



Betriebssysteme und nebenläufige Programmierung

SoSe 2026

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 20. März 2026



Betriebssysteme und nebenläufige Programmierung

SoSe 2026

3 Synchronisation



Klassen der Interaktion zwischen Threads (nach Stallings)

- ➔ Threads kennen sich gegenseitig nicht
 - ➔ nutzen aber gemeinsame Ressourcen (Geräte, Dateien, ...)
 - ➔ unbewußt (**Wettstreit**)
 - ➔ bewußt (**Kooperation durch Teilen**)
 - ➔ wichtig: **Synchronisation** (☞ 3)
- ➔ Threads kennen sich (d.h. die Prozeß-/Threadkennungen)
 - ➔ **Kooperation durch Kommunikation** (☞ 4)
- ➔ **Anmerkungen:**
 - ➔ Threads können ggf. in unterschiedlichen Prozessen liegen
 - ➔ in der Literatur hier i.a. keine klare Unterscheidung zwischen Threads und Prozessen

3 Synchronisation ...



Inhalt (1):

- ➔ Einführung und Motivation
- ➔ Wechselseitiger Ausschluß
- ➔ Wechselseitiger Ausschluß mit aktivem Warten
 - ➔ Lösungsversuche, korrekte Lösungen
- ➔ Synchronisation in Mehrprozessorsystemen
- ➔ Semaphore

- ➔ Tanenbaum 2.3.1-2.3.6
- ➔ Stallings 5.1-5.4.1

Inhalt (2):

- ➔ Klassische Synchronisationsprobleme
 - ➔ Erzeuger/Verbraucher-Problem
 - ➔ Leser/Schreiber-Problem
- ➔ Monitore
- ➔ Schnittstellen zur Thread-Synchronisation
- ➔ Speicherkonsistenz
- ➔ *Lock-Free* Datenstrukturen
- ➔ *Transactional Memory*

- ➔ Tanenbaum 2.4.2, 2.4.3, 2.3.7
- ➔ Stallings 5.4.4, 5.5

Anmerkungen zu Folie 142:

Eine tiefergehende Behandlung des Themas Multiprozessor-Programmierung, insbesondere zu den Themen Speichermodelle, Synchronisationsprimitive, *Lock-Free* Datenstrukturen und *Transactional Memory* finden Sie in dem Buch

- ➔ Maurice Herlihy, Nir Shavit. *The Art of Multiprocessor Programming*, Elsevier, 2012. <https://dl.acm.org/doi/pdf/10.5555/2385452> (zugreifbar aus dem Netz der Universität Siegen)

Einiges zum Thema Synchronisation und *Transactional Memory* finden Sie auch in dem Vorlesungsskript

- ➔ David Sabel: *Prinzipien, Algorithmen und Modelle der Nebenläufigen Programmierung*.
<https://www.tcs.ifi.lmu.de/lehre/ws-2019-20/concurrent/material/skript>

- ➔ Mehrprogrammbetrieb führt zu Nebenläufigkeit
 - Abarbeitung im Wechsel (praktisch) äquivalent zu echt paralleler Abarbeitung
- ➔ Mehrere Threads können gleichzeitig versuchen, auf gemeinsame Ressourcen zuzugreifen
 - Beispiel: Drucker
- ➔ Für korrekte Funktion in der Regel notwendig:
 - zu einem Zeitpunkt darf nur jeweils einem Thread der Zugriff erlaubt werden

Beispiel: Drucker-Spooler

- ➔ Threads tragen zu druckende Dateien in Spool-Bereich ein:
 - Spooler-Verzeichnis mit Einträgen 0, 1, 2, ... für Dateinamen
 - zwei gemeinsame Variable:
 - out: nächste zu druckende Datei
 - in: nächster freier Eintrag
 - in gemeinsamem Speicherbereich oder im Dateisystem
- ➔ Druck-Thread überprüft, ob Aufträge vorhanden sind und druckt die Dateien

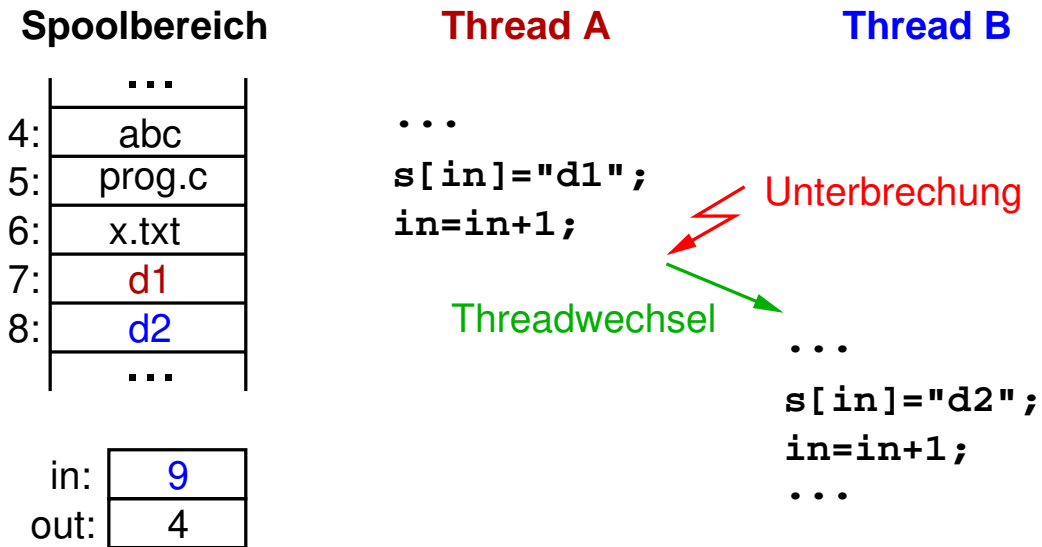
3.1 Einführung und Motivation ...



(Animierte Folie)



Beispiel: Drucker-Spooler, korrekter Ablauf



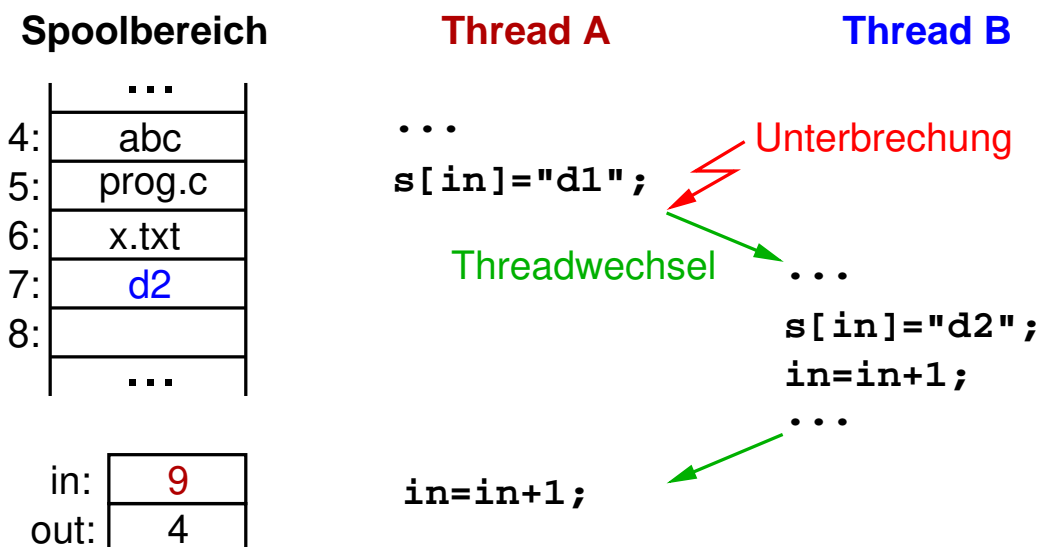
3.1 Einführung und Motivation ...



(Animierte Folie)



Beispiel: Drucker-Spooler, fehlerhafter Ablauf



➔ **Race Condition**



Arten der Synchronisation

- ➔ **Sperrsynchrisation (wechselseitiger Ausschluß)**
 - ➔ stellt sicher, daß Aktivitäten in verschiedenen Threads **nicht gleichzeitig** ausgeführt werden
 - ➔ d.h., die Aktivitäten werden nacheinander (in beliebiger Reihenfolge) ausgeführt
 - ➔ z.B. kein gleichzeitiges Drucken

- ➔ **Reihenfolgesynchronisation**
 - ➔ stellt sicher, daß Aktivitäten in verschiedenen Threads in einer **bestimmten Reihenfolge** ausgeführt werden
 - ➔ z.B. erst Datei erzeugen, dann lesen

3.2 Reihenfolgesynchron. mit aktivem Warten



- ➔ Beispiel: Thread 1 darf eine Datei erst öffnen, nachdem Thread 0 sie erzeugt hat

- ➔ Idee: Nutzung einer Boole'schen Variable
 - ➔ zeigt an, ob Wartebedingung erfüllt ist
 - ➔ Warten erfolgt durch eine (leere) Schleife, die die Bedingung laufend testet (**aktives Warten**)

- ➔ Im Beispiel:

Initialisierung:

```
ready = false; // zeigt an, ob Datei erzeugt wurde
```

Thread 0

```
// Datei erzeugen  
ready = true;
```

Thread 1

```
while (!ready); // aktives Warten  
Datei öffnen
```

➔ Kritischer Abschnitt

➔ Abschnitt eines Programms, der Zugriffe auf ein gemeinsam genutztes Objekt (**kritische Ressource**) enthält

➔ Wechselseitiger Ausschluß von Aktivitäten

- ➔ zu jeder Zeit darf nur ein Thread die Aktivität ausführen
- ➔ Sperrsynchrisation

➔ Gesucht: Methode zum wechselseitigen Ausschluß kritischer Abschnitte

3.3 Wechselseitiger Ausschluß ...



Beispiel: Drucker-Spooler mit wechselseitigem Ausschluß

Spoolbereich

	...
4:	abc
5:	prog.c
6:	x.txt
7:	d1
8:	d2
	...

in:	9
out:	4

Thread A

```
begin_region();  
s[in]="d1";  
in=in+1;  
end_region();
```

Thread B

```
begin_region();  
s[in]="d2";  
in=in+1;  
end_region();
```

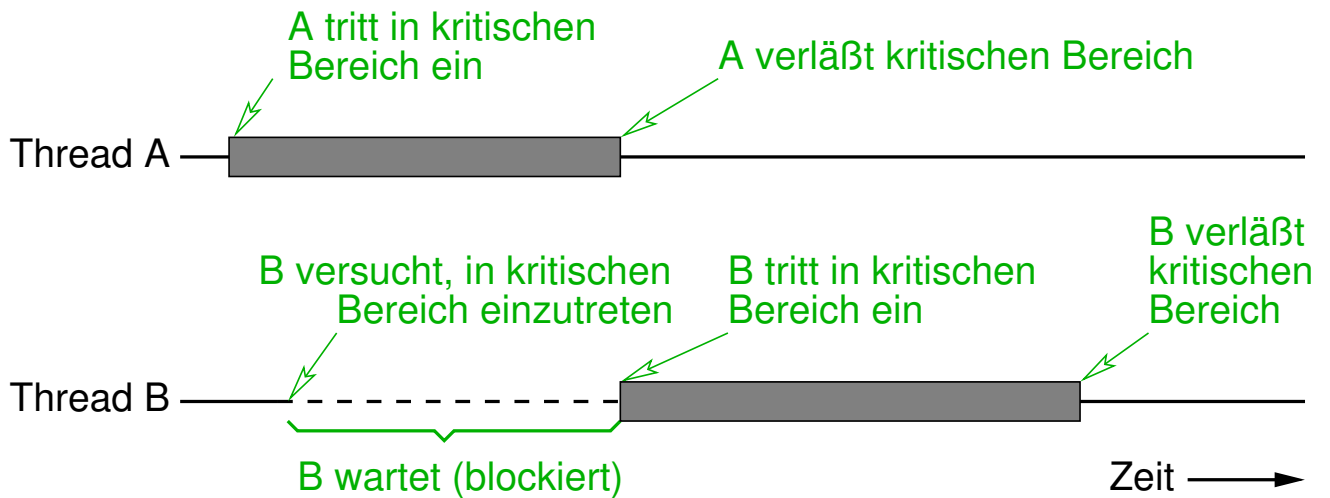
➔ Frage: Implementierung von begin_region() / end_region()?

