Studium neben dem Beruf

FOM Hochschule für Oekonomie & Management
University of Applied Sciences

Betriebssysteme Theorie

SS 2011

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz A (02.03.2011)
Einführung, Prozesse & Threads

Zur Vorlesung (1)

Hilfreiche Vorkenntnisse:

- **C** – Grundlagen der Programmierung in C (oder C++, C#, Java)
- **Rechnerarchitekturen** – grober Aufbau eines Computers (Prozessor, Hauptspeicher, Peripherie etc.)
- **Unix-Shell** – Benutzung der Standard-Shell *bash* unter Linux ➞ Bash-Crashkurs
- **Python** – Programme lesen können (für Praktikum: auch programmieren)

Zur Vorlesung (2)

Praktikum:

- unter Linux
- Praktische Beispiele in **Python** und **C** umsetzen

Prüfung und Benotung

1. Lernfortschrittskontrolle (LFK)
2. Klausur über 120 Minuten

Fragen:

- direkt in der Vorlesung (Handzeichen)
- oder danach
- oder per E-Mail

Zur Vorlesung (3)

Termine

Mi 02.03.2011	Do 05.05.2011	jeweils
Mi 16.03.2011	Do 12.05.2011	18:00-21:15
Do 14.04.2011	Mi 18.05.2011	
Do 21.04.2011	Do 26.05.2011	
Mi 27.04.2011		

Service / Web-Seite: http://fom.hgesser.de

- Folien und Praktikumsaufgaben
- Vorlesungs-MP3s („*test, test* “)
- Probeklausur gegen Semesterende

Über den Dozenten

Hans-Georg Eßer

- Dipl.-Math. (RWTH Aachen, 1997)
  Dipl.-Inform. (RWTH Aachen, 2005)
- Chefredakteur Computerzeitschrift (seit 2000)
- Autor diverser Computerbücher
- seit 2006 Dozent an der Hochschule München und an der FOM: Betriebssysteme, Rechnerarchitektur, IT-Infrastrukturen, Informatik-Grundlagen
- Seit 2010 Doktorand (Univ. Erlangen-Nürnberg)

Einführung und Motivation

Wofür Betriebssysteme?

„Klicken Sie auf Schließen.“

- VHS-Kurs Windows, Linux etc.?
- Nicht: „Wie bediene ich ... ?“, sondern: „Wie und warum funktioniert ... intern?“
- Konsequenzen für Anwendungsentwickler
- Sicherheitsprobleme
- Auswahl eines geeigneten Betriebssystems

... und das Thema ist auch an sich spannend

Aufgaben von Betriebssystemen (1)

- Abstraktionsschicht zwischen Hardware und Programmen (➞ virtuelle Maschine)
- Verwaltung der vorhandenen Ressourcen
- Einheitlicher Zugriff auf Geräte einer groben Kategorie, z. B.:
  - *Datenträger* (Plattenpartition, CD, DVD, Diskette, USB-Stick, Netzwerk-Volume)
  - *Drucker* (PostScript-Laser, Etikettendrucker, Billig-Tintenstrahler, ...)

Aufgaben von Betriebssystemen (2)

- Schützt Hardware vor direkten Zugriffen (➞ defekte oder bösartige Software)
- Befreit Software vom Zwang, die Hardware im Detail zu kennen
- Zulassen mehrerer Benutzer und Abgrenzung (Multi-user)
- Parallelbetrieb mehrerer Anwendungen (Multi-tasking): faire Aufteilung der Ressourcen

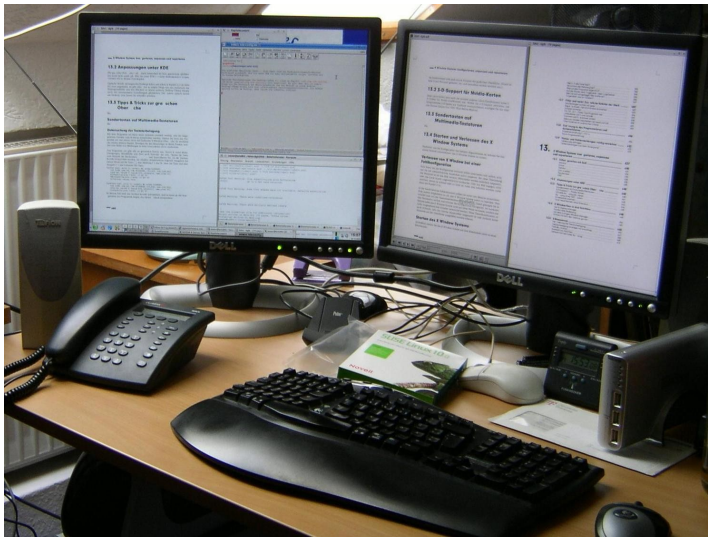
Aufgaben von Betriebssystemen (3)

- Virtualisierung des Speichers
  - ➞ Anwendungen müssen nicht wissen, wo sie im Hauptspeicher liegen
  - ➞ Speicher über phys. RAM hinaus verfügbar (Swap etc.)

Beispiele (1)

Desktop-PC – die Standardaufgabe, Intel & Co.

- Anwendungsprogramme (Office, Grafik, kaufmännische Software etc.)
- Internet-Zugang und Web-basierte Anwendungen (WWW, E-Mail, File Sharing, ...)
- Datenbank-Client
- Software-Entwicklung
- Multimedia



Beispiele (2)

Server-PC

Häufig ähnliche Hardware wie Desktop-PC, aber ganz andere Einsatzgebiete:

- Web- / FTP- / Mail-Server
  (Internet oder Intranet)
- Datenbank-Server
- „Number Crunching“ bzw.
  High Performance Computing (oft: Cluster)

Beispiele (3)

Industrieanwendungen

- Robotersteuerung
- automatische Navigation
- Temperaturregelung
- Motorenkontrolle
- Herzschrittmacher

Bild: Wikipedia, KUKA Schweißanlagen

→ **Echtzeit-Betriebssysteme**
(real time operating systems)

Beispiele (4)

Embedded systems
(ohne Echtzeit-Ansprüche)

