

Betriebssysteme 1

SS 2018

Prof. Dr.-Ing. Hans-Georg Eßer
Fachhochschule Südwestfalen

Foliensatz B:

v1.1, 2017/04/18

- Kurze Einführung in C
- Prozesse
- Threads

Einführung in C (1)

- Vorab das wichtigste:
 - keine Klassen / Objekte
 - statt Objekten: „structs“ (zusammengesetzte Datentypen)
 - statt Methoden nur Funktionen
 - zu bearbeitende Variablen immer als Argument übergeben
 - kein String-Datentyp (sondern Zeichen-Arrays)
 - häufiger Einsatz von Zeigern
 - `int main () {}` ist immer Hauptprogramm

Kurze Einführung in C

Einführung in C (2)

- Ausführlichere Informationen fürs Selbststudium: <http://www.c-howto.de/>
 - Download: Zip-Archiv
- auch als Buch für ca. 20 € erhältlich
- hier: Fokus auf Unterschiede zu C++/C#/Java

Einführung in C (3)

- Damit Sie C-Code lesen und verstehen können
→ ein paar Informationen zu
 - **Structs** (Strukturen, zusammengesetzte Typen)
 - **Pointern**
 - **Quellcode- und Header-Dateien** (Prototypen)

Strukturen und Pointer (2)

Pointer

- Deklaration mit *: `char *ch_ptr;`
 - verwalten Speicheradressen (an welchem Ort befindet sich die Variable?)
 - Operatoren
 - & (Adresse von)
 - * (Dereferenzieren)
- ```
char ch, ch2;
char *ch_ptr; char *ch_ptr2;

ch_ptr = &ch; // Adresse von ch?
ch2 = *ch_ptr; // Inhalt

ch_ptr2 = ch_ptr;
 // kopiert nur Adresse
```

## Strukturen und Pointer (1)

### Structs

- Mehrere Möglichkeiten der Deklaration

|                                                                                                                      |                                                                                                                                                 |                                                                                                                                                  |
|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>struct {   int i;   char c;   float f; } variable;  variable.i = 9; variable.c = 'a'; variable.f = 0.123;</pre> | <pre>struct mystruct {   int i;   char c;   float f; };  struct mystruct variable;  variable.i = 9; variable.c = 'a'; variable.f = 0.123;</pre> | <pre>typedef struct {   int i;   char c;   float f; } mystruct;  mystruct variable;  variable.i = 9; variable.c = 'a'; variable.f = 0.123;</pre> |
|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|

## Strukturen und Pointer (3)

- Struct und Pointer kombiniert
- Oft bei verketteten Listen

```
struct liste {
 struct liste *next;
 struct liste *prev;
 int inhalt;
};

struct liste *anfang;
struct liste *p;

for (p=anfang; p != NULL; p=p->next) {
 use (p->inhalt);
}
```

## Strukturen und Pointer (4)

- Pointer-Typen
  - `typ *ptr;`  
→ `ptr` ist ein Zeiger auf etwas vom Typ `typ`
  - `typ **pptr;`  
→ `pptr` ist ein Zeiger auf einen Zeiger vom Typ `typ`
  - `ptr` bzw. `pptr` sind Speicheradressen
  - `*ptr` gibt den Wert zurück, der an der Speicherstelle abgelegt ist, auf die `ptr` zeigt
  - analog: `**pptr` ist ein Wert, aber `*pptr` ein Zeiger

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-9

## Strukturen und Pointer (6)

- Nicht-initialisierte Pointer: schlecht
  - Beispiel:

```
int *ip;
int **ipp;

printf (ip); // nicht-init. Adresse (0)
printf (*ip); // illegal -> Abbruch

*ip = 42; // auch illegal, schreibt an
// nicht def. Adresse
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-11

## Strukturen und Pointer (5)

- Pointer-Typen
  - `&`-Operator erzeugt zu Variable einen Pointer
  - Beispiele:

```
int i;
int *ip;
int **ipp;

i = 42;
ip = &i; // ip = Adresse von i
ipp = &ip; // ipp = Adresse von ip

printf (*ip); // -> 42
printf (**ipp); // -> auch 42
```

## Strukturen und Pointer (7)

- Vorsicht bei `char* a,b,c;` etc.

