DUALE HOCHSCHULE SH

Synchronisation

Computing-Plattformen und Netzwerke

auf Basis der Unterlagen von
Prof. P. Mandl (HS München)

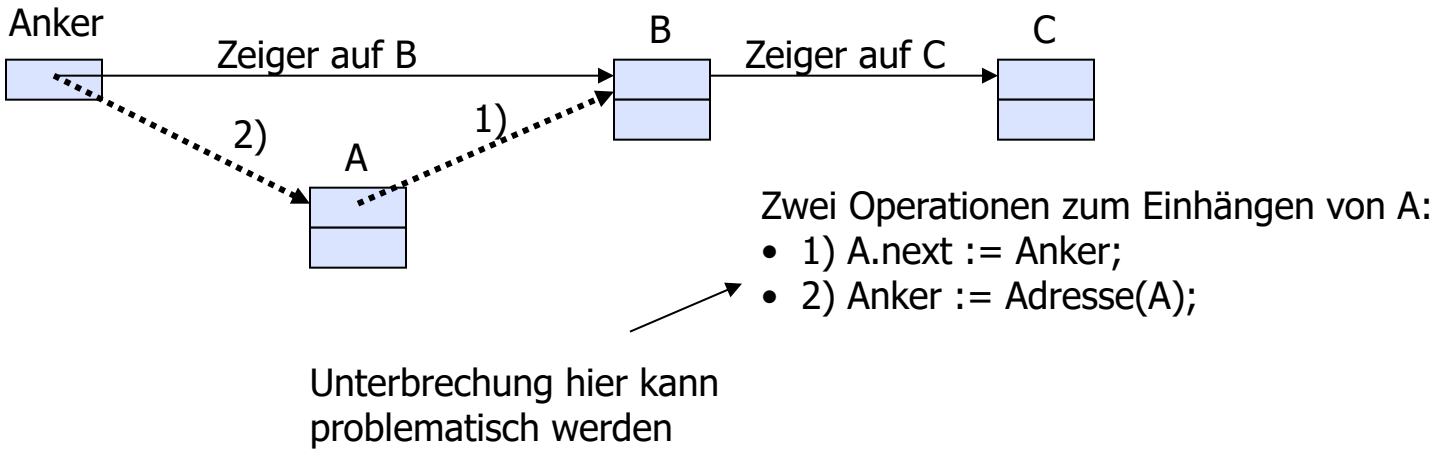
Grundlagen der Synchronisation

Einführung

- Nebenläufigkeit
  - Parallele oder quasi-parallele Ausführung von Befehlsfolgen in Prozessen und Threads
  - Verdrängung jederzeit durch das Betriebssystem möglich ohne Einfluss des Anwendungsprogrammierers
- Atomare Aktionen
  - Codebereiche, die in einem Stück, also atomar, ausgeführt werden müssen, um Inkonsistenzen zu vermeiden
  - Aber: Eine Unterbrechung durch Verdrängung ist jederzeit möglich

Konflikte, Fallbeispiel 1 (1)

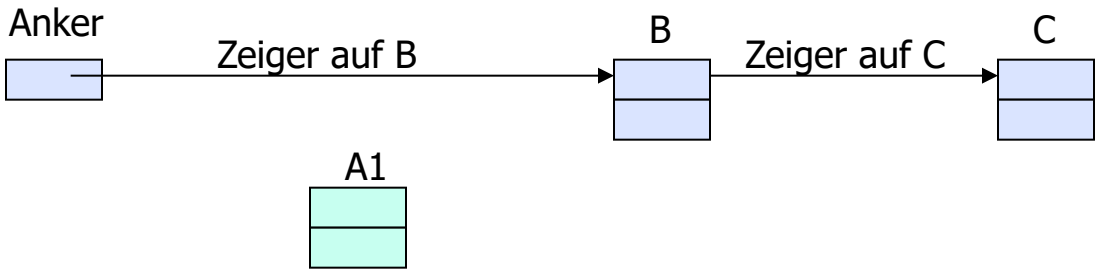
- Beispiel zur Verdeutlichung des Problems:
  - Mehrere Prozesse bearbeiten eine gemeinsame Liste von Objekten (z. B. die Prozesslisten der Run-Queue)
  - Ein Prozess hängt ein neues Objekt vorne in die Liste ein
    - Prozesse können zu beliebigen Zeiten unterbrochen werden
    - Versucht ein zweiter Prozess auch, ein Objekt vorne anzuhängen, gibt es möglicherweise Probleme



Konflikte, Fallbeispiel 1 (2)

- Prozess 1 führt 1. Anweisung aus
  - A1.next := Anker
  - Jetzt wird die CPU entzogen

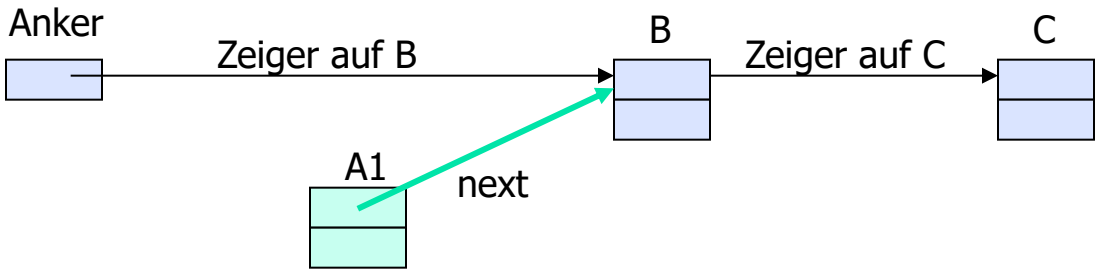
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- 2) Anker := Adresse(A);



Konflikte, Fallbeispiel 1 (2)

- Prozess 1 führt 1. Anweisung aus
  - A1.next := Anker
  - Jetzt wird die CPU entzogen

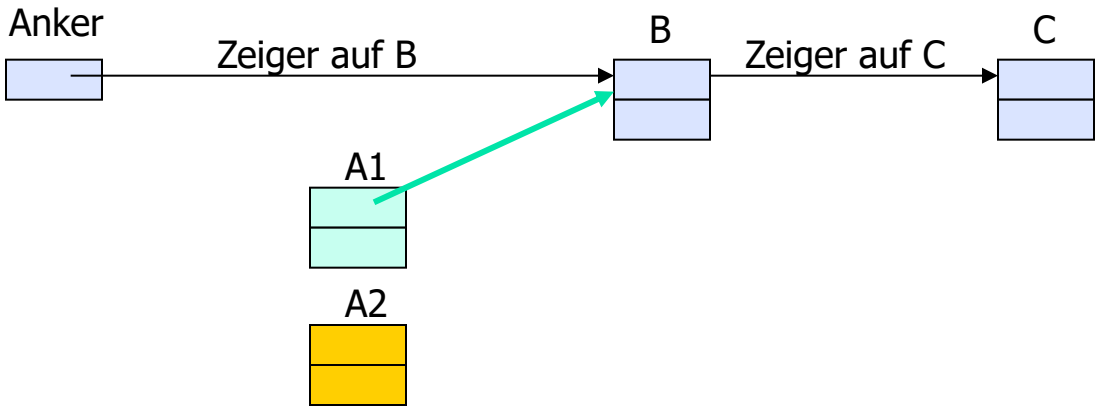
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- 2) Anker := Adresse(A);



Konflikte, Fallbeispiel 1 (3)

- Prozess 2 führt 1. und 2. Anweisung aus
  - A2.next := Anker
  - Anker := Adresse (A2)
  - Jetzt wird die CPU entzogen

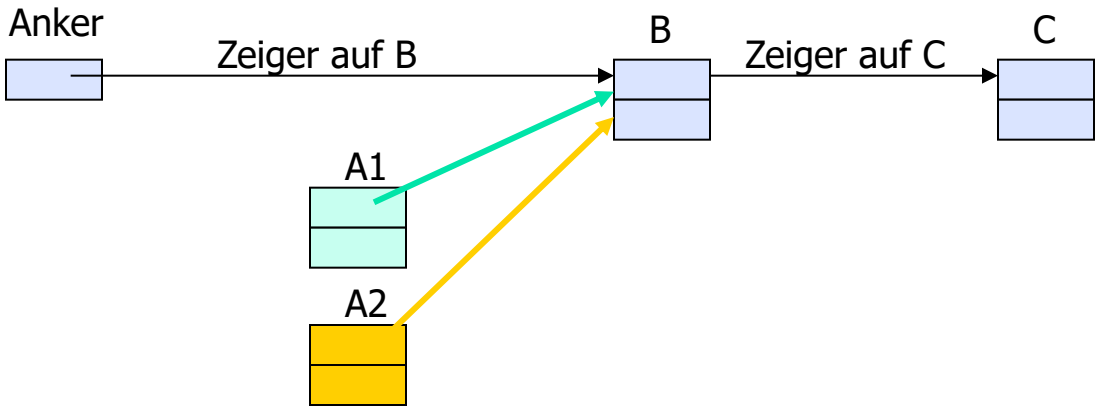
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- 2) Anker := Adresse(A);



Konflikte, Fallbeispiel 1 (3)

- Prozess 2 führt 1. und 2. Anweisung aus
  - A2.next := Anker
  - Anker := Adresse (A2)
  - Jetzt wird die CPU entzogen

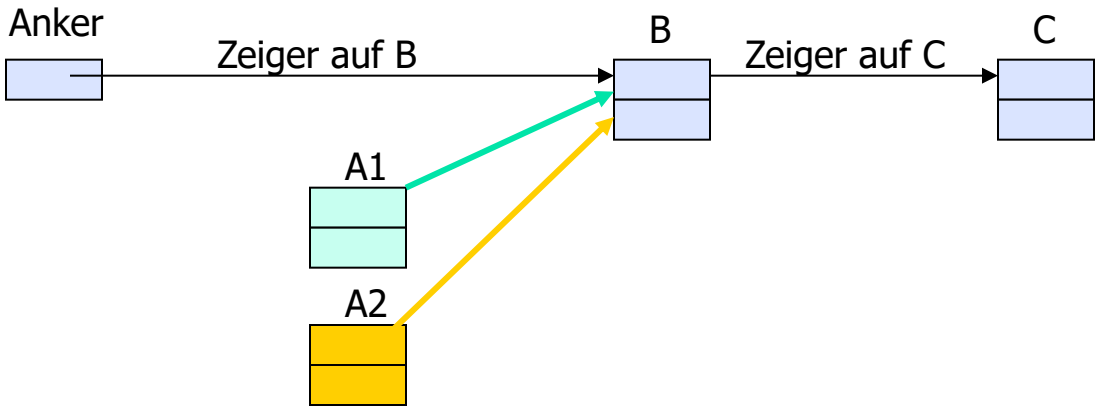
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- 2) Anker := Adresse(A);



Konflikte, Fallbeispiel 1 (3)

- Prozess 2 führt 1. und 2. Anweisung aus
  - A2.next := Anker
  - Anker := Adresse (A2)
  - Jetzt wird die CPU entzogen

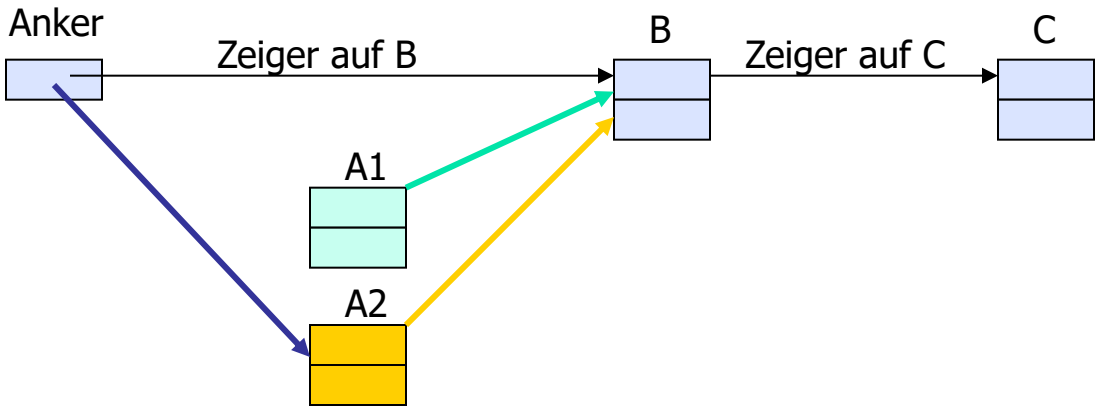
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- **2) Anker := Adresse(A);**



