2. Prozesse und Threads

Prozesse & Threads: Gliederung

Vorlesung	heute	– Einleitung – Theorie bzw. Grundlagen
Praktikum	27./28.03.	– Prozesse auf der Linux-Shell – Prozesse in C-Programmen
Vorlesung	02.04.	– Prozesse & Threads im Linux-Kernel
Praktikum	03./04.04.	– 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:

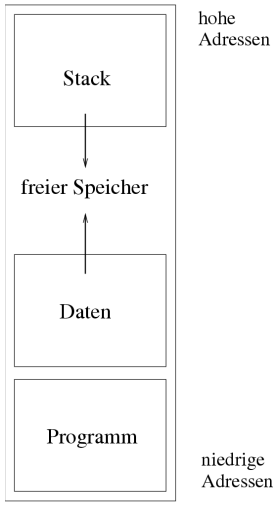
- Informationen über alle Prozesse und ihre Zustände
- Jeder Prozess hat dort einen **Process Control Block** (**PCB**):
  - Identifier (PID)
  - Registerwerte inkl. Befehlszähler
  - Speicherbereich des Prozess
  - Liste offener Dateien und Sockets
  - Informationen wie Vater-PID, letzte Aktivität,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

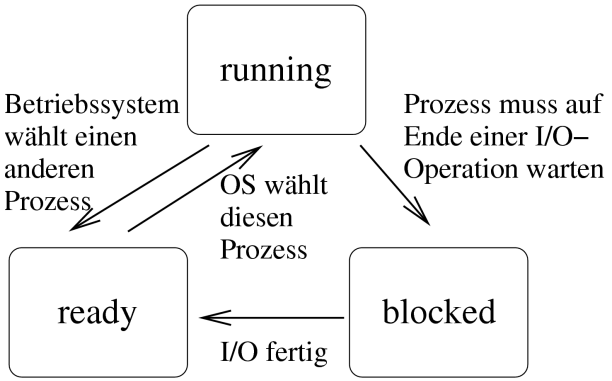
Zustände

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

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

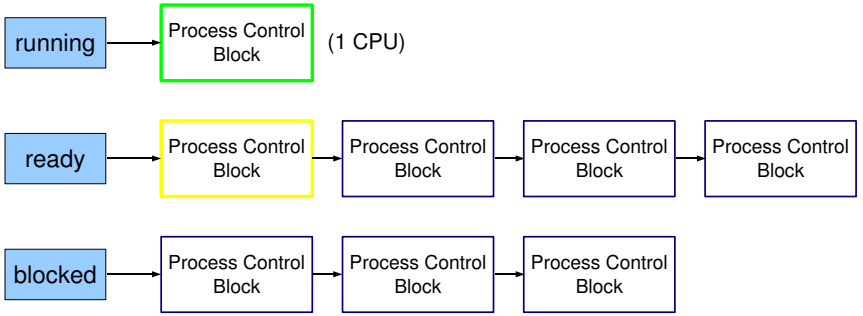
Prozesse (4)

Zustandsübergänge

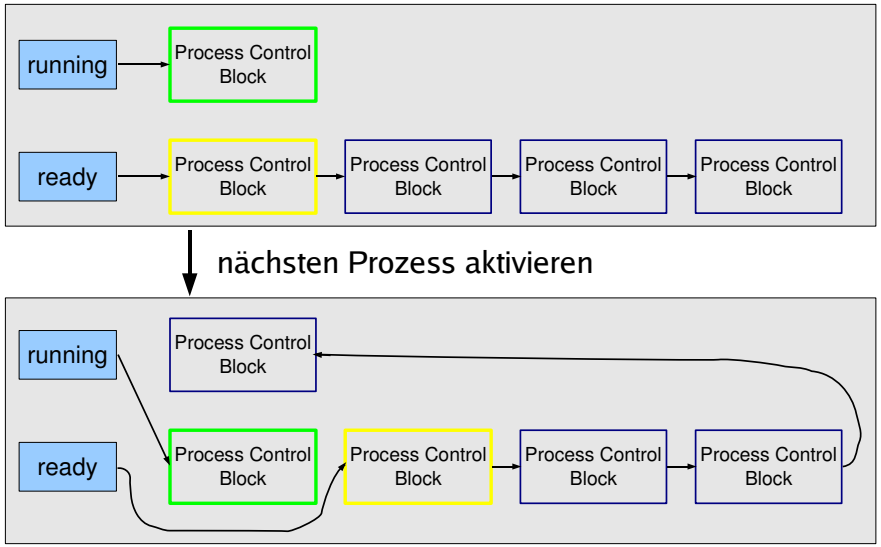


Prozesse (5)

Prozesslisten



Prozesse (6)



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

Threads (1)

Was ist ein Thread?

- Aktivitätsstrang in einem Prozess
- einer von mehreren
- Gemeinsamer Zugriff auf Daten des Prozess
- aber: Stack, Befehlszähler, Stack, 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

