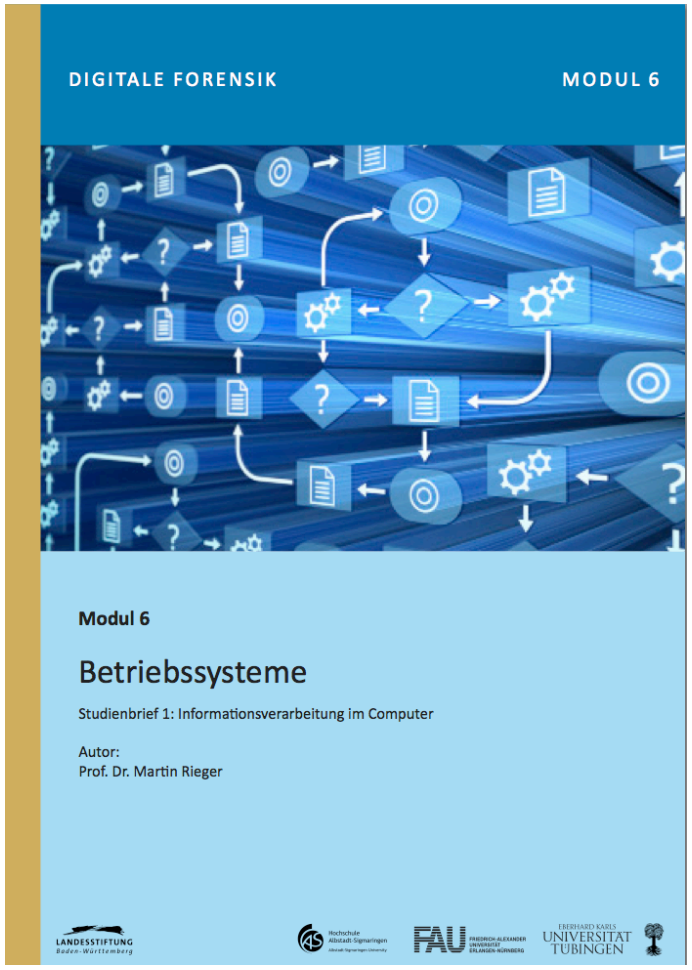
Betriebssysteme

SS 2013

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

SB 2 (14.03.2013)
Prozesse, Prozesssynchronisation und Prozesskommunikation



Studienbrief 2

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

2.3.4 Prozesse/Threads unter Unix

Übersicht

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

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);
  }
}
```

Prozesse und Threads erzeugen (2/11)

Anderes Programm starten: `fork + exec`

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

Andere Betriebssysteme oft nur: „spawn“

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

→ oder `CreateProcess()`, siehe Skript

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 (4/11)

Wirklich mehrere Prozesse:

Nach `fork ()` zwei Prozesse in der Prozessliste

```
> pstree | grep simple
    ... -bash---simplefork---simplefork

> ps w | grep simple
25684 pts/16 S+         0:00 ./simplefork
25685 pts/16 S+         0:00 ./simplefork
```

Prozesse und Threads erzeugen (5/11)

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 (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 (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();
 }
 sleep(5);

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

 sleep(5);

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

 if ( pthread_join ( mythread1, NULL ) ) {
  printf("Fehler beim Join.");
  abort();
 }

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

 if ( pthread_join ( mythread2, NULL ) ) {
  printf("Fehler beim Join.");
  abort();
 }

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

 exit(0);

}
```

Prozesse und Threads erzeugen (9/11)

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: „l“, multi-threaded
- Über `ps -eLf` Thread-Informationen
  - NLWP: Number of light weight processes
  - LWP: Thread ID

```
> ps auxw | grep thread
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    12022  0.0  0.0  17976   436 pts/15   Sl+  22:58   0:00
./thread

> ps -eLf | grep thread
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
esser    12166  4031 12166  0    3 23:01 pts/15   00:00:00 ./thread1
esser    12166  4031 12167  0    3 23:01 pts/15   00:00:00 ./thread1
esser    12166  4031 12177  0    3 23:01 pts/15   00:00:00 ./thread1
```

