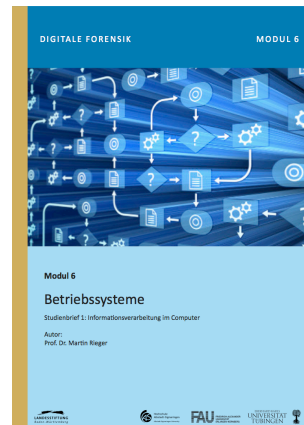


Betriebssysteme

SS 2012

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

SB 2 (15.04.2012)
Prozesse, Prozesssynchronisation und
Prozesskommunikation



2.3.1 Interrupts

Übersicht

- Motivation: Interrupts vs. Polling
- Interrupts unter Linux: „top half“, „bottom half“

Studienbrief 2

- Interrupts
- Prozesse und Threads (nur Linux)
- Scheduling (Prozesszuteilungsstrategien)
- Interprozesskommunikation (IPC)
- Dateisystem /proc (Linux)

Motivation (1)

- Festplattenzugriff ca. um Faktor 1.000.000 langsamer als Ausführen einer CPU-Anweisung
- Naiver Ansatz für Plattenzugriff:

```
naiv () {
    rechne (500 ZE);
    sende_anfrage_an (disk);
    antwort = false;
    while ( ! antwort ) {
        /* diese Schleife rechnet 1.000.000 ZE lang */
        antwort = test_ob_fertig (disk);
    }
    rechne (500 ZE);
    return 0;
}
```

Motivation (2)

- Naiver Ansatz heißt „Pollen“: in Dauerschleife ständig wiederholte Geräteabfrage

- Pollen verbraucht sehr viel Rechenzeit:



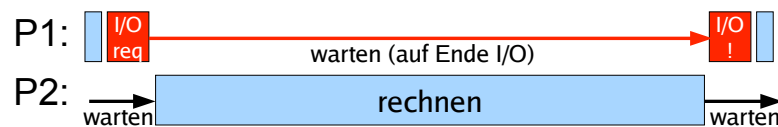
- Besser wäre es, in der Wartezeit etwas anderes zu tun
- Auch bei Parallelbearbeitung mehrerer Prozesse: Polling immer noch ungünstig

Motivation (4)

- Lösung: Interrupts – bestimmte Ereignisse können den „normalen“ Ablauf unterbrechen
- Nach jeder ausgeführten CPU-Anweisung prüfen, ob es einen Interrupt gibt

Motivation (3)

- Idee: Prozess, der I/O-Anfrage gestartet hat, solange schlafen legen, bis die Anfrage bearbeitet ist – in der Zwischenzeit was anderes tun



- Woher weiß das System,
 - wann die Anfrage bearbeitet ist, also
 - wann der Prozess weiterarbeiten kann?

Interrupt-Klassen

- **I/O (Eingabe/Ausgabe, asynchr. Interrupts)**
Meldung vom I/O-Controller: „Aktion ist abgeschlossen“
- **Timer**
- **Hardware-Fehler**
Stromausfall, RAM-Paritätsfehler
- **Software-Interrupts (Exceptions, Traps, synchrone Interrupts)**
Falscher Speicherzugriff, Division durch 0, unbekannte CPU-Instruktion, ...

Interrupts: Vor- und Nachteile

Vorteile

- **Effizienz**
I/O-Zugriff sehr langsam → sehr lange Wartezeiten, wenn Prozesse warten, bis I/O abgeschlossen ist
- **Programmierlogik**
Nicht immer wieder Gerätestatus abfragen (Polling), sondern abwarten, bis passender Interrupt kommt

Nachteile

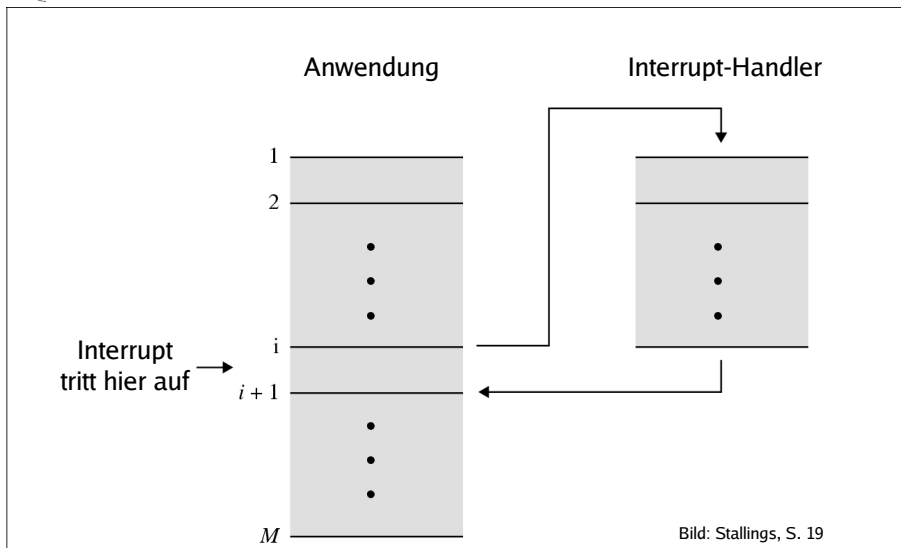
- **Mehraufwand**
Kommunikation mit Hardware wird komplexer, Instruction Cycle erhält zusätzlichen Schritt

Interrupt-Bearbeitung (2)

Grundsätzlich

- Interrupt tritt auf
- Laufender Prozess wird (nach aktuellem Befehl) unterbrochen, BS übernimmt Kontrolle
- BS speichert Daten des Prozesses (wie bei Prozesswechsel → Scheduler)
- BS ruft Interrupt-Handler auf
- Danach: Scheduler wählt Prozess aus, der weiterarbeiten darf (z. B. den unterbrochenen)

Interrupt-Bearbeitung (1)