```
[esser@macbookpro:tmp]$ cat t2.c
int main () {
 char* a,b;
 printf ("|a| = %d \n", sizeof(a));
 printf ("|b| = %d \n", sizeof(b));
}
```

```
[esser@macbookpro:tmp]$ gcc t2.c; ./a.out
|a| = 8
|b| = 1
```

- besser: `char *a, *b, *c;`

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-10

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-12

## Prototypen (1)

---

- Programm- und Header-Dateien
  - Header-Dateien (\*.h) enthalten Funktionsprototypen und Makrodefinitionen (aber keinen normalen Code)
  - Programmdateien (\*.c) enthalten den Code, können aber ebenfalls Prototypen und Makros enthalten (kein Zwang, eine .h-Datei zu erzeugen)

## Prototypen (3)

---

- Wie findet der Compiler die Header-Dateien?  
→ Zwei Varianten:
  - `#include "pfad/zu/datei.h"`  
Dateiname ist Pfad (relativ zu Verzeichnis mit der .c-Datei)
  - `#include <name.h>`  
name.h wird in den Standard-Include-Verzeichnissen gesucht.  
Welche sind das? Beim Bauen des gcc festgelegt...

## Prototypen (2)

---

- Funktionsprototypen (in Header-Dateien)
  - erlauben die Verwendung von Funktionen, deren Implementierung weiter unten im Programm (oder in einer anderen Datei) steht
  - Prototyp enthält nur Rückgabetyt, Name und Argumente, z. B.  

```
int summe (int x, int y);
```



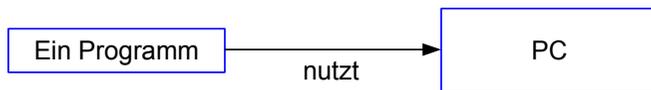
## Prozesse: Grundlagen

## Einleitung (1)

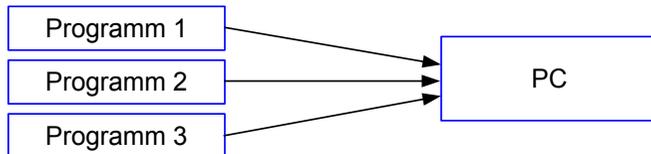
### Single-Tasking / Multitasking:

Wie viele Programme laufen „gleichzeitig“?

- MS-DOS, CP/M:



- Windows, Linux, ...:



## Einleitung (3)

### 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 (2)

### 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 (4)

### Prozess:

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

## Einleitung (5)

### 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, ...

## Einleitung (7)

### Wechsel zwischen Prozessen

- Prozess unterbrechen
  - aktuellen Zustand im PCB sichern
  - neuen Prozess auswählen
  - aus dem PCB (des neuen Prozesses) die Zustandsinformationen wiederherstellen
  - Prozess-Ausführung fortsetzen
- mehr dazu: Foliensatz D (Scheduling)

## Einleitung (6)

- Beispiel: PCBs in UNIX

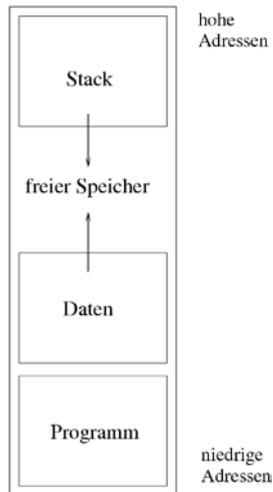
```
typedef struct {
 thread_id pid; // Prozess-ID
 thread_id ppid; // Vaterprozess
 int state; // Prozess-Status
 thread_id next, prev; // linked list
 boolean used; // Eintrag benutzt?
 int files[MAX_PFD]; // offene Dateien
 char cwd[MAX_PATH]; // Arbeitsverzeichnis
 uint16 uid, gid; // user/group ID
 uint16 euid, egid; // effective user/group ID
 uint16 ruid, grid; // real user/group ID
 // [...]
} PCB;
```

## 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): Speicher



- Daten: dynamisch erzeugt
- Stack: Verwaltung der Funktionsaufrufe
- Stack und Daten „wachsen aufeinander zu“
- Details → Foliensatz F (Speicherverwaltung)

## Prozesse (4): Speicher

- **BSS-Segment**
  - nicht initialisierte globale Variablen
  - werden bei Prozess-Start auf 0 initialisiert (landen nicht in der Objektdatei eines Programms)
  - implementiert über Copy-on-Write-Mapping auf Seite mit Nullen
- **Mapped Files** (mmap)

## Prozesse (3): Speicher

- **Text-Segment**
  - Programm-Code, String-Literale, Konstanten
  - read-only, direkt auf Programmdatei gemappt
- **Stack**
  - lokale Variablen, Funktionsrückgabewerte, Rücksprungadressen
  - wächst und schrumpft nach Bedarf
- **Data-Segment**
  - globale, initialisierte Variablen
  - **Heap**, dynamischer Prozess-Speicher
  - verwaltet Prozess über `malloc()`, `free()` etc.

## Prozesse (5): Speicher

- **Anonymous Memory Mappings**
  - für große Speicher-Anforderungen (`malloc`), die nicht auf dem Heap landen
  - `glibc` entscheidet abhängig von Größe, ob Anon-Mapping oder Heap (bis 128 K) verwendet wird
  - vermeidet Fragmentierung des Heap
  - sind schon mit 0 gefüllt

## Prozesse (6): Speicher, Beispiel-Aufteilung