Prozesse und Threads erzeugen (11/11)

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

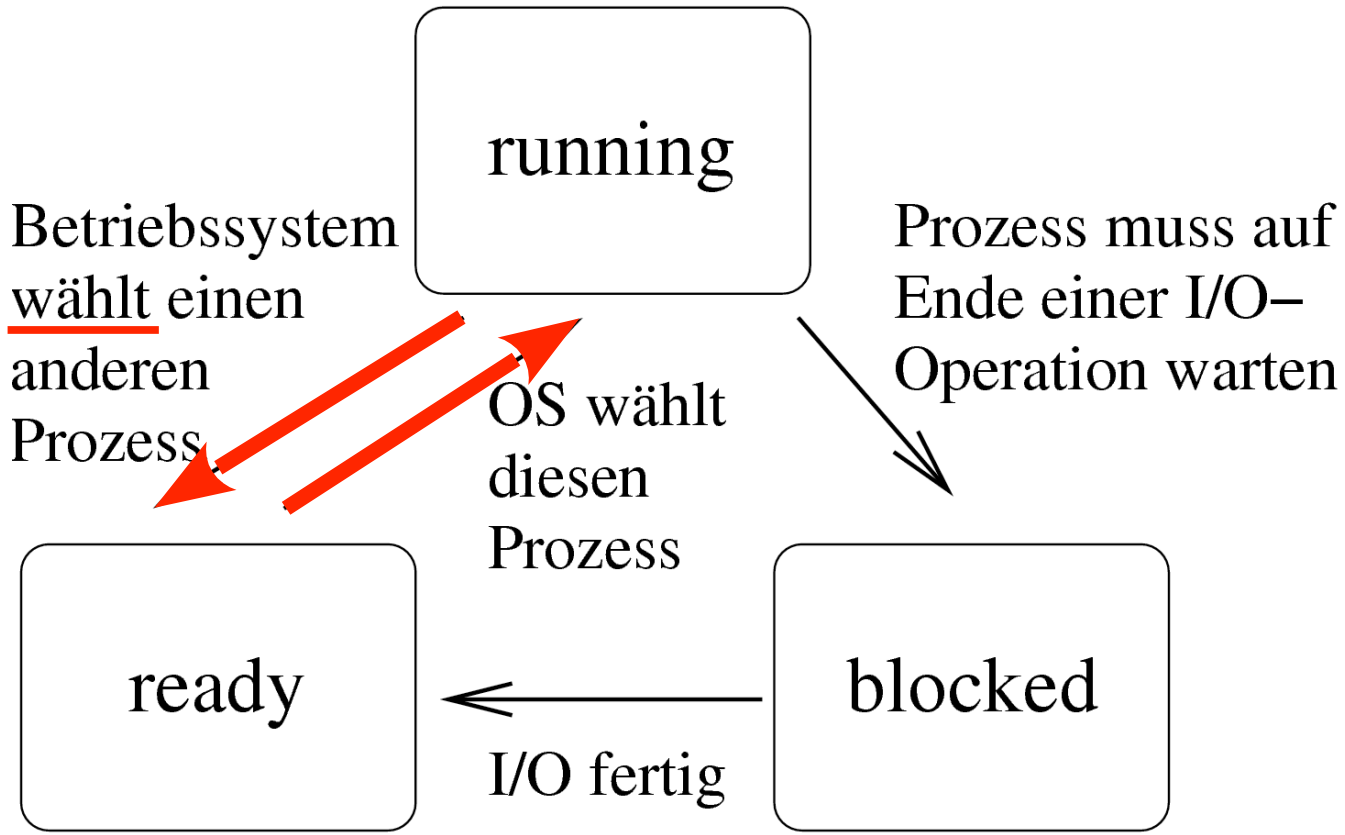
2.3.3 Scheduling

Übersicht

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

Scheduling: Prozess auswählen

Zustandsübergänge



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

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

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-Beispiel

Drei Prozesse mit Rechendauern

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

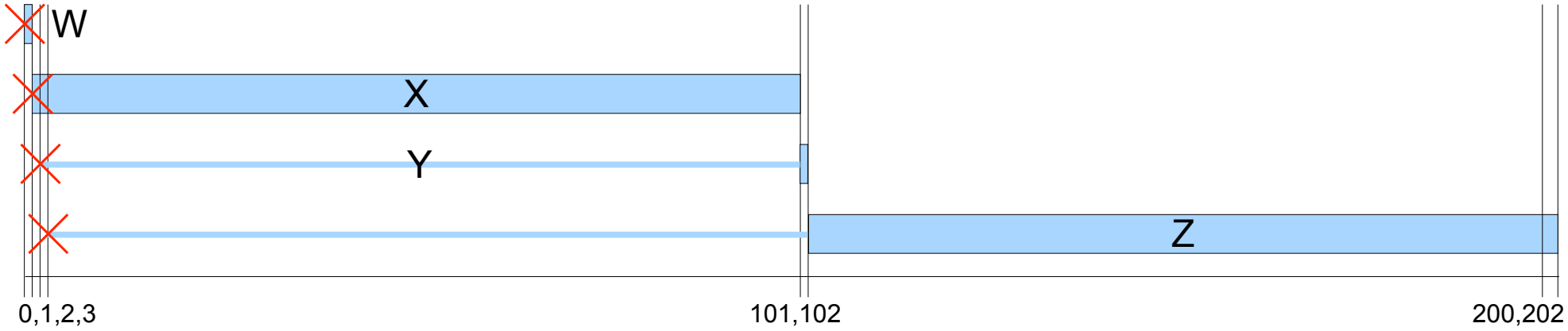
Durchschnittliche Ausführdauer:

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

FCFS: Gut für lange Prozesse

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

