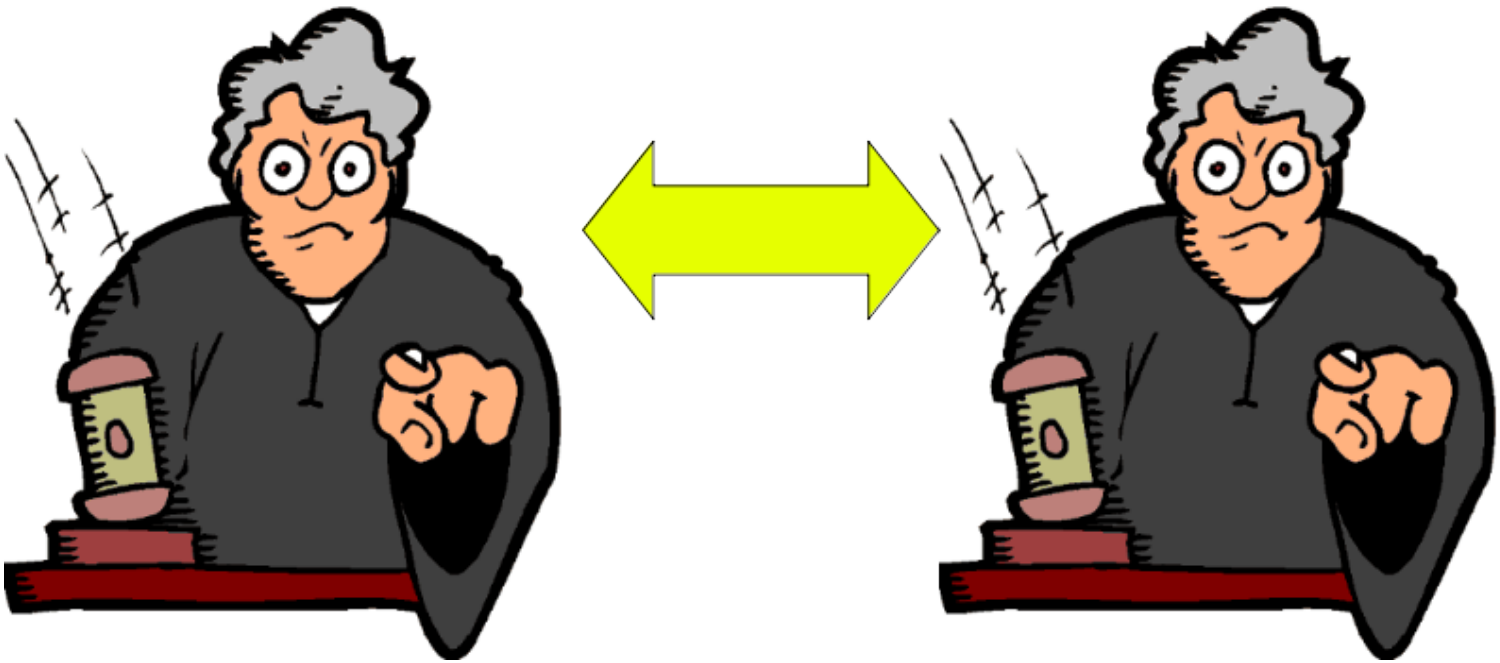
Betriebssysteme

Robert Kaiser

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

5. Prozesssynchronisation



https://www.animierte-gifs.net/data/media/1798/animiertes-richter-bild-0001.gif

Prozesssynchronisation

Hochschule RheinMain

Einführung

Hochschule RheinMain

Problemstellung

- **Gegeben**: Nebenläufige (konkurrente) Prozesse (vgl. Kap. 3, engl.: *concurrent processes*).
- **Ziel**: Vermeidung ungewollter gegenseitiger Beeinflussung.
- **Ziel**: Unterstützung gewollter Kooperation zwischen Prozessen:
  - Gemeinsame Benutzung von Betriebsmitteln (*sharing*)
  - Übermittlung von Signalen
  - Nachrichtenaustausch

$\Rightarrow$ **Fazit: Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig.**

Einführung

Hochschule RheinMain

Problemstellung

- **Gegeben**: Nebenläufige (konkurrente) Prozesse (vgl. Kap. 3, engl.: *concurrent processes*).
- **Ziel**: Vermeidung ungewollter gegenseitiger Beeinflussung.
- **Ziel**: Unterstützung gewollter Kooperation zwischen Prozessen:
  - Gemeinsame Benutzung von Betriebsmitteln (*sharing*)
  - Übermittlung von Signalen
  - Nachrichtenaustausch

⇒ Fazit: Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig.

Beispiel

Hochschule RheinMain

- Gemeinsame Benutzung eines Speicherbereiches (Hier: Datum „Kontostand“)
- Ungewollte gegenseitige Beeinflussung
- Gegeben: 2 Prozesse:

Prozess 1:

```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

Prozess 2:

```
I^^I/* Dauerauftrag Miete */
I^^Ix = lies_kontostand();
I^^Ix = x - 800;
I^^Ischreibe_kontostand(x);
I^^I
```

Möglicher Ablauf

Hochschule RheinMain

Prozess 1:

```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

Prozess 2:

```
I^^I/* Dauerauftrag Miete */
I^^Ix = lies_kontostand();
I^^Ix = x - 800;
I^^Ischreibe_kontostand(x);
I^^I
```

Mögliche Ausführungsreihenfolge der Anweisungen in Prozess 1 und 2:

```
I^^I/* Gehaltsueberweisung */    I^^I/* Dauerauftrag Miete */
I^^Iz = lies_kontostand();
                                 I^^Ix = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I.                            I^^Ix = x - 800;
I^^I.                            I^^Ischreibe_kontostand(x);
I^^I                             I^^I
```

- Pech, die Gehaltsüberweisung ist „verloren gegangen“ :-(
- Bei anderen Reihenfolgen werden die beiden Berechnungen „richtig“ ausgeführt, oder es geht der Dauerauftrag verloren.

Definitionen (1)

Annahme: Prozesse mit Lese/Schreib-Operationen auf Betriebsmitteln

Definition: Prozesskonflikt

Zwei nebenläufige Prozesse heißen **im Konflikt zueinander stehend** oder **überlappend**, wenn es ein Betriebsmittel gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie **unabhängig** oder **disjunkt**.

Definition: Zeitkritische Abläufe (*race conditions*)

Folgen von Lese/Schreib-Operationen der verschiedenen Prozesse heißen **zeitkritische Abläufe** (engl. ***race conditions***), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese/Schreib-Operationen sind.

Definitionen (2)

Hochschule RheinMain

Definition: Wechselseitiger Ausschluss (*mutual exclusion*)

Ein Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess auf ein gemeinsames Datum zugreift, heisst Verfahren zum **wechselseitigen Ausschluss** (engl. ***mutual exclusion***):

- Bemerkung: Ein Verfahren zum wechselseitigen Ausschluss vermeidet zeitkritische Abläufe. Es löst damit ein Basisproblem des Concurrent Programming.

Definition: Kritischer Abschnitt (*critical section*)

Der Teil eines Programms, in dem auf gemeinsam benutzte Datenbereiche zugegriffen wird, heisst **kritischer Abschnitt** oder **kritischer Bereich** (engl. ***critical section*** oder **critical region**):

- Bemerkung: Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe.
- Kritische Abschnitte realisieren sog. komplexe **unteilbare** oder **atomare** Operationen.

(Nicht-)atomare Operationen (1)


```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

```
I^^I/* Ueberweisung 2.0 */
I^^Iint kontostand;
I^^I
I^^Ikontostand += 1000;
I^^I
```

→ entspr. Maschinencode:

```
MOV kontostand,EAX
ADD #1000,EAX
MOV EAX,kontostand
```

(Nicht-)atomare Operationen (1)


```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

Nicht Atomar

```
I^^I/* Ueberweisung 2.0 */
I^^Iint kontostand;
I^^I
I^^Ikontostand += 1000;
I^^I
```

→ entspr. Maschinencode:

```
MOV kontostand,EAX
ADD #1000,EAX
MOV EAX,kontostand
```

(Nicht-)atomare Operationen (1)

Hochschule RheinMain

```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

Nicht Atomar

```
I^^I/* Ueberweisung 2.0 */
I^^Iint kontostand;
I^^I
I^^Ikontostand += 1000;
I^^I
```

Atomar?

→ entspr. Maschinencode:

```
MOV kontostand,EAX
ADD #1000,EAX
MOV EAX,kontostand
```

(Nicht-)atomare Operationen (1)


```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

Nicht Atomar

```
I^^I/* Ueberweisung 2.0 */
I^^Iint kontostand;
I^^I
I^^Ikontostand += 1000;
I^^I
```

Atomar?

→ entspr. Maschinencode:

```
MOV kontostand,EAX
ADD #1000,EAX
MOV EAX,kontostand
```

(Nicht-)atomare Operationen (1)


```
I^^I/* Gehaltsueberweisung */
I^^Iz = lies_kontostand();
I^^Iz = z + 1000;
I^^Ischreibe_kontostand(z);
I^^I
```

Nicht Atomar

```
I^^I/* Ueberweisung 2.0 */
I^^Iint kontostand;
I^^I
I^^Ikontostand += 1000;
I^^I
```

Atomar?

- Auch Operationen, die in Hochsprache (hier: C) atomar zu sein scheinen, sind es auf Maschinenebene u.U. nicht!
- Teilweise architekturabhängig

→ entspr. Maschinencode:

```
MOV kontostand,EAX
ADD #1000,EAX
MOV EAX,kontostand
```

3 Instruktionen

→ Nicht Atomar!

(Nicht-)atomare Operationen (2)

Hochschule RheinMain

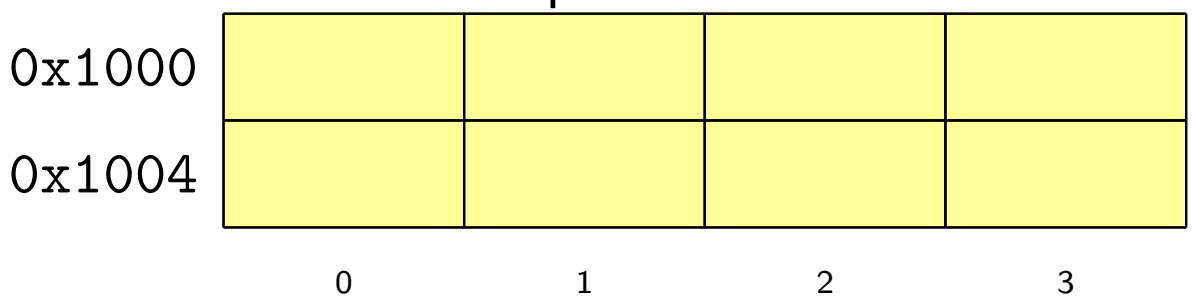
Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

- Beispiel

Assemblercode

```
LA r2,0x1001
SW 0x12345678,(r2)
```

Im Speicher:



- Ungerade Adresse → Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich → unterbrechbar!

(Nicht-)atomare Operationen (2)

Hochschule RheinMain

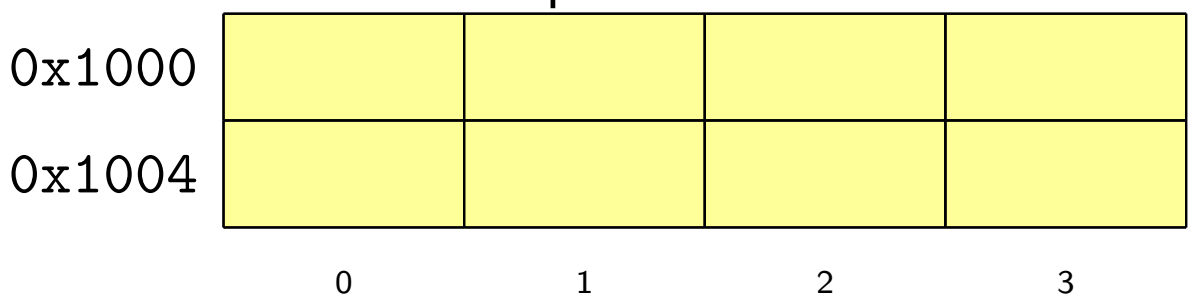
Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

- Beispiel

```
Assemblercode
LA r2,0x1001
SW 0x12345678,(r2)
```

1 Maschinenbefehl → Atomar?

Im Speicher:



- Ungerade Adresse → Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich → unterbrechbar!

(Nicht-)atomare Operationen (2)

Hochschule RheinMain

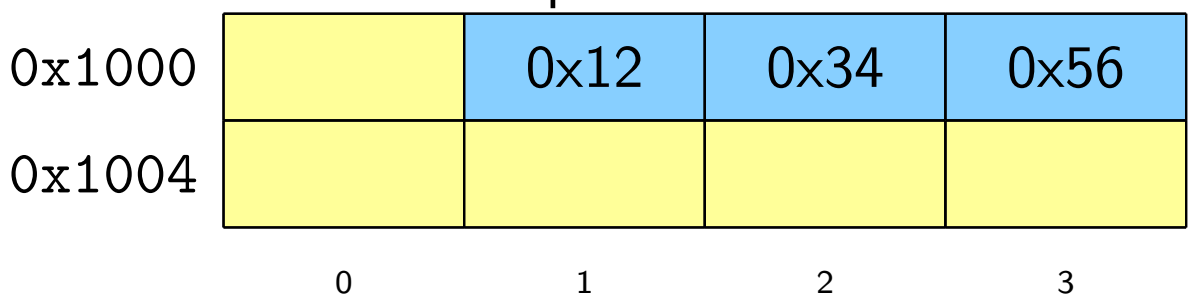
Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

- Beispiel

```
Assemblercode
LA r2,0x1001
SW 0x12345678,(r2)
```

1 Maschinenbefehl → Atomar?

Im Speicher:



- Ungerade Adresse → Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich → unterbrechbar!

