

```
Sep 19 14:20:18 amd64 sshd[20494]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61557
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[29278]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 /usr/sbin/cron[30103]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[6516]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:46:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:48:41 amd64 sshd[6609]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd[6694]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:35 amd64 sshd[9077]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64242
Sep 20 15:27:35 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:37:11 amd64 sshd[10102]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:37:11 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:38:10 amd64 sshd[10140]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63546
Sep 21 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 21 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 02:00:01 amd64 /usr/sbin/cron[17878]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 21 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:43:26 amd64 sshd[31088]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63397
Sep 21 17:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:53:39 amd64 sshd[31269]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64391
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 19:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 01:00:01 amd64 /usr/sbin/cron[1999]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 02:00:01 amd64 /usr/sbin/cron[1999]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:21 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 01:00:01 amd64 /usr/sbin/cron[24739]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 23 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 /usr/sbin/cron[25555]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:05 amd64 sshd[6554]: Accepted public key for esser from ::ffff:192.168.1.1 port 59771 ssh2
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[6606]: Accepted rsa for esser from ::ffff:87.234.201.207 port 6093
Sep 24 01:00:01 amd64 /usr/sbin/cron[1256]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[1353]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[20998]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[23197]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_midi_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[29399]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[662]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1484]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 25 02:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[8889]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[8921]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[9372]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11586]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:33 amd64 sshd[11630]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778
```

2. Prozesse und Threads

Betriebssysteme I
Hochschule München
Hans-Georg Eßer
SS 2011

Foliensatz 2
Prozesse und Threads

Prozesse & Threads: Gliederung

Vorlesung:

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

Praktikum:

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

Einleitung (1)

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,
Linux, Mac OS X, ...: Mehrere CPUs

Einleitung (2)

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

Einleitung (3)

Prozess:

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

Einleitung (4)

Prozessliste:

- Informationen über alle Prozesse und ihre Zustände

- Jeder Prozess hat dort einen

Process Control Block (PCB):

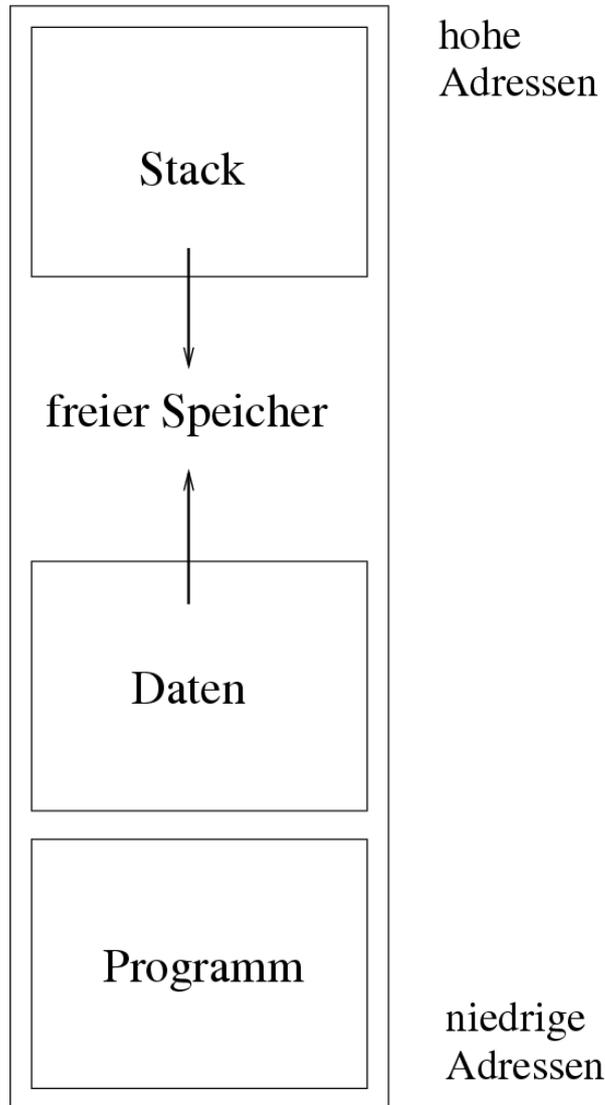
- Identifier (PID)
- Registerwerte inkl. Befehlszähler
- Speicherbereich des Prozess
- Liste offener Dateien und Sockets
- Informationen wie Vater-PID, letzte Aktivität, Gesamtlaufzeit, Priorität, ...

Prozesse (1)

Prozess im Detail:

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

Prozesse (2)



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

Prozesse (3)