3.3 Wechselseitiger Ausschluß ...



★

Idee des wechselseitigen Ausschlusses



3.3 Wechselseitiger Ausschluß ...



★

Anforderungen an Lösung zum wechselseitigen Ausschluß:

1. Höchstens ein Thread darf im kritischen Abschnitt (k.A.) sein
2. Keine Annahmen über Geschwindigkeit / Anzahl der CPUs
3. Thread außerhalb des k.A. darf andere nicht behindern
4. Kein Thread sollte ewig warten müssen, bis er in k.A. eintreten darf
 - ➔ Voraussetzung: kein Thread bleibt ewig im k.A.
5. Sofortiger Zugang zum k.A., wenn kein anderer Thread im k.A. ist

Lösungsversuch 1: Sperren der Interrupts

- ➔ Abgesehen von freiwilliger Abgabe der CPU:
Threadwechsel nur durch Interrupt
- ➔ Sperren der Interrupts in `begin_region()`,
Freigabe in `end_region()`
- ➔ Probleme:
 - Ein-/Ausgabe ist blockiert
 - BS verliert Kontrolle über den Thread
 - Funktioniert nur bei Einprozessor-Rechnern
- ➔ Anwendung aber im BS selbst

3.4 Wechsels. Ausschluß mit aktivem Warten ...



(Animierte Folie)

Lösungsversuch 2: Sperrvariable

- ➔ Variable `belegt` zeigt an ob kritischer Abschnitt belegt
- ➔ Thread 0 Thread 1

```
while(belegt); } begin_region()   while(belegt); // warten ...
belegt = true; }                  belegt = true;
// kritischer Abschnitt           // kritischer Abschnitt
belegt = false } end_region()     belegt = false
```
- ➔ Problem: *Race Condition*:
 - Threads führen gleichzeitig `begin_region()` aus
 - lesen gleichzeitig `belegt`
 - finden `belegt` auf `false`
 - setzen `belegt=true` und betreten kritischen Abschnitt!!!



Lösungsversuch 3: Strikter Wechsel

➔ Variable `turn` gibt an, wer an der Reihe ist

➔ Thread 0

```
while (turn != 0);  
// kritischer Abschnitt  
turn = 1;
```

Thread 1

```
while (turn != 1);  
// kritischer Abschnitt  
turn = 0
```

➔ Problem:

- ➔ Threads **müssen** abwechselnd in kritischen Abschnitt
- ➔ verletzt Anforderungen 3, 4, 5



Lösungsversuch 4: Erst belegen, dann prüfen

➔ Variable `interested[i]` zeigt an, ob Thread `i` in den kritischen Abschnitt will

➔ Thread 0

```
interested[0] = true;  
while (interested[1]);  
// kritischer Abschnitt  
interested[0] = false;
```

Thread 1

```
interested[1] = true  
while (interested[0]);  
// kritischer Abschnitt  
interested[1] = false
```

➔ Problem:

- ➔ Verklemmung, falls Threads `interested` gleichzeitig auf `true` setzen

Eine richtige Lösung: Peterson-Algorithmus

```
➔ Thread 0                                Thread 1
interested[0] = true;                       interested[1] = true;
turn = 1;                                    turn = 0;
while ((turn != 0) &&                         while ((turn != 1) &&
        interested[1]);                       interested[0]);
// kritischer Abschnitt                       // kritischer Abschnitt
interested[0] = false;                       interested[1] = false
```

➔ Verklemmung wird durch `turn` verhindert

➔ Jeder Thread bekommt die Chance, den kritischen Bereich zu betreten

➔ keine **Verhungerung**

Anmerkungen zu Folie 157:

Der erste korrekte Algorithmus zum wechselseitigen Ausschluß ist der nach Theodorus Dekker benannte Dekker-Algorithmus (ca. 1962). Der Peterson-Algorithmus wurde 1981 formuliert und ist deutlich einfacher.



Zur Korrektheit des Peterson-Algorithmus

➔ Wechselseitiger Ausschluß:

- ➔ Widerspruchsannahme: T0 und T1 beide im k.A.
- ➔ damit: `interested[0]=true` und `interested[1]=true`
- ➔ falls `turn=1`:
 - ➔ da T0 im k.A.: in der `while`-Schleife muß `turn==0` oder `interested[1]==false` gewesen sein
 - ➔ falls `turn==0` war: Widerspruch! (wer hat `turn=1` gesetzt?)
 - ➔ falls `interested[1]==false` war:
T1 hat `interested[1]=true` noch nicht ausgeführt, hätte also später `turn==0` gesetzt und blockiert, Widerspruch!
- ➔ falls `turn=0`:
 - ➔ symmetrisch!



Zur Korrektheit des Peterson-Algorithmus

➔ Verklemmungs- und Verhungerungsfreiheit:

- ➔ Annahme: T0 dauernd in `while`-Schleife blockiert
- ➔ Damit: immer `turn=1` und `interested[1]=true`
- ➔ Mögliche Fälle für T1:
 - ➔ T1 will nicht in k.A.: `interested[1]` wäre `false`!
 - ➔ T1 wartet in Schleife: geht nicht wegen `turn==1`!
 - ➔ T1 ist immer im k.A.: nicht erlaubt!
 - ➔ T1 kommt immer wieder in k.A.: geht nicht, da T1 `turn=0` setzt, damit kann aber T0 in k.A.!
- ➔ In allen Fällen ergibt sich ein Widerspruch

Anmerkungen zu Folie 159:

Die angegebenen Argumentationen sind nicht wirklich formale Beweise. Als Hilfsmittel zur Spezifikation und formalen Verifikation nebenläufiger Programme können z.B.

- ➔ Petri-Netze und UML Aktivitätsdiagramme (☞ **Modul 4INFBA007 „Softwaretechnik I“**),
- ➔ Temporale Logik (☞ **Modul 4INFMA301 „Model Checking“**)

eingesetzt werden.

159-1

3.4 Wechsels. Ausschluß mit aktivem Warten ...



★★

Lösungen mit Hardware-Unterstützung

- ➔ Problem bei den Lösungsversuchen:
 - ➔ Abfragen und Ändern einer Variable sind zwei Schritte
- ➔ Lösung: **atomare Read-Modify-Write** Operation der CPU
 - ➔ z.B. Maschinenbefehl *Test-and-Set*

```
boolean testAndSet(boolean &var) { // var: Referenzparameter
    boolean tmp = var; var = true; return tmp;
}
```
 - ➔ ununterbrechbar, auch in Multiprozessorsystemen unteilbar
- ➔ Lösung mit *Test-and-Set*:

```
while (testAndSet(belegt));
// kritischer Abschnitt
belegt = false;
```

Aktives Warten (*Busy Waiting*)

- ➔ In bisherigen Lösungen: Warteschleife (*Spinlock*)
- ➔ Probleme:
 - Thread belegt CPU während des Wartens
 - Bei Einprozessorsystem und Threads mit Prioritäten sind Verklemmungen möglich:
 - Thread H hat höhere Priorität wie L, ist aber blockiert
 - L rechnet, wird in kritischem Abschnitt unterbrochen; H wird rechenbereit
 - H will in kritischen Abschnitt, wartet auf L; L kommt nicht zum Zug, solange H rechenbereit ist ...
- ➔ Notwendig bei Multiprozessorsystemen
 - für kurze kritische Abschnitte im BS-Kern

Anmerkungen zu Folie 161:

In Multiprozessorsystemen kann prinzipiell jede CPU (auch gleichzeitig mit anderen) BS-Code ausführen (z.B. bei gleichzeitigen Systemaufrufen oder Interrupts auf mehreren CPUs). Daher müssen Zugriffe auf Datenstrukturen des BSs unter wechselseitigem Ausschluss stehen.

Im einfachsten Fall kann man das so realisieren, daß das ganze BS als ein einziger großer kritischer Abschnitt realisiert wird. Das bedeutet aber, daß immer nur eine CPU tatsächlich BS-Code ausführen kann (die anderen müssen ggf. warten). Daher skaliert diese Lösung nicht gut, d.h. die Systemleistung steigt nicht linear mit der Zahl der CPUs.

Besser ist es daher, feinergranular zu sperren. Im Idealfall muß nur bei Zugriffen auf *dieselbe* BS-Datenstruktur (z.B. die Threadliste) ein wechselseitiger Ausschluss erfolgen. Das Problem dabei ist die dadurch entstehende Deadlock-Gefahr, vgl. Abschnitt 5.

Read-Modify-Write Befehle

- ➔ Sind auf einer CPU atomar, da Unterbrechungen nur zwischen Befehlen auftreten
- ➔ In Mehrprozessorsystemen:
 - Befehl benötigt zwei Speicherzugriffe (Lesen, Schreiben)
 - andere CPU kann dazwischen auf den Speicher zugreifen
- ➔ Daher: Unterstützung durch Speichersystem nötig
 - Bus bzw. Speicher kann über mehrere Zugriffe hinweg gesperrt werden (wechselseitiger Ausschluß!)
- ➔ Wechselseitiger Ausschluß in Mehrprozessorsystemen damit im Endeffekt immer durch Bus- bzw. Speicherarbiter realisiert

3.5 Synchronisation in Mehrprozessorsystemen ...



Spin-Locks und Caches

- ➔ Verbleibendes Problem bei *Spin-Locks*:
 - extreme Belastung des Busses / Speichers
 - trotz Caches!
- ➔ *Test-and-Set* ist eine **schreibende** Operation
 - Cache-Kohärenz-Protokolle führen zur Invalidation aller anderen Kopien bei jedem *Test-and-Set*
- ➔ Während des Wartens wird der betroffene Cache-Block somit laufend invalidiert
 - hohe Bus-Belastung durch Invalidation und Neu-Laden

Anmerkungen zu Folie 163:

In Multiprozessor-Systemen gibt es bei *Spin Locks* das Problem, dass die *Test-and-Set*-Operation eine schreibende Operation ist. Bei ihrer Ausführung muss daher die betroffene Cache-Zeile in den Caches aller anderen CPUs invalidiert werden, um die Cache-Kohärenz herzustellen. Wenn nun mehrere CPUs an einer Sperre warten, bedeutet dies, daß bei jeder Ausführung von *Test-and-Set* die Sperrvariable aus dem Hauptspeicher gelesen werden muß (da die entsprechende Cache-Zeile laufend invalidiert wird). Dadurch entsteht eine extreme Belastung des Hauptspeichers.

Lösungen für dieses Problem sind u.a.:

- ➔ *Test-and-Test-and-Set*: die Sperrvariable wird zunächst über einen normalen Lesezugriff abgefragt. Nur wenn sie frei ist, erfolgt ein *Test-and-Set*.
- ➔ *Exponential Backoff*: Einführen einer Wartezeit zwischen zwei Sperrversuchen.
- ➔ Listenbasierte Sperren. Sie verhindern das Problem vollständig.

Genauerer dazu findet sich im Tanenbaum-Buch, Kap. 8.1.3.