(Nicht-)atomare Operationen (2)

Hochschule RheinMain

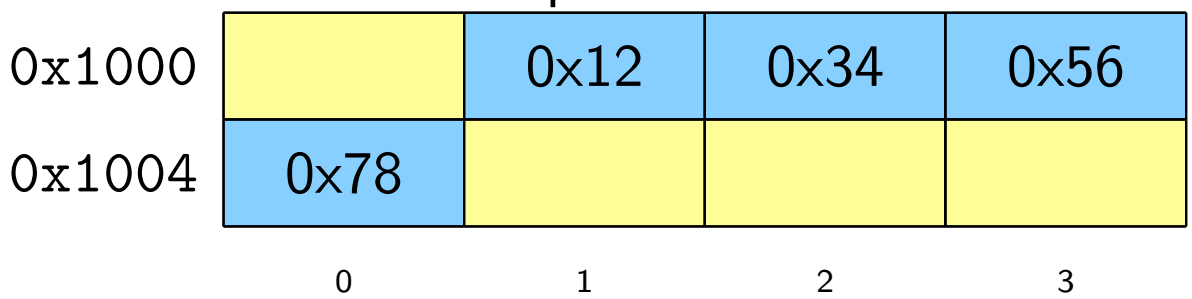
Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

- Beispiel

```
Assemblercode
LA r2,0x1001
SW 0x12345678,(r2)
```

1 Maschinenbefehl → Atomar?

Im Speicher:



- Ungerade Adresse → Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich → unterbrechbar!

(Nicht-)atomare Operationen (2)

Hochschule RheinMain

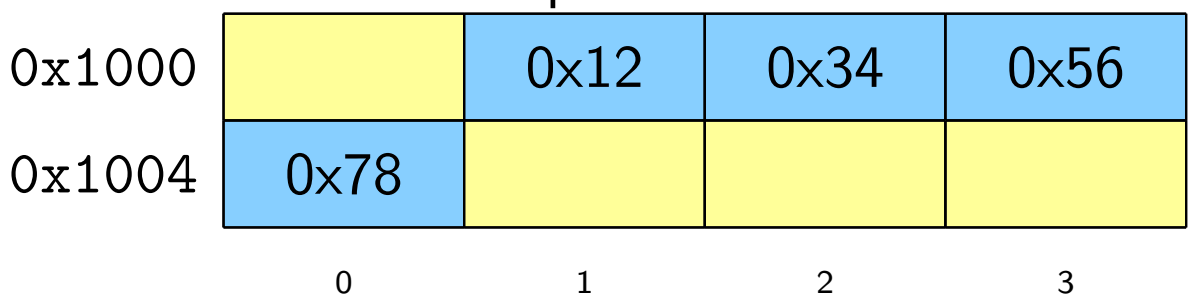
Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

- Beispiel

```
Assemblercode
LA r2,0x1001
SW 0x12345678,(r2)
```

1 Maschinenbefehl → Atomar?

Im Speicher:



- Ungerade Adresse → Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich → unterbrechbar!

Ein „guter“ Algorithmus

Forderungen an einen guten Algorithmus zum wechselseitigen Ausschluss:

1. Zu jedem Zeitpunkt darf sich nur ein Prozess in seinem kritischen Abschnitt befinden (Korrektheit, Basisforderung).
2. Es dürfen keine Annahmen über die Ausführungsgeschwindigkeiten oder die Anzahl der unterliegenden Prozessoren gemacht werden.
3. Kein Prozess, der sich nicht in seinem kritischen Abschnitt befindet, darf andere Prozesse blockieren (Fortschritt).
4. Alle Prozesse werden gleich behandelt (Fairness).
5. Kein Prozess darf unendlich lange warten müssen, bis er in seinen kritischen Abschnitt eintreten kann
   ($\rightarrow$ „Verhungern“, engl. *starvation*).

Synchronisationsprimitive

Gliederung

1. Wechselseitiger Ausschluss mit aktivem Warten
2. Wechselseitiger Ausschluss mit passivem Warten
3. Semaphore
4. Höhere Synchronisationsprimitive

Wechselseit. Ausschluss mit aktivem Warten

Hochschule RheinMain

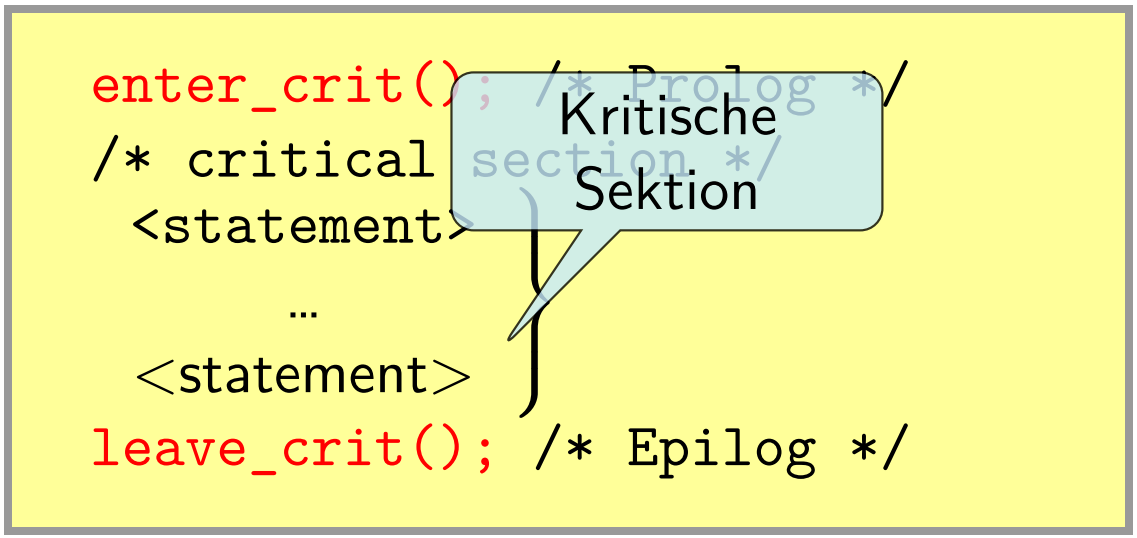
Generelle Vorgehensweise

```
enter_crit(); /* Prolog */
/* critical section */
  <statement>
       …
  <statement>
leave_crit(); /* Epilog */
```

- Prolog/Epilog-Paar
- **Aktives Warten** auf Eintritt in den kritischen Abschnitt (engl. *busy waiting*).
- Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit.
- Alle Prozesse müssen sich an das Vorgehen halten.

Wechselseit. Ausschluss mit aktivem Warten

Generelle Vorgehensweise



- Prolog/Epilog-Paar
- **Aktives Warten** auf Eintritt in den kritischen Abschnitt (engl. *busy waiting*).
- Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit.
- Alle Prozesse müssen sich an das Vorgehen halten.

Lösungsansätze

1. Sperren aller Unterbrechungen
2. Sperrvariablen
3. Striktes Alternieren
4. Lösung von Peterson[1]
5. Atomare read-modify-write Instruktionen[1]

[1] brauchbar

1. Sperren aller Unterbrechungen

Einfachste Lösung, Vorgehen:

- Jeder Prozess **sperrt** vor Eintritt in seinen kritischen Abschnitt alle **Unterbrechungen** (*disable interrupts*).
- Jeder Prozess **lässt** Unterbrechungen am Ende seines kritischen Abschnitts **wieder zu** (*enable interrupts*).

Funktionsweise:

- Der Prozessor kann nur dann zu einem anderen Prozess wechseln, wenn eine Unterbrechung auftritt. Also ist für diese Lösung die **Korrektheitsforderung erfüllt**.

Bemerkungen:

- Lösung ist **unbrauchbar für allgemeine Anwendungsprozesse**, da:
  1. das Sperren/Zulassen von Unterbrechungen i.d.R. ein privilegierter Maschinenbefehl ist
  2. nicht zugesichert werden kann, dass Anwendungen die Interrupts auch wieder zulassen (z.B. wegen Programmierfehler).
- Lösung wird häufig **innerhalb des Betriebssystemkerns** eingesetzt, um wechselseitigen Ausschluss zwischen Kernroutinen (z.B. mit einem Interrupt-Handler) zu gewährleisten.
- Lösung ist auch **unbrauchbar** im Falle eines **Multiprozessor**-Systems, da sich die Interrupt-Sperre i.d.R. nur auf einen Prozessor auswirkt.

2. Sperrvariablen

Hochschule RheinMain

Einfacher, aber falscher Lösungsansatz:

- Jedem kritischen Abschnitt wird eine **Sperrvariable** zugeordnet:
  - Wert = 0 bedeutet „frei“,
  - Wert = 1 bedeutet „belegt“.
- Jeder Prozess prüft die Sperrvariable vor Eintritt in den kritischen Abschnitt:
  - Ist sie 0, so setzt er sie auf 1 und tritt in den kritischen Abschnitt ein.
  - Ist sie 1, so wartet er, bis sie den Wert 0 annimmt.
- Am Ende seines kritischen Abschnitts setzt der Prozess den Wert der Sperrvariablen auf 0 zurück.

Prozess 1:

```
I^^I/* Prolog */
I^^Iwhile(sperrvar) { ; }
I^^Isperrvar = 1;
I^^I/* kritischer Bereich */
I^^I...
I^^I/* Epilog */
I^^Isperrvar = 0;
I^^I
```

Prozess 2:

```
I^^I/* Prolog */
I^^Iwhile(sperrvar) { ; }
I^^Isperrvar = 1;
I^^I/* kritischer Bereich */
I^^I...
I^^I/* Epilog */
I^^Isperrvar = 0;
I^^I
```

2. Sperrvariablen (2)

Hochschule RheinMain

Bemerkungen:

- Prinzipiell der gleiche Fehler wie beim Konto-Beispiel: Zwischen Abfrage der Sperrvariablen und folgendem Setzen kann der Prozess unterbrochen werden.
- Damit ist es möglich, dass sich beide Prozesse im kritischen Abschnitt befinden (→**Korrektheitsbedingung verletzt !**).
- Speicher (-wörter, -variablen) erlauben i.d.R. nur unteilbare Lese-, bzw. Schreibzugriffe, jedoch kein unteilbares *read-modify-write* (Eigenschaft der Architektur des Speicherwerks).

```
I^^I/* Prolog */
I^^Iwhile(sperrvar) { ; }
I^^I...
I^^Isperrvar = 1;
I^^I/* kritischer Bereich */
I^^I...
I^^I/* Epilog */
I^^Isperrvar = 0;
```

```
I^^I/* Prolog */
I^^Iwhile(sperrvar) { ; }
I^^Isperrvar = 1;
I^^I...
I^^I/* kritischer Bereich */
I^^I...
I^^I/* Epilog */
I^^Isperrvar = 0;
```

3. Striktes Alternieren


```
I^^Iint dran = 0; /* gem. Variable: wer ist "dran" ? */
I^^I
```

Prozess 0:

```
I^^I/* Prolog */
I^^Iwhile(dran != 0) { ; }
I^^I/* kritischer Bereich */
I^^I...
I^^I/* Epilog */
I^^Idran = 1;
I^^I
```

Prozess 1:

```
I^^I/* Prolog */
I^^Iwhile(dran != 1) { ; }
I^^I/* kritischer Bereich */
I^^I...
I^^I/* Epilog */
I^^Idran = 0;
I^^I
```

- Gemeinsame Variable `dran` gibt an, welcher Prozess[2] den kritischen Bereich betreten darf
- Prozesse wechseln sich ab: 0,1,0,1,...
- Die Lösung erfüllt die Korrektheitsbedingung, aber

→ Fortschrittsbedingung (3) kann verletzt sein, wenn ein Prozess wesentlich langsamer als der andere ist.

- Fazit: **keine ernsthafte Lösung**.