```
// memtest.c
// Hans-Georg Eßer, Systemprogrammierung
#include <stdio.h> // printf
#include <unistd.h> // sleep
#include <stdlib.h> // malloc

char chararray[1024]; // BSS-Segment (nicht init.)
const int i = 7; // Text-Segment (konstant)
int j = 4095; // Data-Segment (initialisiert)

void testfunc () {
 int array[4096]; // Stack
 array[i] = 3; array[j] = 5;
 chararray[1] = 'B';
 printf ("Test: %d, %d\n", array[i], array[j]);
};

int main () {
 chararray[0] = 'A'; chararray[2] = '\0';
 testfunc ();
 printf ("Test: %s\n", chararray);
 char* s = malloc (40*1024*1024); // 40 MByte, Heap
 sleep (20);
 return 0;
};
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-29

## Prozesse (8): Zustände

### 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

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-31

```
[esser@quadamd:tmp]$ pmap $(pidof memtest)
```

Linux, 32 Bit

```
30387: ./memtest
08048000 4K r-x-- /tmp/memtest
08049000 4K r---- /tmp/memtest
0804a000 4K rw--- /tmp/memtest
b4f22000 4096K rw--- [anon]
b7724000 1496K r-x-- /lib/i386-linux-gnu/libc-2.13.so
b789a000 8K r---- /lib/i386-linux-gnu/libc-2.13.so
b789c000 4K rw--- /lib/i386-linux-gnu/libc-2.13.so
b789d000 12K rw--- [anon]
b78c5000 12K rw--- [anon]
b78c8000 4K r-x-- [anon]
b78c9000 120K r-x-- /lib/i386-linux-gnu/ld-2.13.so
b78e7000 4K r---- /lib/i386-linux-gnu/ld-2.13.so
b78e8000 4K rw--- /lib/i386-linux-gnu/ld-2.13.so
bf99a000 132K rw--- [stack]
total 42776K
```

```
[esser@quadamd:tmp]$ cat /proc/$(pidof memtest)/maps
08048000-08049000 r-xp 00000000 08:04 1156248 /tmp/memtest
08049000-0804a000 r--p 00000000 08:04 1156248 /tmp/memtest
0804a000-0804b000 rw-p 00001000 08:04 1156248 /tmp/memtest
b4f22000-b7724000 rw-p 00000000 00:00 0
b7724000-b789a000 r-xp 00000000 08:04 1966089 /lib/i386-linux-gnu/libc-2.13.so
b789a000-b789c000 r--p 00176000 08:04 1966089 /lib/i386-linux-gnu/libc-2.13.so
b789c000-b789d000 rw-p 00178000 08:04 1966089 /lib/i386-linux-gnu/libc-2.13.so
b789d000-b78a0000 rw-p 00000000 00:00 0
b78c5000-b78c8000 rw-p 00000000 00:00 0
b78c8000-b78c9000 r-xp 00000000 00:00 0 [vdso]
b78c9000-b78e7000 r-xp 00000000 08:04 9569185 /lib/i386-linux-gnu/ld-2.13.so
b78e7000-b78e8000 r--p 0001d000 08:04 9569185 /lib/i386-linux-gnu/ld-2.13.so
b78e8000-b78e9000 rw-p 0001e000 08:04 9569185 /lib/i386-linux-gnu/ld-2.13.so
bf99a000-bf99b000 rw-p 00000000 00:00 0 [stack]
```