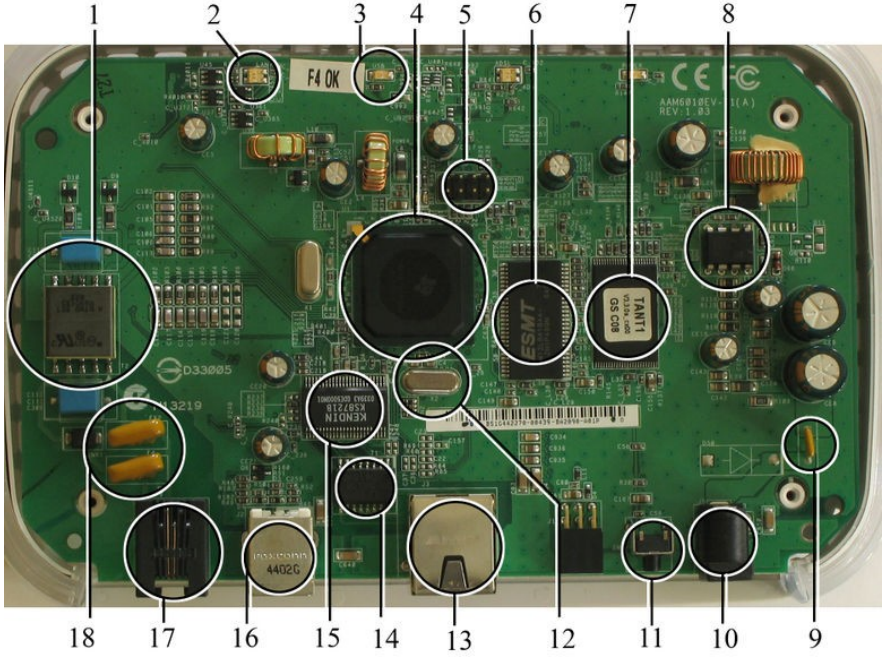


Foto: Wikipedia (Mike1024)

- Mobiltelefone, PDAs, mobile MP3/Video-Player
- Fernseher, Videorekorder, DVD-Player
- DSL-WLAN-Router (mit Firewall etc.)
- Taschenrechner
- Videospiel-Konsolen
- Geldautomaten

Software-Entwicklung (1)

Beim Programmieren tauchen häufig Probleme in zwei Bereichen auf:

- **Zuverlässigkeit**
  Software tut nicht das, was sie soll;
  unerwartetes Verhalten;
  mangelnde Fehlertoleranz
- **Sicherheit**
  Software ist nicht geschützt vor Angriffen durch Dritte

Software-Entwicklung (2)

Funktionsweise des Betriebssystems nicht klar

➡ fehlerhaft programmierte Anwendungen, z. B.

- Race Conditions
- Buffer Overflows

Darum verstehen und lernen, wie Betriebssysteme intern arbeiten

Gliederung

Gliederung

1. Einleitung
2. Prozesse und Threads
3. Interrupts
4. Scheduler
5. Synchronisation und Deadlocks
6. Speicherverwaltung
7. Datei- und I/O-Systeme

2. Prozesse und Threads (1)

Gliederung

Prozess:
Programm, das in den Speicher geladen wurde und ausgeführt wird / werden soll

Mehr als nur der Programmcode:

- Eigene Daten
- Stack
- Programmzähler
- Umgebung

2. Prozesse und Threads (2)

Gliederung

Thread:

Ähnlich wie Prozess, aber:

- mehrere Threads greifen auf gleichen Speicher zu
- Thread-Verwaltung nicht unbedingt im Kernel (➞ weniger Verwaltungs-Overhead)
- User level / Kernel level

3. Interrupts

Gliederung

- Verschiedene Interrupt-Typen
  - Hardware Interrupts
  - Software Interrupts (Trap)
  - Exceptions (z. B. Division 1/0, Zugriff auf falsche Adresse)
- Interrupt Handler

4. Scheduler

Gliederung

- Rechenzeit an Prozesse verteilen
- Scheduling-Prinzipien: präemptiv, kooperativ
- Scheduling-Verfahren: Round Robin Scheduler, Priority Scheduler, Shortest Job First Scheduler etc.
- Was passiert beim Prozess-wechsel?

5. Synchronisation und Deadlocks

Gliederung

- Parallele Threads / Prozesse
- Zugriff auf gemeinsame Daten
- Kritische Abschnitte, gegenseitiger Ausschluss
- Synchronisationsmethoden: Mutex, Semaphor, ...
- Wann / wie kommt es zu Deadlocks (Blockaden)?
- Wie vermeidet man Deadlocks?

6. Speicherverwaltung

Gliederung

- Veraltete und moderne Ansätze für Speichernutzung
- Virtualisierung des Speichers
- Paging / Seitenwechselverfahren
- Seitenersetzungsstrategien

7. Datei- und I/O-Systeme

Gliederung

Schwerpunkt: Dateisysteme

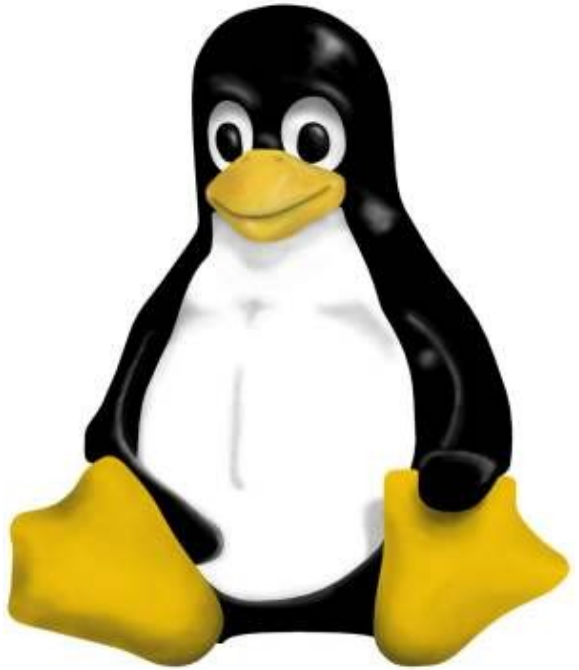
- Klassische Dateisysteme (CP/M, MS-DOS)
- Moderne Dateisysteme (Windows NT/XP/Vista, Linux)
- Theorie und Praxis

I/O-Systeme

- memory-mapped I/O

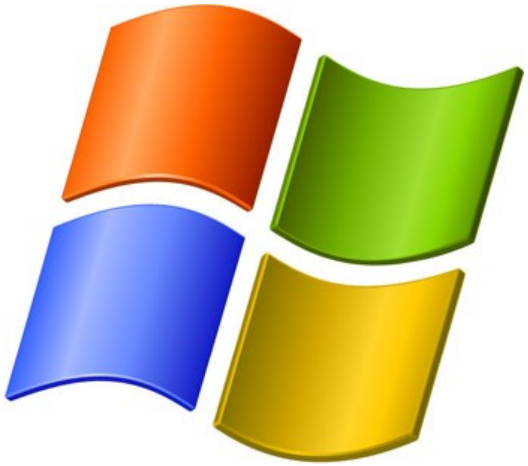
Linux

- Offene Kernel-Quellen:
  - ➡ nachlesen, wie etwas geht
  - ➡ ändern, was nicht gefällt
- Etabliertes Standardsystem für sehr viele Plattformen (PC Desktop / Server, Embedded etc.)
- Image eines virtuellen Linux-PCs für VMware / VirtualBox

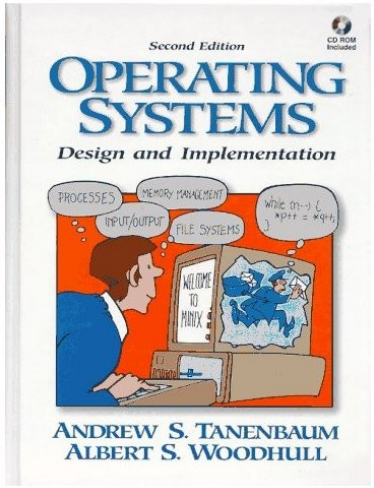


Windows?