Prozess	Ankunftzeit	Service Time T_s (Rechenzeit)	Startzeit	Endzeit	Turnaround T_r (Endzeit-Ankunftzeit)	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



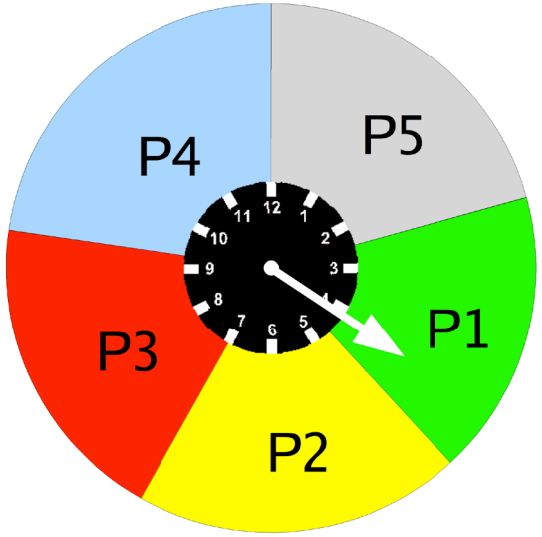
FCFS: CPU- vs. I/O-lastige Prozesse

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

Round Robin / Time Slicing (1)

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



Round Robin (2)

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

Quantenlänge

Kontrollaufgabe 2.7

Angenommen ein Rechnersystem braucht 1 µs, um einen Kontextwechsel durchzuführen. Würde das Zeitquantum auf T = 1 µs gesetzt, so würde sich ein Overhead von 100% ergeben. Wäre T = 100 µs, 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