Konflikte, Fallbeispiel 1 (3)

- Prozess 2 führt 1. und 2. Anweisung aus
  - A2.next := Anker
  - Anker := Adresse (A2)
  - Jetzt wird die CPU entzogen

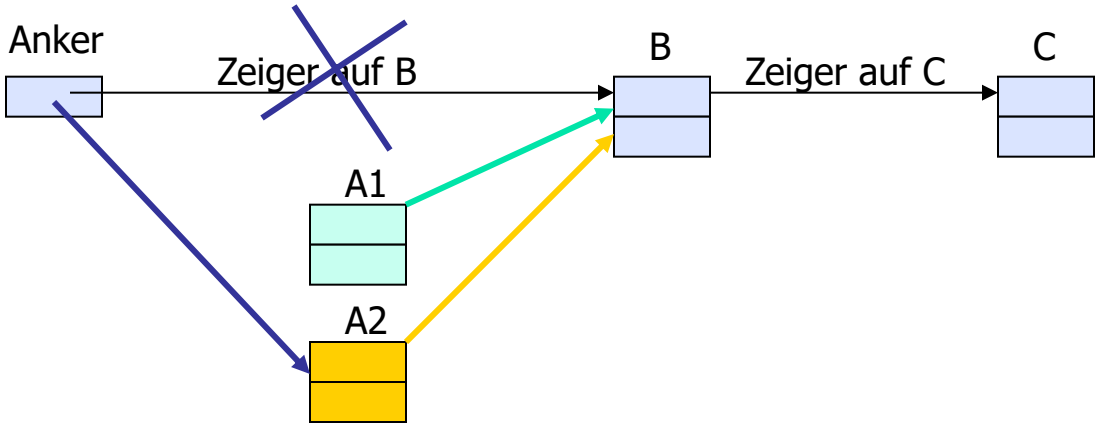
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- **2) Anker := Adresse(A);**



Konflikte, Fallbeispiel 1 (3)

- Prozess 2 führt 1. und 2. Anweisung aus
  - A2.next := Anker
  - Anker := Adresse (A2)
  - Jetzt wird die CPU entzogen

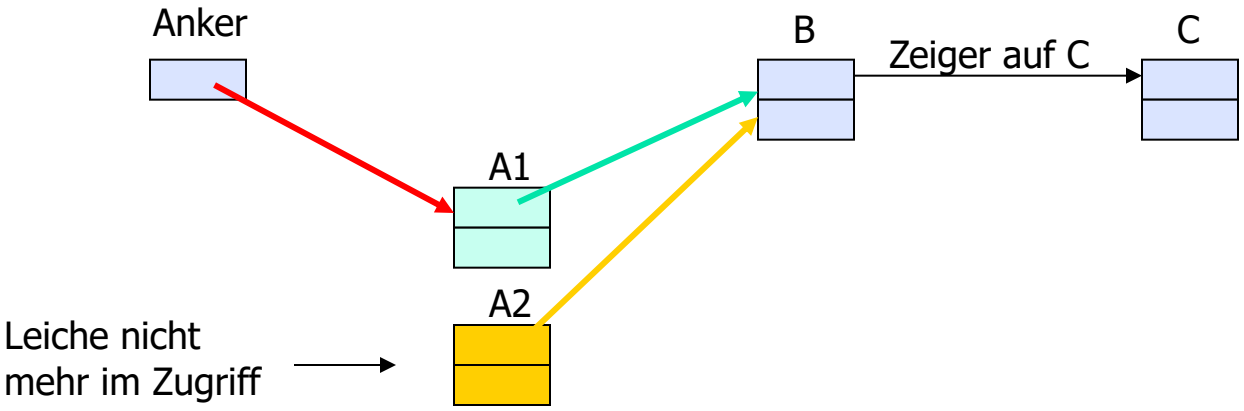
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- **2) Anker := Adresse(A);**



Konflikte, Fallbeispiel 1 (4)

- Prozess 1 führt die 2. Anweisung aus
  - Anker := Adresse (A1)
  - A2 wird zur Leiche

Zwei Operationen zum Einhängen:
- 1) A.next := Anker;
- **2) Anker := Adresse(A);**

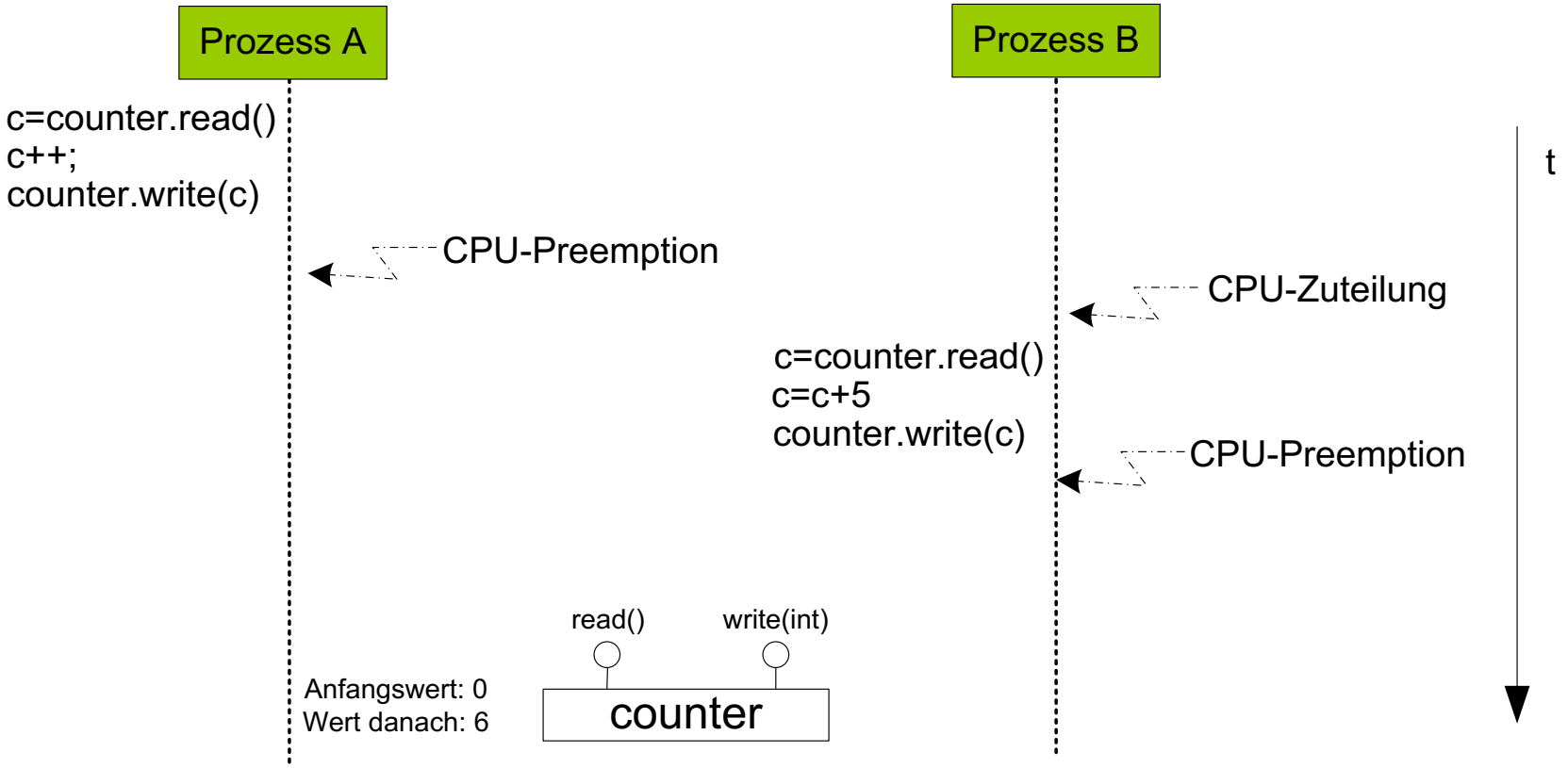


Konflikte, Fallbeispiel 2 (1)

- Ein gemeinsam genutzter Zähler (Counter) wird von zwei Prozessen verändert
- Auch hier kann es zu Inkonsistenzen kommen, die als Lost-Update bezeichnet werden

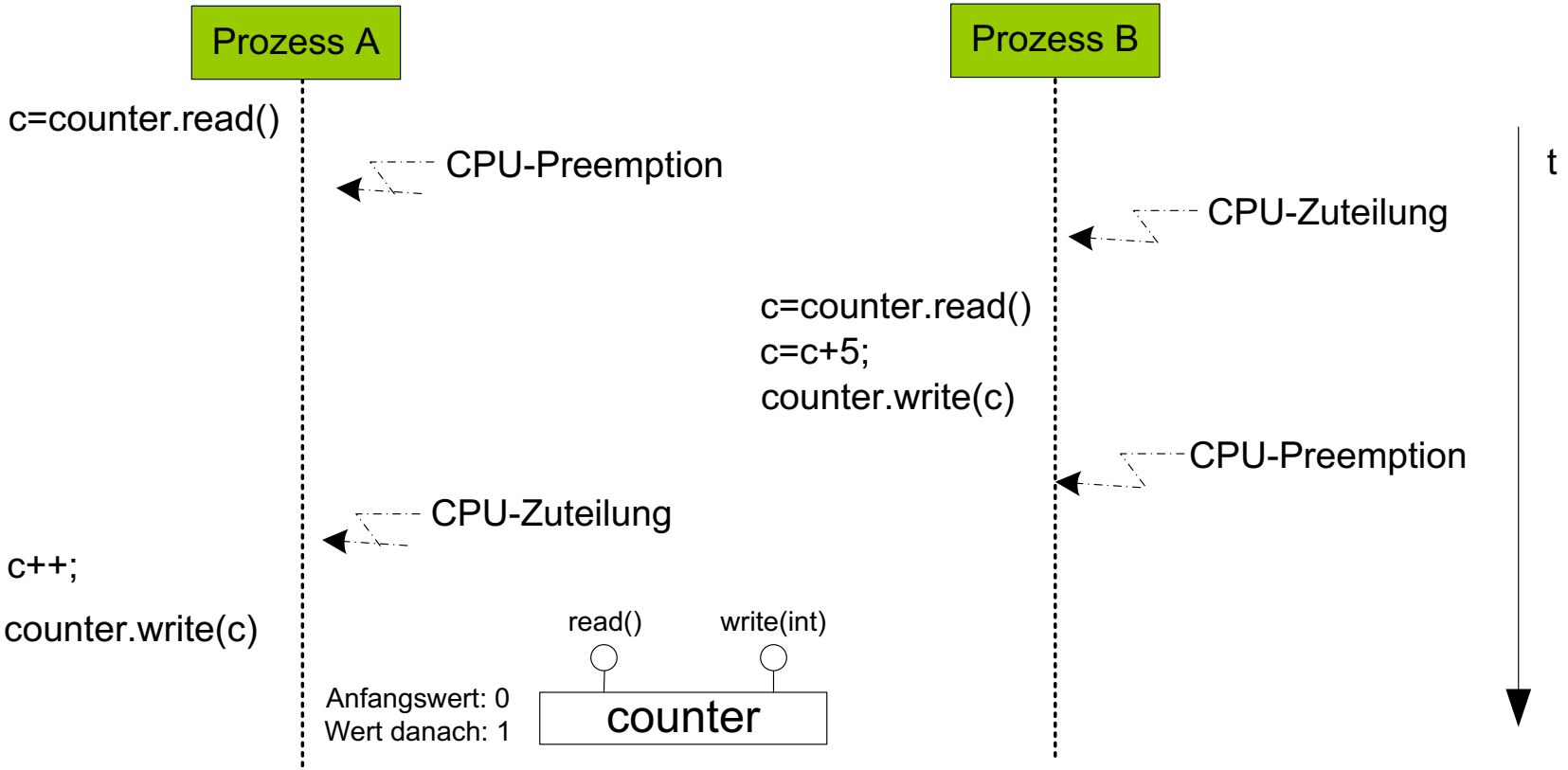
Konflikte, Fallbeispiel 2 (2)

- Counter steht anfangs auf 0
- **Unproblematischer** Ablauf



Konflikte, Fallbeispiel 2 (3)

- **Fehlerfall**: Was passiert bei diesem Ablauf, wenn counter zunächst auf 0 steht?