- evtl. auch (ein bisschen) Windows in dieser Vorlesung
- Kein Windows in der Prüfung



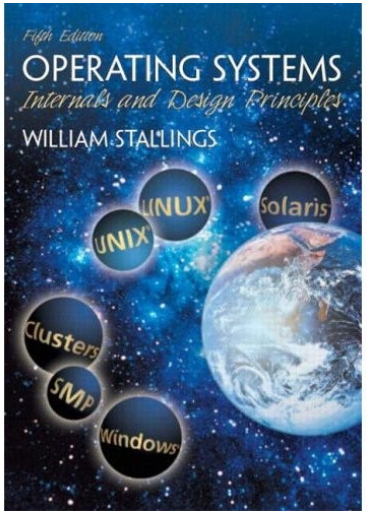
Literatur (1)



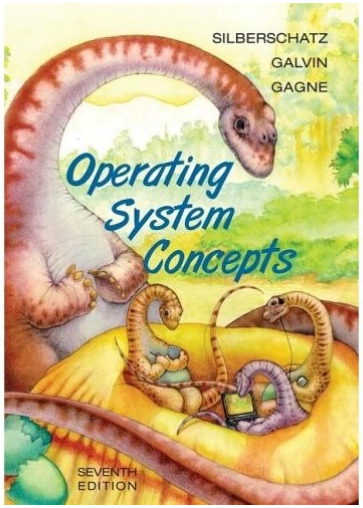
Operating Systems
Design and Implementation
(Tanenbaum, Woodhull)
Prentice Hall
(englisch)

Betriebssysteme
Ein Lehrbuch mit Übungen zur Systemprogrammierung in Unix/Linux (Ehses et al.)
ISBN 3-8273-7156-2
Pearson Studium, 30 Euro

Literatur (2)



Operating Systems
Internals and Design Principles
(Stallings)
Prentice Hall, ca. 80 Euro
(englisch)



Operating System Concepts
(Silberschatz, Galvin, Gagne)
Wiley, ca. 52 Euro
(englisch)

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, Vista, 7, Linux, Mac OS, ...: 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:

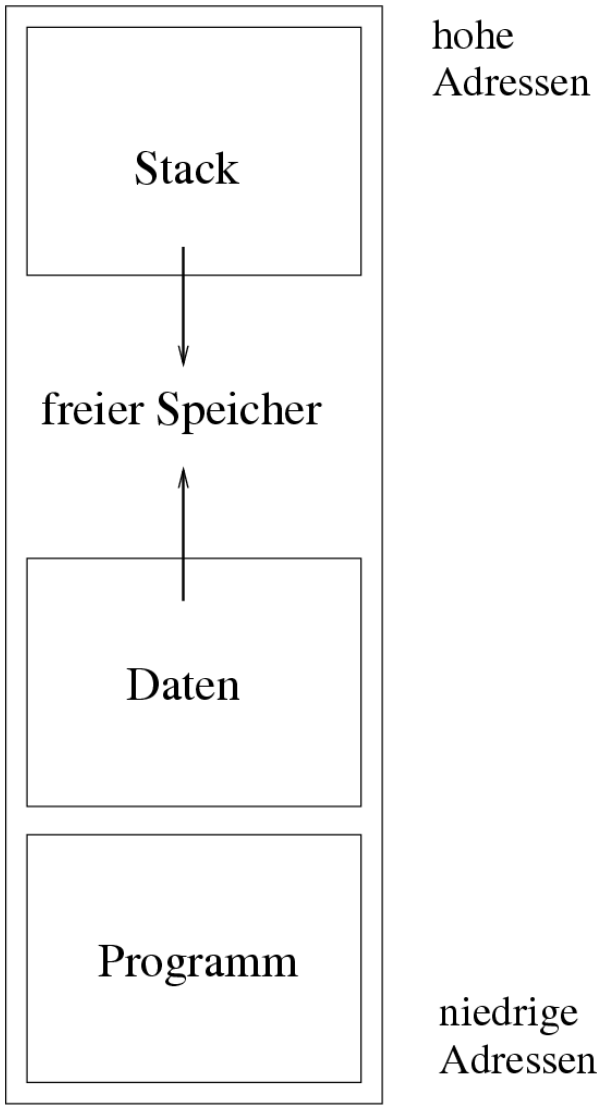
- Informationen über alle Prozesse und ihre Zustände
- Jeder Prozess hat dort einen **Process Control Block** (**PCB**):
  - Identifier (PID)
  - Registerwerte inkl. Befehlszähler
  - Speicherbereich des Prozess
  - Liste offener Dateien und Sockets
  - Informationen wie Vater-PID, letzte Aktivität,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