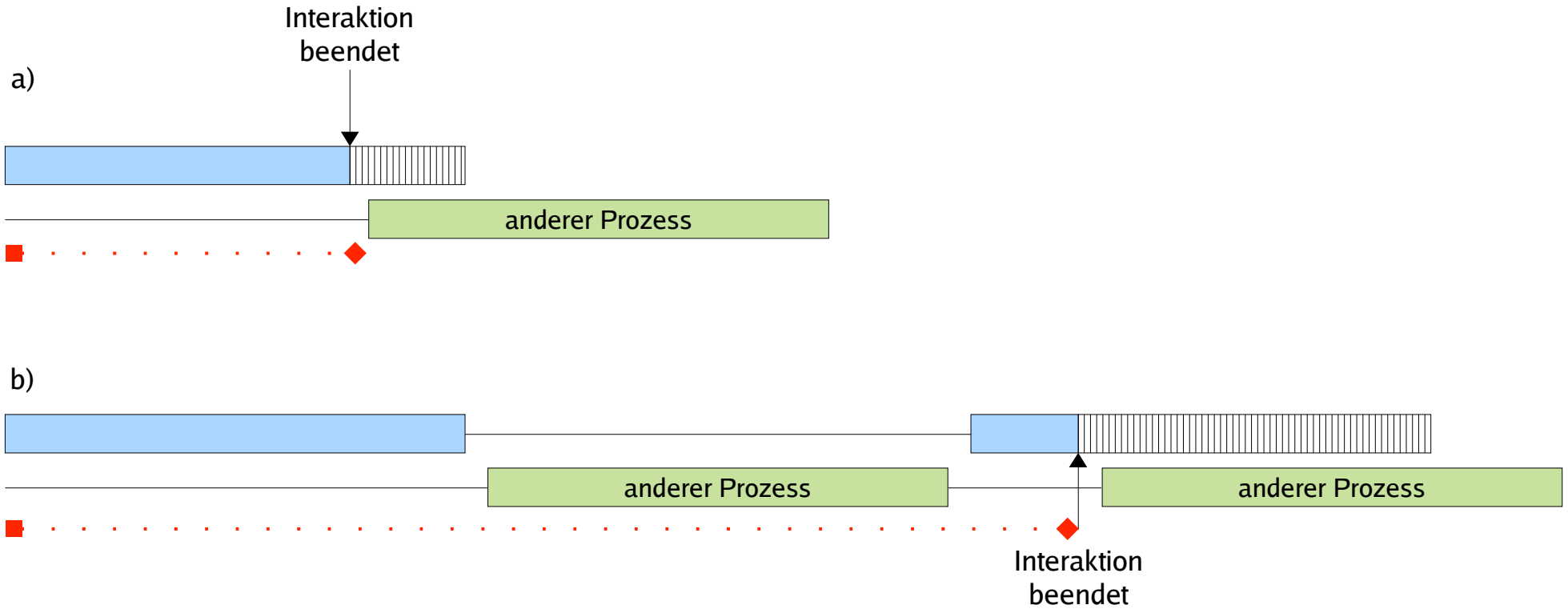
Quantenlänge

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

Quantenlänge

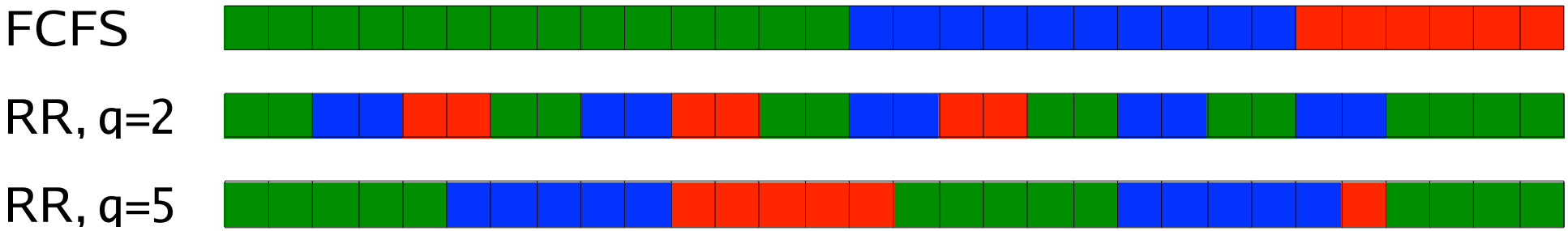
- Oft: Quantum *q* etwas größer als typische Zeit, die das Bearbeiten einer Interaktion benötigt