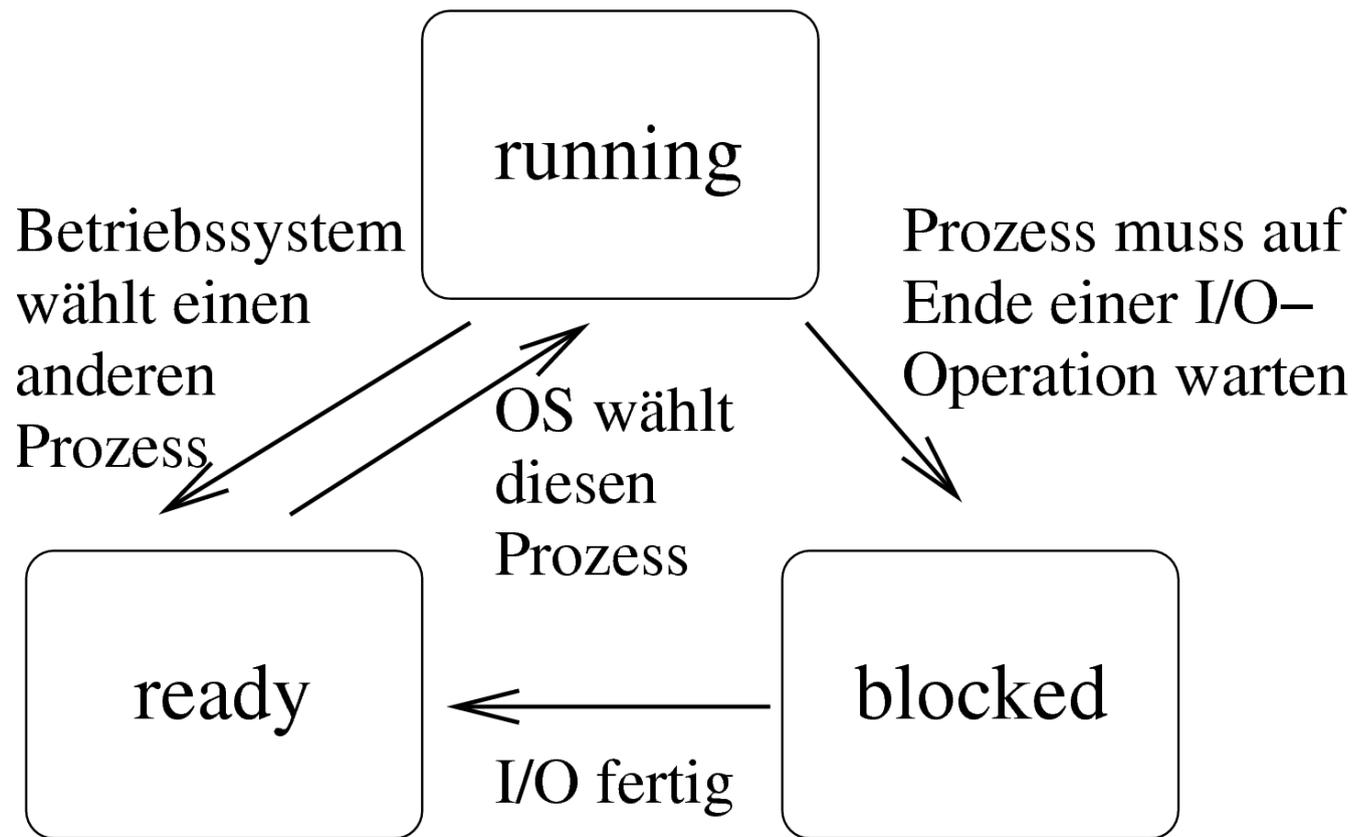
Zustände

- **laufend / running:** gerade aktiv
- **bereit / ready:** würde gerne laufen
- **blockiert / blocked:** wartet auf I/O

- **suspendiert:** vom Anwender unterbrochen
- **schlafend / sleeping:** wartet auf Signal (IPC)
- **ausgelagert / swapped:** Daten nicht im RAM

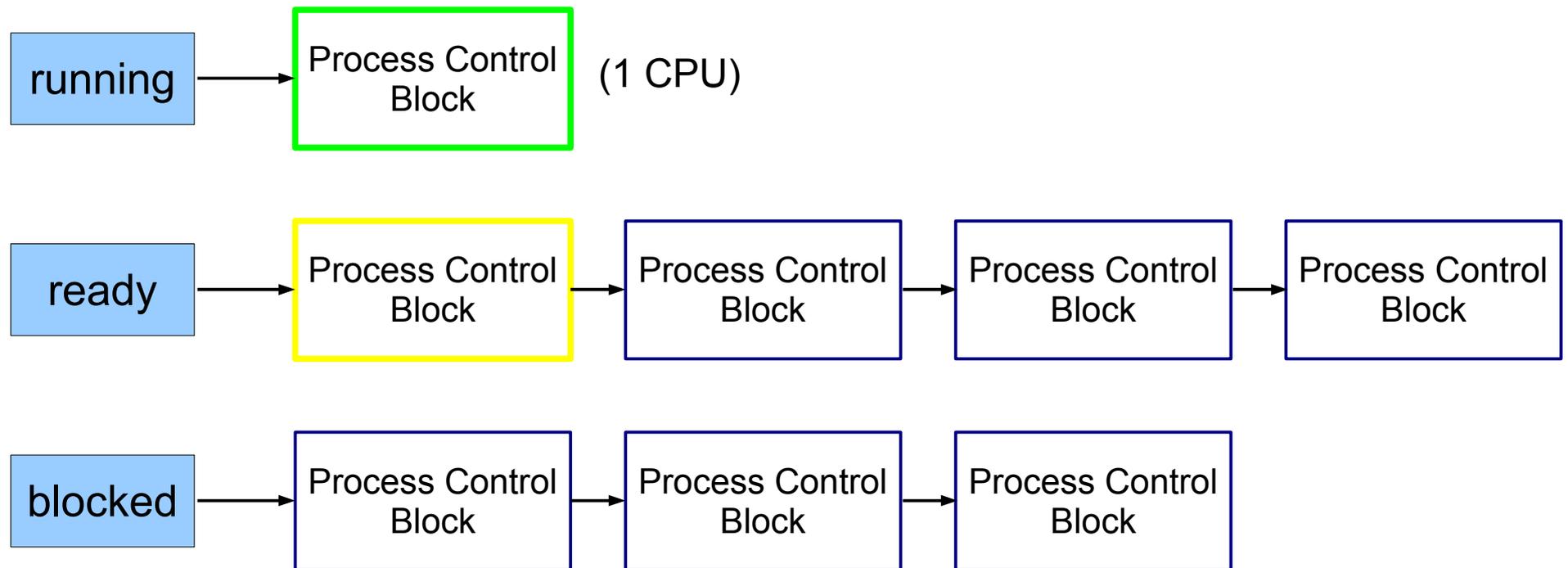
Prozesse (4)

