Betriebssysteme und nebenläufige Programmierung

SoSe 2025

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

Stand: 10. April 2025

Betriebssysteme und nebenläufige Programmierung

SoSe 2025

3 Synchronisation

Vorbemerkung

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

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

3.1 Einführung und Motivation

- 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

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich

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

in:	7
out:	4

Thread A

```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich

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

in:	7
out:	4

Thread A

```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich

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

in:	7
out:	4

Thread A

```
...
s[in]="d1";
in=in+1;
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich

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

in:	8
out:	4

Thread A

```
...
s[in]="d1";
in=in+1;
```

Thread B

Beispiel: Drucker-Spooler, korrekter Ablauf

Spoolbereich

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

in:	9
out:	4

Thread A

```
...
s[in]="d1";
in=in+1;
```

Unterbrechung

Threadwechsel

Thread B

```
...
s[in]="d2";
in=in+1;
...
```

Beispiel: Drucker-Spooler, fehlerhafter Ablauf

Spoolbereich

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

in:	7
out:	4

Thread A

```
...
s[in]="d1";
```

Thread B

Beispiel: Drucker-Spooler, fehlerhafter Ablauf

Spoolbereich

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

in:	7
out:	4

Thread A

```
...
s[in]="d1";
```

Thread B

```
...
s[in]="d2";
in=in+1;
...
```

Unterbrechung

Threadwechsel

Beispiel: Drucker-Spooler, fehlerhafter Ablauf

Spoolbereich

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

in:	8
out:	4

Thread A

```
...
s[in]="d1";
```

Thread B

```
...
s[in]="d2";
in=in+1;
...
```

Unterbrechung

Threadwechsel

Beispiel: Drucker-Spooler, fehlerhafter Ablauf

Spoolbereich

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

in:	9
out:	4

Thread A

```
...
s[in]="d1";
```

Thread B

```
...
s[in]="d2";
in=in+1;
...
```

Unterbrechung

Threadwechsel

```
in=in+1;
```

➥ ***Race Condition***

Arten der Synchronisation

- **Sperrsynchronisation** (**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	Thread 1
`// Datei erzeugen`	`while (!ready); // aktives Warten`
`ready = true;`	`Datei öffnen`

3.3 Wechselseitiger Ausschluß

- **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
  - Sperrsynchronisation
- Gesucht: Methode zum wechselseitigen Ausschluß kritischer Abschnitte

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

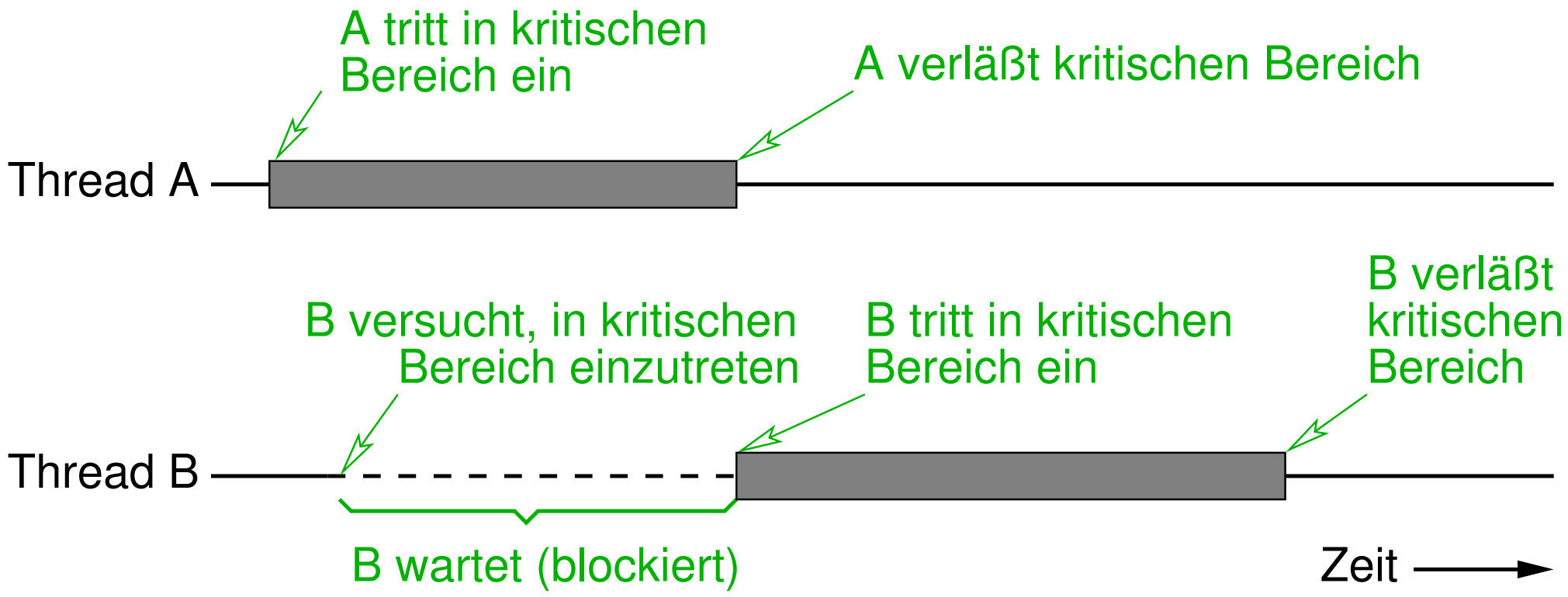
```
begin_region();
s[in]="d1";
in=in+1;
end_region();
```

Thread B

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

➥ Frage: Implementierung von `begin_region()` / `end_region()`?

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

Lösungsversuch 2: Sperrvariable

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

```
while(belegt);
belegt = true;      } begin_region()
// kritischer Abschnitt
belegt = false      } end_region()
```

Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:

Lösungsversuch 2: Sperrvariable

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

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

begin_region()

end_region()

Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`

Lösungsversuch 2: Sperrvariable

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

```
while(belegt);
belegt = true;      } begin_region()
// kritischer Abschnitt
belegt = false      } end_region()
```

Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
belegt = false
```

- Problem: *Race Condition*:
  - Threads führen gleichzeitig `begin_region()` aus
  - lesen gleichzeitig `belegt`
  - finden `belegt` auf `false`
  - setzen `belegt=true`

Lösungsversuch 2: Sperrvariable

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

```
while(belegt);
belegt = true;      } begin_region()
// kritischer Abschnitt
belegt = false      } end_region()
```

Thread 1

```
while(belegt); // warten ...
belegt = true;
// kritischer Abschnitt
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

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

Thread 1

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

- Verklemmung wird durch `turn` verhindert
- Jeder Thread bekommt die Chance, den kritischen Bereich zu betreten
  - keine **Verhungerung**

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

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

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

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 Invalidierung aller anderen Kopien bei jedem *Test-and-Set*
- Während des Wartens wird der betroffene Cache-Block somit laufend invalidiert
  - hohe Bus-Belastung durch Invalidierung und Neu-Laden

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

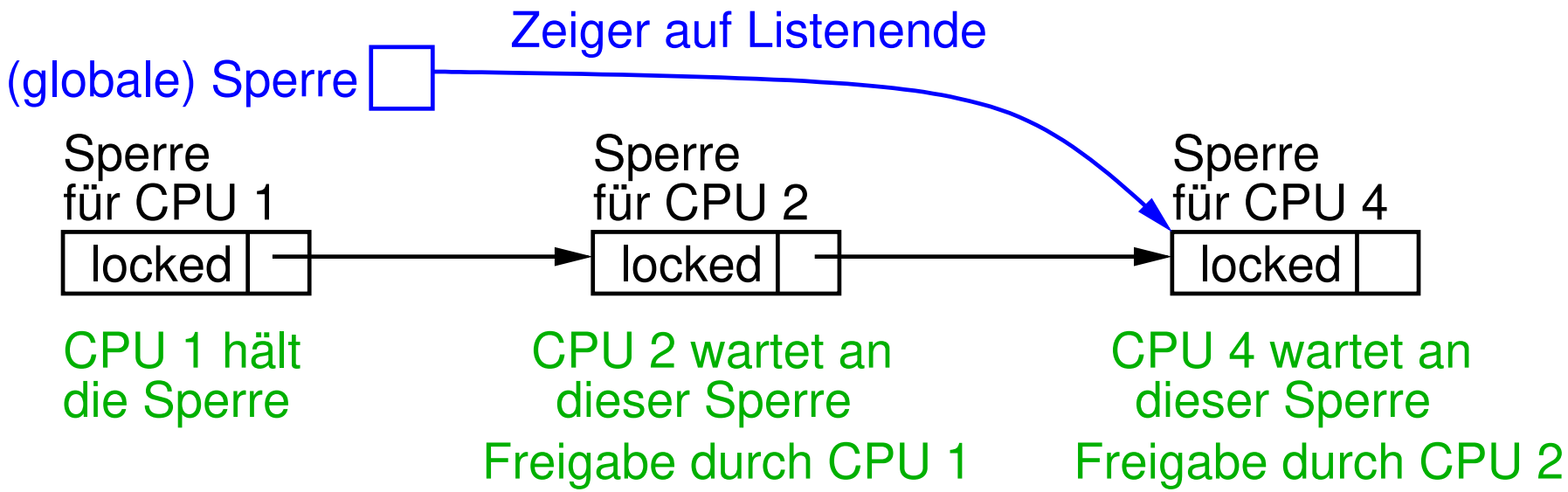
- Spezielle Maschinenbefehle zur Realisierung von *Spin Locks*
- *Load Linked*
  - Laden aus dem Speicher (bzw. Cache)
  - Adresse wird in *Link Register* vermerkt
  - *Link Register* wird bei Invalidierung der Cachezeile gelöscht
- *Store Conditional*
  - speichert einen Wert, falls die Adresse mit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

Reihenfolgesynchronisation über eine gemeinsame Variable

- Typisches Beispiel:

  Initialisierung:

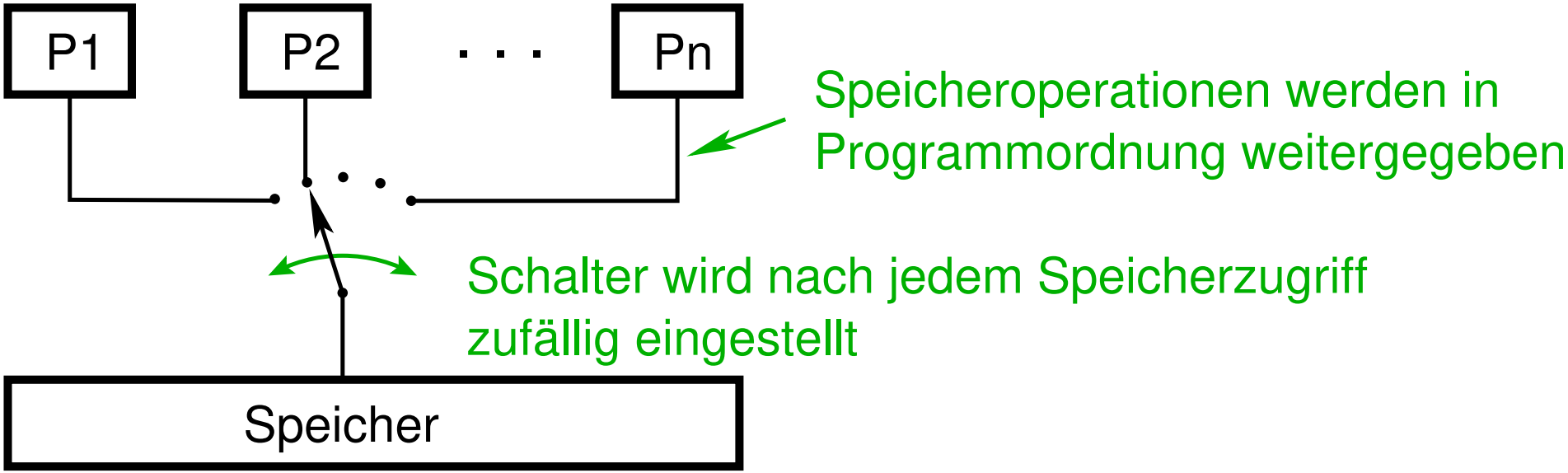
```
double value = 0;
boolean ready = false; // ist 'value' gültig?
```

Thread 0	Thread 1
`value = 0.5*2;`	`while (!ready); // aktives Warten`
`ready = true;`	`use(value);`

- 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**

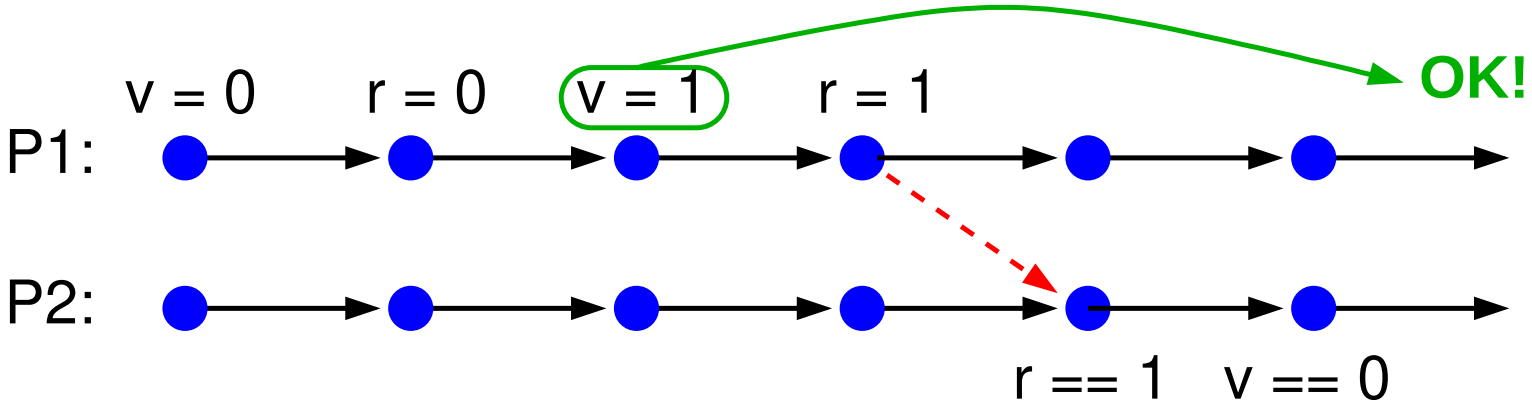
Konsistenzmodelle

- 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:



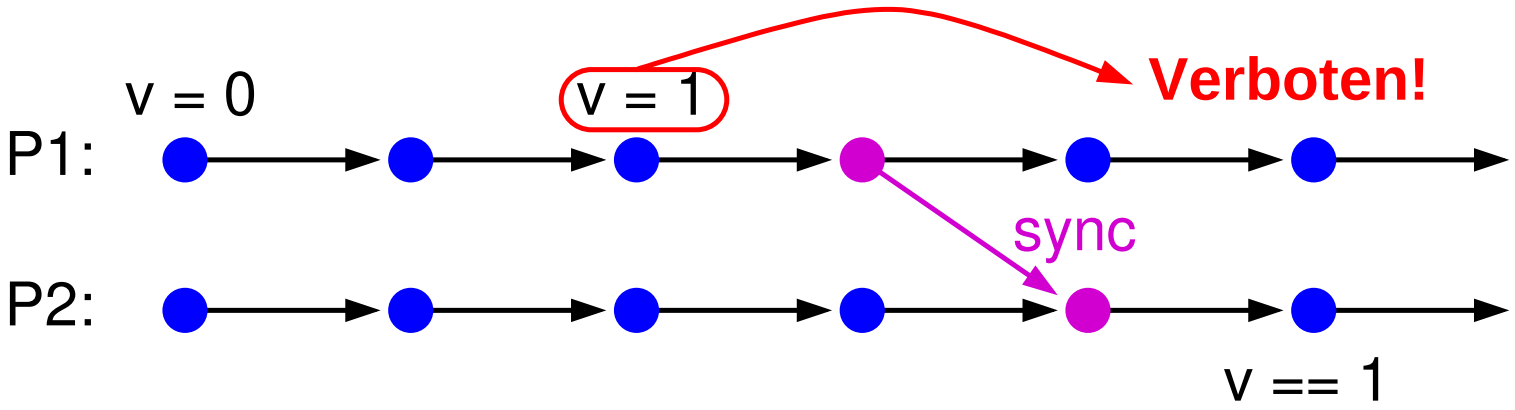
Konsistenzmodelle ...

- 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 Synchronisations-operationen hinweg ist nicht erlaubt



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 Synchronisations-operationen 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	Thread 1
`value = 0.5*2;`	`while (!ready); // aktives Warten`
`ready = true;`	`use(value);`

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)`

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-synchonisation
- 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

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