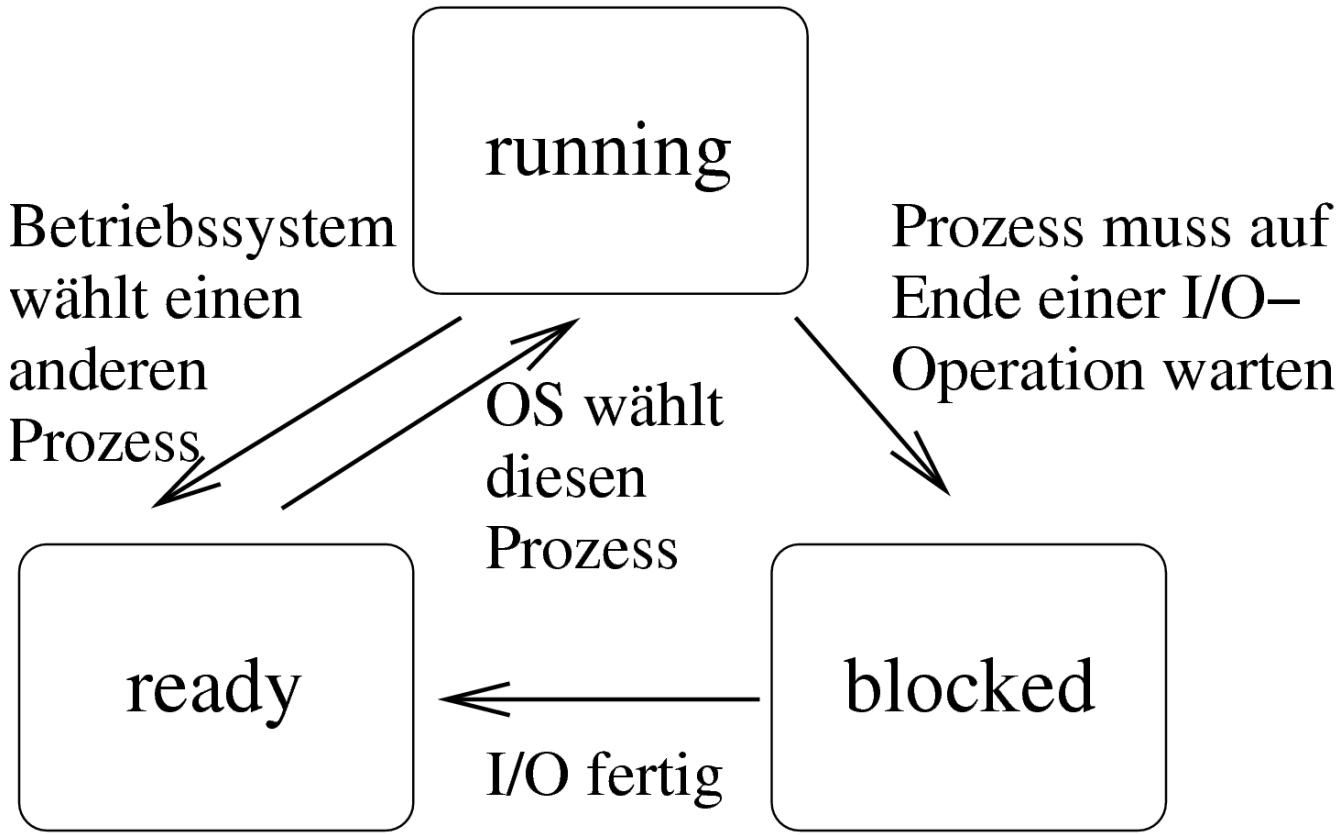
Zustände

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

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

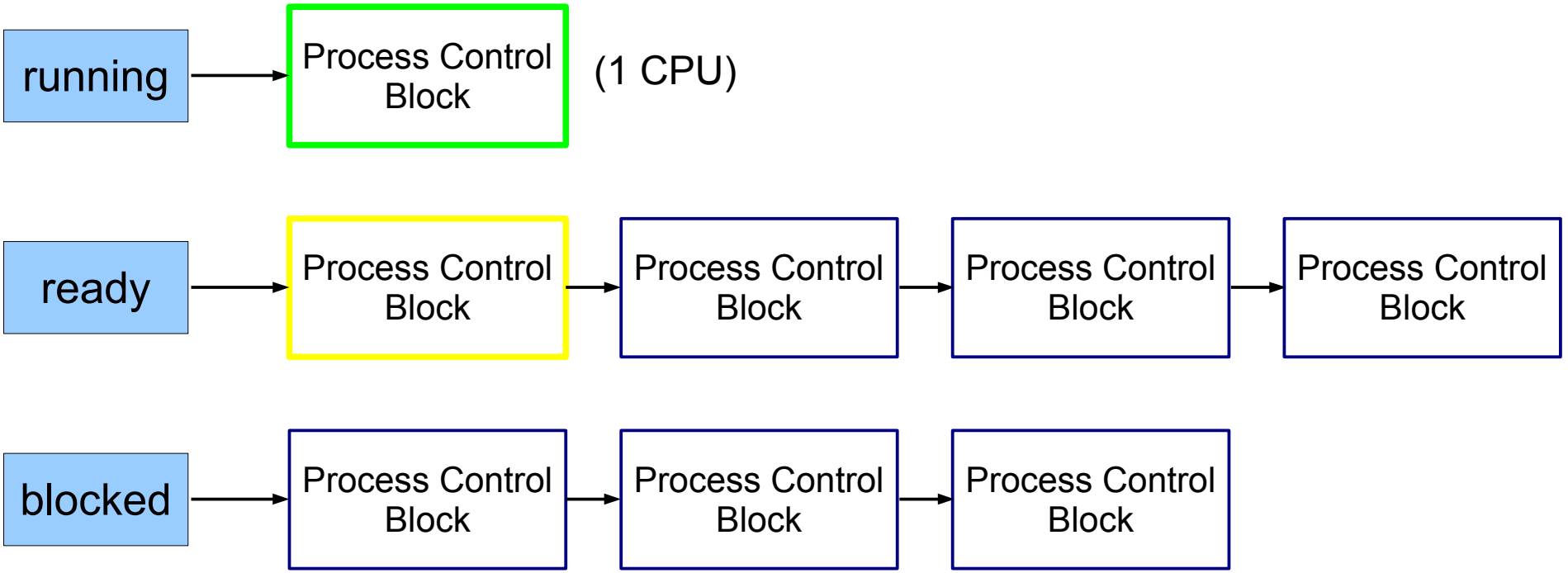
Prozesse (4)