[2] Hier: 0 oder 1

4. Lösung von Peterson

- Historische Vorläufer:
  - Anfang der 60er Jahre viele Lösungsansätze, nur wenige erfüllten alle genannten Bedingungen
  - Erste korrekte Software-Lösung für 2 Prozesse: Algorithmus von Dekker
- Neue Lösung zum wechselseitigen Ausschluss:
  - Lösung von Peterson (1981) (im folgenden betrachtet)
  - Basiert (ebenfalls) auf unteilbaren Speicheroperationen und aktivem Warten, ist aber einfacher
  - Prolog: `enter_crit()`, Epilog: `leave_crit()`
- Weitere Lösungen für mehr als zwei Prozesse von:
  - Dijkstra, Peterson, Knuth, Eisenberg/McGuire, Lamport (hier nicht weiter diskutiert)

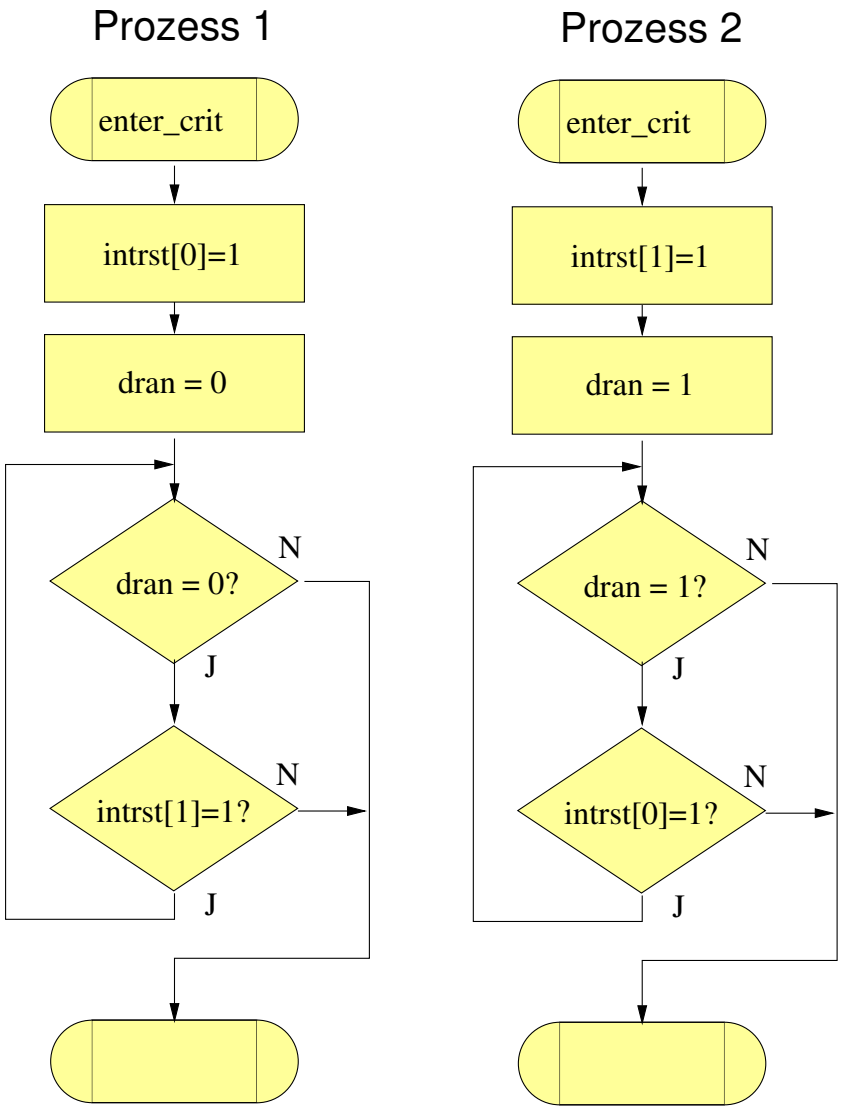
Peterson-Algoritmus (1)


```
#define N    2                    /* Anzahl der Prozesse           */

/* gemeinsame Variablen                                            */
volatile int dran;                /* Wer kommt dran?               */
volatile bool interested[N];      /* Wer will, anfangs alle false */

/* Prolog-Operation, vor Eintritt in den
   kritischen Bereich ausfuehren   */
void enter_crit(int process) { /* process: wer tritt ein: 0,1   */
   int other;                     /* Nummer des anderen Prozesses */
   other = 1 - process;
   interested[process] = true; /* zeige eigenes Interesse        */
   dran = process;                /* setze Marke, unteilbar!      */
   while(dran==process && interested[other]);
                                  /* ev. Aktives Warten !!!       */
}
^^I
```

```
/* Epilog-Operation, nach Austritt aus dem
   krit. Bereich ausfuehren */
void leave_crit(int process) { /* process: wer verlaesst: 0,1   */
   interested[process] = false; /* Verlasse krit. Bereich        */
}
^^I
```

Peterson-Algorithmus (2)

Hochschule RheinMain



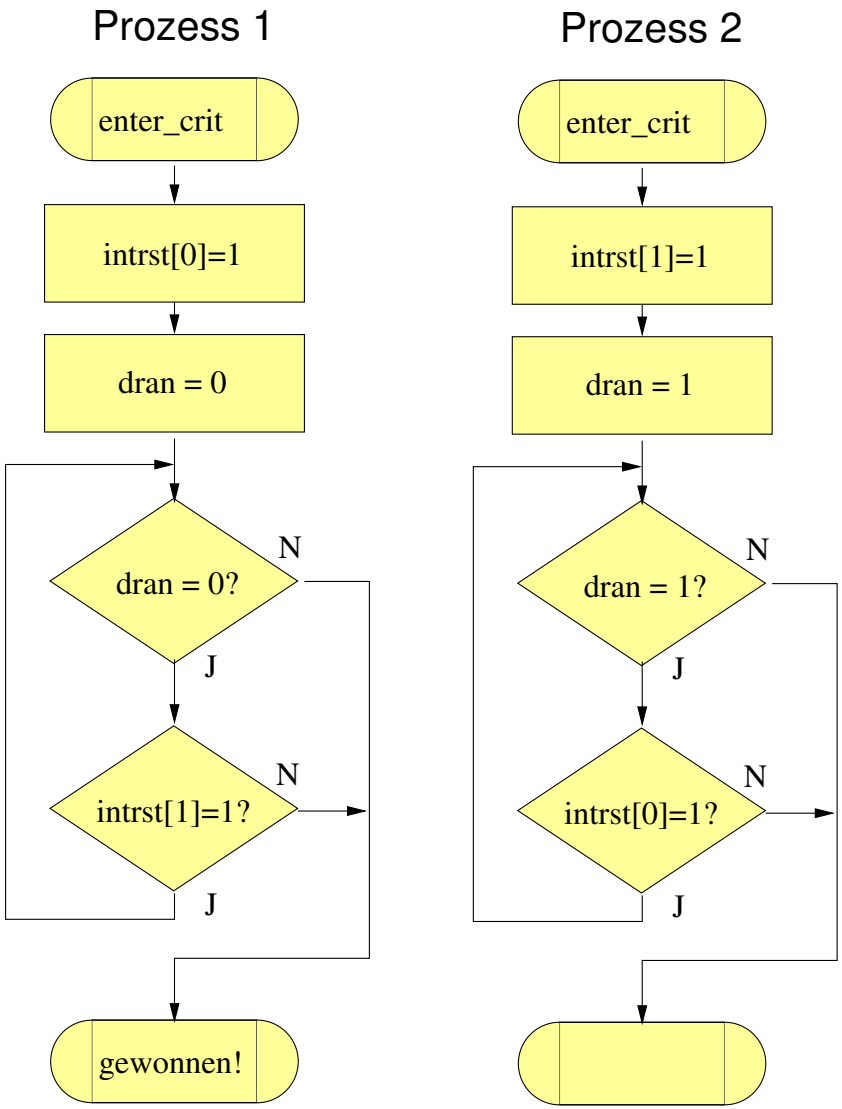
Anfangszustand

intrst[]:	0	0

dran:	0

Peterson-Algorithmus (3)

Hochschule RheinMain

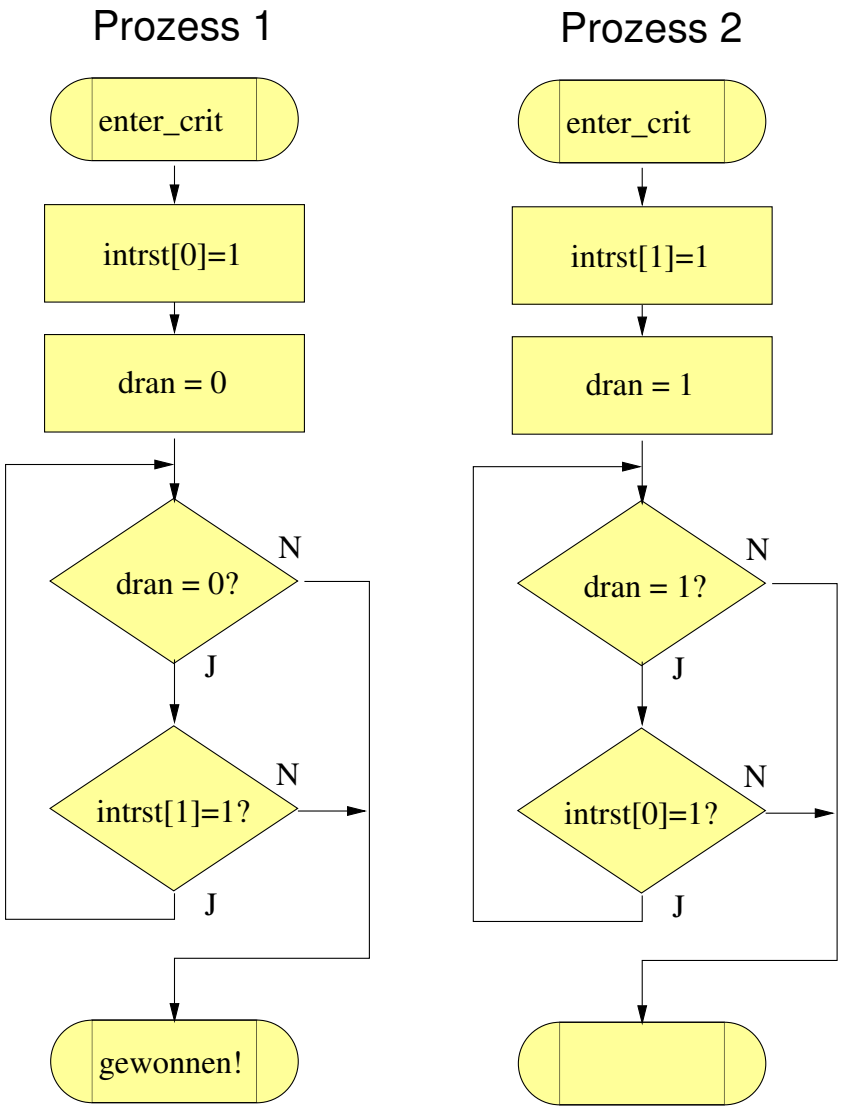


Prozess 1 betritt kritischen Abschnitt, Prozess 2 nicht

intrst[]:	1	0

dran:	0

Peterson-Algorithmus (4)



Prozess 1 ist in kritischem Abschnitt, Prozess 2 versucht, ihn zu betreten

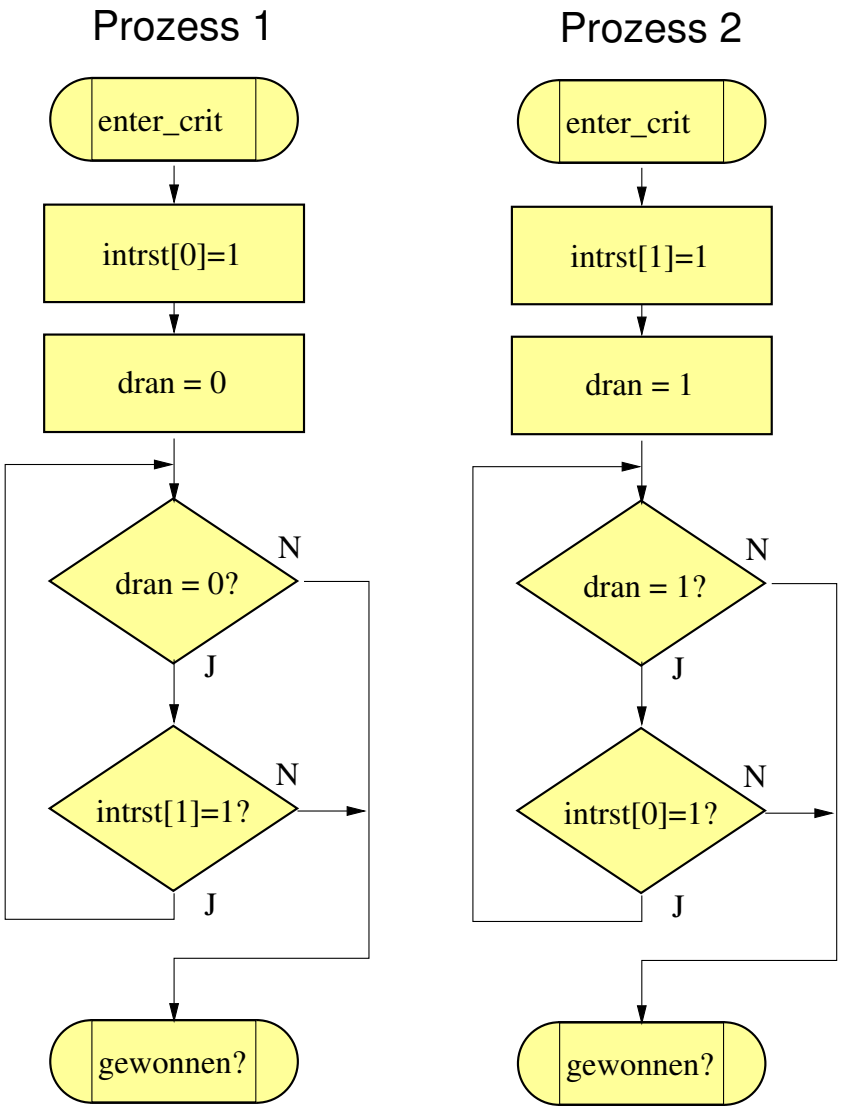
intrst[]: 1 1

dran: 1

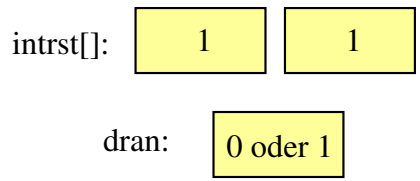
Prozess 2 wartet, bis intrst[0] = 0

Peterson-Algorithmus (5)

Hochschule RheinMain



Kritischer Fall:
beide versuchen gleichzeitig,
den kritischen Abschnitt
zu betreten



Schreibzugriff auf ,,dran'' ist atomar. Nur einer von beiden kann ,,dran'' erfolgreich setzen.