```
P(mutex);
// kritischer Abschnitt
V(mutex);
```

Thread 1

```
P(mutex);
// kritischer Abschnitt
V(mutex);
```

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

Reihenfolgesynchronisation mit Semaphoren

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

```
// Datei erzeugen
V(sema);
```

Thread 1

```
P(sema);
// Datei öffnen
```

- Semaphor `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

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:
  - Sperrsynchronisation: wechselseitiger Ausschluß
  - Reihenfolgesynchronisation:
    - kein Entfernen aus leerem Puffer: Verbraucher muß warten
    - kein Einfügen in vollen Puffer: Erzeuger muß warten

Lösung des Erzeuger/Verbraucher-Problems

Erzeuger

```
while (true) {
   item = produce();

   insertItem(item);

}
```

Verbraucher

```
while (true) {

   item = removeItem();

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

```
Semaphor mutex = 1;
```

für wechselseitigen Ausschluß

Erzeuger

```
while (true) {
   item = produce();

   P(mutex);
   insertItem(item);
   V(mutex);

}
```

Verbraucher

```
while (true) {

   P(mutex);
   item = removeItem();
   V(mutex);

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

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

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

Erzeuger

```
while (true) {
   item = produce();

   P(mutex);
   insertItem(item);
   V(mutex);
   V(full);
}
```

Verbraucher

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

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

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

für wechselseitigen Ausschluß

verhindert Entfernen aus leerem Puffer

Erzeuger

```
while (true) {
   item = produce();

   P(mutex);
   insertItem(item);
   V(mutex);
   V(full);
}
```

Verbraucher

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

   consume(item);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

`Semaphor mutex = 1;`	für wechselseitigen Ausschluß
`Semaphor full = 0;`	verhindert Entfernen aus leerem Puffer
`Semaphor empty = N;`	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);
}
```

Lösung des Erzeuger/Verbraucher-Problems

Semaphore

`Semaphor mutex = 1;`	für wechselseitigen Ausschluß
`Semaphor full = 0;`	verhindert Entfernen aus leerem Puffer
`Semaphor empty = N;`	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);
}
```

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

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   readDataBase();

   useData();
}
```

Semaphore und gemeinsame Variable

Schreiber

```
while(true) {
   createData();

   writeDataBase();

}
```

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   readDataBase();

   useData();
}
```

Semaphore und gemeinsame Variable

```
Semaphor db=1; // Schützt Datenbank
```

Schreiber

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

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

    P(db);

  readDataBase();

    V(db);

  useData();
}
```

Semaphore und gemeinsame Variable

```
Semaphor db=1; // Schützt Datenbank
```

Schreiber

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

Lösung des Leser-Schreiber-Problems

Leser

```
while (true) {

   rc++;
   if (rc == 1)
      P(db);

   readDataBase();

   rc--;
   if (rc == 0)
      V(db);

   useData();
}
```

Semaphore und gemeinsame Variable

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

Schreiber

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

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

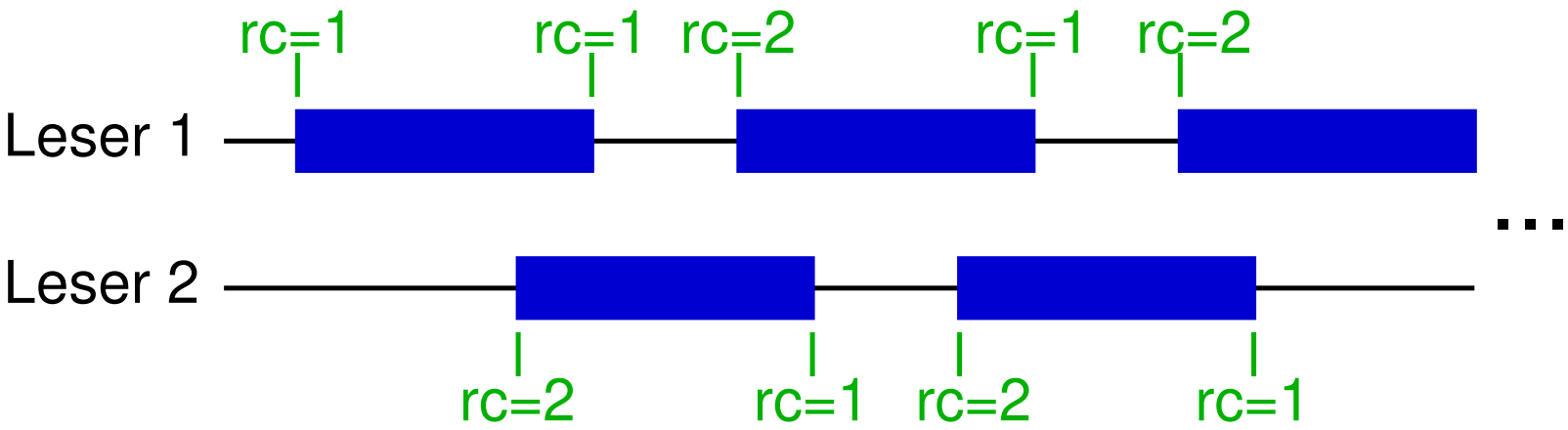
```
int rc=0;          // Anzahl Leser
Semaphor db=1;     // Schützt Datenbank
Semaphor mutex=1;
```

Schreiber

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

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

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 Bedingungs-variable 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()`:

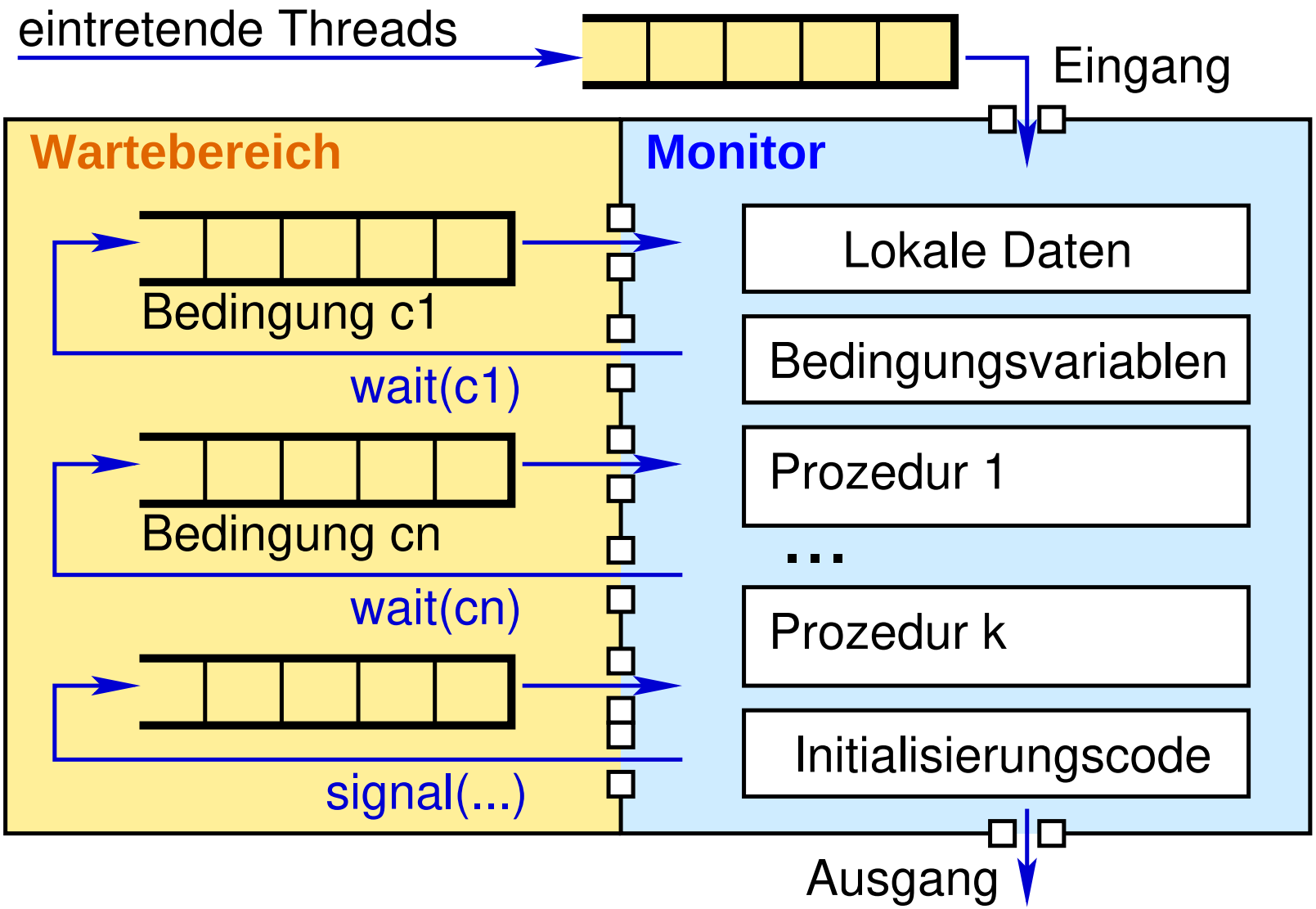
1. Ein Thread wird geweckt (meist der am längsten wartende)
   - **a)** signalisierender Thread bleibt im Besitz des Monitors
   - **b)** geweckter Thread erhält den Monitor sofort
     - **i.** signalisierender Thread muß sich erneut bewerben (Hoare)
     - **ii.** `signal()` muß letzte Anweisung in 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



Semaphor-Realisierung m. Monitor (Pascal-artig, Brinch Hansen)

```
monitor Semaphor
  condition nonbusy;
  integer count;

  procedure P
  begin
    count := count - 1;
    if count < 0 then
      wait(nonbusy);
  end;

  procedure V
  begin
    count := count + 1;
    if count <= 0 then
      signal(nonbusy);
  end;

  count = 1;
end monitor;
```

➥ Umgekehrt können auch Monitore (insbes. Bedingungsvariable) mit Semaphoren nachgebildet werden

Erzeuger/Verbraucher m. Monitor (Pascal-artig, Brinch Hansen)

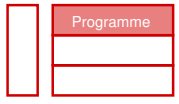
```
monitor ErzeugerVerbraucher
  condition nonfull, nonempty;
  integer count;

  procedure insert(item: integer)
  begin
    if count = N then
      wait(nonfull);
    insertItem(item);
    count := count + 1;
    signal(nonempty);
  end;

  function remove: integer
  begin
    if count = 0 then
      wait(nonempty);
    remove := removeItem();
    count := count - 1;
    signal(nonfull);
  end;

  count = 0;
end monitor;
```

Motivation für Broadcast-Signalisierung (Variante 2)

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

```
procedure insert(item: ..., size: integer)
begin
  while count + size > N do
    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)

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
  }
```

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);
}
```

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)

3.10.2 Java: Synchronisationsklassen ...

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

- Mit diesen Objekten lassen sich Monitore nachbilden:

Monitor

```
monitor Bsp

   condition cond;
   procedure foo
   begin
      if ... then
         wait (cond);
      ...
      signal (cond);

   end;
end monitor;
```

Java−Code

```
public class Bsp {
   Lock mutex;        // = new ReentrantLock();
   Condition cond;    // = mutex.newCondition();
   public void foo() {
      mutex.lock();
      while(...)
         cond.await();
      ...
      cond.signal();
      mutex.unlock();
   }
}
```

Im Konstruktor

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

Anmerkungen ...

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

Thread 1

```
condition = true;
cond.signal();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

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

Thread 1

```
condition = true;
cond.signal();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

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

Thread 1

```
mutex.lock();
condition = true;
cond.signal();
mutex.unlock();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

Anmerkungen ...

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

Thread 1

```
mutex.lock();
condition = true;
cond.signal();
mutex.unlock();
```

Thread 2

```
mutex.lock();
...
while (!condition)
  cond.await();
...
mutex.unlock();
```

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*

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
  while (count == buffer.getSize())      // Puffer voll?
    nonfull.await();                     //   ja: warten...
  buffer.insertItem(item);               // Item eintragen
  count++;
  nonempty.signal();                     // ggf. Thread wecken
  mutex.unlock();                        // Mutex freigeben
}
```

Beispiel: Erzeuger/Verbraucher-Problem ...

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

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 Schreibsperren können rekursiv belegt werden
  - Schreiber kann auch noch Lesesperre belegen, aber nicht umgekehrt
  - für Schreibsperre kann auch `Condition` erzeugt werden
  - im Konstruktor kann Fairness eingeschaltet werden

Mutex Variablen

- Verhälten ähnlich zu Lava `Lock` (binäres Semaphor)
  - Zustände: frei, gesperrt; bei Erzeugung: frei
- Deklaration (und Initialisierung):
  `std::mutex mutex;`
- Zum Sperren des Mutex: Erzeugung eines Objeks 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`

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()`

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!
 }
```

Beispiel: Nachbildung eines Monitors ...

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

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

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)`

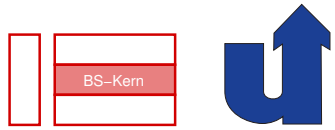
Beispiel: Nachbildung eines Monitors

```
#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;
}
```

3.10.5 Synchronisationsunterstützung in Linux



- 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 Benutzeradreßraum
  - `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

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

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:

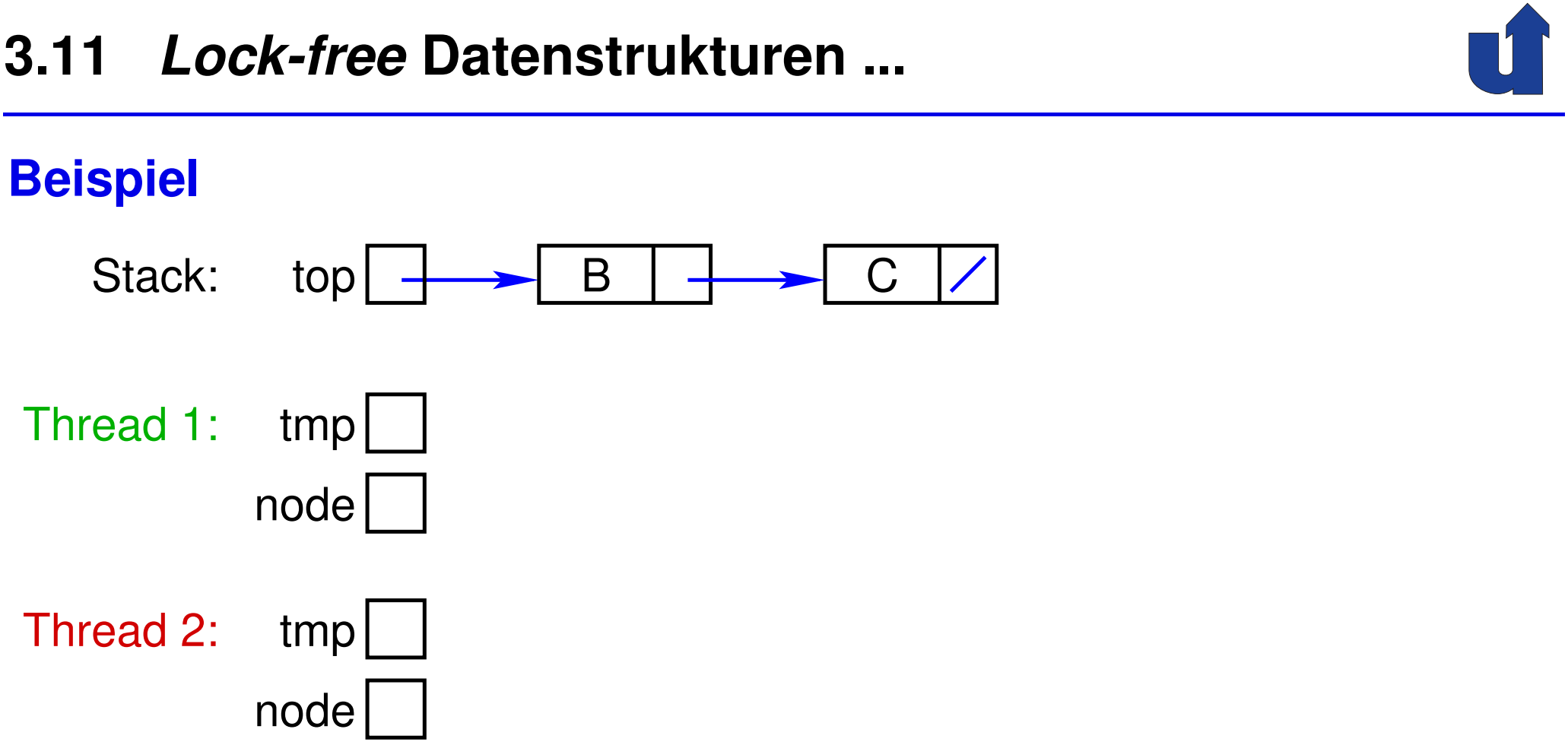
```
bool compareAndSet(T &addr, T expect, T newval) {
  if (addr == expect) {
    addr = newval;
    return true;
  }
  return false;
}
```