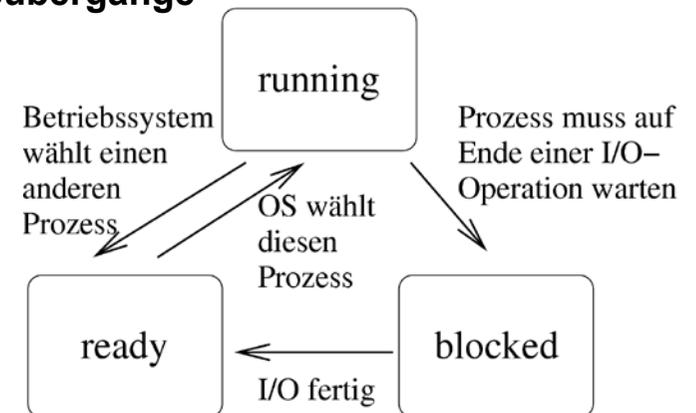
13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-30

## Prozesse (9): Zustände

### Zustandsübergänge



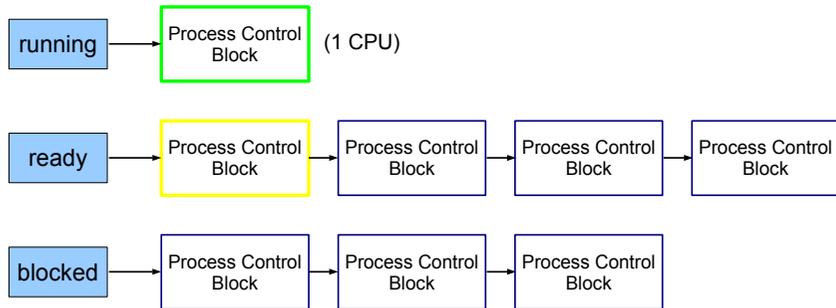
13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-32

## Prozesse (10)

### Prozesslisten



13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-33

## Prozesse (12)

### Hierarchien

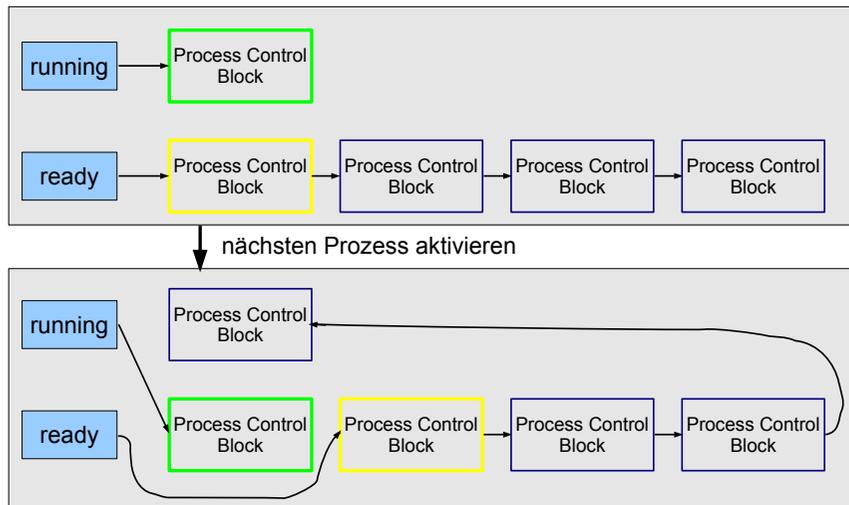
- Prozesse erzeugen einander
- Erzeuger heißt Vaterprozess (parent process), der andere Kindprozess (child process)
- Kinder sind selbständig (also: eigener Adressraum, etc.)
- Nach Prozess-Ende: Exit-Code an Vaterprozess