Wer Erfolg hat, muss hier warten!

5. Atomare read-modify-write Instruktionen

- Algorithmen, die nur auf atomaren Speicherzugriffen basieren sind
  - Komplex → fehlerträchtig
  - Auf eine feste Teilnehmerzahl beschränkt
- Zudem besteht bei einigen Verfahren die Gefahr des „Verhungerns" (engl. *starvation*)

→ Notwendigkeit einer effizienten Lösung

- Lösung durch Hardware-Unterstützung:
- Einführung spezieller Maschinenbefehls: unteilbares Lesen einer Speicherzelle mit anschließendem Schreiben eines Wertes in die Zelle (Speicherbus wird zwischendurch **nicht** freigegeben)
  - Heute Standard auf praktisch allen Architekturen
  - Notwendig für Mehrprozessor-Systeme
  - Häufig auch zusätzliche unteilbare Maschinen-Operationen zur Listenmanipulation (z.B. für Ready-Queue)
  - Oder LOCK/UNLOCK-Paare, um kurze Folgen beliebiger Instruktionen atomar auszuführen (z.B. i860).

Beispiel: Test-and-Set-Befehl (1)

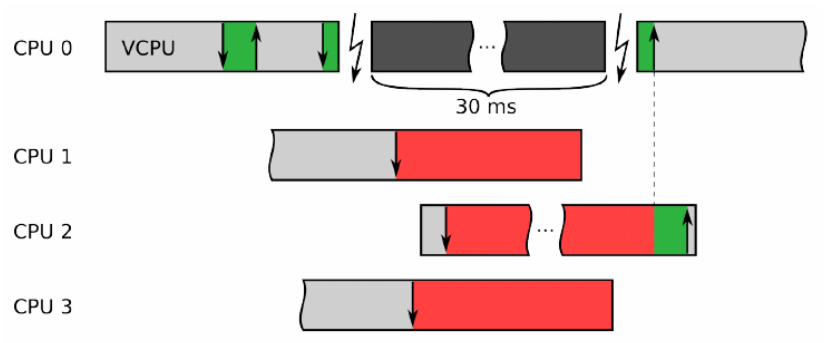
- Jedem kritischen Abschnitt wird eine Sperrvariable `flag` zugeordnet
- Wert = 0 bedeutet „frei“, Wert = 1 bedeutet „belegt“
- Auf Test-and-Set basierende Sperrvariablen mit aktivem Warten heißen auch *spin locks*
- Große Bedeutung in Multiprozessor-Betriebssystemen
- Typische Assembler-Routinen für Prolog und Epilog mit Test-and-Set-Instruktion „TAS“:

```
I^^I^^Ienter_crit:
I^^I^^I^^Itas reg,flag     | kopiere flag in reg, setze flag = 1
I^^I^^I^^Icmp reg,#0       | war flag vorher 0?
I^^I^^I^^Ijnz enter_crit   | nein -> Sperre war belegt -> warten
I^^I^^I^^Iret              | ja -> darf einreten, zurueck zum Aufrufer

I^^I^^Ileave_crit:
I^^I^^I^^Imov flag,#0      | loesche flag -> setze Sperre zurueck
I^^I^^I^^Iret              | zurueck zum Aufrufer
I^^I
```

- (Im Gegensatz zur Darstellung hier erfolgt die Realisierung i.d.R. über Makros, da Prozeduraufrufe zur Laufzeit zu großen Overhead darstellen)

Lock Holder Preemption Problem

- Problem der *Lock Holder Preemption* tritt in Verbindung mit Spinlocks bei Mehrprozessor-Systemen, sowie bei Virtualisierung auch bei Einprozessorsytemen auf.
- Szenario:
  - Ein (ggf. virtueller) Prozessor hat einen durch Spinlock geschützten kritischen Bereich betreten und wird in diesem Zustand unterbrochen.
  - Ein anderer (virtueller) Prozessor der versucht, den kritischen Bereich zu betreten, muss **aktiv** auf Freigabe des Spinlocks warten.
  - Das kann sehr lange dauern (Quantum)



http://www-archive.xenproject.org/files/xensummitboston08/LHP.pdf

Wechselseit. Ausschluss mit passivem Warten

Bisher

- Prolog-Operationen zum Betreten des kritischen Abschnitts führen zum **aktiven Warten**, bis der betreffende Prozess in den kritischen Abschnitt eintreten kann.
- Der wartende Prozess **gibt den Prozessor dabei nicht ab**.
  - → verbraucht viel Rechenzeit
  - → hindert ggf. andere Prozesse an der Freigabe des kritischen Abschnittes, auf die er selbst wartet.
- Lediglich auf Multiprozessor-Systemen kann kurzzeitiges aktives Warten zur Vermeidung eines Prozesswechsels sinnvoll sein (spin locks).

Ziel: Vermeidung von verschwendeter Prozessorzeit. Vorgehensweise:

- Prozesse **blockieren**, wenn sie nicht in ihren kritischen Abschnitt eintreten können.
- Ein blockierter Prozess wird **durch den** aus dem kritischen Abschnitt **austretenden Prozess entblockiert**.

Passives Warten: Einfachste Primitive

- Einfachste Primitive werden vielfach als `SLEEP`[3] und `WAKEUP` bezeichnet.
- `SLEEP()` blockiert den ausführenden Prozess, bis er von einem anderen Prozess geweckt wird.
- `WAKEUP(process)` weckt den Prozess `process`. Der ausführende Prozess wird dabei nie blockiert.
- Häufig wird als Parameter von `SLEEP` und `WAKEUP` ein Ereignis (Speicheradresse einer beschreibenden Struktur) verwendet, um die Zuordnung treffen zu können, (vgl. 3.2 Beispiel UNIX(7) Kernroutinen).
- Diese Primitive können auch der allgemeinen ereignisorientierten Kommunikation dienen.

[3] Achtung: nicht zu verwechseln mit C-Funktion `sleep()`

Mutex-Locks

- Der Begriff **Mutex** ist von *mutual exclusion* (= Wechselseitiger Ausschluss) abgeleitet.
- ein Mutex bietet folgende Funktionen:
  - *lock* als Prolog-Operation zum Betreten des kritischen Abschnitts
  - *unlock* als Epilog-Operation beim Verlassen des kritischen Abschnitts
- Mutexe können als Spezialfall von Semaphoren angesehen werden (vgl. 5.2.3).
- Bei einigen Implementierungen entblockiert unlock **alle** wartenden Prozesse, damit sich diese dann erneut um das Betreten des kritischen Abschnitts bewerben.

Beispiel: Pthreads - API: Mutexe


```
#include <pthread.h>
```

- Anlegen einer Mutex-Variablen (mehrere Varianten):

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
```

- Initialisieren einer Mutex-Variablen:

```
pthread_mutex_init(pthread_mutex_t *mutex,
                   const pthread_mutexattr_t *mutexattr);
```

- Lock anfordern:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Lock anfordern, falls ohne Blockieren möglich:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Lock freigeben:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Lock zerstören:

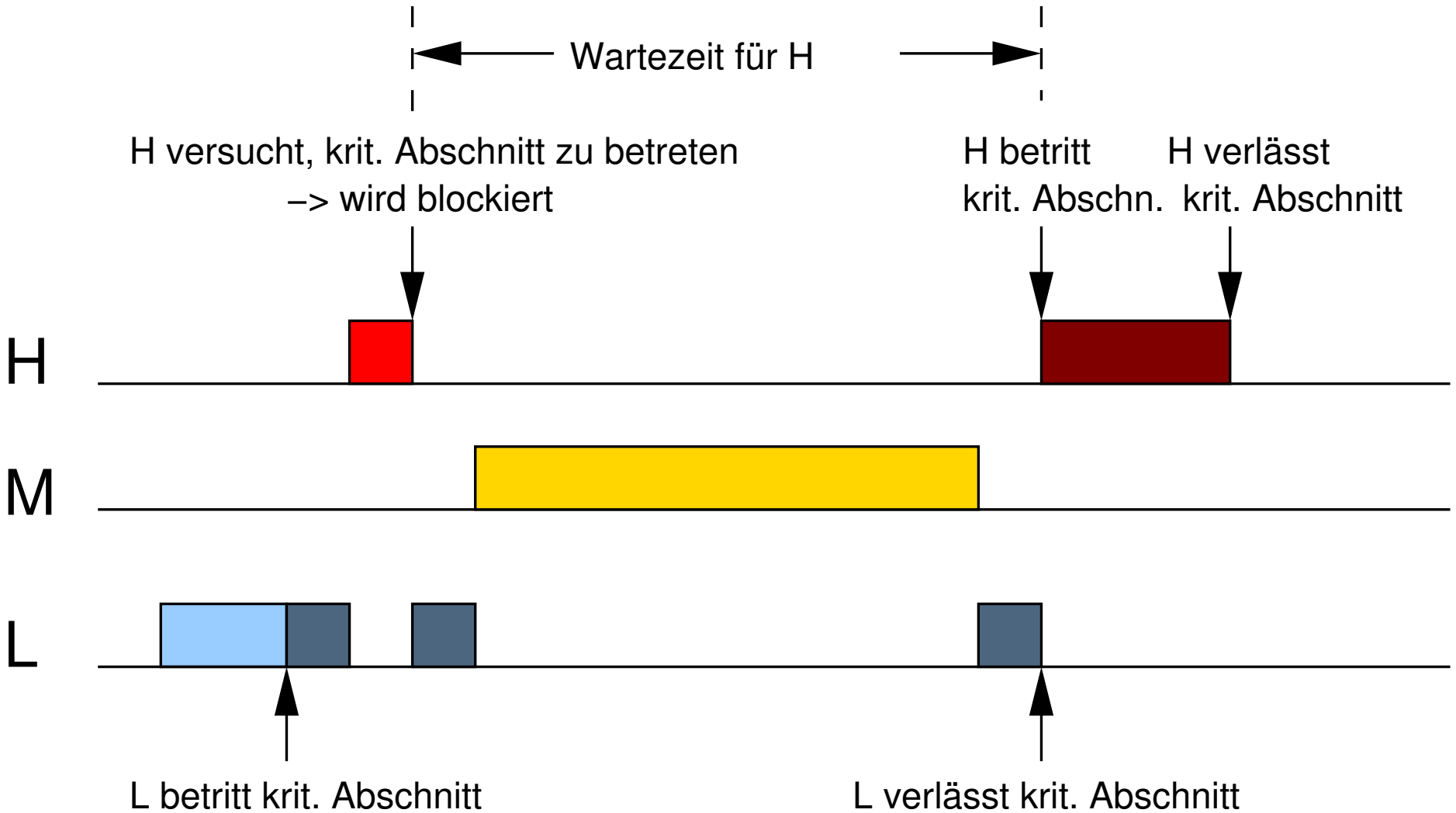
```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Problem der Prioritätsinversion

- Problem der Prioritätsinversion tritt bei prioritätsbasiertem Scheduling unterbrechbarer Prozesse in Zusammenhang mit kritischen Abschnitten auf
- Szenario:
  - Prozess H hat hohe Priorität, Prozess L hat niedere Priorität. Scheduling-Regel: wenn H rechenwillig ist, soll H ausgeführt werden
  - H wird rechenwillig, während L sich in seinem kritischen Abschnitt befindet. H will ebenfalls in seinen kritischen Abschnitt eintreten
  - Der höher priore Prozess H kann nicht weiter ausgeführt werden, da L den kritischen Abschnitt noch nicht freigegeben hat
  - Die Dauer der Verzögerung hängt nicht nur von L ab, sondern von allen Prozessen, die aufgrund ihrer Priorität vor H ausgeführt werden
  - Insbesondere kann ein Prozess M, dessen Priorität über der von L liegt, den krit. Abschnitt beliebig lange blockieren, ohne selbst daran beteiligt zu sein.
- Es muss also der nieder priore Prozess ausgeführt werden! Diese Situation heißt Prioritätsinversionsproblem.