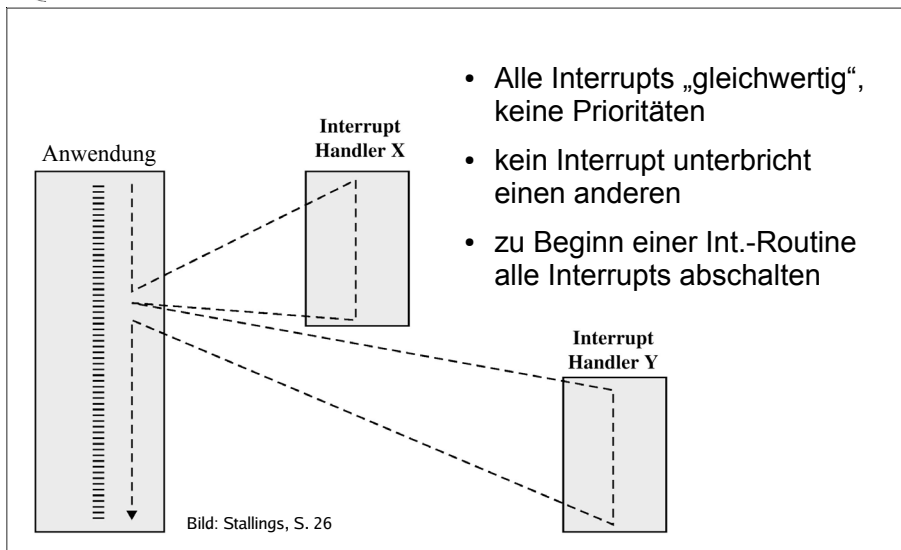
Interrupt-Bearbeitung (3)

Was tun bei Mehrfach-Interrupts?

Drei Möglichkeiten

- Während Abarbeitung eines Interrupts alle weiteren ausschließen (DI, disable interrupts)
→ Interrupt-Warteschlange
- Während Abarbeitung andere Interrupts zulassen
- Interrupt-Prioritäten: Nur Interrupts mit höherer Priorität unterbrechen solche mit niedrigerer

Mehrfach-Interrupts (1)



15.04.2012

Modul 6: Betriebssysteme, SS 2012, Hans-Georg Eßer

Folie B-13

I/O-lastig vs. CPU-lastig (1)

• CPU-lastiger Prozess

- Prozess benötigt überwiegend CPU-Rechenzeit und vergleichsweise wenig I/O-Operationen
- Längere Rechenphasen werden nur gelegentlich durch I/O-Wartezeiten unterbrochen

• I/O-lastiger Prozess

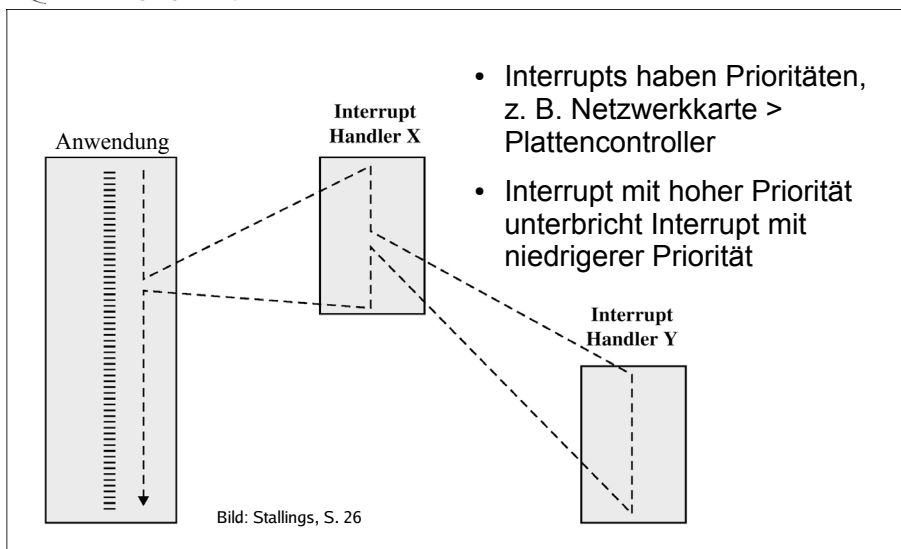
- Prozess führt viele I/O-Operationen durch und benötigt vergleichsweise wenig Rechenzeit
- Sehr kurze Rechenphasen wechseln sich mit häufigen Wartezeiten auf I/O ab

15.04.2012

Modul 6: Betriebssysteme, SS 2012, Hans-Georg Eßer

Folie B-15

Mehrfach-Interrupts (2)



15.04.2012

Modul 6: Betriebssysteme, SS 2012, Hans-Georg Eßer

Folie B-14

I/O-lastig vs. CPU-lastig (2)

Multitasking und Interrupts

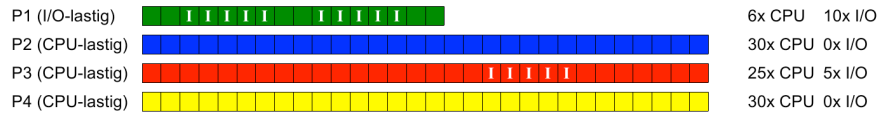
- Multitasking verbessert CPU-Nutzung:
 - I/O-lastiger Prozess wartet auf I/O-Events,
 - CPU-lastiger Prozess rechnet weiter
- Prozess stößt I/O-Operation an und legt sich schlafen (wartet auf Signal)
- optimale Performance: gute Mischung I/O- und CPU-lastiger Prozesse

15.04.2012

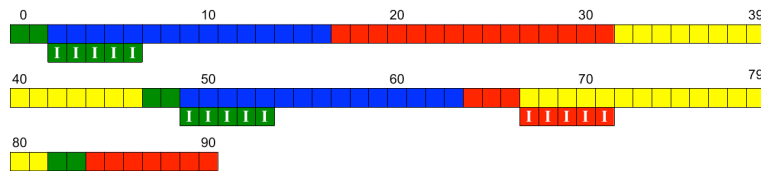
Modul 6: Betriebssysteme, SS 2012, Hans-Georg Eßer

Folie B-16

I/O-lastig vs. CPU-lastig (3)



