Betriebssysteme

Dieter Zöbel

Universität Koblenz-Landau
Fachbereich Informatik
Institut für Informatik

Inhaltsverzeichnis

Kapitel 2

Parallelität und Synchronisierung

- Präzisierung des Prozessbiegriffs
- Notwendigkeit der Synchronisierung paralleler Prozesse
- Verklemmungen zwischen parallelen Prozessen
- Elementare Synchronisierungsmethoden
- Hochsprachliche Synchronisierungsmethoden
- Wiederkehrende Problem- und Lösungschemata bei der parallelen Programmierung

2.1 Parallele Prozesse

Programmiertechnisches Abstraktionsobjekt: **der Prozess**

- eine sequentielle Folge von Anweisungen auf einem eigenen Zustandsraum
- weitgehende Unabhängigkeit von anderen Prozessen

Parallel - pseudo oder echt?

Der Begriff *parallel* wir oftmals sehr unbedarft verwendet. Dennoch ist zu unterscheiden, ob *pseudo-parallel* oder *echt parallel* gemeint ist.

- Werden die Anweisungen gleichzeigig auf getrennten Rechensystemen ausgeführt, so spricht man von echter Parallelität (engl. *parallelism*).
- Ist hingegen nur ein Rechensystem vorhanden, dann kann die Ausführung von Anweisungen nur *zeitlich verzahnt* (vgl. [26]) stattfinden (engl. *concurrent, interleaved*)

Progammiertechnische Sicht auf die Prozesse

Unterscheidung:

- Prozesstyp
  Spezifiziert mittels programmiertechnische Kapselung eine Aufgabe, die in Form eines Prozesses ausgeführt werden soll
  z.B. Berechung einer Bildschirmzeile nach einem Verfahren der Computergraphik

- Prozessobjekt oder Prozessinstanz
  Ein Prozess eines speziellen Typs, dem ein eigener Zustandsraum (d.h. eigene Variablen) zugeordnet ist
  z.B. Berechnung der Bildschirmzeile 10 und aller darin zu besetzenden Pixel

- Prozessausführung
  eine Prozessinstanz in Ausführung auf dem aktuellen Zustandsraum und mit einem aktuellen Ausführungszustand
  z.B. die 5601-te Ausführung der Berechnung der 10-ten Bildschirmzeile ist gerade im Gang

Vergleich von Prozessen mit Threads

Prozess (oft auch Task genannt)

- kann einen oder mehrere Threads enthalten
- besitzt eigenen Adressraum
- besitzt Betriebsmittel z.B. offene Dateien
- Kommunikation mit anderen Prozessen über Pipes oder Nachrichten

Thread

- ist einem Prozess zugeordnet
- gemeinsamer Adressraum (sog. *Speichermodell* [34])
- teilt Betriebsmittel mit den anderen Threads des Prozesses
- Kommunikation mit anderen Threads über den gemeinsam Speicher

Programmiermodelle und -systeme (1)

Modelle und Systeme, denen ein Speichermodell zugrunde liegt:

- Posix
  die `pthread`*-Bibliothek*
- JVM
  java virtual machine mit der Klasse `Thread`
- OpenMP
  open multiprogramming[1]
- UPC
  unified parallel C unterstützt das Speichermodell als auch Systeme mit verteilten Speichern[2]
- CUDA
  Compute Unified Device Architecture konsequente Ausnutzung von Vektorparallelität bei GPUs

[1] unter der URL `www.openmp.org` findet sich folgende Zielsetzung: *... OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer*, aktuell Version3.0, 2008

[2] `upc.gwu.edu`

CUDA (1)

Ursprünglich: einfache C-Erweiterung für die Beschreibung massiv paralleler Berechungungen auf der Basis weniger grundlegender Konzepte:

- Hierarchie von Thread-Strukturen (*grid, block*)
- Kooperation über gemeinsamen Speicher
- Verwendung des *barrier*-Konzeptes zur Synchronisierung

CUDA (2)

Beispiel: sequentielle Ausführung von Operationen auf einem Array (nach [28]):

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
for(int i = 0; i<n; ++i)
y[i] = alpha*x[i] + y[i];
}


// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

CUDA (3)

Beispiel: CUDA-parallele Ausführung von Operationen auf einem Array (nach [28]):

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
if( i<n ) y[i] = alpha*x[i] + y[i];
}


// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Programmiermodelle und -systeme (2)

Modelle und Systeme, die auf der Grundlage von Nachrichten kommunizieren und sich synchronisieren:

- DCE
  distributed computing environment
- PVM
  parallel virtual machine
- MPI
  message passing interface
- Corba
  common object request broker
- Linda
  tuple-space Modell von Carriero und Gelernter [7]
- Go-Language
  basierend auf synchroner Nachrichtenübetragung (jedoch angewandt auf Mehrprozessoersysteme)

Parallele Rechensysteme (1)

Wo gibt es Parallelausführung[3]?

- DMA-Controller *direct memory access*
  direkter Datentransfer zwischen dem E/A-Geräten und dem Hauptspeicher während der Prozessor auf lokalen Daten weiterarbeitet
- Piplining des Prozessors
  Die Stufen
  - Laden des Befehls
  - Dekodieren des Befehls
  - Laden der Operanden
  - Ausführen des Befehls
  - Speichern des Ergebnisses

  gleichzeitig für verschiedene Befehle ausführen kann.
- Parallelität auf Bitebene
  Die Bitbreite 16, 32 oder 64 ermöglicht unterschiedliche Grade der Parallelausführung von Operationen
- Multicore-Prozessoren
  mehrere Prozessoren mit eigenen Speichern auf einem Chip
- Pseudo-Parallelität
  Ausnutzung der Wartezustände anderer Prozesse

[3] Charakterisierung in Anlehnung an [34]

Parallele Rechensysteme (2)

Klassifizierung von Rechensystemen nach Flynn [14]:

- SISD
  single instruction, single data
  z.B. der klassische *von Neumann*-Rechner
- MISD
  multiple instruction, single data
  bestenfalls anwendbar als Programmiermodell auf vergleichende Verfahren für dasselbe Datum
- SIMD
  single instruction, multiple data
  z.B. Vektorrechner, die auf indizierten Daten dieselben Operationen ausführen
- MIMD
  multiple instruction, multiple data
  gängiges Rechensystem, bei dem unterschiedliche Prozessoren auf unterschiedliche Daten zugreifen können

Parallele Rechensysteme (3)

Klassifizierung von Mehrprozessorarchitekturen:

- bzgl. Homogenität
  Unterscheidung zwischen homogenen und heterogenen Mehrprozessorarchitekturen
  Spezialisierung für homogenen Fall: Symmetrische Multiprozessoren (SMP[4]) adressieren mit derselben physikalischen Adresse dieselbe Speicherstelle
- Art des Zugriffs auf den gemeinsamen Speicher
  - UMA[5]-Prinzip, bei dem alle Prozessoren über einen einzigen Bus auf den Hauptspeicher zugreifen
  - NUMA[6]-Prinzip, bei dem jeder Prozessor über seinen eigenen Speicher verfügt

[4] Während SMP-Prozessoren voll ausgebaut sind, haben SMT-Prozessoren (*symmetric multithreading*) keine eigene ALU oder FPU
[5] uniform memory access
[6] non uniform memory access

Parallele Rechensysteme (4a)

Stufen der Paralleltität bei Prozessoren im Übergang von SISD nach MIMD:

- Parallelität auf der Ebene von Instruktionen: Pipeline-Prozessoren erlauben die gleichzeitige Bearbeitung von Befehlen in unterschiedlichen Schritten. Die Prozessoren modifizieren ggf. die Reihenfolge der Befehle und schätzen, wohin der nächste Sprungbefehl führen wird.

- Simultanes Multithreading (SMT), auch Hyperthreading (HT) genannt: Von zentraler Bedeutung ist das mehrfache Vorhandensein von Thread Descriptoren, bestehend aus Befehlszeiger, Statuswort und Stackzeiger. Sobald ein Thread beispielsweise wegen eines Cache-Fehlers nicht weiterarbeiten kann, wird der nächste Thread auf demselben Prozessor aktiv. So ist eine Leistungssteigerung von 20%-30% durch Nutzung der *hidden latency* möglich.

Parallele Rechensysteme (4b)

Stufen der Paralleltität bei Prozessoren im Übergang von SISD nach MIMD:

- Mehrkern-Prozessoren (multi-core processors): Mehr als ein Prozessor kommt auf dem Chip zum Einsatz, sodass unabhängige Threads gleichzeitig ausgeführt werden können. Die Beschleunigung kann nahe an die Anzahl der Kerne herankommen, wenn genügend weitgehend unabhängige Threads abgearbeitet werden.

  Das Mehrkern-Konzept passt sehr gut zum Hyperthreading und zu Pipeline-Prozessoren, weil sich die Vorteile ergänzen.

Parallele Rechensysteme (5)

Cache-Kohärenz (nach [34]): Ein Speichersystem ist kohärent, wenn für jede Speicherzelle gilt, dass jede Leseoperation den letzten geschriebenen Wert zurückliefert.

Rückschreibstrategien

- *write through*
  Veränderungen im Cache werden direkt in den Hauptspeicher geschrieben
- *write back*
  erst wenn ein Cache-Block nicht mehr gebraucht wird, werden Veränderungen im Hauptspeicher sichtbar

Kohärenz-Erhaltung bei *write through* durch *snooping*, d.h. eine veränderte Speicherstelle wird durch Beobachten des Busses erkannt und der neue Wert im lokalen Cache eingepflegt

2.2 Parallelität und Synchronisierung

- Programmiersprachliche Formulierung paralleler Prozesse
- Konsistenzeigenschaftien bei parallelen Prozessen
- Aufgabe der Synchronisierung
- Prinzipielle Vorgehensweise bei der Synchronisierung

Die Parallelanweisung

```
PARBEGIN
  P(0);
    :
  P(N-1);
PAREND
```

oder

```
[P(0) || ... || P(N-1)]
```

Beispiel: zwei Prozesse arbeiten an derselben Variablen

```
PARBEGIN
  x++;
  x--;
PAREND
```

Ziel: Am Ende soll `x` unverändert sein

Die atomare Anweisung `A`

- Ideal:
  eine terminierende Anweisung `A`
  - wird zum selben Zeitpunkt begonnen wie beendet und
  - zum selben Zeitpunkt wird keine andere Anweisung durchgeführt
- Real:
  Eine Anweisung `A` heißt atomar, wenn ihre Wirkung vom Ideal nicht zu unterscheiden ist, d.h.:
  - die Operationen auf Variablen sind unteilbar
  - Zwischenzustände sind nicht sichtbar
  - andere atomare Anweisungen finden vor oder nach `A` statt
- Notation: `<A>`

Beispiele für atomare Anweisungen

- Zuweisung einer Variablen an eine andere:
  `y=x;`
- Zuweisung eines Literals an eine Variable:
  `y=c;`

Grundlage: atomare Maschinenbefehle
Beispiel

```
P1::                    P2::
  :                       :
y=x;                    y=c;
  :                       :
```

Mögliche Ergebnisse: `y==x` oder `y==c`

Formen atomarer Anweisungen

- Unmittelbare Atomarität beim lesenden und schreibenden Zugriff auf den Hauptspeicher.
- Die ”at-most-once”-Eigenschaft für Ausdrücke `e` und Zuweisungen `x=e;`:
  Diese Eigenschaft ist erfüllt, wenn eine der folgenden Bedingungen zutrifft [3]:
  1. Auswertung eines Ausdrucks `e`, der höchstens eine Variable `y` enthält, die ein anderer Prozess während der Ausdrucksauswertung verändern kann. Bei dieser Auswertung darf `y` höchstens einmal gelesen werden.
  2. Ausführung der Zuweisung `x=e;`, wobei `x` währenddessen von keinem anderen Prozess gelesen wird und für `e` die Bedingung (1) gilt.
  3. Ausführung der Zuweisung `x=e;`, wobei `e` keine Variable erhält, die von anderen Prozessen verändert wird.
- Die Anweisungsfolgen `A`, die so geklammert sind: `<A>`.

Atomare Anweisungen in GoLang (1)

Welche Werte für `value` sind zu erwarten?

```
var value int // global counter

func operation() {
       for i := 0; i < 1000000; i++ {
              value++
       }
}

func main() {
       go operation()
       go operation()
}
```

Atomare Anweisungen in GoLang (2)

Was wird aus der Operation `value++`?
Assemblercode:

```
lw $t0, counter
addi $t0, $t0, 1
sw $t0, counter
```

Alle Werte zwischen 1000000 und 2000000 sind zu erwarten.

Atomare Anweisungen in GoLang (3)

Wie kann man die Operation `value++` in GoLang schützen?

```
var value int32 // global counter

func operation() {
	for i := 0; i < 1000000; i++ {
		atomic.AddInt32(&value, 1)
	}
}

func main() {
	go operation()
	go operation()
}
```

Erst jetzt ergibt sich der Wert 2000000 für `value`.

Beispiel: Erzeuger-Verbraucher-Problem

Aufgabenstellung: Gegeben seien zwei Prozesse[7] `erzeuger` und `verbraucher` sowie ein gemeinsamer `puffer`. Ein Verbraucher soll dabei die Daten so aus dem Puffer abholen, wie sie vom Erzeuger abgelegt wurden.

```
DEF
  ARRAY:puffer;
PARBEGIN
  erzeuger();
  verbraucher();
PAREND
```

[7] Prozesstypen oder Prozessinstanzen?

Erzeuger und Verbraucher als unabhängige Prozesse

Der Erzeuger-Prozess und der Verbraucher-Prozess werden als unabhängige Teilaufgaben aufgefasst.

```
erzeuger::
   i=0;
   while(TRUE){
     /* erzeuge Datum */
     puffer[i]=x;
     i++;
   }
```

```
verbraucher::
   j=0;
   while(TRUE){
     y=puffer[j];
     j++;
     /* verbrauche Datum */
   }
```

Was kann in der obigen Lösung alles schief gehen, was ist unrealistisch?