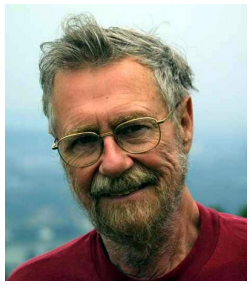
Problem der Prioritätsinversion (2)

Hochschule RheinMain



Semaphore

Hochschule RheinMain

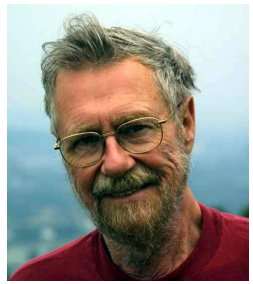


- 1965 von Edsger W. Dijkstra eingeführt
- Supermarkt-Analogie:
  - Kunde darf den Laden nur mit einem Einkaufswagen betreten
  - Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
  - Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird

- Semaphor besteht aus:
  - einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
  - einer **Warteschlange** für (passiv) wartende Prozesse

Semaphore

Hochschule RheinMain



- 1965 von Edsger W. Dijkstra eingeführt
- Supermarkt-Analogie:
  - Kunde darf den Laden nur mit einem Einkaufswagen betreten
  - Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
  - Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird

- Semaphor besteht aus:
  - einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
  - einer **Warteschlange** für (passiv) wartende Prozesse

Operationen auf Semaphoren (1)

- **Initialisierung**
  - Zähler auf initialen Wert setzen
  - „Anzahl der freien Einkaufswagen“


- **Operation P()**: Passier(-Wunsch) (auch: **DOWN()**)
  - Zähler $= 0 \rightarrow$ Prozess in Warteschlange setzen, blockieren
  - Zähler > 0: Prozess kann passieren
  - In beiden Fällen wird der Zähler (ggf. nach dem Ende der Blockierung) erniedrigt
  - P steht für „proberen” (niederländisch für „testen”)
- **Operation V()**: Freigeben (auch: **UP()**)
  - Zähler wird erhöht
  - Falls es Prozesse in der Warteschlage gibt, wird einer de-blockiert (und erniedrigt den Zähler dann wieder, s.o.)
  - V steht für „verhogen” (niederländisch für „erhöhen”)

Operationen auf Semaphoren (2)

Bemerkungen

- P() und V()sind atomare Operationen:
  - Das Überprüfen, Dekrementieren und Sich-Schlafen-Legen des Aufrufers bei P() sowie das inkrementieren und ggf. Wecken eines Prozesses bei V() ist jeweils eine Transaktion, die nur vollständig „in einem Zuge" ausgeführt werden kann
- Kein Prozess wird bei der Ausführung der V()-Operation blockiert

Implementierung von Semaphoren

- I.d.R. als Systemaufrufe
- Intern Nutzung für Sich-Schlafen-Legen und Aufwecken
- Wesentlich ist die Unteilbarkeit von P() und V():
  - Auf Einprozessorsystemen: durch Sperren aller Unterbrechungen während der Ausführung von P() und V(). Zulässig, da nur wenige Maschineninstruktionen zur Implementierung nötig sind, und da dem Betriebssystemkern vertraut wird.
  - Auf Multiprozessorsystemen: Jedem Semaphor wird ein Spinlock mit aktivem Warten vorgeschaltet. So kann stets höchstens ein Prozessor den Semaphor manipulieren
  - Man beachte: Unterscheide zwischen aktivem Warten auf den Zugang zum Semaphor, der einen kritischen Abschnitt schützt (einige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt)

Binärsemaphore und Zählsemaphore

- Semaphore, die nur die Werte 0 und 1 annehmen, heißen *binäre Semaphore*
- ansonsten heißen sie *Zählsemaphore*
- Binäre Semaphore dienen zur Realisierung des wechselseitigen Ausschlusses
- Ein mit $n > 1$ initialisierter Zählsemaphor kann zur Kontrolle der Benutzung eines in n Exemplaren vorhandenen Typs von sequentiell wiederverwendbaren Betriebsmitteln verwendet werden
- Semaphore sind weit verbreitet, innerhalb von Betriebssystemen und zur Programmierung nebenläufiger Anwendungen
- Programmierung mit Semaphoren ist oft fehleranfällig, wenn mehrere Semaphore benutzt werden müssen (vgl. 5.3 und Kap. 7 (Deadlocks))
- Mutex-locks (vgl. 5.2.2) können als binäre Semaphore angesehen werden.

Anwendung: Wechselseitiger Ausschluss


```
semaphore sem = 1;   /* Init. mit 1            */
...                  /* -> binaere Semaphore   */
P(sem);              /*    Prolog              */
     <statement> ;   /* krit. Abschnitt        */
     ...
     <statement> ;   /* krit. Abschnitt        */
V(sem);              /*    Epilog              */
...
^^I
```

- Der mit 1 initialisierte Semaphor `sem` wird von allen beteiligten Prozessen benutzt
- Jeder Prozess klammert seinen kritischen Abschnitt mit `P(sem)` zum Eintreten und `V(sem)` zum Verlassen

Anwendung: Betriebsmittelvergabe

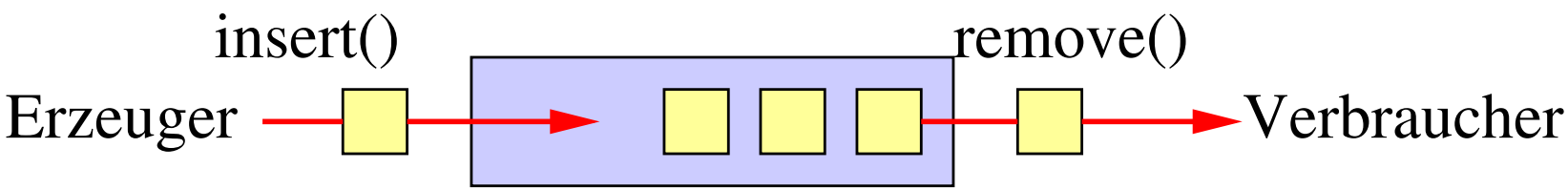
- Es seien `n` Exemplare vom Betriebsmitteltyp `bm` vorhanden
- Jedes BM dieses Typs sei sequentiell benutzbar

```
semaphore bm = n;

P(bm);^^I        /*   Beantragen     */

    Benutze das Betriebsmittel

V(bm);^^I        /*   Freigeben      */
^^I
```

- Dem BM-Typ `bm` wird ein Semaphor `bm`, zugeordnet
- Beantragen eines BM durch `P(bm)`, evtl. blockieren, bis ein BM verfügbar wird
- Nach der Benutzung freigeben des BM durch `V(bm)`
  (Es kann damit von einem weiteren Prozess benutzt werden)

Beispiel: Erzeuger/Verbraucher-Problem

- Gegeben:
  - Datenobjekte (z.B. Nachrichten) fester Größe
  - Pufferspeicher von begrenzter Größe (n Objekte)
- Aufgabe
  - Verschiedene Prozesse legen Daten mit Hilfe einer Funktion `insert()` im Puffer ab („Erzeuger"), bzw. entnehmen sie mit einer Funktion `remove()` („Verbraucher")
  - `remove()` soll warten, wenn keine Objekte im Puffer sind
  - `insert()` soll warten, wenn kein Platz mehr im Puffer ist

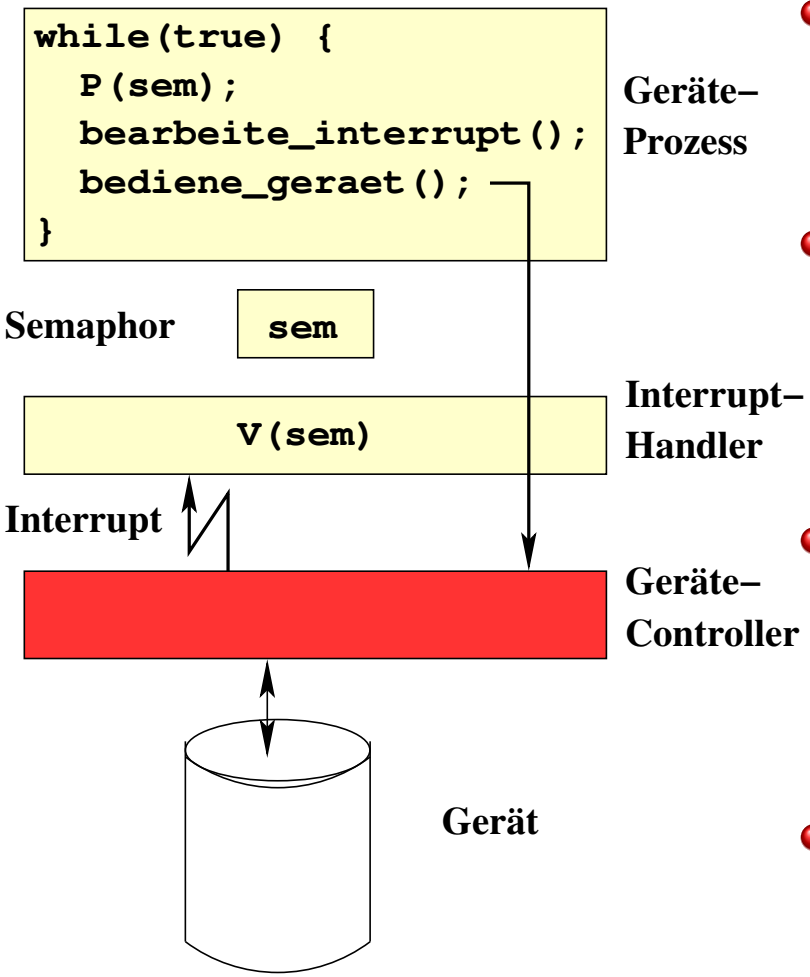


Lösung mit Semaphoren


```
/* Semaphore initialisieren:
 * empty = Puffergroesse,
 * full=0, mutex=1
 */

void insert(int item) {
    P(&empty);
    P(&mutex);
/* hier: item in Puffer
   stellen                  */
    V(&mutex);
    V(&full);
}

int remove(void) {
    P(&full);
    P(&mutex);
/* hier: vorderstes Item
   aus Puffer holen         */
    V(&mutex);
    V(&empty);
}
```

- **full**: zählt belegte Einträge im Puffer, verhindert Entnahme aus leerem Puffer
- **empty**: verwaltet freie Plätze im Puffer, verhindert Einfügen in vollen Puffer
- **mutex**: schützt den kritischen Bereich vor gleichzeitigem Betreten (binär-Semaphor: nimmt nur Werte 1/0 an)

Anwendung: Verstecken von Interrupts



- Einem E/A-Gerät wird ein Geräteprozess und ein mit 0 initialisierter Semaphor zugeordnet
- Nach dem Start des Geräteprozesses führt dieser eine P-Operation auf dem Semaphor aus (und blockiert)
- Im Falle eines Interrupts des Gerätes führt der Interrupt-Handler eine V-Operation auf dem Semaphor aus
- Der Geräteprozess wird entblockiert (und dadurch rechenwillig) und kann das Gerät bedienen

Anwendung: Durchsetzen einer Vorrangrelation

- Sei $\mathbb{P}$ eine Menge von kooperierenden Prozessen (ein *Prozesssystem*), und sei „$<$“ eine partielle Ordnung auf der Menge $\mathbb{P}$ mit:

  $$P_1 < P_2 :\Leftrightarrow \text{Prozess } P_1 \text{ muss vor } P_2 \text{ ausgeführt werden;}$$

  „$<$“ wird **Vorrangrelation** genannt.
- Graphisch dargestellt als **gerichteter azyklischer Graph**[4], wird dabei häufig $P_1 < P_2$ dargestellt durch: $P_1 \rightarrow P_2$
- Jeder Vorrangbeziehung $P_1 < P_2$ wird ein Semaphor `s` zugeordnet, auf das P_1 `V(s)` und P_2 `P(s)` ausführt.
- Alle Semaphore werden mit 0 initialisiert.

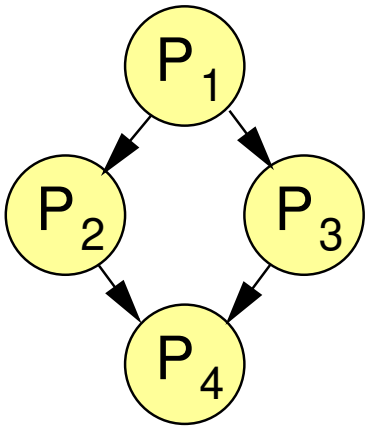
→ Dann können alle Prozesse gleichzeitig gestartet werden. Die Vorrangrelation zwischen ihnen wird korrekt durchgesetzt.

[4] *directed acyclic graph* - DAG

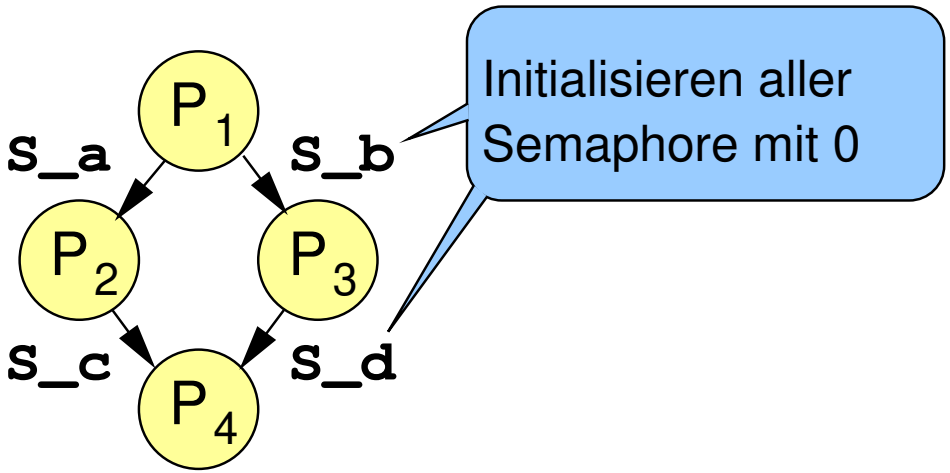
Beispiel: Vorrangrelation

Hochschule RheinMain

Gegebenes Prozessystem



Lösung




```
^I^^I^^IP1() {
^I^^I^^I.. wor
^I^^I^^I V(S_a
^I^^I^^I V(S_b
^I^^I^^I exit(
^I^^I^^I}
^I^^I^^I
```

```
^I^^I^^IP2() {
^I^^I^^I P(S_a
^I^^I^^I.. wor
^I^^I^^I V(S_c
^I^^I^^I exit(
^I^^I^^I}
^I^^I^^I
```

```
^I^^I^^IP3() {
^I^^I^^I P(S_b
^I^^I^^I.. wor
^I^^I^^I V(S_c
^I^^I^^I exit(
^I^^I^^I}
^I^^I^^I
```

```
^I^^I^^IP4() {
^I^^I^^I P(S_c
^I^^I^^I P(S_d
^I^^I^^I.. wor
^I^^I^^I exit(
^I^^I^^I}
^I^^I^^I
```

P() und V() in Betriebssystemen

- Die meisten (alle?) Betriebssysteme bieten Dienste zum Umgang mit Semaphoren
- In der Regel heißen diese aber *nicht* P() und V()
- Beispiele:
  - POSIX Threads Interface (pthreads): Mutex-Funktionen `phtread_mutex_xxx()` implementieren binäre Semaphoren (vgl. 5.2.2)
  - UNIX System V Release 4 (SVR4): Funktionen `semget()`, `semop()`, `semctl()` arbeiten auf *Mengen* von semaphoren
  - Windows NT: Zählsemaphore als Systemobjekte, Funktionen: `CreateSemaphore()`, `ReleaseSemaphore()`, `OpenSemaphore()`, `WaitForSingleObject()`

Pthread Mutex Beispiel (1)

- Ziel: Alle Threads führen ihren „kritischen Abschnitt“ (in rot) jeweils nacheinaner ohne Überschneidungen aus.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread(void *arg)
{ /* hier: arg ist Zeiger auf integer (s.u.) */
^^Iint i, num = *(int*)arg;
^^Ipthread_mutex_lock(&mutex);   /* prolog */
^^Ifor(i = 0; i < 5; i++)
^^I{
^^I^^Iprintf("Thread %d mit i=%d\n", num, i);
^^I^^Isleep(1);
^^I}
^^Ipthread_mutex_unlock(&mutex); /* epilog */
^^Ipthread_exit((void*)0);
}
^^I^^I
```

Pthread Mutex Beispiel (2)

- main()-Programm dazu...