Race Conditions

- Die gezeigten Situationen bezeichnet man auch als **Race Conditions**
  - Zwei oder mehrere Prozesse oder Threads nutzen ein gemeinsames Betriebsmittel (Liste, Counter,...)
  - Endergebnisse der Bearbeitung sind von der zeitlichen Reihenfolge abhängig

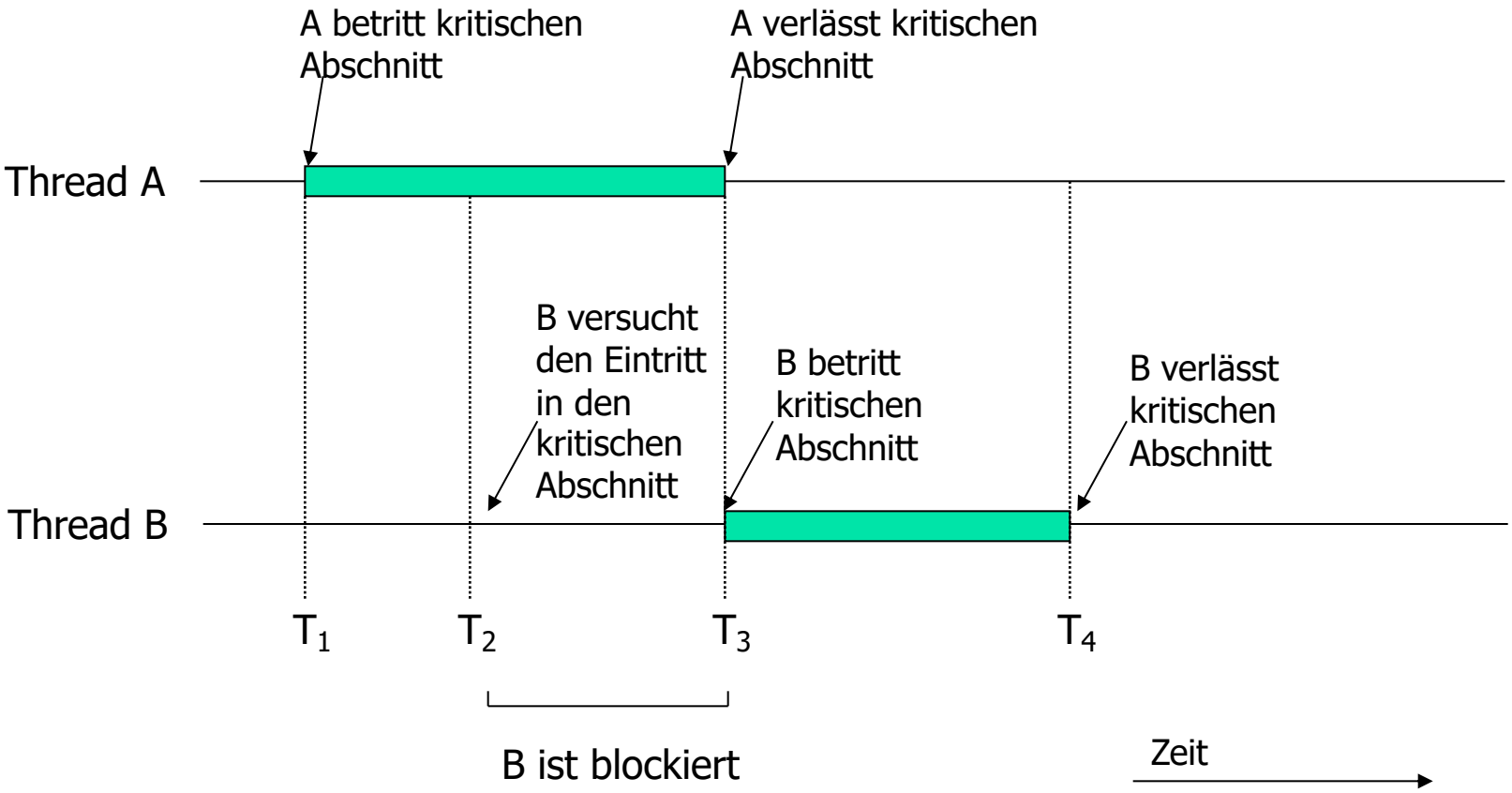
Kritische Abschnitte und gegenseitiger Ausschluss

- Auf gemeinsam von mehreren Prozessen oder Threads bearbeitete Daten darf nicht beliebig zugegriffen werden
  - Prozesse bzw. Threads müssen sich zur Bearbeitung gemeinsamer (shared) Ressourcen miteinander **koordinieren**
  - **Synchronisation** erforderlich

Kritische Abschnitte und gegenseitiger Ausschluss

- Man benötigt ein Konzept, das es ermöglicht, gewisse Arbeiten **logisch nicht unterbrechbar** zu machen
  - Die Codeabschnitte, die nicht unterbrochen werden dürfen, werden auch als **kritische Abschnitte** (critical sections) bezeichnet
  - In einem kritischen Abschnitt darf sich immer nur ein Prozess zu einer Zeit befinden
  - Das Betreten und Verlassen eines kritischen Abschnitts muss unter den Beteiligten abgestimmt (synchronisiert) werden
- Ziel: **Gegenseitigen Ausschluss** (mutual exclusion) garantieren

Kritische Abschnitte und gegenseitiger Ausschluss



Siehe Tanenbaum, Andrew, S.; Bos, Herbert: *Moderne Betriebssysteme*, 4. aktualisierte Auflage, Pearson Studium, 2016.

Anforderungen an kritische Abschnitte

- Kriterien von Dijkstra (1965):
  - **Keine zwei Prozesse/Threads dürfen gleichzeitig** in einem kritischen Abschnitt sein (mutual exclusion)
  - **Keine Annahmen über die Abarbeitungsgeschwindigkeit** und die Anzahl der Prozesse/Threads bzw. der verfügbaren Prozessoren
  - Kein Prozess/Thread außerhalb eines kritischen Abschnitts darf einen anderen Prozess/Thread **blockieren**
  - Jeder Prozess/Thread, der am Eingang eines kritischen Abschnitts wartet, muss ihn irgendwann betreten dürfen
    → **kein ewiges Warten** (fairness condition)

Sperren und Semaphore

Sperren: Implementierungsvarianten (1)

- Eine einfache Lösung zur Realisierung von kritischen Abschnitten ist **busy waiting** (aktives Warten)
  - Ein Prozess/Thread testet eine sog. Synchronisationsvariable
  - Test solange, bis Variable einen Wert hat, der den Zutritt erlaubt → Man braucht einen speziellen Befehl dazu
- Dieses **Polling** wird auch als **Spinlock** bezeichnet
  - Spinlocks in Betriebssystemen sind oft anzutreffen
- Manchmal ist es besser, einen Prozess/Thread „schlafen“ zu legen und erst wieder zu wecken, wenn er in den kritischen Bereich darf
  - Das ist aber oft nicht so leistungsfähig wie Spinlocks

Sperren: Implementierungsvarianten (2)

- **Vorteile** von Spinlocks:
  - Weniger Kontextwechsel notwendig
  - Gut, wenn Sperrzeit üblicherweise sehr kurz ist, kürzer als Kontextwechsel
- **Nachteile** von Spinlocks:
  - Warten benötigt CPU-Zeit
  - Blockierungen bei Singleprozessoren nicht auszuschließen, Beispiel:
    - Thread mit niedriger Priorität hält den Lock und wird suspendiert
    - Höher priorisierter Thread möchte den Lock und bleibt in der Warteschleife (bekommt immer vor dem anderen Thread die CPU)
    - Niedrig priorisierter Thread bekommt die CPU nicht mehr (wird aber bei guten Schedulern vermieden)

Sperren, Implementierungsvarianten (3)

- Hardware-Unterstützung zur Synchronisation:
  - Alle **Interrupts ausschalten**; Geht nur bei Monoprozessoren. Warum?

    → Ist aber meist eine schlechte Lösung!

```
...

Interrupts sperren
   (Maskieren, z.B. Windows IRQL hochsetzen)

/* Kritischer Abschnitt beginnt */
…

/* Kritischer Abschnitt endet */

Interrupts freigeben
   (Demaskieren)
```

Sperren, Implementierungsvarianten (4)

- Hardware-Unterstützung zur Synchronisation erforderlich:
  - **Atomare Instruktionsfolge** über nicht unterbrechbare Maschinenbefehle in einem einzigen **nicht** unterbrechbaren Speicherzyklus → wichtig bei Multiprozessoren!
  - Praktische Beispiele hierfür:
    - **Test-and-Set-Lock** (TSL = test and set lock) bzw. **TAL**
      - Lesen und Ersetzen einer Speicherzelle in einem Speicherzyklus
    - **Compare-and-Swap**
      - Vergleich und Austausch zweier Variablenwerte in einem Speicherzyklus
    - **Fetch-and-Add**
      - Lesen und Inkrementieren einer Speicherzelle in einem Speicherzyklus
    - **Exchange-Befehl CMPXCHG dest, src (im Intel-Befehlssatz)**
      - Inhalte von src und dest werden ausgetauscht (Register und Speicherbereiche als Quelle und Ziel möglich)

Sperren, Beispiel: TSL-Befehl

- Einfache Lock-Implementierung (Spinlock) mit TSL-Befehl
- LOCK ist eine Speichervariable, die vom TSL-Befehl in einem Speicherzyklus gesetzt und ausgelesen wird

```
...
01: MyLock_lock:
02: TSL R1, LOCK        // Lies LOCK in R1 ein und setze Wert von
                        // LOCK auf 1
04: CMP R1, #0          // Vergleiche Registerinhalt mit 0
                        // Wenn Vergleich zutrifft, dann ist Lock
                        // gesetzt
05: JNE MyLock_lock     // Ansonsten erneut versuchen
06: RET                 // Kritischer Abschnitt kann betreten werden

07: MyLock_unlock:
08: MOVE LOCK, #0       // LOCK auf 0 setzen (freigeben)
09: RET                 // Kritischer Abschnitt kann von anderem
                        // Prozess betreten werden
```

Siehe auch Tanenbaum, Andrew, S.; Bos, Herbert: *Moderne Betriebssysteme*, 4. aktualisierte Auflage, Pearson Studium, 2016.

