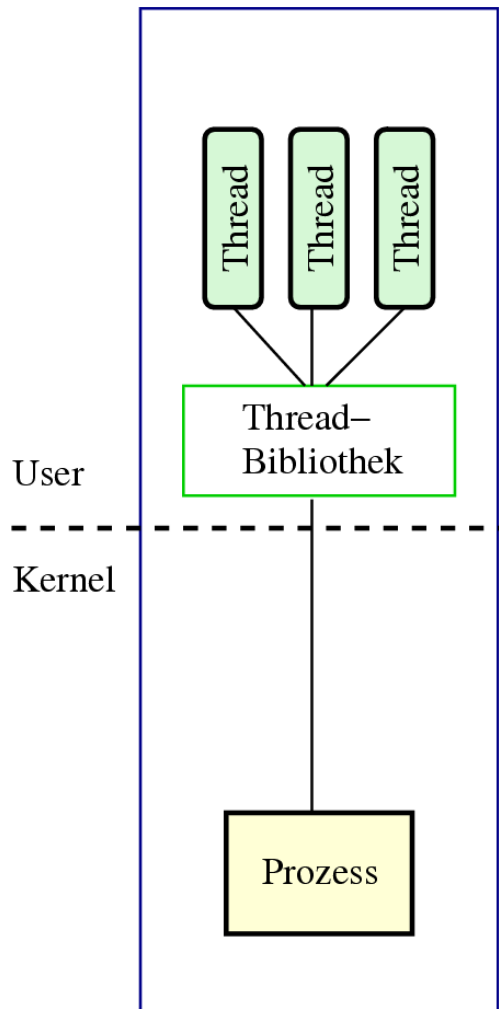
PID: Process ID

PPID: Parent Process ID

LWP: Light Weight Process ID (Thread-ID)

