Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Kap. 5: Prozesssynchronisation

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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
University of Applied Sciences
Wiesbaden Rüsselsheim

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

Prozess 1

```
/* Gehaltsüberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

Prozess 2

```
/* Dauerauftrag Miete */
x = lies_kontostand();
x = x - 800;
schreibe_kontostand(x);
```

Möglicher Ablauf

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Prozess 1

```
/* Gehaltsüberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

Prozess 2

```
/* Dauerauftrag Miete */
x = lies_kontostand();
x = x - 800;
schreibe_kontostand(x);
```

Mögliche Ausführungsreihenfolge der Anweisungen in Prozess 1,2

```
/* Gehaltsüberweisung */        /* Dauerauftrag Miete */
z = lies_kontostand();
                                x = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
                                x = x - 800;
                                schreibe_kontostand(x);
```

Pech, die Gehaltsüberweisung ist „verloren gegangen“ :-(

Bei anderen Reihenfolgen werden die beiden Berechnungen „richtig“ ausgeführt, oder es geht der Dauerauftrag verloren.

Definitionen (1)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Annahme: Prozesse mit Lese/Schreib-Operationen auf Betriebsmitteln

Def

- 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.

Def

- 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
University of Applied Sciences
Wiesbaden Rüsselsheim

Def

- Verfahren zum wechselseitigen Ausschluss (engl. mutual exclusion):
  Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess zugreift.
- Bemerkung: Ein Verfahren zum wechselseitigen Ausschluss vermeidet zeitkritische Abläufe. Es löst damit ein Basisproblem des Concurrent Programming.

Def

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

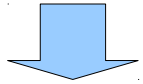
Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Nicht atomar

```
/* Gehaltsüb  weisung */
z = lies_kont ostand();
z = z + 1000;
schreibe_kontostand(z);
```

Atomar ?

```
/* Gehaltsüberweisung  2.0 */
int kontostand;

kontostand += 1000;
```



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

Teilweise architekturabhängig

ASM:

3 Instruktionen, unterbrechbar!

```
lw r2,kontostand
addu r3,r2,#1000
sw r3,kontostand
```

(Nicht-)atomare Operationen

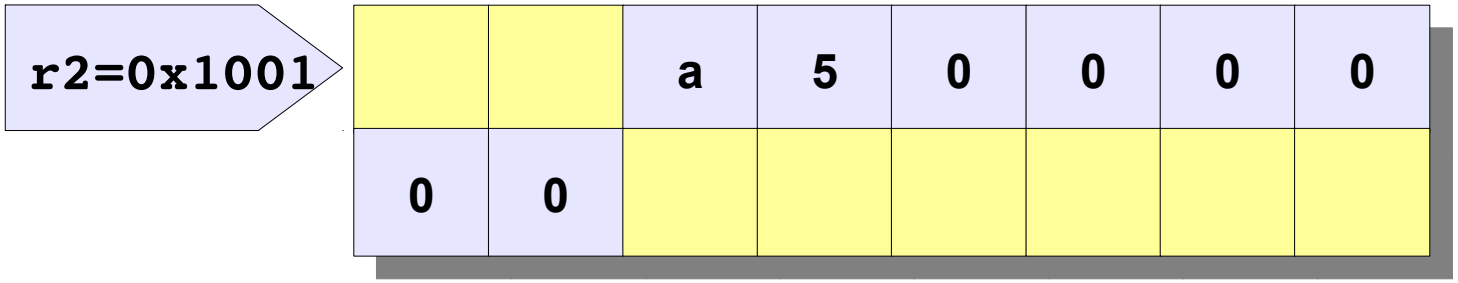
Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

Beispiel:

```
ASM:
la r2,0x1001
sw 0xa5000000,(r2)
```

Ein Maschinenbefehl
=> atomar?



Ungerade Adresse => Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich => unterbrechbar!

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 (→ „Verhungern“, engl. starvation).

Gliederung

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

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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.

Lösungsansätze

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

1. Sperren aller Unterbrechungen
2. Sperrvariablen
3. Striktes Alternieren
4. Lösung von Peterson (+)
5. Test-and-Set-Instruktion (+)

\+ brauchbar

5.2.1.1 Sperren aller Unterbrechungen

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Einfachste Lösung:

- Jeder Prozess sperrt vor Eintritt in seinen kritischen Abschnitt alle Unterbrechungen (Disable Interrupts).
- Jeder Prozess lässt die Unterbrechungen am Ende seines kritischen Abschnitts wieder zu (Enable Interrupts).

Sperren aller Unterbrechungen (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Bemerkungen:

- Der Prozessor kann nur dann zu einem anderen Prozess wechseln, wenn eine Unterbrechung auftritt. Also ist für diese Lösung die Korrektheitsforderung erfüllt.
- Lösung ist unbrauchbar für allgemeine Benutzerprozesse, da nicht zugesichert werden kann, dass sie die Interrupts auch wieder zulassen (z.B. wegen Programmierfehler).
- Lösung wird häufig innerhalb des Betriebssystemkerns selbst eingesetzt, um wechselseitigen Ausschluss zwischen Kernroutinen zu gewährleisten (z.B. im alten Einprozessor-UNIX-Kern, um wechselseitigen Ausschluss mit einem Interrupt-Handler sicherzustellen).
- Lösung unbrauchbar im Falle eines Multiprozessor-Systems, da sich die Interrupt-Sperre i.d.R. nur auf einen Prozessor auswirkt.

5.2.1.2 Sperrvariablen

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Einfacher, falscher Lösungsansatz:

- Jedem krit. 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

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

Prozess 2

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

Sperrvariablen (2)

Bemerkungen:

- Prinzipiell gleicher Fehler wie bei 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 nur unteilbare Lese- und Schreibzugriffe (Eigenschaft der Architektur des Speicherwerks).

Prozess 1

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

Prozess 2

```
while (sperrvar) { }
sperrvar = 1;
/* kritischer Bereich */
sperrvar = 0;
```

5.2.1.3 Striktes Alternieren

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Prozess 0

```
int turn = 0;
```

Prozess 1

```
while (TRUE) {
  while (turn != 0) /* Warte */
      ;
  critical_section();
  turn = 1;
  noncritical_section()
}
```

```
while (TRUE) {
  while (turn != 1) /* Warte */
      ;
  critical_section();
  turn = 0;
  noncritical_section()
}
```

- Gem. Variable „turn“ gibt an, welcher Prozess den kritischen Bereich betreten darf
- Warten wird aktiv durchgeführt.
- Prozesse wechseln sich ab: 0,1,0,1,...
- Lösung erfüllt Korrektheitsbedingung, aber
- Fortschrittsbedingung (3) kann verletzt sein, wenn ein Prozess wesentlich langsamer als der andere ist.
- Fazit: keine ernsthafte Lösung.

5.2.1.4 Lösung von Peterson

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Historische Vorläufer:
  - Anfang der 60er Jahre viele Lösungsansätze, nur wenige erfüllten alle Bedingungen aus 5.1.
  - 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_region()`, Epilog: `leave_region()`
- Weitere Lösungen für mehr als zwei Prozesse von:
  - Dijkstra, Peterson, Knuth, Eisenberg/McGuire, Lamport (hier nicht weiter diskutiert).