Zustandsübergänge

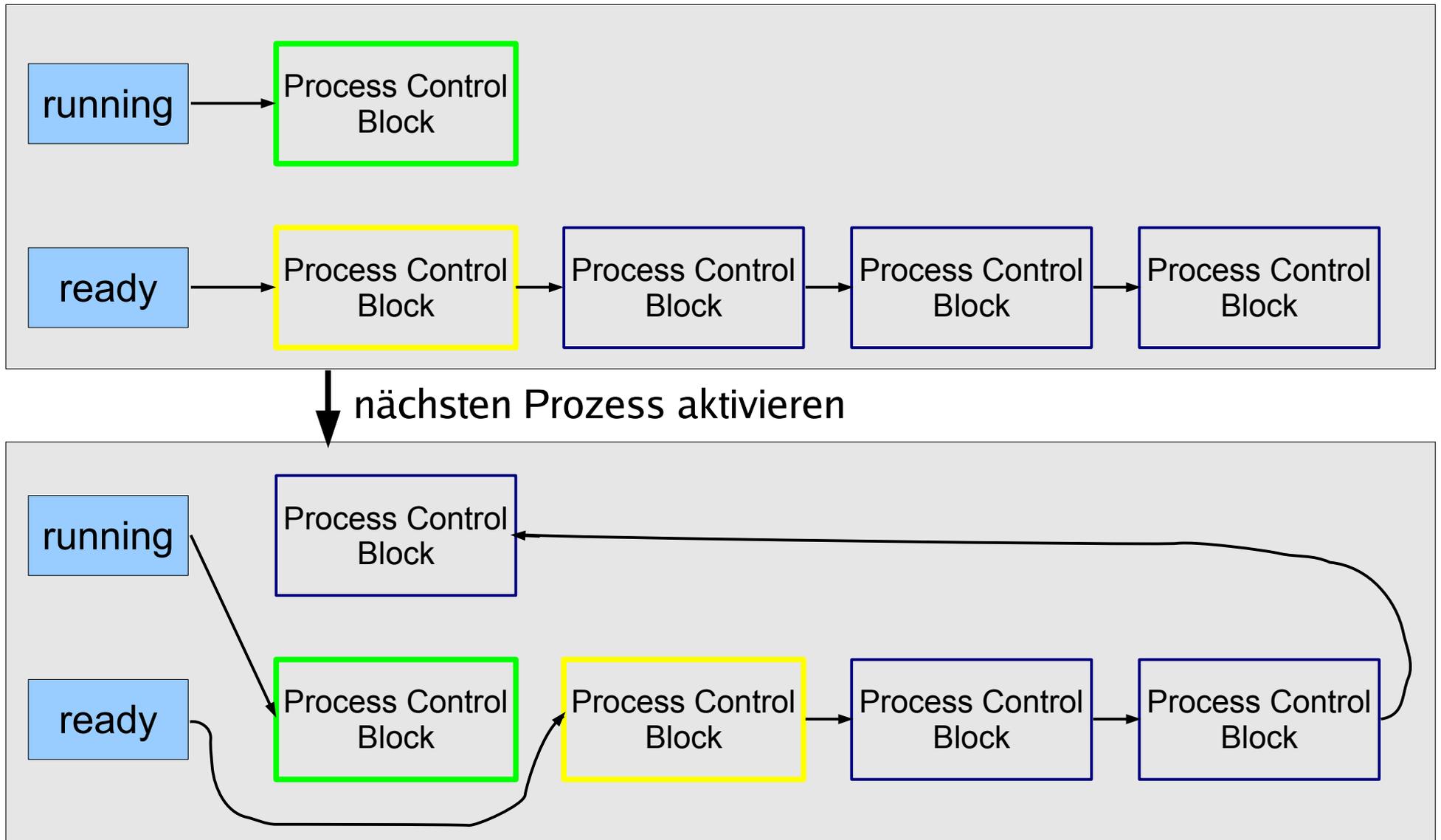


Prozesse (5)

Prozesslisten



Prozesse (6)



Prozesse (7)

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

Praxis: Anwender (1)

```
esser@sony:Folien> emacs test.txt &
```

```
[3] 24469
```

```
esser@sony:Folien> _
```

```
[...]
```

```
[3]+ Done
```

```
emacs test.txt
```

Praxis: Anwender (2)

```
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
```

Praxis: Anwender (3)

```
> ps auxw
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	720	92	?	S	Jun24	0:01	init [5]
root	2	0.0	0.0	0	0	?	SN	Jun24	1:09	[ksoftirqd/0]
root	3	0.0	0.0	0	0	?	S<	Jun24	0:11	[events/0]
root	4	0.0	0.0	0	0	?	S<	Jun24	0:00	[khelper]
root	5	0.0	0.0	0	0	?	S<	Jun24	0:00	[kthread]
root	7	0.0	0.0	0	0	?	S<	Jun24	0:02	[kblockd/0]
root	8	0.0	0.0	0	0	?	S<	Jun24	0:00	[kacpid]
root	128	0.0	0.0	0	0	?	S<	Jun24	0:00	[aio/0]
[.....]										
esser	5733	0.2	12.2	82420	63428	?	S	Jul24	4:05	/usr/bin/opera
root	2670	0.3	0.0	1368	300	?	Ss	08:24	2:39	zmd /usr/lib/zmd
esser	8037	0.0	0.6	6452	3384	pts/13	S+	11:23	0:05	ssh -X amd64

Praxis: Anwender (5)

- Programm unterbrechen: `Strg-Z`
- Fortsetzen im Vordergrund: `fg`
- Fortsetzen im Hintergrund: `bg`
- Signale an Prozess schicken: `kill`
 - ◊ unterbrechen (STOP), fortsetzen (CONT)
 - ◊ beenden (TERM), abschießen (KILL)
- Verbindung zu Vater lösen: `disown`

Praxis: Anwender (6)