NLWP: Number of Light Weight Processes

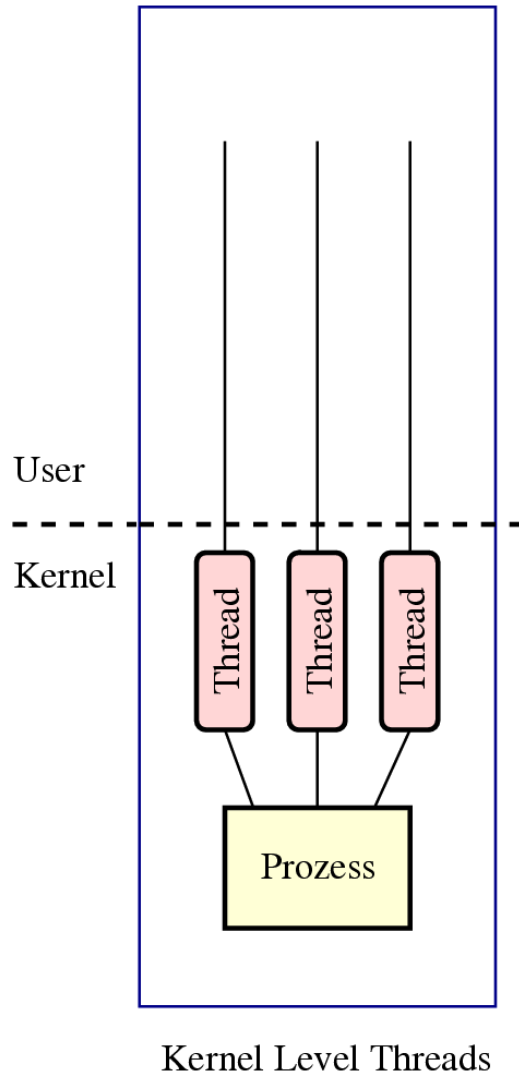
# User Level Threads



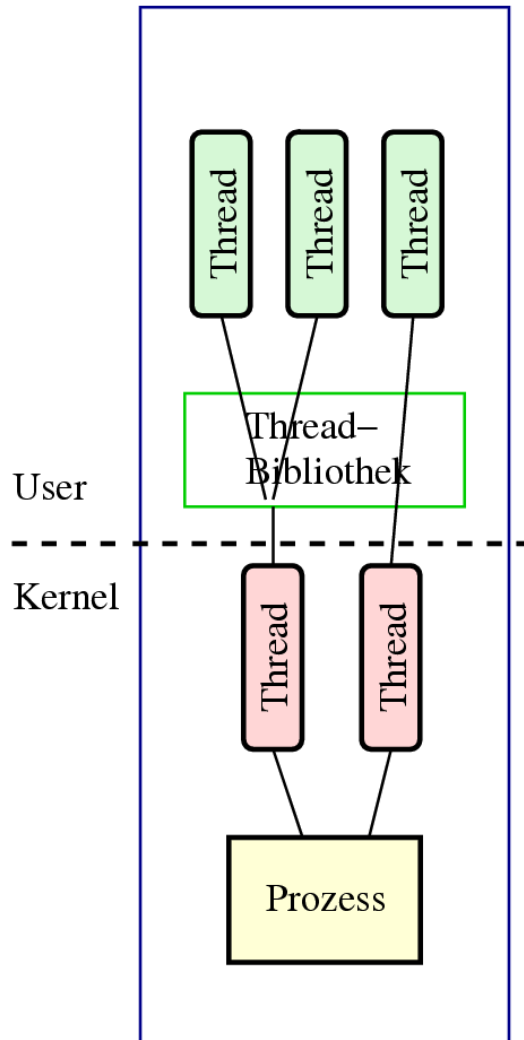
User Level Threads

- BS kennt kein Thread-Konzept, verwaltet nur Prozesse
- Programm bindet Thread-Bibliothek ein, zuständig für:
  - Erzeugen, Zerstören
  - Scheduling
- Wenn ein Thread wegen I/O wartet, dann der ganze Prozess
- Ansonsten sehr effizient