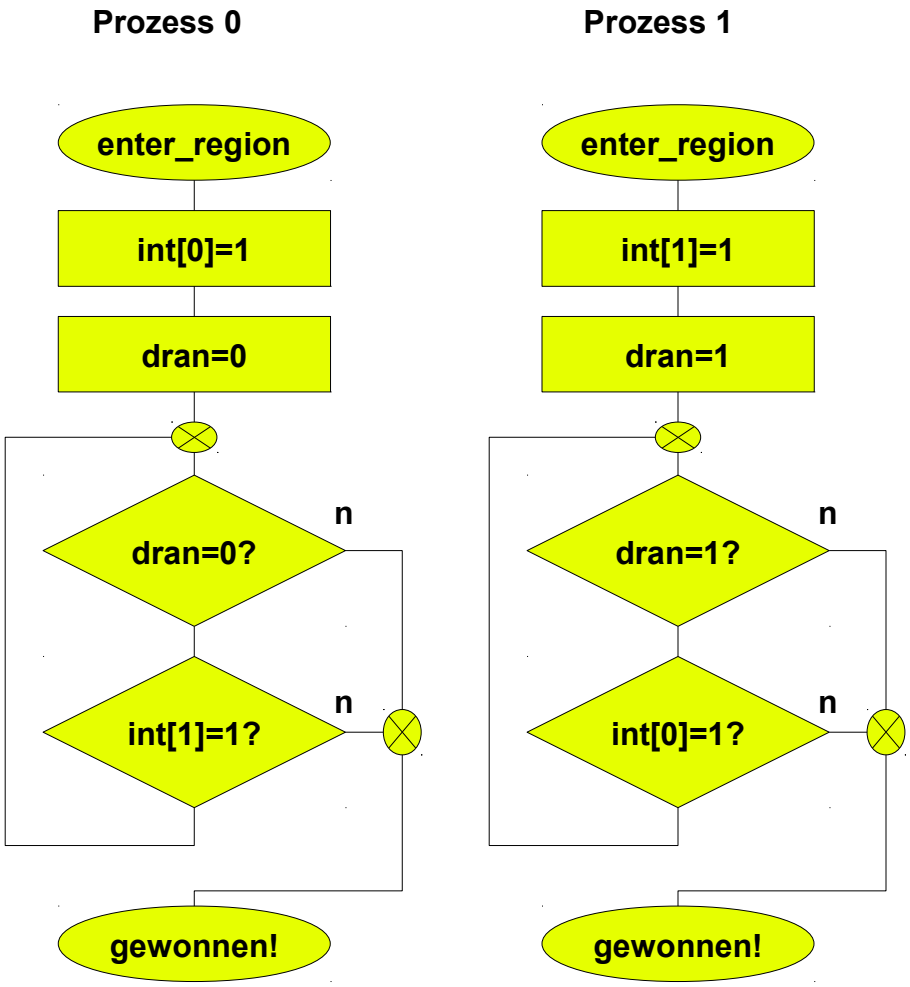
Algorithmus

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
/* Algorithmus von Peterson fuer 2 Prozesse                          */

#define N    2                          /* Anzahl der Prozesse       */

/* gemeinsame Variablen                                              */

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

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

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

Algorithmus

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



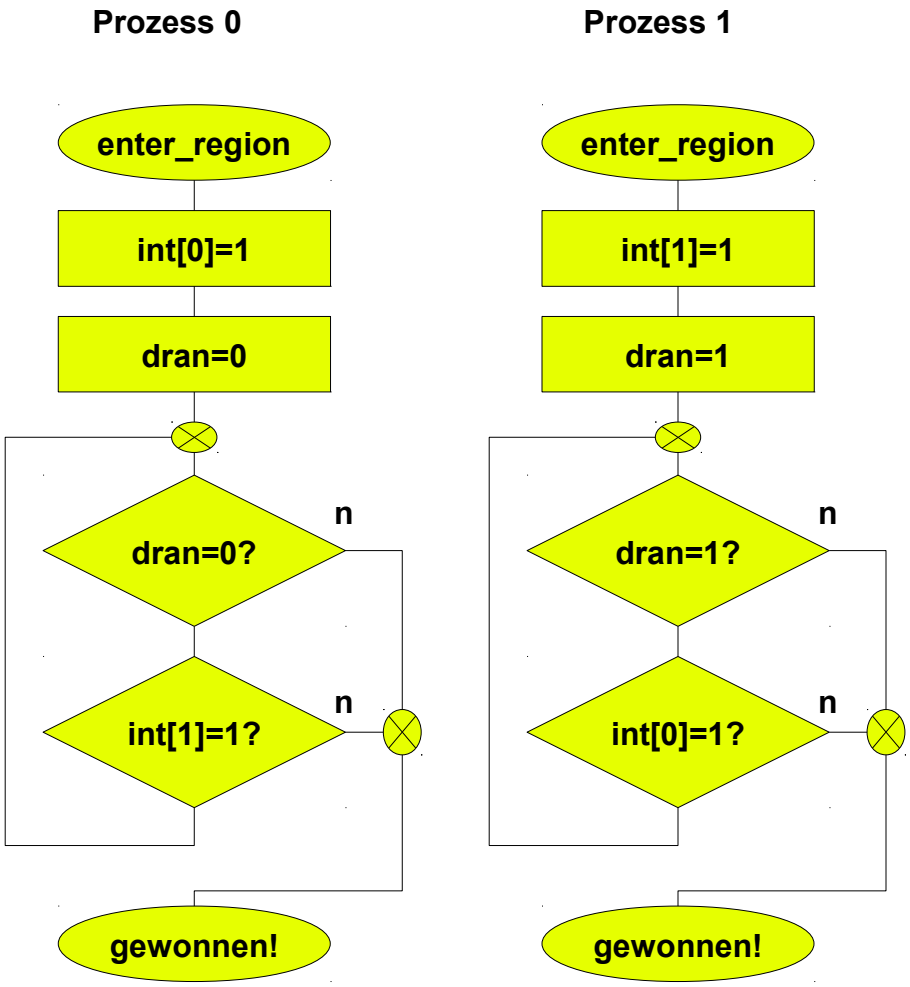
Anfangszustand

interested[]: 0 0

dran: 0

Algorithmus

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



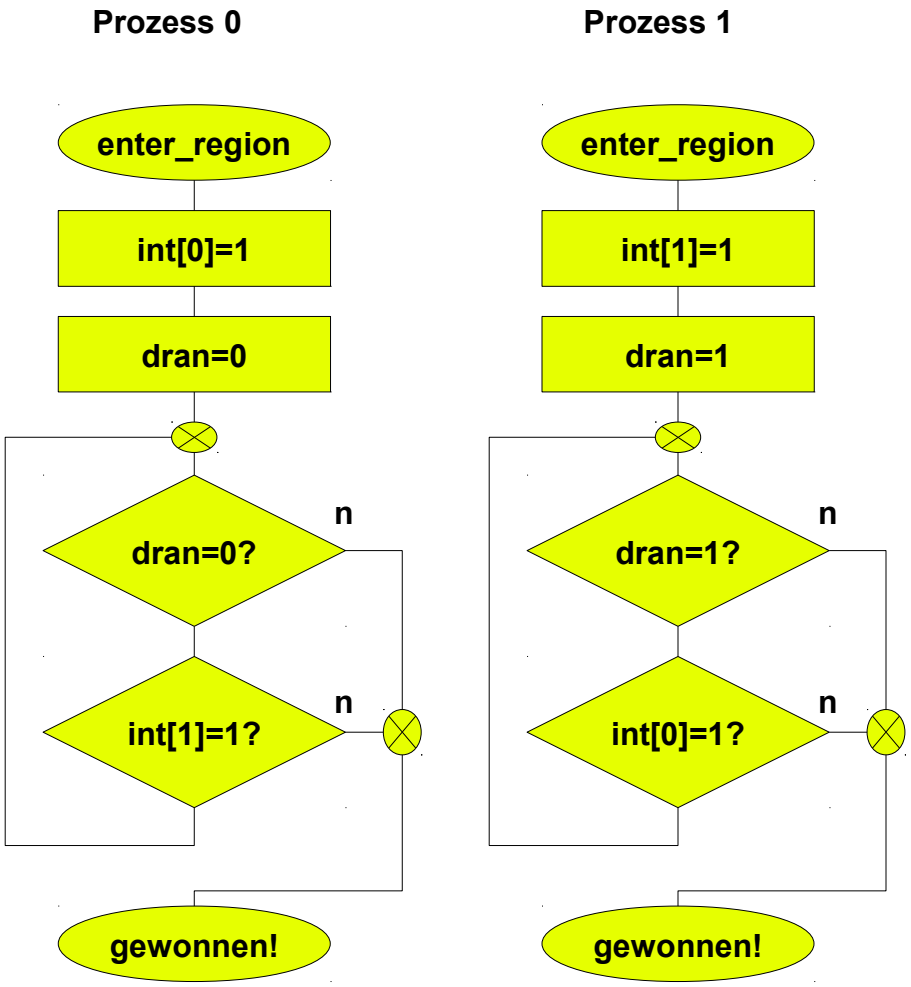
Prozess 0 betritt krit. Sektion Prozess 1 nicht