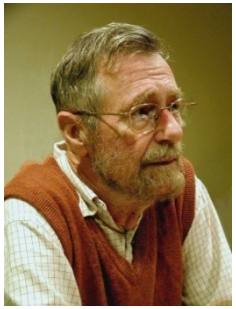
Sperren, Beispiel: Lock über XCHG-Befehl

- Pseudocode mit Intel-80386-Maschinenbefehlen

```
01: void akquireLock (var boolean lock)
02: {
03:  CODE {SYSTEM.i386}
04:  MOV EBX, lock[EBP]   ; EBX := ADR(lock)
05:  MOV AL, 1            ; AL := 1
06:  test:
07:  CMPXCHG [EBX], AL    ; Setze und lese Lockvariable atomar
08:  CMP AL, 1            ; gesperrt?
09:  JNE exit             ; ja
10:  NOP                  ; nein, erneut versuchen
11:  JMP test
12:  exit:
13: }

14: void releaseLock (var boolean lock)
15: {
16:  lock := FALSE;
17: }
```

Semaphore

- Dijkstra (1965) führte das Konzept der Semaphore zur Lösung des Mutual-Exclusion-Problems ein
- Zwei elementare Operationen
  - **P()**
    - Aufruf bei Eintritt in den kritischen Abschnitt, Operation auf Semaphor
    - Aufrufender Prozess wird in den Wartezustand versetzt, wenn sich ein anderer Prozess im kritischen Abschnitt befindet → Warteschlange
  - **V()**
    - Aufruf bei Verlassen des kritischen Abschnitts
    - Evtl. wird einer der wartenden Prozesse aktiviert und darf den kritischen Abschnitt betreten



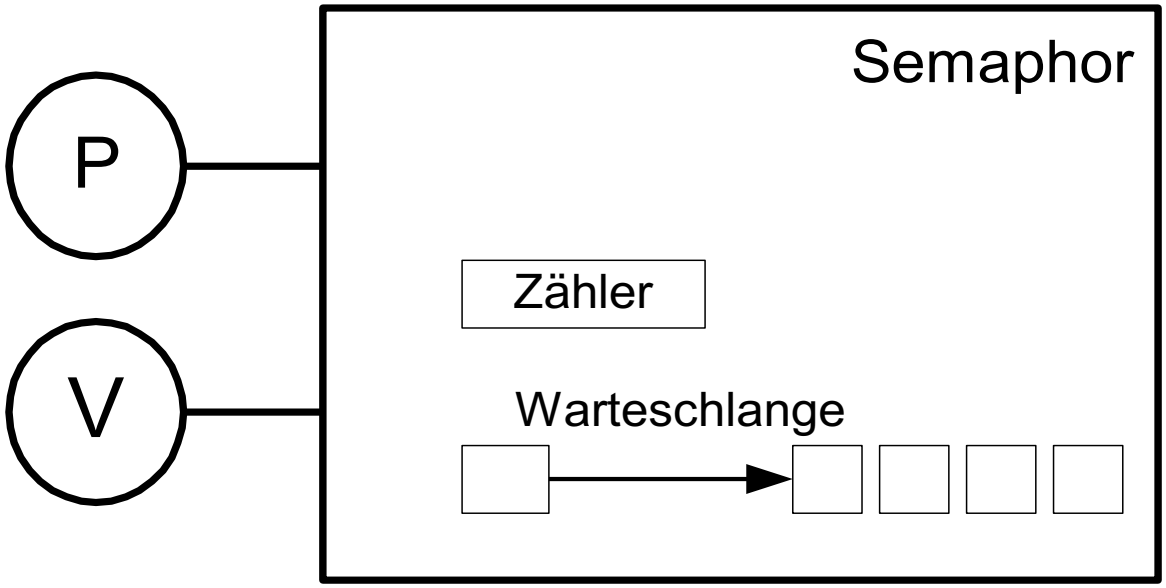
Edsger Wybe Dijkstra (11.5.1930 – 06.08.2002)

Eisenbahnsignale
Quelle: wikipedia.de

Semaphore

- Internas
  - Semaphor-Zähler
  - Warteschlange

Niederländisch:
- P kommt evtl. von probeeren = versuchen oder passeeren = passieren
- V kommt evtl. von verhogen = erhöhen oder vrijgeven = freisetzen



P: P-Operation auch Down-Operation genannt
V: V-Operation, auch Up-Operation genannt

Semaphore, Beispielnutzung

```
...
01: P();                  // kritischer Abschnitt besetzt
02: c=counter.read();     // Im kritischen Abschnitt
03: c++;                  // Im kritischen Abschnitt
04: counter.write(c);
05: V();                  // Verlassen des kritischen
                          // Abschnitts, Aufwecken eine
...                       // wartenden Prozesses
```

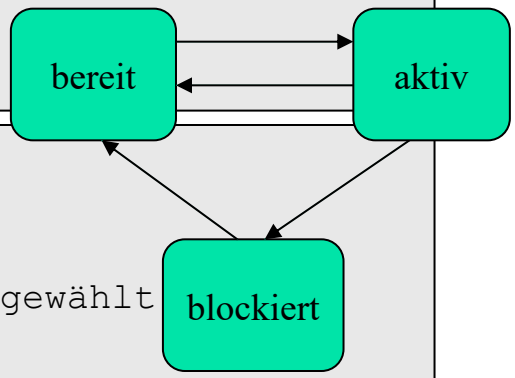
- P() und V() sind selbst wieder ununterbrechbar, also **atomare Aktionen**
- Atomare Aktionen werden **ganz** oder **gar nicht** ausgeführt

Semaphore, Algorithmus

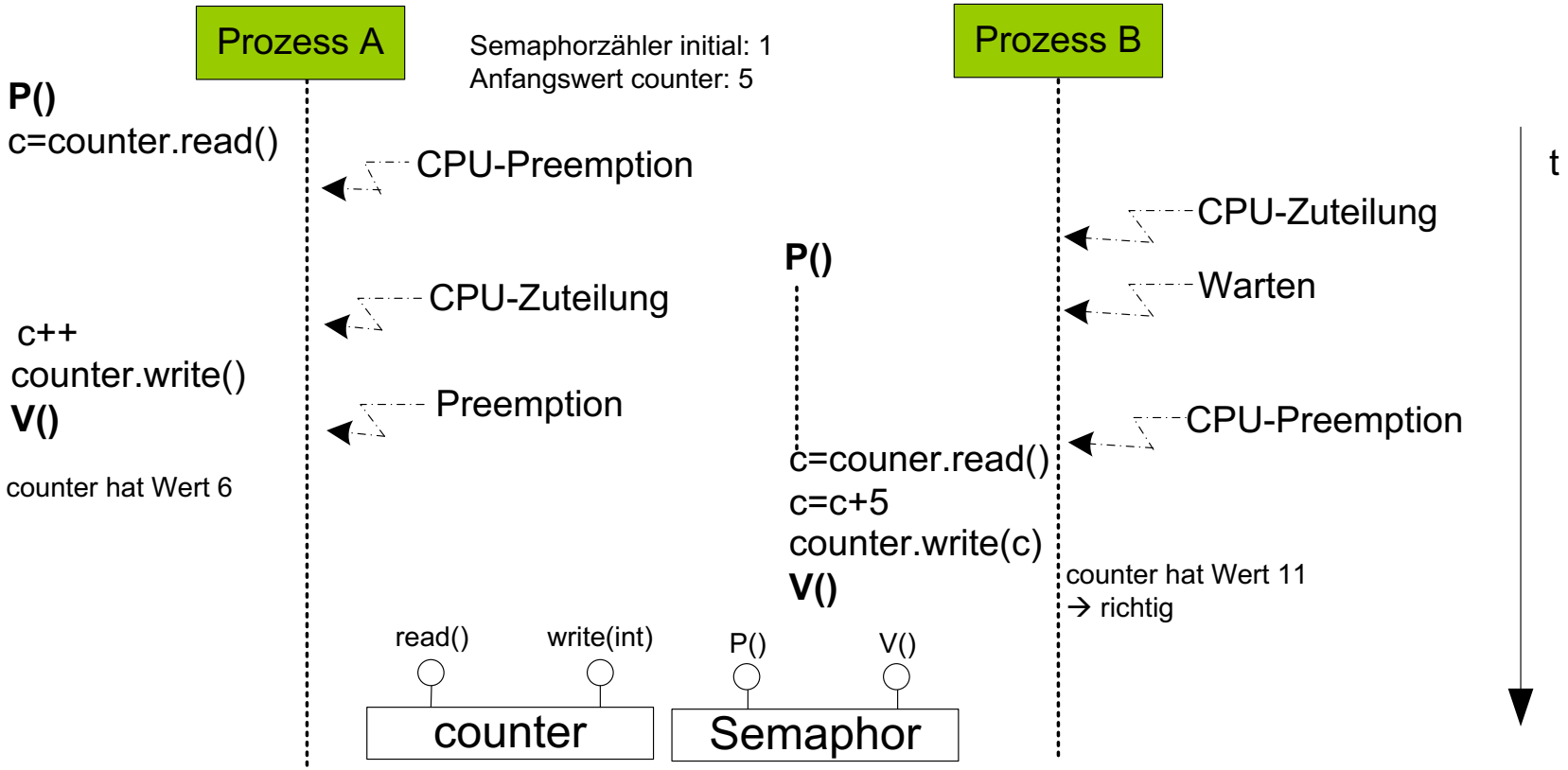
- s ist der Semaphor-Zähler, Init: s >= 1

```
01: void P() {
02:    if (s >= 1) {
03:       s = s - 1;  // Der die P-Operation ausführende Prozess
                      // setzt seinen Ablauf fort
04:    } else {
05:       // Der die P-Operation ausführende Prozess wird zunächst
          // gestoppt, in den Wartezustand versetzt und in einer
          // dem Semaphor S zugeordneten Warteliste eingetragen
06:    }
07: }
```

```
01: void V() {
02:    s = s + 1;
03:    if (Warteliste ist nicht leer) {
04:       // Aus der Warteliste wird ein Prozess ausgewählt
          // und aufgeweckt
05:    }
06: }
07: // Der die V-Operation ausführende Prozess macht weiter
```



Semaphore: Vermeidung des Lost-Update-Problems



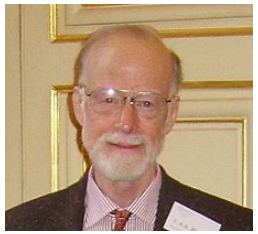
Semaphore, einfache Form: Mutex

- Wenn man auf den Zähler im Semaphor verzichten kann, kann eine einfachere Form angewendet werden

  → Diese wird als **Mutex** bezeichnet
- Ein Mutex ist leicht und effizient zu implementieren
- Ein Mutex ist eine Variable, die nur zwei Zustände haben kann:
  - locked und unlocked
- Man braucht also nur 1 Bit zur Implementierung
- Zwei Operationen:
  - mutex_lock
  - mutex_unlock

Philosophenproblem

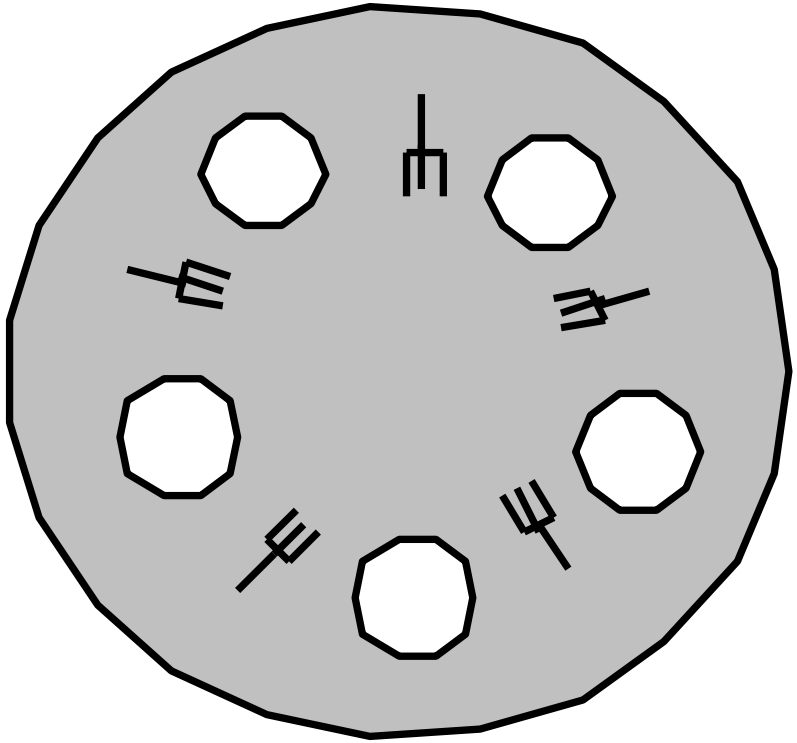
- Dijkstra und Hoare (1965), Dining Philosphers Problem:
  - Fünf Philosophen sitzen um einen Tisch herum
  - Jeder hat einen Teller mit Spaghetti vor sich
  - Zwischen den Tellern liegt je eine Gabel (5 Gabeln)
  - Zum Essen braucht ein Philosoph 2 Gabeln
  - Ein Philosoph isst und denkt abwechselnd
  - Wenn er hungrig wird, versucht er in beliebiger Reihenfolge die beiden Gabeln links und rechts von seinem Teller zu nehmen
  - Hat er sie bekommen, isst er und legt sie dann wieder auf ihren Platz zurück