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-35

## Prozesse (11)



13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-34

## Prozesse: Administration

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-36

## Vorder-/Hintergrund (1)

- 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)

## Vorder-/Hintergrund (3)

- 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`

## Vorder-/Hintergrund (2)

- 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)

## Job-Verwaltung (1)

- 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 / > ...&
```

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

Kommandos

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-41

## Job-Verwaltung (4)

### 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

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-43

## Job-Verwaltung (3)

- 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 &
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-42

## Job-Verwaltung (5)

### 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`

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-44

## Job-Verwaltung (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
```

## Prozesse (1)

- 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 [esser@quadamd:~]$ ps
[1]+ Angehalten vi /tmp/test4 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

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-45

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-47

## Jobs vs. Prozesse

- 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

## Prozesse (2)

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
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-46

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-48

## Prozesse (3)

```
[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 +kcmim
esser 3174 0.0 0.0 1832 244 tty1 S Apr17 0:00 kwrapper4 ksmserver
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 13:17 /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+ Apr17 19:40 0:00 ssh backup
esser 28613 0.0 0.1 11744 7264 pts/5 Ss+ Apr17 22:45 0:00 -bash
esser 29091 0.0 0.1 12156 7704 pts/0 Ss Apr17 22:58 0:00 -bash
esser 29307 0.0 0.0 5856 1172 pts/0 R+ Apr17 23:42 0:00 ps auw
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-49

## Prozesse (5)

- 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

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-51

## Prozesse (4)

- 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)

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-50

## Prozesse (6): pstree

- Darstellung der Prozessliste auch in Baumansicht möglich: `pstree`
- Jeder Prozess hat einen Vaterprozess
- identische Teilbäume nur 1x
- Option `-p`: Prozess-IDs anzeigen

```
[esser@quadamd:~]$ pstree
init--NetworkManager--dhclient
|-acpid
|-akonadi_control--2*[akonadi_contact]
| |--3*[akonadi_ical_re]
| |--akonadi_maildir
| |--akonadi_maildis
| |--akonadi_nepomuk
| |--akonadi_vcard_r
| |--akonadi_server--mysqld--23*[{mysqld}]
| |--15*[{akonadiserver}]
| |--3*[{akonadi_contro}]
|-atd
|-avahi-daemon---avahi-daemon
|-console-kit-dae---64*[{console-kit-da}]
|-cron
|-cupsd
[...]
|-knotify4---6*[{knotify4}]
|-konsole--2*[bash---sudo---su---bash]
| |--bash---ssh
| |--bash---opera--operapluginlea
| | |--operapluginwrap---gtk-gnash
| | |--6*[{opera}]
| |--2*[{konsole}]
|-krunner---11*[{krunner}]
|-kuiserver
|-kwalletd
|-login---bash---startx---xinit--.xinitrc---kwrapper4
| |--Xorg
|-upstart-socket-
|-upstart-udev-br
|-vpngagentd
|-wpa_supplicant
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-52

## Hang-up, No Hang-up

- 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

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-53

## nohup

- `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
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-54

## top (1)

- 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   | krunker       |
| 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   | ksftirqd/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 |

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-55

## top (2)

- 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
```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-56

## free, uptime

- 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
```

## Prozess-Priorität (2)

- `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
```

## Prozess-Priorität (1)

- 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

## Prozess-Priorität (3)

- 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
```

## Prozess-Priorität (4)

- 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:~]$ _
```

## Prozess-Priorität (5)

- 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 (Motivation)

## Motivation (1)

- Szenario: Bankzentrale + Geldautomaten
- Automaten sind mit Zentrale verbunden



## Motivation (2)

- Zentrale führt Server-Anwendung aus, vereinfacht:

```

init (accounts[]); // Konten-DB

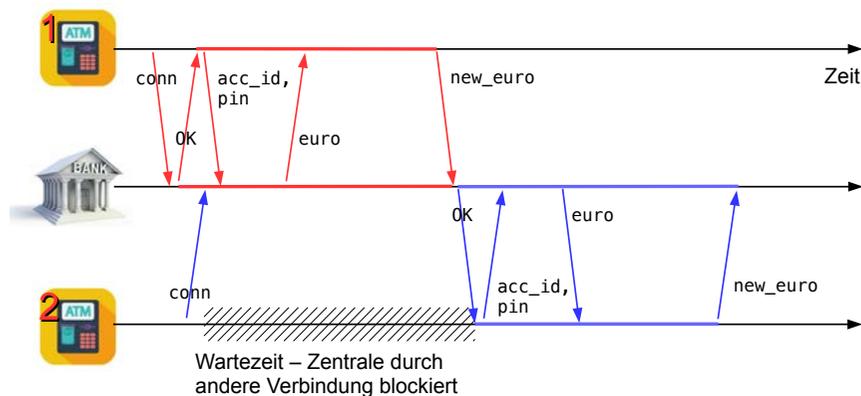
do forever {
 conn = accept_atm_connection ();
 acc_no = recv (conn);
 pin = recv (conn);
 if (!auth(acc_no,pin)) { terminate (conn); continue; }

 euro = accounts[acc_no].euro; // Abfrage
 send (conn, euro);
 recv (conn, new_euro);
 accounts[acc_no].euro = new_euro; // Aktualisierung
 terminate (conn);
}

```

## Motivation (3)

- Verbindung eines Automaten blockiert Server

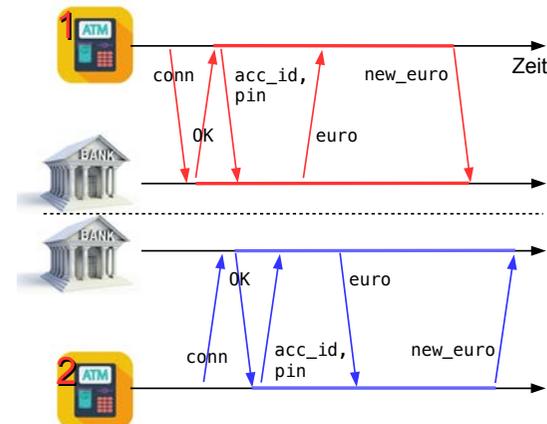


## Motivation (4)

- Wartezeiten verkürzen → parallele Verbindung mehrerer Automaten zulassen
- Erste Idee: mehrere Server-Prozesse, Anfrage an einen freien Server-Prozess weiter reichen

## Motivation (5)

- Mehrere Server-Prozesse



- Problem: Prozess-Daten (DB) doppelt → Abstimmung der Prozesse erforderlich

## Threads (Theorie)

## Threads (2)

### Warum Threads?

- Multi-Prozessor- bzw. Multi-Core-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 (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 (Kontext) separat pro Thread
- Scheduler verwaltet auch Threads – oder nicht (→ Kernel- oder User-Level-Threads)

## 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 (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

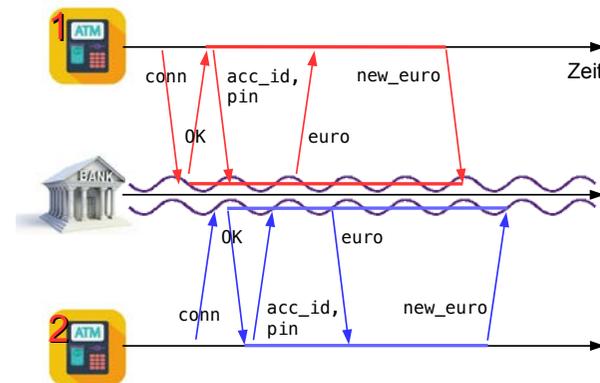
## Threads (5): Beispiele

### 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 (7): Geldautomaten

- Server-Prozess mit mehreren Threads (~~~~)



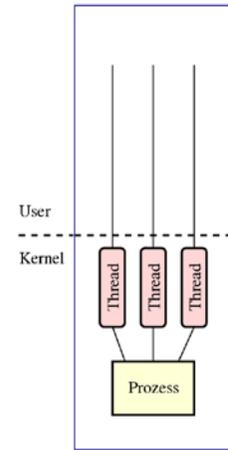
## Threads (8): Geldautomaten