interested[]: 1 0

dran: 0

Algorithmus

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



Prozess 0 ist in krit. Sektion, Prozess 1 versucht einzutreten

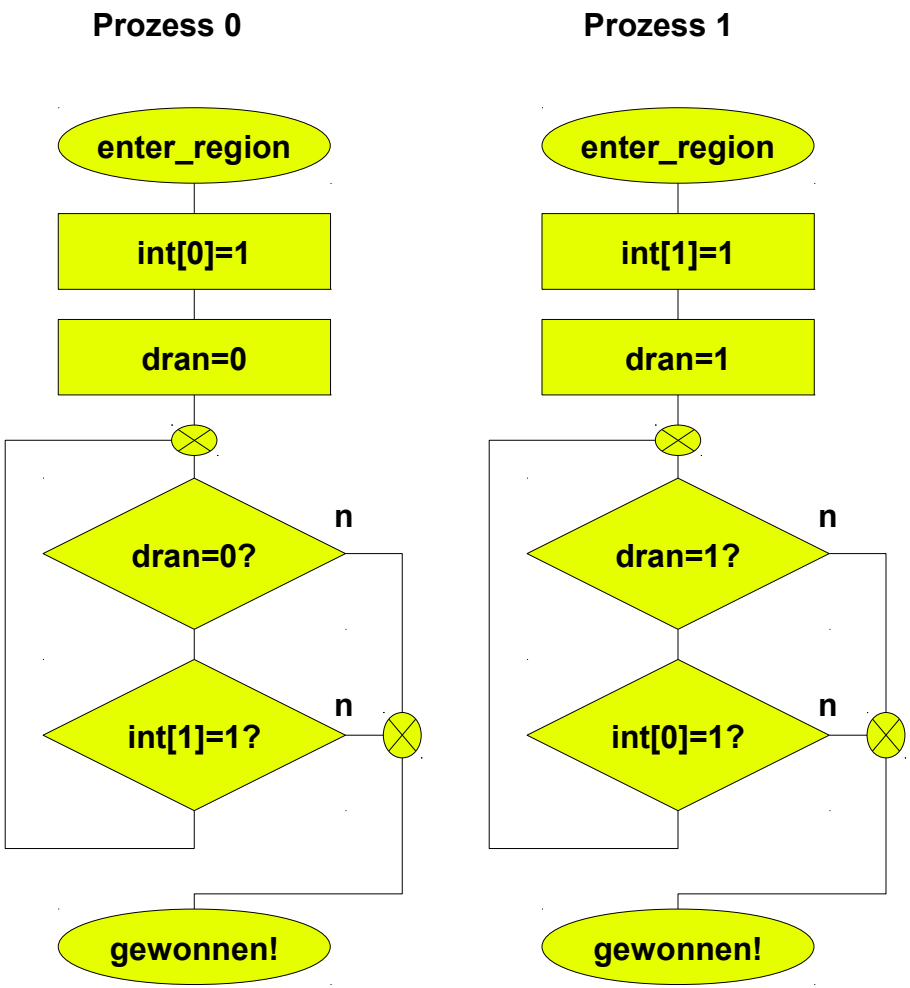
interested[]: 1 1

dran: 1

=> Prozess 1 wartet bis int[0] = 0.

Algorithmus

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



Kritischer Fall:
Beide versuchen gleichzeitig, krit. Sektion zu betreten

interested[]: 1 1

dran: 0 oder 1

Schreibzugriff auf `dran` ist atomar => nur einer von beiden kann `dran` erfolgreich setzen! Wer Erfolg hat, muss hier warten!

5.2.1.5 Test-and-Set - Instruktion

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Algorithmen zu komplex, die nur auf atomaren Speicherzugriffen basieren (siehe oben).

⇒ Notwendigkeit für effiziente Lösung

- Lösung durch Hardware-Unterstützung:
  Einführung eines Maschinenbefehls Test-and-Set:
  unteilbares Lesen einer Speicherzelle mit anschließendem Schreiben eines Wertes in die Zelle
  (Speicherbus wird dazwischen nicht freigegeben).
- Heute
  - Standard auf praktisch allen Architekturen (mit geringen Abweichungen)
  - notwendig für Multiprozessor-Systeme
  - häufig zusätzliche unteilbare Maschinen-Operationen zur Listenmanipulation (z.B. für Ready-Queue, MC680x0)
  - oder LOCK/UNLOCK-Paare, um kurze Folgen beliebiger Instruktionen atomar auszuführen (z.B. i860).

Lösung mit Test-and-Set

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- 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. Hohe Bedeutung in Multiprozessor-Betriebssystemen.
- Typische Assembler-Routinen für Prolog und Epilog mit Test-and-Set - Instruktion `TSL`:

```
enter_region:
   tsl register,flag    | kopiere flag in register und setze flag auf 1
   cmp register,#0      | war flag vorher Null?
   jnz enter_region     | nein, d.h. Sperre war gesetzt, aktiv warten!
   ret                  | ja, darf Eintreten, zurueck zum Aufrufer

leave_region:
   mov flag,#0          | loesche flag, d.h. setze Sperre zurueck
   ret                  | zurueck zum Aufrufer
```

- Im Gegensatz zur Darstellung hier erfolgt die Realisierung i.d.R. über Makros, da Prozeduraufruf zur Laufzeit zu großen Overhead darstellt.

Problem der Prioritätsinversion

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Dieses Problem kann trotz korrekter Lösungen mit aktivem Warten (Peterson, Test-and-Set, s.o.) auftreten.
- Prozess H habe hohe Priorität, Prozess L habe niedere Priorität. Scheduling-Regel: wenn H rechenwillig ist, soll H ausgeführt werden.
- Es werde H rechenwillig, während L sich in seinem kritischen Abschnitt befindet. H wolle ebenfalls in seinen kritischen Abschnitt eintreten.
- Wird nun der höher priore Prozess H ausgeführt, so wartet er aktiv für immer, da L seinen kritischen Abschnitt nicht beenden kann.
- Es muss also der nieder priore Prozess ausgeführt werden ! Diese Situation heißt auch Prioritätsinversionsproblem.
- (Darüberhinaus tritt das Problem in Lösungen mit passivem Warten ebenfalls auf. Insbesondere in Echtzeitbetriebssystemen ist eine Lösung hierfür extrem wichtig.)