Sir Charles Antony Richard Hoare (11.01.1934)
Britischer Computerwissenschaftler

Edsger Wybe Dijkstra (11.5.1930 – 06.08.2002)
Niederländischer Computer-Wissenschaftler

Prozessverwaltung: Philosophenproblem



Lösungsalgorithmus:

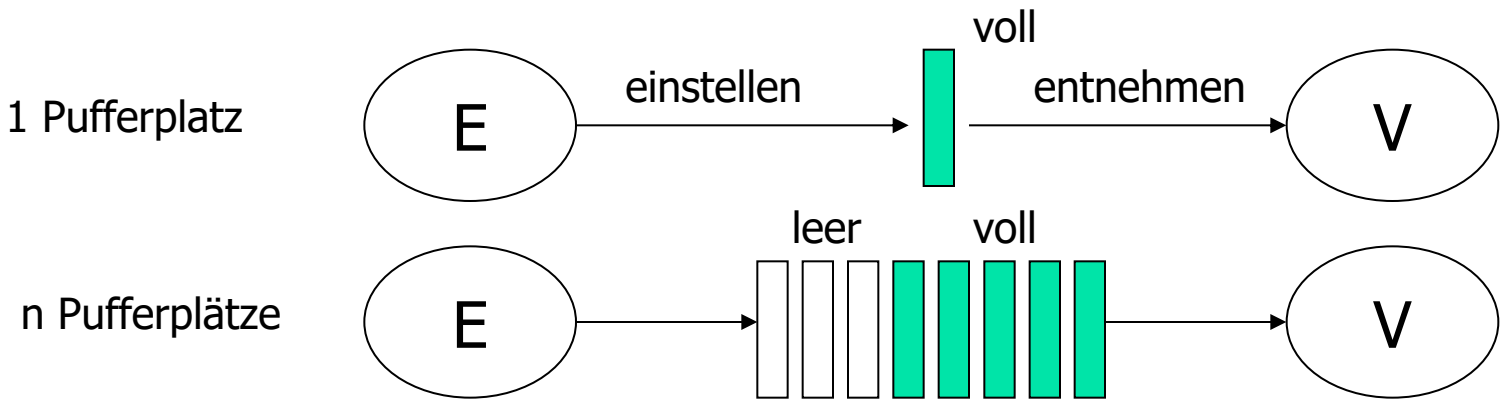
```
...
01: static int n = 5;

02: void philosopher(int i) {
03:  while (true) {
04:    think();
05:    take_fork(i);
06:    take_fork((i+1) % n);
07:    eat();
08:    put_fork(i);
09:    put_fork((i+1) % n);
10:  }
11: }
```

Warum funktioniert der angegebene Algorithmus nicht?

Finden Sie eine Lösung, bei der zwei Philosophen gleichzeitig essen können und keiner verhungern muss!

Erzeuger-Verbraucher-Problem

- Ein oder mehrere Erzeugerprozesse (producer) produzieren
- Ein oder mehrere Verbraucherprozesse (consumer) konsumieren
- Endlich große Pufferbereiche zwischen den Prozessen
  - Erzeuger füllt auf
  - Verbraucher nimmt heraus
- **Flusskontrolle** erforderlich
  - Erzeuger legt sich schlafen, wenn Puffer voll ist und wird vom Verbraucher aufgeweckt, wenn wieder Platz ist
  - Verbraucher legt sich schlafen, wenn Puffer leer ist und wird vom Erzeuger wieder aufgeweckt, wenn wieder ein Objekt im Puffer ist



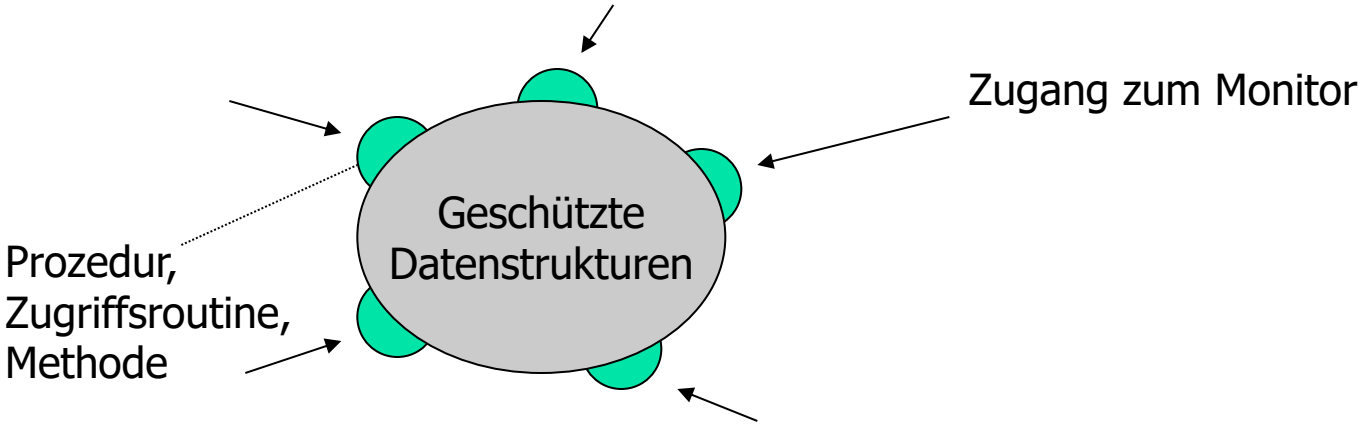
Monitore

Monitore

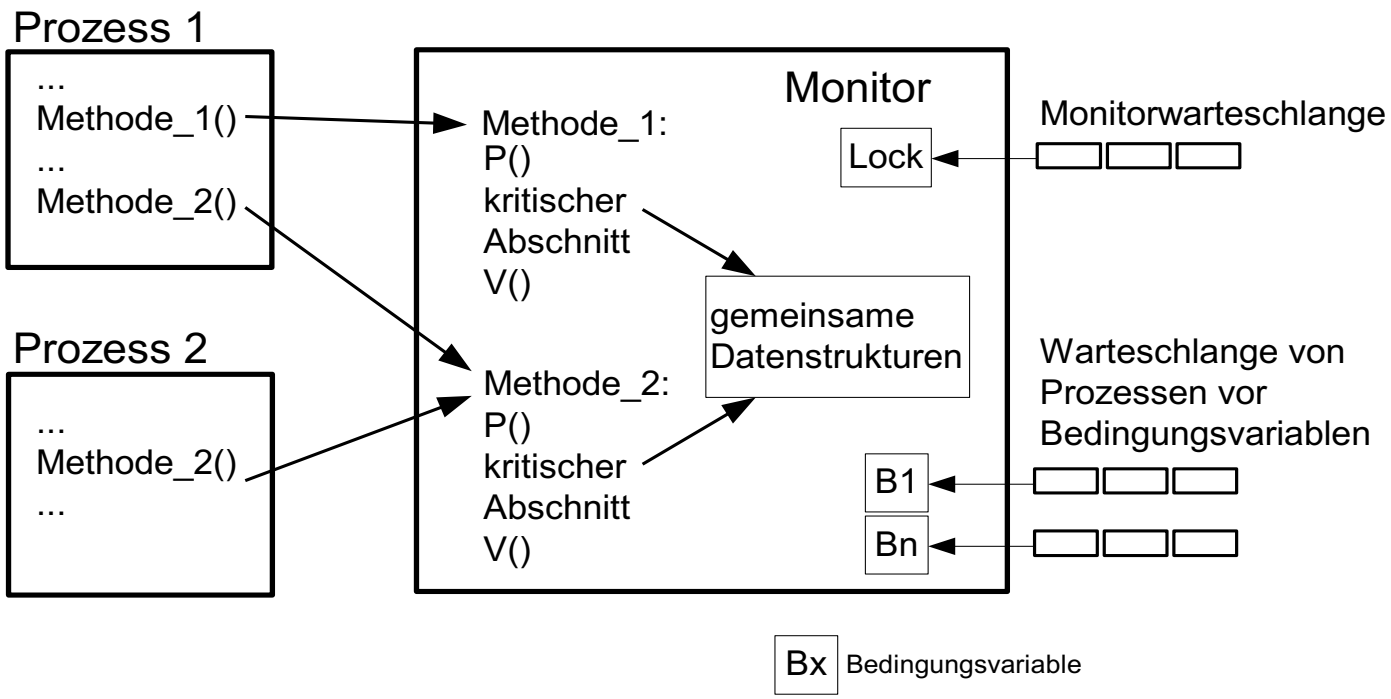
- Die Nutzung von Semaphoren ist zwar eine Erleichterung, aber nicht unproblematisch
  - Man kann als Programmierer eine Operation vergessen
  - Man kann sie versehentlich verwechseln
  - **V**(); ... Kritischer Abschnitt ...; **P**();
    - führt z.B. dazu, dass alle Prozesse im kritischen Abschnitt zugelassen sind
- **Hoare** und **Hansen** schlugen daher vor, die Erzeugung und Anordnung der Semaphor-Operationen dem Compiler zu überlassen
  - Sie entwickelten einen Abstrakten Datentypen für diese Zwecke (1973/1974), den sie mit **Monitor** bezeichneten
- In einem Monitor werden gemeinsam benutzte Daten durch eine Sperre und durch Synchronisationsvariablen (Conditions) geschützt

Monitore, Definition

- Man versteht unter einem Monitor
  - eine Menge von Prozeduren und Datenstrukturen, die als Betriebsmittel betrachtet werden
  - und mehreren Ablaufeinheiten (Prozessen/Threads) zugänglich sind,
  - aber nur von einem Prozess/Thread zu einer Zeit benutzt werden können



Monitore: Grundstruktur



Monitore, Beispielnutzung Producer-Consumer (1)

```
01: Monitor ProducerConsumer
02: {
03:   final static int N = 5; // Maximale Puffergröße
04:   static int count = 0;   // Anzahl gefüllte
                              // Pufferbereiche

05:   condition not_full;  // Signal -> Puffer nicht voll
06:   condition not_empty; // Signal -> Puffer nicht leer
      …

07:   void insert(item: integer) {
08:        if (count == N) wait(not_full);
           // Warten bis Puffer nicht mehr voll ist
09:        // Insert item
10:        count+=1;
11:        if (count == 1) signal(not_empty);
12:   }

13:   item remove() {
14:        if (count == 0) wait(not_empty);
           // Warten bis Puffer nicht mehr leer ist
15:        // Remove item
16:        count--;
17:        if (count == (N-1)) then signal(not_full);
18:        return (item);
19:   }
20: }
```

Die condition-Variablen werden in zwei Operationen genutzt:

- signal
- wait

Bei Aufruf von **wait** wird der Monitor verlassen, um auf die Erfüllung einer Bedingung zu Warten

Ein anderer Prozess kann ihn betreten → wichtig!

signal signalisiert das Erfüllen einer Bedingung

Siehe Tanenbaum, Andrew, S.; Bos, Herbert: *Moderne Betriebssysteme*, 4. aktualisierte Auflage, Pearson Studium, 2016.