Zustandsübergänge

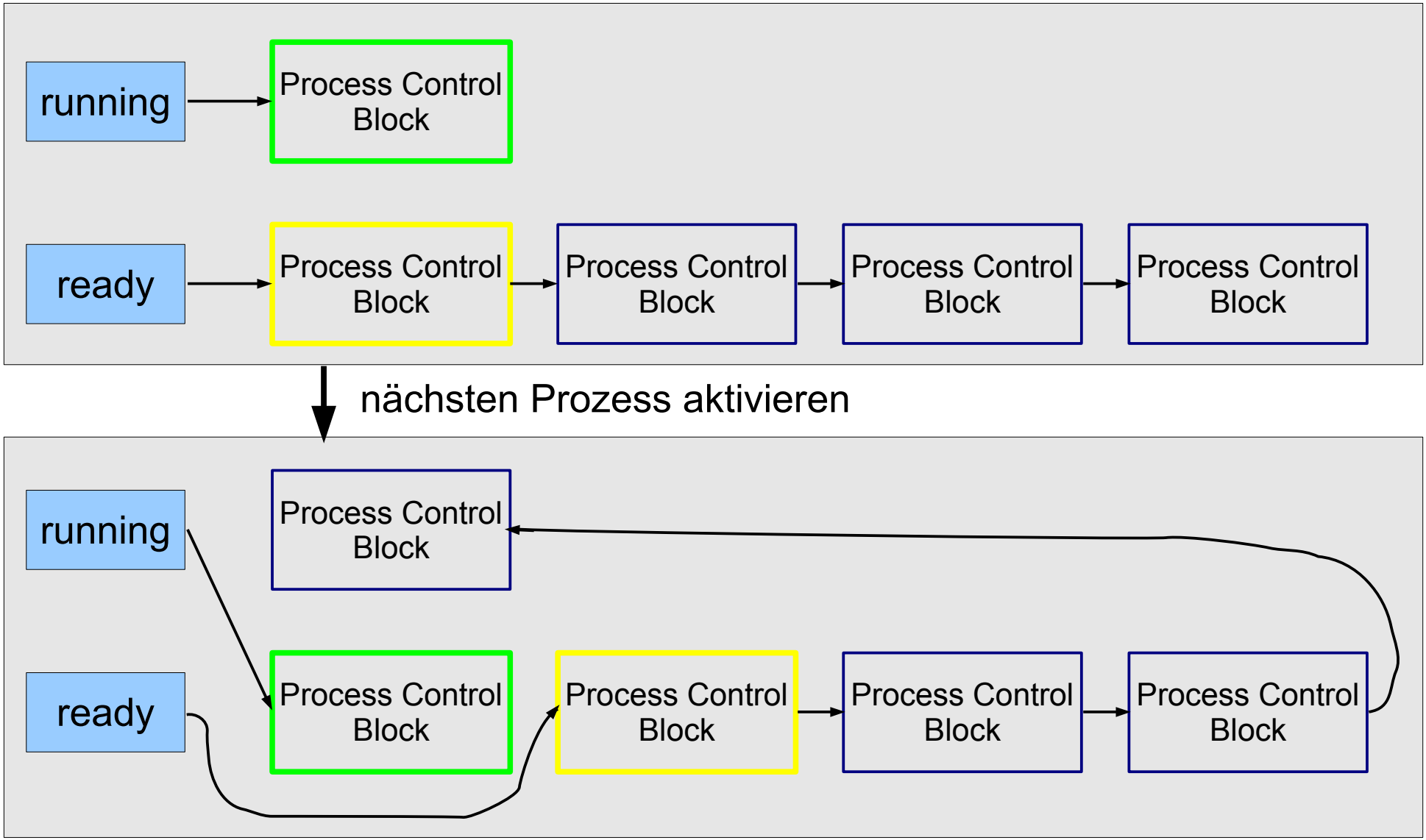


Prozesse (5)

Prozesslisten



Prozesse (6)


running
Process Control Block
ready
Process Control Block
Process Control Block
Process Control Block
Process Control Block
nächsten Prozess aktivieren
running
Process Control Block
ready
Process Control Block
Process Control Block
Process Control Block
Process Control Block

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 (4)

```
> pstree -p
init(1)-+-acpid(2266)
        |-auditd(2727)---{auditd}(2728)
        |-cron(3234)
        |-cupsd(2706)
        |-gpg-agent(4031)
        |-hald(2309)-+-hald-addon-acpi(2616)
        |            |-hald-addon-stor(2911)
        |            `-hald-addon-stor(2914)
        |-kded(4079)
        |-kdeinit(4072)-+-artsd(7184)
        |               |-kio_file(4402)
        |               |-klauncher(4077)
        |               |-konqueror(22430)
        |               |-konsole(11064)-+-bash(11065)---ssh(31205)
        |               |                |-bash(11119)---sux(11444)---bash(11447)
        |               |                |-bash(11137)
        |               |                |-bash(25637)-+-ssh(4522)
        |               |                |             `-xmms(7169)-+-{xmms}(7170)
        |               |                |                          `-{xmms}(7171)
        |               |                `-bash(15608)
        |               |-konsole(4773)-+-bash(4774)---ssh(8037)
        |               |               |-bash(8040)---ssh(8058)
        |               |               `-bash(8061)-+-less(15188)
        |               |                            |-nedit(9628)
        |               |                            `-xpdf(8103)
```