Round-Robin-Beispiel

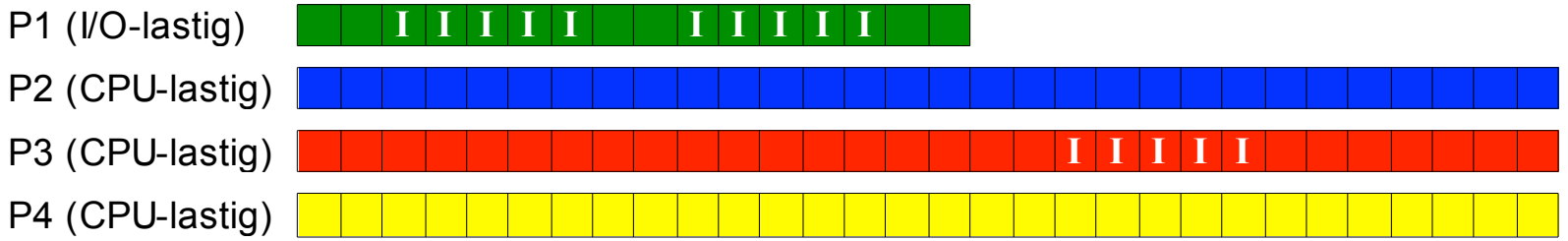
Szenario: Drei Prozesse

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

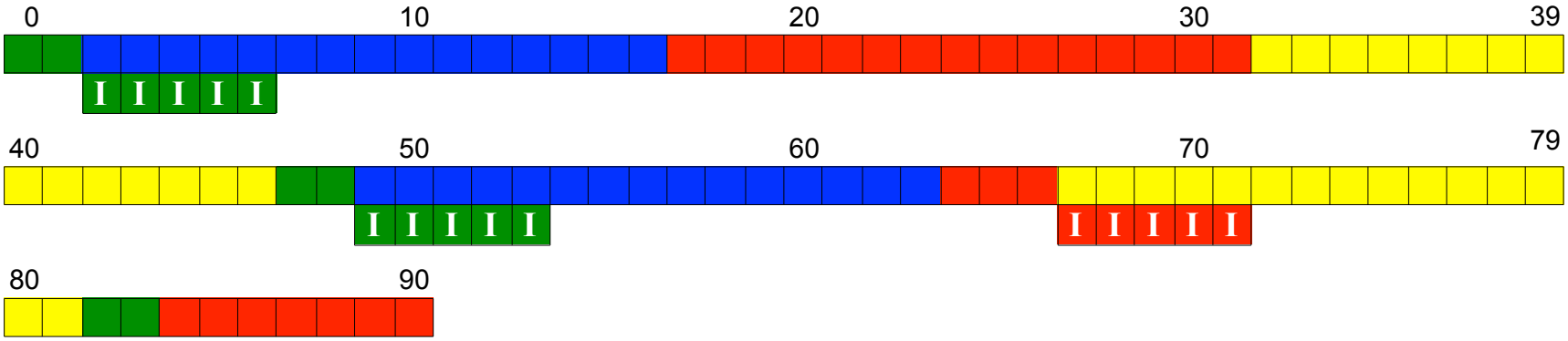


Round Robin: I/O- vs. CPU-lastig

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:

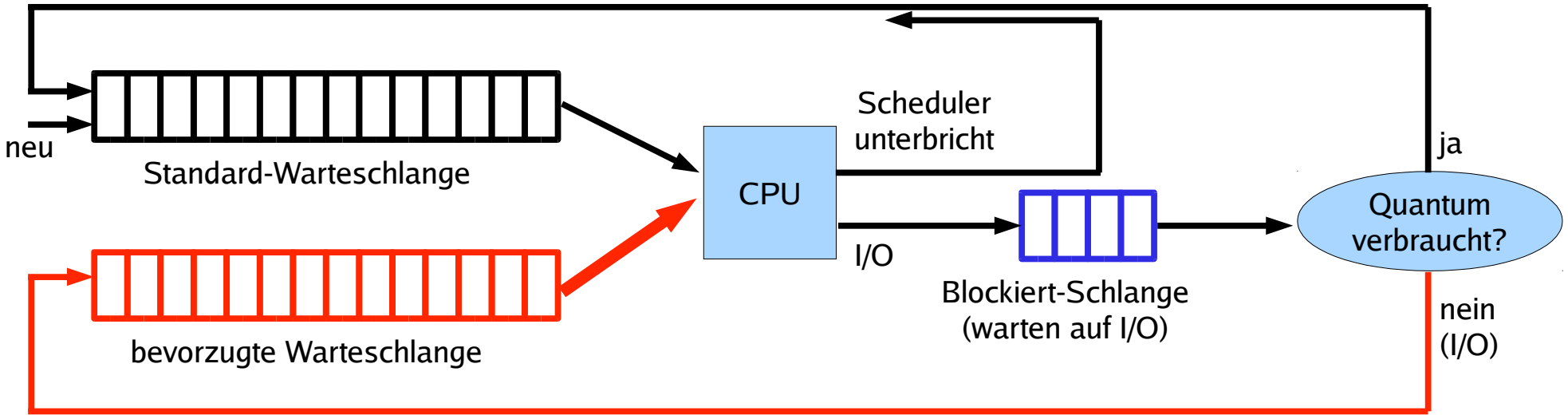
- Round Robin unfair gegenüber I/O-lastigen Prozessen:
  – CPU-lastige nutzen ganzes Quantum,
  – I/O-lastige nur einen Bruchteil

Lösungsvorschlag:

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

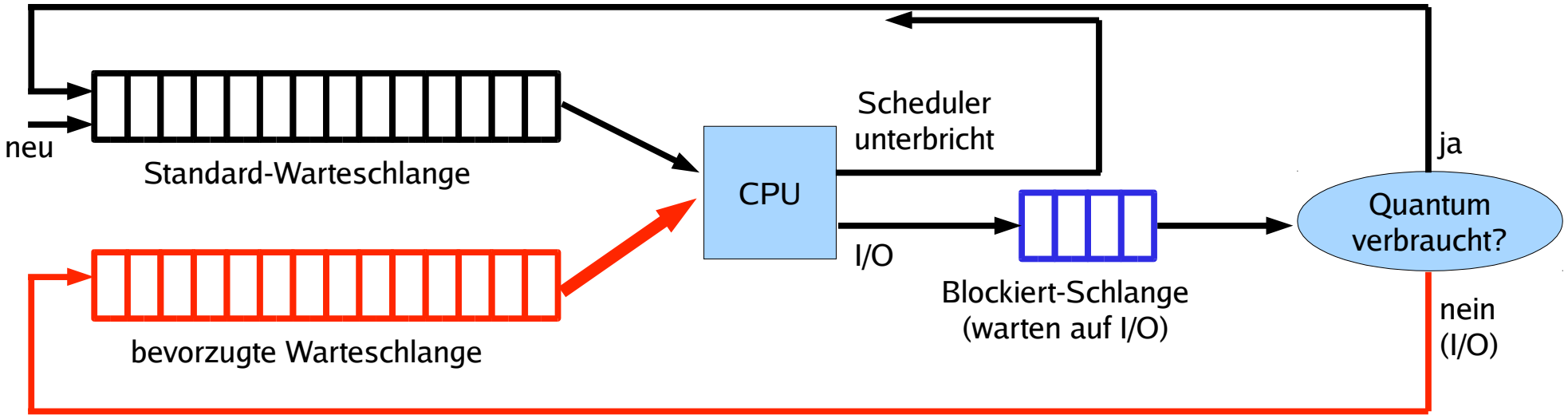
Virtual Round Robin (2)