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;
}
```



Beispiel

Stack: top → B → C

Thread 1: tmp

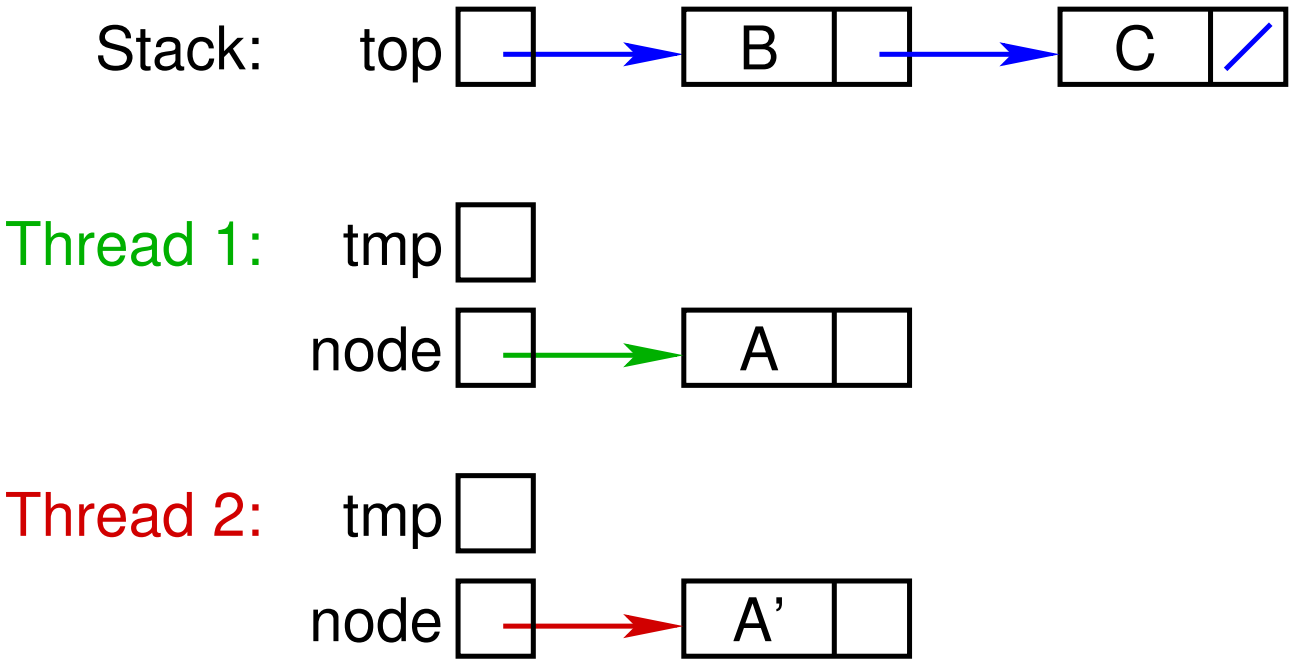
node

Thread 2: tmp

node

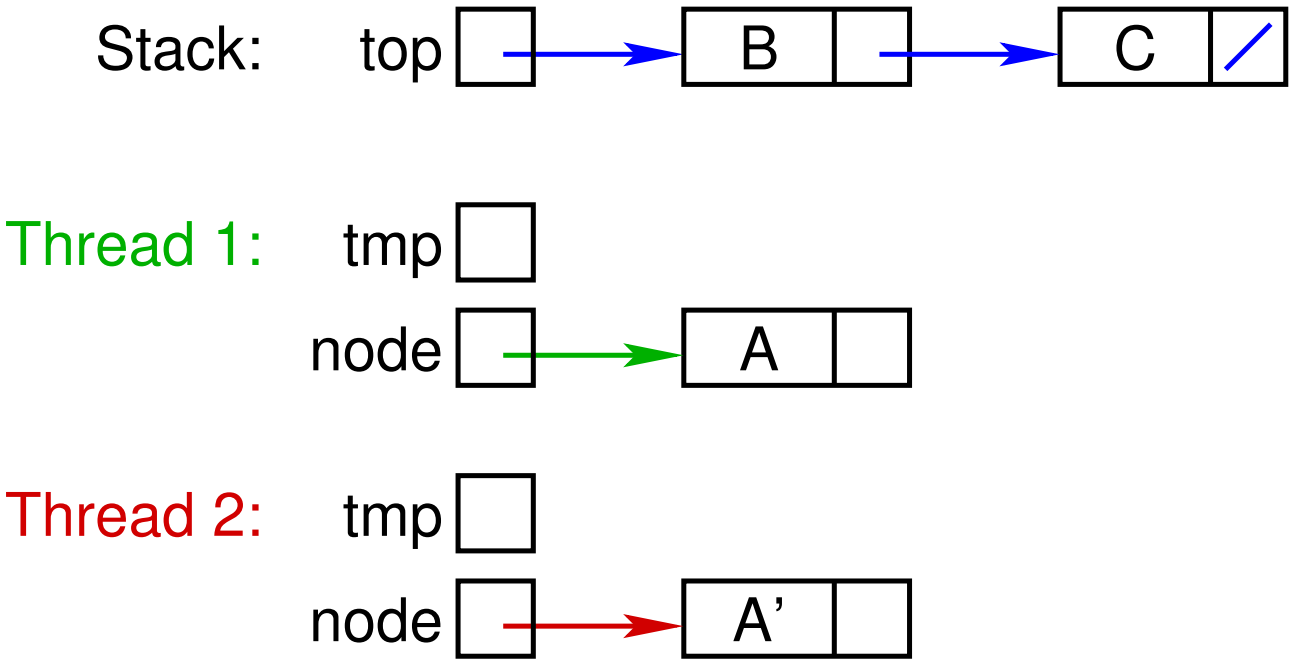
```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