163-1

3.5 Synchronisation in Mehrprozessorsystemen ...



Test and Test-and-Set

- ➔ Idee: einfache Abfrage des Locks vor dem Sperrversuch mit *Test-and-Set*

```
while (belegt || (testAndSet (belegt)));  
// kritischer Abschnitt  
belegt = false;
```

- ➔ Während des Wartens wird `belegt` nur gelesen (aus dem Cache)
 - ➔ Verbesserung gegenüber einfacher Sperre
- ➔ Bei Freigabe: Invalidierung der Caches
 - ➔ mehrere Threads können `belegt == false` sehen und versuchen, mit `testAndSet()` die Sperre zu bekommen
 - ➔ dadurch viele weitere (überflüssige) Invalidierungen



Exponential Backoff

- ➔ Weitere Möglichkeit: zeitliche Entzerrung der Sperrversuche
 - Warteschleife zwischen zwei Abfragen der Sperre
 - falls Sperre belegt: Wartezeit verdoppeln (bis zu einem Maximum)
- ➔ Ggf. kombiniert mit *Test and Test-and-Set*
- ➔ Busbelastung wird (auch bei Freigabe) geringer
- ➔ Aber: erhöhte Reaktionszeit bei Freigabe der Sperre



Load Linked / Store Conditional

- ➔ Spezielle Maschinenbefehle zur Realisierung von *Spin Locks*
- ➔ *Load Linked*
 - Laden aus dem Speicher (bzw. Cache)
 - Adresse wird in *Link Register* vermerkt
 - *Link Register* wird bei Invalidierung der Cachezeile gelöscht
- ➔ *Store Conditional*
 - speichert einen Wert, falls die Adresse mit der im *Link Register* übereinstimmt
- ➔ Maximal eine Invalidierung bei freiwerdender Sperre

Code:

```
li r2,#1
wt: ll r1,locked
bnz wt
sc locked,r2
bz wt

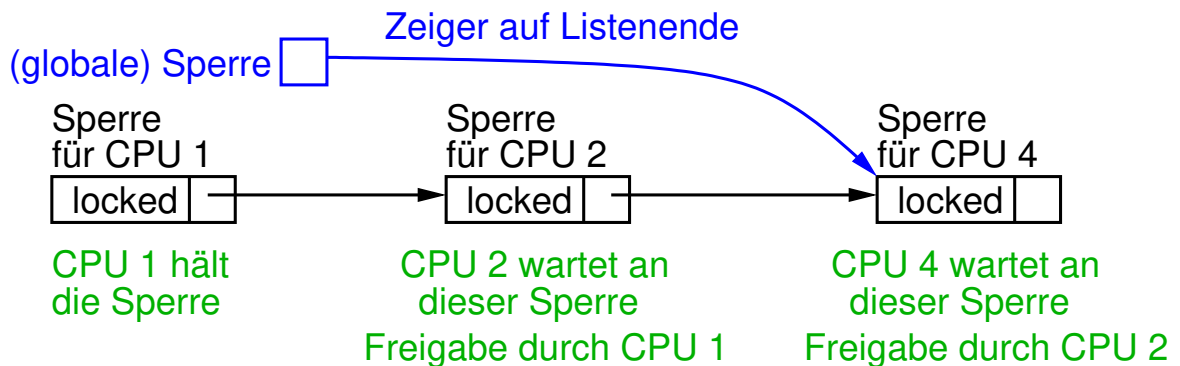
... ;k.A.

st locked,#0
```



Listenbasierte Sperren (Mellor-Crummey / Scott)

➔ Idee: Sperre als Liste mit lokalen Sperr-Variablen realisiert



➔ Falls gesperrt: CPU fügt Listenelement mit lokaler Sperre an

➔ jede CPU wartet nur an ihrer lokalen Sperre

➔ Belegen / Freigeben des Locks mit $\mathcal{O}(1)$ Speichertransaktionen

➔ Garantiert FIFO-Reihenfolge

3.5 Synchronisation in Mehrprozessorsystemen ...



```

Node l; // Globale Variable: Listenende
void lock(Node n) { // n = eigener Knoten
    n.next = null;
    Node pred = fetchAndStore(l, n); // pred = l; l = n, Einfügen (Teil 1)
    if (pred != null) { // falls Sperre belegt (Liste nicht leer)
        n.locked = true;
        pred.next = n; // Einfügen in Liste (Teil 2)
        while (n.locked); // Warte auf Freigabe
    }
}
void unlock(Node n) { // n = eigener Knoten
    if (n.next == null) { // kein Nachfolger?
        if (compareAndSet(l, n, null)) // falls noch l == n ist: l = null
            return; // und fertig
        while (n.next == null); // sonst: warten bis Nachfolger
    } // eingetragen wurde
    n.next.locked = false; // Freigabe an Nachfolger
}
    
```



Aktives Warten oder Blockierung des Threads?

- ➔ Aktives Warten in Multiprozessorsystemen möglich / sinnvoll
 - in Einprozessorsystemen extrem schlechte Leistung bzw. Verklemmung möglich
- ➔ In einigen Fällen ist aktives Warten unvermeidlich
 - innerhalb des Betriebssystems
- ➔ Threadwechsel bedeutet immer zusätzlichen Overhead
 - Systemaufruf, Umladen der CPU-Register, Caches, ...
- ➔ Optimale Entscheidung daher abhängig von mittlerer Wartezeit
- ➔ Praxis: warte einige Zeit aktiv, dann Blockierung des Threads

3.6 Speicherkonsistenz



Reihenfolgesynchronisation über eine gemeinsame Variable ★

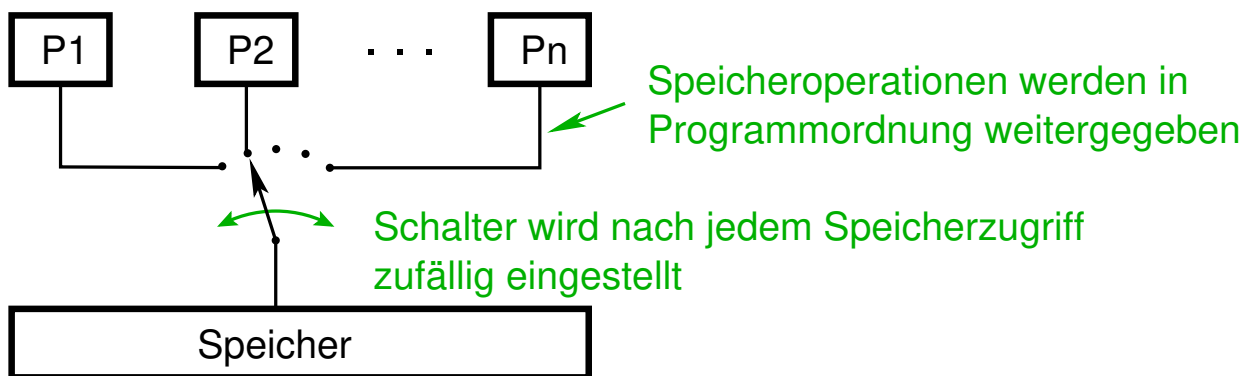
- ➔ Typisches Beispiel:
Initialisierung:

```
double value = 0;  
boolean ready = false; // ist 'value' gültig?
```

Thread 0	Thread 1
<pre>value = 0.5*2; ready = true;</pre>	<pre>while (!ready); // aktives Warten use(value);</pre>
- ➔ Annahme dabei: Thread 1 sieht die Ergebnisse der Schreiboperationen in der Reihenfolge, in der Thread 0 sie ausführt
- ➔ Diese Annahme wird durch heutige Prozessoren verletzt!
 - z.B. durch *out-of-order execution* und Schreibpuffer
- ➔ Problem der **Speicherkonsistenz**

Konsistenzmodelle

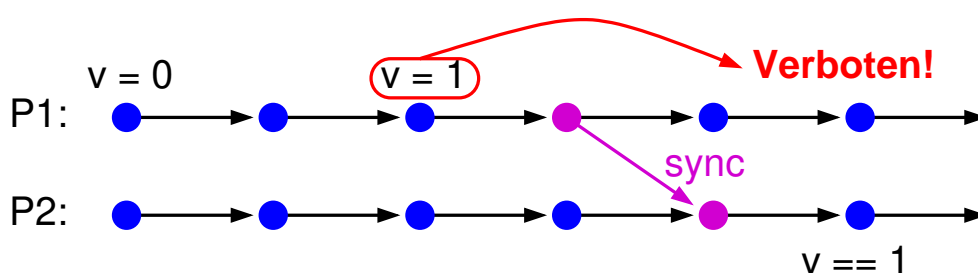
- ➔ Legen fest, in welcher Reihenfolge Schreibzugriffe auf den Speicher „gesehen“ werden können
 - ➔ d.h., welche Werte eine Leseoperation zurückgeben darf
- ➔ **Sequentielle Konsistenz:** Ergebnis jeder Programmausführung ist durch folgendes Modell erklärbar:



(Animierte Folie)

Konsistenzmodelle ...

- ➔ Sequentielle Konsistenz bedeutet: alle Prozessoren sehen alle Speicheroperationen in derselben Reihenfolge
 - ➔ Operationen eines Prozessors dürfen nicht vertauscht werden
- ➔ **Abgeschwächte Konsistenzmodelle** erlauben Vertauschung von Speicheroperationen, außer in speziellen Fällen
 - ➔ typisch: Vertauschung über explizite Synchronisationsoperationen hinweg ist nicht erlaubt





Das Java Speichermodell

- ➔ Herausforderung: Modell muss prozessorunabhängig sein!
- ➔ Bei Betrachtung nur eines Threads: *as-if-serial* Semantik
 - die eigenen Zugriffe erscheinen wie in Programmordnung ausgeführt
 - entspricht dem Verhalten heutiger CPUs
- ➔ Bei Betrachtung mehrerer Threads:
 - wenn Operation *a* aufgrund von **Programmordnung** und/oder expliziter **Synchronisation** vor *b* aufgeführt werden **muß**, dann sieht *b* die Effekte von *a*
 - explizite Synchronisation: `start()` / `join()` sowie von Java bereitgestellte Konstrukte (siehe später)



Das Java Speichermodell: Schlüsselwort `volatile`

- ➔ Attribute können durch das Schlüsselwort `volatile` als Synchronisationsvariable deklariert werden
- ➔ Schreiben und Lesen einer Synchronisationsvariable wird dann wie explizite Synchronisation behandelt
- ➔ Damit Beispiel korrekt implementierbar:

Initialisierung:

```
double value = 0;
```

```
volatile boolean ready = false; // ist 'value' gültig?
```

Thread 0

```
value = 0.5*2;
```

```
ready = true;
```

Thread 1

```
while (!ready); // aktives Warten
```

```
use(value);
```

Anmerkungen zu Folie 174:

Weitere Informationen zum Java Speichermodell finden Sie z.B. in:

- ➔ <http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>
- ➔ https://download.oracle.com/otndocs/jcp/memory_model-1.0-pfd-spec-oth-JSpec

174-1

3.6 Speicherkonsistenz ...



Das C++ Speichermodell

- ➔ Ähnlich zum Java-Speichermodell, aber mit mehr Möglichkeiten
- ➔ Basis: Datentyp `std::atomic<T>`, z.B. `std::atomic<int>`
 - ➔ Zugriff nur über explizite `load()` und `store()` Methoden
 - ➔ werden garantiert atomar ausgeführt
- ➔ Konsistenzanforderungen können über Argument von `load()` bzw. `store()` festgelegt werden
 - ➔ Standardeinstellung: sequentielle Konsistenz
- ➔ Nachbildung des Java-Speichermodells:
 - ➔ Synchronisationsvariablen als `std::atomic<T>` deklarieren
 - ➔ Lesen mit `load(std::memory_order_acquire)`
 - ➔ Schreiben mit `store(std::memory_order_release)`

Anmerkungen zu Folie 175:

- ➔ Das Lesen bzw. Schreiben eines Werts ist nicht immer automatisch atomar. Z.B. werden auf heutigen Prozessoren zum Lesen einer 128-Bit Werts zwei Speicherzugriffe benötigt. Das bedeutet: eine andere CPU könnte dazwischen den Wert verändern!
- ➔ Der Wert `std::memory_order_relaxed` als Argument von `load()` bzw. `store()` definiert keinerlei Einschränkungen bezüglich Konsistenz, d.h. es wird lediglich zugesichert, daß das Lesen und Schreiben atomar erfolgt.
- ➔ Eine detaillierte Diskussion des Speichermodells finden Sie z.B. hier:
https://en.cppreference.com/w/cpp/atomic/memory_order
- ➔ Es gibt im C++ Standard auch ein Schlüsselwort `volatile`, das jedoch eine völlig andere Bedeutung hat als in Java. In C++ sagt `volatile` lediglich aus, dass der Compiler Lese- und Schreibzugriffe auf die Variable nicht optimieren darf. Siehe dazu z.B.
 - <http://www.drdoobs.com/parallel/volatile-vs-volatile/212701484>
 - <https://en.cppreference.com/w/cpp/language/cv>Der Microsoft Visual Studio Compiler implementiert `volatile` allerdings so, daß es auch der Semantik in Java entspricht.