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

Moderne Maschine mit vier Cores

```
[esser@quad:~]$ cat /proc/interrupts
CPU0      CPU1      CPU2      CPU3
0:         5224          3          1          1  IO-APIC-edge      timer
1:       298114         774         793         793  IO-APIC-edge      i8042
3:           9          8          6          9  IO-APIC-edge
4:           8          9          8          6  IO-APIC-edge
8:           0          0          0          1  IO-APIC-edge      rtc0
9:           0          0          0          0  IO-APIC-fasteoi   acpi
12:       3070145      16539      16542      16485  IO-APIC-edge      i8042
16:       2760924         881         904         886  IO-APIC-fasteoi   uhci_hcd:usb1, nvidia
18:       24122388        6538        6698        6647  IO-APIC-fasteoi   ehci_hcd:usb6, uhci_hcd:usb7
19:          281          28          27          10  IO-APIC-fasteoi   uhci_hcd:usb3, uhci_hcd:usb5
21:       22790          0          0          0  IO-APIC-fasteoi   uhci_hcd:usb2
22:       7786588      10464141      8251870      8439964  IO-APIC-fasteoi   HDA Intel
23:          899          0          1          1  IO-APIC-fasteoi   uhci_hcd:usb4, ehci_hcd:usb8
221:       9519152      10751650      9745810      10326363  PCI-MSI-edge      eth0
222:       14462926        38205        38095        38178  PCI-MSI-edge      ahci
NMI:         0          0          0          0  Non-maskable interrupts
LOC:       724999305      786034088      748693018      748218173  Local timer interrupts
RES:       5334382        3576152        3464671        3357556  Rescheduling interrupts
CAL:       2111668        4233550        4067655        3871450  function call interrupts
TLB:       101757        113319        88752        107777  TLB shootdowns
TRM:         0          0          0          0  Thermal event interrupts
SPU:         0          0          0          0  Spurious interrupts
ERR:         0
MIS:         0
```

Interrupts unter Linux

```
[esser@server ~]$ cat /proc/interrupts
CPU0
0: 3353946487      XT-PIC timer
2: 0              XT-PIC cascade
3: 4663           XT-PIC NVidia CK804
5: 159275991      XT-PIC ohci1394, nvidia
7: 971775         XT-PIC hsfpcibasic2
8: 2              XT-PIC rtc
9: 0              XT-PIC acpi
10: 31052         XT-PIC libata, ohci_hcd
11: 197906977     XT-PIC libata, ehci_hcd
12: 16904921      XT-PIC eth0
14: 60349322     XT-PIC ide0
NMI: 0
LOC: 0
ERR: 0
MIS: 0
```

Interrupt Handler (1)

Für jedes Gerät:

- Interrupt Request (IRQ) Line
- Interrupt Handler (Interrupt Service Routine, ISR) → Teil des Gerätetreibers
- C-Funktion
- läuft in speziellem Context (Interrupt Context)
- „top half“ und „bottom half“

Interrupt Handler (2)

„top half“ und „bottom half“

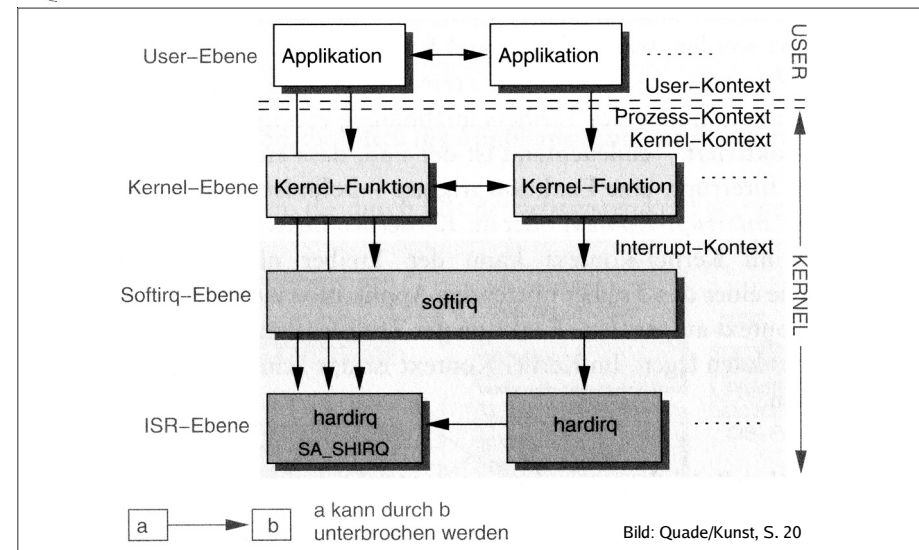
top half

- Interrupt handler
- startet sofort, erledigt zeitkritische Dinge
- bestätigt (der Hardware) den Erhalt des Interrupts, setzt Gerät zurück etc.
- Alles andere → bottom half

bottom half

- startet später, macht die eigentliche Arbeit

Interrupt Handler (4)



Interrupt Handler (3)

Wichtig: In welchem Context läuft was?

- **User Context:** unterbrechbar (HW oder SW interrupts), kann system calls aufrufen,
- **Process Context:** nach Software Interrupt aus User Context, läuft im Kernel, Daten zwischen Kernel- und Prozessspeicher übertragen, nur durch HW-Interrupt unterbrechbar
- **Kernel Context:** Funktionen des Kernels, kein Datenaustausch zwischen Kernel- und User-Space, nur durch HW-Interrupt unterbrechbar
- **Interrupt Context:** Software- und Hardware-Interrupts

Interrupt Handler (5)

Top und bottom half / Tasklet

Bottom half heißt im Linux-Kernel (seit Version 2.6) Tasklet

- Interrupt Service Routine (top half) erledigt das Wichtigste (zeitkritische Dinge), erzeugt Tasklet und beendet sich – dabei sind Interrupts gesperrt
- Tasklets führen längere Berechnungen durch, die zur Interrupt-Verarbeitung gehören – dabei sind Interrupts zugelassen

Tasklets

- Tasklet ist kein Prozess (struct tasklet_struct), läuft direkt im Kernel; im Interrupt-Context
- Zwei Prioritäten:
 - *tasklet_hi_schedule*: startet direkt nach ISR
 - *tasklet_schedule*: startet erst, wenn kein anderer Soft IRQ mehr anliegt

Übersicht

- Prozesse unter Linux (fork, exec, wait)
- Threads unter Linux (pthread)
- Prozesse und Threads aus Linux-Kernel-Sicht

Mehr Informationen:

- [1] Linux Kernel 2.4 Internals, Kapitel 2,
http://www.faqs.org/docs/kernel_2_4/lki-2.html
- [2] J. Quade, E.-K. Kunst: „Linux-Treiber entwickeln“, dpunkt-Verlag,
<http://ezs.kr.hsnr.de/TreiberBuch/html/>