```
> 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
```

Threads (1)

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)

Threads (2)

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

Threads (3): Beispiele

Zwei unterschiedliche Aktivitätsstränge: Komplexe Berechnung mit Benutzeranfragen

Ohne Threads:

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

Threads (4): Beispiele

Komplexe Berechnung mit Benutzeranfragen

Mit Threads:

T1:

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

T2:

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

Threads (5): Beispiele

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

Threads (6): Beispiel MySQL

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
mysql       27870   27833   27870  0   9 Jan04  ?           00:00:00 /usr/sbin/mysqld --basedir=/usr
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)

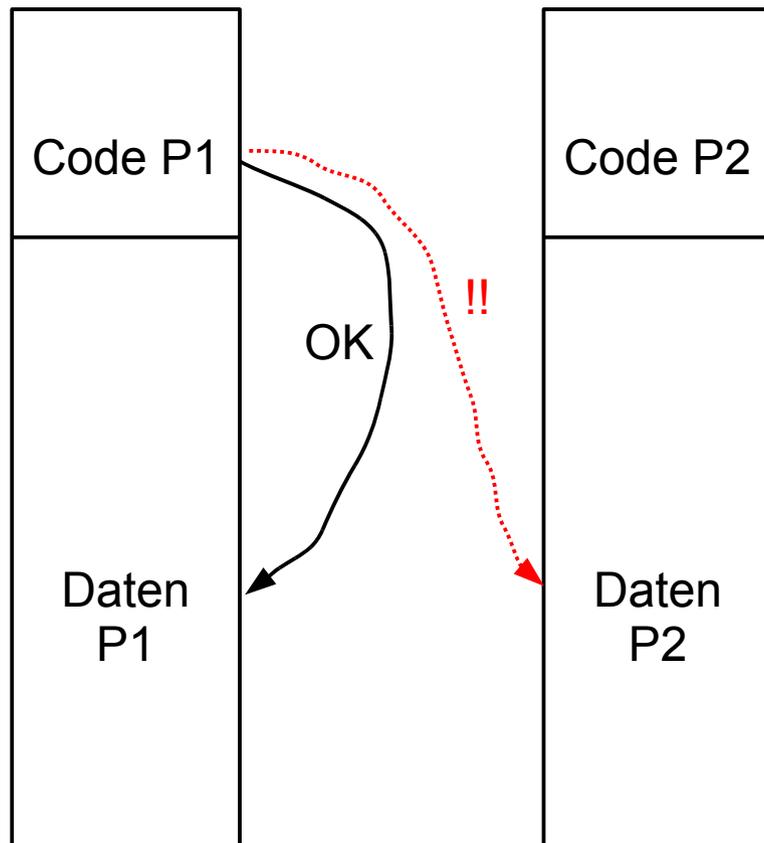
NLWP: Number of Light Weight Processes

Unterschied Prozesse / Threads (1/2)

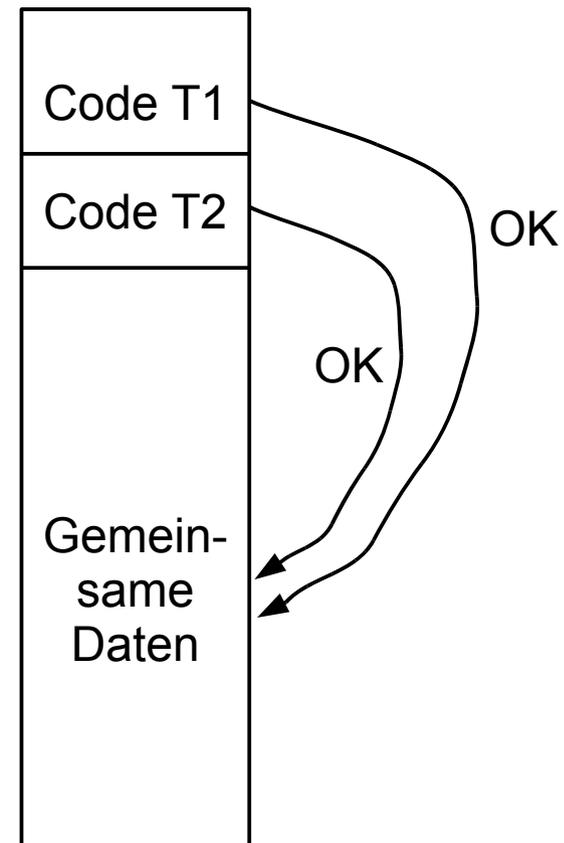
- Parallel programmieren wahlweise mit mehreren Prozessen / mehreren Threads
- Austausch / Kommunikation untereinander
 - **Prozesse:** kein gemeinsamer Speicher. Austausch z. B. über Nachrichten, Zugriff auf Datei
 - **Threads:** gemeinsamer Speicher, Austausch z. B. durch direktes Auslesen von Variablen

Unterschied Prozesse / Threads (2/2)

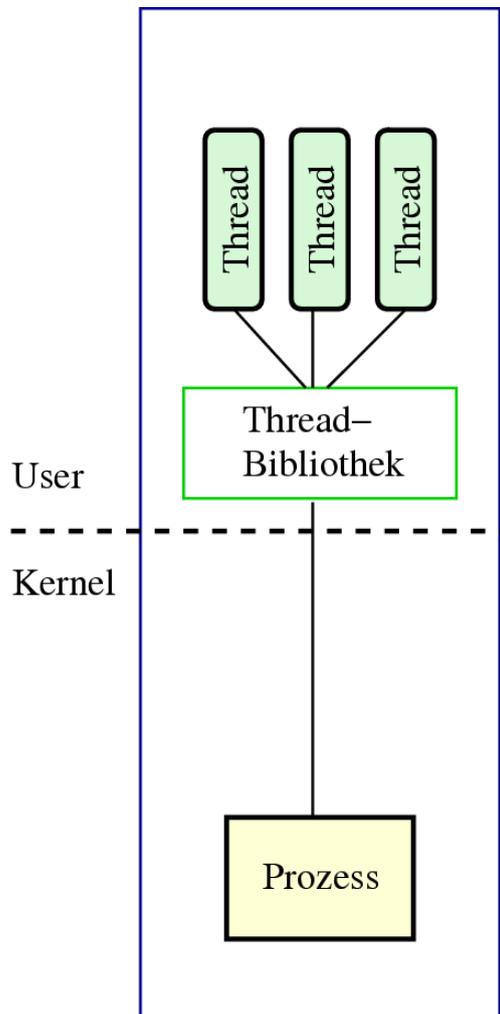
Zwei Prozesse



Zwei Threads



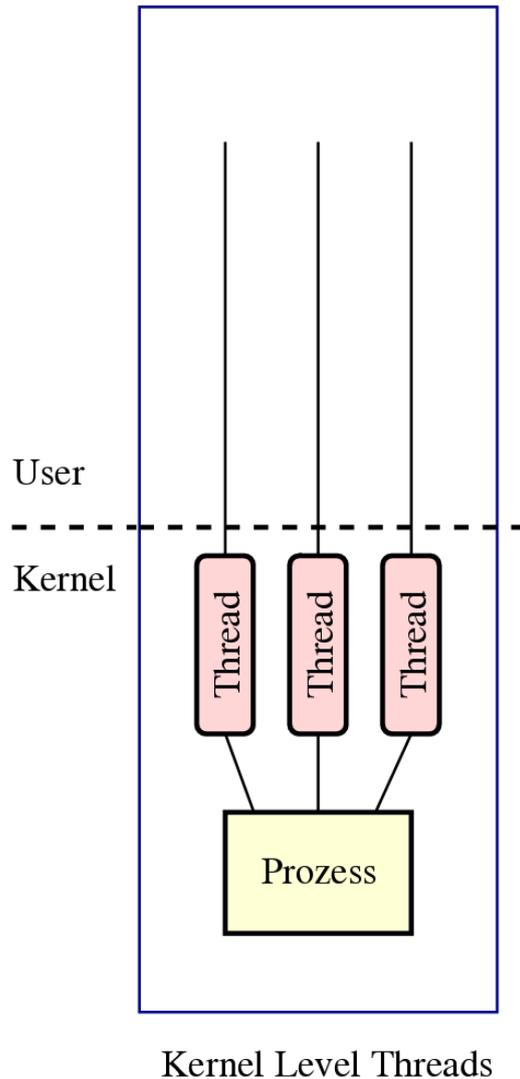
User Level Threads



User Level Threads

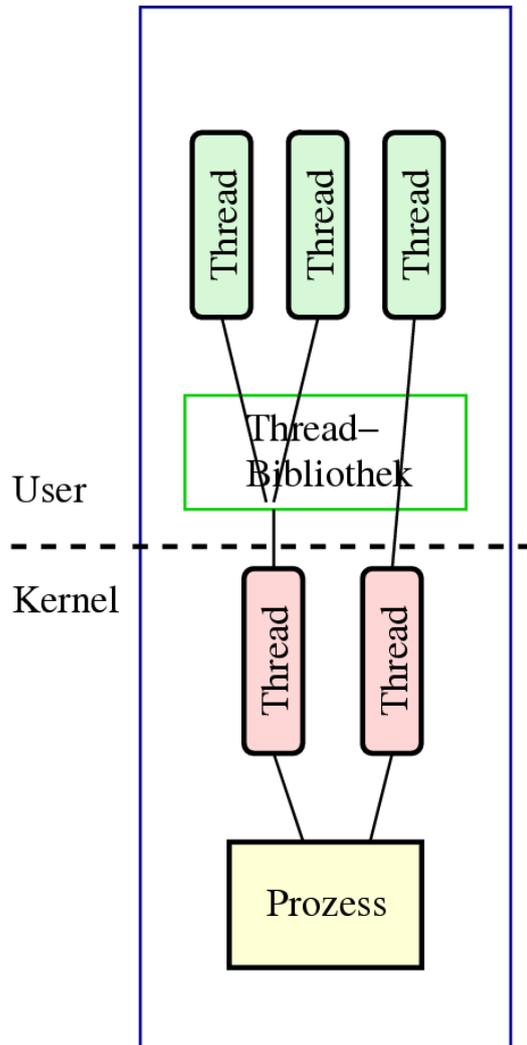
- BS kennt kein Thread-Konzept, verwaltet nur Prozesse
- Programm bindet Thread-Bibliothek ein, zuständig für:
 - Erzeugen, Zerstören
 - Scheduling
- Wenn ein Thread wegen I/O wartet, dann der ganze Prozess
- Ansonsten sehr effizient

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

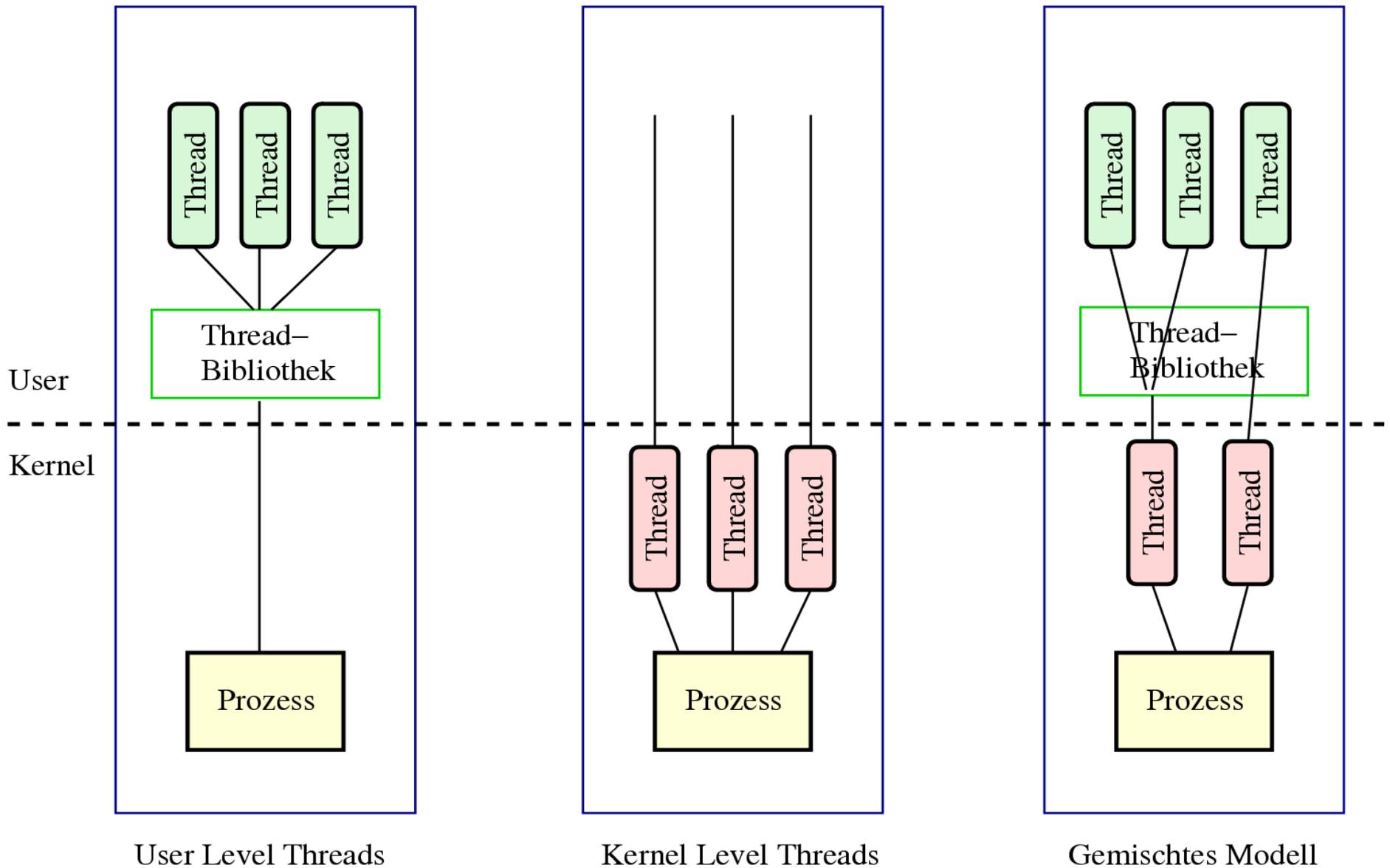
Gemischte Threads



Gemischtes Modell

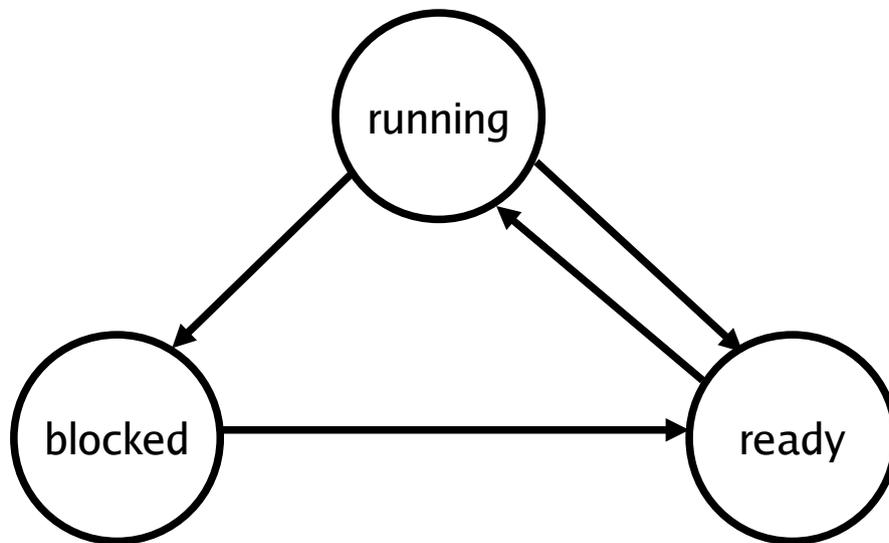
- Beide Ansätze kombinieren
- KL-Threads + UL-Threads
- Thread-Bibliothek verteilt UL-Threads auf die KL-Threads
- z.B. I/O-Anteile auf einem KL-Thread
- Vorteile beider Welten:
 - I/O blockiert nur einen KL-Thread
 - Wechsel zwischen UL-Threads ist effizient
- SMP: Mehrere CPUs benutzen

Thread-Typen, Übersicht



Thread-Zustände

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