```
function main () {
 init (accounts[]); // Konten-DB
 do forever {
 conn = accept_atm_connection ();
 create_thread (handler, conn);
 }
}

function handler (int conn) {
 acc_no = recv (conn);
 pin = recv (conn);
 if (!auth(acc_no,pin)) { terminate (conn); return; }

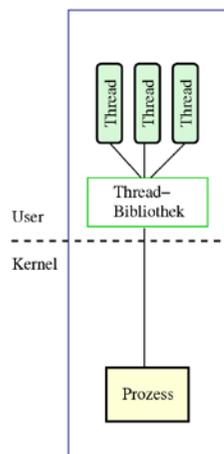
 euro = accounts[acc_no].euro; // Abfrage
 send (conn, euro);
 recv (conn, new_euro);
 accounts[acc_no].euro = new_euro; // Aktualisierung
 terminate (conn);
}
```

## Kernel Level Threads



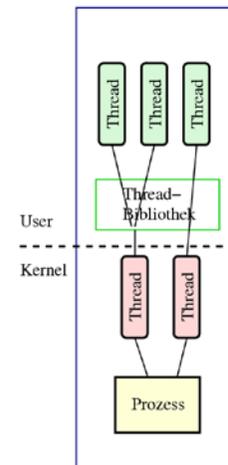
- BS kennt Threads
- BS verwaltet die Threads:
  - Erzeugen, Zerstören
  - Scheduling
- I/O eines Threads blockiert nicht die übrigen
- Aufwendig: Context Switch zwischen Threads ähnlich komplex wie bei Prozessen

## User Level Threads



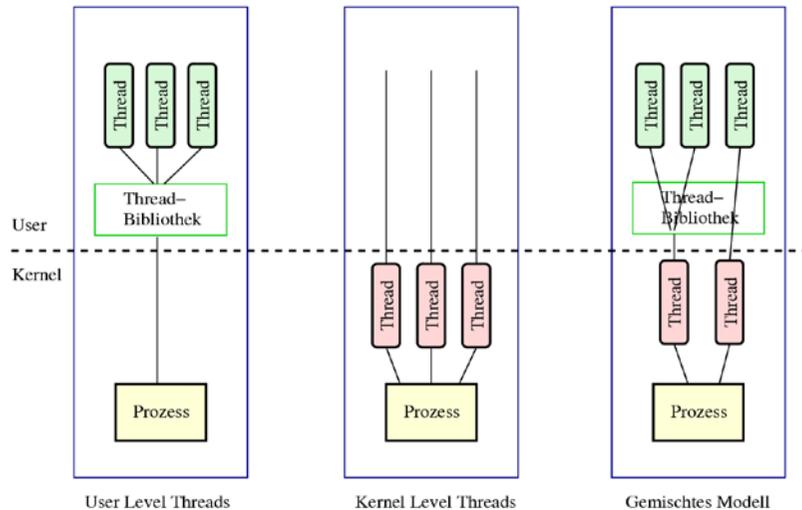
- BS kennt kein Thread-Konzept, verwaltet nur Prozesse
- Programm bindet Thread-Bibliothek ein, zuständig für:
  - Erzeugen, Zerstören
  - Scheduling
- Wenn ein Thread wegen I/O wartet, dann der ganze Prozess
- Ansonsten sehr effizient

## Gemischte Threads



- 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



13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-81

## Prozesse und Threads (Praxis, Entwicklersicht)

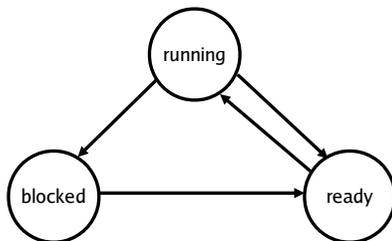
13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-83

## Thread-Zustände

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



## Prozesse und Threads erzeugen (1/11)

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);
 }
}