Neuer Prozess: fork ()

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

Anderes Programm starten: fork + exec

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

Anderer Betriebssysteme oft nur: „spawn“

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

→ oder CreateProcess(), siehe Skript

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
```

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 */
    }
}
```

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 (6/11)

- 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("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, ..)`

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    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
```

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)

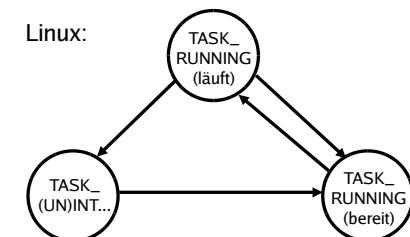
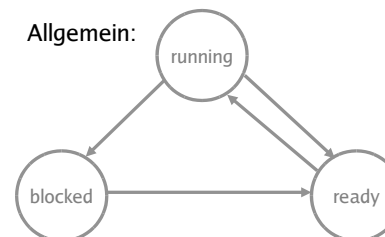
Unterschiedliche Semantik:

- 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

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

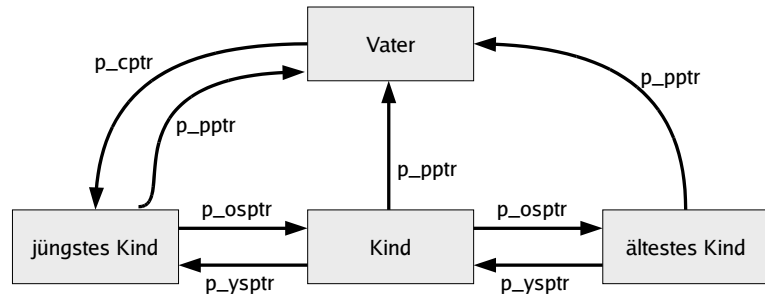
```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_STOPPED 4
#define TASK_TRACED 8
/* in tsk->exit_state */
#define EXIT_ZOMBIE 16
#define EXIT_DEAD 32
/* in tsk->state again */
#define TASK_NONINTERACTIVE 64
#define TASK_DEAD 128
```

- `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



Verwandtschaftsverhältnisse (alte Linux-Version)

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



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`

Verwandtschaftsverhältnisse (neue Linux-Version)

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

Zugriff auf alle Kinder:

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

Vom aktuellen Pfad durch den Prozessbaum bis zu `init`:

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

Prozessgruppen

- 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“

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

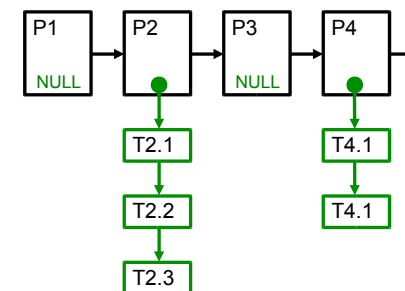
Prozessliste (8/8)