```
#define THREADS 5
int main()
{
^^Ivoid *rc;
^^Ipthread_t threads[THREADS];
^^Iint arg[THREADS];
^^Ifor(int i = 0; i < THREADS; i++)
^^I{   /* erzeuge 5 Threads */
^^I^^Iarg[i] = i; /* Thread-Nummer als Argument uebergeben */
^^I^^Iif(!!pthread_create(&threads[i], NULL, thread, &arg[i]))
^^I^^I{
^^I^^I^^Ifprintf (stderr, "Konnte Thread %d nicht starten\n", i);
^^I^^I^^Iexit(EXIT_FAILURE);
^^I^^I}
^^I}
^^Ifor(int i = 0; i < THREADS; i++)
^^I{   /* Auf Ende von 5 Threads warten */
^^I^^Ipthread_join(threads[i], &rc);
^^I}
^^Ireturn(EXIT_SUCCESS);
}

^^I^^I
```

Höhere Synchronisationsprimitive

Condition Variable

- Synchronisationsobjekt des Betriebssystems.
- Warteschlange für wartende Threads.
- Wechselseitiger Ausschluss durch „Support-Mutex“
- Mutex wird beim Warten implizit freigegeben:

```
cond_wait(cond_t *cv, mutex_t *m) {
    mutex_unlock(m);
    warte_im_betriebssystem(cv, ...);
    mutex_lock(m);
}
^^I^^I^^I^^I

5
```

[5] Nur zur Verdeutlichung: **Kein** echter Programmcode

POSIX Condition Variables

- Synchronisation von Threads basierend auf Bedingung über aktuellem Wert einer Variable
- Benutzung immer im Zusammenhang mit assoziiertem Mutex
- Condition Variable
  - deklarieren vom Typ `pthread_cond_t`
  - `pthread_cond_init()` (Initialisieren)
  - `pthread_cond_destroy()` (Zerstören)
- Warten und Signalisieren
  - `pthread_cond_wait(condition,mutex)` Blockieren bis Bedingung signalisiert wird
  - `pthread_cond_signal(condition)` Signalisieren der Bedingung (Wecken mindestens eines anderen Threads)
  - `pthread_cond_broadcast(condition)` Signalisieren der Bedingung (Wecken aller wartenden Threads)
- Details siehe Manpages oder:
  `https://computing.llnl.gov/tutorials/pthreads/`

POSIX Condition Variables

Hochschule RheinMain

Typische Verwendung

```
^I^^I^^I/* gemeinsame Daten */
^I^^I^^Ipthread_mutex_t m;
^I^^I^^Ipthread_cond_t cv;
^I^^I^^IBool tuwas = FALSE;
^I^^I
```

```
^I^^I^^I// Wartender Thread
^I^^I^^Ipthread_mutex_lock(&m);
^I^^I^^I...
^I^^I^^Iwhile (!tuwas) {
^I^^I^^I^^Ipthread_cond_wait(&cv, &
^I^^I^^I}
^I^^I^^I...
^I^^I^^Ipthread_mutex_unlock(&m);
^I^^I^^I
```

```
^I^^I^^I// Signalisierender Thread
^I^^I^^Ipthread_mutex_lock(&m);
^I^^I^^I...
^I^^I^^Ituwas = TRUE;
^I^^I^^Ipthread_cond_signal(&cv);
^I^^I^^I...
^I^^I^^I...
^I^^I^^Ipthread_mutex_unlock(&m);
^I^^I^^I
```

Monitore

- Vorschlag Hoare (1974), Brinch Hansen (1975):
- In Programmiersprache eingebettete Primitive zur einfacheren Entwicklung korrekter nebenläufiger Programme.
- Ein **Monitor** besteht aus
  - Variablen und Datenstrukturen (die das eigentliche Betriebsmittel darstellen)
  - Sichtbaren Operationen zur Manipulation der Daten.
  - Wechselseitiger Ausschluss in der Ausführung der Operationen ist garantiert, d.h. zu jedem Zeitpunkt ist höchstens ein Prozess im Monitor aktiv.
- Code zur Sicherstellung des wechselseitigen Ausschlusses wird durch den Compiler erzeugt, z.B. auf der Basis von Semaphoren.
- Heutige Sicht:
  - Monitor entspricht einer Instanz eines abstrakten Datentyps mit automatischem wechselseitigen Ausschluss
  - Vergleiche Java „synchronized instance method“

Monitore (2)

- Programmierung von Synchronisationsvorgängen innerhalb von Monitoren mit interner Condition Variablen mit Operationen `wait()` und `signal()`.
- `wait(cv)` blockiert den Aufrufer, ein evtl. wartender anderer Aufrufer einer Monitor-Operation kann nun den Monitor betreten.
- `signal(cv)` weckt einen Prozess (aus der Sicht des Monitors zufällig aus der Menge der Wartenden ausgewählt), der Aufrufer muss den Monitor sofort verlassen (Annahme: letzte Anweisung, Brinch Hansen-Modell). Falls kein Prozess wartet, ist `signal(cv)` ohne Wirkung (keine Zähler-Semantik!).
- Monitore haben kaum Eingang in Programmiersprachen gefunden (Gegenbeispiele: Mesa (Xerox), eingeschränkt Java synchronized)).

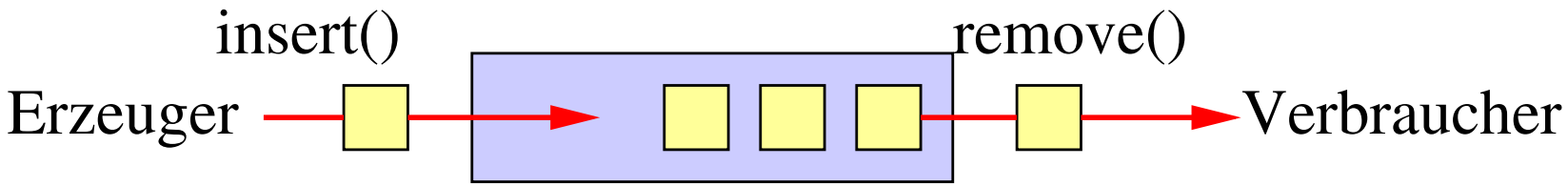
Monitore in Java

Keyword: `synchronized`

- macht aus einer Methode einen Monitor
  → impliziter kritischer Bereich
- `wait()`
  - blockiert den aufrufenden Thread
  - Monitor ist danach wieder frei
- `notify()`
  - weckt einen beliebigen Thread
- `notifyAll()`
  - weckt alle wartenden Threads

Beispiel: Erzeuger/Verbraucher-Problem

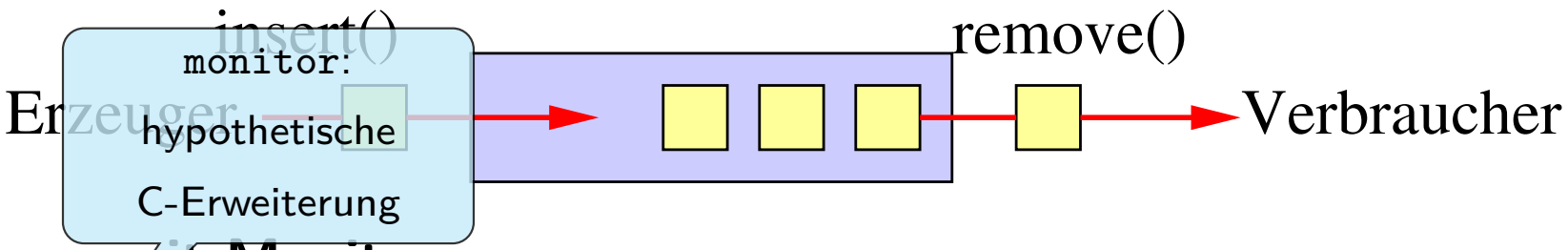
Hochschule RheinMain



Lösung mit Monitor

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Beispiel: Erzeuger/Verbraucher-Problem

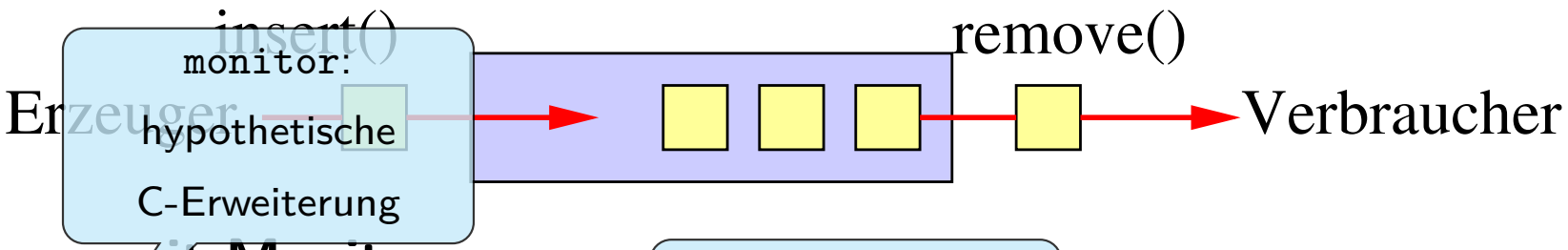


Lösung mit Monitor

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Beispiel: Erzeuger/Verbraucher-Problem

Hochschule RheinMain



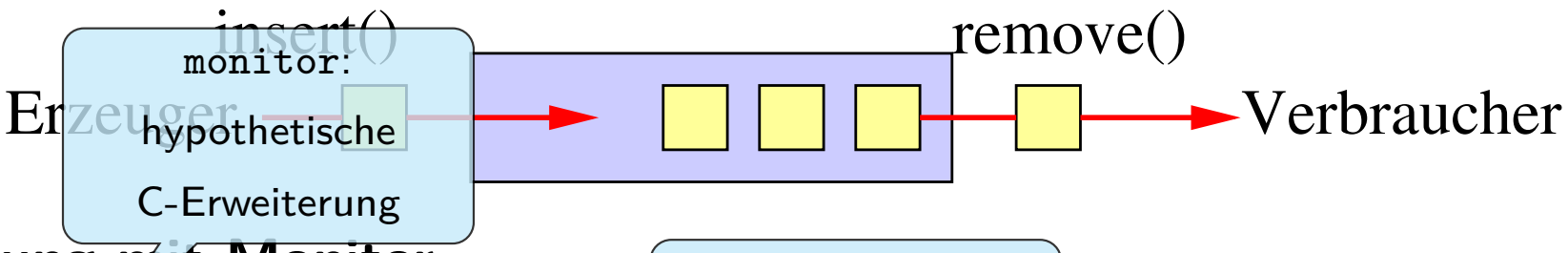
Lösung mit Monitor

Erzeuger muß warten, solange der Puffer voll ist

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Beispiel: Erzeuger/Verbraucher-Problem

Hochschule RheinMain



Lösung mit Monitor

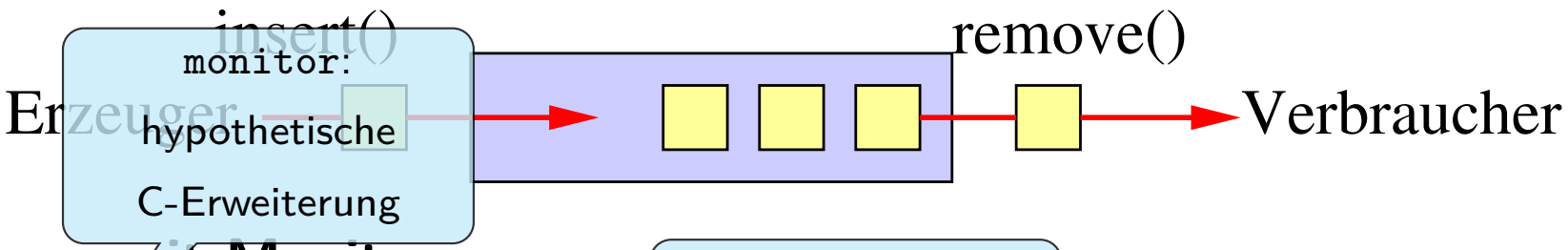
```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Ein wegen der empty- Bedingung wartender Prozess, könnte nun weitermachen