175-1

3.6 Speicherkonsistenz ...



Das C++ Speichermodell ...

- ➔ Beispiel als C++ Code:

Initialisierung:

```
double value = 0;  
std::atomic<bool> ready = false; // ist 'value' gültig?
```

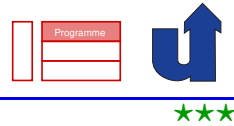
Thread 0

```
value = 0.5*2;  
ready.store(true, std::memory_order_release);
```

Thread 1

```
// aktives Warten  
while (!ready.load(std::memory_order_acquire));  
use(value);
```

3.7 Semaphore



- ➔ Eingeführt durch Edsger Wybe Dijkstra (1965)
- ➔ Allgemeines Synchronisationskonstrukt
 - nicht nur wechselseitiger Ausschluß, auch Reihenfolge-synchronisation
- ➔ Semaphor ist i.W. eine ganzzahlige Variable
 - Wert kann auf nichtnegative Zahl initialisiert werden
 - zwei **atomare** Operationen:
 - P() (auch wait, down oder acquire)
 - verringert Wert um 1
 - falls Wert < 0: Thread blockieren
 - V() (auch signal, up oder release)
 - erhöht Wert um 1
 - falls Wert ≤ 0: einen blockierten Thread wecken

3.7 Semaphore ...



Semaphor-Operationen

```
struct Semaphor {
    int count;           // Semaphor-Zähler
    ThreadQueue queue; // Warteschlange für blockierte Threads
}

void P(Semaphor &s) {
    s.count--;
    if (s.count < 0) {
        Thread in s.queue
        ablegen;
        Thread blockieren;
    }
}

void V(Semaphor &s) {
    s.count++;
    if (s.count <= 0) {
        Einen Thread T aus s.queue
        entfernen;
        T auf bereit setzen;
    }
}
```

- ➔ Hinweis: Tanenbaum definiert Semaphore etwas anders (Zähler zählt höchstens bis 0 herunter)

Interpretation des Semaphor-Zählers

- ➔ Zähler ≥ 0 : Anzahl freier Ressourcen
- ➔ Zähler < 0 : Anzahl wartender Threads

Wechselseitiger Ausschluß mit Semaphoren

- | | |
|--------------------------------------|--------------------------------------|
| ➔ Thread 0 | Thread 1 |
| <code>P(mutex);</code> | <code>P(mutex);</code> |
| <code>// kritischer Abschnitt</code> | <code>// kritischer Abschnitt</code> |
| <code>V(mutex);</code> | <code>V(mutex);</code> |

- ➔ Semaphore `mutex` wird mit 1 vorbelegt
- ➔ Semaphore, das an positiven Werten nur 0 oder 1 haben kann, heißt **binäres Semaphore**

Reihenfolgesynchronisation mit Semaphoren

- ➔ Beispiel: Thread 1 darf Datei erst öffnen, nachdem Thread 0 sie erzeugt hat
 - ➔ Thread 0
 - ➔ Thread 1
- | | |
|--------------------------------|------------------------------|
| <code>// Datei erzeugen</code> | <code>P(sema);</code> |
| <code>V(sema);</code> | <code>// Datei öffnen</code> |
- ➔ Semaphore `sema` wird mit 0 vorbelegt
 - ➔ damit: Thread 1 wird blockiert, bis Thread 0 die `V()`-Operation ausgeführt hat
 - ➔ Merkregel:
 - ➔ `P()`-Operation an der Stelle, wo gewartet werden muß
 - ➔ `V()`-Operation signalisiert, daß Wartebedingung erfüllt ist
 - ➔ vgl. die alternativen Namen `wait()` / `signal()` für `P()` / `V()`

Realisierung von Semaphoren

- ➔ Eng verbunden mit Thread-Implementierung
- ➔ Bei Kernel-Threads:
 - Implementierung im BS-Kern
 - Operationen sind Systemaufrufe
 - atomare Ausführung durch Interrupt-Sperre und Spinlocks gesichert

3.8 Klassische Synchronisationsprobleme



Das Erzeuger/Verbraucher-Problem

- ➔ Situation:
 - Zwei Thread-Typen: Erzeuger, Verbraucher
 - Kommunikation über einen gemeinsamen, beschränkten Puffer der Länge N
 - Operationen `insertItem()`, `removeItem()`
 - Erzeuger legen Elemente in Puffer, Verbraucher entfernen sie
- ➔ Synchronisation:
 - Sperrsynchrisation: wechselseitiger Ausschluß
 - Reihenfolgesynchronisation:
 - kein Entfernen aus leerem Puffer: Verbraucher muß warten
 - kein Einfügen in vollen Puffer: Erzeuger muß warten

3.8 Klassische Synchronisationsprobleme ...



(Animierte Folie)

★★★

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;  
Semaphor full = 0;  
Semaphor empty = N;
```

für wechselseitigen Ausschluß
verhindert Entfernen aus leerem Puffer
verhindert Einfügen in vollen Puffer

Erzeuger

```
while (true) {  
    item = produce();  
    P(empty);  
    P(mutex);  
    insertItem(item);  
    V(mutex);  
    V(full);  
}
```

Verbraucher

```
while (true) {  
    P(full);  
    P(mutex);  
    item = removeItem();  
    V(mutex);  
    V(empty);  
    consume(item);  
}
```

3.8 Klassische Synchronisationsprobleme ...



★

Das Leser/Schreiber-Problem

- ➔ Gemeinsamer Datenbereich mehrerer Threads
- ➔ Zwei Klassen von Threads (bzw. Zugriffen)
 - ➔ Leser (*Reader*)
 - ➔ dürfen gleichzeitig mit anderen Lesern zugreifen
 - ➔ Schreiber (*Writer*)
 - ➔ stehen unter wechselseitigem Ausschluß, auch mit Lesern
 - ➔ verhindert Lesen von inkonsistenten Daten
- ➔ Typisches Problem in Datenbank-Systemen

3.8 Klassische Synchronisationsprobleme ...



(Animierte Folie)

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {  
    P(mutex);  
    rc++;  
    if (rc == 1)  
        P(db);  
    V(mutex);  
    readDataBase();  
    P(mutex);  
    rc--;  
    if (rc == 0)  
        V(db);  
    V(mutex);  
    useData();  
}
```

Semaphore und gemeinsame Variable

```
int rc=0; // Anzahl Leser  
Semaphor db=1; // Schützt Datenbank  
Semaphor mutex=1;
```

Schreiber

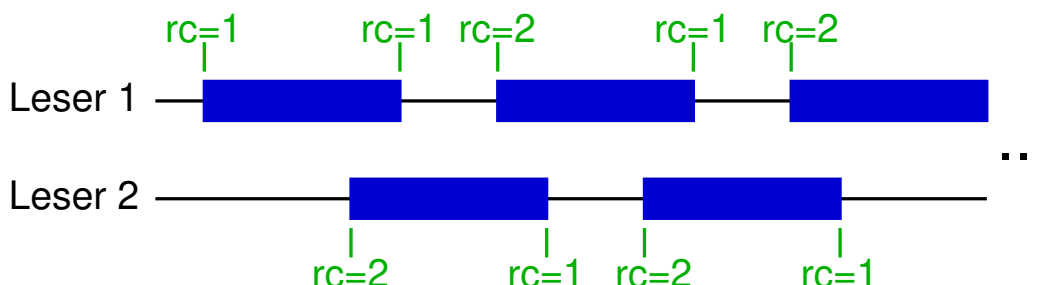
```
while(true) {  
    createData();  
    P(db);  
    writeDataBase();  
    V(db);  
}
```

3.8 Klassische Synchronisationsprobleme ...



Eigenschaft der skizzierten Lösung

- ➔ Die Synchronisation ist unfair: Leser haben Priorität vor Schreibern
- ➔ Schreiber kann verhungern



- ➔ Mögliche Lösung:
- ➔ neue Leser blockieren, wenn ein Schreiber wartet

Motivation

- ➔ Programmierung mit Semaphoren ist schwierig
 - Reihenfolge der P/V-Operationen: Verklemmungsgefahr
 - Synchronisation über gesamtes Programm verteilt

Monitor (Hoare, 1974; Brinch Hansen 1975)

- ➔ Modul mit Daten, Prozeduren und Initialisierungscode
 - Zugriff auf die Daten nur über Monitor-Prozeduren
 - (entspricht in etwa einer Klasse)
- ➔ Alle Prozeduren stehen unter wechselseitigem Ausschluß
 - nur jeweils ein Thread kann Monitor benutzen
- ➔ Programmiersprachkonstrukt: Realisierung durch Übersetzer

3.9 Monitore ...



Bedingungsvariable (Zustandsvariable, *condition variables*)

- ➔ Zur Reihenfolgesynchronisation zwischen Monitor-Prozeduren
- ➔ Darstellung anwendungsspezifischer Bedingungen
 - z.B. voller Puffer im Erzeuger/Verbraucher-Problem
- ➔ Zwei Operationen:
 - `wait()`: Blockieren des aufrufenden Threads
 - `signal()`: Aufwecken blockierter Threads
- ➔ Bedingungsvariable verwaltet Warteschlange blockierter Threads
- ➔ Bedingungsvariable hat kein „Gedächtnis“:
 - `signal()` weckt nur einen Thread, der `wait()` bereits aufgerufen hat

Funktion von `wait()`:

- ➔ Aufrufender Thread wird blockiert
 - ➔ nach Ende der Blockierung kehrt `wait()` zurück
- ➔ Aufrufender Thread wird in die Warteschlange der Bedingungsvariable eingetragen
- ➔ Monitor steht bis zum Ende der Blockierung anderen Threads zur Verfügung

Funktion von `signal()`:

- ➔ Falls Warteschlange der Bedingungsvariable nicht leer:
 - ➔ mindestens einen Thread wecken:
aus Warteschlange entfernen und Blockierung aufheben

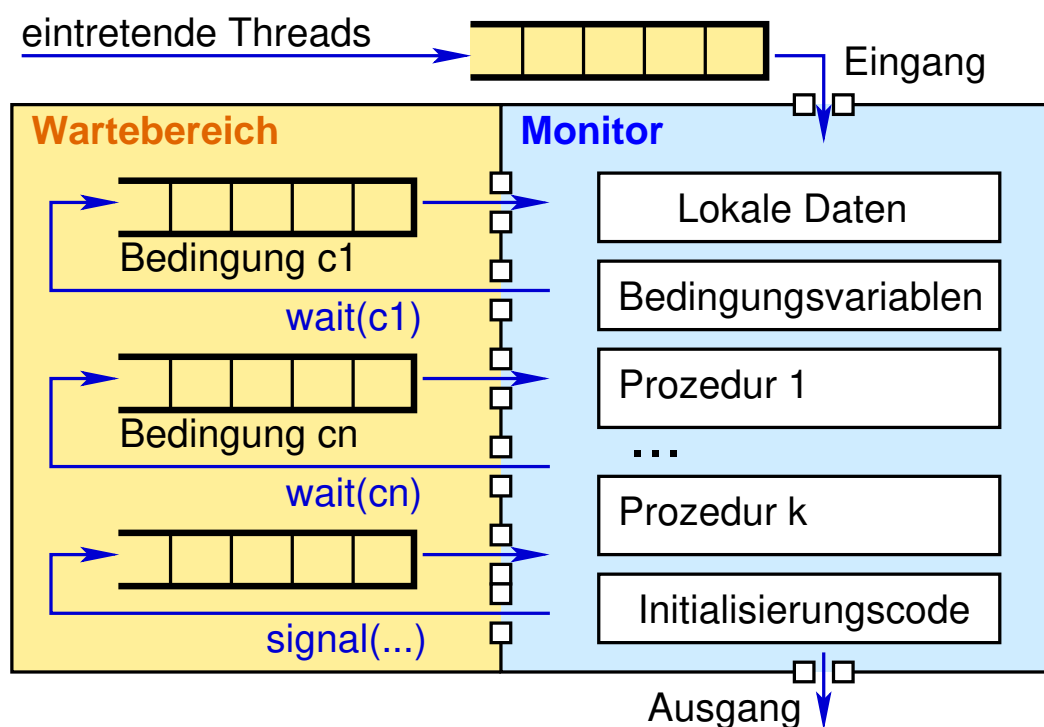
Varianten für `signal()`:

1. Ein Thread wird geweckt (meist der am längsten wartende)
 - a) signalisierender Thread bleibt im Besitz des Monitors
 - b) geweckter Thread erhält den Monitor sofort
 - i. signalisierender Thread muß sich erneut bewerben (Hoare)
 - ii. `signal()` muß letzte Anweisung in Monitorprozedur sein (Brinch Hansen)
 2. Alle Threads werden geweckt
 - ➔ signalisierender Thread bleibt im Besitz des Monitors
- ➔ Bei 1a) und 2) ist nach Rückkehr aus `wait()` **nicht** sicher, daß die Bedingung (noch) erfüllt ist!