# Kernel Level Threads



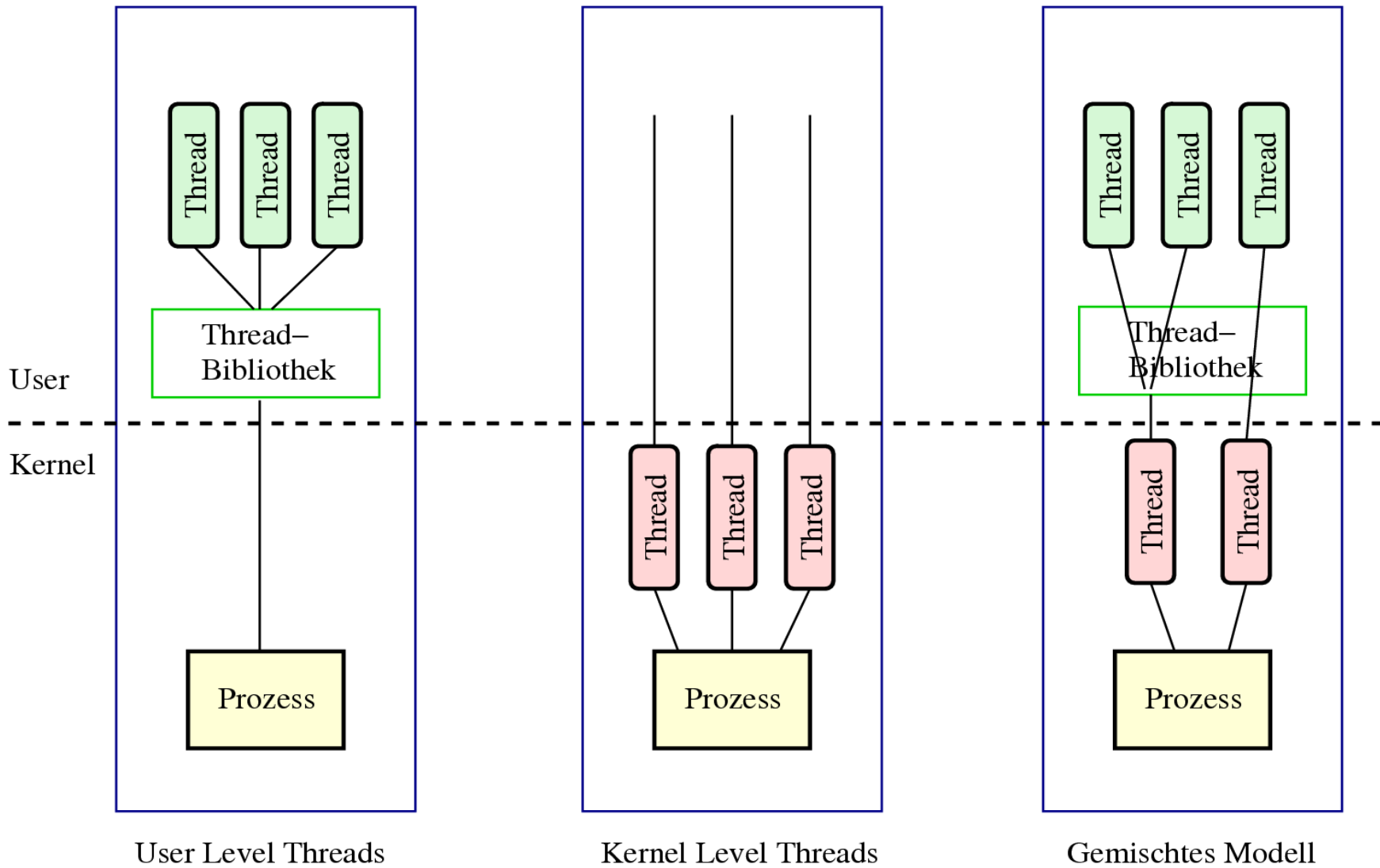
- BS kennt Threads
- BS verwaltet die Threads:
  - Erzeugen, Zerstören
  - Scheduling
- I/O eines Threads blockiert nicht die übrigen
- Aufwendig: Context Switch zwischen Threads ähnlich komplex wie bei Prozessen



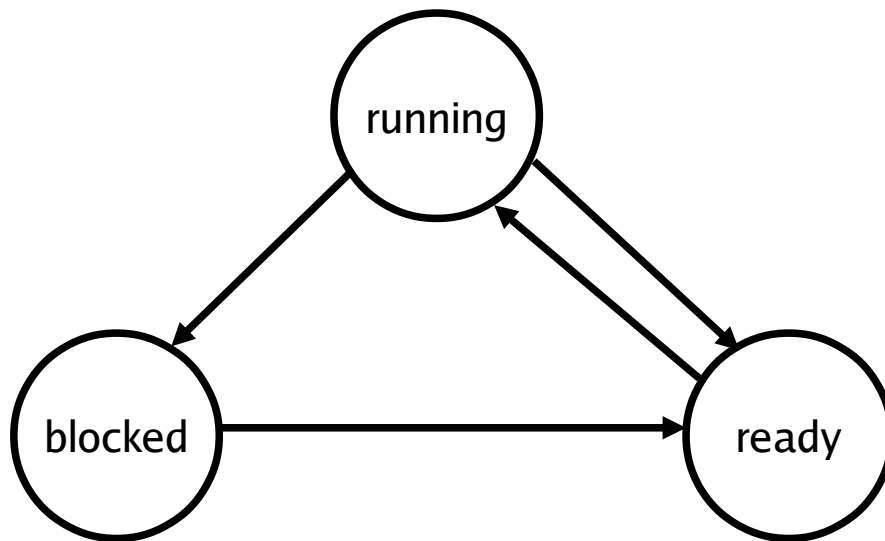
Gemischtes Modell

- Beide Ansätze kombinieren
- KL-Threads + UL-Threads
- Thread-Bibliothek verteilt UL-Threads auf die KL-Threads
- z.B. I/O-Anteile auf einem KL-Thread
- Vorteile beider Welten:
  - I/O blockiert nur einen KL-Thread
  - Wechsel zwischen UL-Threads ist effizient
- SMP: Mehrere CPUs benutzen