- Entnahme von Daten, die nicht erzeugt wurden
- unbeschränkte Datenstruktur `puffer`
- unbeschränkter Zugriff auf gemeinsame Variablen

Aufgabe der Synchronisierung

Beschränkung der Parallelität

- Aus dem Wort *Synchronisierung* heraus:
  Erzwingung von Gleichzeitigkeit
- Aus der Praxis der parallelen Programmierung:
  Erzwingung einer Ordnung der Ausführung[8]

Korrektheit durch Synchonisierung im Zusammenhang mit einer Invarianten I:
Die erzwungene Ordnung bei der Ausführung paralleler Prozesse garantiert für die Gültigkeit der Invarianten I.

[8] Es ist eine Frage der Granularität, auf welche Ausführungseinheiten (z.B. Maschinenbefehle oder Anweisungen in einer Programmiersprache) die erwähnte Ordnung anzuwenden ist.

Kritische Gebiete

Häufige Aufgabe der Synchronisierung: Erzeugung kritischer Gebiete

Kritische Gebiete (*critical section*):
Programmabschnitte unterschiedlicher Prozesse, in denen sich zu einem Zeitpunkt höchstens ein Prozess aufhalten darf.

Gegenseitiger Ausschluss (*mutual exclusion*):
Das Problem der Erzeugung kritischer Gebiete ist dann gelöst, wenn es gelingt, dass sich Prozesse von der Ausführung bestimmter Anweisungen gegenseitig ausschließen.

Kritische Gebiete beim Erzeuger-Verbraucher-Problem

Welche kritischen Gebiete sind zu unterscheiden?

```
erzeuger::
   i=0;
   while(TRUE){

     /* erzeuge Datum */

     puffer[i%N]=x;

     i++;

     b++;
   }
```

```
verbraucher::
   j=0;
   while(TRUE){

     y=puffer[j%N];

     j++;

     b--;

     /* verbrauche Datum */

   }
```

Konsistenzprobleme von parallelen Listenopreationen

Auch Operationen auf Listen funktionieren bei paralleler Ausführung nicht mehr korrekt, hier eine Operation zum Eliminieren eines Listenelementes, dass hinter dem Listenelement liegt, auf das `p` zeigt:

```
void *elimNext(void *p){
  if (p==nil) return nil;
  if (p.next==nil) return nil;
  void *h;
  h=p.next;
  p.next=h.next;
  h.next=nil;
  return h;
}
```

2.3 Systematische Entwicklung paralleler Programme

Eine einfache Synchronisierungsanweisung[9], die als atomare Aktion ausgeführt wird und einen booleschen Ausdruck `B` sowie eine terminierende Anweisungsfolge `A` beinhaltet:

$$\texttt{<await B} \rightarrow \texttt{A>}$$

Bedeutung:
Die Ausführung der Anweisungsfolge `A` wird verzögert, bis die Bedingung `B` erfüllt ist. Durch die Atomarität der gesamten Anweisung ist gewährleistet, dass bei Beginn der Ausführung von `A` die Bedingung `B` erfüllt ist.

[9] Ansatz entnommen bei [3] , der damit ”fine grained”- von ”coarse grained”-Synchronisierungsoperationen unterscheidet.

Gegenseitiger Ausschluss (1)

Gegeben sei folgender Ansatz:

```
#define draussen 0
#define drinnen  1

common int p[N] = 0;

P[i] :: while (TRUE)
          { /* ausserhalb des kritischen Gebietes */
            p[i] = drinnen;
            /* endlicher Aufenthalt im kritischen Gebiet */
            p[i] = draussen;
          }
```

Durch die Anweisungen zum Betreten und Verlassen des kritischen Gebietes soll die Bedingung, die ein kritisches Gebiet kennzeichnet, invariant sein[10]:

$$MUTEX \equiv \sum_{i=0}^{N-1} p_i \leq 1$$

Die bisherige Lösung kann die Invariante $MUTEX$ nicht einhalten.

[10] Aufgrund der Binärwertigkeit von `p[i]` wird mit p_i der entsprechende boolesche Wert identifiziert.

Gegenseitiger Ausschluss (2)

Schema eines Prozesses, der ein kritisches Gebiet betreten will:

```
P:: /* Initialisierung */
    while (TRUE)
     { /* ausserhalb des kritischen Gebietes */
       /* Betreten des kritischen Gebietes */
       /* endlicher Aufenthalt im kritischen Gebiet */
       /* Verlassen des kritischen Gebietes */
     }
```

Damit gibt es zwei ausgezeichnete Operationen:

- das **Betreten** des kritischen Gebietes
- das **Verlassen** des kritischen Gebietes

Gegenseitiger Ausschluss (3)

Ziel: Entwicklung von Anweisungsfolgen, die zunächst für zwei Prozesse P_1 und P_2 das Betreten und Verlassen des kritischen Gebietes realisieren.

Gütekriterien für eine Lösung:

1. Nur ein Prozess darf im kritischen Gebiet sein (Korrektheitseigenschaft).
2. Keine Aktion von außerhalb hat Einfluss auf das Betreten und Verlassen des kritischen Gebietes.
3. Ein einzelner Prozess, der sich um das kritische Gebiet bewirbt, soll auch hineingelangen.
4. Bei mehreren Bewerbern um das kritische Gebiet soll schließlich einer hineingelangen.
5. Die Ablaufgeschwindigkeiten einzelner Prozesse sollen keinen Einfluss auf obige Bedingungen haben.

Gegenseitiger Ausschluss (4): Betreten

Frage: Was muss unmittelbar vor Ausführung von `p[i]=drinnen` durch Prozess `P[i]` gelten, damit $MUTEX$ nach Ausführung von `p[i]=drinnen` weiterhin gültig bleibt?

$$\sum_{0 \leq j \leq N-1,\ i \neq j} p_j \ = \ 0$$

Mit der Kenntnis, dass p_i vorher auch den Wert 0 hat, folgt:

$$\left\{ MUTEX \ \wedge \bigwedge_{0 \leq j \leq N-1} \neg p_j \right\} \ \texttt{p[i]=drinnen;} \ \{MUTEX\}$$

Somit erhält die folgende Anweisung die Eigenschaft $MUTEX$:

$\{MUTEX\}$

`<await p[0]+...+p[N-1]==0` $\rightarrow$ `p[i]=drinnen;>`

$\{MUTEX\}$

Gegenseitiger Ausschluss (5): Verlassen

Frage: Was muss unmittelbar vor Ausführung von `p[i]=draussen` gelten, damit $MUTEX$ weiterhin gültig bleibt?

Da während des kritischen Gebietes unbedingt gilt:

$$MUTEX \wedge \sum_{0 \leq j \leq N-1,\ i \neq j} p_j = 0 \qquad \wedge p_i = 1$$

kann die Anweisung `p[i]=draussen` ohne Bedingung ausgeführt werden. $MUTEX$ bleibt erfüllt, d.h.:

$\{MUTEX\}$

`<p[i]=draussen;>`

$\{MUTEX\}$

Gegenseitiger Ausschluss (6)

Kommentierte Lösung zum gegenseitigen Ausschluss:

```
#define draussen 0
#define drinnen  1

common int p[N] = 0;

P[i] :: /* p[i]==0 && MUTEX */
        while (TRUE)
          {
           /* ausserhalb des kritischen Gebietes */
           <await p[0]+ ... +p[N-1]==0 -> p[i]=1;>
           /* endlicher Aufenthalt im kritischen Gebiet */
           /* p[i]==1 && MUTEX */
           p[i]=0;
           /* p[i]==0 && MUTEX */
          }
```

Gegenseitiger Ausschluss (7)

Einführung einer einzigen Synchronisierungsvariablen `mutex`

Die Variable `mutex` soll die Eintrittskarte ins kritische Gebiet sein:

```
mutex == 1-(p[0]+ ... +p[N-1])
```

Das hat zur Folge, dass jede Veränderung eines `p[i]` auch eine unmittelbare Veränderung von `mutex` mit sich bringen muss, so beim Betreten[11]:

```
<await mutex == 1 → mutex--; p[i]++;>
```

bzw. beim Verlassen:

```
<mutex++; p[i]--;>
```

Schließlich kann der Vektor `p` ganz weggelassen werden, bzw. zum Zwecke der Programmverifikation als gedachte Variable geführt werden.

[11] `mutex` wird als arithmetische Größe betrachtet.

Gegenseitiger Ausschluss (8)

Lösung zum gegenseitigen Ausschluss mit der Variablen `mutex`:

```
#define draussen 0
#define drinnen  1

common int mutex = 0;

P[i] :: while (TRUE)
          {/* ausserhalb des kritischen Gebietes */
           <await mutex==1 -> mutex = 0;>
           /* endlicher Aufenthalt im kritischen Gebiet */
           mutex = 1;
          }
```

Fairness

Der Fortschritt, den ein Prozess innerhalb seiner Berechnung macht, hängt vom Grad der Fairness des Scheduling-Verfahrens ab.
Grundsätzlich wird vorausgesetzt, dass jede atomare Anweisung, die keine `await`-Bedingung besitzt, schließlich[12] auch ausgeführt wird.
Schwieriger wird die Betrachtung der Fairness für die Synchronisierungsoperation

`<await B` $\rightarrow$ `A>`

Denn hier ist nicht nur der Scheduler, sondern auch die Gültigkeit der Bedingung `B` entscheidend dafür, ob der Prozess einen Fortschritt in seiner Berechnung macht oder nicht.

[12] Im Englischen findet man an dieser Stelle den Begriff *eventually*, für den im Deutschen die Begriffe *schließlich* und *irgendwann* Verwendung finden.

Schwache und starke Fairness (1)

Bezogen auf eine Synchronisierungsoperation

<await B → A>

heißt ein Scheduling-Verfahren

- **schwach fair**, wenn schließlich die Anweisung A ausgeführt wird unter der Voraussetzung, dass B ab einem Berechnungsschritt ununterbrochen erfüllt ist,
- **stark fair**, wenn schließlich die Anweisung A ausgeführt wird unter der Voraussetzung, dass B ab einem Berechnungsschritt unbegrenzt oft erfüllt ist.

Es gilt, dass jedes stark faire Scheduling-Verfahren auch schwach fair ist.

Schwache und starke Fairness (2)

Beispiel: Unterschiede zwischen schwacher und starker Fairness anhand von Variationen paralleler Programme zur Erzeugung von Zufallszahlen:

```
x=0; ende=FALSE; [P1 || P2]
```

Mit

```
P1 :: <await x>0 -> ende=TRUE;>
```

und

```
P2 :: while (!ende) <x=x+1;>
```

Hier genügt die schwache Fairness, um eine Termination des Programms und damit die Realisierung eines Zufallszahlengenerators zu erhalten[13].
Sei hingegen

```
P1 :: <await prim(x) -> ende=TRUE;>
```

wobei `prim(x)` genau für Primzahlen `x` den Wert `TRUE` liefert. Jetzt lässt sich mit der schwachen Fairness die Termination des Programms nicht mehr sicherstellen, wohl aber mit der starken Fairness. Denn da es unendlich viele Primzahlen gibt, ist `prim(x)` unbegrenzt oft erfüllt, sodass schliesslich die Anweisung `ende=TRUE;` ausgeführt wird.

[13] Auf der Grudlage der Definition von Fairness sind keinerlei Schlüsse über die Wahrscheinlichkeitsverteilung der sich ergebenden Werte für `x` herzuleiten.

Schwache und starke Fairness (3)

Fairnessbetrachtung beim gegenseitigen Ausschluss mit Hilfe der Variablen `mutex`.

```
#define draussen 0
#define drinnen  1

common int mutex = 1;

P[i] :: while (TRUE)
         {/* ausserhalb des kritischen Gebietes */
          <await mutex==1 -> mutex = 0;>
          /* endlicher Aufenthalt im kritischen Gebiet */
          <mutex = 1;>
         }
```

Für einen Prozess, der das kritische Gebiet betreten will, ist die Bedingung `mutex==1` immer wieder kurzfristig erfüllt, und zwar sobald ein Prozess das kritische Gebiet verlässt. Das kann unbegrenzt oft der Fall sein, sodass **nur** ein stark faires Scheduling-Verfahren sicherstellt, dass schließlich jeder Prozess, der ins kritische Gebiet will, auch hineingelangt.

2.4 Deadlocks in der parallelen Programmierung

Das Betriebssystem verwaltet Betriebsmittel, die zu einem Zeitpunkt

- frei sind,
- einem Prozess alleine (exklusiv) gehören oder
- einer Menge von Prozessen gemeinsam (engl. *shared*) zugeordnet sind.

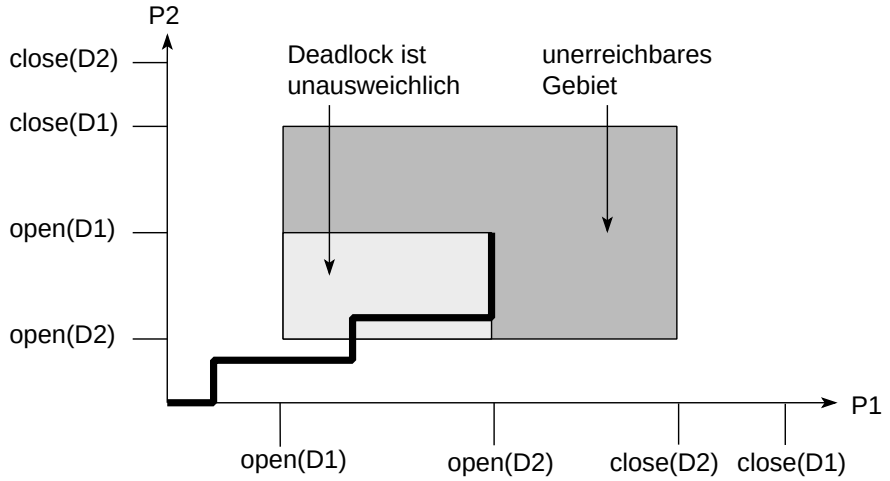
Im Zuge des Mehrprogrammbetriebes kommen Deadlocks (zu Deutsch Verklemmungen) dadurch zustande, dass Prozesse unkoordiniert auf Betriebsmittel zugreifen bzw. zugreifen wollen. Typische Betriebsmittel sind Speicher, Dateien, Geräte, Kommunikationswege, Datenpuffer.