```
> 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
```

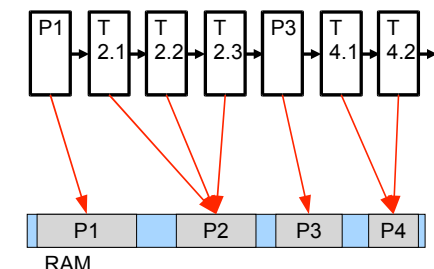
Threads im Kernel (2/2)

- Fundamental anders als z. B. Windows und Solaris

Modell 1:
reine Prozesslisten



Modell 2 (Linux):
Prozesse + Threads gemischt



2.3.3 Scheduling

Übersicht

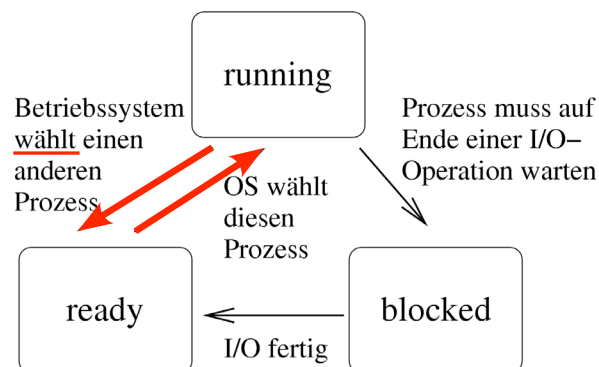
- preemptive / non-preemptive scheduling
- First Come First Served (FCFS)
- Round-Robin-Scheduler (RR)
- Virtual Round Robin (VRR)
-

Wann wird Scheduler aktiv?

- Neuer Prozess entsteht (`fork`)
- Aktiver Prozess blockiert wegen I/O-Zugriff
- Blockierter Prozess wird bereit
- Aktiver Prozess endet (`exit`)
- Prozess rechnet schon zu lange
- Interrupt tritt auf

Scheduling: Prozess auswählen

Zustandsübergänge



Unterbrechendes Scheduling

Prozess-Unterbrechung möglich?

- **Kooperatives (non-preemptive) Scheduling:**
 - Prozess rechnet solange, wie er will; bis zum nächsten I/O-Aufruf oder bis `exit()`
 - Scheduler wird nur bei Prozess-Blockieren oder freiwilliger CPU-Aufgabe aktiv
- **Präemptives (unterbrechendes) Scheduling:**
 - Timer aktiviert regelmäßig Scheduler, der neu entscheiden kann, „wo es weiter geht“

Häufige Prozesswechsel?

Faktoren

- **Zeit für Kontext-Switch:** Scheduler benötigt Zeit, um Prozesszustand zu sichern
→ verlorene Rechenzeit
- **Wartezeit der Prozesse:** Häufigere Wechsel erzeugen stärkeren Eindruck von Gleichzeitigkeit

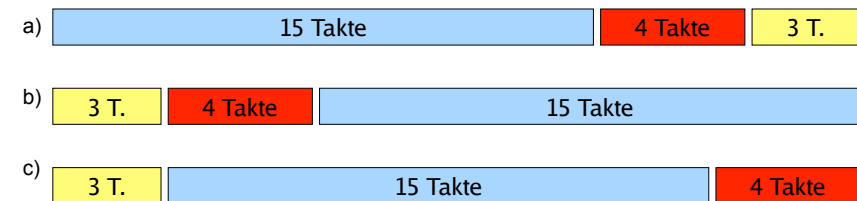
FCFS-Beispiel

Drei Prozesse mit
Rechendauern

T1: 15 Takte
T2: 4 Takte
T3: 3 Takte

Durchschnittliche
Ausführdauer:

- a) $(15+19+22) / 3 = 18,67$
 b) $(3+7+22) / 3 = 10,67$
 c) $(3+18+22) / 3 = 14,33$



First Come, First Served (FCFS)

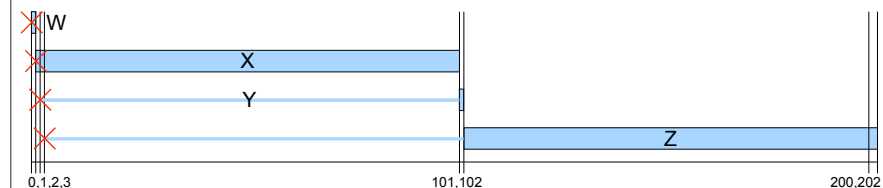
Einfache Warteschlange

- Neue Prozesse reihen sich in Warteschlange ein
- Scheduler wählt jeweils nächsten Prozess in der Warteschlange
- Prozess arbeitet, bis er fertig ist (kooperatives Scheduling)

FCFS: Gut für lange Prozesse

- FCFS bevorzugt lang laufende Prozesse
- Beispiel: 4 Prozesse W, X, Y, Z

Prozess	Ankunftszeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit-Ankunftszeit)	T_r/T_s
W	0	1	0	1	1	1,00
X	1	100	1	101	100	1,00
Y	2	1	101	102	100	100,00
Z	3	100	102	202	199	1,99

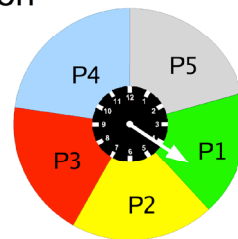


FCFS bevorzugt CPU-lastige Prozesse

- Während CPU-lastiger Prozess läuft, müssen alle anderen Prozesse warten
- I/O-lastiger Prozess kommt irgendwann dran, läuft nur sehr kurz und muss sich dann wieder hinten anstellen
- Ineffiziente Nutzung sowohl der CPU als auch der I/O-Geräte

- Blockierten Prozess, der wieder bereit wird, hinten in Warteschlange einreihen

- Wie FCFS – aber mit Unterbrechungen
- Alle bereiten Prozesse in einer Warteschlange
- Jedem Thread eine Zeitscheibe (quantum, time slice) zuordnen
- Ist Prozess bei Ablauf der Zeitscheibe noch aktiv, dann:
 - Prozess verdrängen (preemption), also in den Zustand „bereit“ versetzen
 - Prozess ans Ende der Warteschlange hängen
 - Nächsten Prozess aus Warteschlange aktivieren



Kontrollaufgabe 2.7

Angenommen ein Rechnersystem braucht $1 \mu\text{s}$, um einen Kontextwechsel durchzuführen. Würde das Zeitquantum auf $T = 1 \mu\text{s}$ gesetzt, so würde sich ein Overhead von 100% ergeben. Wäre $T = 100 \mu\text{s}$, so hätte man einen Overhead von 1%. Schlagen Sie einen sinnvollen Wert für T vor und begründen Sie Ihre Antwort.

K

- Sinnvolle Quantenlänge hängt nicht nur von der Zeit ab, die für einen Kontextwechsel nötig ist
- mehr dazu: folgende Folien

Quantenlänge

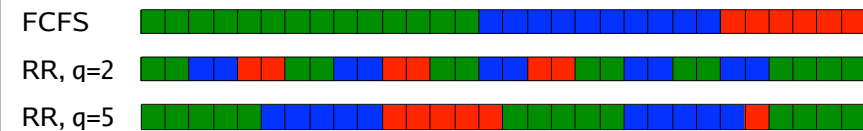
Kriterien für Wahl des Quantums:

- Größe muss in Verhältnis zur Dauer eines Context Switch stehen
- Großes Quantum: evtl. lange Verzögerungen
- Kleines Quantum: kurze Antwortzeiten, aber Overhead durch häufigen Context Switch

Round-Robin-Beispiel

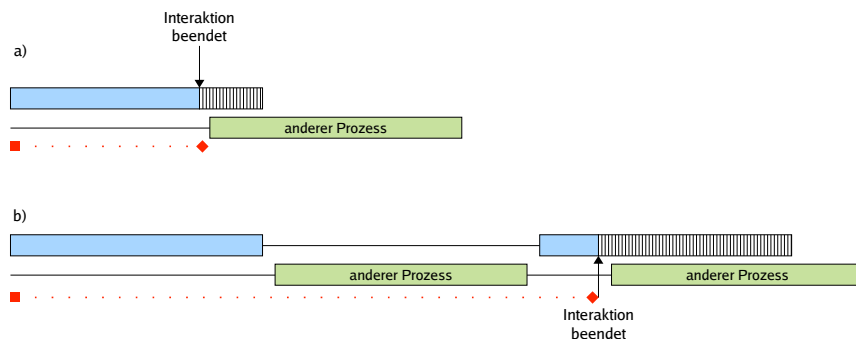