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_alles ();
}
```

```
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

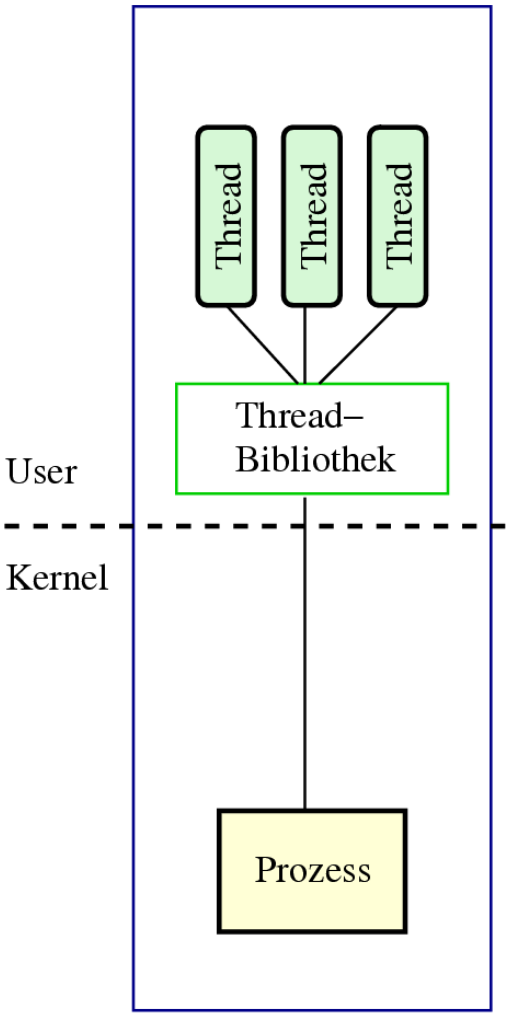
Ein Prozess, neun Threads:

```
[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

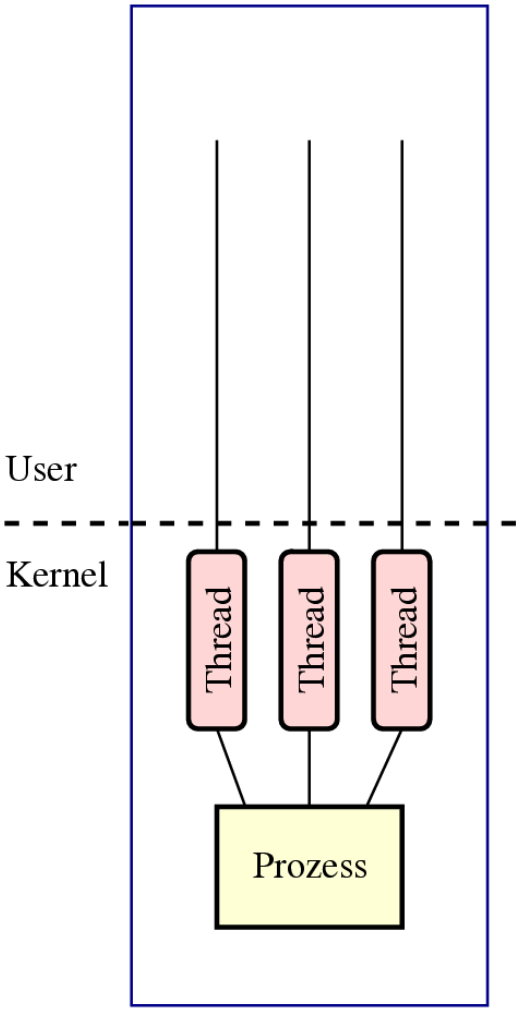
User Level Threads



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

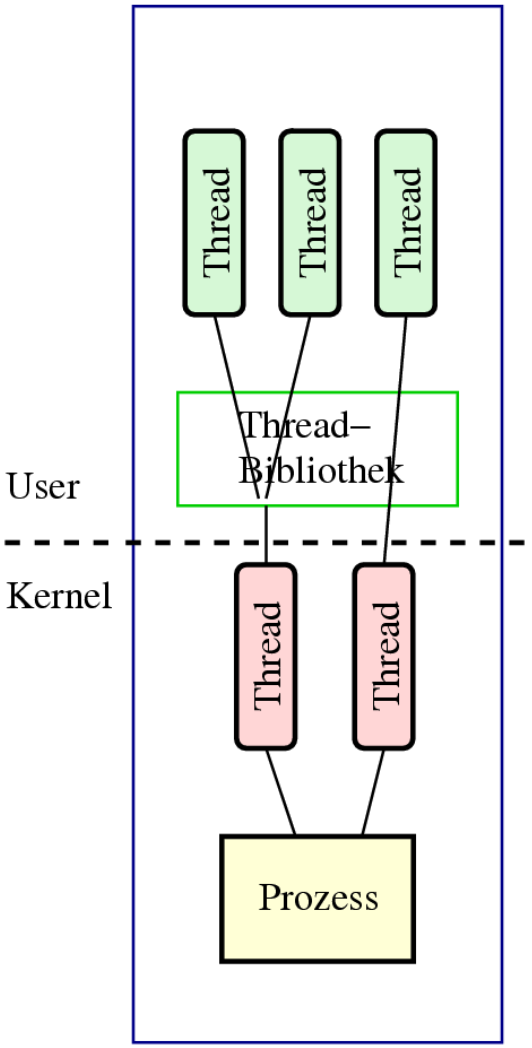
Kernel Level Threads



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

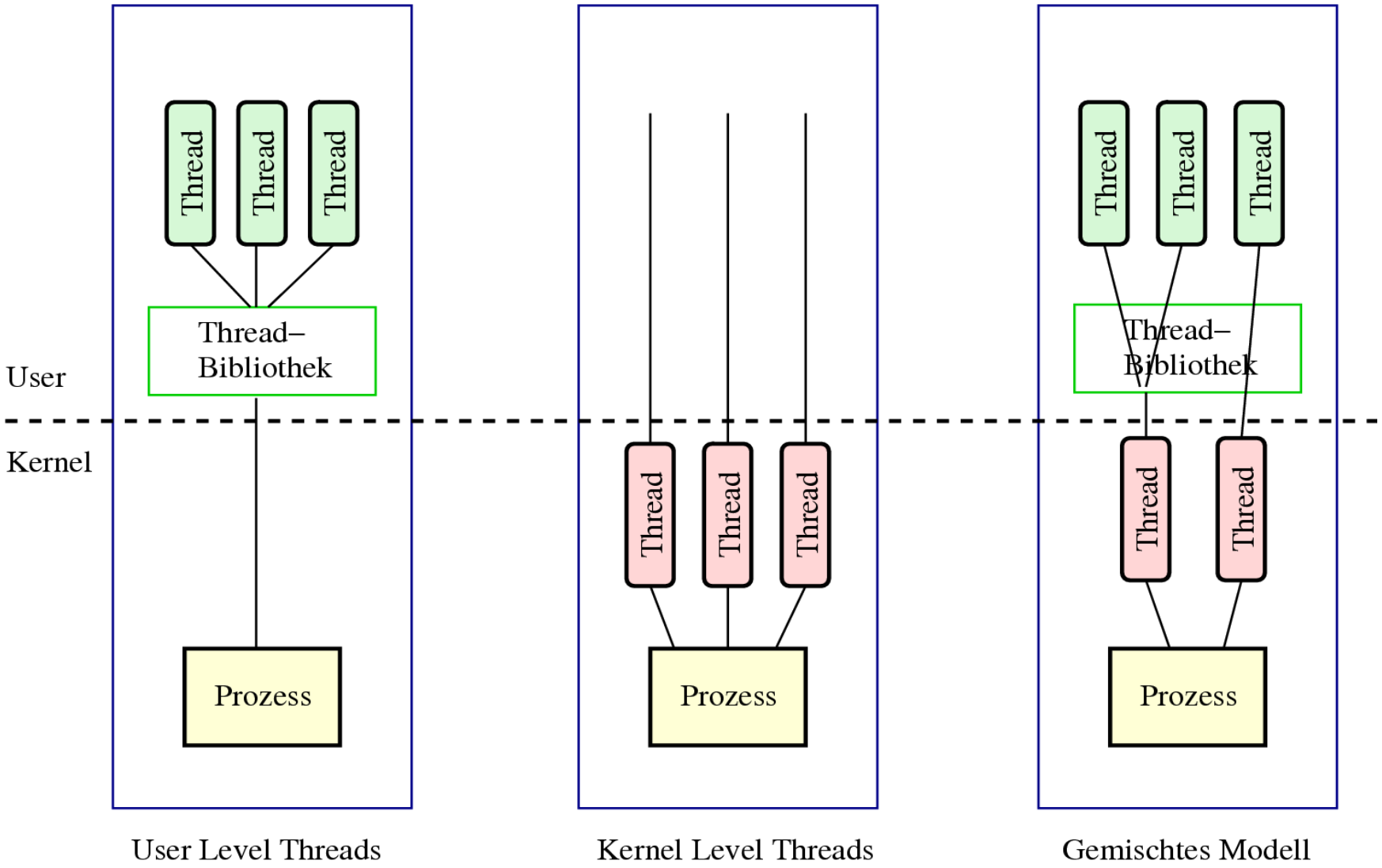
Gemischte Threads



Gemischtes Modell

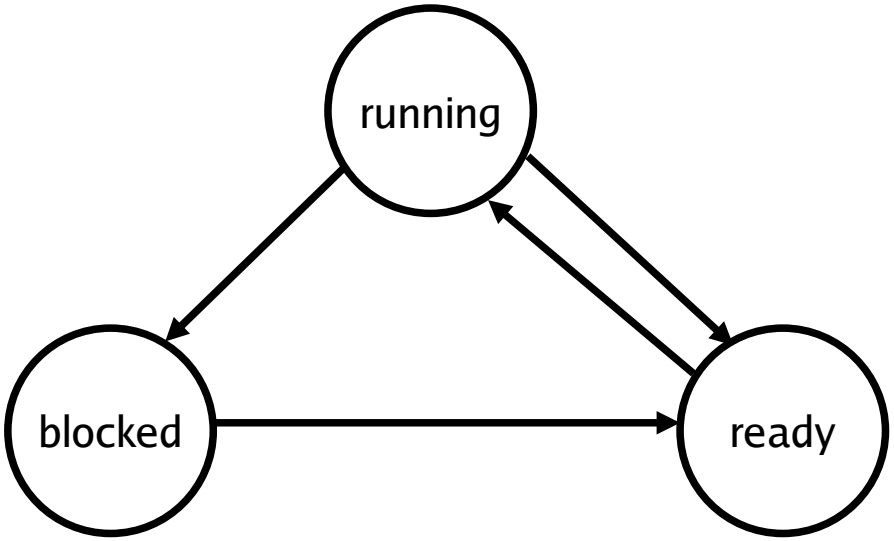
- Beide Ansätze kombinieren
- KL-Threads + UL-Threads
- Thread-Bibliothek verteilt UL-Threads auf die KL-Threads
- z.B. I/O-Anteile auf einem KL-Thread
- Vorteile beider Welten:
  - I/O blockiert nur einen KL-Thread
  - Wechsel zwischen UL-Threads ist effizient
- SMP: Mehrere CPUs benutzen

Thread-Typen, Übersicht



Thread-Zustände

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



Programmierpraxis: Linux

Prozesse und Threads erzeugen (1/12)

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 (2/12)

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 (3/12)

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 (4/12)

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 (5/12)

Linux: `pthread`-Bibliothek (POSIX Threads)

	Thread	Prozess
Erzeugen	`pthread_create()`	`fork()`
Auf Ende warten	`pthread_join()`	`wait()`

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

Prozesse und Threads erzeugen (6/12)

- 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 (7/12)

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();
}
```