Bisher:

- Prolog-Operationen zum Betreten des kritischen Abschnitts führen zum Aktiven Warten, bis der betreffende Prozess in den krit. Abschnitt eintreten kann.
- 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 seinem krit. Abschnitt austretenden Prozess entblockiert.

Einfachste Primitive

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Einfachste Primitive werden als SLEEP 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.

Beispiel: Windows NT Critical Sections

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- NT Critical Sections erlauben den wechselseitigen Ausschluss von Threads *eines* Prozesses (nicht über einen Adressraum hinaus) mittels passivem Warten.
- `CRITICAL_SECTION cs;`
  `void InitializeCriticalSection(LPCRITICAL_SECTION cs)`
  `void DeleteCriticalSection(LPCRITICAL_SECTION cs)`
  Definition, Initialisierung und Zerstörung einer Critical Section-Variable.
- `void EnterCriticalSection(LPCRITICAL_SECTION cs)`
  Prolog zum Betreten eines kritischen Abschnitts
- `void LeaveCriticalSection(LPCRITICAL_SECTION cs)`
  Epilog zum Verlassen eines kritischen Abschnitts
- `BOOL TryEnterCriticalSection(LPCRITICAL_SECTION cs)`

  Versuch des Betretens eines kritischen Abschnitts ohne Blockierung.

Mutex-Locks

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Der Begriff Mutex ist von mutual exclusion abgeleitet.
- Ein Mutex offeriert Operationen
  - *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).
- Gelegentlich wird angenommen, dass unlock *alle* wartenden Prozesse entblockiert und sich diese dann erneut um das Betreten des kritischen Abschnitts bewerben.

Beispiel: Pthreads - API: Mutexe

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
#include <pthread.h>

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
```

Anlegen einer Mutex-Variablen (mehrere Varianten)

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

Initialisieren einer Mutex-Variablen

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

Lock anfordern

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

Lock anfordern, falls ohne Blockieren möglich

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

Lock freigeben

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

Lock zerstören

Beispiel: Windows NT Mutexe

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Windows NT offeriert Mutexe als Systemobjekte für den wechselseitigen Ausschluss zwischen Threads bel. Prozesse.

Mutex-Operationen:

- ```
  HANDLE CreateMutex (
     LPSECURITY_ATTRIBUTES security,
     BOOL FInitialOwner,
     LPSTR name);
  ```
  Erzeugen neues Mutex.

- ```
  HANDLE OpenMutex (
     DWORD access,
     BOOL inherit,
     LPCTSTR name);
  ```
  Öffnen exist. Mutex.

- ```
  DWORD WaitForSingleObject (
     HANDLE hmutex,

     DWORD Timeout);
  ```
  lock
  wird allg. zur Synch. verwendet

- ```
  BOOL ReleaseMutex (
     HANDLE hsem);
  ```
  unlock

- **Bemerkung: NT Mutexe werden von Threads in "Besitz" genommen: Nur der Thread, der `WaitForSingleObject` erfolgreich ausführte, kann anschließend `ReleaseMutex` ausführen. Nach erfolgreichem lock sind weitere Wait-Aufrufe desselben Threads ebenfalls erfolgreich (Idempotenz).**

Semaphoren

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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
- und einer **Warteschlange** für (passiv) wartende Prozesse

5.2.3 Semaphor-Operationen

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Atomare Operationen!

Initialisierung

- Zähler auf initialen Wert setzen
- "Anzahl der freien Einkaufswagen"

Operation `P()`: Passier(-Wunsch)

- Zähler = 0: 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

- Zähler wird erhöht
- Falls es Prozesse in der Warteschlage gibt, wird einer deblockiert (und erniedrigt den Zähler dann wieder, s.o.)
- `V` steht für „*verhogen*“ (niederländisch für "erhöhen")

Semaphor-Operationen (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

P(s) oder DOWN(s):

i > 0 ⇒ i := i - 1; Prozess fährt fort.

i = 0 ⇒ ausführender Prozess blockiert.
(Er legt sich am Semaphor schlafen).

- **Wichtig: Atomarität bedeutet, dass Überprüfen des Wertes, Verändern des Wertes und sich Schlafen legen als eine einzige, unteilbare Operation ausgeführt werden.**
  Dieses vermeidet zeitkritische Abläufe.

V(s) oder UP(s):

i := i + 1;

Wenn es blockierte Prozesse gibt, wird einer von ihnen ausgewählt.
Er kann seine P-Operation jetzt erfolgreich beenden.
Das Semaphor hat in diesem Fall dann weiterhin den Wert Null, aber es gibt einen Prozess weniger, der an dem Semaphor schläft.

Erhöhen des Semaphor-Wertes und ev. Wecken eines Prozesses werden ebenfalls als unteilbare Operation ausgeführt.

Kein Prozess wird bei der Ausführung einer V-Operation blockiert.

Semaphore (3)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Semaphore, die nur die Werte 0 und 1 annehmen, heißen binäre Semaphore, ansonsten heißen sie Zählsemaphore.
- Binäre Semaphore zur Realisierung des wechselseitigen Ausschlusses.
- Ein mit n>1 initialisiertes Zählsemaphor kann zur Kontrolle der Benutzung eines in n Exemplaren vorhandenen Typs von sequentiell wiederverwendbaren Betriebsmitteln verwendet werden.

Bemerkungen:

- Semaphore sind weit verbreitet
  - innerhalb von Betriebssystemen
  - zur Programmierung nebenläufiger Anwendungen
- Programmierung mit Semaphoren ist oft fehleranfällig, wenn mehrere Semaphore benutzt werden müssen (vgl. 5.3).
- Mutex-locks (vgl. 5.2.2) können als binäre Semaphore angesehen werden.

Anwendung: Wechselseitiger Ausschluss

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

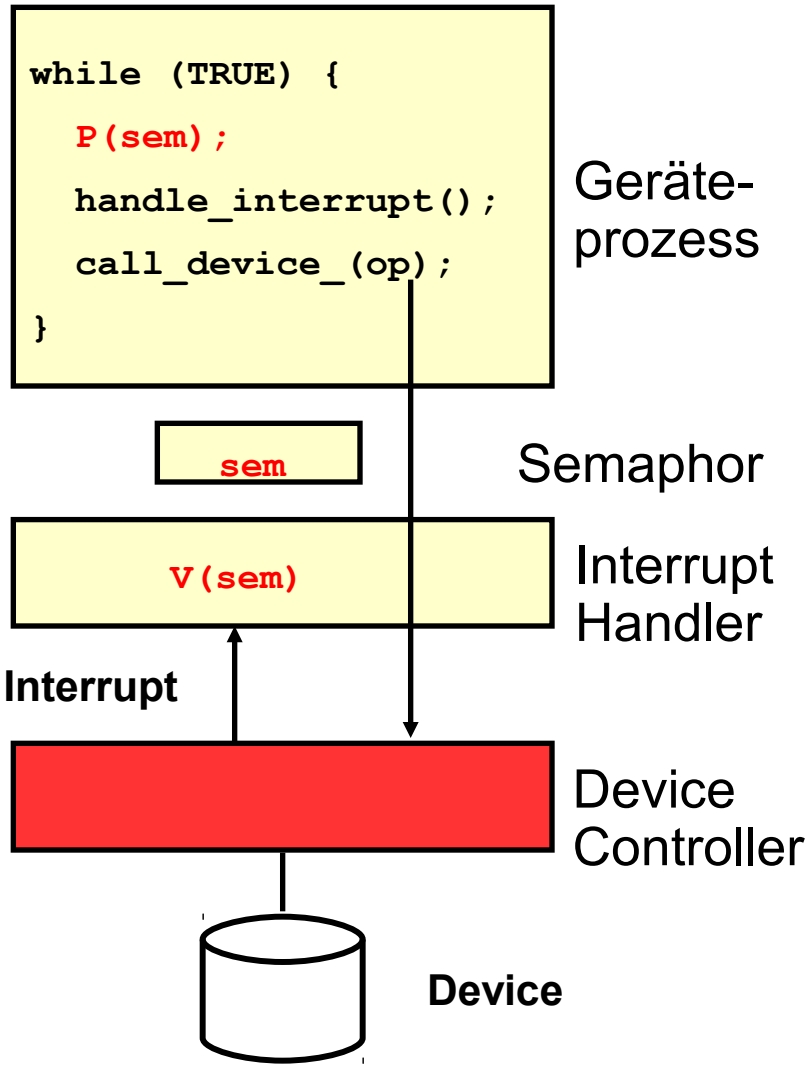
```
semaphore sem = 1;   /* Init. mit 1 */
...
P(sem);              /*   Prolog    */
     <statement> ;
     ...
     <statement> ;
V(sem);              /*   Epilog    */
...
```

geklammerter kritischer Abschnitt

- Das 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: Verstecken von Interrupts

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



- **Jedem I/O-Gerät wird ein Geräteprozess und ein mit 0 initialisiertes Semaphor zugeordnet.**
- **Nach dem Start des Geräte-prozesses 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: Betriebsmittelvergabe

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