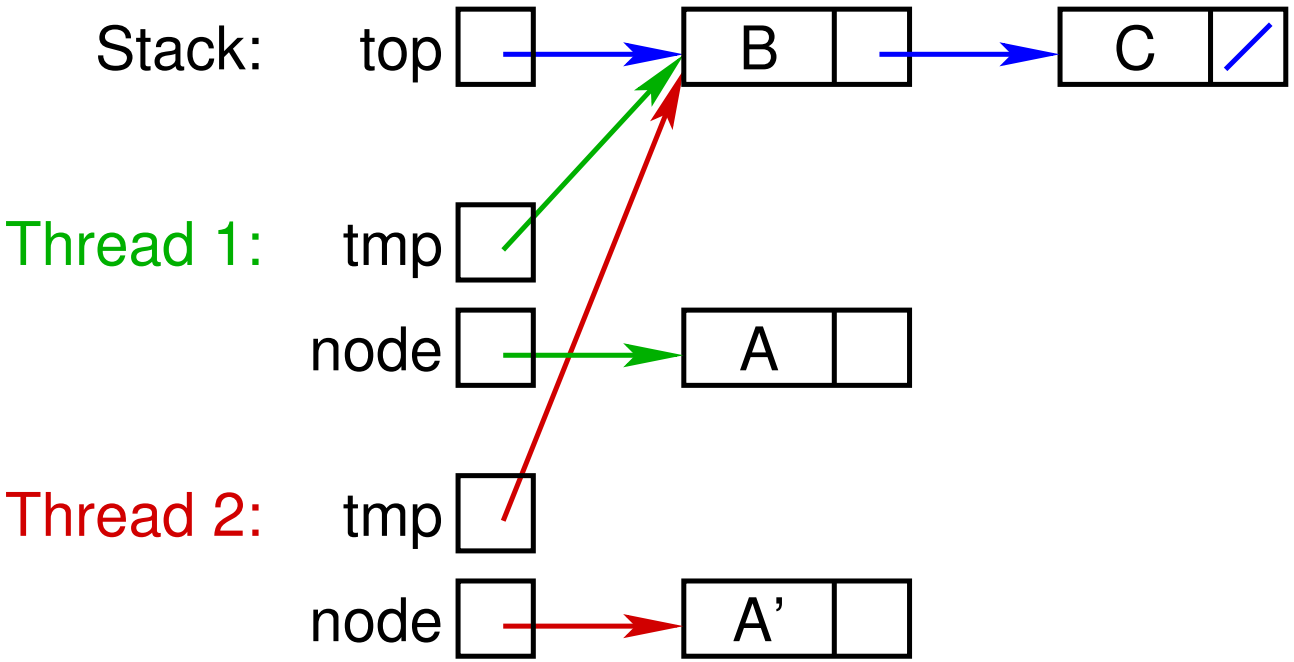

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



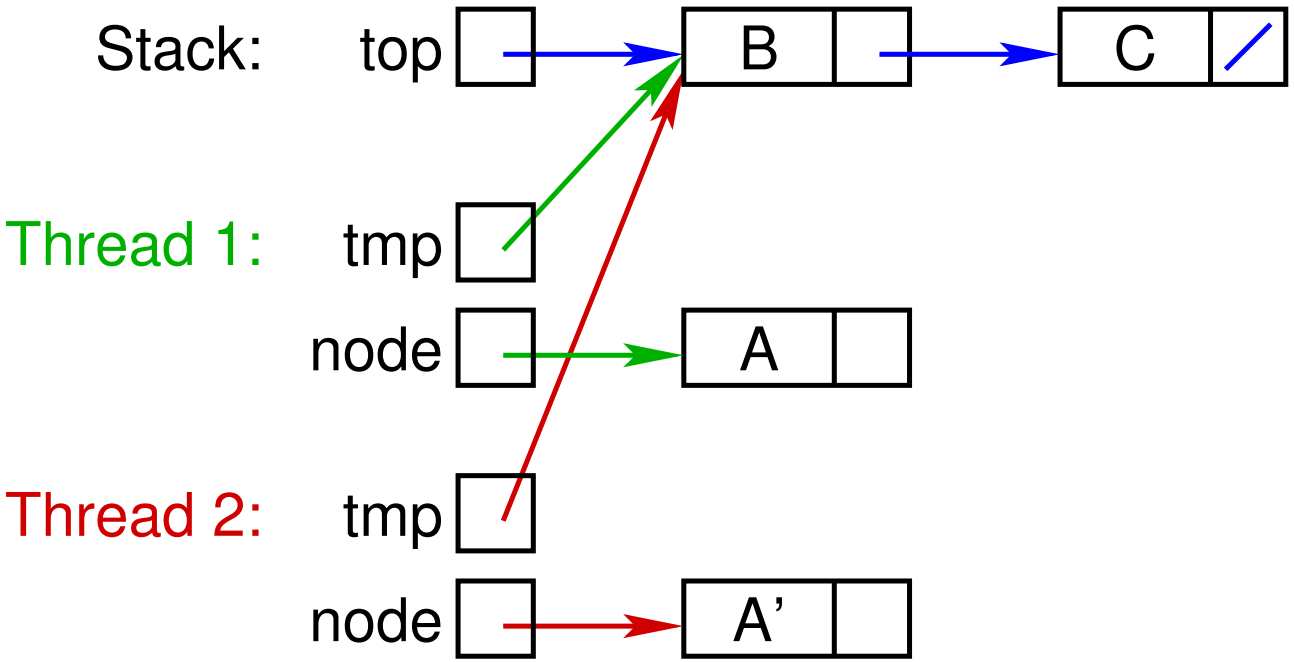

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



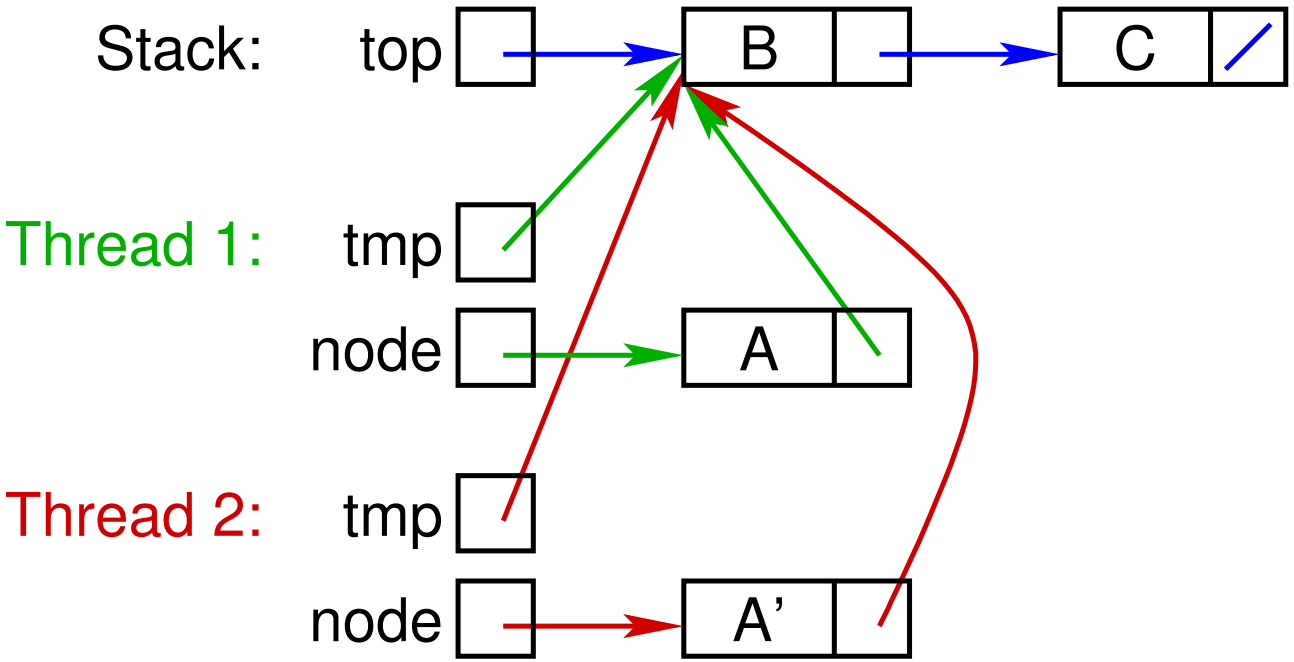

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



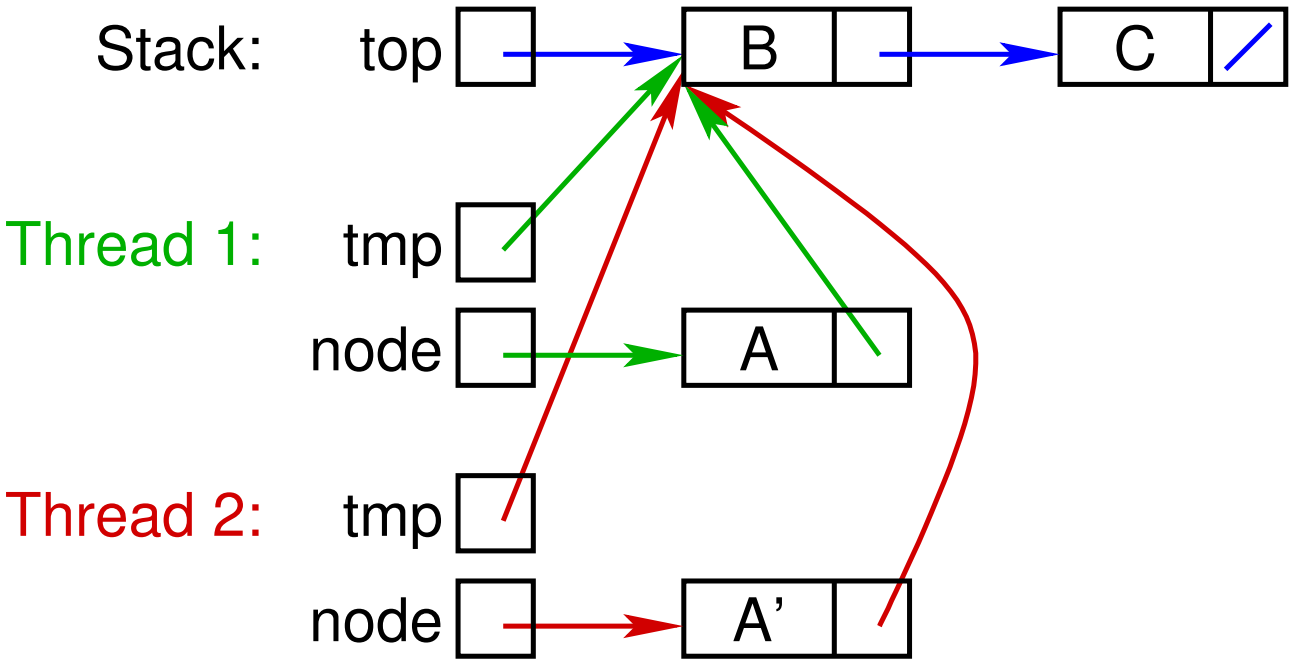

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



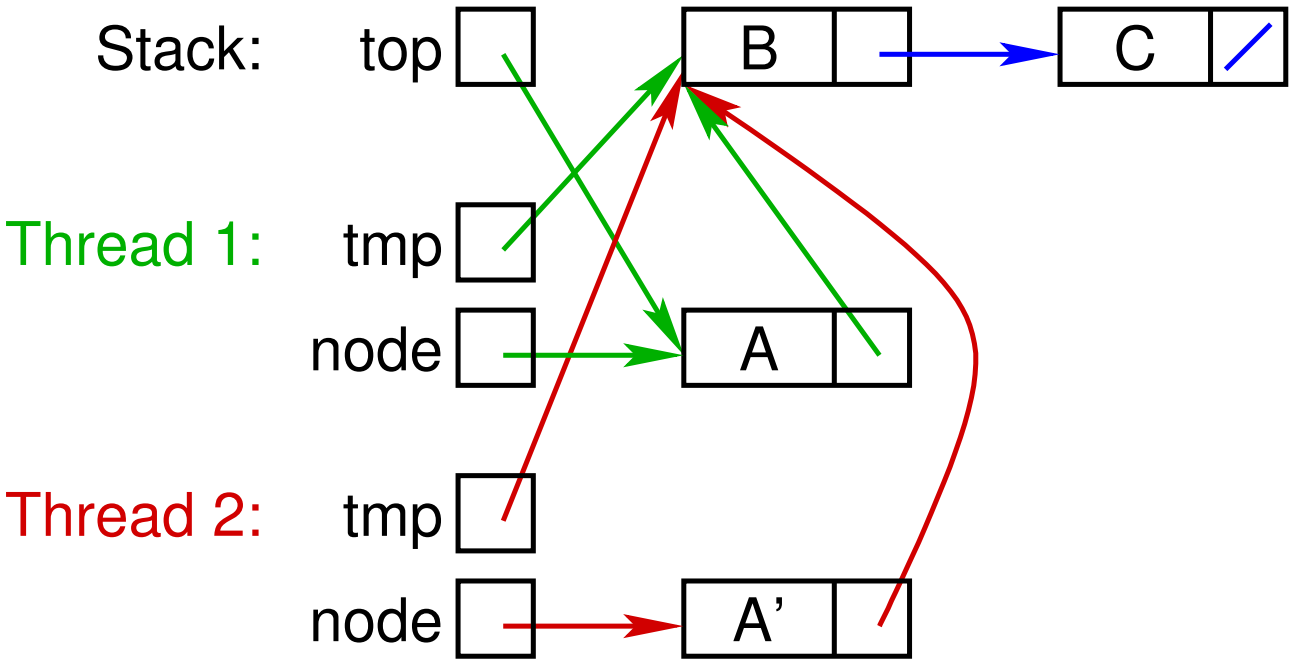

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



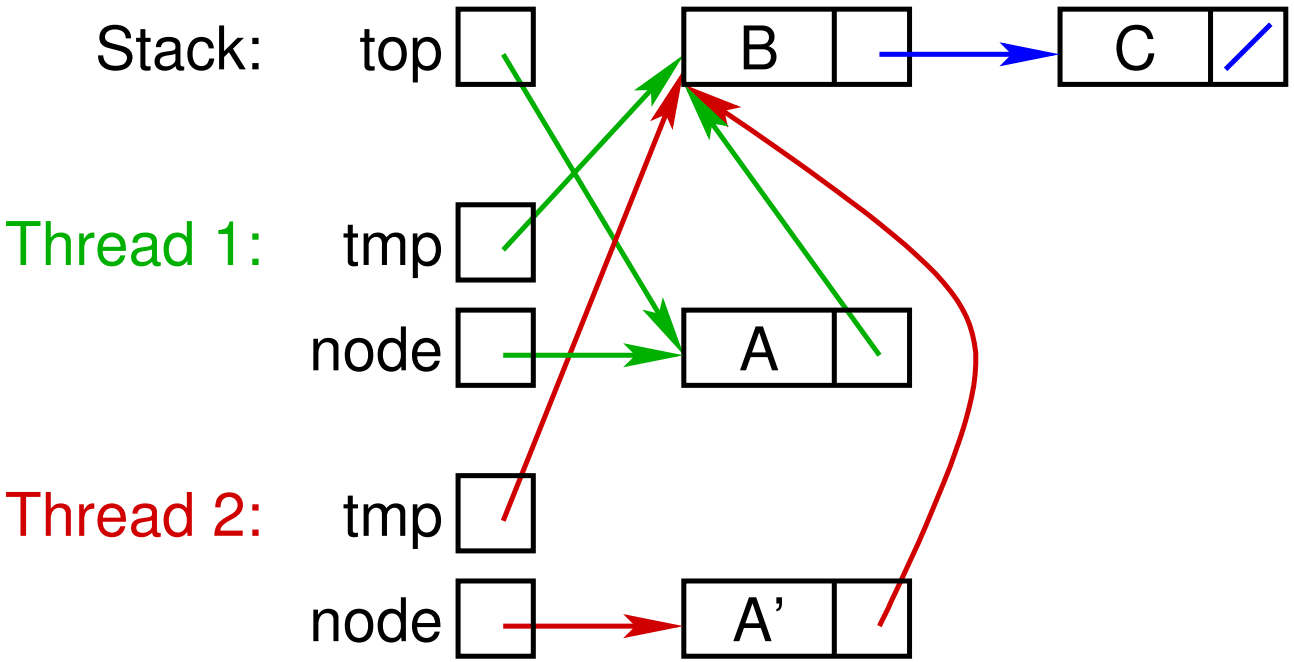