Prozesse und Threads erzeugen (8/12)

```
#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 (9/12)

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 (10/12)

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 (11/12)

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 (12/12)

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)

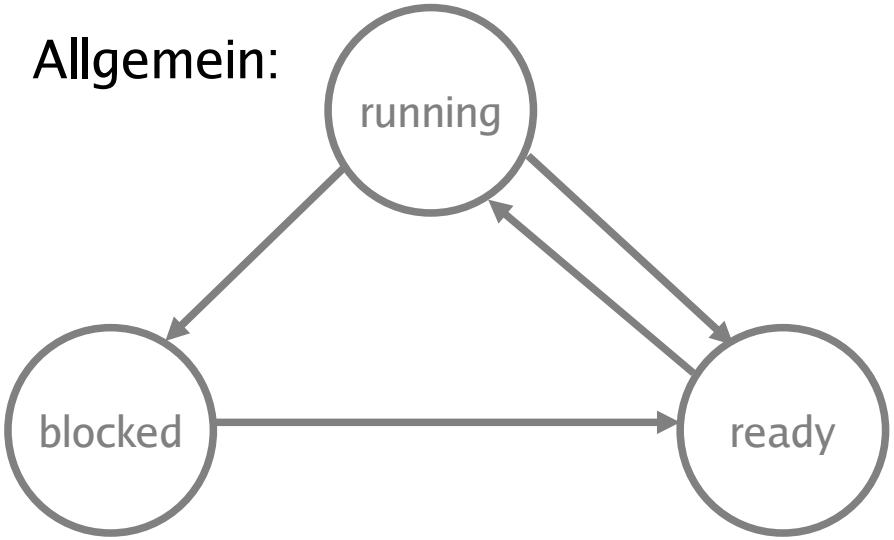
Prozessliste (2/8)

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

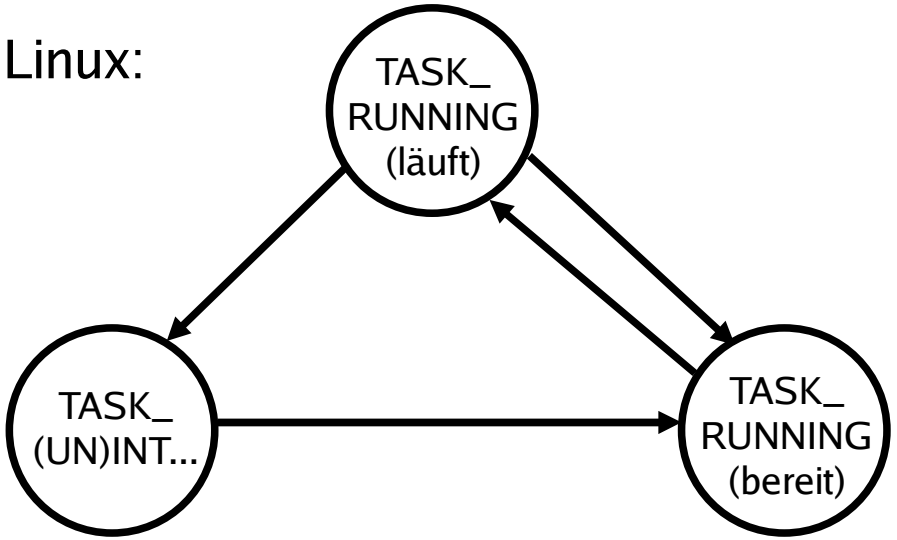
```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_TRACED             8
/* in tsk->exit_state */
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
/* in tsk->state again */
#define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

Allgemein:



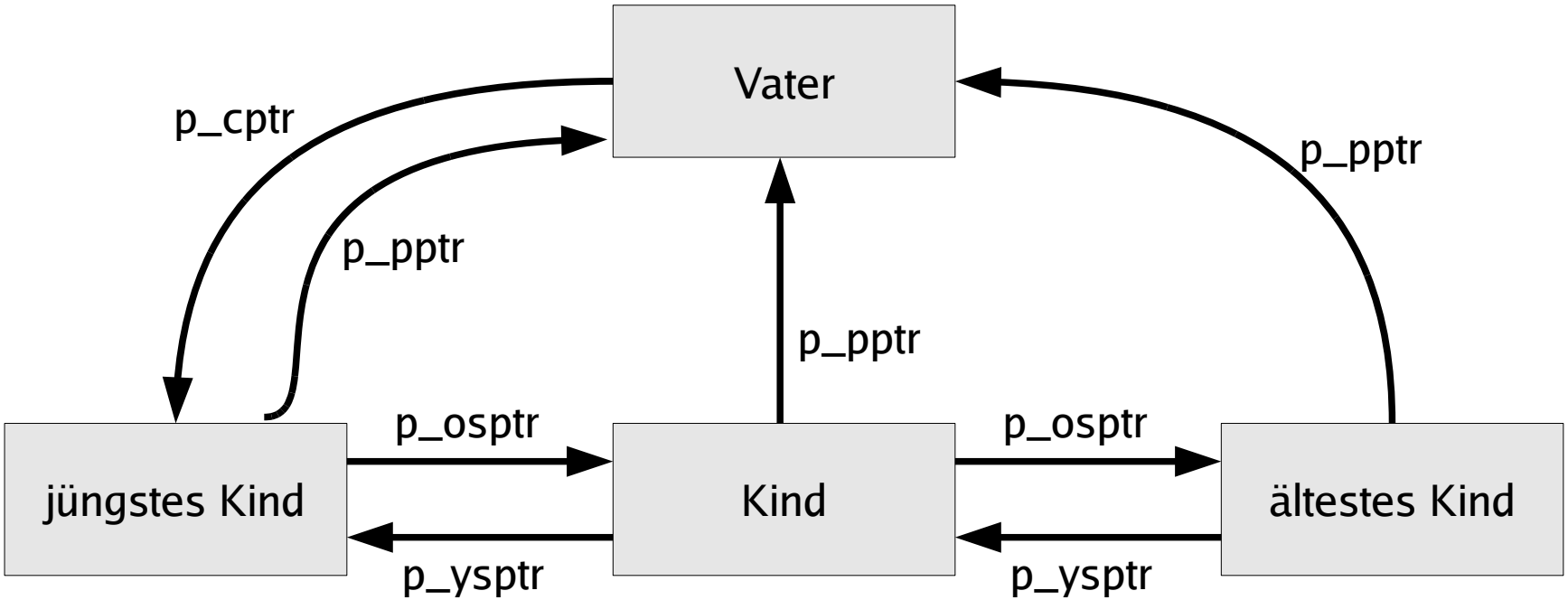
Linux:



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 ksmserver
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)

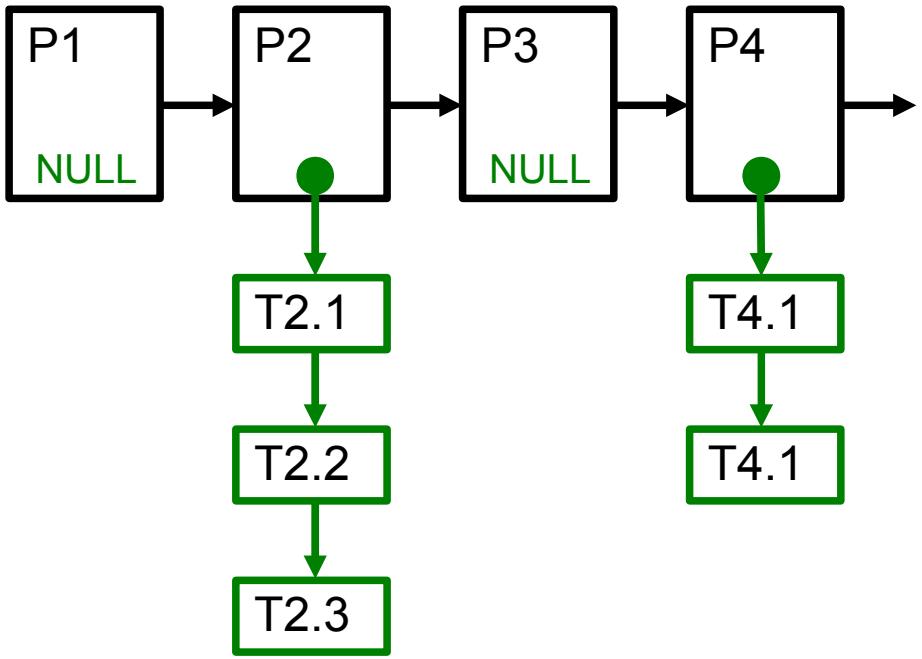
Threads im Kernel (1/3)

- 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

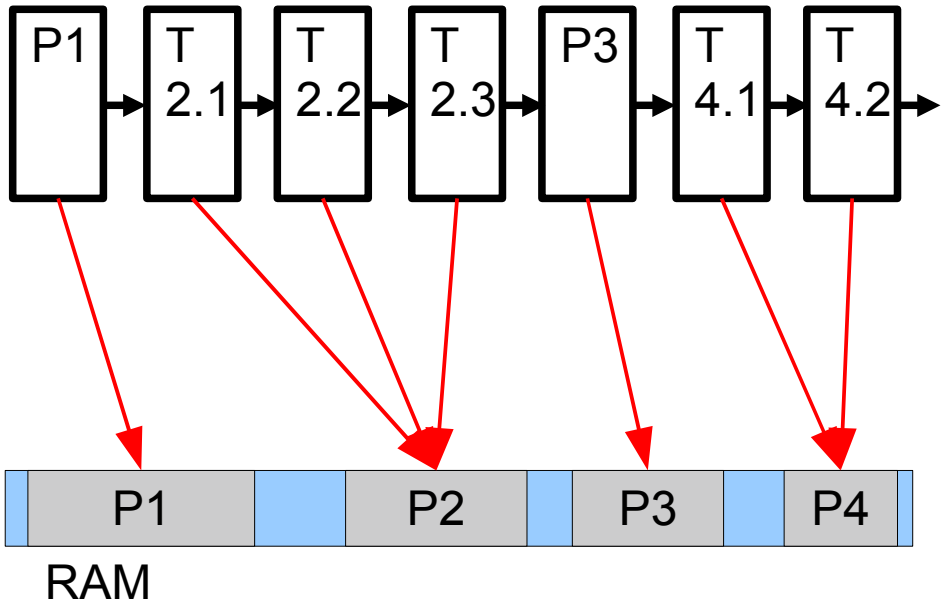
Threads im Kernel (2/3)

- 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)