```
semaphore bm = n;

P(bm);          /*   Beantragen     */

     Benutze das Betriebsmittel

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

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

Anwendung: Durchsetzen einer Vorrangrelation

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Sei P eine Menge von kooperierenden Prozessen (Prozesssystem), und sei < eine partielle Ordnung auf der Menge der Prozesse mit

  P1 < P2 :<=> Prozess P1 muss vor P2 ausgeführt werden;

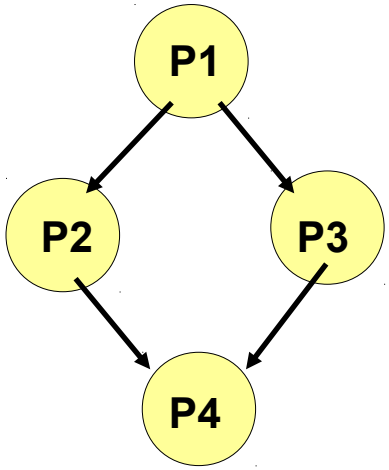
  < wird Vorrangrelation genannt.
  In einem Graph wird P1 < P2 häufig durch P1 → P2 dargestellt.
- Jeder Vorrangbeziehung P1 < P2 wird ein Semaphor s zugeordnet, auf das P1 `V(s)` und P2 `P(s)` ausführt.
- Alle Semaphore werden mit 0 initialisiert.

⇒ Alle Prozesse können gleichzeitig gestartet werden. Die Vorrangrelation zwischen ihnen wird dennoch korrekt durchgesetzt.

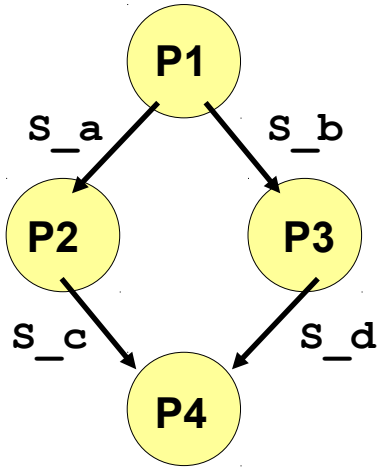
Beispiel: Vorrangrelation

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

gegebenes Prozesssystem:



Lösung:



Initialisieren aller Semaphore mit 0

```
P1() {
...work...
V(S_a);
V(S_b);
exit();
}
```

```
P2() {
P(S_a);
...work...
V(S_c);
exit();
}
```

```
P3() {
P(S_b);
...work...
V(S_d);
exit();
}
```

```
P4() {
P(S_c);
P(S_d);
...work...
exit();
}
```

Implementierung von Semaphoren

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Üblich als Systemaufrufe.
  Intern Nutzung von Sich-Schlafen-Legen und Aufwecken.
- Wesentlich ist die Unteilbarkeit der Implementierung von P() und V():
  - Auf Einprozessorsystemen:
    Unteilbarkeit kann durch Sperren aller Unterbrechungen während der Ausführung von P() und V() erreicht werden. Zulässig, da nur wenige Maschineninstuktionen zur Implementierung nötig sind.
  - Auf Multiprozessorsystemen:
    Jedem Semaphor wird eine mittels Test-and-Set realisierte Sperrvariable mit Aktivem Warten vorgeschaltet. Hierdurch kann zu jedem Zeitpunkt nur höchstens ein Prozessor das Semaphor manipulieren.
  - Beachte: Unterscheide zwischen Aktivem Warten auf den Zugang zum Semaphor, das einen kritischen Abschnitt schützt (einige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt).

Beispiel 1: Semaphore in UNIX

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- System Calls zum Umgang mit Semaphoren (gehören zum System V IPC-Mechanismus)
- UNIX-Semaphore sind in der Benutzung erheblich komplexer als oben beschrieben.
- Es werden Gruppen von Semaphoren betrachtet. Mit einem Operationsaufruf kann eine Teilmenge dieser Semaphoren atomar verändert werden, d.h. alle Operationen werden ausgeführt oder keine und Blockierung.
- Über Flags werden Varianten gesteuert, z.B.
  - nicht blockierender Aufruf mit Fehlercode (Vermeidung einer Blockierung)
  - undo aller Effekte von Semaphor-Operationen bei Beendigung eines Prozesses (Vermeidung von Inkonsistenzen).

Beispiel 1: Semaphore in UNIX (2)

Operationen:

- `int semget(key_t key, int nsems, int semflag):`

  liefert den zum externen `key` gehörenden Id der Semaphor-Gruppe und macht so eine exist. Gruppe zur Benutzung in einem Prozess zugänglich. Wird `key` auf `IPC_PRIVATE` gesetzt oder ein neuer Schlüssel verwendet und in `semflag` das `IPC_CREAT`-Bit gesetzt und existiert noch keine Gruppe zu `key`, so wird eine neue Semaphor-Gruppe mit `nsems` Semaphoren erzeugt.

- `int semop(int semid, struct sembuf *sops[], int nsops)`

  zur Ausführung von Semaphor-Operationen auf der Semaphor-Gruppe `semid`. Die durchzuführenden Operationen werden in einem Vektor `sops` der Länge `nsops` von `sembuf`-Strukturen beschrieben:

```
struct sembuf {
        ushort sem_num;     /* # des Sem. in Gruppe */
        short  sem_op;      /* Op.: <0=P(), >0=V()  */
        short  sem_flg; }   /* IPC_NOWAIT, SEM_UNDO */
```

  Für jedes Semaphor kann damit individuell die durchzuführende Operation festgelegt werden.

- `int semctl(int semid, int semnum, int cmd, union semun arg )`

  Kontrolloperationen auf Semaphoren zum Lesen, Setzen, Löschen, ...

wird im Praktikum vertieft.