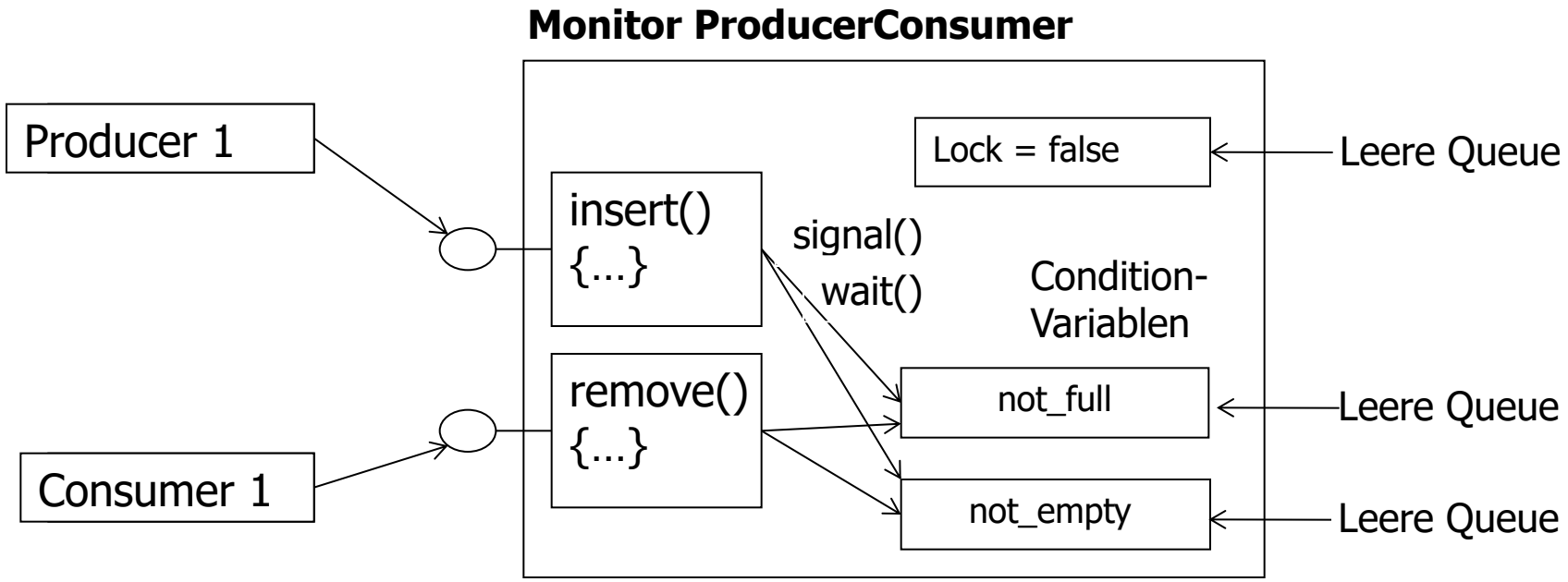
Monitore, Beispielnutzung Producer-Consumer (2)

```
01: class UseMonitor
02: {
03:    ProducerConsumer mon = new ProducerConsumer();
       ...
04:    void producer() {
05:       while (true) {
06:          // Produce item
07:          mon.insert(item);
08:       }
09:    }

10:    void consumer() {
11:       while (true) {
12:         item = mon.remove();
13:        // Consume item
14:       }
15:    }
16: }
```

Siehe Tanenbaum, Andrew, S.; Bos, Herbert: *Moderne Betriebssysteme*, 4. aktualisierte Auflage, Pearson Studium, 2016.

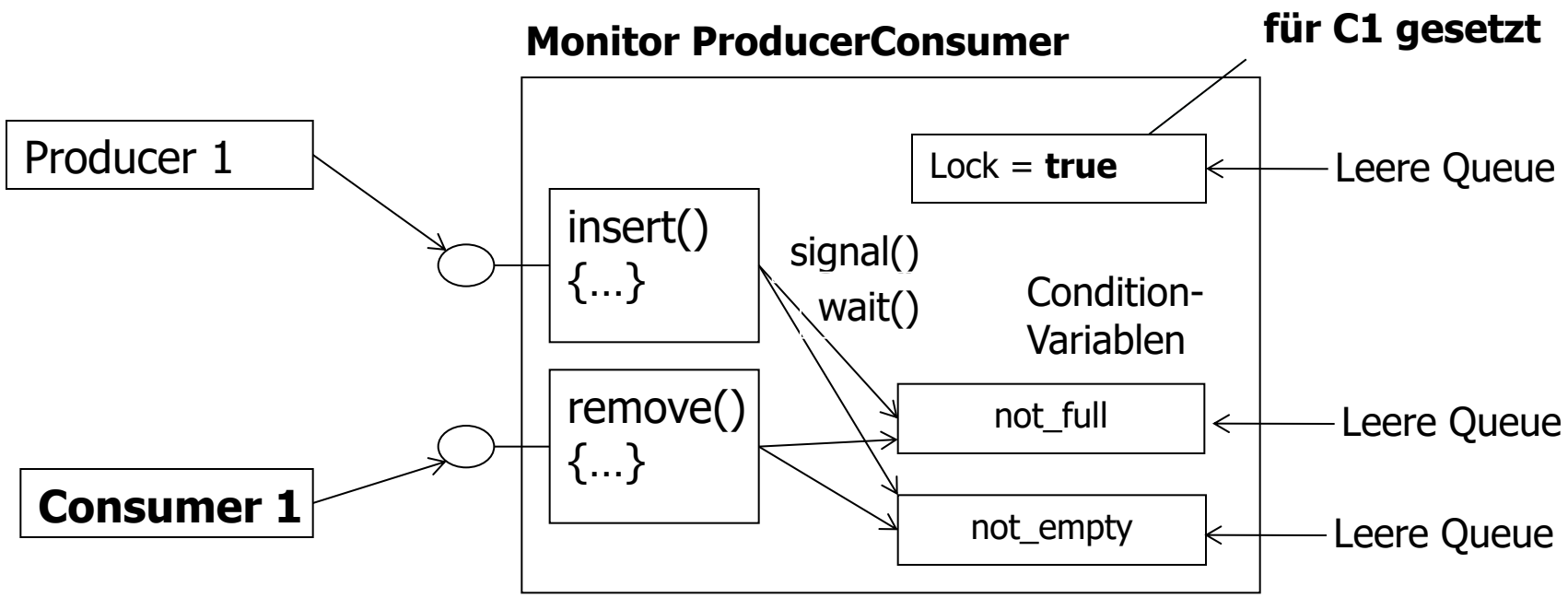
Monitore, Producer-Consumer, Szenario 1 (1)

- Initialzustand: Nichts produziert, nichts konsumiert



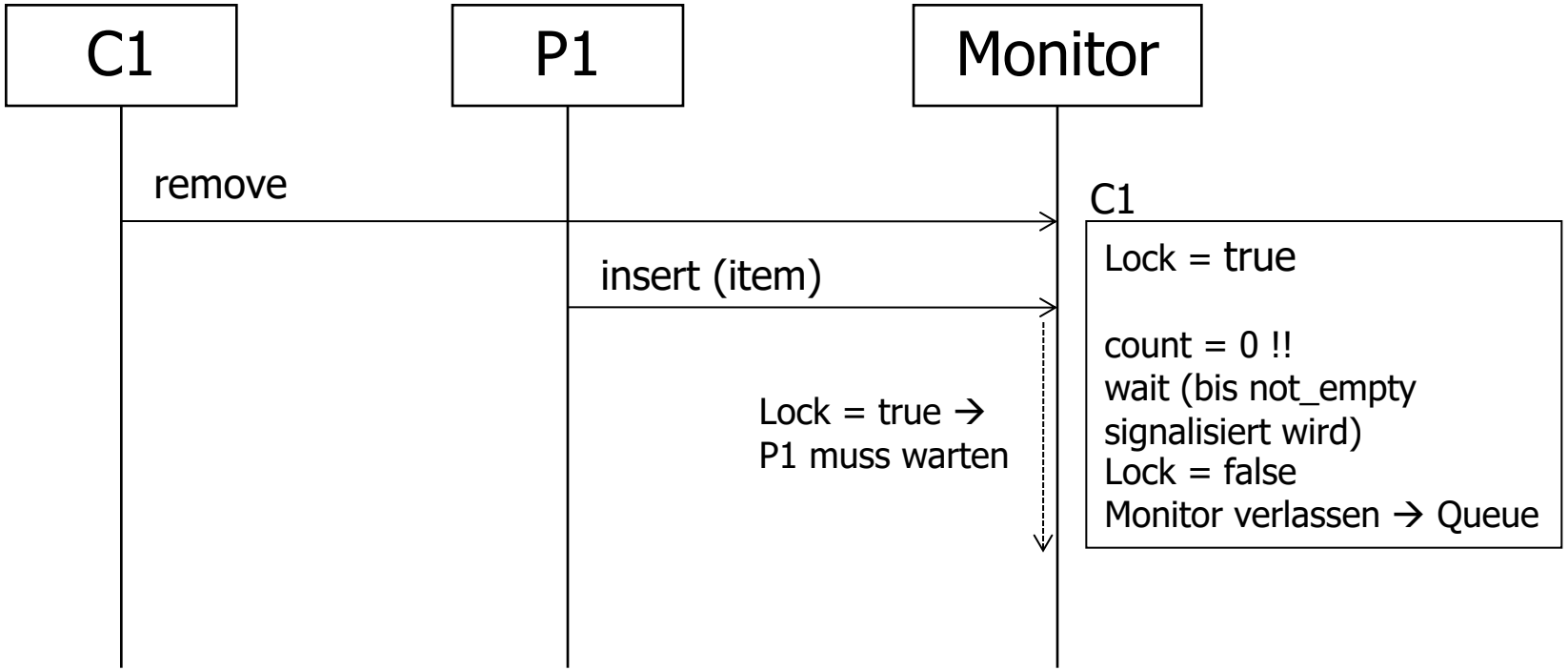
Monitore, Producer-Consumer, Szenario 1 (2)

- Consumer 1 möchte als erster konsumieren



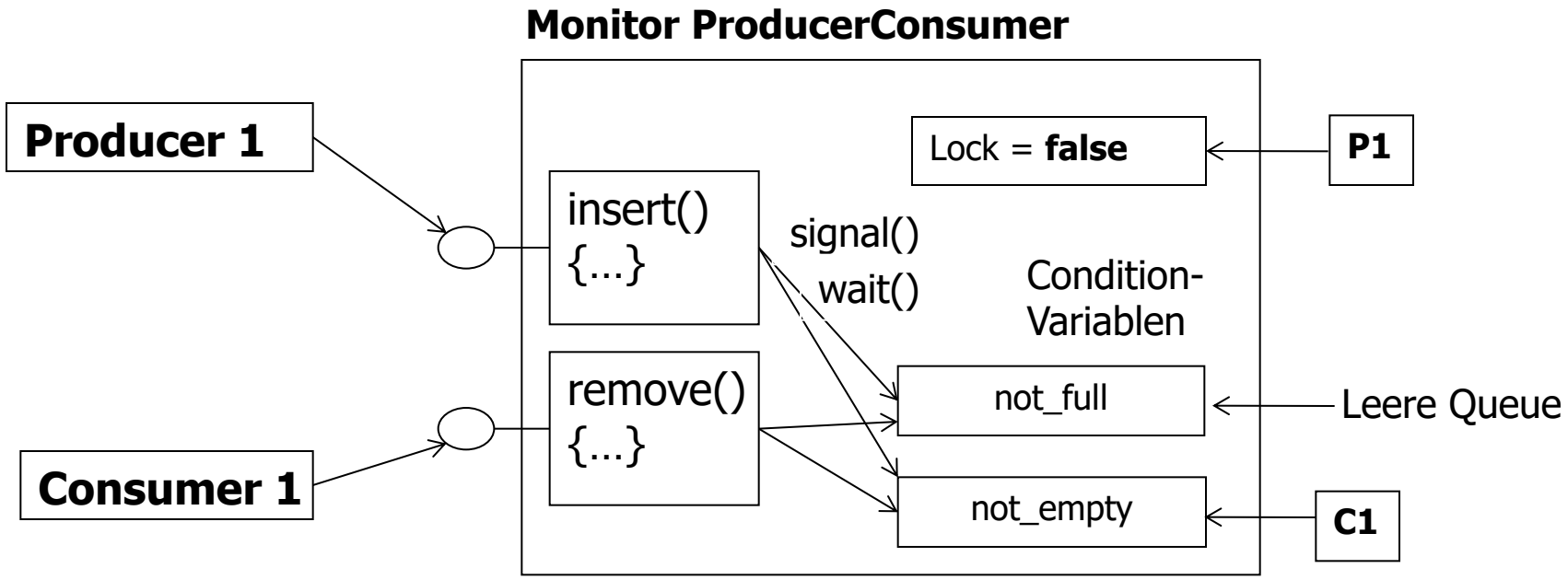
Monitore, Producer-Consumer, Szenario 1 (3)