# Thread-Typen, Übersicht



- Prozess-Zustände suspended, sleeping, swapped etc. nicht auf Threads übertragbar (warum nicht?)
- Darum nur drei Thread-Zustände



# Prozesse und Threads (Praxis, Entwicklersicht)

## Neuer Prozess: fork ( )

```
main() {
    int pid = fork();    /* Sohnprozess erzeugen */
    if (pid == 0) {
        printf("Ich bin der Sohn, meine PID ist %d.\n", getpid() );
    }
    else {
        printf("Ich bin der Vater, mein Sohn hat die PID %d.\n", pid);
    }
}
```



## Anderes Programm starten: `fork` + `exec`

```
main() {
    int pid=fork();    /* Sohnprozess erzeugen */
    if (pid == 0) {
        /* Sohn startet externes Programm */
        execl( "/usr/bin/gedit", "/etc/fstab", (char *) 0 );
    }
    else {
        printf("Es sollte jetzt ein Editor starten...\n");
    }
}
```

## Andere Betriebssysteme oft nur: „spawn“

```
main() {
    WinExec("notepad.exe", SW_NORMAL);    /* Sohn erzeugen */
}
```

## Warten auf Sohn-Prozess: `wait ()`

```

#include <unistd.h>                /* sleep()                */

main()
{
    int pid=fork();              /* Sohnprozess erzeugen  */
    if (pid == 0)
    {
        sleep(2);                 /* 2 sek. schlafen legen */
        printf("Ich bin der Sohn, meine PID ist %d\n", getpid() );
    }
    else
    {
        printf("Ich bin der Vater, mein Sohn hat die PID %d\n",
pid);
        wait();                 /* auf Sohn warten */
    }
}

```

Wirklich mehrere Prozesse:

Nach `fork ( )` zwei Prozesse in der Prozessliste

```
> pstree | grep simple
... -bash---simplefork---simplefork
```

```
> ps w | grep simple
25684 pts/16 S+      0:00 ./simplefork
25685 pts/16 S+      0:00 ./simplefork
```

## Linux: pthread-Bibliothek (POSIX Threads)

	Thread	Prozess
Erzeugen	<code>pthread_create()</code>	<code>fork()</code>
Auf Ende warten	<code>pthread_join()</code>	<code>wait()</code>

- Bibliothek einbinden:  
`#include <pthread.h>`
- Kompilieren:  
`gcc -lpthread -o prog prog.c`

- Neuer Thread:  
`pthread_create()` erhält als Argument eine Funktion, die im neuen Thread läuft.
- Auf Thread-Ende warten:  
`pthread_join()` wartet auf einen *bestimmten* Thread.

## 1. Thread-Funktion definieren:

```
void *thread_funktion(void *arg) {
    ...
    return ...;
}
```

## 2. Thread erzeugen:

```
pthread_t thread;
```

```
if ( pthread_create( &thread, NULL,
    thread_funktion, NULL) ) {
    printf("Fehler bei Thread-Erzeugung.\n");
    abort();
}
```

```

#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_function1(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread 1 sagt Hi!\n");
        sleep(1);
    }
    return NULL;
}

void *thread_function2(void *arg) {
    int i;
    for ( i=0; i<10; i++ ) {
        printf("Thread 2 sagt Hallo!\n");
        sleep(1);
    }
    return NULL;
}

int main(void) {

    pthread_t mythread1;
    pthread_t mythread2;

    if ( pthread_create( &mythread1, NULL,
        thread_function1, NULL) ) {
        printf("Fehler bei Thread-Erzeugung.");
        abort();
    }