Semaphore-Beispiel

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Ziel: 5 parallele Prozesse zählen *nacheinander* sekundenweise von 0...4 (keine Überschneidungen)

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 424242
#define SEMPERM 0777
#define PROCS 5

struct sembuf op_lock   = { 0, -1, 0 };/* Sem 0 erniedr.*/
struct sembuf op_unlock = { 0, 1, 0 }; /* Sem 0 erhöhen */

int main(void) {
  int i, j, semid, status, initwert=1;
  if ((semid = semget(SEMKEY, 1, IPC_CREAT | SEMPERM)) < 0)
{
    perror("semget"); exit(1);
  }
  semctl(semid,0,SETALL,&initwert);
  ...
```

Semaphore-Beispiel (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
  for (i=0; i < PROCS; i++) {
    if (!fork()) {
      if (semop(semid, &op_lock, 1) < 0) {
        perror("semop op_lock"); exit(1);
      }
      for (j=0; j < 5; j++) {
       fprintf(stderr,"Prozess %d mit j=%d\n",getpid(), j);
       sleep(1);
      }
      if (semop(semid, &op_unlock, 1) < 0) {
        perror("semop op_unlock"); exit(1);
      }
      exit(0);
    }
  }
  for (j=0; j < PROCS; j++) wait(&status);
  semctl(semid, 0, IPC_RMID, 0);
  return 0;
}
```

Sohnprozess, 5x

kritisch

Beispiel 2: Semaphoren in Windows NT

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- **Windows NT offeriert Zählsemaphore als Systemobjekte für die Synchronisation von Threads beliebiger Prozesse.**
- **Operationen:**

- `HANDLE CreateSemaphore (` — **Erzeugen eines neuen Semaphors**

```
   LPSECURITY_ATTRIBUTES security,
   LONG initialValue,
   LONG maxValue,
   LPCTSTR name);
```

- `HANDLE OpenSemaphore (` — **Öffnen eines existierenden Semaphors**

```
   DWORD access,
   BOOL inherit,
   LPCTSTR name);
```

- `DWORD WaitForSingleObject (` — **entspricht P(), aber allg. verwendbar. wie auch WaitForMultipleObjects()**

```
   HANDLE hsem,
   DWORD Timeout);
```

- `BOOL ReleaseSemaphore (` — **entspricht V()**

```
   HANDLE hsem,
   LONG incrementValue,
   LPLONG prevCount);
```

Monitore:

- Vorschlag Hoare (1974), Brinch Hansen (1975):
- In Programmiersprache eingebettete Primitive zur einfacheren Entwicklung korrekter nebenläufiger Programme.
- Monitor besteht aus
  - Variablen und Datenstrukturen (die das eigentliche Betriebsmittel repräsentieren)
  - Sichtbaren Operationen zur Manipulation der Daten.
  - Wechselseitige 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)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Programmierung von Synchronisationsvorgängen innerhalb von Monitoren mit internen Bedingungsvariablen (condition variables) mit Operationen WAIT und SIGNAL.
- WAIT(c) blockiert den Aufrufer, ein ev. wartender anderer Aufrufer einer Monitor-Operation kann nun den Monitor betreten.
- SIGNAL(c) 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(c) ohne Wirkung (keine Zähler-Semantik!).
- Monitore haben kaum Eingang in Programmiersprachen gefunden (Gegenbeispiele: Mesa (Xerox), eingeschränkt Java `synchronized`)).

Beispiel

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Gedachte Pascal-Erweiterung

```
monitor example

     integer i;
     condition c;

     procedure producer(x);
     ...
     end;

     procedure consumer(x)
     ...
     end;



end monitor;
```

POSIX Condition Variables

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- 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: `https://computing.llnl.gov/tutorials/pthreads/`

POSIX Condition Variables (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Erläuterungen siehe Man Pages:

- `pthread_cond_wait()` blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.
- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait()` routine to complete.
- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.
- It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

Weitere Ansätze

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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: Campbell, Habermann (1974).**
- **Read/Write-Locks (vgl. Datenbanken).**
- **Barrieren.**
- **fork/join.**

Literatur hierzu:

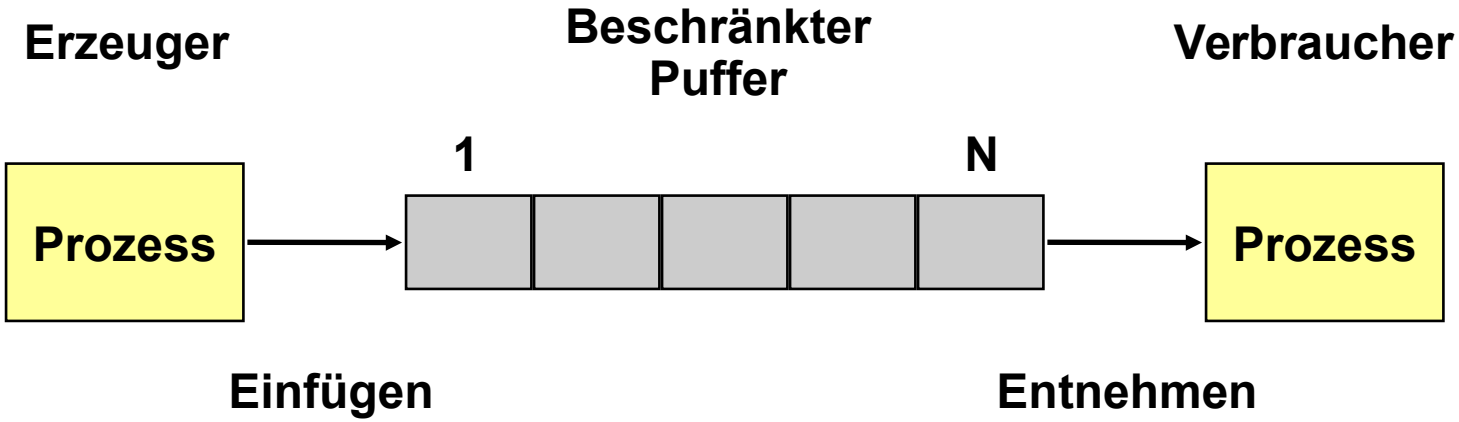
z.B. Tanenbaum, Peterson/Silberschatz, Maekawa et. al. (siehe Literaturliste).

Gliederung

1. Erzeuger-Verbraucher-Problem
2. Philosophen-Problem
3. Leser-Schreiber-Problem

5.3.1. Erzeuger-Verbraucher-Problem

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



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.**

Bemerkungen

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

Im weiteren werden folgende Lösungsansätze betrachtet:

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

5.3.1.1. Lösung mit Semaphoren

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

/* gemeinsame Variablen                                                */

semaphore mutex = 1;                   /* kontroll. kritischen Bereich */
semaphore empty = N;                   /* zaehlt leere Eintraege       */
semaphore full  = 0;                   /* zaehlt belegte Eintraege     */

void producer(void) {                  /* Erzeuger                     */
   int item;
   while(TRUE) {
      produce_item(&item);             /* erzeuge Eintrag              */
      down(&empty);                    /* besorge freien Platz         */
      down(&mutex);                    /* tritt in krit. Abschnitt ein */
      enter_item(item);                /* fuege Eintrag in Puffer ein  */
      up(&mutex);                      /* verlasse krit. Bereich       */
      up(&full);                       /* erhoehe Anz. belegter Eintr. */
   }
}
```

Lösung mit Semaphoren (2)