Im weiteren Sinne wirken auch Synchonisierungsoperationen, die Prozesse verzögern, wie Betriebsmittel, die gerade exklusiv von anderen Prozessen benutzt werden.

Deadlocks beim Zugriff auf Dateien

Deadlock zwischen Prozess P_1 und P_2 Fallbeispiel einer zeitlichen Abarbeitung beim exklusiven Zugriff auf die Dateien D1 der Prozesse P1 und P2 und D2:

```
P1::   :            P2::    :
     open(D1);             open(D2);
       :                     :
     open(D2);             open(D1);
       .                     .
       .                     .
       .                     .
     close(D2);            close(D1);
       :                     :
     close(D1),            close(D2);
       :                     :
```



Bedingungen für einen Deadlock

Vier notwendige und hinreichende Bedingungen zum Zustandekommen eines Deadlocks (in Anlehnung an [8]):

(1) Die beteiligten Prozesse wollen exklusiven Zugriff auf Betriebsmittel (*mutual exclusion*).

(2) Die Betriebsmittel, die von den Prozessen bereits belegt werden, können nicht kurzfristig entzogen werden (*no preemption*).

(3) Prozesse können auf Betriebsmittel zugreifen, während sie auf den Zugriff auf andere Betriebsmitteln warten (*wait-for-condition*).

(4) Es findet sich eine geschlossene Kette von Prozessen, die Betriebsmittel besitzen und gleichzeitig auf Betriebsmittel warten, die Vorgänger in der Kette besitzen (*circular wait*).

Einschub: Banker's Algorithmus (1)

In dem Modell von Dijkstra [12] stehen allen Akteuren eine feste Anzahl Res identischer Betriebsmitteln zur Verfügung.
Im Folgenden sind die Akteure Prozesse, genauer die Prozessmenge $P = \{p_1, ..., p_n\}$.
Jeder Prozess kann für eine begrenzte Zeit exklusiven Zugriff auf die Betriebsmittel erlangen. Jeder Prozess p_i darf jedoch nur eine Anzahl max_i besitzen. Diese Zahl ist im Vorhinein anzugeben
Typischerweise ist die Zahl aller Betriebsmittel, die alle Prozesse insgesamt besitzen möchten, höher als die Zahl der zur Verfügung stehenden:

$$Res < \sum_{i \in \{1,...,n\}} max_i$$

Deshalb ist es notwendig, die Vergabe der Betriebsmittel so zu steuern, dass alle Prozesse irgendwann Zugriff auf ihre Betriebsmittel erlangen.

Einschub: Banker's Algorithmus (2)

Um den Zugriff der Prozesse auf Betriebsmittel zu regulieren, muss Prozess p_i mit req_i die Zahl k explizit ordern und mit $free_i$ wieder frei geben.
Anfänglich besitzt Prozess p_i $alloc_i = 0$ Betriebmittel. Werden dem Prozess req_i Betriebsmittel zugeordnet, dann erhöht sich $alloc_i$ um diese Zahl. Insgesamt muss gelten:

$$0 \leq alloc_i \leq max_i$$

Für eine Betriebsmittelfordernung eines Prozesses p_i muss darüber hinaus gelten:

$$req_i \leq max_i - alloc_i$$

eine Betriebsmittelfordernung eines Prozesses p_i vorübergehend nicht zugeordnet, so muss Prozess p_i warten.

Einschub: Banker's Algorithmus (3)

Jede Anforderung req_i eines Prozesses ist zu prüfen. Eine notwendige Bedingung für die Zuteilung ist, dass noch genügend Betriebsmittel zur Verfügung stehen.
Sei $0 \leq res \leq Res$ die Anzahl der Betriebsmittel, die noch nicht an Prozesse vergeben wurden.

$$res = Res - \sum_{i \in \{1,\ldots,n\}} alloc_i$$

Dann muss notwendigerweise für eine Anforderung gelten:

$$req_i \leq res$$

Wird dann ein Betriebsmittel zugeteilt, so sind folgende Werte zu ändern:

$$\begin{aligned} alloc_i &= alloc_i + req_i \\ res &= res - req_i \end{aligned}$$

Einschub: Banker's Algorithmus (4)

Deadlocks sind jedoch möglich. Dann gibt es Prozessen, die unbegrenzt lange auf Betriebsmittelanforderungen warten.

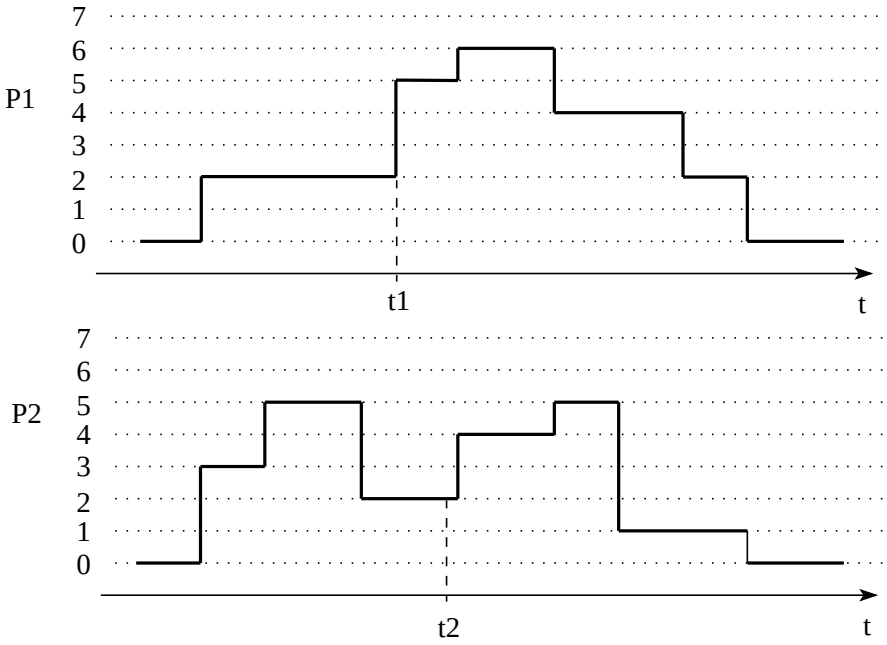
Wie können Deadlocks verhindert werden?

Weitere Deadlocksituationen (1)

Deadlocks bei mehrfach vorhandenen Betriebsmitteln.

Für die Prozesse P_1 und P_2 seien insgesamt 7 Exemplare eines Betriebsmittels vorhanden. Während ihres Ablaufes greifen P_1 bzw. P_2 in folgender Weise auf diese Betriebsmittel zu.

Angenommen P_1 befindet sich innerhalb seines Ablaufes an der Stelle t_1 und P_2 an der Stelle t_2. P_1 besitzt 2 Betriebsmittel und fordert 3 weitere, während P_2 ebenfalls zwei Betriebsmittel besitzt. Es stellt sich die Frage, ob man der Forderung von P_1 stattgeben kann (es sind ja noch 3 Betriebsmittel vorhanden), ohne dass es zu einem Deadlock kommt.

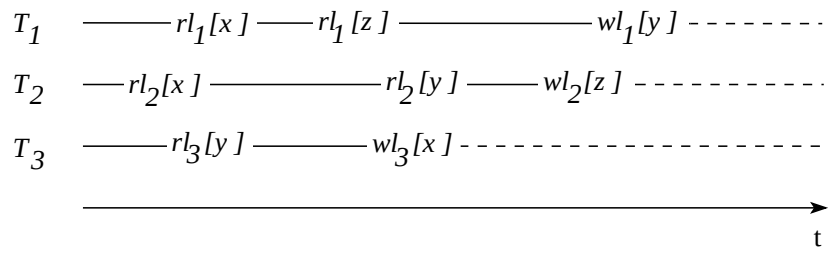


Weitere Deadlocksituationen (2a)

Deadlocks in Datenbanken

Anwendungen auf Datenbanken werden mittels Transaktionen abgewickelt. Eine Transaktion T_i führt Lese- ($r_i[x]$) und Schreiboperationen ($w_i[x]$) auf Objekten x der Datenbank durch. Die Operation von Transaktionen auf demselben Objekt x sind nur dann konfliktfrei, wenn beide lesend zugreifen. Um auf ein Objekt x zugreifen zu können, muss eine Transaktion T_i es zunächst sperren ($rl_i[x]$ oder $wl_i[x]$). Existiert bereits ein kollidierender Zugriff auf das Objekt x, so führt die Sperroperation in einen Wartezustand.

Folgende Zugriffsfolge der Transakionen T_1, T_2 und T_3 führt zu einem Deadlock:



Weitere Deadlocksituationen (2b)

Deadlocks in Datenbanken

Zu dem Grundmodell der Transaktionsverarbeitung existiert eine Reihe von Erweiterungen, die die Deadlockerkennung erschwert:

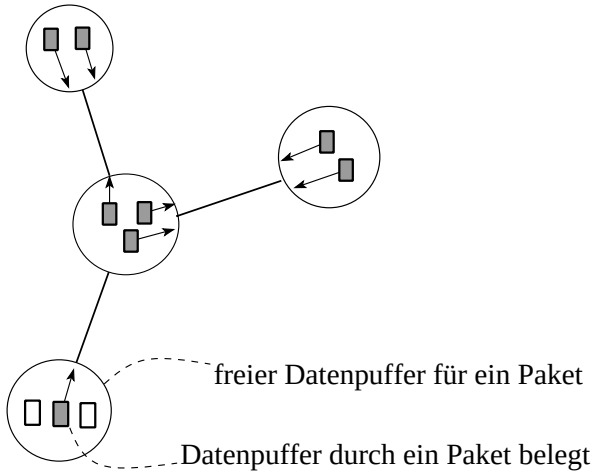
- verschachtelte Transaktionen
  Transaktionen rufen Subtransaktionen auf, die selbständig Objekte sperren können und diese Sperren nach ihrer Termination an ihre Elterntransaktion weiterreichen.
- verteilte Datenbanken
  Transaktionen können Subtransaktionen an vielen Orten eines verteilten Systems besitzen. Für die Deadlockerkennung ist die Zusammenarbeit der Orte des verteilten Systems erforderlich.
- alternative Betriebsmittelforderungen
  Beim Grundmodell benötigt die Transaktion alle angeforderten Objekte (AND-Modell). Bei alternativen Forderungen wird immer nur ein Objekt aus einer Menge von Objekten benötigt (OR-Modell). Typischerweise treten beide Arten von Anforderungen gemeinsam auf (AND-OR-Modell).

Weitere Deadlocksituationen (3)

Deadlocks in paketvermittelnden Rechnernetzten

Gegeben sei ein verteiltes System von Rechnern (Knoten), zwischen denen Nachrichten verschickt werden. Dazu werden Nachrichten in Pakete fester Größe zerlegt, mit der Adresse des Zielknotens versehen und an einen entsprechenden Nachbarknoten verschickt. So hoppeln (engl. *to hop*) die Pakete von Nachbarknoten zu Nachbarknoten. Ist bei einem nächsten Nachbarknoten die Kapazität für die Aufnahme von Paketen erschöpft, so muss das Paket bei dem aktuellen Knoten auf Weiterbeförderung warten.

Es stellt sich die Frage, ob und wie es in den Knoten möglich ist, stets soviel Speicherplatz freizuhalten, dass letztendlich jedes Paket sein Ziel erreichen kann.

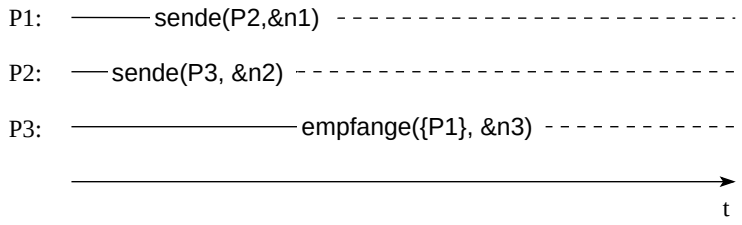


Weitere Deadlocksituationen (4)

Deadlocks in der parallelen und verteilten Programmierung

Gegeben seien drei Prozesse, die mit Hilfe der Operationen `sende()` und `empfange()` kommunizieren.

Die Erkennung von Deadlocks dieser Art ist besonders in verteilten Systemen schwierig. Für einen einzelnen Knoten sind immer nur unmittelbare Wartebeziehungen sichtbar. Deshalb ist es notwendig, mit Hilfe eines verteilten Algorithmus, an dem alle Knoten kooperativ beteiligt sind, die Wartebeziehungen so weit wie möglich sichtbar zu machen, und dann zu entscheiden, ob ein Deadlock vorliegt.



i

Methoden der Deadlockbehandlung (1)

Klassifizierung in Anlehnung an [44] :

(a) Deadlockerkennung und Beseitigung (engl. *detection and recovery*)
Deadlocks können auftreten.

Das Betriebssystem hat Kenntnis über die Wartebeziehungen zwischen Prozessen und muss (von Zeit zu Zeit) prüfen, ob ein Deadlock vorliegt. Nachdem ein Deadlock erkannt ist, hat das Betriebssystem die Aufgabe, einen oder mehrere Prozesse abzubrechen, um den übrigen am Deadlock beteiligten Prozessen eine Fortsetzung zu ermöglichen. Damit ist der erkannte Deadlock beseitigt. Die abgebrochenen Prozesse sind neu zu starten.

Methoden der Deadlockbehandlung (2)

Klassifizierung in Anlehnung an [44] :

(b) Deadlockvermeidung (engl. *avoidance*)
Deadlocks treten nicht auf.

Dazu ist bei jeder Betriebsmittelanforderung zu prüfen, ob der so entstehende Zustand noch **sicher** ist. Ist das nicht der Fall, so wird das Betriebsmittel nicht zugeordnet und der Prozess muss so lange warten, bis sich bei der Zuordnung des Betriebsmittels ein sicherer Zustand ergibt.

Ein Zustand heißt sicher, wenn noch eine Folge von Prozessausführungen existiert, so dass jeder Prozess terminieren kann.