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



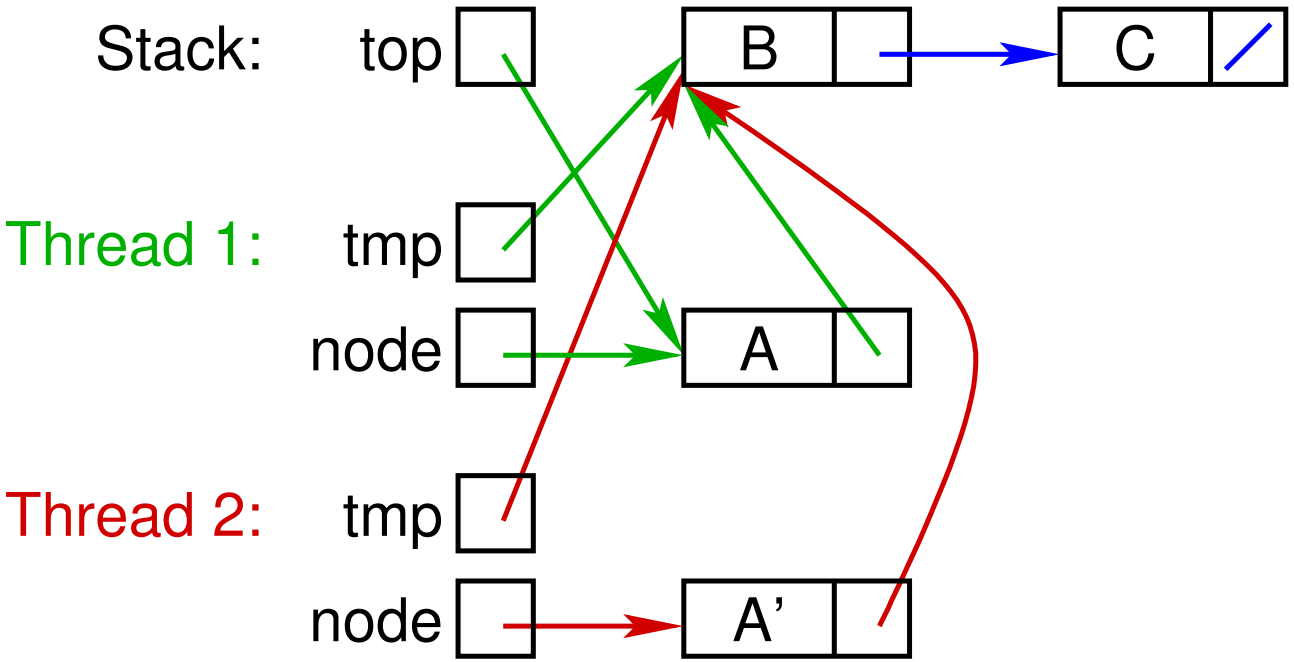

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



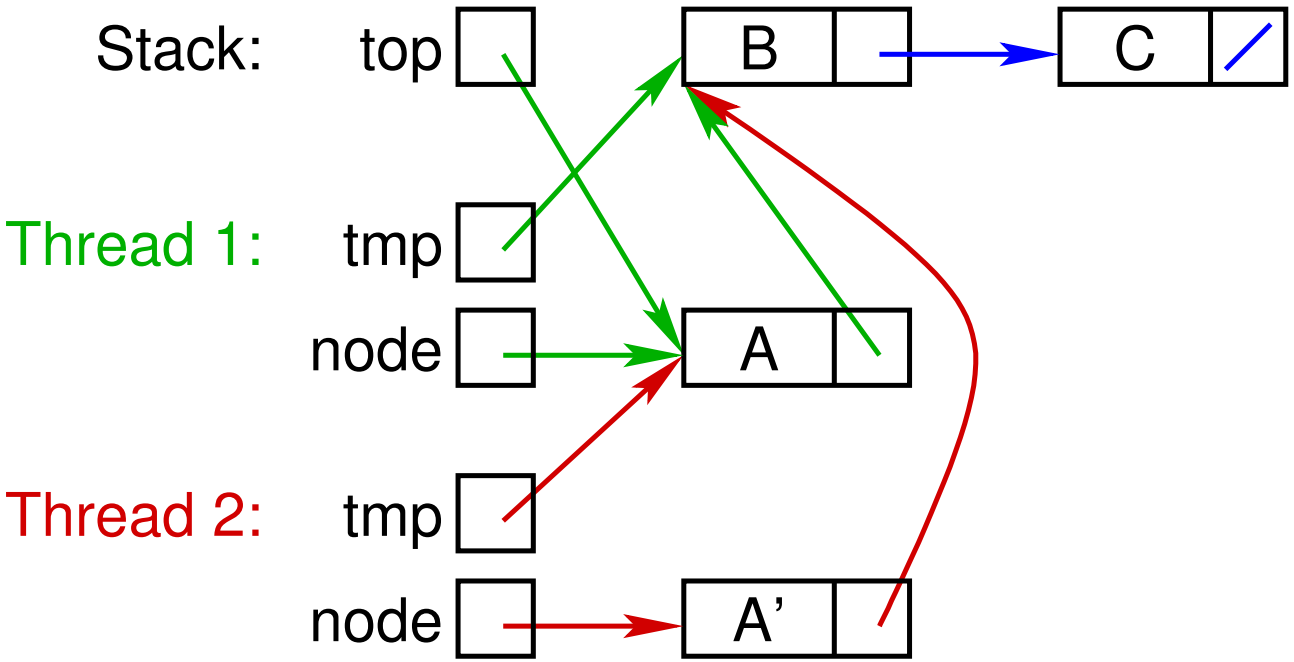

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



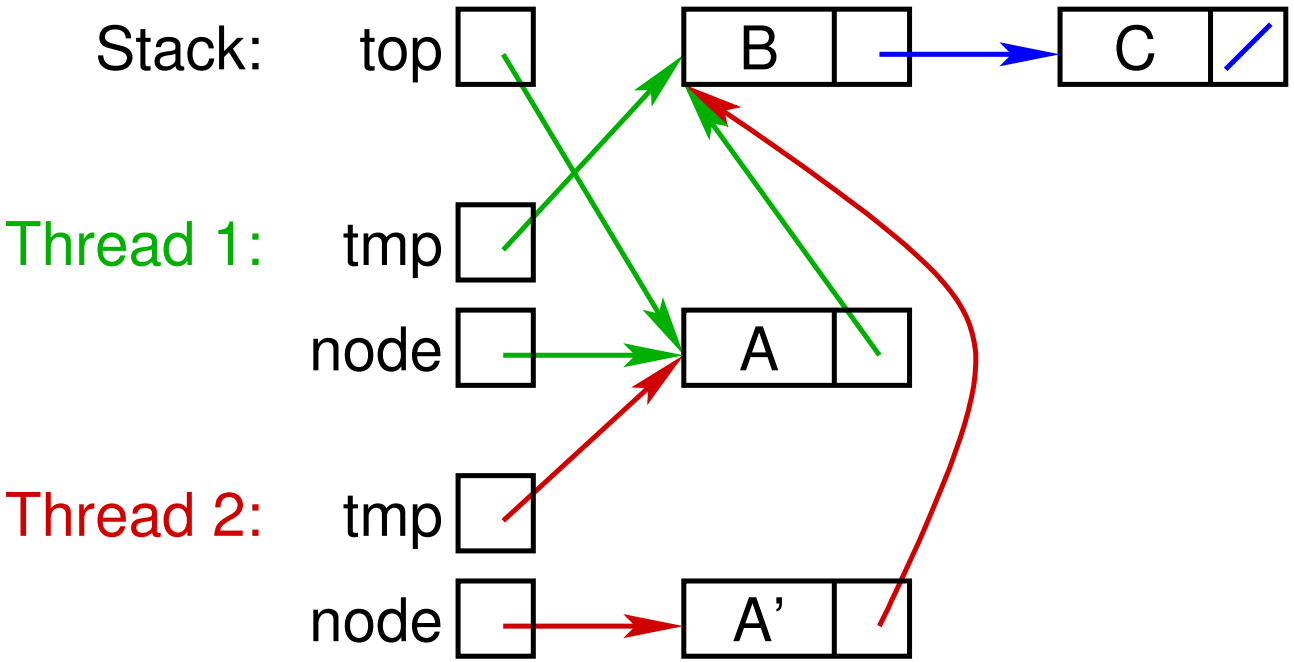

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



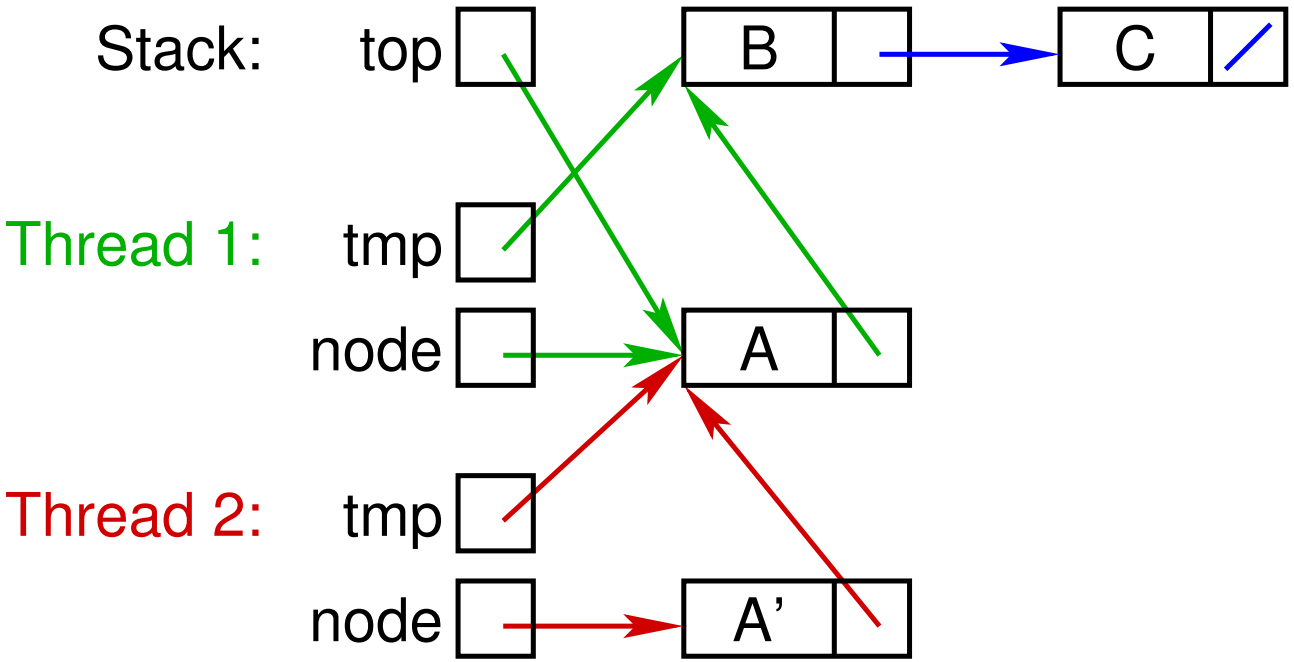

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



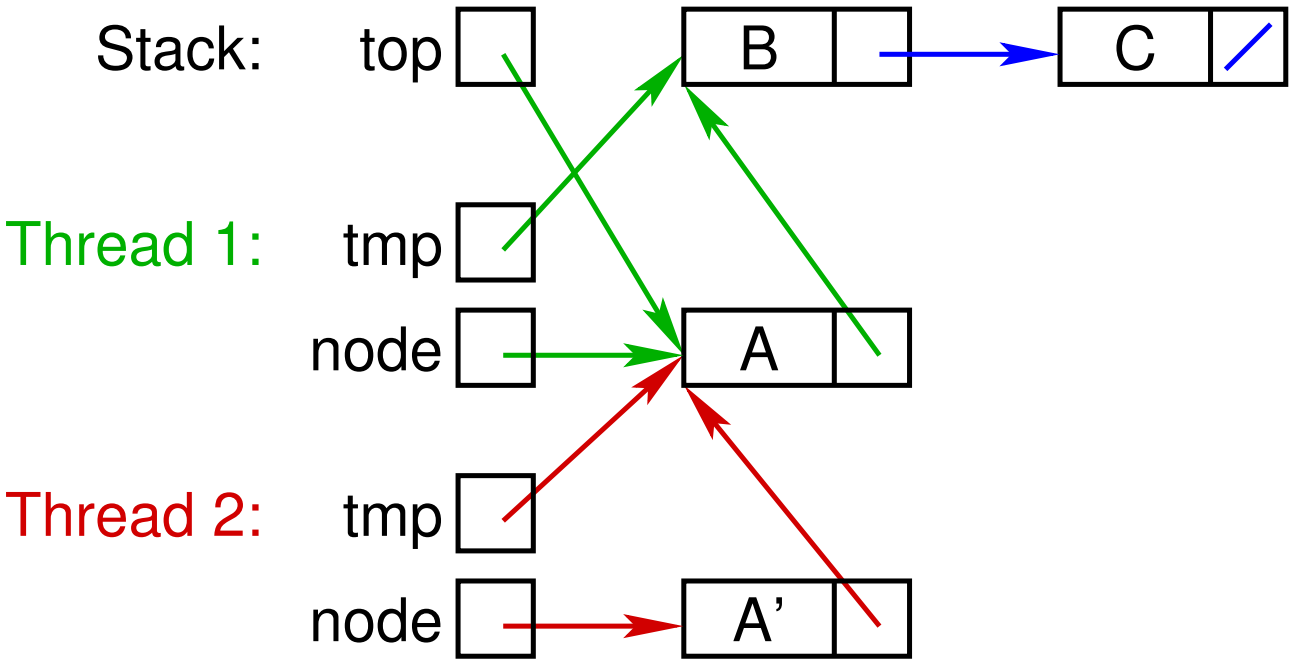

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel



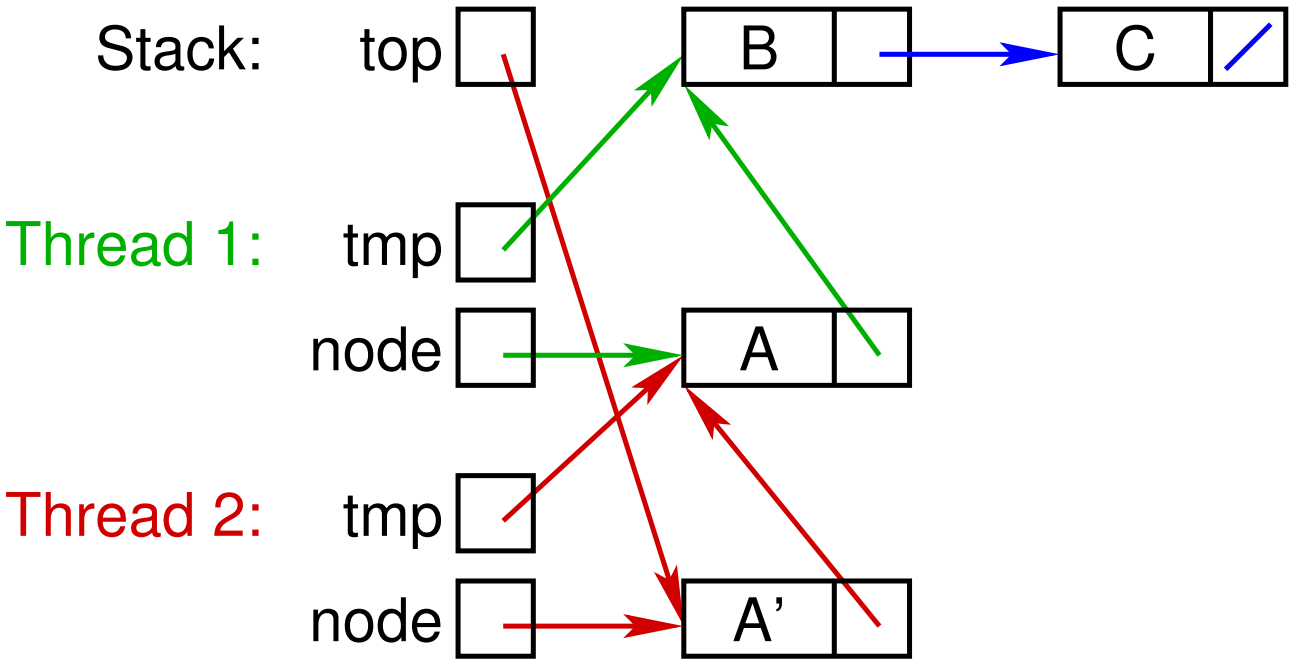