```
Sep 19 14:20:18 amd64 sshd[20494]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61557
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[29278]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 /usr/sbin/cron[30103]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[6516]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:46:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:48:41 amd64 sshd[6609]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd[6694]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:35 amd64 sshd[9077]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64242
Sep 20 15:27:35 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:37:11 amd64 sshd[10102]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:37:11 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:38:10 amd64 sshd[10140]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63546
Sep 21 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 21 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 02:00:01 amd64 /usr/sbin/cron[17878]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 21 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:43:26 amd64 sshd[31088]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63397
Sep 21 17:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 17:53:39 amd64 sshd[31269]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64391
Sep 21 18:43:26 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 21 19:43:26 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 22 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 02:00:01 amd64 /usr/sbin/cron[17055]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 22 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 22 20:23:21 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 01:00:01 amd64 /usr/sbin/cron[24739]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 23 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 /usr/sbin/cron[25555]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:05 amd64 sshd[6554]: Accepted publickey for esser from ::ffff:87.234.201.207 port 62004 ssh2
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[6606]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62004
Sep 24 01:00:01 amd64 /usr/sbin/cron[12436]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[13253]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[20998]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[23197]: Accepted rsa for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_midi_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[29399]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[662]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1484]: (root) CMD (/sbin/evlogmgr -c 'age > "30d"')
Sep 25 02:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[8889]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[8921]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[9372]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11586]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:33 amd64 sshd[11630]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778
```

Programmierpraxis: Linux

Prozesse und Threads erzeugen (1/15)

- 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);
    }
}
```