```

```

sleep(5);

    if ( pthread_create( &mythread2, NULL,
        thread_function2, NULL) ) {
        printf("Fehler bei Thread-Erzeugung .");
        abort();
    }

    sleep(5);

    printf("bin noch hier...\n");

    if ( pthread_join ( mythread1, NULL ) ) {
        printf("Fehler beim Join.");
        abort();
    }

    printf("Thread 1 ist weg\n");

    if ( pthread_join ( mythread2, NULL ) ) {
        printf("Fehler beim Join.");
        abort();
    }

    printf("Thread 2 ist weg\n");

    exit(0);
}

```

## Keine „Vater-“ oder „Kind-Threads“

- POSIX-Threads kennen keine Verwandtschaft wie Prozesse (Vater- und Sohnprozess)
- Zum Warten auf einen Thread ist Thread-Variable nötig: `pthread_join (thread, ..)`



## Prozess mit mehreren Threads:

- Nur ein Eintrag in normaler Prozessliste
- Status: „l“, multi-threaded
- Über `ps -eLf` Thread-Informationen
  - NLWP: Number of light weight processes
  - LWP: Thread ID

```
> ps auxw | grep thread
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    12022  0.0  0.0  17976   436 pts/15    Sl+  22:58   0:00 ./thread
```

```
> ps -eLf | grep thread
UID      PID  PPID  LWP  C  NLWP  STIME  TTY      TIME  CMD
esser    12166  4031 12166 0    3  23:01 pts/15    00:00:00 ./thread1
esser    12166  4031 12167 0    3  23:01 pts/15    00:00:00 ./thread1
esser    12166  4031 12177 0    3  23:01 pts/15    00:00:00 ./thread1
```

## Unterschiedliche Semantik:

- Prozess erzeugen mit `fork ( )`
  - erzeugt zwei (fast) identische Prozesse,
  - beide Prozesse setzen Ausführung an gleicher Stelle fort (nach Rückkehr aus `fork`-Aufruf)
- Thread erzeugen mit `pthread_create (..., funktion, ...)`
  - erzeugt neuen Thread, der in die angeg. Funktion springt
  - erzeugender Prozess setzt Ausführung hinter `pthread_create`-Aufruf fort

Kernel unterscheidet nicht zwischen Prozessen und Threads.

- Doppelt verkettete, ringförmige Liste
- Jeder Eintrag vom Typ `struct task_struct`
- Typ definiert in [include/linux/sched.h](#)
- Enthält alle Informationen, die Kernel benötigt
- `task_struct`-Definition 132 Zeilen lang!
- Maximale PID: 32767 (short int)

Auszug aus *include/linux/sched.h*:

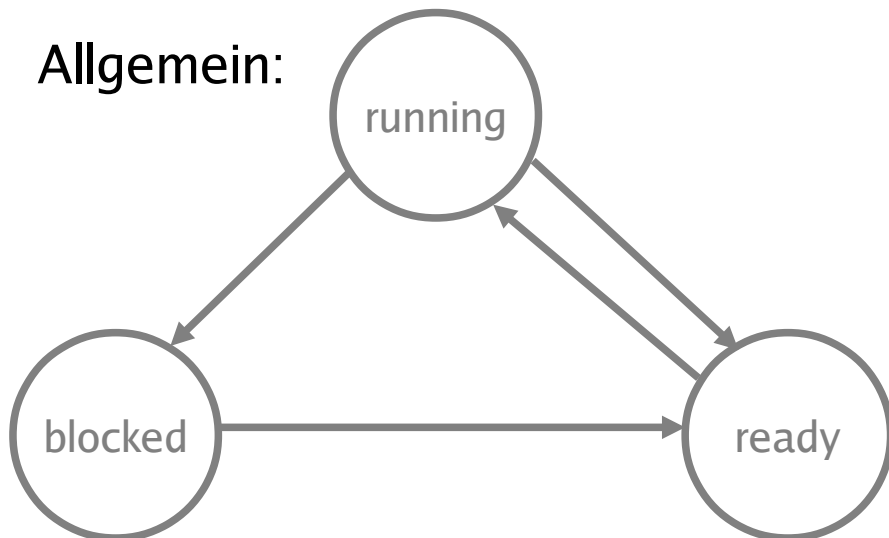
```

#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE   1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_STOPPED         4
#define TASK_TRACED          8
/* in tsk->exit_state */
#define EXIT_ZOMBIE          16
#define EXIT_DEAD            32
/* in tsk->state again */
#define TASK_NONINTERACTIVE  64
#define TASK_DEAD            128

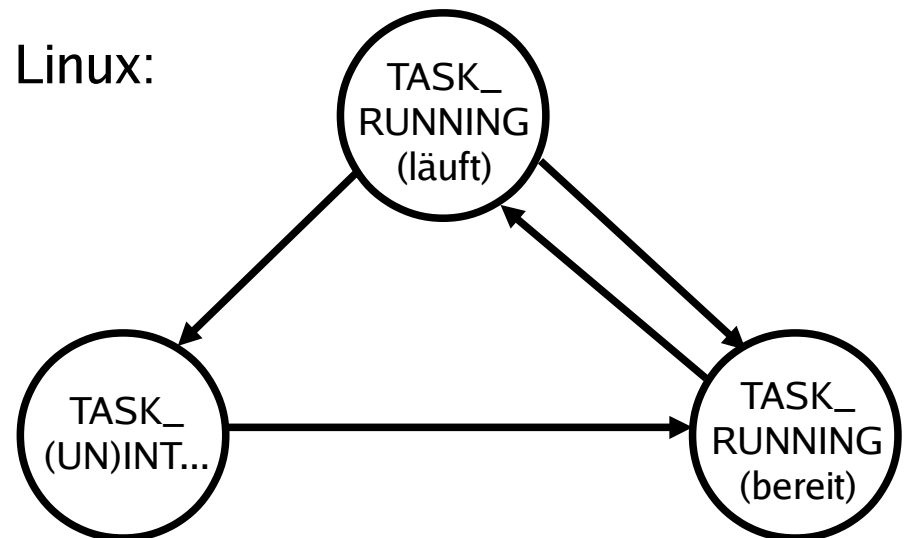
```

- TASK\_RUNNING: ready oder running
- TASK\_INTERRUPTIBLE: entspricht blocked
- TASK\_UNINTERRUPTIBLE: auch blocked
- TASK\_STOPPED: angehalten (z. B. von einem Debugger)
- TASK\_ZOMBIE: beendet, aber Vater hat Rückgabewert nicht gelesen

Allgemein:

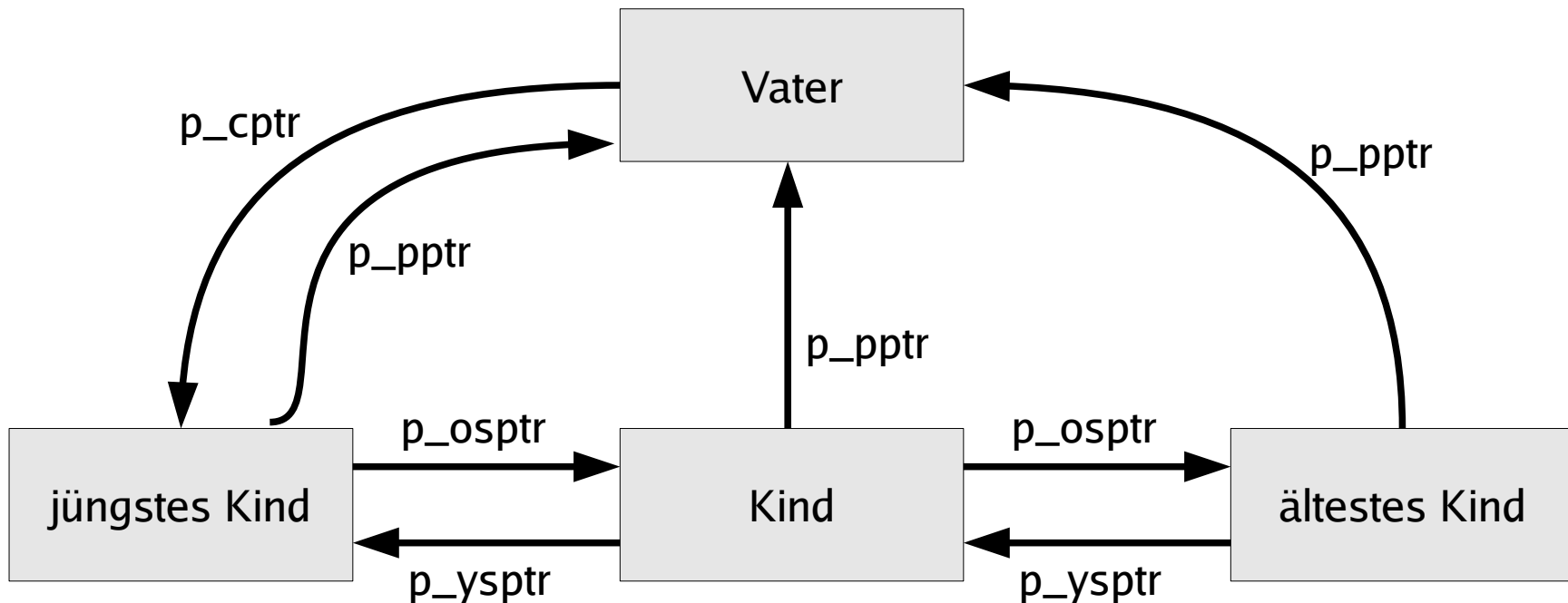


Linux:



## Verwandtschaftsverhältnisse (alte Linux-Version)

```
struct task_struct {
    [...]
    struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
```



## Verwandtschaftsverhältnisse (neue Linux-Version)

```
struct task_struct {
    [...]
    struct task_struct *parent; /* parent process */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
}
```

Zugriff auf alle Kinder:

```
list_for_each(list, &current->children) {
    task = list_entry(list, struct task_struct, sibling);
    /* task zeigt jetzt auf eines der Kinder */
}
```

Vom aktuellen Pfad durch den Prozessbaum bis zu init:

```
for (task = current; task != &init_task; task = task->parent) {
    ...
}
```

## Prozessgruppen und Sessions

```

struct task_struct {
    [...]
    struct task_struct *group_leader;
        /* threadgroup leader */
    [...]
    /* signal handlers */
    struct signal_struct *signal;

```

```

struct signal_struct {
    /* job control IDs */
    pid_t pgrp;          Process Group ID
    pid_t tty_old_pgrp;
    pid_t session;     Session ID
    /* boolean value for session
       group leader */
    int leader;

```

- Jeder Prozess Mitglied einer Prozessgruppe
- Process Group ID (PGID) – `ps j`
- `current->signal->pgrp`

## Prozessgruppen

- Signale an alle Mitglieder einer Prozessgruppe:  
`killpg(pgrp, sig);`
- Warten auf Kinder aus der eigenen Prozessgruppe:  
`waitpid(0, &status, ...);`
- oder einer speziellen Prozessgruppe:  
`waitpid(-pgrp, &status, ...);`