Typische Verwendung von `wait()` und `signal()`

- ➔ Testen einer Bedingung
 - ➔ bei Variante 1b):
 - ➔ `if (!Bedingung) wait(condVar);`
 - ➔ bei Varianten 1a) und 2):
 - ➔ `while (!Bedingung) wait(condVar);`
- ➔ Signalisieren der Bedingung
 - ➔ `[if (Bedingung)] signal(condVar);`

Aufbau eines Monitors nach Hoare



Anmerkungen zu Folie 192:

Bei einem Monitor nach Brinch Hansen oder der Signalisierungsvariante 1a) bzw. 2) entfällt in dem Bild die Warteschlange für `signal()`, da der signalisierende Thread im Besitz des Monitors bleibt (bzw. bei Brinch Hansen diesen sofort nach dem Signalisieren verlässt).

192-1

3.9 Monitore ...



Semaphor-Realisierung m. Monitor (Pseudo-Code, Brinch Hansen) ^{★★}

```
monitor Semaphore {  
    condition nonbusy;  
    int count;  
  
    void P() {  
        if (count == 0)  
            wait(nonbusy);  
        count = count - 1;  
    }  
  
    void V() {  
        count = count + 1;  
        signal(nonbusy);  
    }  
    count = 1;  
}
```

- ➔ Anmerkung: die Implementierung weicht von der Definition auf Folie 178 ab
- ➔ Umgekehrt können auch Monitore (insbes. Bedingungsvariable) mit Semaphoren nachgebildet werden

Anmerkungen zu Folie 193:

Die direkte Umsetzung der Definition von Folie 178 würde zu folgender Implementierung der beiden Monitor-Funktionen führen:

```
void P() {
    count = count - 1;
    if (count < 0)
        wait(nonbusy);
}

void V() {
    count = count + 1;
    if (count <= 0)
        signal(nonbusy);
}
```

Diese Realisierung wäre mit der Signalisierungsvariante **1b**) auch korrekt, nicht aber mit der Signalisierungsvariante **1a**)! Der Grund ist letztlich, dass der Test der Bedingung mit `while` natürlich voraussetzt, dass die Bedingung auch die korrekte Wartebedingung ist. In der obigen Variante ist `!(count < 0)` jedoch nicht die Bedingung, auf die in `P()` gewartet werden muß.

Wenn z.B. zwei Threads warten, ist `count` gleich -2. Nach einer `V()` Operation muss aber einer der in `P()` wartenden Threads aufgeweckt werden. Das würde aber nicht passieren, wenn das `if` durch ein `while` ersetzt wird, da `count` immer noch -1 wäre.

Im modifizierten Code wird `count` nie negativ, zählt also nur die freien Ressourcen. Daher ist hier `!(count == 0)` die korrekte Wartebedingung, die dann auch mit einem `while` funktioniert.

193-1

3.9 Monitore ...



Erzeuger/Verbraucher m. Monitor (Pseudo-Code, Brinch Hansen)

```
monitor ErzeugerVerbraucher {
    condition nonfull, nonempty;
    int count;

    void insert(int item) {
        if (count == N)
            wait(nonfull);
        insertItem(item);
        count = count + 1;
        signal(nonempty);
    }

    int remove() {
        int res;
        if (count == 0)
            wait(nonempty);
        res = remove(item);
        count = count - 1;
        signal(nonfull);
        return res;
    }

    count = 0;
}
```

Anmerkungen zu Folie 194:

Bei Verwendung einer Monitors nach Brinch Hansen oder nach Hoare kann man die Signalisierung optimieren, indem man die **signal()**-Aufrufe nur dann ausführt, wenn die Bedingung gerade hergestellt wurde, also:

```
void insert(int item) {
    ...
    if (count == 1)
        signal(nonempty);
}

int remove() {
    ...
    if (count == N-1)
        signal(nonfull);
    return res;
}
```

Mit der Signalisierungsvariante 1a, bei der der aufgeweckte Thread nicht sofort den Monitor betreten kann, ist diese Optimierung aber nicht korrekt, wenn es mehr als einen Erzeuger oder mehr als einen Verbraucher gibt! Wenn z.B. beide Verbraucher warten, weil der Puffer leer ist, wird beim Eintragen in den Puffer zunächst nur einmal ein **signal(nonempty)** ausgeführt, also auch nur ein Verbraucher geweckt. Bevor dieser den Puffer wieder leert, kann aber ein anderer Erzeuger ein Element in den Puffer legen, so daß es passieren kann, daß der Puffer danach nie wieder leer wird. In diesem Fall wird der zweite Verbraucher dann aber nie mehr aufgeweckt!

194-1

3.9 Monitore ...



Motivation für Broadcast-Signalisierung (Variante 2)

- ➔ Aufwecken aller Threads sinnvoll, wenn unterschiedliche Wartebedingungen vorliegen
- ➔ Beispiel: Erzeuger/Verbraucher-Problem mit variablem Bedarf an Puffereinträgen

```
void insert(int item, int size) {
    while (count + size > N)
        wait(nonfull);
    ...
}
```

- ➔ Nachteil: viele Threads konkurrieren um Wiedereintritt in den Monitor

3.10.1 Standard-Synchronisations-Mechanismen in Java

★

- ➔ Sind bereits in der Programmiersprache definiert
- ➔ Monitor-ähnlich, aber Klassen statt Module
- ➔ Synchronisierte Methoden
 - müssen explizit als `synchronized` deklariert werden
 - stehen (pro Objekt!) unter wechselseitigem Ausschluß
- ➔ Keine expliziten Bedingungsvariablen, stattdessen genau eine implizite Bedingungsvariable pro Objekt
 - Basisklasse `Object` definiert Methoden `wait()`, `notify()` und `notifyAll()`
 - diese Methoden werden von allen Klassen geerbt
 - `notify()`: Signalisierungsvariante 1a)
 - `notifyAll()`: Signalisierungsvariante 2)

3.10.1 Standard-Synchronisation in Java ...



★

Beispiel: Erzeuger/Verbraucher-Problem

```
public class ErzVerb {  
    ...  
    public synchronized void insert(int item) {  
        while (count == buffer.getSize()) { // Puffer voll?  
            try {  
                wait(); // ja: warten ...  
            }  
            catch (InterruptedException e) {}  
        }  
        buffer.insertItem(item); // Item eintragen  
        count++;  
        notifyAll(); // alle wecken  
    }  
}
```

Anmerkungen zu Folie 197:

Warum der try-catch-Block für die `InterruptedException` benötigt wird, wurde bereits in der Anmerkung zu Folie 124 erläutert.

197-1

3.10.1 Standard-Synchronisation in Java ...



Beispiel: Erzeuger/Verbraucher-Problem ...

```
public synchronized int remove() {
    int result;
    while (count == 0) {                // Puffer leer?
        try {
            wait();                      // ~~~ja: warten ...
        }
        catch (InterruptedException e) {}
    }
    result = buffer.removeItem();        // Item entfernen
    count--;
    notifyAll();                        // alle wecken
    return result;
}
}
```



Anmerkungen zum Beispiel

- ➔ Vollständiger Code ist im WWW (Vorlesungsseite) verfügbar
- ➔ Bedingungen müssen immer in `while`-Schleife geprüft werden
- ➔ `notify()` statt `notifyAll()` ist **nicht** korrekt!
 - ➔ funktioniert nur bei genau einem Erzeuger und genau einem Verbraucher
 - ➔ da nur eine Bedingungsvariable für zwei verschiedene Bedingungen benutzt wird, kann es sein, daß der falsche Thread aufgeweckt wird
 - ➔ Übungsaufgabe:
 - ➔ mit Programm aus WWW ausprobieren!
 - ➔ mit Simulator (siehe Webseite) nachvollziehen!



`synchronized` Blöcke

- ➔ Java unterstützt auch wechselseitigen Ausschluß beliebiger Code-Blöcke über das eingebaute Mutex eines Objekts
 - ➔ Syntax: `synchronized(Objekt) { Block }`
- ➔ Beispiel: gegeben ist die Klasse

```
public class Queue {  
    public synchronized void enqueue(Object data) { ... }  
    ...  
}
```
- ➔ atomares Einfügen von zwei Elementen:

```
synchronized(queue) { // Sperrt das Objekt 'queue'  
    queue.enqueue(elem1);  
    queue.enqueue(elem2);  
}
```

Anmerkungen zu Folie 200:

Das gezeigte Beispiel funktioniert, da das eingebaute Mutex eines Java-Objekts ein *reentrant lock* ist. Das bedeutet, dass das Mutex vom selben Thread mehrfach gesperrt werden kann, ohne dass der Thread deshalb blockiert wird. Das Mutex hat dazu einen Zähler, der dafür sorgt, dass der Thread das Mutex dann auch entsprechend oft wieder freigeben muss, bis es wirklich frei ist.

Diese Semantik ist auch deshalb notwendig, da ein `synchronized`-Methode eines Objekts ja auch eine andere `synchronized`-Methode desselben Objekts aufrufen können muss.

200-1

3.10 Schnittstellen zur Thread-Synchronisation ...



3.10.2 Java Klassenbibliothek: Synchronisationsklassen

- ➔ Java-Paket `java.util.concurrent.lock`
- ➔ Klasse `Semaphore`
- ➔ Schnittstellen zur Implementierung des Monitor-Konzepts:
 - *Mutual Exclusion Locks* (Mutex): Schnittstelle `Lock`
 - Verhalten wie binäres Semaphor
 - Zustände: gesperrt, frei
 - Bedingungsvariable: Schnittstelle `Condition`
 - fest an ein `Lock` gebunden
 - `Lock` wird für wechselseitigen Ausschluß der Monitor-Prozeduren genutzt
- ➔ Schnittstelle `ReadWriteLock` (Leser-Schreiber-Sperren)



Klasse Semaphore

- ➔ Konstruktor: `Semaphore(int wert)`
 - ➔ erzeugt Semaphor mit angegebenem Initialwert
- ➔ Wichtigste Methoden:
 - ➔ `void acquire()`
 - ➔ entspricht P-Operation
 - ➔ `void release()`
 - ➔ entspricht V-Operation



Schnittstelle Lock

★★★

- ➔ Wichtigste Methoden:
 - ➔ `void lock()`
 - ➔ sperren (entspricht P bei binärem Semaphor)
 - ➔ `void unlock()`
 - ➔ freigeben (entspricht V bei binärem Semaphor)
 - ➔ `Condition newCondition()`
 - ➔ erzeugt neue Bedingungsvariable, die an dieses Lock-Objekt gebunden ist
 - ➔ beim Warten an der Bedingungsvariable wird dieses Lock freigegeben
- ➔ Implementierungsklasse: `ReentrantLock`
 - ➔ neu erzeugte Sperre ist zunächst frei

Schnittstelle `Condition`

➔ Wichtigste Methoden:

- ➔ `void await()`
 - ➔ *wait*-Operation: warten auf Signalisierung
 - ➔ Thread wird blockiert, zur `Condition` gehöriges Lock wird freigegeben
 - ➔ nach Signalisierung: `await` kehrt erst zurück, wenn Lock wieder erfolgreich gesperrt ist
- ➔ `void signal()`
 - ➔ Signalisierung eines wartenden Threads (Variante 1a)
- ➔ `void signalAll()`
 - ➔ Signalisierung aller wartenden Threads (Variante 2)

Anmerkungen zu Folie 204:

Beachten Sie, daß die Methode `await()` genau wie die Methode `join()` der Klasse `Thread` und die Methode `wait()` eine `InterruptedException` werfen kann, die entweder gefangen oder weitergegeben (und damit in der Methodendeklaration entsprechend angegeben) werden muß, vgl. Folie 124.