```
void consumer(void) {                  /* Verbraucher                      */
   int item;
   while(TRUE) {
      down(&full);                     /* belegter Eintrag vorhanden?  */
      down(&mutex);                    /* tritt in krit. Abschnitt ein */
      remove_item(&item);              /* entnimm Eintrag aus Puffer   */
      up(&mutex);                      /* verlasse krit. Bereich       */
      up(&empty);                      /* erhoehe Anz. freier Eintraege*/
      consume_item(item);              /* verarbeite Eintrag           */
   }
}
```

Bemerkungen

- Beachte: `down()` entspricht P(), `up()` entspricht V().
- 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: `down(&empty)`, Verbraucher: `up(&empty)` realisiert: Wenn der Puffer voll ist, muss zuerst ein Eintrag entnommen werden, bevor ein neuer eingefügt werden kann.
- Die Algorithmen für Erzeuger und Verbraucher gelten unverändert, wenn mehrere Erzeuger und/oder mehrere Verbraucher zugelassen werden.

5.3.1.2. Lösung mit Monitoren

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
monitor ProducerConsumer
   condition full, empty;
   integer count;

   procedure enter;
   begin
     if count=N then wait(full);
     enter_item;
     count:=count+1;
     if count=1 then signal(empty);
   end;

   procedure remove;
   begin
     if count=0 then wait(empty);
     remove_item;
     count:=count-1;
     if count=N-1 then signal(full);
   end;

   count:=0;
end monitor;
```

(*hypothetische* Pascal-Erweiterung)

Der Erzeuger muß warten, solange der Puffer voll ist

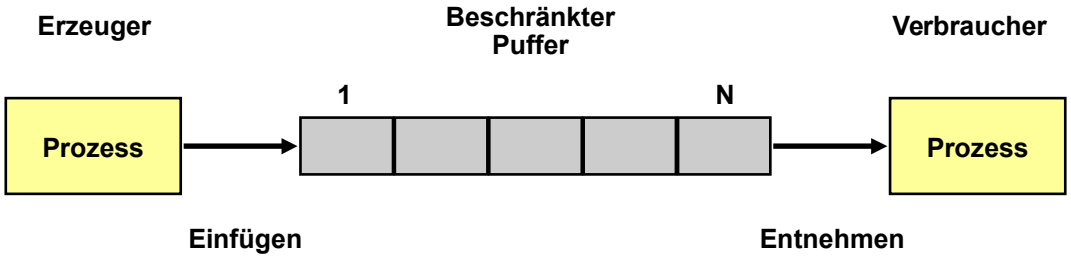
Ein wegen der "empty"-Bedingung wartender Prozess, könnte nun weitermachen

Der Verbraucher muß warten, solange der Puffer leer ist

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

Lösung mit Monitoren (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim




```
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter;
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item;
  end
end;
```

Bemerkungen

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

5.3.1.3. Lösung mit Nachrichtenaustausch

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

typedef int message[MSIZE];

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*/
   }
}
```

Lösung mit Nachrichtenaustausch (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
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             */
   }
}
```

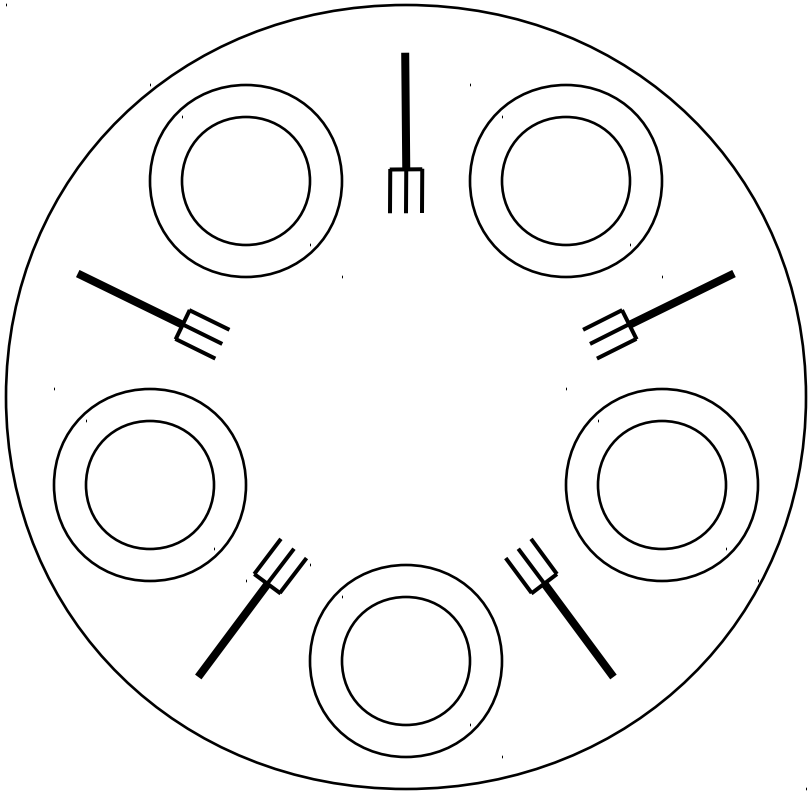
Bemerkungen

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Beachte: Kein gemeinsamer Speicher zwischen Erzeuger und Verbraucher.
- Annahmen: Alle Nachrichten haben dieselbe feste Länge. Gesendete, aber noch nicht empfangene Nachrichten werden innerhalb des Nachrichtensystems automatisch gepuffert.
- Es werden insgesamt N Nachrichtenhülsen (Umschläge) benutzt. Leere Nachrichten werden vom Verbraucher an den Erzeuger gesendet, der Erzeuger füllt leere Nachrichten mit erzeugten Einträgen und sendet "gefüllte Umschläge" an den Verbraucher.
- Die Algorithmen für Erzeuger und Verbraucher sind aufgrund der direkten Adressierung auf einen Erzeuger und einen Verbraucher zugeschnitten.

5.3.2. Philosophen-Problem

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



- Dijkstra (1965): dining philosophers problem.
- 5 Philosophen sitzen am runden Tisch. Jeder 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 jeden Philosophen?

Eine zu einfache, fehlerhafte Lösung

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

void philosopher(int i)         /* i:0..N-1, welcher Philosoph */
{
   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        */
   }
}
```

Annahme: Alle Philosophen greifen die linke Gabel durch `take_fork(i)`.

Dann sind alle für immer blockiert. Es liegt ein sogenannter Deadlock vor.

Verbesserte, immer noch fehlerhafte Lösung

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Verbesserung: Nach dem erfolgreichen Aufnehmen der linken Gabel überprüfe, ob die rechte Gabel verfügbar ist. Falls nicht, lege die linke Gabel ab, warte eine Zeitlang, dann beginne von vorn.
- Dieser Ansatz vermeidet den alten Fehler (Deadlock), enthält aber einen neuen:

  Alle Philosophen greifen gleichzeitig die linke Gabel, erkennen, dass die rechte nicht verfügbar ist, legen die linke wieder ab, warten gleich lang, greifen wiederum gleichzeitig die linke usw.
- Das Problem der endlosen Ausführung ohne Fortschritt wird Verhungern (Starvation) genannt.

Korrekte, aber unbefriedigende Lösung

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

   void philosopher (int i)         /* i:0..N-1, welcher Philosoph */
   {
      while (TRUE) {
         think();                   /* Denken                      */
         down(&mutex);
         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        */
         up(&mutex)
      }
   }
```

kritisch

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

Lösung mit Semaphoren

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

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

/* gemeinsame Variablen                                               */

int state[N];                  /* Zustaende aller Philosophen         */
semaphore mutex = 1;           /* fuer wechselseitigen Aussschluss    */
semaphore s[n];                /* Semaphor fuer jeden Philosoph       */


void philosopher(int i) {      /* i:0..N-1, welcher Philosoph         */
   while (TRUE) {
      think();                 /* Denken                              */
      take_forks(i);           /* Greife beide Gabeln oder blockiere*/
      eat();                   /* Essen                               */
      put_forks(i);            /* Ablegen beider Gabeln               */
   }
}
```

Lösung mit Semaphoren (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