- Consumer 1 möchte als erster konsumieren, Producer 1 kommt später und muss warten



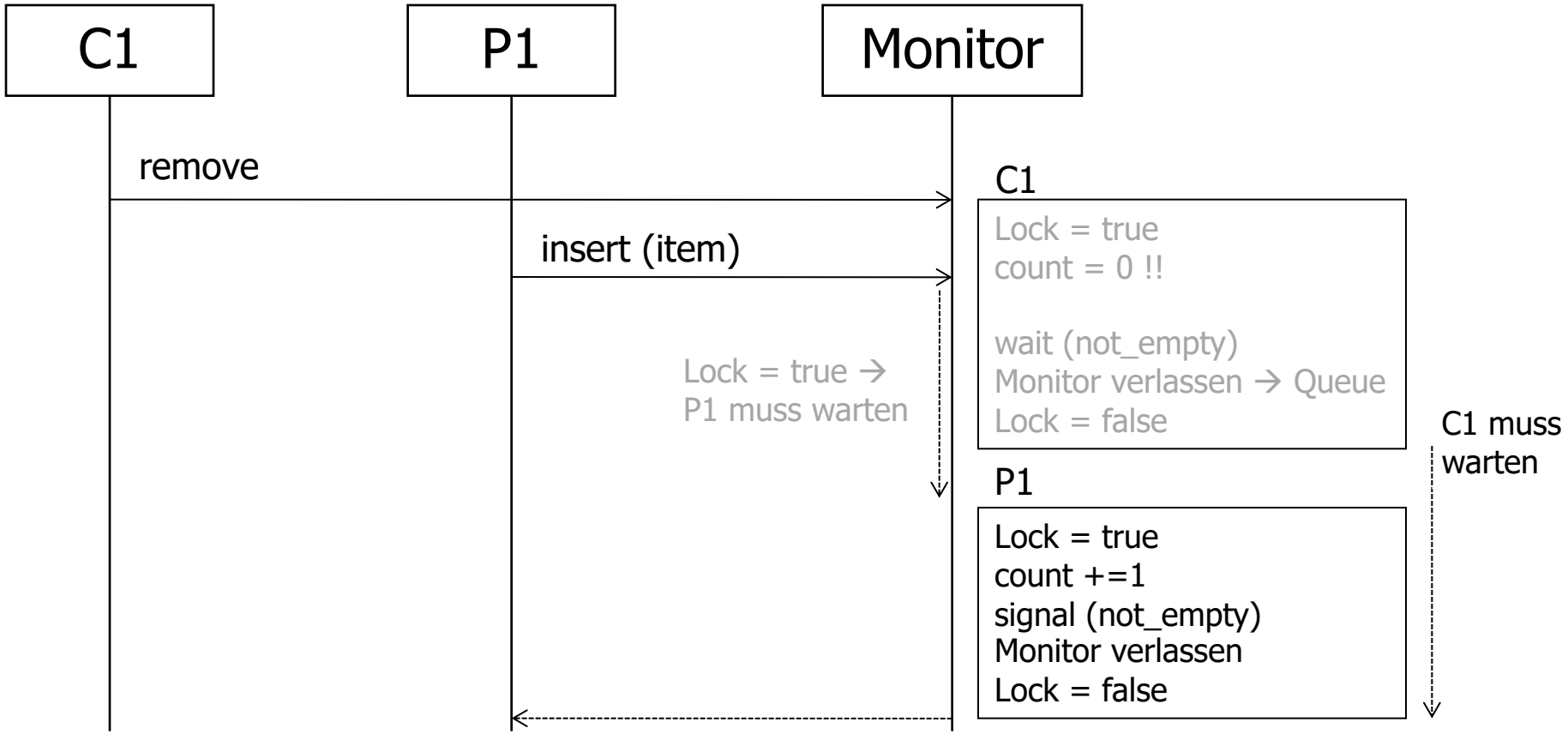
Monitore, Producer-Consumer, Szenario 1 (4)

- Consumer 1 kann nichts vorfinden (kein Item)
- Producer 1 wartet bis Consumer 1 den Monitor verlassen hat und betritt ihn dann



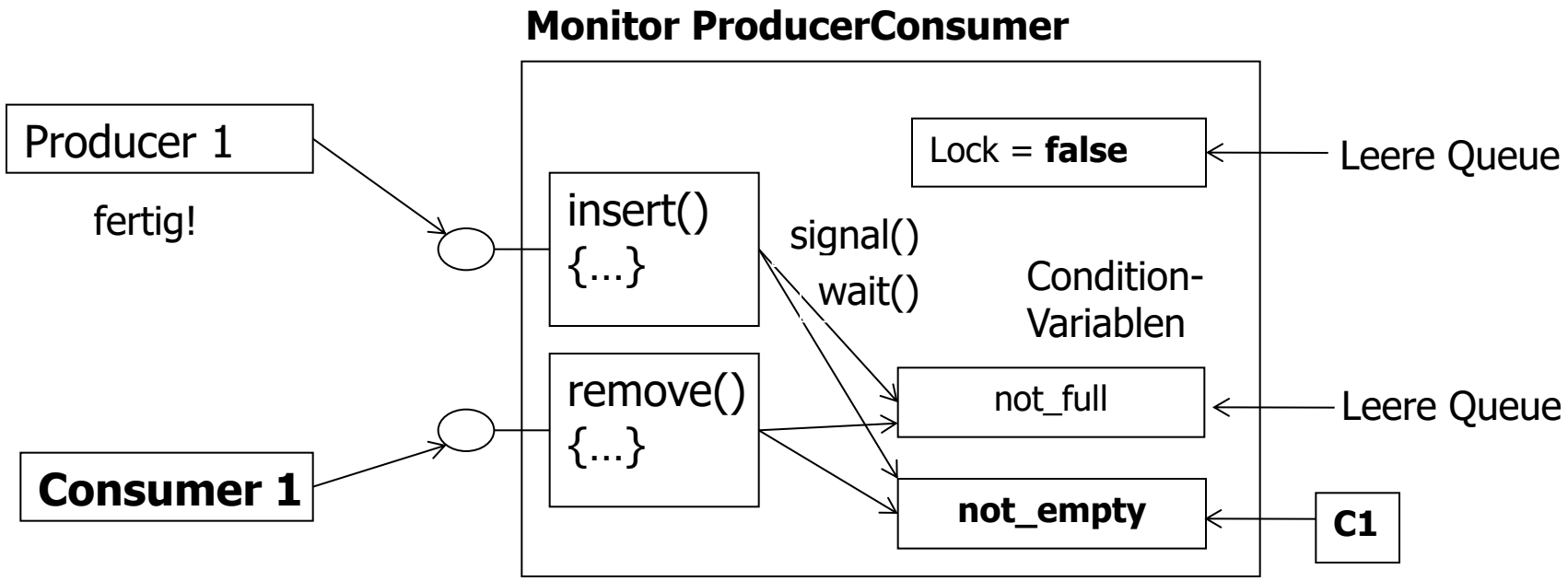
Monitore, Producer-Consumer, Szenario 1 (5)

- Producer 1 produziert
- Anm: signal() wirkt nur, wenn ein Thread darauf wartet, sonst hat es keine Wirkung und wird nicht gespeichert



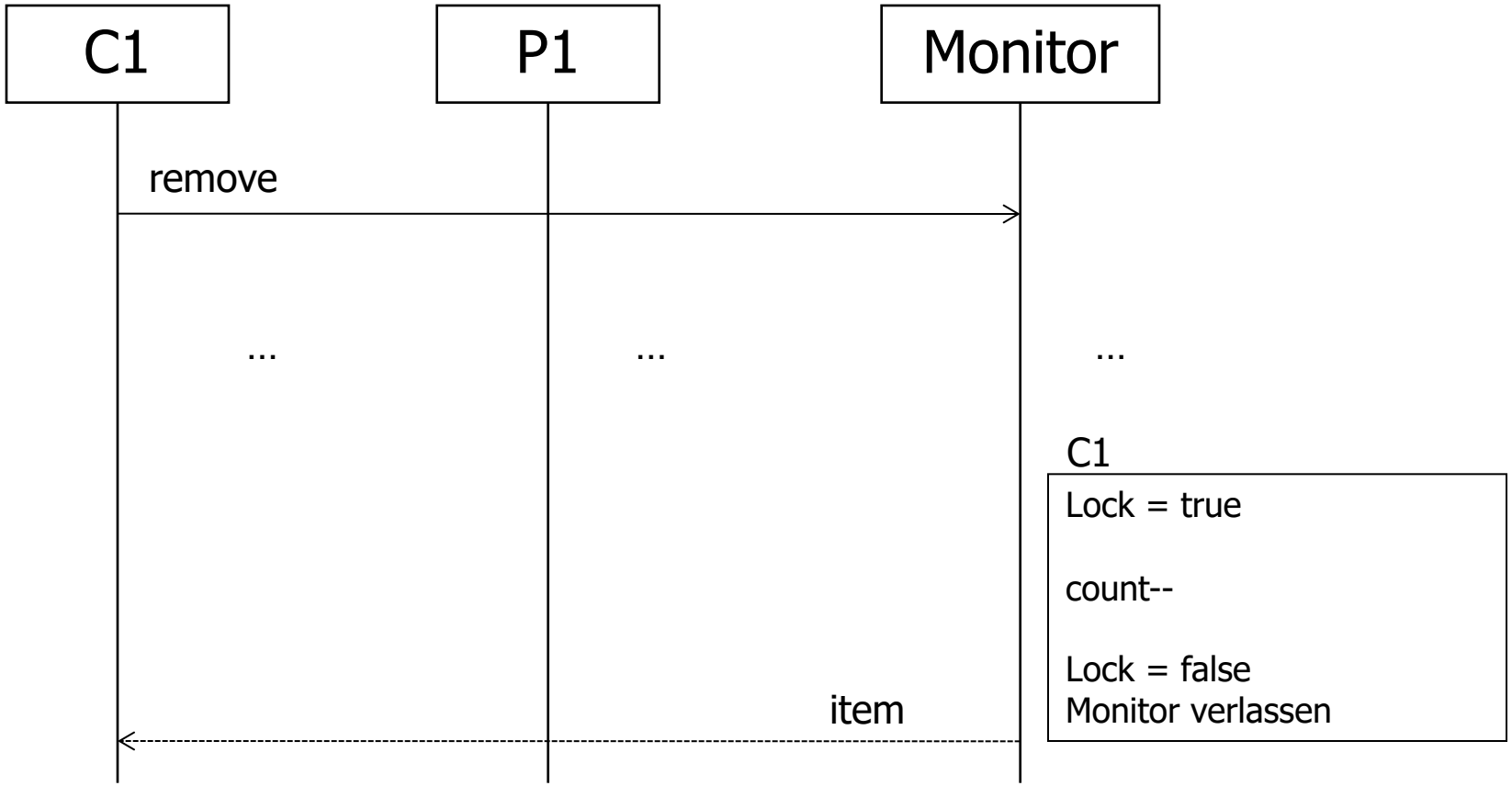
Monitore, Producer-Consumer, Szenario 1 (6)