Anmerkungen

➔ Mit diesen Objekten lassen sich Monitore nachbilden:

<p>Monitor</p> <pre>monitor Bsp { condition cond; void foo() { if (...) wait(cond); ... signal(cond); } }</pre>		<p>Java-Code</p> <pre>public class Bsp { Lock mutex; // = new ReentrantLock(); Condition cond; // = mutex.newCondition(); public void foo() { mutex.lock(); while(...) cond.await(); ... cond.signal(); mutex.unlock(); } }</pre> <p style="color: green; text-align: center;">Im Konstruktor</p>
--	--	--

➔ Ähnliche Konzepte wie Lock und Condition auch in anderen Thread-Schnittstellen

(Animierte Folie)

Anmerkungen ...

- ➔ Herstellen der signalisierten Bedingung und Signalisierung muß bei gesperrtem Mutex erfolgen!
- ➔ Sonst: Gefahr des *lost wakeup*

<p>Thread 1</p> <pre>condition = true; cond.signal();</pre>		<p>Thread 2</p> <pre>mutex.lock(); ... while (!condition) cond.await(); ... mutex.unlock();</pre>
---	--	---

Anmerkungen ...

- ➔ Freigabe der Sperre sollte über `try - finally` erfolgen
 - ➔ sonst bleibt `mutex` ggf. gesperrt, wenn die Methode `return` ausführt oder eine Exception wirft

- ➔ Beispiel:

```
public void foo() {
    mutex.lock();
    try {
        ... // Code der Methode
    }
    finally { mutex.unlock(); }
}
```

- ➔ `finally`-Block wird **immer** ausgeführt, nachdem `try`-Block verlassen wird

- ➔ Alternative ab Java 7: *try-with-resources*

Anmerkungen zu Folie 207:

Das Konstrukt *try-with-resources* in Java entspricht der `with`-Anweisung einiger anderer Sprachen wie z.B. Python. Die Idee ist, daß man eine Resource (z.B. Datei) beim Eintritt in den `try`-Block öffnet und diese beim Verlassen des Blocks automatisch wieder geschlossen wird. Siehe dazu z.B.:

docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Zum Beispiel wird im folgenden Code ein `FileReader` für eine Datei erzeugt, der bei Verlassen des Blocks automatisch wieder geschlossen wird.

```
try (FileReader fr = new FileReader(path)) {
    ...
}
```

Im Wesentlichen ist das eine syntaktisch einfachere Schreibweise für

```
FileReader fr = new FileReader(path);
try {
    ...
} finally {
    fr.close();
}
```

Leider besitzen Locks keine `close()` Methode, so daß zur Verwendung von *try-with-resources* etwas mehr Aufwand erforderlich ist, siehe z.B. die Diskussion in

<https://stackoverflow.com/questions/6965731/are-locks-autocloseable>



Synchronisationspaket `BSsync` für die Übungen

- ➔ Für die Übungen verwenden wir eine vereinfachte Version der `java.util.concurrent.lock`-Klassen
 - ➔ weniger Methoden (nur die wichtigsten, siehe vorherige Folien)
 - ➔ keine `InterruptedException` bei `await()`
 - ➔ Optionen zur besseren Fehlersuche
 - ➔ `Lock` direkt als Klasse implementiert
 - ➔ d.h. `new Lock()` statt `new ReentrantLock()`
- ➔ JAR-Archiv `BSsync.jar` und API-Dokumentation im WWW verfügbar
 - ➔ über die Vorlesungsseite



Unterstützung der Fehlersuche in `BSsync`

- ➔ Konstruktor `Semaphore(int wert, String name)`
Konstruktor `Lock(String name)`
Methode `newCondition(String name)` von `Lock`
 - ➔ Erzeugung eines Objekts mit gegebenem Namen
- ➔ Attribut `public static boolean verbose`
in den Klassen `Semaphore` und `Lock`
 - ➔ schaltet Protokoll aller Operationen auf Semaphoren bzw. Locks und Bedingungsvariablen ein
 - ➔ Protokoll benutzt obige Namen
- ➔ Erlaubt Verfolgung des Programmablaufs
 - ➔ z.B. bei Verklemmungen



Beispiel: Erzeuger/Verbraucher-Problem

```
public class ErzVerb {
    private Lock mutex;           // Wechsels. Ausschluß
    private Condition nonfull;   // Warten bei vollem Puffer
    private Condition nonempty; // Warten bei leerem Puffer
    private int count;          // Zählt belegte Pufferplätze

    Buffer buffer;

    public ErzVerb(int size) {
        buffer = new Buffer(size);
        mutex = new Lock("mutex");           // Lock erzeugen
        nonfull = mutex.newCondition("nonfull"); // Bedingungsvar.
        nonempty = mutex.newCondition("nonempty"); // erzeugen
        count = 0;
    }
}
```



Beispiel: Erzeuger/Verbraucher-Problem ...

```
public void insert(int item) {
    mutex.lock();           // Mutex sperren
    try {
        while (count == buffer.getSize()) // Puffer voll?
            nonfull.await();           // ja: warten...
        buffer.insertItem(item);       // Item eintragen
        count++;
        nonempty.signal();           // ggf. Thread wecken
    }
    finally { mutex.unlock(); }       // Mutex freigeben
}
```

3.10.2 Java: Synchronisationsklassen ...



★★

Beispiel: Erzeuger/Verbraucher-Problem ...

```
public int remove() {
    int result;
    mutex.lock(); // Mutex sperren
    try {
        while (count == 0) // Puffer leer?
            nonempty.await(); // ja: warten...
        result = buffer.removeItem(); // Item entfernen
        count--;
        nonfull.signal(); // ggf. Thread wecken
        return result;
    }
    finally { mutex.unlock(); } // Mutex freigeben
}
```

3.10.2 Java: Synchronisationsklassen ...



Beispiel: Erzeuger/Verbraucher-Problem ...

```
public static void main(String argv[]) {
    ErzVerb ev = new ErzVerb(5);
    Lock.verbose = true; // Protokoll anschalten
    Producer prod = new Producer(ev);
    Consumer cons = new Consumer(ev);
    prod.start();
    cons.start();
}
}
```

➔ Vollständiger Code im WWW (Vorlesungsseite)!



Reader/Writer-Locks

- ➔ Schnittstelle `ReadWriteLock`
 - ➔ zwei Methoden `readLock()` und `writeLock()`
 - ➔ geben ein Objekt der Schnittstelle `Lock` zurück
- ➔ Implementierungsklasse `ReentrantReadWriteLock`
 - ➔ Lese- und Schreibsperrern können rekursiv belegt werden
 - ➔ Schreiber kann auch noch Lesesperre belegen, aber nicht umgekehrt
 - ➔ für Schreibsperrern kann auch `Condition` erzeugt werden
 - ➔ im Konstruktor kann `Fairness` eingeschaltet werden

3.10.3 Synchronisation in C++



Mutex Variablen

- ➔ Verhalten ähnlich zu `Lava Lock` (binäres Semaphor)
 - ➔ Zustände: frei, gesperrt; bei Erzeugung: frei
- ➔ Deklaration (und Initialisierung):
`std::mutex mutex;`
- ➔ Zum Sperren des Mutex: Erzeugung eines Objekts der Klasse `std::unique_lock`:
 - ➔ `std::unique_lock<std::mutex> lock(mutex);`
 - ➔ Mutex wird automatisch freigegeben, wenn `lock` deallokiert wird, d.h., wenn aktueller Code-Block verlassen wird
- ➔ Klasse `mutex` erlaubt kein rekursives Sperren
 - ➔ d.h. selber Thread kann Mutex nicht zweimal sperren
 - ➔ Alternative: Klasse `recursive_mutex`

Anmerkungen zu Folie 215:

Mehr Informationen zu den Mutex und Lock Klassen finden Sie im C++ Referenz-Handbuch, z.B. <http://www.cplusplus.com/reference/mutex/>.

215-1

3.10.3 Synchronisation in C++ ...



Bedingungsvariablen

- ➔ Deklaration (und Initialisierung):
`std::condition_variable cond;`
- ➔ Wichtige Methoden:
 - ➔ wait: `void wait(unique_lock<mutex>& lock)`
 - ➔ Thread wird blockiert, das von `lock` verwaltete Mutex wird temporär freigegeben
 - ➔ signalisierender Thread behält das Mutex (Signalisierungsvariante 1a)
 - ➔ daher typisch: `while (!condition_met) cond.wait(lock);`
 - ➔ Signalisierung eines Threads: `void notify_one()`
 - ➔ Signalisierung aller Threads: `void notify_all()`

Anmerkungen zu Folie 216:

Die Syntax `unique_lock<mutex>& lock` zeigt an, dass das Argument der Methode `wait()` per Referenz (und nicht per Wert) übergeben wird.

216-1

3.10.3 Synchronisation in C++ ...



Beispiel: Nachbildung eines Monitors

```
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mutex;
std::condition_variable cond;
volatile bool ready = false;
volatile int result;

void StoreResult(int arg) {
    std::unique_lock<std::mutex> lock(mutex);
    result = arg; /* Ergebnis speichern */
    ready = true;
    cond.notify_all();
    // 'lock' wird bei Verlassen der Methode deallokiert, damit wird 'mutex' freigegeben!
}
```

Anmerkungen zu Folie 217:

Das Schlüsselwort `volatile` zeigt an, dass der Wert der Variablen „flüchtig“ ist, sich also jederzeit ändern kann. Das verbietet bestimmte Optimierungen im Compiler (insbesondere das „Caching“ des Werts in einem Register), siehe Anmerkung zu Folie 175.

Bei den Deklarationen von Mutex- und Bedingungsvariablen ist dabei kein `volatile` notwendig.

217-1

3.10.3 Synchronisation in C++ ...



Beispiel: Nachbildung eines Monitors ...

```
int ReadResult()  
{  
    std::unique_lock<std::mutex> lock(mutex);  
    while (!ready)  
        cond.wait(lock);  
    return result;  
    // mutex wird automatisch freigegeben  
}
```

Anmerkungen zu Folie 218:

Der C++ Standard erlaubt, dass `wait()` zurückkehrt, auch ohne dass eine Signalisierung erfolgt ist (Das ist manchmal notwendig zur Behandlung von Signalen, siehe Kap. 3.2.4). Auch aus diesem Grund muss `wait()` immer in einer `while`-Schleife verwendet werden.

218-1

3.10.4 Synchronisation in C



Mutex Variablen

- ➔ Analog zu Java `Lock` und C++ `std::mutex`
 - ➔ Zustände: gesperrt, frei; Initialzustand: frei
- ➔ Deklaration und Initialisierung (globale/statische Variable):
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- ➔ Operationen:
 - ➔ Sperren: `pthread_mutex_lock(&mutex)`
 - ➔ Verhalten bei rekursiver Belegung nicht festgelegt
 - ➔ Freigeben: `pthread_mutex_unlock(&mutex)`
 - ➔ bei Terminierung eines Threads werden Sperren *nicht* automatisch freigegeben!
 - ➔ Sperrversuch: `pthread_mutex_trylock(&mutex)`
 - ➔ blockiert nicht, liefert ggf. Fehlercode

Anmerkungen zu Folie 219:

Die allgemeine Methode zur Deklaration und Initialisierung eines Mutex, die auch dann funktioniert, wenn das Mutex in einer lokalen Variablen liegt, ist:

```
pthread_mutex_t mutex;           /* Deklaration */
pthread_mutex_init(&mutex, NULL); /* Initialisierung */
/* verwende das Mutex ... */
pthread_mutex_destroy(&mutex);  /* Freigabe */
```

Bei der Initialisierung können als zweites Argument zusätzliche Attribute angegeben werden. Unter anderem kann so das Verhalten des Mutex bei rekursiver Belegung durch denselben Thread definiert werden:

- ➔ Verklemmung: Thread wartet, da Mutex (durch ihn selbst) gesperrt ist
- ➔ Fehlermeldung
- ➔ Belegungszähler: Thread kann das Mutex nochmal sperren, muss es dann aber auch entsprechend oft wieder freigeben

Viele dieser Attribute sind jedoch plattformspezifisch.