```
void take_forks(int i) {          /* i:0..N-1, welcher Philosoph       */
   down(&mutex);                  /* tritt in krit. Bereich ein        */
   state[i] = HUNGRY;             /* zeige, dass du hungrig bist       */
   test(i);                       /* versuche, beide Gabeln zu bekommen*/
   up(&mutex);                    /* verlasse krit. Bereich            */
   down(&s[i]);                   /* bockiere, falls Gabeln nicht frei */
}

void put_forks(int i) {           /* i:0..N-1, welcher Philosoph       */
   down(&mutex);                  /* tritt in krit. Bereich ein        */
   state[i] = THINKING;           /* zeige, dass du fertig bist        */
   test(LEFT);                    /* kann linker Nachbar jetzt essen ? */
   test(RIGHT);                   /* kann rechter Nachbar jetzt essen ?*/
   up(&mutex);                    /* verlasse krit. Bereich            */
}

void test(int i) {                /* i:0..N-1, welcher Philosoph       */
   if (state[i]== HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
   {
      state[i]=EATING;            /* jetzt kann Phil i essen !         */
      up(&s[i]);                  /* "sage es ihm"                     */
   }
}
```

Bemerkungen

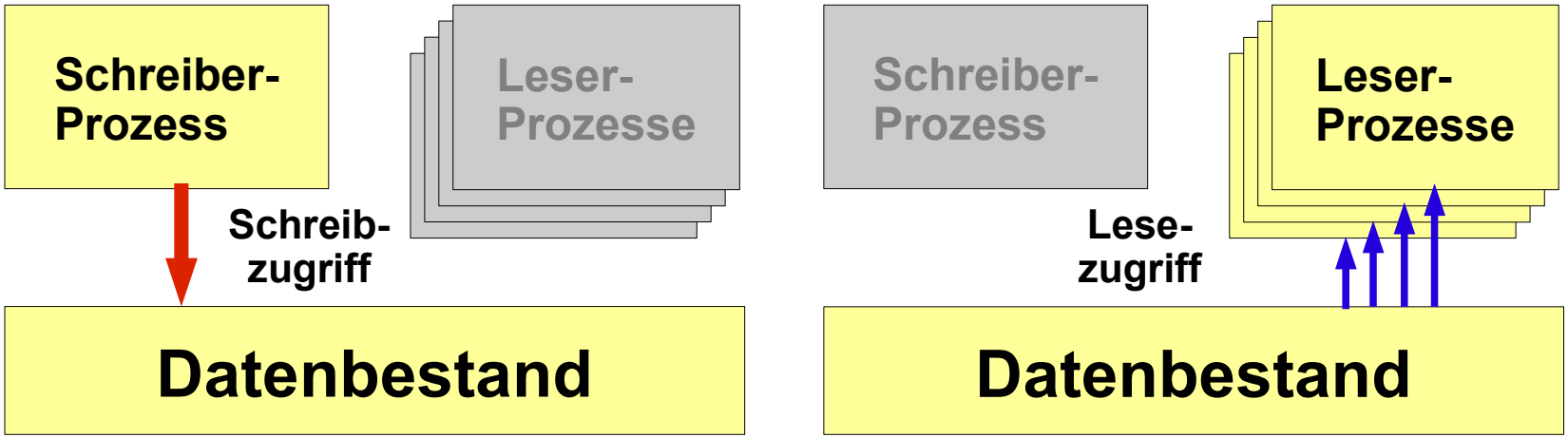
- Jeder Prozess führt die Prozedur `philosopher` als Hauptprogramm aus, die anderen Prozeduren sind gewöhnliche Unterprogramme, keine separaten Prozesse.
- Das array `state` speichert die aktuellen Zustände der Philosophen: essend, denkend, hungernd (versucht, Gabeln zu bekommen).
- Jeder Philosoph blockiert 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.

Bemerkungen (2)

- 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 Philosophen korrekt.

5.3.3. Leser-Schreiber-Problem

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



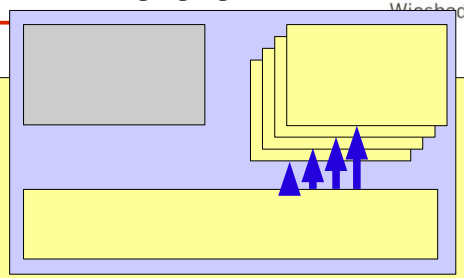
Zu jedem Zeitpunkt dürfen entweder mehrere Leser oder ein Schreiber zugreifen.

Verboten: gleichzeitiges Lesen und Schreiben

Wie sollten Leser- und Schreiber-Programme aussehen?

Lösung mit Semaphoren

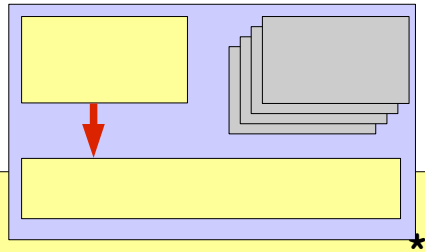
Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



```
/* 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) {
      down(&mutex);            /* erhalten exkl. Zugriff auf rc      */
      rc=rc+1;                 /* ein zusaetzlicher Leser            */
      if (rc==1) down (&db);   /* falls 1.Leser, reserviere Daten    */
      up(&mutex);              /* freigeben exkl. Zugriff auf rc     */

      read_data_base();        /* lies Datenbestand                  */

      down(&mutex);            /* erhalten exkl. Zugriff auf rc      */
      rc=rc-1;                 /* ein Leser weniger                  */
      if (rc==0) up (&db);     /* falls letzter Leser, Daten freigeb*/
      up(&mutex);              /* freigeben exkl. Zugriff auf rc     */
      use_data_read();         /* unkrit. Bereich                    */
   }
}
```

Lösung mit Semaphoren (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



```
void writer(void) {              /* Schreiber                          */
   while (TRUE) {
      create_data();             /* unkrit. Bereich                    */
      down(&db);                 /* erhalten exkl. Zugriff auf Daten   */
      write_data_base();         /* schreib Datenbestand               */
      up(&db);                   /* freigeben exkl. Zugriff auf Daten  */
   }
}
```

Bemerkungen

- Semphore `mutex` sichert den kritischen Abschnitt in der Benutzung des Read-Counters `rc`.
- Semaphore `db` sichert den Zugriff auf den Datenbestand, so dass entweder mehrere Leser oder ein Schreiber zugreifen können. Der erste Leser führt eine P-Operation durch, alle weiteren inkrementieren nur `rc`. Der letzte Leser führt eine V-Operation auf dem Semaphore `db` aus, so dass ein wartender Schreiber Zugriff erhält.
- Lösung bevorzugt Leser gegenüber Schreibern. Neu eintreffende Leser erhalten Zugriff vor einem schon wartenden Schreiber, wenn noch mindestens ein Leser Zugriff hat.

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 gleichmächtig. Besprochen wurden insbesondere Semaphoren 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.