Für die Methode der Deadlockvermeidung ist es notwendig, dass vorweg bekannt ist, wie viele Betriebsmittel ein Prozess jemals anfordern wird.

Beispiel für die Deadlockvermeidung: *Banker's Algorithm*

Methoden der Deadlockbehandlung (3)

Klassifizierung in Anlehnung an [44] :

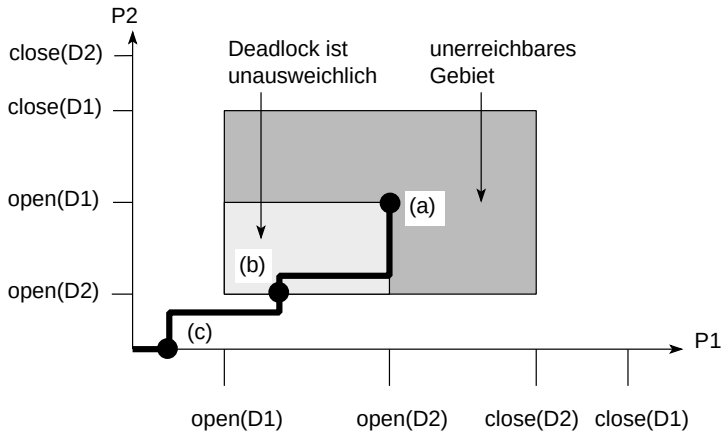
(c) Deadlockverhinderung (engl. *prevention*)
Deadlocks treten nicht auf.

Dazu hat das Betriebssystem beim Start eines Prozesses zu prüfen, ob dieser Prozess zusammen mit den schon aktiven Prozessen in einen Deadlock geraten kann. Ist das der Fall, so wird der Start des Prozesses so lange verzögert, bis ein Deadlock unmöglich wird.

Für die Methode der Deadlockverhinderung ist es notwendig, dass vorweg bekannt ist, wie viele Betriebsmittel ein Prozess jemals anfordern wird.

Methoden der Deadlockbehandlung (4)

Vergleich der Methoden zur Deadlockbehandlung [14]



(a) Der Deadlock ist aufgetreten. Wird beispielsweise Prozess P_2 abgebrochen, so kann P_1 auf das Betriebsmittel D2 zugreifen und schließlich terminieren.

(b) Das Betriebssystem verhindert, dass P_1 auf das Betriebsmittel D2 zugreift, da in diesem Fall ein Deadlock nicht mehr zu vermeiden wäre.

(c) Das Betriebssystem verhindert, dass Prozess P_2 startet, da zusammen mit Prozess P_1 ein Deadlock möglich ist.

[14] anhand des Beispiels von Seite 81.

2.5 Elementare Methoden der Synchronisierung

Frage: Welche Grundoperationen genügen, um den gegenseitigen Ausschluss zweier Prozesse zu gewährleisten?

Antwort: Es genügen zwei atomare Operationen
- Lesen vom Speicher
- Schreiben in den Speicher

Auf dieser Basis funktionieren die Dekkerschen Lösungen und die Lösung von Peterson [31] jeweils zum gegenseitigigen Ausschluss von zunächst zwei Prozessen[15]. Insbesondere wird an den Dekkerschen Lösungen deutlich, welche Fehler und Schwächen eine Lösung haben kann. Die Überlegungen, die dabei anzustellen sind, lassen sich auf andere Synchronisierungsprobleme übertragen.

[15] Alle Lösungen sind in der Programmiersprache C formuliert, wobei das Attribut `common` angeben soll, dass es sich um gemeinsame Variablen handelt.

Dekkersche Lösungen (1)

```
common marke = 1;  /*Initialisierung */
```

```
P1 :: while (marke != 1);     P2 :: while (marke != 2);
      /* kritisches Gebiet */        /* kritisches Gebiet */
      marke = 2;                    marke = 1;
```

Wertung:
Alternatives Betireten des kritischen Gebietes widerspricht der Forderung 3 und auch der Forderung 2 (siehe Gütekriterien Seite 70).

Dekkersche Lösungen (2)

```
#define auf  0
#define zu   1

common int eingang = auf;   /* Initialisierung */


P1 :: while (eingang == zu);  P2 :: while (eingang == zu);
      eingang = zu;                 eingang = zu;
      /* kritisches Gebiet */       /* kritisches Gebiet */
      eingang = auf;                eingang = auf;
```

Wertung:
Es kann mehr als ein Prozess im kritischen Gebiet sein. Das geschieht immer dann, wenn die Anweisungen der Prozesse P_1 und P_2 in etwa zur selben Zeit ausgeführt werden

Damit ist die entscheidenste Forderung 1 (siehe Gütekriterien Seite 70) verletzt.

Dekkersche Lösungen (3)

```
#define drinnen  1
#define draussen 0

common int p1 = draussen; /* Initialisierung */
common int p2 = draussen;


P1 :: p1 = drinnen;           P2 :: p2 = drinnen;
      while (p2 == drinnen);        while (p1 == drinnen);
      /* kritisches Gebiet */       /* kritisches Gebiet */
      p1 = draussen;                p2 = draussen;
```

Wertung:
Es kann dazu kommen, dass P_1 wegen `p2 == drinnen` und P_2 wegen `p1 == drinnen` endlos aufeinander warten.

Damit liegt ein Deadlock vor, womit insbesondere die Forderung 4 (siehe Gütekriterien Seite 70) verletzt ist.

Dekkersche Lösungen (4a)

```
#define drinnen  1
#define draussen 0

common int p1 = draussen;  /* Initialisierung */
common int p2 = draussen;


P1 :: do                        P2 :: do
       {p1 = drinnen;                  {p2 = drinnen;
        if (p2 == drinnen)              if (p1 == drinnen)
          p1 = draussen;                  p2 = draussen;
        while (p2 == drinnen);          while (p1 == drinnen);
       }                               }
      while (p1 == draussen);         while (p2 == draussen);
      /* kritisches Gebiet */         /* kritisches Gebiet */
      p1 = draussen;                  p2 = draussen;
```

Dekkersche Lösungen (4b)

Wertung:
Aufgrund der absoluten Symmetrie zwischen P_1 und P_2 ist es prinzipiell möglich, dass sich beide Prozesse immer wieder in derselben Zeile aufhalten, wobei immer dann, wenn ein Prozess auf `p1` und der andere in derselben Zeile auf `p2` zugreift. In diesem Falle erkennt jeder Prozess, dass der andere auch gerade ins kritische Gebiet möchte und zieht sich zurück. Diese Ausführungsfolge lässt sich beliebig lange fortsetzen und führt dazu, die Prozesse beliebig lange vom Betreten des kritischen Gebietes abzuhalten.

Diese Situation heißt **Livelock** und widerspricht der 4. Forderung (siehe Gütekriterien Seite 70). Dass die oben konstruierte Anweisungsfolge sich tatsächlich ergibt, ist unwahrscheinlich und es genügt bereits, dass ein Prozess, ohne vom anderen unterbrochen zu werden, die Zeilen 3 und 4 ausführt.

Dekkersche Lösungen (5a)

```
#define drinnen  1
#define draussen 0

common int marke = 1;      /* Initialisierung */
common int p1 = draussen;
common int p2 = draussen;
```

```
P1 :: p1 = drinnen;
      if (p2 == drinnen)
        {if (marke == 2)
          {p1 = draussen;
          while (marke == 2);
          p1 = drinnen; }
        while (p2 == drinnen);
        }
      /* kritisches Gebiet */
      marke = 2;
      p1 = draussen;
```

```
P2 :: p2 = drinnen;
      if (p1 == drinnen)
        {if (marke == 1)
          {p2 = draussen;
          while (marke == 1);
          p2 = drinnen; }
        while (p1 == drinnen);
        }
      /* kritisches Gebiet */
      marke = 1;
      p2 = draussen;
```

Dekkersche Lösungen (5b)

Wertung:
Im Gegensatz zur vorangegangenen Lösung ist nun der Livelock verbannt. Mittels der Variable `marke` wird eine Asymmetrie im Verhalten der Prozesse P_1 und P_2 für den Fall erzwungen, dass sie gleichzeitig in das kritische Gebiet wollen. Ähnlich wie bei der ersten Lösung wird dann im Wechsel einer der beiden Prozesse bevorzugt.

Lösung nach Peterson

```
#define drinnen  0
#define draussen 1

common int marke;
common int p1, p2;

P1:: p1 = drinnen;              P2:: p2 = drinnen;
     marke = 1;                      marke = 2;
     while (marke == 1 &&            while (marke == 2 &&
       p2 == drinnen);                 p1 == drinnen);
       /* krit. Gebiet */              /* krit. Gebiet */
     p1 = draussen;                  p2 = draussen;
```

Beachte:
Wollen beide in das kritische Gebiet (d. h. `p1 == drinnen` und `p2 == drinnen`), dann entscheidet `marke`, welcher Prozess in das kritische Gebiet gelangt.

Lösung von Dekker und Peterson

Wertung:

- Drei Variablen sind notwendig, um gegenseitigen Ausschluss zu gewährleisten.
- Für das Betreten des kritischen Gebietes ist relativ viel Programmcode abzuarbeiten.
- Die Lösung eignet sich nur für ein Paar von Prozessen und ist nur schwer auf mehrere Prozesse ausdehnbar[16]. Insbesondere muss die Maximalzahl paralleler Prozesse bekannt sein.

Abhilfe: Mehr Unterstützung durch die Hardware!
Idee: Unteilbare komplexere Operation.

[16] z. B. durch Kaskadierung

Mächtige Hardwarebefehle

Schema in Anlehnung an die 2. Dekker'sche Lösung:

```
#define auf 1
#define zu 0

common eingang;

P ::   /* Testen und Setzen der Variablen  */
       /* eingang als unteilbare Operation */
       do
         <if (eingang == auf){eingang = zu; break;}>
       while(TRUE);
      /* kritisches Gebiet */
      eingang = auf;
```

Sperren von Unterbrechungsanforderungen

Grobe Methode zum Betreten und Verlassen des kritischen Gebietes:

```
P::     :
    /* Sperren von Interrupts */
    /* krit. Gebiet */
    /* Freigabe von Interrupts */
       .
       .
       .
```

Sind alle Interrupts gesperrt, so kann keine Prozessumschaltung stattfinden.
Vorteile:

- Einfache Programmierung.
- Bei beliebig vielen Prozessen anwendbar.

Nachteile:

- Nur im privilegierten Modus möglich.
- Nur für Prozesse auf einem Prozessor geeignet.

Test and Set (1)

Synchronisierung von beliebig vielen Prozessen auf Prozessoren mit gemeinsamen Speicher.
Gegeben:
Variablen `x` und `y` vom Typ `Boolean` und der Maschinenbefehl[17] `ts(x,y)`, der nicht unterbrechbar (`atomar`) folgendes ausführt:

- kopiere `y` auf `x`
- setze dann `y` auf den Wert `false`

Ein Befehl dieser Art (*test and set*) ist auf jedem modernen Prozessor vorhanden.

Frage:
Wie lässt sich das Betreten und Verlassen kritischer Gebiete mit Hilfe des `ts`-Befehls realisieren?

[17] Hier in Form eines Funktionsaufrufes in der Programmiersprache C.

Test and Set (2)

Betreten und Verlassen eines kritschen Gebietes mit Hilfe des `ts`-Befehls:

```
common int y = TRUE;      /* Initialisierung */

    P:: do
         {
           ts(xp,y);
         }
        while(!xp);
        \* kritisches Gebiet */
        y = TRUE;
```

Dabei ist `y` gemeinsame Variable aller Prozesse und hat anfänglich den Wert `TRUE`. Weiterhin besitzt jeder Prozess `P` eine private Variable `xp`, an der er erkennt, ob er ins kritische Gebiet gelangen kann.

Compare and Swap (1)

Vorrangige Zielsetzung: erleichterte Implementierung von Stapel- und Listenoperationen, die bei der Verwaltung von Prozessausführungen sehr häufig vorkommen (z.B. bei der Prozessverwaltung durch das Betriebssystem)

Synapse:

```
cas (ListElement **arg1,
     ListElement *arg2,
     ListElement *arg3)
```

Funktionsweise: atomare Ausführung der Anweisungen:

```
<
  if (*arg!=arg2) return FALSE;
  *arg1=arg3;
  return TRUE;
>
```

Compare and Swap (2)

Eine Implementierung der Operation `pop()` im *lock-free* Stil:

```
ListElement *head

ListElement *pop(){
  if (head== NULL) return NULL;
  while(TRUE){
    ListElement *a1=head;
    ListElement *a2=head;
    ListElement *a3=head->next;
    if(cas(&head,a2,a3)) return a1;
  }
}
```

2.6 Fortgeschrittene Methoden der Synchronisierung

- Semaphore
- Monitore

Definition von Semaphore

Abstrakter Datentyp (vereinfacht)

- Datenstruktur
- Operationen

Semaphore (eingeführt von Dijkstra [12]) bilden einen abstrakten Datentyp:

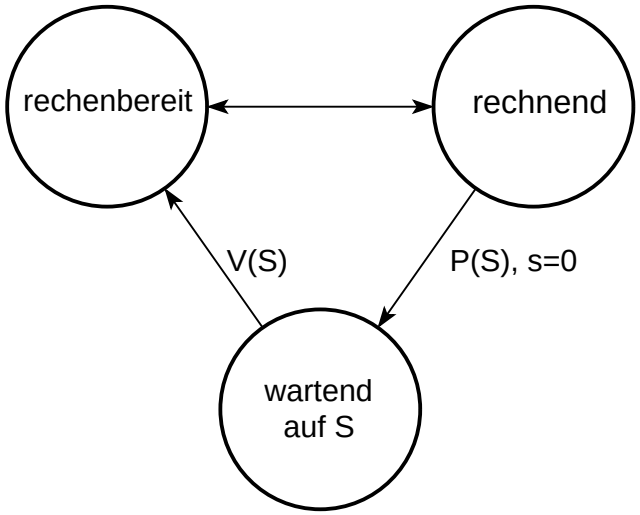
- Datenstruktur: `s` (natürliche Zahl)
- Operationen: `P(s)`, `V(s)`

Die Bedeutung der Operation `P(s)` läßt sich durch die `await`-Anweisung beschreiben:

```
<await s>0 → s=s-1;>
```

Zustandsmodell bei Semaphoren

Zustandsmodell, das auf Betriebssystemebene für Semaphor-Operationen hinterlegt ist



Wirkungsweise der Semaphoroperationen

`P(s)` Mit `P(s)` wird der Wert von `s` um 1 verringert. Wenn `s` bereits den Wert 0 hat, wird die Ausführung dieser Operation so lange verzögert, bis `s` einen Wert größer 0 besitzt.