219-1

3.10.4 Synchronisation in C ...



Bedingungsvariablen

- ➔ Deklaration und Initialisierung (globale/statische Variable):

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- ➔ Operationen:

- ➔ Warten: `pthread_cond_wait(&cond, &mutex)`

- ➔ Thread wird blockiert, `mutex` wird temporär freigegeben

- ➔ signalisierender Thread behält `mutex`

- ➔ typische Verwendung:

```
while (!condition_met)
```

```
    pthread_cond_wait(&cond, &mutex);
```

- ➔ Signalisieren:

- ➔ eines Threads: `pthread_cond_signal(&cond)`

- ➔ aller Threads: `pthread_cond_broadcast(&cond)`

Anmerkungen zu Folie 220:

Die allgemeine Methode zur Deklaration und Initialisierung einer Bedingungsvariable, die auch dann funktioniert, wenn sie in einer lokalen Variablen liegt, ist:

```
pthread_cond_t cond;           /* Deklaration */
pthread_cond_init(&cond, NULL); /* Initialisierung */
/* ... verwende die Bedingungsvariable */
pthread_cond_destroy(&cond);   /* Freigabe */
```

Wie beim Mutex können bei der Initialisierung als zweites Argument zusätzliche Attribute angegeben werden.

220-1

3.10.4 Synchronisation in C ...



Beispiel: Nachbildung eines Monitors

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

volatile bool ready = false;
volatile int result;

void StoreResult(int arg)
{
    pthread_mutex_lock(&mutex);
    ergebnis = arg; /* speichere Ergebnis */
    ready = true;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Beispiel: Nachbildung eines Monitors ...

```
int ReadResult()  
{  
    int tmp;  
    pthread_mutex_lock(&mutex);  
    while (!ready)  
        pthread_cond_wait(&cond, &mutex);  
    tmp = ergebnis; /* lies Ergebnis */  
    pthread_mutex_unlock(&mutex);  
    return tmp;  
}
```

Anmerkungen zu Folie 222:

Der PThread Standard definiert noch eine Reihe weiterer Funktionen, u.a.:

<code>pthread_self</code>	Liefert eigene Thread-ID
<code>pthread_once</code>	Führt (Initialisierungs-)Funktion genau einmal aus
<code>pthread_kill</code>	Sendet Signal an anderen Thread innerhalb des Prozesses – von 'außen' nur Signal an Prozeß (d.h. irgendeinen Thread) möglich
<code>pthread_sigmask</code>	Setzt Signalmaske – jeder Thread hat eigene Signalmaske
<code>pthread_cond_timedwait</code>	Wie ...wait, aber mit max. Wartezeit

- ➔ Ziel: BS-Aufrufe vermeiden, wann immer das möglich ist
- ➔ Basis: Systemaufruf `futex` (*Fast User space muTEX*)
- ➔ Argumente u.a.:
 - ➔ `addr1`: Adresse einer `int`-Variable im Benutzeradreibraum
 - ➔ `op`: auszuführende Futex-Operation
 - ➔ `val1`: abhängig von Operation
- ➔ Operationen u.a.:
 - ➔ `FUTEX_WAIT`: falls `*addr1 == val1` blockiere den Thread
 - ➔ Abfrage und Blockierung erfolgen atomar, da im BS-Kern
 - ➔ `FUTEX_WAKE`: wecke `val1` Threads auf, die wegen eines `FUTEX_WAIT` mit Adresse `addr1` blockiert sind

Realisierung eines Mutex

```
class Mutex
{
    int nThreads = 0;           // Zahl der Threads vor/im kritischen Abschnitt
    bool signaled = false;     // Zur Signalisierung

    void lock()
    {
        if (fetchAndAdd(nThreads, 1) == 0) // Erster Thread
            return;                       // darf in kritischen Abschnitt

        while (!fetchAndSet(signaled, false)) // Warte auf
            futex(&signaled, FUTEX_WAIT, false); // Signalisierung
    }
}
```

Realisierung eines Mutex ...

```
void unlock()
{
    if (fetchAndAdd(nThreads, -1) == 1) // Einziger Thread
        return; // muss niemanden wecken

    signaled = true;
    futex(&signaled, FUTEX_WAKE, 1); // Einen Thread wecken
}
};
```

- ➔ Systemaufruf nur dann, wenn ein Thread blockiert bzw. geweckt werden muß

Anmerkungen zu Folie 225:

- ➔ Die angegebene Implementierung stammt aus dem Forum <https://softwareengineering.stackexchange.com/questions/340284/mutex-vs-semaphore-how-to-implement-them-not-in-terms-of-the-other>
- ➔ Die `fetchAndSet`-Operation in `lock()` dient zusammen mit der weiteren Abfrage im `futex`-Systemaufruf zur Vermeidung von *lost wakeups*.

Betrachten Sie z.B. die Verzahnung

```
                    signaled = true;
                    futex(&signaled, FUTEX_WAKE, 1);
while (!fetchAndSet(signaled, false))
    futex(&signaled, FUTEX_WAIT, false);
```

Das `fetchAndSet` stellt hier sicher, daß nur genau ein Thread, der sich in `lock()` befindet, nicht blockiert und in den kritischen Abschnitt gehen kann.

In der Situation

```
while (!fetchAndSet(signaled, false))
    signaled = true;
                    futex(&signaled, FUTEX_WAKE, 1);
    futex(&signaled, FUTEX_WAIT, false);
```

stellt die Abfrage `signaled == false`, die im `futex`-Systemaufruf gemacht wird sicher, daß der Thread nicht blockiert wird.

- ➔ Beachten Sie, daß der Code keinerlei Fehlerüberprüfungen enthält. Z.B. würde ein Aufruf von `unlock()` auf einem freien Mutex zu Fehlverhalten führen.
- ➔ Das Problem der Speicherkonsistenz (siehe Abschnitt 3.6) wird in diesem Codebeispiel ebenfalls nicht explizit behandelt.
- ➔ Eine gute Beschreibung von *Futexes* mit weitere Implementierungen und einer Diskussion möglicher Fehler, die man machen kann, finden Sie in dem Artikel <http://www.akkadia.org/drepper/futex.pdf>.

225-2

3.10.5 Synchronisationsunterstützung in Linux ...



Realisierung von `signalAll()` für Bedingungsvariable

- ➔ Problem: alle Threads werden geweckt und versuchen gleichzeitig, das Mutex zu sperren
 - ➔ alle bis auf einen werden wieder blockiert
- ➔ Lösung: weitere Operation `FUTEX_CMP_REQUEUE`
 - ➔ wie bei `FUTEX_WAIT` wird eine Anzahl Threads aufgeweckt
 - ➔ alle anderen Threads, die am *Futex* (der Bedingungsvariable) warten, werden direkt in die Warteschlange eines zweiten *Futex* (des Mutex) verschoben

Anmerkungen zu Folie 226:

Eine ausführlichere Diskussion finden Sie in <http://www.akkadia.org/drepper/futex.pdf> im Abschnitt 8.

226-1

3.11 Lock-free Datenstrukturen



- ➔ Ziel: Datenstrukturen (typ. *Collections*) ohne wechselseitigem Ausschluss
 - ➔ performanter, keine Gefahr von Verklemmungen
- ➔ Typische Vorgehensweise:
 - ➔ Verwendung atomarer *Read-Modify-Write*-Befehle anstelle von Sperren
 - ➔ im Konfliktfall, d.h. bei gleichzeitiger Änderung durch einen anderen Thread, wird die betroffene Operation wiederholt
- ➔ **Lock-free**: unter allen Umständen macht **mindestens ein** Thread nach endlich vielen Schritten Fortschritt
- ➔ **Wait-free**: unter allen Umständen macht **jeder** Thread nach endlich vielen Schritten Fortschritt



Beispiel: Anfügen am Ende eines Arrays

```
Data buffer[N]; // Puffer-Array
int wrPos = 0; // Position für nächstes einzutragendes Element

void append(Data data) {
    int wrPosOld = fetchAndAdd(wrPos, 1);
    buffer[wrPosOld] = data;
}
```

➔ Nutzt atomare *fetch-and-add* Operation:

```
int fetchAndAdd(int &addr, int val) {
    int tmp = addr;
    addr += val;
    return tmp;
}
```

} **Atomar!**

➔ erhöht Wert bei `addr` um `val`, gibt alten Wert zurück



Beispiel: Treiber-Stack

➔ *Lock-free* Implementierung eines Stacks auf Basis einer verketteten Liste (von R. Kent Treiber)

➔ Basis: atomare *compare-and-set* Operation:

```
bool compareAndSet(T &addr, T expect, T newval) {
    if (addr == expect) {
        addr = newval;
        return true;
    }
    return false;
}
```

} **Atomar!**

➔ falls der Wert bei `addr` noch dem erwarteten Wert `expect` übereinstimmt, überschreibe ihn mit `newval`

3.11 Lock-free Datenstrukturen ...



```
Node top = null; // zeigt auf oberstes Kellerelement

void push(Data data) { // Neues Element auf den Keller schreiben
    Node tmp;
    Node node = new Node(data); // Neues Listenelement erzeugen
    do {
        tmp = top;           // top merken
        node.next = tmp; // Verkettung
    } while (!compareAndSet(top, tmp, node)); // top aktualisieren
}

Data pop() { // Oberstes Element vom Keller entfernen
    Node tmp, next; // Hier ohne Fehlerbehandlung (leerer Keller) gezeigt!
    do {
        tmp = top;           // top merken
        next = tmp.next; // next = zweitoberstes Element
    } while (!compareAndSet(top, tmp, next)); // top aktualisieren
    return tmp.data;
}
```

Anmerkungen zu Folie 230:

Der Treiber Stack ist *lock-free*, da bei mehreren in Konflikt stehenden Threads immer genau einer noch `top == tmp` findet, also die Operation erfolgreich beendet.

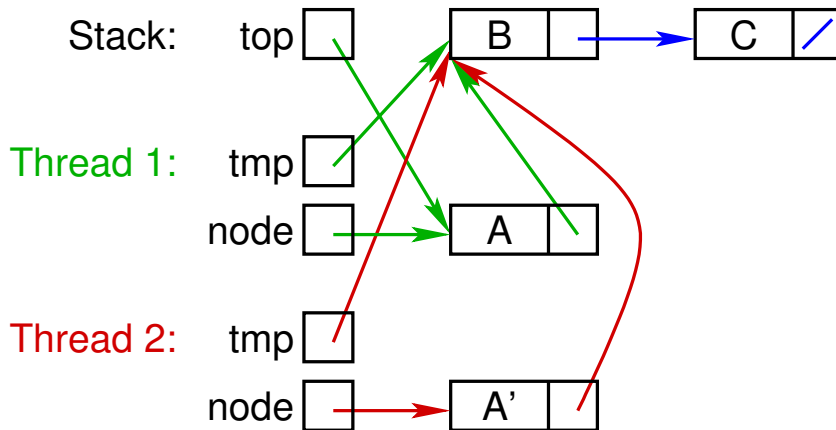
Er ist aber nicht *wait-free*, da es nicht ausgeschlossen ist, dass immer wieder derselbe Thread die Operation wiederholen muß, da ihm jedesmal ein anderer zuvor kommt.