```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

Beispiel




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

Beispiel

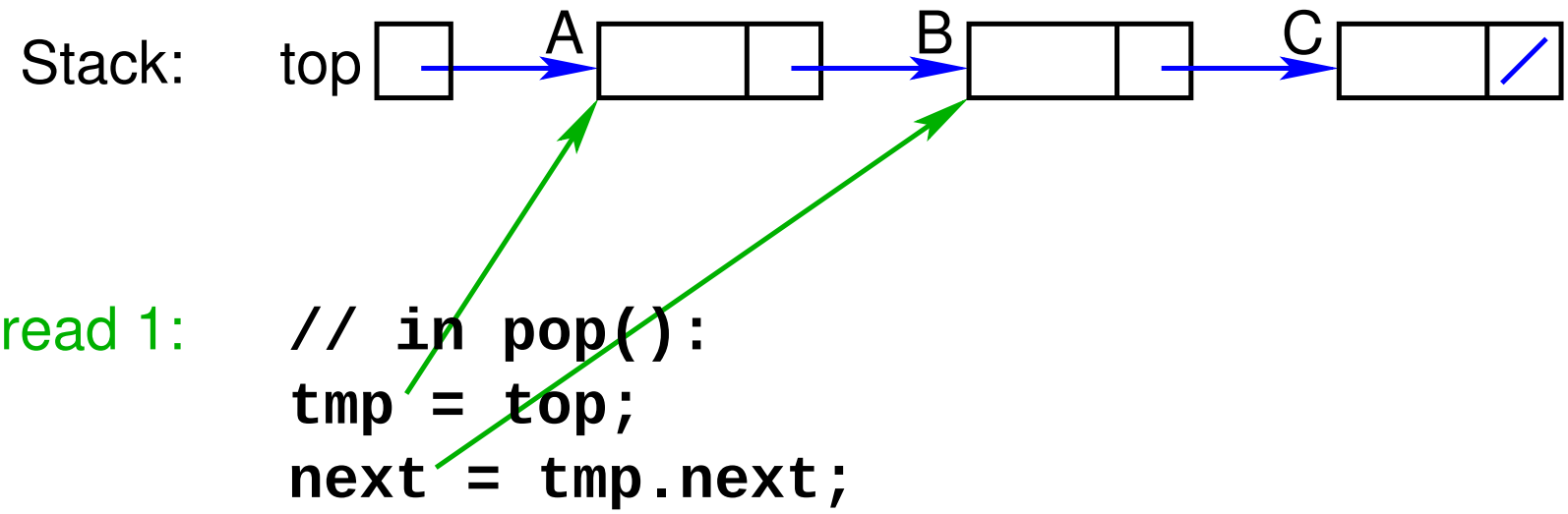



```
Node node = new Node(data);
do{
   tmp = top;
   node.next = tmp;
} while (!compareAndSet(top, tmp, node);
```

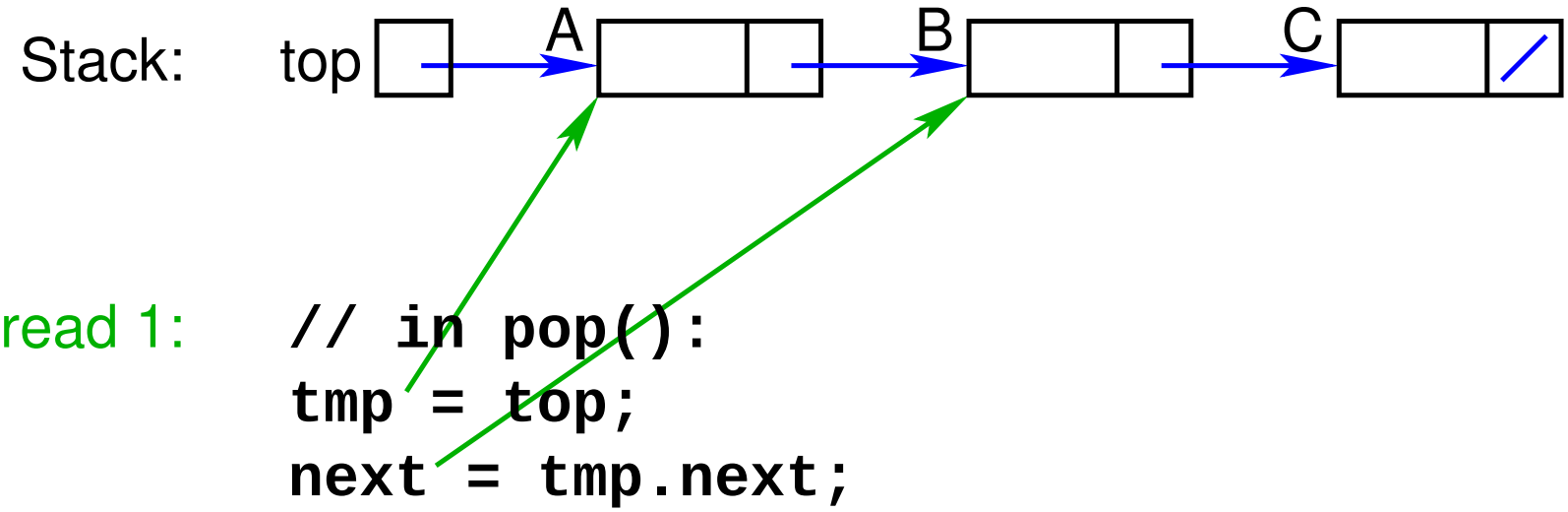
Das ABA-Problem

- *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

Das ABA Problem beim Treiber-Stack



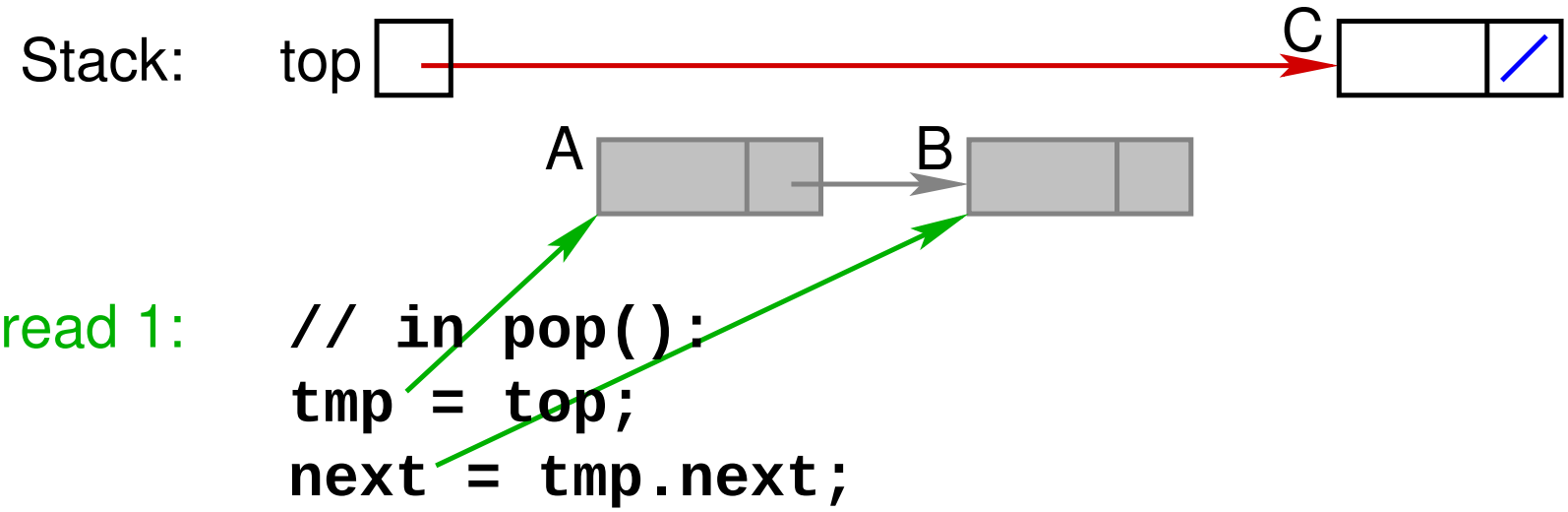
Das ABA Problem beim Treiber-Stack



Thread 2:

```
a = pop(); // gibt Speicherblock A frei
b = pop(); // gibt Speicherblock B frei
```

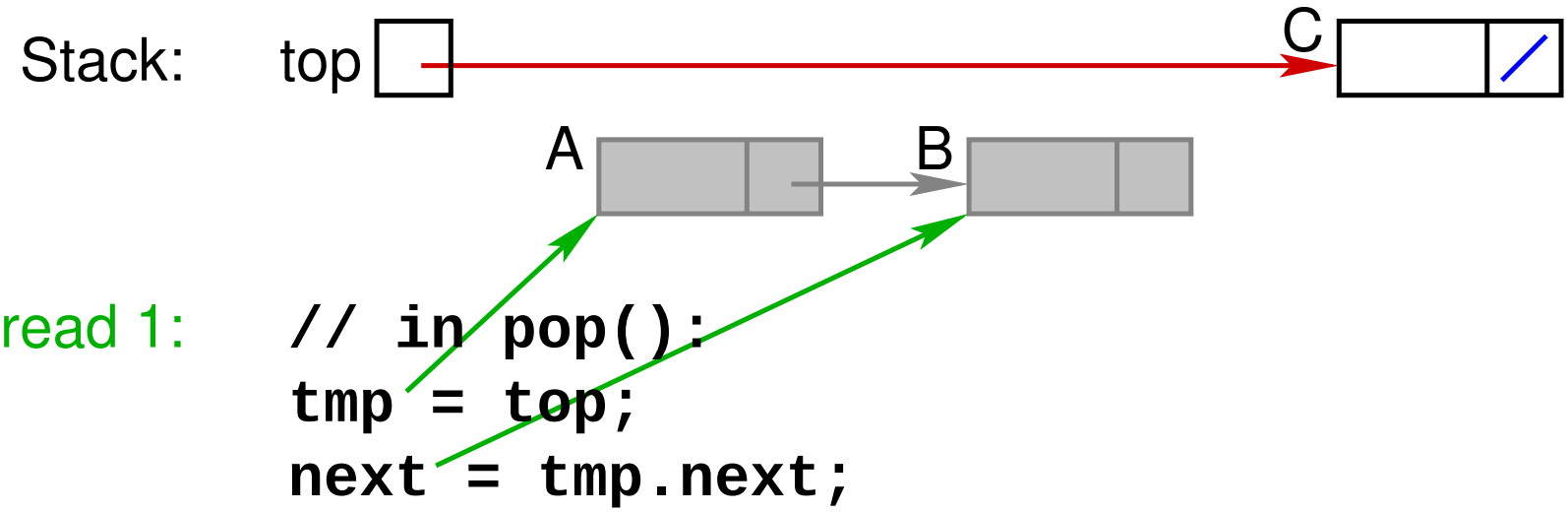
Das ABA Problem beim Treiber-Stack



Thread 2:

```
a = pop(); // gibt Speicherblock A frei
b = pop(); // gibt Speicherblock B frei
```

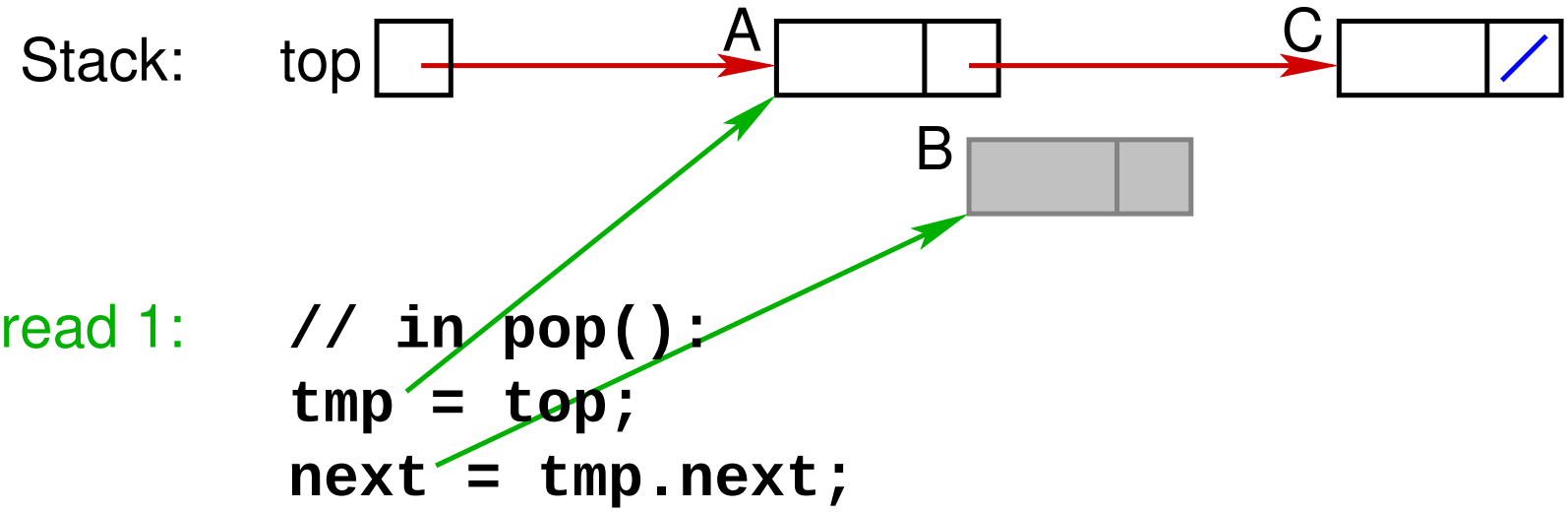
Das ABA Problem beim Treiber-Stack



Thread 2:

```
a = pop();  // gibt Speicherblock A frei
b = pop();  // gibt Speicherblock B frei
push(x);    // Allokiert Speicherblock
            // an Adresse A
```

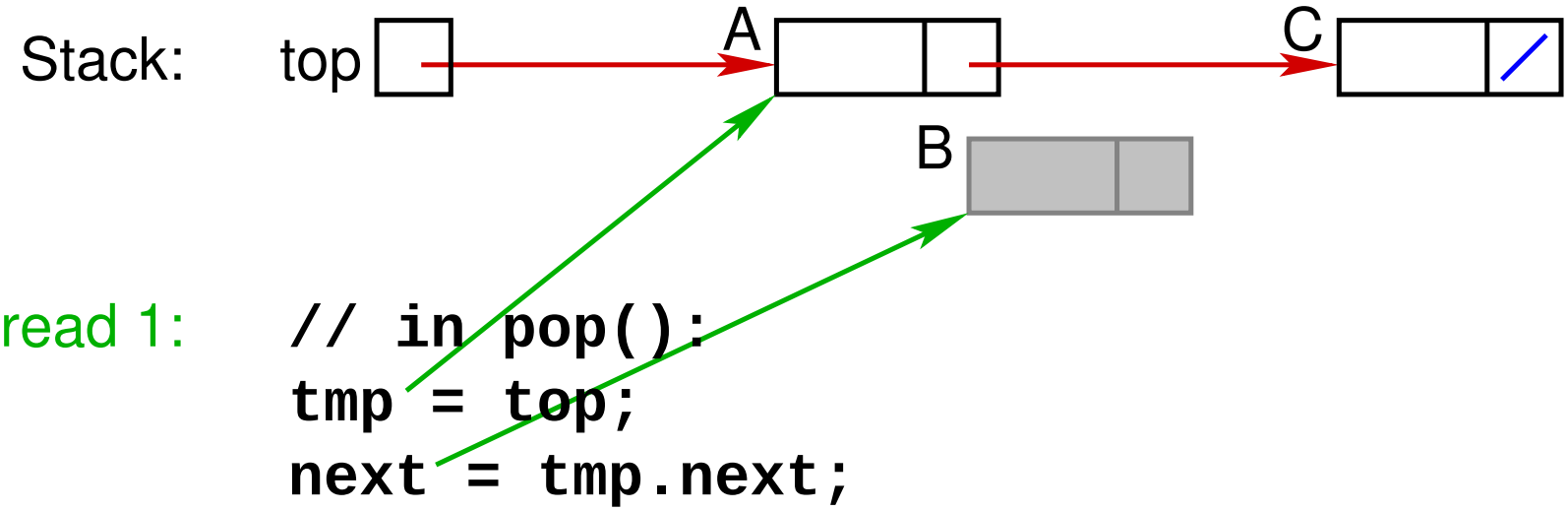
Das ABA Problem beim Treiber-Stack



Thread 2:

```
a = pop();  // gibt Speicherblock A frei
b = pop();  // gibt Speicherblock B frei
push(x);    // Allokiert Speicherblock
            // an Adresse A
```

Das ABA Problem beim Treiber-Stack



Thread 1:

```
// in pop():
tmp = top;
next = tmp.next;
```

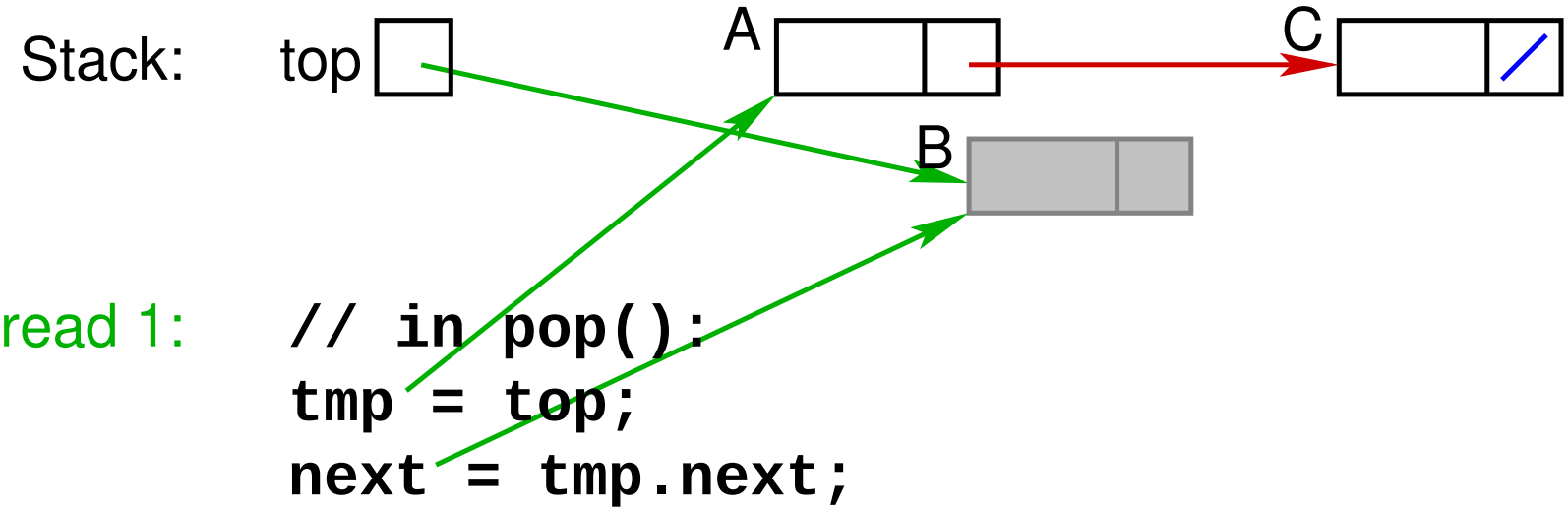
Thread 2:

```
a = pop();  // gibt Speicherblock A frei
b = pop();  // gibt Speicherblock B frei
push(x);    // Allokiert Speicherblock
            // an Adresse A
```

Thread 1:

```
while (!compareAndSet(top, tmp, next));
```

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, liblfds
- 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_...()`

3.12 *Transactional Memory*

- 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 durchgeführten Transaktionen entspricht immer einer (beliebigen) sequentiellen Ausführung

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

Beipiel: 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());
}
```

Beipiel: 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

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)

3.13 Zusammenfassung / Wiederholung ...

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

Thread 0	Thread 1
`P(Mutex);`	`P(Mutex);`
`// kritischer Abschnitt`	`// kritischer Abschnitt`
`V(Mutex);`	`V(Mutex);`

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

3.13 Zusammenfassung / Wiederholung ...

- 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 sofot den Monitor?
      - falls nicht: Bedingung nach Rückkehr aus `wait()` erneut prüfen!

3.13 Zusammenfassung / Wiederholung ...

- 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