## Sessions

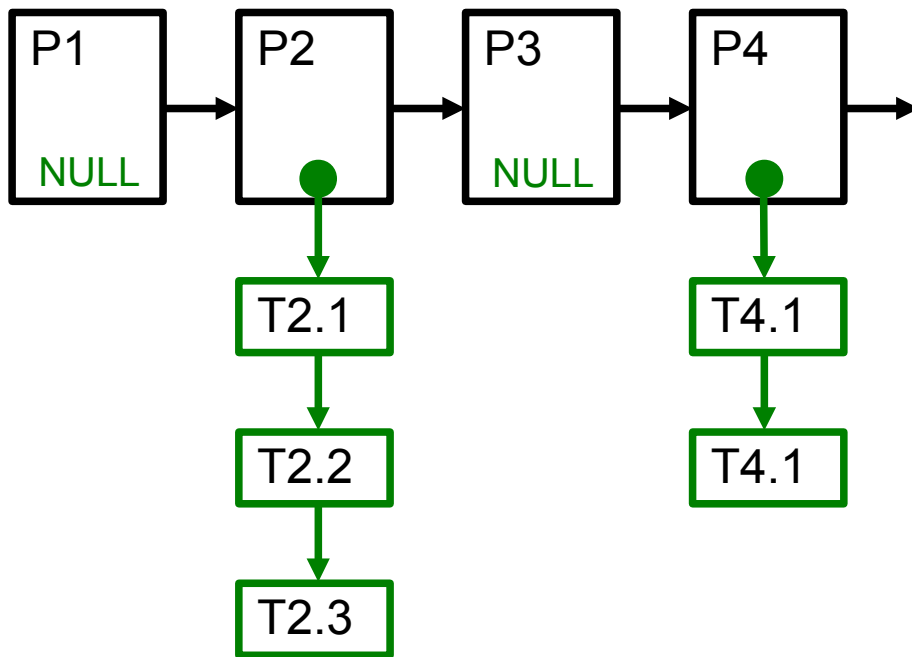
- Meist beim Starten einer Login-Shell neu erzeugt
- Alle Prozesse, die aus dieser Shell gestartet werden, gehören zur Session
- Gemeinsames „kontrollierendes TTY“

```
> ps j
  PPID   PID   PGID   SID  TTY      TPGID  STAT   UID    TIME  COMMAND
19287   7628   7628   19287 pts/8    19287  S      500    0:00  /bin/sh /usr/bin/mozilla -mail
   7628   7637   7628   19287 pts/8    19287  Sl     500    20:50 /opt/moz/lib/mozilla-bin -mail
   9634  10095  10095   10095 tty1     10114  Ss     500    0:00  -bash
10095  10114 10114 10095 tty1     10114  S+     500    0:00  /bin/sh /usr/X11R6/bin/startx
10095  10115  10114 10095 tty1     10114  S+     500    0:00  tee /home/esser/.X.err
10114  10135  10114 10095 tty1     10114  S+     500    0:00  xinit /home/esser/.xinitrc
10135  10151 10151 10095 tty1     10114  S      500    0:00  /bin/sh /usr/X11R6/bin/kde
10151  10238  10151 10095 tty1     10114  S      500    0:00  kwrapper ksmsserver
10258  10270  10270  10270 pts/2    10270  Ss+    500    0:00  bash
10276  10278  10278  10278 pts/4    10278  Ss+    500    0:00  bash
10260  10284  10284  10284 pts/5    10284  Ss+    500    0:00  bash
10275  10292  10292  10292 pts/6    10989  Ss     500    0:00  bash
10259  10263 10263 10263 pts/1    10263  Ss+    500    0:00  bash
10263  28869 28869 10263 pts/1    10263  S      500    0:16  konqueror /media/usbdisk/dcim
10263  28872 28872 10263 pts/1    10263  S      500    0:13  konqueror /home/esser
29201  29203  29203  29203 pts/7    29203  Ss+    500    0:00  bash
   4822   4823 4823 4823 pts/14   4823  Ss+    500    0:00  -bash
   4823  31118 31118 4823 pts/14   4823  S      500    0:00  nedit kernel/sched.c
   4823  31297 31297 4823 pts/14   4823  S      500    0:00  nedit kernel/fork.c
23115  32703  32703  23115 pts/13   32703  R+     500    0:00  ps j
```

- Linux verwendet für Threads und Prozesse die gleichen Verwaltungsstrukturen (task list)
- Thread: Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt, z. B.
  - virtueller Speicher
  - offene Dateien
- Jeder Thread hat `task_struct` und sieht für den Kernel wie ein normaler Prozess aus

- Fundamental anders als z. B. Windows und Solaris

Modell 1:  
reine Prozesslisten



Modell 2 (Linux):  
Prozesse + Threads gemischt