Prozesse und Threads erzeugen (2/15)

- 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 */
}
```

Prozesse und Threads erzeugen (3/15)

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 */
    }
}
```

Prozesse und Threads erzeugen (4/15)

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
```

Prozesse und Threads erzeugen (5/15)

Abfrage, ob Programmstart über `fork()`, `exec()` erfolgreich war:

```
#include <errno.h>
main() {
    int pid = fork();
    int errno2;
    if (pid==0) {
        execl("/bin/xls",0);
        errno2=errno;
        perror ();
        printf("Fehlercode errno = %d\n",
            errno2);
    } else { wait(); }
}
```

```
> gcc -o fork-exec-fail fork-exec-fail.c
> ./fork-exec-fail
/bin/xls: No such file or directory
Fehlercode errno = 2
```

- `perror()`: Fehlermeldung in lesbarem Format
- `errno`: Globale Fehlervariable
- Nicht mit jeder `gcc`-Version...

Prozesse und Threads erzeugen (6/15)

Abbruch aller Kind-Prozesse

Zwei Szenarien:

1. Shell wird mit `exit` verlassen
→ Kind-Prozesse laufen weiter.
2. Shell wird gewaltsam geschlossen
(`kill`, Fenster schließen etc.)
→ Kind-Prozesse werden auch beendet.

Prozesse und Threads erzeugen (7/15)

```
[ In xterm-Fenster ] > nedit &
```

```
> pstree | grep nedit
```

```
    |    | -xterm---bash---nedit
```

```
> ps auxw | grep nedit
```

```
esser    24676  1.0  0.8   8248  4336 pts/4    S   15:13   0:00 nedit
```

```
> cat /proc/24676/status | grep PPid
```

```
PPid:    24659
```

```
> ps auxw|grep 24659
```

```
esser    24659  0.0  0.3   4424  1936 pts/4    Ss+ 15:12   0:00 bash
```

```
[ In xterm-Fenster ] > exit
```

```
> cat /proc/24676/status | grep PPid
```

```
PPid:    1
```

Prozesse und Threads erzeugen (8/15)

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
```

Prozesse und Threads erzeugen (9/15)

- 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.

Prozesse und Threads erzeugen (10/15)

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);
}
```

Prozesse und Threads erzeugen (12/15)

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, ..)`

Prozesse und Threads erzeugen (13/15)

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
```

Prozesse und Threads erzeugen (14/15)

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

Prozesse und Threads erzeugen (15/15)

Posix-Thread vs. Kernel-Thread:

- Ein mit `clone` erzeugter (Kernel-) Thread ist nicht dasselbe wie ein mit `pthread_create` erzeugter Posix-Thread!
- Posix-Bibliothek muss das gewünschte (Standard-) Verhalten über die von Linux bereitgestellten (`clone`-/Kernel-) Threads implementieren.

Prozessliste (1/8)

Kernel unterscheidet nicht zwischen Prozessen und Threads.