Szenario: Drei Prozesse

- FCFS (einfache Warteschlange, keine Unterbrechung)
- Round Robin mit Quantum 2
- Round Robin mit Quantum 5



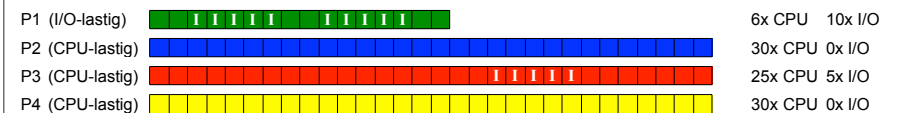
Quantenlänge

- Oft: Quantum q etwas größer als typische Zeit, die das Bearbeiten einer Interaktion benötigt

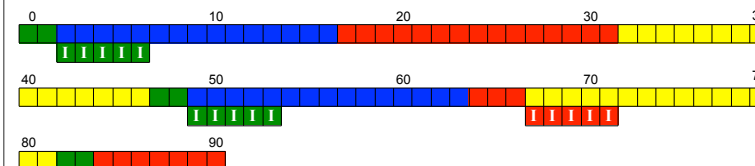


Round Robin: I/O- vs. CPU-lastig

Idealer Verlauf (wenn jeder Prozess exklusiv läuft)



Ausführreihenfolge mit Round Robin, Zeitquantum 15:



Prozess	CPU-Zeit	I/O-Zeit	Summe	Laufzeit	Wartezeit *)
P1	6	10	16	84	68
P2	30	0	30	64	34
P3	25	5	30	91	61
P4	30	0	30	82	52

*) im Zustand bereit, nicht blockiert!

Virtual Round Robin (1)

Beobachtung:

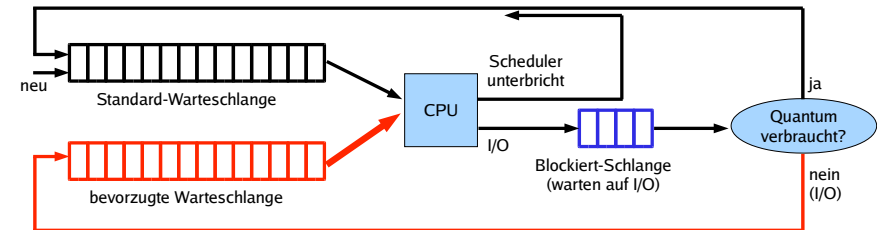
- Round Robin unfair gegenüber I/O-lastigen Prozessen:
 - CPU-lastige nutzen ganzes Quantum,
 - I/O-lastige nur einen Bruchteil

Lösungsvorschlag:

- Idee: Nicht verbrauchten Quantum-Teil als „Guthaben“ des Prozesses merken
- Sobald blockierter Prozess wieder bereit ist (I/O-Ergebnis da): Restguthaben sofort aufbrauchen

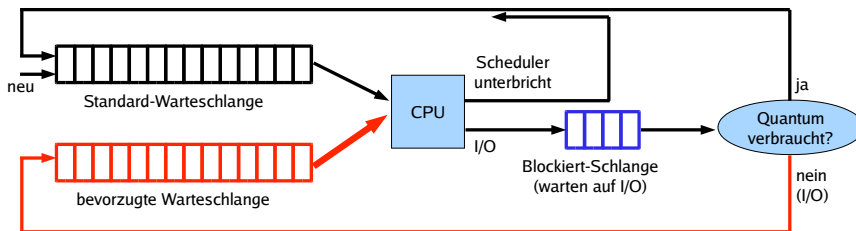
Virtual Round Robin (3)

- Scheduler bevorzugt Prozesse in Zusatzschlange
- Quantum für diesen Prozess: $q-u$ (kriegt nur das, was ihm „zusteht“, was er beim letzten Mal nicht verbraucht hat)



Virtual Round Robin (2)

- Prozesse, die Zeitquantum verbrauchen, wie bei normalem Round Robin behandeln: zurück in Warteschlange
- Prozesse, die wegen I/O blockieren und nur Zeit $u < q$ ihres Quantums verbraucht haben, bei Blockieren in Zusatzwarteschlange stecken



Übung zu Round Robin

K

Kontrollaufgabe 2.5

Die Prozesse A, B, und C werden nacheinander in eine Warteschlange eingereiht, aus der die lafbereiten Prozesse nach dem Round-Robin-Verfahren mit gleicher Priorität behandelt werden.

Die Länge einer Zeitscheibe ist $\Delta t = 0.1$ s.

Die Gesamtdauer der Prozesse ist $t_A = 0,3$ s, $t_B = 0,3$ s und $t_C = 0,5$ s.

Zunächst sind alle Prozesse lafbereit. Der Prozess B ist nach seinem 2. Durchlauf für 0.2 s blockiert der Prozess A ist nach seinem 2. Durchlauf für 0.3 s blockiert.

Stellen Sie dar, zu welchen Zeitscheiben welche Prozesse bearbeitet werden.

Anmerkung: Nicht lafbereite Prozesse werden aus der Warteschlange herausgenommen. Sobald ein nicht lafbereiter Prozess wieder in den Zustand bereit wechselt, wird er hinten in der Warteschlange eingereiht.

Restlaufzeit: A: 0.3, B: 0.3, C: 0.5

t=0.0: Auswahl A (Rest: 0.2)

t=0.1: Auswahl B (Rest: 0.2)

t=0.2: Auswahl C (Rest: 0.4)

t=0.3: Auswahl A (Rest: 0.1; erst ab 0.7 bereit)

t=0.4: Auswahl B (Rest: 0.1, erst ab 0.7 bereit)

t=0.5: Auswahl C (Rest: 0.3)

t=0.6: Auswahl C (Rest 0.2; A und B waren noch nicht bereit)

t=0.7: Auswahl A (A danach fertig)

t=0.8: Auswahl B (B danach fertig)

t=0.9: Auswahl C (Rest 0.1)

t=1.0: Auswahl C (danach fertig)

t=1.1: Warteschlange leer

K

Kontrollaufgabe 2.6

Wenn bei einem Scheduler mit Round-Robin-Strategie jeder Prozess pro Ausführungseinheit ein Zeitquantum erhält, wie lange würde der letzte Prozess in der Warteschlange mit n wartenden Prozessen warten müssen, bis er mit seiner Ausführungseinheit beginnt?

- Der letzte Prozess muss $(n-1)$ Quanten lang warten, weil vor ihm $n-1$ Prozesse jeweils ein Quantum lang rechnen.

- Es gibt Prozesse (oder Threads oder Kernel-Funktionen) mit gemeinsamem Zugriff auf bestimmte Daten, z. B.
 - Threads des gleichen Prozesses: gemeinsamer Speicher
 - Prozesse mit gemeinsamer Memory-Mapped-Datei
 - Prozesse / Threads öffnen die gleiche Datei zum Lesen / Schreiben
 - SMP-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

Interprozesskommunikation (IPC)

- Synchronisationsprobleme
- Wechselseitiger Ausschluss (mutual exclusion)
- Semaphore
- Signale (Unix)

- Synchronisation: Probleme mit „gleichzeitigem“ Zugriff auf Datenstrukturen
- Beispiel: Zwei Threads erhöhen einen Zähler