`V(s)` Mit `V(s)` wird `s` um 1 erhöht.

Beispiel: Bewahrung der Invarianten $I \equiv i = j + b$ mit Hilfe von Semaphoren (`s` sei mit 1 initialisiert).

```
e:: while (TRUE)
     {P(s);
      b++;
      i++;
      V(s);
     }
```

```
v:: while (TRUE)
     {P(s);
      b--;
      j++;
      V(s)
     }
```

Beachte:
Die arithmetischen Operationen auf `b` und `i` bzw. `b` und `j` bilden ein kritisches Gebiet.
Es gilt:
Wann immer sich die Berechnung außerhalb des kritischen Gebietes aufhält, gilt die Invariante.

Mutexe unter Posix

```
#include <pthread.h>

pthread_mutex_t mutex1;
int count;

void increment_count(){
    pthread_mutex_lock(&mutex1);
    count = count + 1;
    pthread_mutex_unlock(&mutex1);
}

int get_count(){
    int c; // lokale Variable
    pthread_mutex_lock(&mutex1);
    c = count;
    pthread_mutex_unlock(&mutex1);
    return (c);
}

int main(void){
  pthread_mutex_init(&mutex1, NULL);
  // Hauptprogramm
  // Erzeugen von Threads
  // Aufruf der Funktionen
  pthread_mutex_destroy(&mutex1);
}
```

Implementierung der Semaphoren

Systemaufrufe `P(s)` und `V(s)` führen zu den beiden Operationen:

`warte_auf(s)` Der Prozess wird in eine Warteschlange ”wartend auf `s`“ eingereiht und ein anderer rechenwilliger Prozess kann in den Besitz des Prozessors gelangen[18].

`wecke(s)` Aus der Warteschlange ”wartend auf `s`“ wird ein Prozess (sofern einer vorhanden ist) in die Schlange der rechenbereiten Prozesse eingereiht.

[18] Damit ist der Funktionsaufruf `P(s)` zwar beendet, aber der ausführende Prozess ist im Zustand ”wartend auf `s`” und kann vorerst nicht fortfahren.

Lösung des Erzeuger-Verbraucher Problems mit Semaphoren

Die Prozesse `e()` und `v()` aufgrund der Entwicklung:

```
e()
{while(TRUE)
 {/* erzeuge ein Datum x */
  P(frei);
  /* frei>=0 && belegt<N */
  puffer[i%N] = x;
  i++;
  V(belegt);
  /* frei>=0 && belegt<=N */

 }
}
```

```
v()
{while(TRUE)
 {
  P(belegt;);
/* belegt>=0 && frei<N */
  y = puffer[j%N];
  j++;
  V(frei);
  /* belegt>=0 && frei<=N */
  /* verbrauche ein Datum y */
 }
}
```

Semaphore zwischen schwergewichtigen Prozessen (1)

Posix Semaphore treten in zwei Varianten auf:

- unbenannte: Das Semaphor wird in einen gemeinsamen Speicherbereich angelegt, mit der Operation `sem_init()`.
- benannte: Mittels eines gemeinsamen Pfadnamens können unterschiedliche schwergewichtige Prozesse auf dieselben Semaphore zugreifen.

Die eigentlichen Semaphoroperationen lauten `sem_wait()` und `sem_post()`.

Semaphore zwischen schwergewichtigen Prozessen (2)

Unbenannte Semaphore

```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(void){
/* place semaphore in shared memory */
  sem_t *sema = mmap(NULL, sizeof(*sema),
      PROT_READ
                  |PROT_WRITE,MAP_SHARED
                  |MAP_ANONYMOUS,
                  -1, 0);
  if (sema == MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
  }

  /* create/initialize semaphore */
  if ( sem_init(sema, 1, 0) < 0) {
    perror("sem_init");
    exit(EXIT_FAILURE);
  }
```

Semaphore zwischen schwergewichtigen Prozessen (3)

```
int nloop=10;
 int pid = fork();
 if (pid < 0) {
   perror("fork");
   exit(EXIT_FAILURE);
 }
 if (pid == 0) {
   /* child process*/
   for (int i = 0; i < nloop; i++) {
     printf
       ("child unlocks semaphore: %d\n",i);
     if (sem_post(sema) < 0) {
         perror("sem_post");
     }
     sleep(1);
   }
   if (munmap(sema, sizeof(sema)) < 0) {
     perror("munmap");
     exit(EXIT_FAILURE);
   }
     exit(EXIT_SUCCESS);
```

Semaphore zwischen schwergewichtigen Prozessen (4)

```
if (pid > 0) {
  /* back to parent process */
  for (int i = 0; i < nloop; i++) {
    printf("parent starts waiting: %d\n", i);
    if (sem_wait(sema) < 0) {
      perror("sem_wait");
    }
    printf("parent finished waiting: %d\n", i);
  }
  if (sem_destroy(sema) < 0) {
    perror("sem_destroy failed");
    exit(EXIT_FAILURE);
  }
  if (munmap(sema, sizeof(sema)) < 0) {
    perror("munmap failed");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

Semaphore zwischen schwergewichtigen Prozessen (5)

Benannte Semaphore

```
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/stat.h>
  :
sem_t   *mysemp;
int     oflag = O_CREAT;
mode_t  mode = 0644;
const char semname[] = "/tmp/mysem"
unsigned int value = 3;
int     sts;
  :
mysemp = sem_open(semname,
    oflag, mode, value);
if (mysemp == (void *)-1) {
  perror(sem_open() failed ");
}
```

Jeder andere schwergewichtige Prozess kann nun mit `sem_open(semname, ...)` auf dieses Semaphor zugreifen.

Monitorkonzept

Ziel des Monitorkonzeptes[19]: Zusammenführung von strukturierter Programmierung und Synchronisierung paralleler Prozesse.
Aus der strukturierten Programmierung stammt das Konzept des abstrakten Datentyps, bestehend aus

- einer gekapselten Datenstruktur und
- einer Menge von Operationen.

Nur mit diesen Operationen kann auf diese Datenstruktur zugegriffen werden[20]. Zur Unterstützung des Programmierens im Großen gibt es einen sichtbaren Teil des abstrakten Datentyps, in dem seine Benutzung festgelegt ist, und einen verdeckten Teil, der seine Implementierung festlegt.

[19] Dem Monitorkonzept unterliegt der Klassengedanke, wobei ein Monitor den Datentyp eines exklusiv benutzbaren Betriebsmittels spezifiziert. Dieser Gedanke wurde insbesondere von Hoare [17] geprägt.
[20] Der Compiler kann prüfen, ob das tatsächlich geschieht.

Eigenschaften der Monitore

Durch Zuordnung von Speicherplatz wird von der Monitorklasse ein Monitorobjekt[21] erzeugt.
Ein Monitorobjekt bildet ein kritisches Gebiet. Es gilt:

> ”In einem Monitor kann zu einem Zeitpunkt höchstens ein Prozess aktiv sein.”

Durch den (Prozedur-) Aufruf einer Operation eines Monitorobjektes

```
monitorobjekt.operation(parameter)
```

betritt der Prozess `i` den Monitor. Das Betreten des Monitors setzt voraus, dass kein anderer Prozess zur Zeit im Monitor aktiv ist und der Prozess `i` als einziger im Monitor aktiv wird. Ist das nicht gegeben, dann muss sich Prozess `i` in eine Schlange “wartend-auf-den-Monitor“ einreihen. Ein im Monitor aktiver Prozess verlässt den Monitor, sobald seine Operation zu Ende geführt ist.

[21] Dieses Konzept lässt sich leicht in eine objektorientierte Programmierumgebung einbetten.

Synchronisierung im Monitor (1)

Es kann notwendig sein,

- Operationen auf den Daten des Monitors nicht zu Ende zu führen, z. B. das Ablegen eines Datums in einem zur Zeit vollen Puffer,
- anderen Prozessen zwischenzeitlich zu ermöglichen auf die Daten des Monitors zuzugreifen, z. B. die Entnahme von Daten aus dem zur Zeit vollen Puffer.

In der Programmiersprache Modula-2 [43] wird dazu der Datentyp `SIGNAL` mit den Operationen `WAIT`, `SEND` sowie `Init` und `Awaited`[22] verwendet. Dabei dient `SIGNAL` als Typ von Variablen, bezüglich derer ein Prozess `i` im Monitor passiv werden kann. Beispiel: Mittels der Synchronisierungsvariablen `voll` soll ein Erzeugerprozess seine Operation `in_puffer` nicht zu Ende führen, wenn kein Platz mehr im Puffer ist.

[22] In dem ursprünglichen Konzept von Hoare [17] werden andere Bezeichner verwendet. Anstelle von `SIGNAL` heißt der Typ `CONDITION` und anstelle von `SEND` heißt die Operation `SIGNAL`. Daneben wird bei Hoare nur noch die Operation `WAIT`.

Synchronisierung im Monitor (2)

Die `WAIT`- und `SEND`-Operation dürfen nur im Monitor aufgerufen werden und haben als Argument eine `SIGNAL`-Variable.
Mit `WAIT(s)` reiht sich ein Prozess `i` in eine Warteschlange ”wartend-auf-`s`” ein und der Monitor ist wieder offen. Erst mit `SEND(s)` kann ein solcher Prozess wieder angestoßen werden.
`SEND(s)` ist wirkungslos, wenn es keinen Prozess gibt, der auf `s` wartet. Andernfalls wird der Prozess `j`, der `SEND(s)` ausführt, in die Warteschlange ”wartend-auf-den-Monitor” eingereiht und ein Prozess `i` im Zustand ”wartend-auf-`s`” ausgewählt und fortgesetzt.
Damit ist gewährleistet, dass

- höchstens ein Prozess im Monitor aktiv ist und
- der Prozess `i` den Monitor so vorfindet, wie er bei der `SEND`-Operation von `j` hinterlassen wird.

Zustandsmodell für Monitore

Ein Monitor ist in einem von zwei Zuständen:

- offen
  Kein Prozess ist im Monitor aktiv, d. h. die Schlange ”wartend-auf-den-Monitor” ist leer.
- belegt
  Ein Prozess ist im Monitor aktiv.

Ein Prozess kann bezogen auf einen Monitor in einem der folgenden Zuständen sein:

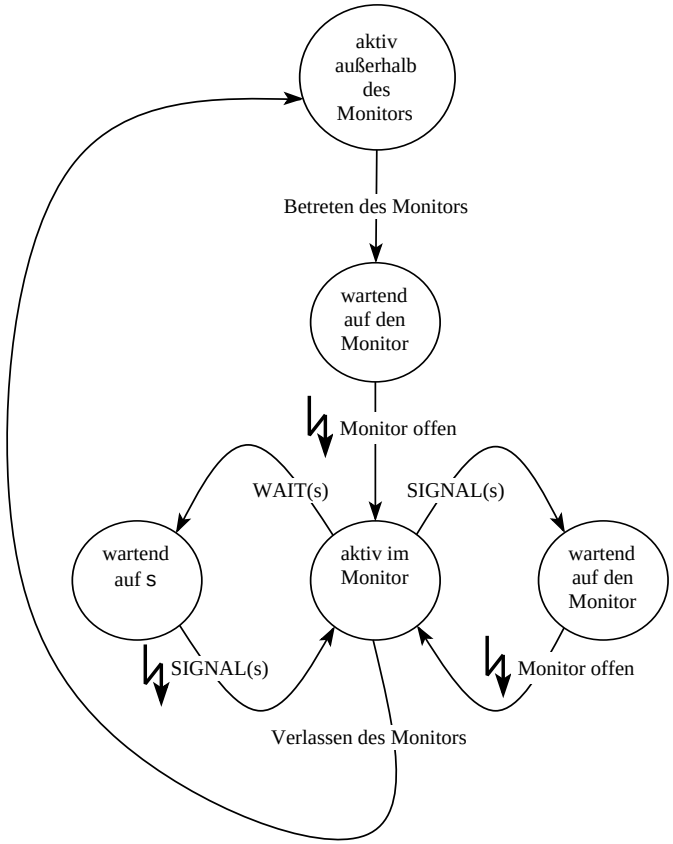
- aktiv
  - außerhalb des Monitors
  - im Monitor
- passiv[23]
  - ”wartend-auf-den-Monitor” beim Betreten des Monitors
  - ”wartend-auf-den-Monitor” durch eine `SEND(s)`-Operation
  - ”wartend-auf-`s`” durch eine Operation `WAIT(s)`

[23] In der Behandlung der Schlangen ”wartend-auf-`s`” können sich spezielle Implementierungen des Monitorkonzeptes nachhaltig unterscheiden (vgl. hierzu [3]). Im Folgenden wird die Implementierung zugrundegelegt, die im Modul `PROCESSES` zur Programmiersprache Modula-2 festgelegt ist (vgl. [43]). Dabei werden alle Schlangen nach der FIFO-Strategie verwaltet und ein `SEND(s)` stößt unmittelbar einen mit `WAIT(s)` wartenden Prozess an.

Zustandsdiagramm für Monitore

Ausgehend vom Zustand ”aktiv-außerhalb-des-Monitors” kann ein Prozess in einer Reihe von Zuständen beobachtet werden:

Das Zeichen ↯ bedeutet, dass der Zustandsübergang nicht von dem beobachteten Prozess ausgelöst wird.



Erzeuger-Verbraucher-Problem in Modula-2 (1)

Benutzerschnittstelle des Monitors `puffer`:

```
DEFINITION MODULE puffer[1];

IMPORT infotyp;
EXPORT QUALIFIED in_puffer, aus_puffer;