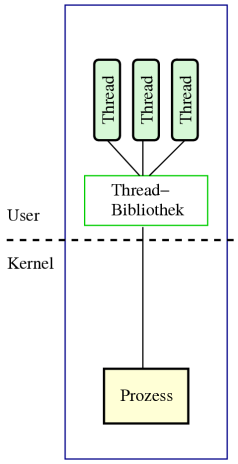
Ein Prozess, neun Threads:

```
[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

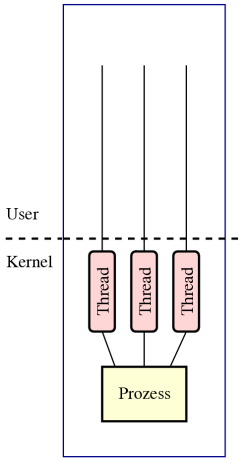
User Level Threads



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

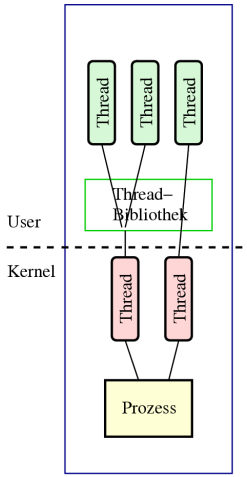
Kernel Level Threads



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

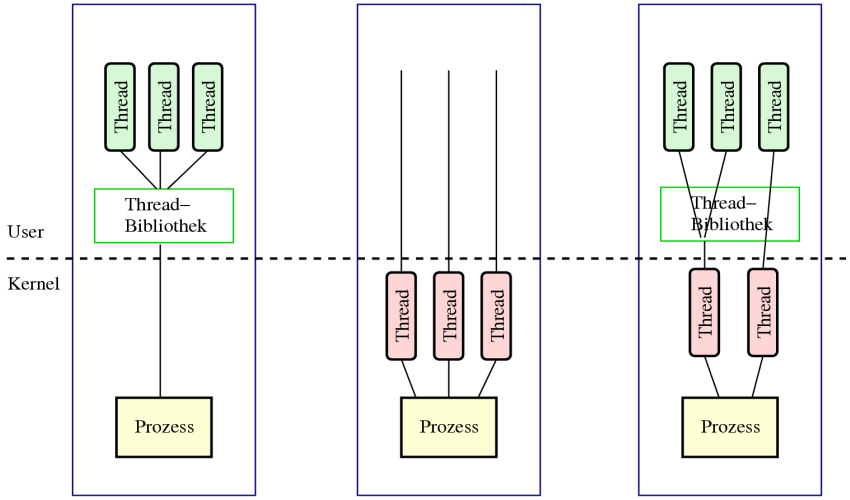
Gemischte Threads



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

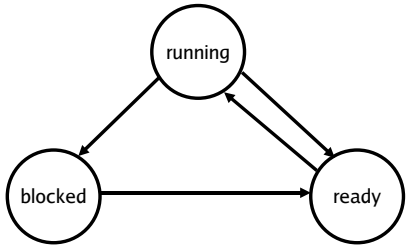
Thread-Typen, Übersicht



User Level Threads Kernel Level Threads Gemischtes Modell

Thread-Zustände

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



Praxis: Linux-Kernel

Prozessliste (1/9)

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 (2/9)

```
struct task_struct {
   volatile long state;/* -1 unrunnable, 0 runnable, >0 stopped */
```

`#define TASK_RUNNING`	0	ausführbar – läuft gerade oder wartet („ready“)
`#define TASK_INTERRUPTIBLE`	1	schläft – wird geweckt, wenn er Signal erhält oder bestimmter Zustand eintritt (Kernel)
`#define TASK_UNINTERRUPTIBLE`	2	schläft – wie oben, aber ohne Signale
`#define TASK_STOPPED`	4	läuft nicht und kann auch nicht (nach Signalen SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU)
`#define TASK_TRACED`	8	
`#define TASK_ZOMBIE`	16	terminiert, aber Vaterprozess hat noch nicht `wait()` aufgerufen.
`#define TASK_DEAD`	32	

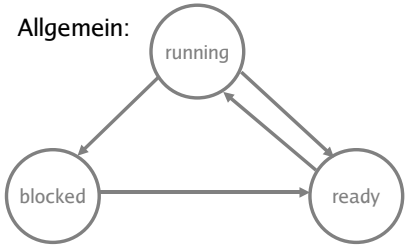
Prozessliste (3/9)

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

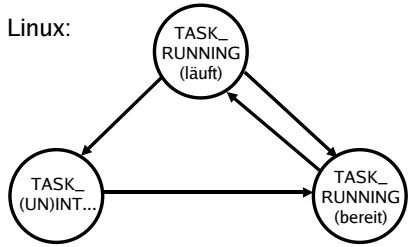
```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_TRACED             8
/* in tsk->exit_state */
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32
/* in tsk->state again */
#define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

- TASK_RUNNING: ready <u>oder</u> 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:



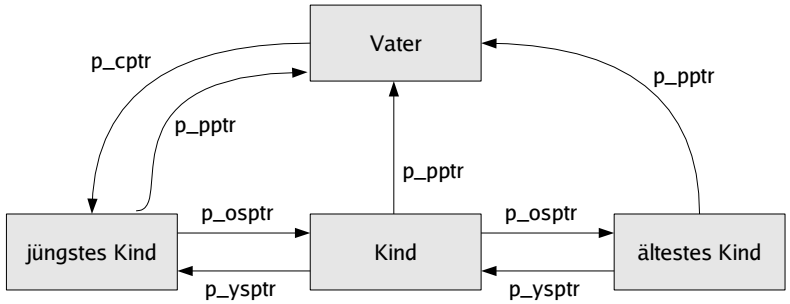
Linux:



Prozessliste (4/9)

Verwandtschaftsverhältnisse (alte Linux-Version)

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



Prozessliste (5/9)

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 (6/9)

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 (7/9)

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 (8/9)

Sessions

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

Prozessliste (9/9)

```
> 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, darin:

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

- Linux kennt keine Threads, sondern betrachtet diese als Prozesse
- Thread: Prozess, der sich mit anderen Prozessen bestimmte Ressourcen teilt
- Jeder Thread hat `task_struct` und sieht für den Kernel wie ein normaler Prozess aus
- Fundamental anders als z. B. Windows und Solaris