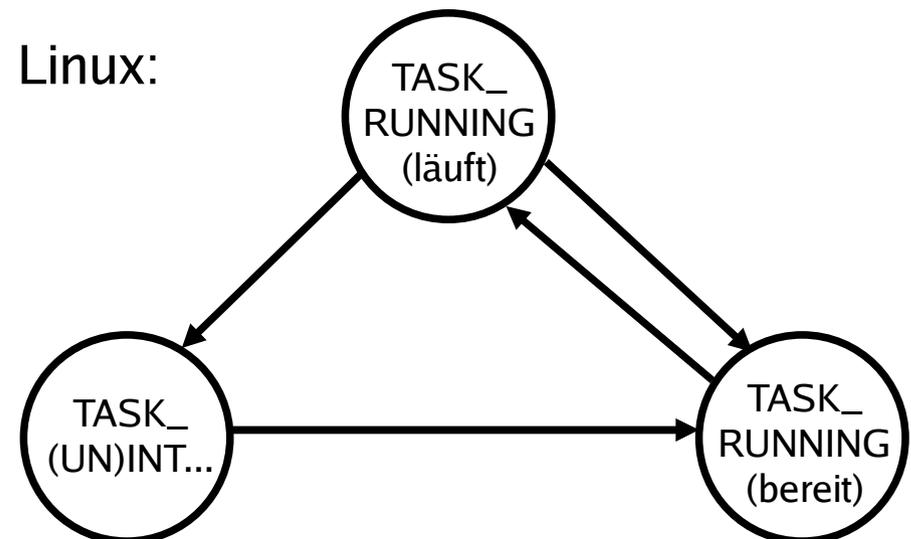
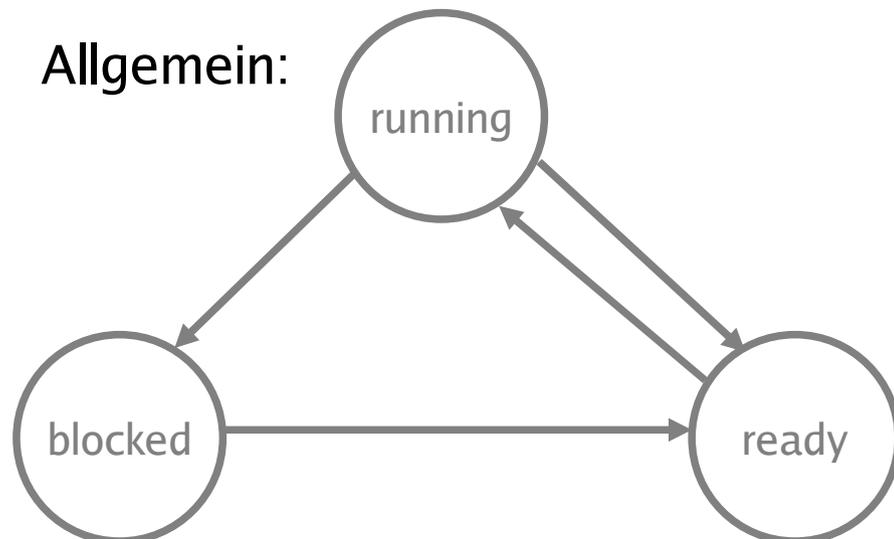
- Doppelt verkettete, ringförmige Liste
- Jeder Eintrag vom Typ `struct task_struct`
- Typ definiert in [include/linux/sched.h](#)
- Enthält alle Informationen, die Kernel benötigt
- `task_struct`-Definition 132 Zeilen lang!
- Maximale PID: 32767 (short int)

Prozessliste (2/8)

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
```

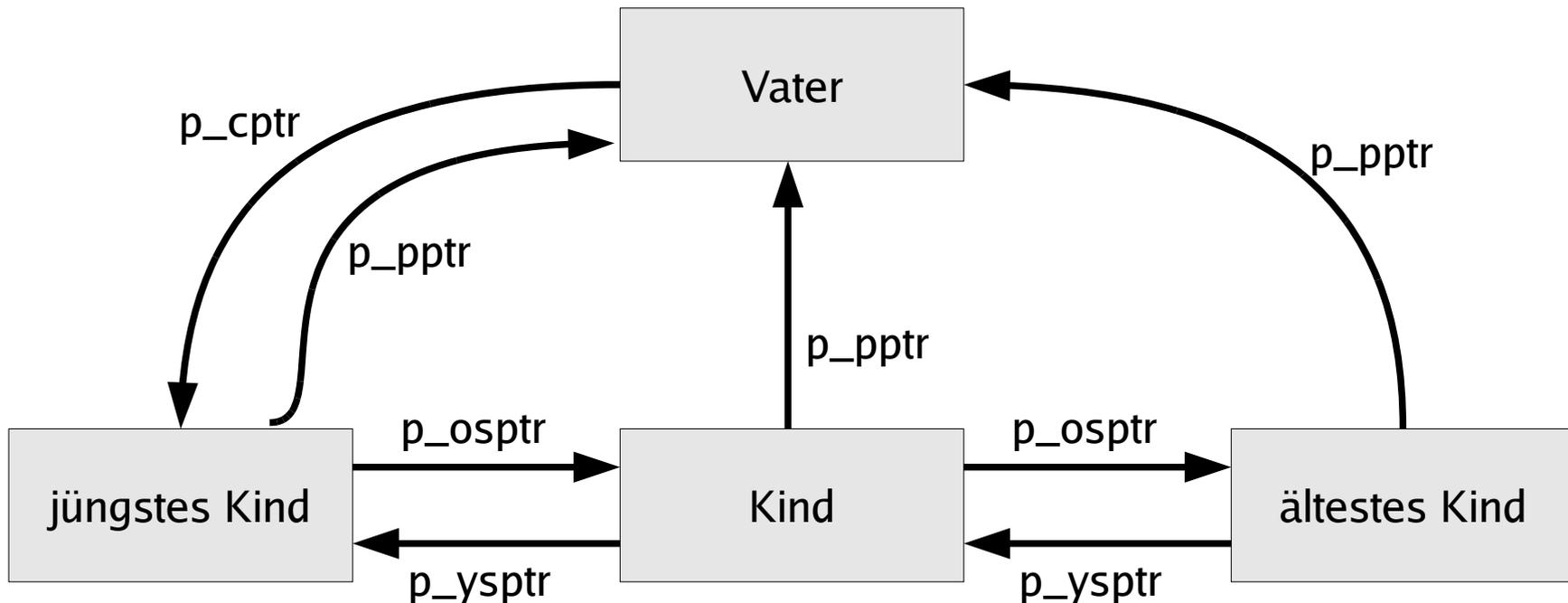
- TASK_RUNNING: ready oder running
- TASK_INTERRUPTIBLE: entspricht blocked
- TASK_UNINTERRUPTIBLE: auch blocked
- TASK_STOPPED: angehalten (z. B. von einem Debugger)
- TASK_ZOMBIE: beendet, aber Vater hat Rückgabewert nicht gelesen



Prozessliste (3/8)

Verwandtschaftsverhältnisse (alte Linux-Version)

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



Prozessliste (4/8)

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) {  
    ...  
}
```