PROCEDURE in_puffer(x: infotyp);
PROCEDURE aus_puffer(VAR y:infotyp);

END puffer;
```

Mit Hilfe von Operationen aus dem Modul `SYSTEM` können beliebig viele Prozesse, hier also Erzeuger- und Verbraucherprozesse, erzeugt werden.

Ein Erzeugerprozess, der in seiner lokalen Variablen `x` Informationen erzeugt hat, wird in folgender Weise die Operation `inpuffer` des Monitors `puffer` benutzen:

```
    .
    .
puffer.in_puffer(x);
    .
    .
```

Analog dazu ein Verbraucherprozess:

```
    .
    .
puffer.aus_puffer(y);
    .
    .
```

Erzeuger-Verbraucher-Problem in Modula-2 (2)

```
MODULE puffer[1]

  EXPORT in_puffer, aus_puffer;
  IMPORT SIGNAL, SEND, WAIT, Init, infotyp;

CONST N = 1000       (* Puffergroesse *)

VAR op: INTEGER; (* Aufrufe in_puffer - aus_puffer *)
    voll, leer: SIGNAL;
    i, j: INTEGER;  (* Indizes in den Puffer *)
    speicher_puffer: ARRAY[0..N-1] OF infotyp;

 PROCEDURE in_puffer(x: infotyp)
 BEGIN
    op := op + 1;
    IF op > N THEN WAIT(voll) END;
    speicher_puffer[i MOD N] := x; (* b++ *)
    i:= i + 1;
    IF op =< 0 THEN SEND(leer) END;
 END;

 PROCEDURE aus_puffer(VAR y: infotyp);
 BEGIN
    op := op - 1;
    IF op < 0 THEN WAIT(leer) END;
    y := speicher_puffer[j MOD N]; (* b-- *)
    j:= j + 1;
    IF op >= N THEN SEND(voll) END;
 END;

BEGIN  (* Initialisierung *)
   op := 0;
   i := 0;
   j := 0;
   Init(voll);
   Init(leer);
END.
```

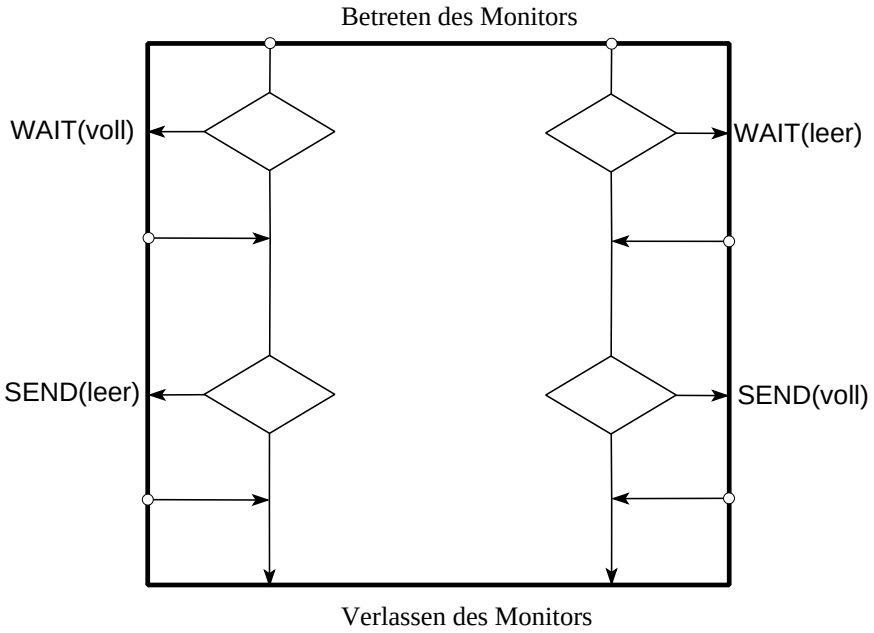
Erhaltung der Monitor-Invarianten

Eine boole'sche Bedingung I, die für einen Prozess an den Beobachtungspunkten

- Betreten des Monitors
- Verlassen des Monitors
- Ausführung von `WAIT(s)`
- Anstoßen eines wartenden Prozesses mit `SEND(s)`

erfüllt ist, heißt Monitor-Invariante.

Beobachtungspunkte für den Monitor `puffer`:



Monitorkonzept und Abwandlungen

Es gibt Abwandlungen vom ursprünglichen Monitorkonzept von Hoare [17]. Die Unterschiede zeigen sich vor allem darin, wie `SIGNAL(...)` ausgeführt wird [3]:

- **SW** *signal and wait*
  Dies verkörpert das klassische Konzept, dass der Prozess, der ein `SIGNAL` ausführt, in die Warteschlange des Monitors eingereiht wird, falls ein Prozess auf diese Operation wartet. Beipeisweise Modula-2 implementiert dieses Konzept.
- **SC** *signal and continue*
  Der Prozess, der das `SIGNAL` ausführt, setzt einen Prozess, der darauf wartet, in den Zustand rechenbereit und führt in seiner Ausführung fort. Beispielsweise Java implementiert dieses Konzept.
- **SU** *signal and urgent wait*
  Die Prozesse, die mit `SIGNAL` in den Zustand ”wartend auf den Monitor” gelangen, werden sobald der Monitor von einem anderen Prozess verlassen wird, vor denjenigen Prozessen geweckt, die den Monitor betreten wollen.

Monitore unter POSIX (1)

Es geht darum, dass sich ein Prozess innerhalb eines kritischen Gebietes verzögert, ohne dass kritische Gebiet für nachfolgende Prozesse zu sperren, d.h. es muss unmittelbar davor freigegeben werden.

Hilfsmittel: *Condition*-Variablen

```
pthread_cond_t cond;
```

Es stellt sich die Frage, wie der so auf `cond` wartende Prozess wieder rechenbereit wird.

Monitore unter POSIX (2)

Folgende Operationen bietet POSIX auf *Condition*-Variablen (u.a.):

Zunächst geht es darum, ein kritisches Gebiet `mutex`, in dem sich ein Prozess befindet, freizugeben und auf `cond` zu warten.

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

Im kritischen Gebiet `mutex` ist ein Prozess anzustoßen, der auf `cond` wartet.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Im kritischen Gebiet `mutex` werden alle Prozesse angestoßen, die auf `cond` warten.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Monitore unter POSIX (3)

```
e()
while(true) {
  /* Erzeuge Datum x */
  pthread_mutex_lock (&mutex);
  while ((i + 1) % N == j)
    pthread_cond_wait(&frei, &mutex);
  puffer[i] = x;
  i = (i + 1) % N;
  pthread_cond_signal (&belegt);
  pthread_mutex_unlock (&mutex);
}
```

```
v()
while(true) {
  pthread_mutex_lock (&mutex);
  while (i == j)
    pthread_cond_wait(&belegt, &mutex);
  y = puffer[j];
  j = (j + 1) % N;
  pthread_cond_signal (&frei);
  pthread_mutex_unlock (&mutex);
  /* Verbrauche Datum y */
}
```

Parallele Prozesse und Monitore in Java

Jedes ausgeführte Java-Programm startet ausgeführt von der *java virtual machine (JVM)* als ein leichtgewichtiger Prozess (*thread*) beginnend bei der `main()`-Routine.

Die vordefinierte Klasse `Thread` aus der Bibliothek `java.lang` steht zur Verfügung, um weitere Thread-Objekte zu erzeugen und zu starten. In der Klasse `Thread` ist die Methode `run()` bereits implementiert, die ein Objekt `x` der Klasse `Thread` mit der Methode `start()` im Aufruf `x.start()` als Thread ausgeführt wird.

Weiterhin kann mit der Schnittstelle `Runnable` erzwungen werden, dass in einer Klasse die Methode `run()` implementiert wird. Hier genügt der Aufruf `start()`, um die `run()`-Methode als Thread auszuführen.

Erzeugung paralleler Prozesse in Java (1)

Klassen zur expliziten Erzeugung von Threads

```
public interface Runnable{
   public void run();
}
...
public class Thread extends Object implements Runnable {
   public Thread();
   public Thread(Runnable target);
   public Thread(String name);
   public Thread(Runnable target, String name);
   ...
   public synchronized native void start();
   public void run();
}
```

Erzeugung paralleler Prozesse in Java (2)

Ableiten eines leichtgewichtigen Prozesses aus der Klasse `Thread`

```
class Prozess extends Thread{
  :
  public Prozess(...){
    // Konstruktor
  }
  :
  public void run(){
    // Prozesstyp
    :
}
```

```
class Hp ... {
  :
  // Erzeuge Prozessobjekt
  Prozess p = new Prozess(...);
  // Prozessausführung
  p.start();
  :
}
```

Erzeugung paralleler Prozesse in Java (3)

Erzeugen eines leichtgewichtigen Prozesses in der Klasse selbst

```
class Irgendwas extends Etwas
     implements Runnable
{
  :
  // z.B. im Konstruktor
  Irgendwas(...){
  // Prozessobjekt
  Thread q = new Thread (this);
  // Referenz this zeigt auf das
  // target_object, wo der Code liegt
  :
  // Prozessausführung
  q.start ();
  }
```

```
  :
  public void run(){
    // Prozesstyp
    :
  }
}
```

Erzeugung paralleler Prozesse in Java (4)

Statusabfragen aus der JVM an die Plattform geht über die Klasse `Runtime`:

```
Runtime.getRuntime().availableProcessors();
```

Die Abfrage steht über die vordefinierte Klasse Runtime aus der Bibliothek `java.lang` zur Verfügung und gibt Auskunft über die Umgebung, in der die Anwendung gerade läuft, d.h. insbesondere auch über die Zahl der zur Verfügung stehenden Prozessoren. Zur Laufzeit kann der Status eines Thread `q` abgefragt werden:

```
if (q.isAlive()){...}
```

Liefert den Wert `true`, sofern der Thread `q` noch ausgeführt wird.

Erzeugung paralleler Prozesse in Java (5)

Möchte man einmalig Operationen, die auf relativ wenigen Codezeilen basieren, durch einen Thread ausführen lassen, so lässt sich dies durch eine anonyme Klasse realisieren:

```
Thread t =
    new Thread(new Runnable() {
        public void run(){
            // Codezeilen
        }
    }, "Thread-Name");
t.start();
```

Dasselbe lässt sich durch Verwendung sog. Lambda-Ausdrücke in verkürzter Schreibweise darlegen:

```
Thread t = new Thread(
  () -> {
       // Codezeilen
  }, "Thread-Name"
);
t.start();
```

Parallele Prozesse in Java mit den Executor-Interface (1)

Mittels des Paketes `java.util.concurrency` bietet Java seit 2004 in der Version SE 5 die Möglichkeit sog. Thread Pools anzulegen. Mittels `Executor`-Methoden lassen sich dann Prozessobjekte als Threads ausführen.

`public interface Executor`

Hinter der Schnittstelle `Executor` steht der Gedanke, Threads als Hülsen zu betrachten. Ist die Hülse leer, d.h. gerade nicht an ein Prozessobjekt gebunden, dann kann man ihr die `run()`-Methode des Prozessobjektes übergeben.

```
class DirectExecutor implements Executor {
   public void execute(Runnable r) {
      r.run();
   }
}
```

Parallele Prozesse in Java mit den Executor-Interface (2)

Mittels der abgeleiteten Schnittstelle

`public interface ExecutorService extends Executor`

lässt sich eine Menge von Threads, ein sog. *thread pool*[24], als Hülsen nutzen, um Prozessobjekte auszuführen.

```
Runnable r1 = new Irgendwas("Erster");
Runnable r2 = new Irgendwas("Zweiter");
Runnable r3 = new Irgendwas("Dritter");
ExecutorService pool = Executors.newFixedThreadPool(2);
pool.execute(r1);
pool.execute(r2);
pool.execute(r3);
```

Einem solchen Thread Pool werden `Runnable` Objekte übergeben, deren `run()`-Methode dann als Thread ausgeführt werden, sofern es im Thread Pool noch freie Plätze gibt. So werden zunächst die `run`-Methoden von `r1` und `r2` ausgeführt. Sobald eine davon endet, kann erst die `run`-Methode von `r3` ausgeführt werden.

[24] man spricht hier auch von den *Workem*.

Speichermodelle der parallelen Programmierung (1)

Gründe für Speichermodelle bei der Thread-Programmierung: Unter JVM wird ein Speichermodell benötigt, um die Auswirkungen der Plattform (d.h. Betriebssystem, Hardware, Compiler, u.s.w.) auf die Programmausführung zu begrenzen. Folgende Einfüsse sind zu beachten:

- Einflüsse durch Verletzung der Cache-Kohärenz-Eigenschaft von Prozessor-Caches (siehe u.a [37])
- Einfluss durch die Umordnung von der Befehlsausführung durch den Compiler

Beispiel: Vertauschung der Reihenfolge von Anweisungen bei der Lösung zum Gegenseitigen Ausschluss nach Peterson:

```
P1::   :                    P2::   :
     p1 = drinnen;               p2 = drinnen;
     marke = 1;                  marke = 2;
       :                           :
```

Speichermodelle der parallelen Programmierung (2)

Das *Java Memory Model* (JVM)[25] basiert auf der partiellen Ordnung von sogenannten Aktionen, die u.a. sind: Lesen und Schreiben von Variablen sowie Sperren und Entsperren mittels Lock-Variablen.
Für zwei Aktionen A und B gilt die *happens-before* Relation, wenn bei der Ausführung von B die Auswirkungen von A vorliegen.
Es gelten folgende Regeln, die durch die JVM zu realisieren sind:

- Eine Aktion in einem Thread *happens-before* einer Aktion desselben Thread in der Reihenfolge der Aufschreibung
- Ein Entsperren der Lockvariablen *happens-before* einem nachfolgenden Sperren derselben Lock-Variablen
- Die *happens-before* Relation ist transitiv

[25] Auch für C++ gibt es seit der Version C++11 ein prinzipiell ähnliches Speichermodell (siehe u.a. in [42] und [37]).

Speichermodelle der parallelen Programmierung (3)

Auswahl von Maßnahmen in Java[26], um Thread-sichere Programmierung zu ermöglichen (siehe [15] und [19]):

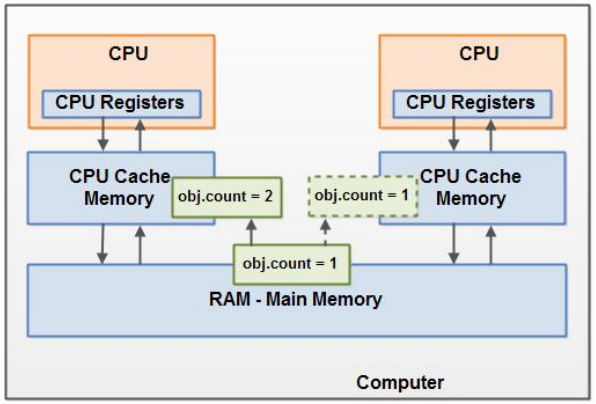
- Direktes Zurückschreiben geänderter Werte in den Hauptspeicher mit dem Attribut `volatile`
- `Callable`- und `Future`-Schnittstellen zur Rückgabe von Werten am Ende der Ausführung von Threads
- Atomic-Variablen wie z.B. `AtomicInteger`, abgebildet auf Compare-And-Swap-Befehle der Hardware
- Lock-Objekte, die beliebige Wartebeziehungen zwischen Threads erlauben[27], z.B. Objekte der Klasse `ReentrantLock`
- Thread-sichere Container, z.B. `ConcurrentHashMap`

[26] In den Versionen SE 5 (2004) und SE 8 (2012)

[27] Dies ist grundsätzlich flexibler als bei bei `synchronized`-Blöcken, die immer an die Objekte gebunden sind, zu denen sie gehören, d.h. nur eine Wartebeziehung zu einem Objekt ist möglich. Gleichzeitig verliert sich dadurch jedoch bei Locks die Beziehung zu einer syntaktischen Einheit, wie sie eine Klasse darstellt.

Speichermodelle der parallelen Programmierung (4)

Generell gilt, dass nicht jede Änderung der Variablenwerte für andere Threads sofort sichtbar werden.



Um das zu beheben, gibt es das `volatile`-Attribut, das garantiert, dass jede Änderung für andere Threads sichtbar ist, indem der geänderte Wert im Cache mit dem Hauptspeicher und den anderen Caches synchronisiert wird. Damit gibt es eine klare *happens-before* Relation zu den Anweisungen vor und nach der Änderung einer `volatile`-Variablen.
Zuweisungen an `volatile`-Variablen können als atomare Anweisungen verstanden werden.

Speichermodelle der parallelen Programmierung (5)

Beispiel: Verwendung des `volatile`-Attributs:

```
  static int a=0,b=0,d=2,e=1,z=0;
  static volatile int c=0;
    :
// Thread 1
  a=8; b=4;
  c=16; // volatile-Variable
  d=2; e=1;
    :
// Thread 2
   z=a+b+c+d+e;
    :
```

Vor und nach der Zuweisung des Wertes 16 an die `volatile`-Variable `c` darf der Complier Anweisungen umordnen. Ist diese Zuweisung durch Thread 1 wirksam geworden, dann wird sie auch für Thread 2 sichtbar und `z` erhält mindestens den Wert 28. Damit bildet die Zuweisung `c=16;` eine sogenannte *StoreLoad-Barrier*, d.h. es gilt die *happens-before*-Relation zwischen den Anweisungen vor und nach der Barriere. So ist beispielsweise garantiert, dass die Anweisung `d=2;` nach der Anweisung `b=4;` ausgeführt wird.

Speichermodelle der parallelen Programmierung (6)

Die Java-Lösung zum gegenseitigen Ausschluss nach Peterson mit `volatile`-Variablen `marke`, `p1` und `p2`:

```
public class Peterson {
    static final int drinnen = 0;
    static final int draussen = 1;
    static final int N_LOOPS = 1000000;

    static volatile int p1 = draussen;
    static volatile int p2 = draussen;
    static volatile int marke = 0;
    static int count = 0;

    public static void main(String[] args)
             throws InterruptedException {
        Runnable P1 = () -> {
            for(int i = 0; i < N_LOOPS; i++){
                p1 = drinnen;
                marke = 1;
                while(marke == 1 && p2 == drinnen);
                count++;
                p1 = draussen;
            }
        };
        Runnable P2 = () -> {
            for(int i = 0; i < N_LOOPS; i++){
                p2 = drinnen;
                marke = 2;
                while(marke == 2 && p1 == drinnen);
                count++;
                p2 = draussen;
            }
        };
        Thread t1 = new Thread(P1);
        Thread t2 = new Thread(P2);
        t1.start(); t2.start(); t1.join(); t2.join();
        System.out.println(count);
    }
}
```

Speichermodelle der parallelen Programmierung (7)

Grenzen bei der Konsistenzerhaltung mittels `volatile`-Attributen:

```
public class VolatileCounter {
  private volatile int counter;
  public void inc() {
    counter++;
  }
  public void dec() {
    counter--;
  }
  public int get() {
    return counter;
  }
}
```

Da arithmetische Operationen mit jeweils einer Lade- und Speicher-Anweisung (hier jeweils sogar atomar) ausgeführt werden, kann es dennoch zu Inkonsistenzen kommen.

Speichermodelle der parallelen Programmierung (8)

Garantierte Konsistenzerhaltung mittels `synchronized`-Attribut:

```
public class SynchronizedCounter {
  private int counter;
  public synchronized void inc() {
    counter++;
  }
  public synchronized void dec() {
    counter--;
  }
  public synchronized int get() {
    return counter;
  }
}
```

Jetzt wird jede Methode ununterbrochen zu Ende ausgeführt.

Speichermodelle der parallelen Programmierung (9)

Java bietet im Paket `java.util.concurrent.atomic` Kapseln für elementare Datentypen mit atomaten Operationen (sog. Atomic Variablen):

```
public class AtomicCounter {
  private AtomicInteger atomicInteger = new AtomicInteger();
  public void inc() {
    atomicInteger.incrementAndGet();
  }
  public void dec() {
    atomicInteger.decrementAndGet();
  }
  public int get() {
    return atomicInteger.intValue();
  }
}
```

Dieser Mechanismus basiert intern darauf, den gleichzeitigen Zugriff auf die die jeweilige Variable durch Compare-And-Swap-Befehle zu schützen und damit ohne Systemaufrufe auszukommen.

Erzeuger-Verbraucher-Problem in Java

Lösung nach [15], die mit verschiedenen Lock-Objekten:

```
public class BoundedFIFOQueueWithLock<T>{
    private final Object[] data;
    private int head;
    private int tail;
    private int count;
    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
    public BoundedFIFOQueueWithLock(int cap){
        data = new Object[cap];
        head = 0;
        tail = 0;
        count = 0;
    }
    public void put(T elem) throws InterruptedException{
        lock.lock();
        try{
            while (count == data.length){notFull.await();}
            count++;
            data[tail] = elem;
            tail = (tail+1)%data.length;
           if (count == 1){notEmpty.signalAll();}
        }
        finally{
            lock.unlock();
        }
    }
    public T get() throws InterruptedException{
        lock.lock();
        try{
            while (count == 0){notEmpty.await();}
            count--;
            T obj = (T) data[head];
            data[head] = null;
            head = (head+1)%data.length;
            if (count == data.length-1){notFull.signalAll();}
        return obj;
         }
        finally{
            lock.unlock();
        }
    }
```

2.7 Paradigmen der parallelen Programmierung

- Erzeuger-Verbraucher-Problem
- Leser-Schreiber-Problem
- Fünf-Philosophen-Problem
- *barrier*-Problem
- *barber shop*-Problem
- *cigarette smoker*-Problem
- *jungle bridge*-Problem
- *roller-coaster*-Problem

Das wissenschaftliche Paradigma

Aufgabenstellungen, die dazu beitragen, wesentliche Eigenschaften eines ganzen Fachgebietes zu verdeutlichen, werden als paradigmatisch[28] bezeichnet. Sie sollen ureigene und wiederkehrende Merkmale des Fachgebietes in einer knappen und griffigen Form wiedergeben. Voraussetzung ist, dass sie dem Kreis der auf diesem Fachgebiet tätigen Wissenschaftler vertraut sind und damit als Ausdrucksmittel für wissenschaftliche Erkenntnisse herangezogen werden können.
Als paradigmatisch für den Bereich der parallelen Programmierung sind die bereits vorgestellten Probleme

- das Erzeuger-Verbraucher-Problem und
- der gegenseitige Ausschluss

zu betrachten. Die von diesen Problemen abgeleiteten Fragestellungen und Lösungsansätze lassen sich auf das ganze Fachgebiet ausdehnen und anwenden.

[28] Der Begriff des wissenschaftlichen Paradigmas stammt von dem Physiker und Wissenschaftstheorethiker Thomas S. Kuhn [22] . Vereinfacht versteht Kuhn unter einem Paradigma alles das, was den Wissenschaftlern eines Fachgebietes gemeinsam ist, seien es einheitliche Vorgehensweisen, wohlverstandene Eigenschaften oder signifikante Bewertungskriterien.

Das Leser-Schreiber-Problem (1)

Ein häufig auftretendes Problem der parallelen Programmierung ist der lesende und schreibende Zugriff auf einen Datenbestand `D` [9] .

Im Sinne eines abstrakten Datentyps sollen zu diesem Zweck die Operationen

- `schreib(D, x)`
  ein Prozess schreibt seine private Variable `x` in den Datenbestand `D`,
- `lies(D, y)`
  ein Prozess liest seine private Variable `y` aus dem Datenbestand `D`

vorhanden sein. Eine 1. Lösung ist auf der Basis eines Semaphors `sD` (Initialwert `1`) gegeben als:

```
schreib(D, x){
  P(sD);
  D = x;
  V(sD);
}
```

```
lies(D, y) {
  P(sD);
  y = D;
  V(sD);
}
```

Wertung:
Von der Problemstellung her stören sich die Leser nicht, wenn sie gleichzeitig auf dem Datenbestand `D` arbeiten. Bei der obigen Lösung kann der nächste Leser jedoch frühestens dann an den Datenbestand `D`, wenn ein vorangehender Leser seinen ggf. lange dauernden Zugriff beendet.

Das Leser-Schreiber-Problem (2)

Invariante Eigenschaften beim Leser-Schreiber-Problem

- `anzleserD`
  Anzahl der Prozesse, die zu einem Zeitpunkt lesenden Zugriff auf `D` haben.
- `anzschreiberD`
  Anzahl der Prozesse,die zu einem Zeitpunkt schreibenden Zugriff auf `D` haben.

Drei Zustände sind zu unterscheiden:

F das Datum `D` ist nicht beansprucht:
`anzleserD==0 && anzschreiberD==0`

L das Datum `D` befindet sich im lesenden Zugriff:
`anzleserD>0 && anzschreiberD==0`

S das Datum `D` befindet sich im schreibenden Zugriff:
`anzleserD==0 && anzschreiberD==1`

Das Leser-Schreiber-Problem (3)

Protokollartig geschützte Zugriffe auf den Datentyp `D`

Leser- bzw. Schreiber bedienen sich beim Zugriff auf `D` spezieller Operationen:

```
schreib(D, x)
{
 anf_schreib(D);
 /* S */
 D = x;
 ende_schreib(D);
}
```

```
lies(D, y)
{
 anf_lies(D);
 /* L */
 y = D;
 ende_lies(D);
}
```

Hält sich ein Prozess außerhalb dieser Anweisungen auf, dann gilt:

$$F \vee S \vee L$$

Das Leser-Schreiber-Problem (4)

2. Lösung: Leserseite

Demjenigen Leser, der als erster Zugang zu `D` erhält, fällt die Aufgabe zu, anderen Schreibern den Zugang zu verwehren. Dieser Leser muss im Zustand F die Operation `P(sD);` ausführen und dann in den Zustand L übergehen.

In der Variablen `anzleserD` werden die Leser gezählt, sodass der letzte Leser, der seinen Zugriff auf `D` aufgibt und den Zustand F wiederherstellt, ein `V(sD);` ausführt. Zum Schutz der Variablen `anzleserD` ist ein Semaphor `mutexD` notwendig.