- 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



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)



Übung zu Round Robin

K

Kontrollaufgabe 2.5

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

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

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

Zunächst sind alle Prozesse laufbereit. 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 laufbereite Prozesse werden aus der Warteschlange herausgenommen. Sobald ein nicht laufbereiter 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

Übung zu Round Robin

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.
- Genauer: (n-1) Quanten + die Zeit, die für (n-1) Kontextwechsel nötig sind

Interprozesskommunikation (IPC)

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

Einführung (1)

- 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

Einführung (2)

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

```
erhoehe_zaehler( )
{
  w=read(Adresse);
  w=w+1;
  write(Adresse,w);
}
```

Ausgangssituation: w=10

T1:	T2:
w=read(Adresse); // 10	
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:
```
w=read(Adr); // 10
w=w+1;       // 11
write(Adr,w); // 11
```

P2:
```
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 (4)

- Ursache: `erhoehe_zaehler()` 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 (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 (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

Einführung (7)

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

Einführung (8)

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

```
erhoehe_zaehler( ) {
  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

Einführung (9)

- 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 (2)

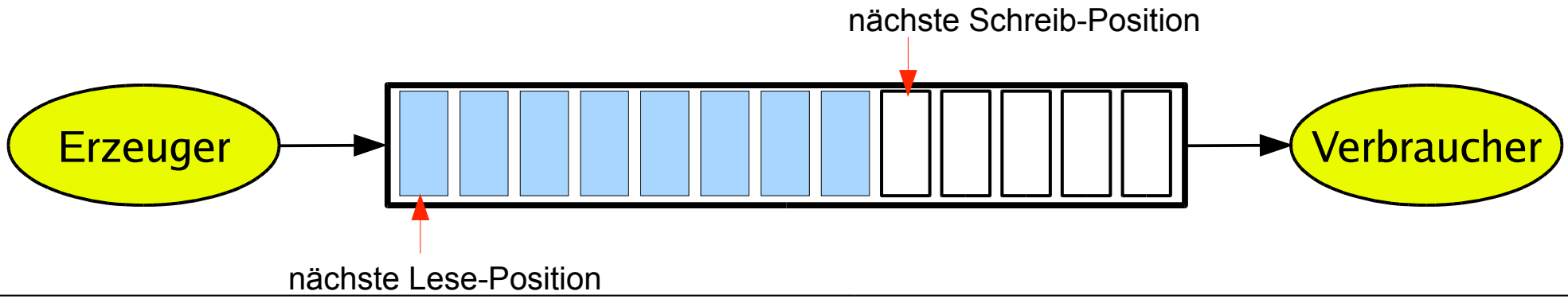
- Anforderung an parallele Threads:
  - Es darf maximal ein Thread gleichzeitig im kritischen Bereich sein
  - Kein Thread, der außerhalb kritischer Bereiche ist, darf einen anderen blockieren
  - Kein Thread soll ewig auf das Betreten eines kritischen Bereichs warten
  - Deadlocks sollen vermieden werden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

Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: **mut**ual **ex**clusion, 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 Programmier-fehlern

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.



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 (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.

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)

- Pseudo-Code für Semaphor-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 (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 → Warteschlange
- Bei Freigabe:
  - Warteschlange enthält Threads → einen wecken
  - Warteschlange leer: Mutex auf true setzen

Mutexe (2)

- **Mutex** (**mut**ual **ex**clusion) **= binärer Semaphor**, also ein Semaphor, 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 → lock
  signal → unlock
- Mutexe für exklusiven Zugriff (kritische Bereiche)

Atomare Operationen

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

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

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)

2.4.1.2 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`

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