Prozessliste (5/8)

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`

Prozessliste (6/8)

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, ...);
```

Prozessliste (7/8)

Sessions

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

Prozessliste (8/8)

```
> 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
```

Prozesserzeugung (1/2)

Wichtigste Datei in den Kernel-Quellen: `kernel/fork.c`
(enthält u. a. `copy_process`)

- `fork()` ruft `clone()` auf,
- `clone()` ruft `do_fork()` auf, und
- `do_fork()` ruft `copy_process()` auf

Prozesserzeugung (2/2)

`copy_process()` macht:

- `dup_task_struct()`: neuer Kernel Stack, `thread_info` Struktur, `task_struct`-Eintrag
- Kind-Status auf `TASK_UNINTERRUPTIBLE`
- `copy_flags()`: `PF_FORKNOEXEC`
- `get_pid()`: Neue PID für Kind vergeben
- Je nach `clone()`-Parametern offene Dateien, Signal-Handler, Prozess-Speicherbereiche etc. kopieren oder gemeinsam nutzen
- Verbleibende Rechenzeit aufteilen (→ Scheduler)

Danach: aufwecken, starten (Kind kommt vor Vater dran)

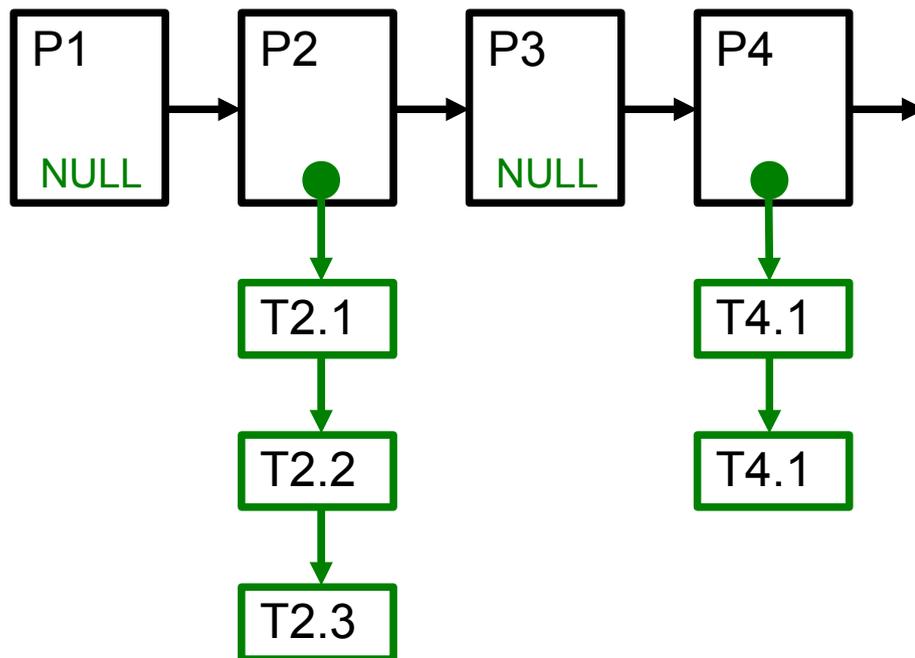
Threads im Kernel (1/3)

- 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

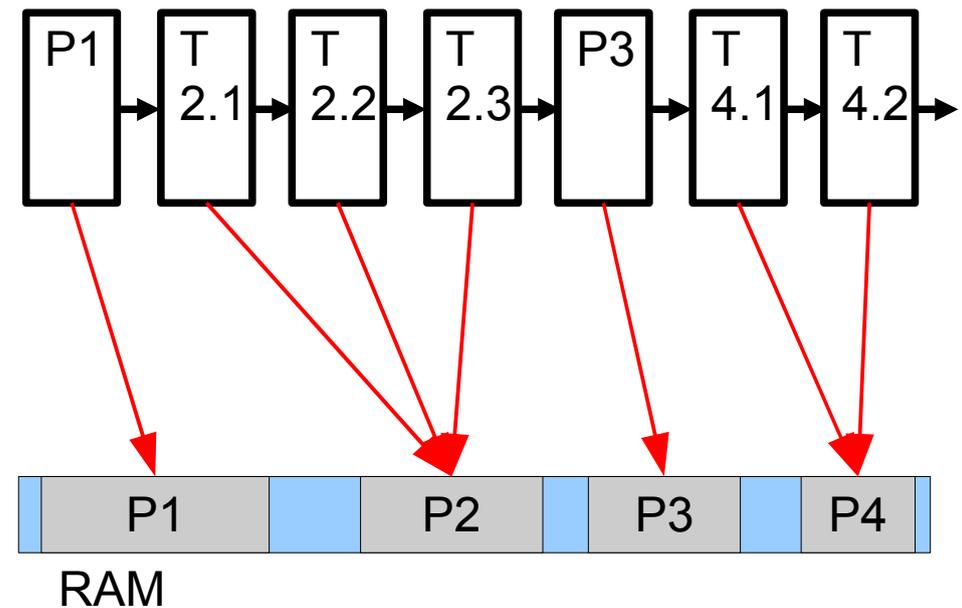
Threads im Kernel (2/3)

- Fundamental anders als z. B. Windows und Solaris

Modell 1:
reine Prozesslisten



Modell 2 (Linux):
Prozesse + Threads gemischt



Threads im Kernel (3/3)

- Thread-Erzeugung: auch über `clone ()`
- einfach andere Aufrufparameter:
 - Prozess: `fork ->`
`clone (SIGCHLD, 0);`
 - Thread:
`clone (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);`
(`vm`: virtual memory, `fs`: Dinge wie Arbeitsverzeichnis, `umask`,
Root-Verzeichnis des Prozesses, `files`: offene Dateien,
`sighand`: Signal Handlers)