```
anf_lies(D) {
  P(mutexD);
  if (anzleserD==0) /* F || S */ P(sD);
  anzleserD++; /* L */
  V(mutexD);
}

ende_lies(D) {
  P(mutexD);
  anzleserD--;
  if (anzleserD==0) /* F */ V(sD);
  V(mutexD);
}
```

Das Leser-Schreiber-Problem (5)

3. Lösung: Leser und Schreiber (in Anlehnung an [9])

Schreiberseite:

```
anf_schreib(D){
   P(mutex1);
   anzschreiberD++;
   if (anzschreiberD==1) P(lD);
   V(mutex1);
   P(sD);}

ende_schreib(D){
   V(sD);
   P(mutex1);
   anzschreiberD--;
   if (anzschreiberD==0) V(lD);
   V(mutex1);}
```

Leserseite:

```
anf_lies(D){
   P(mutex3);
   P(lD);
   P(mutex2);
   anzleserD++;
   if (anzleserD==1) P(sD);
   V(mutex2);
   V(lD);
   V(mutex3);}

ende_lies(D){
   P(mutex2);
   anzleserD--;
   if (anzleserD==0) V(sD);
   V(mutex2);}
```

Leser-Schreiber-Problem in Java

„... Besitzt ein Thread eine Lesesperre und versucht ein weiterer Thread, eine Schreibsperre zu erhalten, muss dieser warten. Die Frage ist nun, was passiert, wenn jetzt ein Thread kommt, der eine Lesesperre erwerben möchte. Aus Gründen des Durchsatzes bekommt er sie sofort ...“ [15]

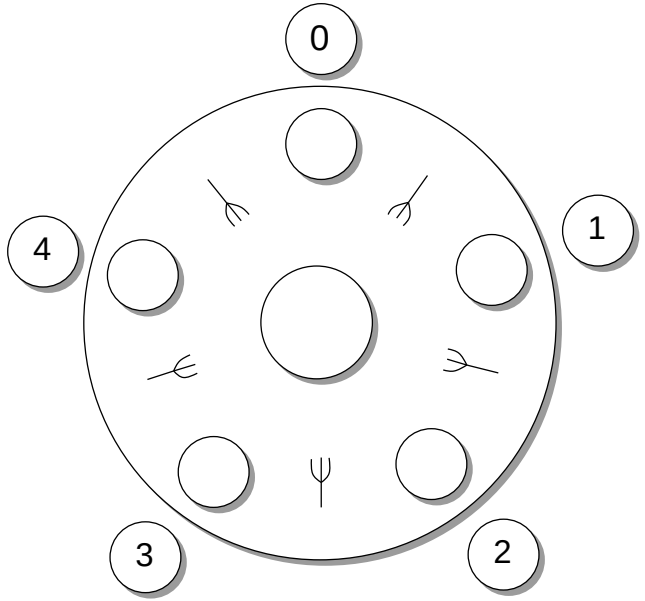
```
public class MyHashMap<K, V>{
private final Map<K, V> hashMap;
// non-fair wegen der Performance!
private final ReadWriteLock lock
    = new ReentrantReadWriteLock();
private final Lock readLock
    = lock.readLock();
private final Lock writeLock
    = lock.writeLock();
public MyHashMap(Map<K, V> map)
{hashMap = map;}

public void put(K key, V value){
  writeLock.lock();
  try{hashMap.put(key, value);}
  finally{writeLock.unlock();}
}
```

```
public V get(K key){
  readLock.lock();
  try{return hashMap.get(key);}
  finally{readLock.unlock();}
}

public V remove(K key){
  writeLock.lock();
  try{return hashMap.remove(key);}
  finally{writeLock.unlock();}
}

public boolean containsKey(K key){
  readLock.lock();
  try{return hashMap.containsKey(key);}
  finally{readLock.unlock();}
}
}
```

Das Fünf-Philosophen-Problem (1)

Problembeschreibung:[29]

Fünf Philosophen sitzen um einen runden Tisch herum und verbringen ihr Leben mit Denken und Essen. In der Mitte des Tisches steht ein Topf mit Spaghetti. Jeder Philosoph hat einen eigenen Teller und zwischen benachbarten Philosophen liegt jeweils eine Gabel.



Die Gabeln zu seiner Linken und Rechten benötigt ein Philosoph, wenn er hungrig wird und essen möchte. Hat er beide in seinen Besitz gebracht, dann isst der Philosoph solange, bis er satt ist, um dann die Gabeln an ihre ursprüngliche Position zurückzulegen.

[29] Das Fünf-Philosophen-Problem stammt von Dijkstra [12] und ist aufgrund seiner Bildhaftigkeit eines der bekanntesten Standardprobleme der parallelen Programmierung. Beschrieben wird ein Ablauf, der sich bei knappen Betriebsmitteln für jeden der fünf Philosophen in gleicher Weise stellt. Gesucht ist deshalb eine symmetrische, sichere und faire Lösung.

Das Fünf-Philosophen-Problem (2)

Rahmen für die Lösung des Fünf-Philosophen-Problems

Vorgegeben sind fünf Semaphoren, die die freien bzw. vergebenen Gabeln darstellen:

```
#define N 5

semaphor gabel[N] = {1, 1, 1, 1, 1};

philosoph(int i)
   {
    while (TRUE)
      {
       denken(i);
       will_essen(i);
       essen(i);
       will_denken(i);
      }
   }
```

In einer Parallelanweisung werden die fünf Philosophenprozesse gestartet:

```
[   philosoph(0);
 || philosoph(1);
 || philosoph(2);
 || philosoph(3);
 || philosoph(4);
]
```

Während die Funktionen `denken(i)` und `essen(i)` nicht weiter von Interesse sind, besteht die Aufgabe innerhalb des vorgegebenen Rahmens darin, Lösungen für `will_essen(i)` und für `will_denken(i)` anzugeben.

Das Fünf-Philosophen-Problem (3)

1. Lösung:

Da sich die Aufgabe für alle Philosophen in gleicher Weise stellt, hier eine symmetrische[30] Lösung des Problems:

```
will_essen(int i)
   {
    P(gabel[i]);
    P(gabel[(i+1)%N]);
   }
```

```
will_denken(int i)
   {
    V(gabel[(i+1)%N]);
    V(gabel[i]);
   }
```

Diese Lösung ist, insbesondere aufgrund ihrer Symmetrieeigenschaft nicht deadlockfrei. Ein Deadlock entsteht, wenn jeder Philosoph `i` im Besitz der Gabel `i` ist.

[30] In Anlehnung an Bougé [6] bezieht sich (semantische) Symmetrie auf die Austauschbarkeit der Berechnungen zwischen parallelen Prozessen. Ein paralleles Programm ist demnach symmetrisch, wenn für jeden Automorphismus σ, der die Nachbarschaftsbeziehungen beläßt, gilt, daß die Berechnungsfolge C_i eines Prozesses `i` bei Umnumerierung der Prozeß-Indizes auch von Prozeß $\sigma(i)$ ausgeführt werden könnte:

$$\forall C_i \exists \acute{C}_{\sigma(i)} : \acute{C}_{\sigma(i)} = \sigma(C_i)$$

Das Fünf-Philosophen-Problem (4)

2. Lösung des Fünf-Philosophen-Problems durch Hierarchisierung der Betriebsmittel.

Hierarchisierung ist gegeben, wenn jeder Philosoph zuerst auf die Gabel mit niedrigerem und dann auf die Gabel mit höherem Index zugreifen wird. Sei:

```
int min(int i)                int max(int i)
{switch(i)                    {switch(i)
  {case 0,1,2,3: return i;      {case 0,1,2,3: return i+1;
   case 4:       return 0;       case 4:       return i;
  }                             }
}                             }
```

Mit der folgenden Anordnung der Aufrufe von `gabel()`

```
will_essen(int i)
   {P(gabel[min(i)]);
    P(gabel[max(i)]);
   }
```

wird immer erst nach der niedrigeren Gabel und dann nach der höheren gegriffen. So kann es nicht mehr zum Deadlock kommen.

Das *barrier*-Problem (1)

Ein vielfach auftretendes Problem ist damit beschrieben, dass eine Menge von Prozessen erst dann weiterarbeiten soll, wenn alle eine gewisse Stelle im Programm passiert haben. Diese Stelle heißt Barriere.

Eine solche Situation ergibt sich dann, wenn zwei Parallelanweisungen hintereinander auszuführen sind:

```
PARBEGIN
  P(0); ... P(N-1);
PAREND
S;
PARBEGIN
  P(0); ... P(N-1);
PAREND
```

Damit bildet die `PAREND`-Anweisung eine Barriere.

Das *barrier*-Problem (2)

Beispiel für Barrier unter Benutzung der Posix-Bibliothek:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>

#define THREAD_COUNT 10

pthread_barrier_t mybarrier;

void* threadFn(void *id_ptr) {
  int thread_id = *(int*)id_ptr;
  int wait_sec = 1 + rand() % 10;
  printf("thread %d:
         Wait for %d seconds.\n", thread_id, wait_sec);
  sleep(wait_sec);
  pthread_barrier_wait(&mybarrier);
  return NULL;
}
```

```
int main() {
  int i;
  pthread_t ids[THREAD_COUNT];
  int short_ids[THREAD_COUNT];

  srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);

  for (i=0; i < THREAD_COUNT; i++) {
    short_ids[i] = i;
    pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
  }

  pthread_barrier_wait(&mybarrier);

for (i=0; i < THREAD_COUNT; i++) {
    pthread_join(ids[i], NULL);
  }

  pthread_barrier_destroy(&mybarrier);
}
```

Das *barrier*-Problem (3)

Eine Lösung des Barrier-Problems sieht so aus, dass ein weiterer Prozess `Q` die Arbeit der anderen N Prozesse überwacht und synchronisiert. Dazu seien zwei Semaphore `s` und `t` vorhanden, eines womit Prozess `Q` angestoßen wird und ein weiteres, das die N Prozesse solange anhält, bis der Prozess `Q` sie wieder frei gibt.

```
Q:: cnt=0;
  while(TRUE) do{
    P(t);
    cnt++;
    if (cnt==N)
      for(i=N-1, i>0, i--) {cnt--; V(s)};
  }
```

Dabei besteht die Barriere `barrier()` jedes der `N` Prozesses darin, dass er in Folge erst eine Operation `V(t)` und dann `P(s)` ausführt. Beide Semaphore sind mit dem Wert 0 initialisiert.

Das *barrier*-Problem (4)

Beispiel für Barrier unter Benutzung der `CyclicBarrier` von Java:

```
class Solver {
   final int N;
   final float[][] data;
   final CyclicBarrier barrier;

   class Worker implements Runnable {
     int myRow;
     Worker(int row) { myRow = row; }
     public void run() {
       while (!done()) {
         processRow(myRow);

         try {
           barrier.await();
         } catch (InterruptedException ex) {
           return;
         } catch (BrokenBarrierException ex) {
           return;
         }
       }
     }
   }

   public Solver(float[][] matrix) {
     data = matrix;
     N = matrix.length;
     Runnable barrierAction =
       new Runnable() { public void run() { mergeRows(...); }};
     barrier = new CyclicBarrier(N, barrierAction);

     List<Thread> threads = new ArrayList<Thread>(N);
     for (int i = 0; i < N; i++) {
       Thread thread = new Thread(new Worker(i));
       threads.add(thread);
       thread.start();
     }

     // wait until done
     for (Thread thread : threads)
       thread.join();
   }
}
```

Literaturverzeichnis

[1] *DIN 44300, Normen über Informationsverarbeitung*. Beuth Verlag, Berlin, 1975.

[2] Albrecht Achilles. *Betriebssysteme*. eXamen.press. Springer-Verlag, Berlin, 2006.

[3] G. R. Andrews. *Concurrent Programming*. The Benjamin/Cummings Publishing Company, 1991.

[4] G. Bengel, Chr. Baun, M. Kunze, and K.-U. Stucky. *Masterkurs Parallele und Verteilte Systeme*. Vieweg+Teubner, Wiesbaden, 2008.

[5] Lothar Borrmann. Kleine und kleinste Betriebssysteme mit Mikro- und Nanokernen. *Informationstechnik und Technische Informatik it+ti, Oldenbourg Verlag*, 38(2):18–25, 1996.

[6] L. Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Informatica*, 25:179–201, 1988.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[8] E.G. Coffman, M.J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(3):67–68, 1971.

[9] P.J. Courtois, D.L. Heymans, and F. Parnas. Concurrent control with readers and writers. *CACM*, 14(10):667–668, October 1974.

[10] H. M. Deitel. *Operating Systems*. Addison Wesley, Reading, Massachusetts, 1990.

[11] P.J. Denning. The working set model for program behavior. *CACM*, 11(5):323–333, May 1968.

[12] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[13] Ramez Elmasri, Gil Carrik, and David Levine. *Operating Systems - A Spiral Approach*. McGraw-Hill, Boston, 2009.

[14] M. Flynn. Very high speed computing systems. *Proc. of the IEEE*, 54(9):1901–1909, 1966.

[15] Jörg Hettel and Man Tien Tran. *Nebenläufige Programmierung mit Java*. Dpunkt Verlag, Heidelberg, 2016.

[16] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[17] C.A.R. Hoare. Monitors: An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.

[18] Matthias Homann. *OSEK - Betriebssystem-Standard für Automotive und Embedded Systeme*. mitp-Verlag, Bonn, 2005.

[19] Cay Horstmann. *Big Java - Late Objects*. John Wiley & Sons, New York, 2013.

[20] D.G. Kendall. Stochastic processes occuring in the theory of queues and their analysis by the method of embedded Markov chains. *Annals of Math. Statistics,*, 24:338–354, 1953.

[21] Nathan Koyzra. *Mastering concurrency in GO*. Packt-Publishing, Bermingham, United Kingdom, 2014.

[22] Thomas Kuhn. *Die Struktur der wissenschaftlichen Revolution*. Surkamp Verlag, Frankfurt, 1969.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[24] J. D. C. Little. A proof of the queueing formula $L = \lambda W$. *Operations Research*, 9:383 – 387, 1961.

[25] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Workshop on Parallel and Distributed Systems*, pages 215–226, Chateau de Bonas, France, October 1988. Elsevier.

[26] Jörg Mühlbacher. *Betriebssysteme*. Universitätsverlag Rudolf Trauner, Linz, 2009.

[27] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184, 2016.

[28] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[29] Gary Nutt. *Operating Systems - A Modern Perspective*. Addison-Wesley, Reading, 1997.

[30] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference on Management of Data*, San Francisco, June 1988.

[31] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[32] Hans-Joachim Picht. *Xen Kochbuch*. O'Reilly Verlag, K¨oln, 2009.

[33] Gerals J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architecture. *Communications of the ACM*, 7(7):412–421, July 1974.

[34] Thomas Rauber and Gundula Rünger. *Parallele Programmierung*. Springer-Verlag, Berlin, 2nd. ed. edition, 2007.

[35] Lutz Richter. *Betriebssysteme*. Teubner Verlag, Stuttgart, 1985.

[36] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, 4 edition, 1994.

[37] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool Publishers, Williston, Vermont, 2011.

[38] Brian L. Stuart. *Principles of Operating Systems: Design and Application*. Thomson Learning, Boston, Mass., 2008.

[39] A. S. Tanenbaum. *Operating Systems*. Prentice-Hall International Editions, Englewood Cliffs, NJ, 1987.

[40] Andrew Tanenbaum. *Moderne Betriebssysteme*. Hanser Verlag, München, 1995.

[41] Michael Weber. *Verteilte Systeme*. Spektrum Akademischer Verlag, Heidelberg, 1998.

[42] Anthony Williams. *C++ Concurrency in Action - Practical Multithreading*. Manning Press, New York, 2012.

[43] Nikolaus Wirth. *Programming in Modula-2*. Springer-Verlag, Heidelberg, 1985.

[44] Dieter Zöbel. The deadlock problem: A classifying bibiography. *Operating Systems Review*, 17(4):5–15, 1983.