Beispiel: Erzeuger/Verbraucher-Problem

Hochschule RheinMain



Lösung mit Monitor

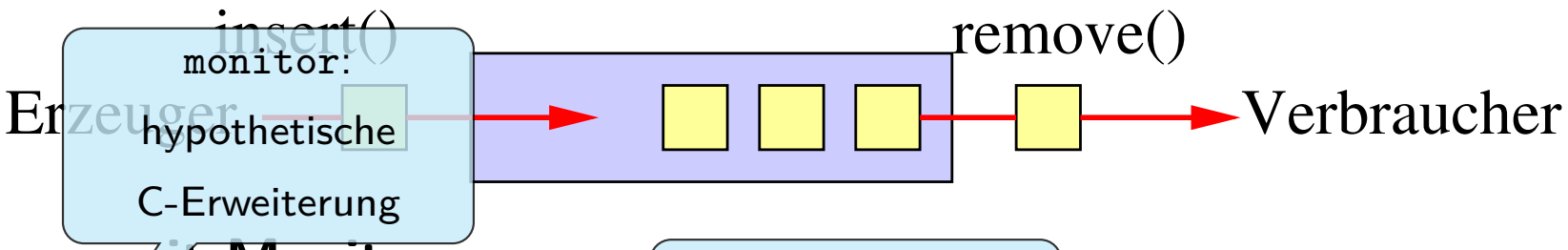
```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Der Verbraucher muß warten, solange der Puffer leer ist

Ein wegen der empty- Bedingung wartender Prozess, könnte nun weitermachen

Beispiel: Erzeuger/Verbraucher-Problem



Lösung mit Monitor

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Der Verbraucher muß warten, solange der Puffer leer ist

Ein wegen der empty-Bedingung wartender Prozess, könnte nun weitermachen

Ein wegen der full-Bedingung blockierter Prozess, könnte nun weitermachen

Weitere Ansätze

Weitere, hier nicht besprochene Synchronisationsprimitive:

- Eventcounts: Reed, Kanodia (1979)
- Serializer: Atkinson, Hewitt (1979)
- Objekte mit Pfadausdrücken (path expressions) zur Festlegung von zulässigen Ausführungsfolgen der Operationen einschl. deren Nebenläufigkeit:
- Read/Write-Locks (vgl. Datenbanken).
- Barrieren
- fork/join

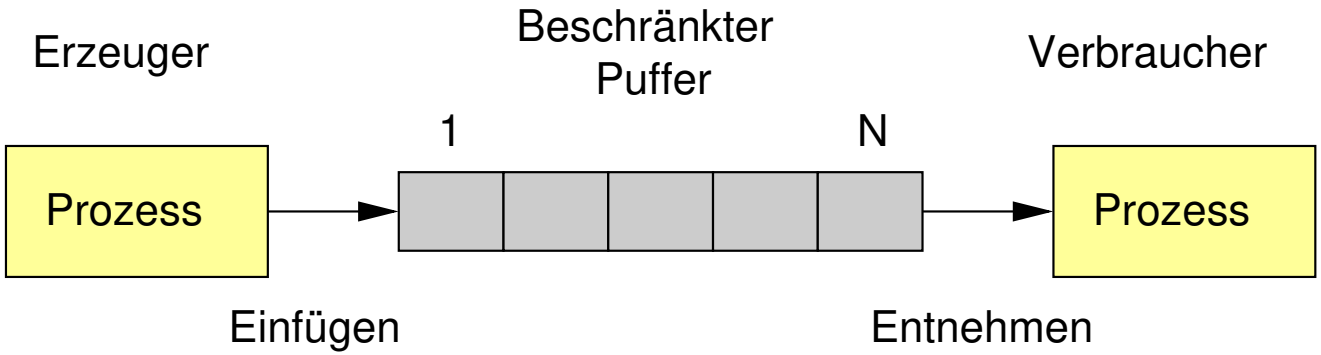
Literatur hierzu: z.B. Tanenbaum, Peterson/Silberschatz, Maekawa et. al.

Klassische Synchronisationsprobleme

Gliederung

- Erzeuger-Verbraucher-Problem
- Philosophen-Problem
- Leser-Schreiber-Problem

Erzeuger-Verbraucher-Problem



Probleme

- Erzeuger: Will Einfügen, aber Puffer ist voll.
  Lösung: Lege dich schlafen, lass dich vom Verbraucher wecken, wenn er ein Datum entnommen hat.
- Verbraucher: Will Entnehmen, aber Puffer ist leer.
  Lösung: Lege dich schlafen, lass dich vom Erzeuger wecken, wenn er ein Datum eingefügt hat.

Erzeuger-Verbraucher-Problem (2)

Hochschule RheinMain

- Problem wird auch „Problem des beschränkten Puffers“ (*bounded buffer problem*) genannt.
- Sieht einfach aus, enthält aber kritische Abläufe.
- Mögliche Erweiterung: mehrere Erzeuger, mehrere Verbraucher.

Betrachtete Lösungsansätze:

1. Lösung mit Semaphoren
2. Lösung mit Monitoren
3. Lösung mit Nachrichtenaustausch

Lösung mit Semaphoren (1)

Wurde bereits besprochen (vgl. 5.2.3)

```
#define N 100                   /* Kapazitaet des Puffers          */

/* gemeinsame Variablen
*/
semaphore mutex = 1;            /* kontrolliert krit. Bereich      */
semaphore empty = N;            /* zaehlt leere Eintraege          */
semaphore full  = 0;            /* zaehlt belegte Eintraege        */

void erzeuger(void)             /* Erzeuger                        */
{
^^Iint item;
^^Iwhile (TRUE) {
^^I^^Iproduce_item(&item);      /* erzeuge Eintrag                 */
^^I^^IP(&empty);                /* besorge freien Platz            */
^^I^^IP(&mutex);                /* tritt in krit. Abschnitt ein    */
^^I^^Ienter_item(item);         /* fuege Eintrag in Puffer ein     */
^^I^^IV(&mutex);                /* verlasse krit. Bereich          */
^^I^^IV(&full);                 /* erhoehe Anz. belegter Eintr.    */
^^I}
}
```

Lösung mit Semaphoren (2)

Hochschule RheinMain

```
void verbraucher(void)/* Verbraucher                          */
{
^^Iint item;
^^Iwhile(TRUE) {
^^I^^IP(&full);            /* belegter Eintrag vorhanden?   */
^^I^^IP(&mutex);           /* tritt in krit. Abschnitt ein  */
^^I^^Iremove_item(&item); /* entnimm Eintrag aus Puffer     */
^^I^^IV(&mutex);           /* verlasse krit. Bereich        */
^^I^^IV(&empty);           /* erhoehe Anz. freier Eintraege */
^^I^^Iconsume_item(item); /* verarbeite Eintrag             */
^^I}
}
```

- Semaphore `mutex` wird zur **Durchsetzung des wechselseitigen Ausschlusses** in der Benutzung des Puffers verwendet.
- Semaphore `empty` und `full` werden zur **Synchronisation von Erzeuger und Verbraucher** benutzt, um bestimmte Operationsreihenfolgen zu erreichen bzw. zu vermeiden.
  (Beispiel: Erzeuger: `P(&empty)`, Verbraucher: `V(&empty)` → Wenn Puffer voll: ⇒ Erzeuger blockiert, bis Verbraucher einen Eintrag entnimmt.)
- Algorithmen gelten unverändert auch für mehrere Erzeuger und/oder Verbraucher.

Lösung mit Monitor (1)

Wurde bereits besprochen (vgl. 5.2.4)

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    enter_item(item);
    count++;
    if (count == 1) signal(empty);
  }

  int remove(int *item_p) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    remove_item(item_p);
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Lösung mit Monitor (1)

Hochschule RheinMain

Wurde bereits besprochen (vgl. 5.2.4)

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    enter_item(item);
    count++;
    if (count == 1) signal(empty);
  }

  int remove(int *item_p) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    remove_item(item_p);
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Lösung mit Monitor (1)

Wurde bereits besprochen (vgl. 5.2.4)

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    enter_item(item);
    count++;
    if (count == 1) signal(empty);
  }

  int remove(int *item_p) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    remove_item(item_p);
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Ein wegen der empty-Bedingung wartender Prozess, könnte nun weitermachen

Lösung mit Monitor (1)

Hochschule RheinMain

Wurde bereits besprochen (vgl. 5.2.4)

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    enter_item(item);
    count++;
    if (count == 1) signal(empty);
  }

  int remove(int *item_p) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    remove_item(item_p);
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Ein wegen der empty-Bedingung wartender Prozess, könnte nun weitermachen

Der Verbraucher muß warten, solange der Puffer leer ist

Lösung mit Monitor (1)

Wurde bereits besprochen (vgl. 5.2.4)

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    enter_item(item);
    count++;
    if (count == 1) signal(empty);
  }

  int remove(int *item_p) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    remove_item(item_p);
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Ein wegen der empty-Bedingung wartender Prozess, könnte nun weitermachen

Der Verbraucher muß warten, solange der Puffer leer ist

Ein wegen der full-Bedingung blockierter Prozess, könnte nun weitermachen

Lösung mit Monitor (2)

- **Beachte**: Die Ausführung der Operationen `insert` und `remove` ist wechselseitig ausgeschlossen (Monitor!).
- Die Variable `count` gibt die augenblickliche Anzahl der belegten Pufferplätze an.
- `full` und `empty` sind Condition Variablen, **keine Zähler**.
- Algorithmen gelten unverändert auch für mehrere Erzeuger und/oder Verbraucher.

Lösung mit Nachrichtenaustausch (1)

Hochschule RheinMain

Grundlage: Funktionen zum Nachrichtenaustausch

- `send(process, int* message)` - Nachricht an Prozess senden
- `receive(process, int* message)` - Nachricht von Prozess empfangen

```
#define N      100                  /* Kapazitaet des Puffers          */
#define MSIZE    4                  /* Nachrichtengroesse              */

typedef int message[MSIZE];         /* Nachrichtentyp                  */

void producer(void)                 /* Erzeuger                        */
{
   int item;
   message m;

   while (TRUE) {
      produce_item(&item);          /* erzeuge Eintrag                 */
      receive(consumer, &m);        /* warte auf leere Nachricht       */
      build_message(&m, item);      /* erzeuge zu sendende Nachricht*/
      send(consumer, &m);           /* sende Nachricht z Verbraucher*/
   }
}
^^I^^I
```

Lösung mit Nachrichtenaustausch (2)

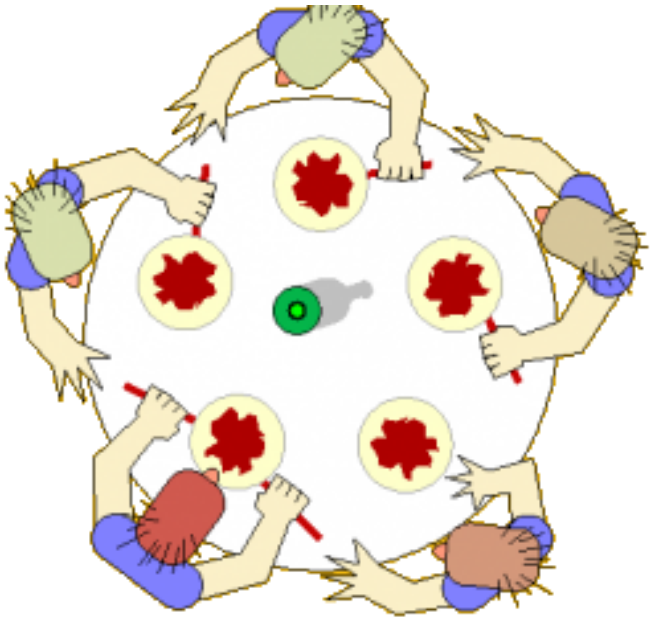

```
void consumer(void)              /* Verbraucher                    */
{
   int item, i;
   message m;

   for (i = 0; i < N; i++)       /* sende N leere Nachrichten      */
      send (producer, &m);

   while (TRUE) {
      receive(producer, &m);     /* empfange Nachricht v Erzeuger*/
      extract_item(&m, &item);   /* entnimm Eintrag                */
      send(producer, &m);        /* sende leere Nachricht zurueck*/
      consume_item(item);        /* verarbeite Eintrag             */
   }
}
^^I^^I
```

- Beachte: **Kein** gemeinsamer Speicher zw. Erzeuger und Verbraucher.
- **Annahmen**: Nachrichten haben feste Länge. Gesendete, noch nicht empfangene Nachrichten werden automatisch gepuffert.
- Insgesamt `N` Nachrichten-„Hülsen“ (Umschläge) werden benutzt. Leere Nachrichten werden vom Verbraucher an den Erzeuger gesendet, der Erzeuger füllt leere Nachrichten mit Daten und sendet „gefüllte Umschläge“ an den Verbraucher.
- Wegen direkter Adressierung sind diese Algorithmen nur für **einen** Erzeuger und **einen** Verbraucher geeignet.

Dining Philosophers Problem

Hochschule RheinMain

Dijkstra (1965): *dining philosophers problem*.



- 5 PhilosophInnen sitzen am runden Tisch. Jede(r) hat einen Teller, zwischen je zwei benachbarten Tellern liegt eine Gabel. Linke und rechte Gabel werden zum Essen benötigt. Nach dem Essen werden beide Gabeln abgelegt. Essen und Denken wechseln einander fortlaufend ab.
- Wie lautet der Algorithmus für die PhilosophInnen?

Eine zu einfache, **fehlerhafte** Lösung

Hochschule RheinMain

```
#define N 5                     /* Anzahl der PhilosophInnen */

void philosopher(int i)         /* i:0..N-1, welche(r)? */
{
    while (TRUE) {
        think();                /* Denken                */
        take_fork(i);           /* Greife linke Gabel    */
        take_fork((i+1)%N);     /* Greife rechte Gabel   */
        eat();                  /* Essen                 */
        put_fork(i);            /* Ablegen linke Gabel   */
        put_fork((i+1)%N);      /* Ablegen rechte Gabel  */
    }
}
^^I^^I
```

Gedankenexperiment: Was passiert wenn ...

- ... Alle die linke Gabel durch `take_fork(i)` nehmen,
- ... und dann die rechte Gabel durch `take_fork((i+1)%N)` zu nehmen versuchen?

→ Dann sind alle für immer blockiert!

→ Es liegt ein sogenannter **Deadlock** vor.

Verbesserte, immer noch fehlerhafte Lösung

Hochschule RheinMain

- Verbesserung: Nach dem Aufnehmen der linken Gabel prüfe, ob rechte Gabel verfügbar ist. Falls nicht, lege linke Gabel ab, warte eine Zeitlang, dann beginne von vorn.