```

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-82

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-84

## Prozesse und Threads erzeugen (2/11)

Anderes Programm starten: `fork` + `exec`

```
main() {
 int pid=fork(); /* Sohnprozess erzeugen */
 if (pid == 0) {
 /* Sohn startet externes Programm */
 exec1("/usr/bin/gedit", "/etc/fstab", (char *) 0);
 }
 else {
 printf("Es sollte jetzt ein Editor starten...\n");
 }
}
```

Anderer Betriebssysteme oft nur: „spawn“

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

## Prozesse und Threads erzeugen (4/11)

Wirklich mehrere Prozesse:

Nach `fork` ( ) zwei Prozesse in der Prozessliste

```
> ps tree | 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 (3/11)

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 (5/11)

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 (6/11)

- 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 (8/11)

```
#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();
 }

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

 pthread_join(mythread1, NULL);
 pthread_join(mythread2, NULL);

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

 printf("Thread 1 ist weg\n");
 printf("Thread 2 ist weg\n");

 exit(0);
}
```

## Prozesse und Threads erzeugen (7/11)

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 (9/11)

### 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 (10/11)

Prozess mit mehreren Threads:

- Nur ein Eintrag in normaler Prozessliste
- Status: „I“, 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 S1+ 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
```

## 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)

## Prozesse und Threads erzeugen (11/11)

**Unterschiedliche Semantik:**

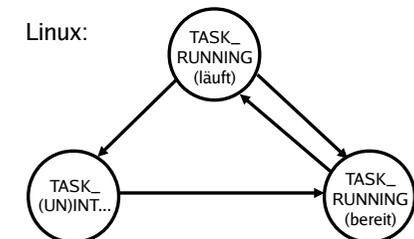
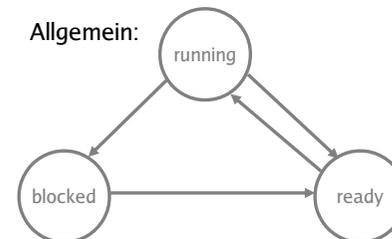
- Prozess erzeugen mit `fork ()`
  - erzeugt zwei (fast) identische Prozesse,
  - beide Prozesse setzen Ausführung an gleicher Stelle fort (nach Rückkehr aus `fork`-Aufruf)
- Thread erzeugen mit `pthread_create (... , funktion, ...)`
  - erzeugt neuen Thread, der in die angeg. Funktion springt
  - erzeugender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

## 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 (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“

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-101

## Threads im Kernel (1/2)

- 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

13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-103

## 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
```

13.04.2018

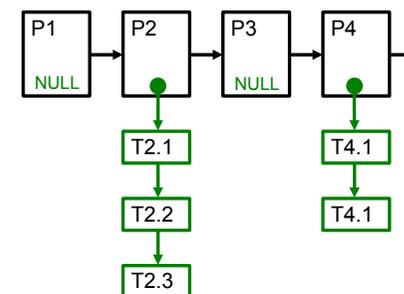
Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-102

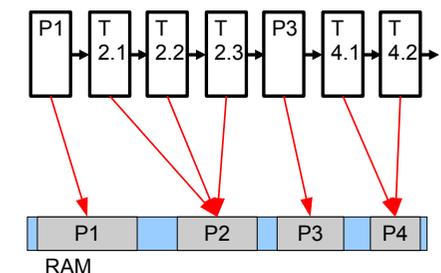
## Threads im Kernel (2/2)

Fundamental anders als z. B. Windows und Solaris

Modell 1:  
reine Prozesslisten



Modell 2 (Linux):  
Prozesse + Threads gemischt



13.04.2018

Betriebssysteme 1, SS 2018, Hans-Georg Eßer

Folie B-104