```

erhoehe_zaebler()   Ausgangssituation: w=10
{
  w=read(Adresse);  T1:                               T2:
  w=w+1;             w=read(Adresse); // 10
  write(Adresse,w); w=w+1; -- -- -- // 11
}
                                     w=read(Adresse); // 10
                                     w=w+1;           // 11
                                     write(Adresse,w); // 11
                                     write(Adresse,w); // 11 !!
    
```

Ergebnis nach P1, P2: $w=11$ – nicht 12!

Einführung (3)

- Gewünscht wäre eine der folgenden Reihenfolgen:

Ausgangssituation: w=10

```
P1:      P2:
w=read(Adr); // 10
w=w+1;    // 11
write(Adr,w); // 11
-----
w=read(Adr); // 11
w=w+1;    // 12
write(Adr,w); // 12
```

Ergebnis nach P1, P2: w=12

Ausgangssituation: w=10

```
P1:      P2:
w=read(Adr); // 10
w=w+1;    // 11
write(Adr,w); // 11
-----
w=read(Adr); // 11
w=w+1;    // 12
write(Adr,w); // 12
```

Ergebnis nach P1, P2: w=12

Einführung (5)

- Analoges Problem bei Datenbanken:

```
exec sql CONNECT ...
exec sql SELECT kontostand INTO $var FROM KONTO
      WHERE kontonummer = $knr
$var = $var - abhebung
exec sql UPDATE Konto SET kontostand = $var
      WHERE kontonummer = $knr
exec sql DISCONNECT
```

Bei parallelem Zugriff auf gleichen Datensatz kann es zu Fehlern kommen

- Definition der (Datenbank-) **Transaktion**, die u. a. **atomar und isoliert** erfolgen muss

Einführung (4)

- Ursache: `erhoehe_zaebler()` arbeitet nicht **atomar**:
 - Scheduler kann die Funktion unterbrechen
 - Funktion kann auf mehreren CPUs gleichzeitig laufen
- Lösung: Finde alle Code-Teile, die auf gemeinsame Daten zugreifen, und stelle sicher, dass immer nur ein Prozess auf diese Daten zugreift (gegenseitiger Ausschluss, mutual exclusion)

Einführung (6)

Race Condition:

- Mehrere parallele Threads / Prozesse nutzen eine gemeinsame Ressource
- Zustand hängt von Reihenfolge der Ausführung ab
- Race: die Threads liefern sich „ein Rennen“ um den ersten / schnellsten Zugriff

Warum Race Conditions vermeiden?

- Ergebnisse von parallelen Berechnungen sind nicht eindeutig (d. h. potenziell falsch)
- Bei Programmtests könnte (durch Zufall) immer eine „korrekte“ Ausführreihenfolge auftreten; später beim Praxiseinsatz dann aber gelegentlich eine „falsche“.
- Race Conditions sind auch Sicherheitslücken

- Idee: Zugriff via Lock auf einen Prozess (Thread, ...) beschränken:

```
erhoehe_zaebler( ) {  
    flag=read(Lock);  
    if (flag == LOCK_UNSET) {  
        set(Lock);  
        /* Anfang des „kritischen Bereichs“ */  
        w=read(Adresse);  
        w=w+1;  
        write(Adresse,w);  
        /* Ende des „kritischen Bereichs“ */  
        release(Lock);  
    }  
}
```

- Problem: Lock-Variable nicht geschützt

Race Condition als Sicherheitslücke

- Wird von Angreifern genutzt
- Einfaches Beispiel: Primitive Shell

```
for (;;) {  
    read(command);  
    f = creat ("/tmp/script"); // Datei erzeugen  
    write (f,command); // Befehl rein schreiben  
    f.close (); // speichern/schließen  
    chmod ("/tmp/script","a+x"); // ausführbar machen  
    system ("/tmp/script"); // Skript ausführen  
}
```

Annahme: Dateisystem ohne Zugriffsrechte (z. B. VFAT)

Angreifer ändert Dateiinhalt vor dem chmod; Programm läuft mit Rechten des Opfers

- Nicht alle Zugriffe sind problematisch:
 - Gleichzeitiges Lesen von Daten stört nicht
 - Prozesse, die „disjunkt“ sind (d. h.: die keine gemeinsamen Daten haben) können ohne Schutz zugreifen
- Sobald mehrere Prozesse/Threads/... gemeinsam auf ein Objekt zugreifen – und mindestens einer davon schreibend –, ist das Verhalten des Gesamtsystems **unvorhersehbar und nicht reproduzierbar.**

Kritische Bereiche (1)

- Programmteil, der auf gemeinsame Daten zugreift
 - Müssen nicht verschiedene Programme sein: auch mehrere Instanzen des gleichen Programms!
- Block zwischen erstem und letztem Zugriff
- Nicht den Code schützen, sondern die Daten
- Formulierung: kritischen Bereich „betreten“ und „verlassen“ (enter / leave critical section)

Kritische Bereiche (3)

- Anforderung an parallele Threads:
 - Es darf maximal ein Thread gleichzeitig im kritischen Bereich sein
 - Kein Thread, der außerhalb kritischer Bereiche ist, darf einen anderen blockieren
 - Kein Thread soll ewig auf das Betreten eines kritischen Bereichs warten
 - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen kritischen Bereichen und blockieren sich gegenseitig)

Kritische Bereiche (2)

- Bestimmen des kritischen Bereichs nicht ganz eindeutig:

```
void test () {  
    z = global[i];  
    z = z + 1;  
    global[i] = z;  
    // was anderes tun  
    z = global[j];  
    z = z - 1;  
    global[j] = z;  
}
```

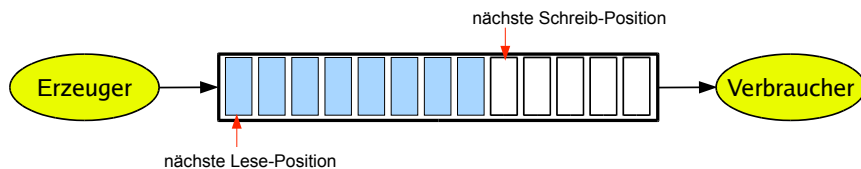
- zwei kritische Bereiche oder nur einer?

Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: **mutual exclusion**, kurz: **mutex**)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmierfehlern

Erzeuger-Verbraucher-Problem (1)

- Beim **Erzeuger-Verbraucher-Problem** (producer consumer problem, bounded buffer problem) gibt es zwei kooperierende Threads:
 - Der Erzeuger speichert Informationen in einem **beschränkten Puffer**.
 - Der Verbraucher liest Informationen aus diesem Puffer.



Semaphore (1)

Ein **Semaphor** ist eine Integer- (Zähler-) Variable, die man wie folgt verwendet:

- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
- Beim **Anfordern** eines Semaphors (**P-** oder **Wait-Operation**): P = (niederl.) probeer
 - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
 - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

Erzeuger-Verbraucher-Problem (2)

- **Synchronisation**
 - **Puffer nicht überfüllen:**
Wenn der Puffer voll ist, muss der Erzeuger warten, bis der Verbraucher eine Information aus dem Puffer abgeholt hat, und erst dann weiter arbeiten.
 - **Nicht aus leerem Puffer lesen:**
Wenn der Puffer leer ist, muss der Verbraucher warten, bis der Erzeuger eine Information im Puffer abgelegt hat, und erst dann weiter arbeiten.

Semaphore (2)

- Bei **Freigabe** eines Semaphors (**V-** oder **Signal-Operation**): V = (niederl.) vrijgeven
 - einen Thread aus der Warteschlange wecken, falls diese nicht leer ist,
 - Semaphor-Wert um 1 erhöhen (wenn es keinen auf den Semaphor wartenden Thread gibt)
- Code sieht dann immer so aus:


```

P (&sem);
/* Code, der die Ressource nutzt */
V (&sem);
            
```
- in vielen Büchern: `wait(&sem)`, `signal(&sem)`

Semaphore (3)

- Variante: Negative Semaphore-Werte
 - Semaphore zählt Anzahl der wartenden Threads
 - **Anfordern** (P, WAIT):
 - Semaphore-Wert um 1 erniedrigen
 - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphore-Wert ≤ 0 ist.
 - **Freigabe** (V, SIGNAL):
 - Thread aus der Warteschlange wecken (falls nicht leer)
 - Semaphore-Wert um 1 erhöhen

Mutexe (1)

- **Mutex**: boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert
 - true: Zugang erlaubt
 - false: Zugang verboten
- **blockierend**: Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert \rightarrow Warteschlange
- Bei Freigabe:
 - Warteschlange enthält Threads \rightarrow einen wecken
 - Warteschlange leer: Mutex auf true setzen

Semaphore (4)

- Pseudo-Code für Semaphore-Operationen

```

P (sem) {
    if (sem > 0)
        sem--;
    else
        BLOCK_CALLER;
}

V (sem) {
    if (P in QUEUE(sem)) {
        wakeup (P);
        remove (P, QUEUE);
    }
    else
        sem++;
}
    
```

Mutexe (2)

- **Mutex (mutual exclusion) = binärer Semaphore**, also ein Semaphore, der nur die Werte 0 / 1 annehmen kann. Pseudo-Code:

```

wait (mutex) {
    if (mutex == 1)
        mutex = 0;
    else
        BLOCK_CALLER;
}

signal (mutex) {
    if (P in QUEUE(mutex)) {
        wakeup (P);
        remove (P, QUEUE);
    }
    else
        mutex = 1;
}
    
```

- Neue Interpretation: wait \rightarrow lock
signal \rightarrow unlock
- Mutexe für exklusiven Zugriff (kritische Bereiche)

Blockieren?

- Betriebssysteme können Mutexe und Semaphore **blockierend** oder **nicht-blockierend** implementieren
- blockierend:
wenn der Versuch, den Zähler zu erniedrigen, scheitert
→ warten
- nicht blockierend:
wenn der Versuch scheitert
→ vielleicht etwas anderes tun

Warteschlangen

- Mutexe / Semaphore verwalten Warteschlangen (der Prozesse, die schlafen gelegt wurden)
- Beim Aufruf von signal() muss evtl. ein Prozess geweckt werden
- Auswahl des zu weckenden Prozesses ist ein ähnliches Problem wie die Prozess-Auswahl im Scheduler
 - FIFO: **starker** Semaphor / Mutex
 - zufällig: **schwacher** Semaphor / Mutex

Atomare Operationen

- Bei Mutexen / Semaphore müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:

Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

Erzeuger-Verbraucher-Problem mit Semaphore und Mutexen

```
typedef int semaphore;
semaphore mutex = 1;           // Kontrolliert Zugriff auf Puffer
semaphore empty = N;          // Zählt freie Plätze im Puffer
semaphore full = 0;           // Zählt belegte Plätze im Puffer

producer() {
    while (TRUE) {             // Endlosschleife
        produce_item(item);    // Erzeuge etwas für den Puffer
        wait (empty);          // Leere Plätze dekrementieren bzw. blockieren
        wait (mutex);          // Eintritt in den kritischen Bereich
        enter_item (item);     // In den Puffer einstellen
        signal (mutex);        // Kritischen Bereich verlassen
        signal (full);         // Belegte Plätze erhöhen, evtl. consumer wecken
    }
}

consumer() {
    while (TRUE) {             // Endlosschleife
        wait (full);           // Belegte Plätze dekrementieren bzw. blockieren
        wait (mutex);          // Eintritt in den kritischen Bereich
        remove_item(item);     // Aus dem Puffer entnehmen
        signal (mutex);        // Kritischen Bereich verlassen
        signal (empty);        // Freie Plätze erhöhen, evtl. producer wecken
        consume_entry (item);  // Verbrauchen
    }
}
```

2.4.1.2 Signale

- Linux: Signale mit
 - `kill -sig pid` (Shell) oder
 - `kill (pid, sig)` (C)an einen Prozess schicken
- Prozess erhält Signal und springt in Signal-Handler (ähnelt einem Interrupt-Handler, aber prozess-lokal)

2.5 Das /proc-Dateisystem

- Linux bietet über `/proc` Zugriff auf wichtige Eigenschaften aller laufenden Prozesse
 - Für jeden Prozess gibt es einen Ordner `/proc/PID/` (PID = Prozess-ID)
- Übungen

2.4.1.2 Signale

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`