```
#define N 5                 /* Anzahl der PhilosophInnen */

void philosopher(int i)                 /* i:0..N-1, welche(r)? */
{
^^Iwhile (TRUE) {
^^I^^Ithink();                          /* Denken               */
^^I^^Ido {
^^I^^I^^Itake_fork(i);                   /* Greife linke Gabel   */
^^I^^I^^Iok = try_take_fork((i+1)%N);    /* Versuche rechte Gabel*/
^^I^^I^^Iif(!ok) {                       /* Erfolg?              */
^^I^^I^^I^^Iput_fork(i);                  /* nein -> linke zurueck*/
^^I^^I^^I^^Iwait();                       /* warte                */
^^I^^I^^I}
^^I^^I} while(!ok);
^^I^^Ieat();                    /* Essen                */
^^I^^Iput_fork(i);              /* Ablegen linke Gabel  */
^^I^^Iput_fork((i+1)%N);        /* Ablegen rechte Gabel */
^^I}
}
^^I^^I
```

- Dieser Ansatz vermeidet den alten Fehler (Deadlock), enthält aber einen neuen:
- **Gedankenexperiment:** Alle greifen gleichzeitig die linke Gabel, erkennen, dass die rechte nicht verfügbar ist, legen die linke wieder ab, warten gleich lang, greifen wieder gleichzeitig die linke usw.
- Dieses Problem der endlosen Ausführung ohne Fortschritt wird „Verhungern“ (*Starvation*) genannt.

Korrekte, aber **unbefriedigende** Lösung


```
#define N 5                     /* Anzahl der PhilosophInnen */
semaphore mutex = 1;

void philosopher(int i)         /* i:0..N-1, welche(r)? */
{
^^Iwhile (TRUE) {
^^I^^Ithink();                  /* Denken                */
^^I^^IP(&mutex);                /* Beginn krit. Bereich */
^^I^^Itake_fork(i);             /* Greife linke Gabel    */
^^I^^Itake_fork((i+1)%N);       /* Greife rechte Gabel   */
^^I^^Ieat();                    /* Essen                 */
^^I^^Iput_fork(i);              /* Ablegen linke Gabel   */
^^I^^Iput_fork((i+1)%N);        /* Ablegen rechte Gabel */
^^I^^IV(&mutex);                /* Ende krit. Bereich    */
^^I}
}
^^I^^I
```

- Semaphore schützt gesamten „Ess-Abschnitt“
- Kein Deadlock, aber unbefriedigend:
  nur ein(e) PhilosophIn kann gleichzeitig essen !

Lösung mit Semaphoren (1)

Hochschule RheinMain

```
#define N               5^^I^^I/* Anzahl der Philosophen                */
#define LEFT  (i-1+N)%N^^I/* Nummer des linken Nachbarn von i    */
#define RIGHT (i+1)%N^^I^^I/* Nummer des rechten Nachbarn von i   */
#define THINKING    0^^I^^I/* Zustand: Denkend                    */
#define HUNGRY      1^^I^^I/* Zust: Versucht, Gabeln zu bekommen */
#define EATING      2^^I^^I/* Zustand: Essend                     */

/* gemeinsame Variablen
*/
int state[N];^^I^^I^^I^^I/* Zustaende aller PhilosophInnen          */
semaphore mutex = 1;^^I/* fuer wechselseitigen Aussschluss    */
semaphore s[n];^^I^^I^^I^^I/* Semaphor fuer jeden Philosoph       */

void philosopher(int i) {^^I/* i:0..N-1, welcher Philosoph      */
^^Iwhile (TRUE) {
^^I^^Ithink();^^I^^I^^I^^I/* Denken                             */
^^I^^Itake_forks(i);^^I/* Greife beide Gabeln oder blockiere */
^^I^^Ieat();^^I^^I^^I^^I^^I/* Essen                            */
^^I^^Iput_forks(i);^^I^^I/* Ablegen beider Gabeln              */
   }
}
^^I^^I
```

Lösung mit Semaphoren (1)

Hochschule RheinMain

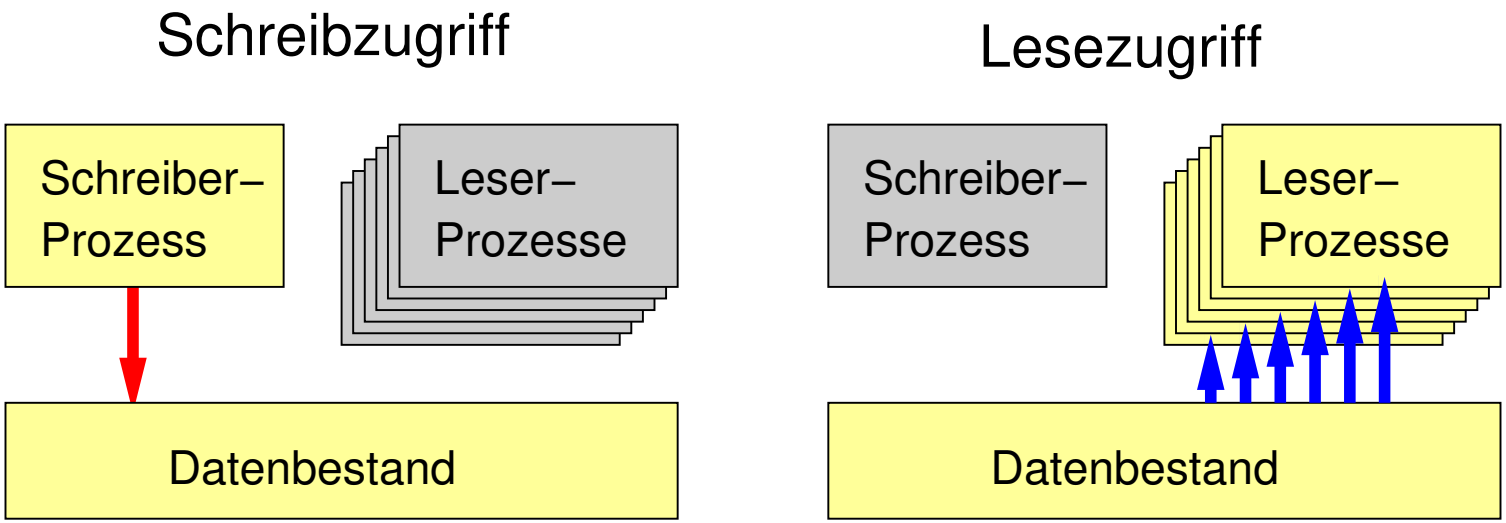
```
void take_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?  */
{
^^IP(&mutex);          /* tritt in krit. Bereich ein          */
^^Istate[i] = HUNGRY;  /* zeige, dass du hungrig bist         */
^^Itest(i);            /* versuche, beide Gabeln zu bekommen */
^^IV(&mutex);          /* verlasse krit. Bereich              */
^^IP(&s[i]);           /* bockiere, falls Gabeln nicht frei   */
}

void put_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?    */
{
^^IP(&mutex);          /* tritt in krit. Bereich ein          */
^^Istate[i] = THINKING;/* zeige, dass du fertig bist          */
^^Itest(LEFT);         /* kann linker Nachbar jetzt essen ?   */
^^Itest(RIGHT);        /* kann rechter Nachbar jetzt essen ? */
^^IV(&mutex);          /* verlasse krit. Bereich              */
}

void test(int i) /* i:0..N-1, welche(r) PhilosophIn? */
{
^^Iif (state[i]==HUNGRY &&
^^I    state[LEFT]!=EATING && state[RIGHT]!=EATING) {
^^I^^Istate[i]=EATING;/* jetzt kann Phil i essen !           */
^^I^^IV(&s[i]);       /* "sage es ihm"                       */
^^I}
}
^^I^^I
```

Bemerkungen

Hochschule RheinMain

- Jeder Prozess führt die Prozedur `philosopher` als Hauptprogramm aus, die anderen Prozeduren sind gewöhnliche Unterprogramme, keine separaten Prozesse.
- Das Array `state` speichert die aktuellen Zustände der Philosophen: essend, denkend, hungernd (versucht, Gabeln zu bekommen).
- Jeder Prozess blockiert an einem ihm zugeordneten Semaphor `s[i]`, wenn die benötigten Gabeln nicht verfügbar sind.
- Das Semaphor `mutex` sichert den kritischen Abschnitt der Benutzung der Zustandsinformation.
- **Die Lösung ist korrekt**, sie enthält keinen Deadlock und kein Verhungern.
- Die Lösung lässt eine möglichst hohe Nebenläufigkeit zwischen den Philosophen zu.
- Die Lösung ist für eine beliebige Anzahl von PhilosophInnen korrekt.

Leser-Schreiber-Problem



- Zu jedem Zeitpunkt dürfen entweder mehrere Leser oder ein Schreiber zugreifen.
- Verboten: gleichzeitiges Lesen und Schreiben
- Wie sollten Leser- und Schreiber-Programme aussehen?

Lösung mit Semaphoren (1)

Lesezugriff:

```
/* gemeinsame Variablen: */
semaphore mutex = 1;        /* wechsels. Aussschluss fuer rc      */
semaphore db = 1;           /* Semaphor fuer Datenbestand         */
int rc = 0;                 /* readcount: Anzahl Leser            */

void reader(void)           /* Leser                              */
{
   while (TRUE) {
      P(&mutex);            /* erhalten exkl. Zugriff auf rc      */
      rc = rc + 1;          /* ein zusaetzlicher Leser            */
      if (rc==1)            /* Erster Leser?                      */
          P(&db);           /* ja -> reserviere Daten             */
      V(&mutex);            /* freigeben exkl. Zugriff auf rc     */

      read_data_base();     /* lies Datenbestand                  */

      P(&mutex);            /* erhalten exkl. Zugriff auf rc      */
      rc = rc - 1;          /* ein Leser weniger                  */
      if (rc==0)            /* letzter Leser ?                    */
          V(&db);           /* ja -> Daten freigeb.               */
      V(&mutex);            /* freigeben exkl. Zugriff auf rc     */
      use_data_read();      /* unkrit. Bereich                    */
   }
}
^^I^^I
```

Lösung mit Semaphoren (2)

Schreibzugriff:

```
void writer(void)          /* Schreiber                              */
{
   while (TRUE) {
      create_data();       /* unkrit. Bereich                        */
      P(&db);              /* erhalten exkl. Zugriff auf Daten       */
      write_data_base();   /* schreibe Datenbestand                  */
      V(&db);              /* freigeben exkl. Zugriff auf Daten      */
   }
}
^^I^^I
```

- `mutex` sichert krit. Abschnitt bezüglich des Read-Counters `rc`.
- `db` sichert Zugriff auf den Datenbestand, so dass **entweder** mehrere Leser **oder** ein Schreiber zugreifen können.
- Der erste Leser führt eine P-Operation auf `db` aus, alle weiteren inkrementieren nur `rc`.
- Der letzte Leser führt eine V-Operation auf `db` aus, so dass ein wartender Schreiber Zugriff erhält.
- Die **Lösung bevorzugt Leser**: Neu eintreffende Leser erhalten Zugriff vor einem schon wartenden Schreiber, wenn noch mindestens ein Leser Zugriff hat.

Zusammenfassung (1)

Was haben wir in Kap. 5 gemacht?

- Interaktionen zwischen Prozessen können zu zeitkritischen Abläufen führen, d.h. Situationen, in denen das Ergebnis vom zeitlichen Ablauf abhängt. Zeitkritische Abläufe führen zu einem nicht reproduzierbaren Verhalten und müssen vermieden werden.
- Kritische Bereiche als Teile von Programmen, in denen mit anderen Prozessen gemeinsamer Zustand manipuliert wird, bieten die Möglichkeit des wechselseitigen Ausschlusses. Sie vermeiden damit zeitkritische Abläufe und erlauben komplexe unteilbare (atomare) Aktionen.

Zusammenfassung (2)

- Viele Primitive zur Synchronisation und Kommunikation von Prozessen wurden vorgeschlagen. Sie machen verschiedene Annahmen über die unterlagerten elementaren unteilbaren Operationen, sind aber im Prinzip gleich mächtig. Besprochen wurden insbesondere Semaphoren, Monitore, Condition Variablen und Nachrichtenaustausch, die in aktuellen Systemen weit verbreitet sind.
- Es gibt eine Reihe von klassischen Problemen der Interprozesskommunikation, an denen die Nutzbarkeit neuer vorgeschlagener Primitive gezeigt wird. Von diesen wurden das Erzeuger-Verbraucher-Problem, das Philosophen-Problem und das Leser-Schreiber-Problem besprochen.
- Auch mit den heute üblichen Primitiven muss sorgfältig umgegangen werden, um inkorrekte Lösungen, Deadlocks und Starvation zu vermeiden.