3.11 Lock-free Datenstrukturen ...



(Animierte Folie)

Beispiel



```
Node node = new Node(data);
do{
    tmp = top;
    node.next = tmp;
} while (!compareAndSet(top, tmp, node));
```

3.11 Lock-free Datenstrukturen ...



Das ABA-Problem

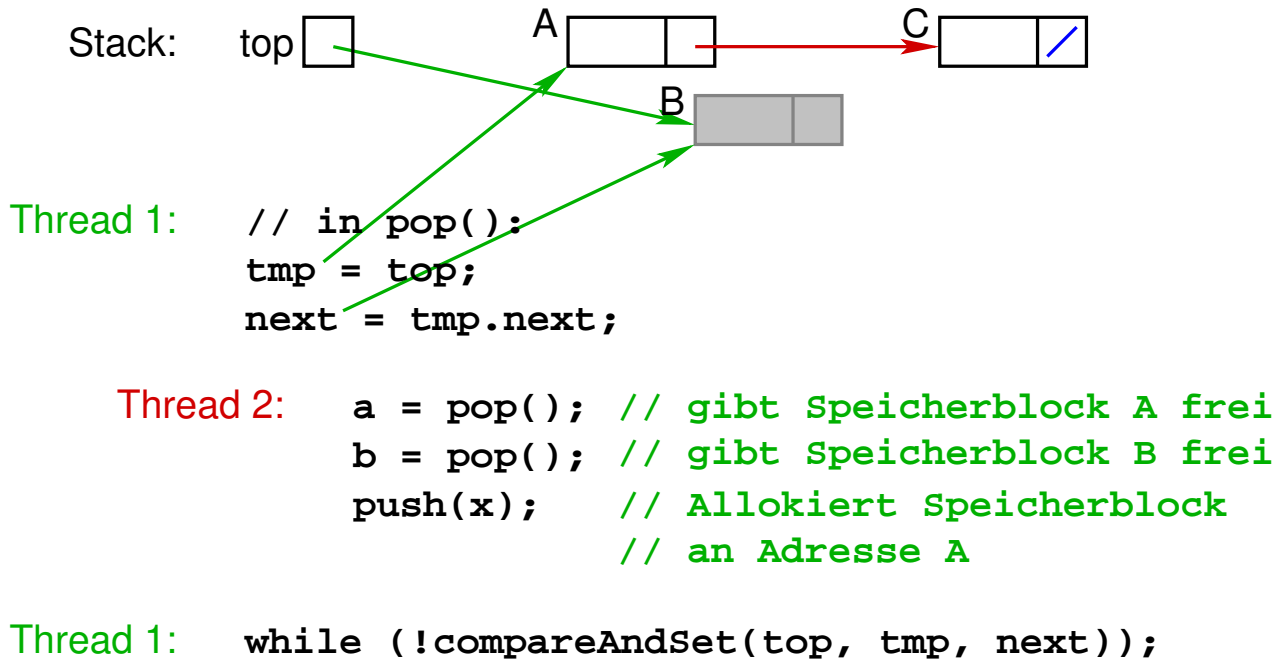
- ➔ Lock-free Implementierungen müssen Konflikte feststellen können
 - ➔ d.h.: wurde die Datenstruktur zwischenzeitlich von einem anderen Thread geändert?
 - ➔ falls ja: Operation wiederholen
- ➔ Dazu häufig: Test, ob eine bestimmte Referenz verändert wurde
 - ➔ beim Treiber Stack: Referenz auf oberstes Kellerelement
- ➔ Bei Sprachen ohne *Garbage-Collector* nicht ausreichend
 - ➔ z.B. beim Treiber Stack:
 - ➔ Entfernen / Freigeben der obersten beiden Elemente (A, B)
 - ➔ Schreiben eines neues Element auf den Stack
 - ➔ kann an selber Adresse allokiert sein, wie das alte A!
- ➔ Lösungen sind verfügbar, jedoch komplex

3.11 Lock-free Datenstrukturen ...



(Animierte Folie)

Das ABA Problem beim Treiber-Stack



3.11 Lock-free Datenstrukturen ...



- ➔ Es gibt etliche Bibliotheken mit *lock-free* bzw. *wait-free* Datenstrukturen für Java und C++
 - ➔ Java: z.B. Amino Concurrent Building Blocks, Highly Scalable Java
 - ➔ C++: z.B. boost.lockfree, libcds, Concurrency Kit, liblfd
- ➔ Programmiersprachen und/oder Compiler stellen *Read-Modify-Write*-Operationen bereit, z.B.:
 - ➔ Java: Paket `java.util.concurrent.atomic`
 - ➔ C++: Klasse `std::atomic<T>`
 - ➔ gcc/g++: eingebaute Funktionen `__sync_...()` bzw. `__atomic_...()`

- ➔ Probleme der bisherigen Ansätze:
 - Sperren: Performanz, fehleranfällig (z.B. Verklemmungen)
 - *Lock-free*: Einschränkung durch Wortbreite
 - Kompositionalität, z.B. atomares Verschieben eines Elements von einem Stack in einen anderen
- ➔ Lösungsidee: Nutzung von Transaktionen auf dem Speicher
- ➔ Dafür relevante Eigenschaften von Transaktionen:
 - *Atomicity*: entweder werden alle Operationen der Transaktion durchgeführt, oder keine
 - *Consistency*: Transaktionen erhalten die Konsistenz der Daten
 - *Isolation*: Ergebnis nebenläufig durchgeführter Transaktionen entspricht immer einer (beliebigen) sequentiellen Ausführung

Anmerkungen zu Folie 235:

- ➔ Zur Realisierung von *Lock-free*-Datenstrukturen wäre es häufig notwendig, mehrere Verweise gleichzeitig mit *Compare-and-Set* setzen zu können. Dies ist technisch aber zumindest schwierig.
- ➔ Zur Kompositionalität: Angenommen, es gibt eine korrekt synchronisierte Stack-Implementierung, wie z.B. den Treiber-Stack. Ein Thread soll nun ein Element zwischen den beiden Stacks verschieben, also

```
stack2.push(stack1.pop());
```

Dies allerdings so, dass kein anderer Thread einen Zustand sehen kann, in dem das Element „verloren“ ist, also in `stack1` schon entfernt, aber in `stack2` noch nicht eingefügt ist.
Es gibt mit den bisherigen Ansätzen keine Möglichkeit, dies zu gewährleisten, ohne die Stack-Implementierung zu ändern.
- ➔ Die vierte der sog. ACID-Eigenschaften von klassischen Datenbank-Transaktionen ist *Durability*, d.h. das Ergebnis der Transaktion wird dauerhaft gespeichert. Diese Eigenschaft kann bei Hauptspeicher-Transaktionen natürlich nicht umgesetzt werden, da der Hauptspeicher ein flüchtiger Speicher ist.



Basisprimitive für Transactional Memory

- ➔ `atomic`-Blöcke
 - ➔ Anweisungen innerhalb des Blocks werden atomar und serialisierbar ausgeführt
 - ➔ in Bezug auf alle(!) anderen `atomic`-Blöcke
- ➔ `abort`-Befehl (innerhalb eines `atomic`-Blocks)
 - ➔ führt zum Abbruch der Transaktion
 - ➔ Verlassen des Blocks; Wiederherstellen des alten Zustands
- ➔ `retry`-Befehl (innerhalb eines `atomic`-Blocks)
 - ➔ Abbruch der Transaktion (mit Wiederherstellung)
 - ➔ neuer Versuch
- ➔ `orElse`-Befehl (optionale Erweiterung)
 - ➔ alternative Transaktion, falls erste mit `retry` abbricht



Beispiel: Warteschlangen

- ➔ Element aus Warteschlange entfernen:

```
Data dequeue() {  
    atomic { // Atomare Ausführung  
        if (first == null) // Falls Liste leer:  
            retry; // Transaktion neu starten  
        Data res = first.data;  
        first = first.next;  
        return res;  
    }  
}
```

- ➔ Atomares Verschieben zwischen zwei Listen:

```
atomic {  
    q1.enqueue(q2.dequeue());  
}
```



Beispiel: Warteschlangen ...

➔ Alternatives Lesen aus zwei Warteschlangen:

```
atomic {  
    x = q1.dequeue();  
}  
orElse {  
    x = q2.dequeue();  
}
```

- ➔ falls Transaktion in `q1.dequeue()` mit `retry` abbricht: versuche `q2.dequeue()`
- ➔ falls diese auch mit `retry` abbricht: wieder von vorne



Transactional Memory Systeme

- ➔ Realisierungen in Software (Bibliotheken, Compiler) oder Hardware verfügbar
- ➔ Unterscheidung *weak / strong isolation*
 - ➔ werden in `atomic`-Blöcken genutzte Variablen auch ausserhalb der `atomic`-Blöcke geschützt?
- ➔ Unterschiedliche Behandlung geschachtelter Transaktionen:
 - ➔ flache Tr.: gesamte Tr. bricht ab, falls innere Tr. abbricht
 - ➔ offene Tr.: Ergebnisse erfolgreicher innerer Tr. sofort sichtbar (selbst wenn äußere Tr. abbricht)
 - ➔ geschlossene Tr.: Ergebnisse erfolgreicher innerer Tr. erst sichtbar, wenn äußere Tr. erfolgreich beendet ist

Anmerkungen zu Folie 239:

Eine Zusammenstellung von *Transactional Memory* Implementierungen finden Sie z.B. auf der Wikipedia-Seite:

https://en.wikipedia.org/wiki/Software_transactional_memory

239-1

3.13 Zusammenfassung / Wiederholung



- ➔ Synchronisation
 - ➔ wechselseitiger Ausschluß
 - ➔ nur jeweils ein Thread darf im kritischen Abschnitt sein
 - ➔ kritischer Abschnitt: Zugriff auf gemeinsame Ressourcen
 - ➔ Lösungsansätze:
 - ➔ Sperren der Interrupts (nur im BS, Einprozessorsysteme)
 - ➔ Sperrvariable: Peterson-Algorithmus
 - ➔ mit Hardware-Unterstützung: *Read-Modify-Write*
 - ➔ Nachteil: Aktives Warten (Effizienz, Verklemmungsgefahr)



➔ Semaphore

- ➔ besteht aus Zähler und Threadwarteschlange
 - ➔ $P()$: herunterzählen, ggf. blockieren
 - ➔ $V()$: hochzählen, ggf. blockierten Thread wecken
 - ➔ Atomare Operationen (im BS realisiert)

➔ wechselseitiger Ausschluß:

Thread 0

```
P(Mutex);
```

```
// kritischer Abschnitt
```

```
V(Mutex);
```

Thread 1

```
P(Mutex);
```

```
// kritischer Abschnitt
```

```
V(Mutex);
```

- ➔ auch für Reihenfolgesynchronisation nutzbar
 - ➔ Beispiel: Erzeuger/Verbraucher-Problem



➔ Monitor

- ➔ Modul mit Daten, Prozeduren, Initialisierung
- ➔ Datenkapselung
- ➔ Prozeduren stehen unter wechselseitigem Ausschluß
- ➔ Bedingungsvariable zur Synchronisation
 - ➔ `wait()` und `signal()`
- ➔ Varianten bei der Signalisierung:
 - ➔ einen / alle wartenden Threads wecken?
 - ➔ erhält geweckter Thread sofort den Monitor?
 - ➔ falls nicht: Bedingung nach Rückkehr aus `wait()` erneut prüfen!

- ➔ Synchronisation in Java:
 - ➔ Klassen mit `synchronized` Methoden
 - ➔ wechselseitiger Ausschluß der Methoden (pro Objekt)
 - ➔ `wait()`, `notify()`, `notifyAll()`
 - ➔ genau eine (implizite) Bedingungsvariable pro Objekt
 - ➔ JDK 1.5: Semaphore, *Locks* und Bedingungsvariablen
 - ➔ *Locks* und Bedingungsvariable erlauben die genaue Nachbildung des Monitorkonzepts
 - ➔ *Locks* für wechselseitigen Ausschluß der Methoden
 - ➔ Bedingungsvariablen sind fest an *Lock* gebunden
 - ➔ mehrere Bedingungsvariablen pro Objekt möglich