- Consumer 1 kann nun konsumieren



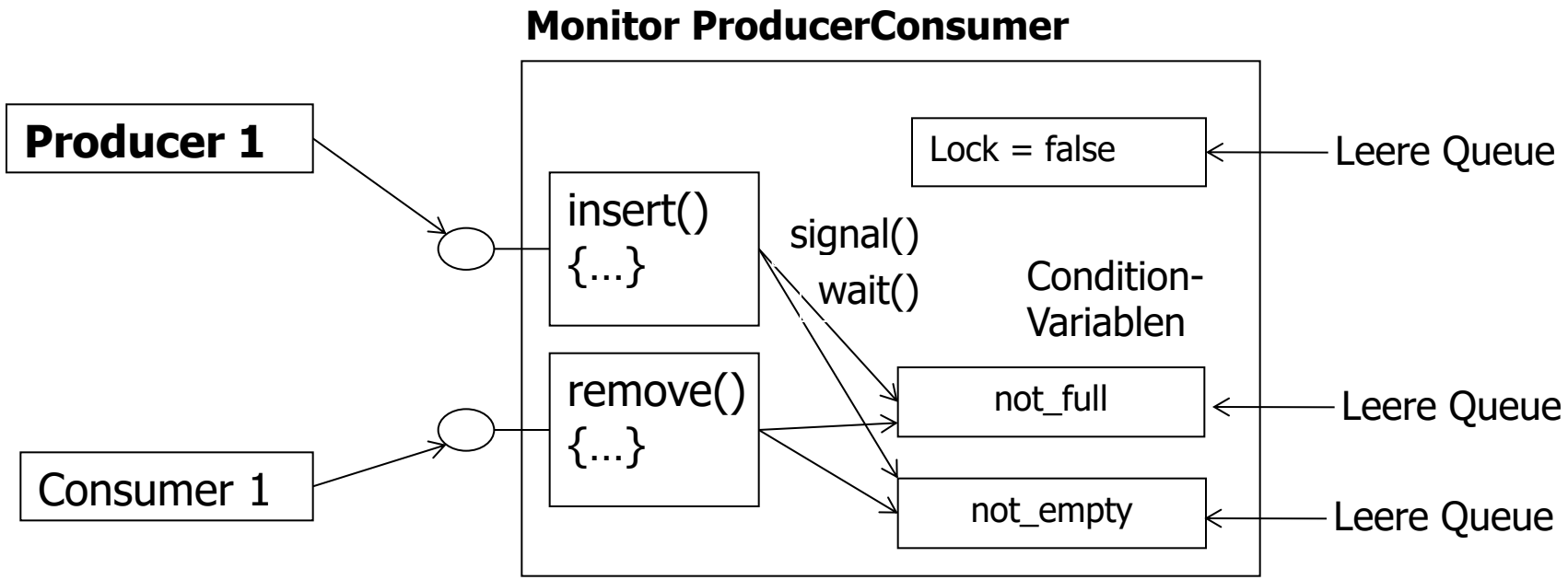
Monitore, Producer-Consumer, Szenario 1 (7)

- Consumer 1 konsumiert



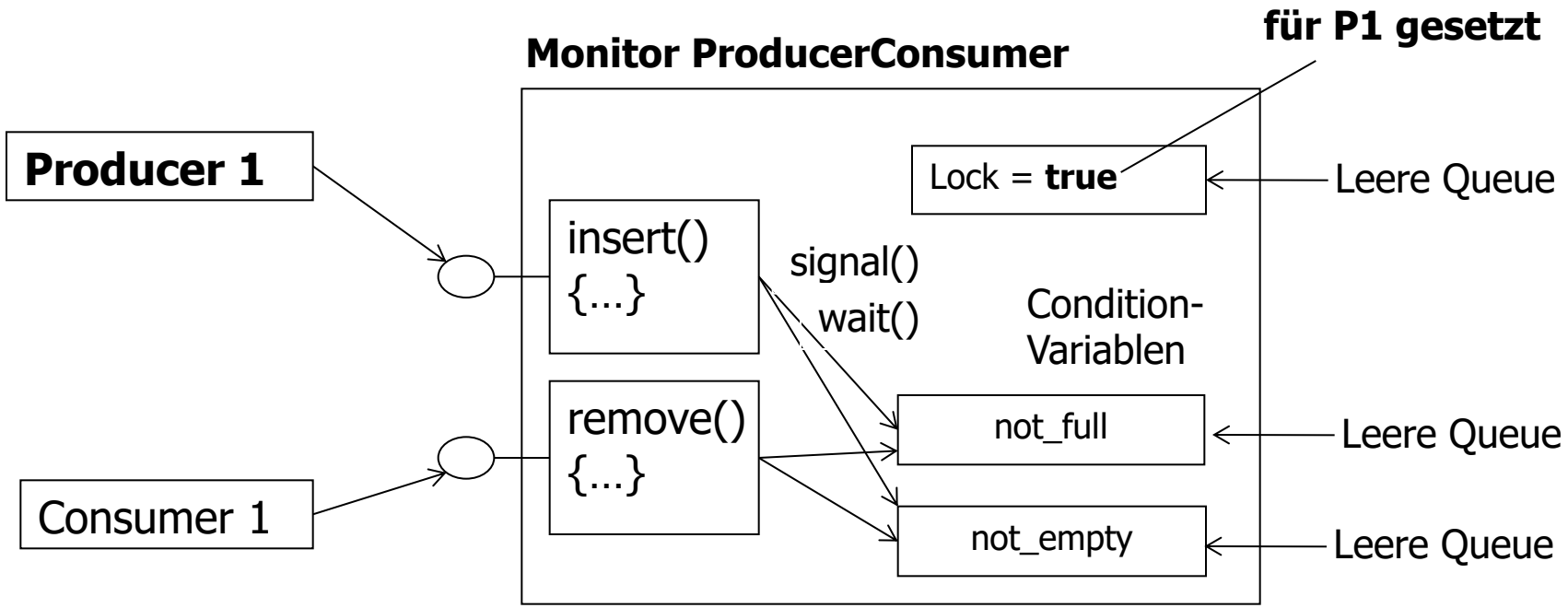
Monitore, Producer-Consumer, Szenario 2 (1)

- Anderes Szenario: Producer 1 möchte als erster produzieren



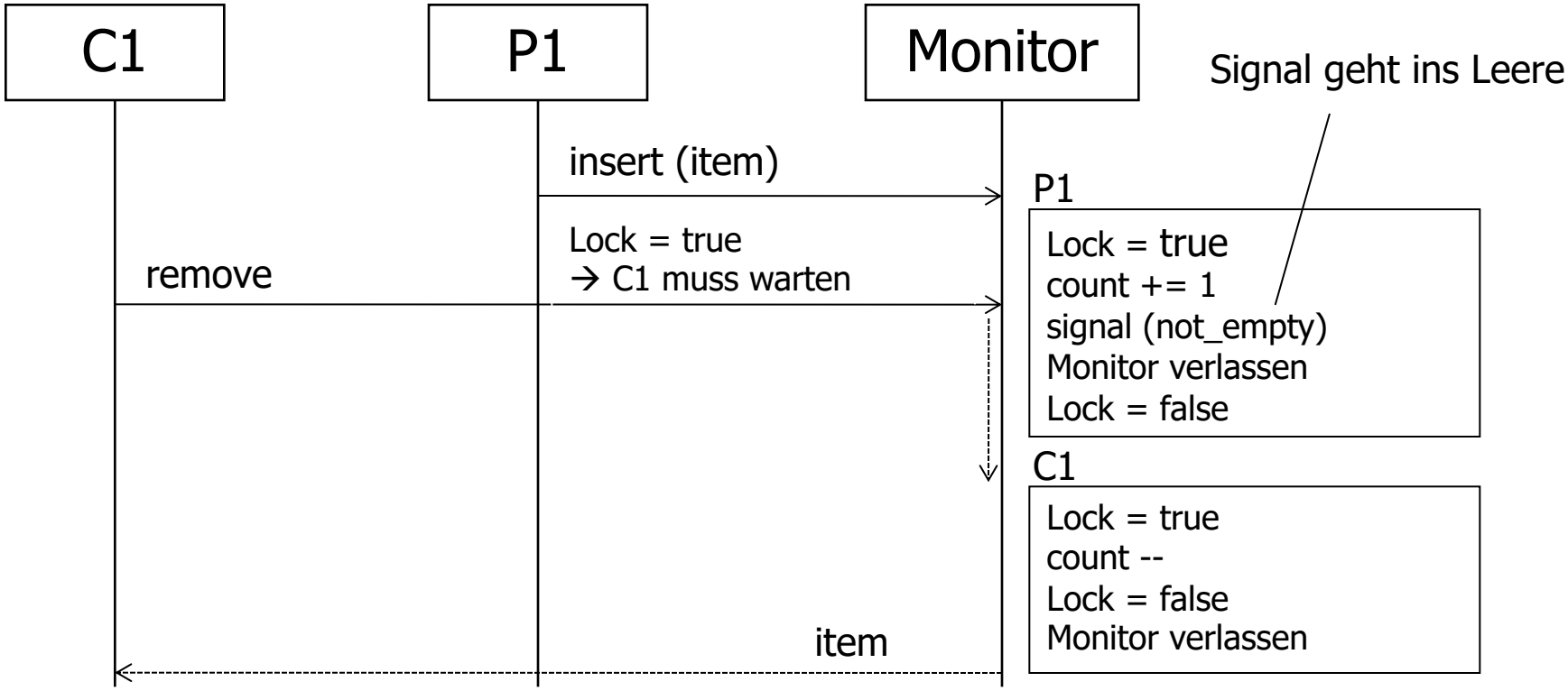
Monitore, Producer-Consumer, Szenario 2 (2)

- Anderes Szenario: Producer 1 möchte als erster produzieren



Monitore, Producer-Consumer, Szenario 2 (2)

- Consumer 1 muss warten bis Producer 1 den Monitor verlassen hat und konsumiert dann



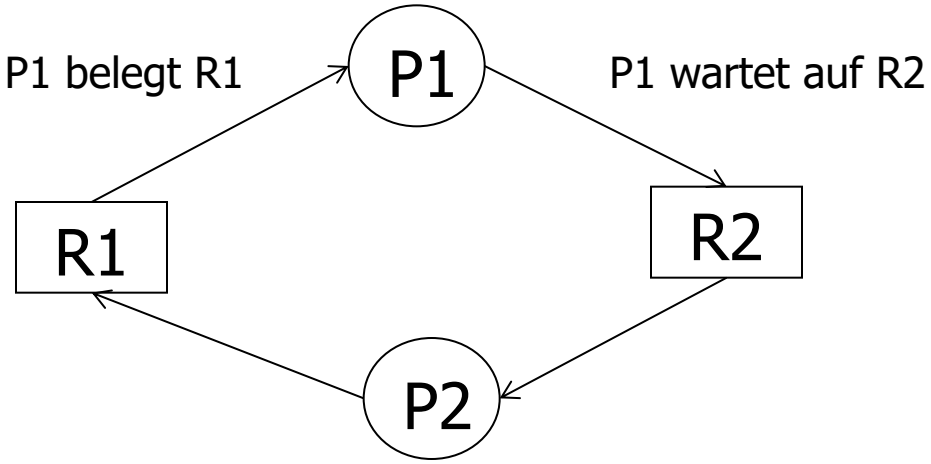
Deadlocks

Deadlocks: Definition

- Prozesse bzw. Threads können sich gegenseitig behindern oder sogar blockieren
  - Programme können nicht mehr ausgeführt werden
- Das typische Problem:
  - Prozess P1 hat das Betriebsmittel B1 reserviert und wartet auf B2, das aber von P2 reserviert ist
  - P2 wiederum wartet auf B1
  - Beide Prozesse warten ewig
- Hier haben wir es mit einer **Verklemmung** bzw. mit einem **Deadlock** zu tun

Deadlocks: Modellierung

- Klassischer Deadlock mit zwei Prozessen und zwei Betriebsmitteln (Ressourcen)
- Zur Darstellung nutzt man u. a. Betriebsmittelbelegungsgraphen



Deadlock: Bedingungen

- Es gibt vier notwendige und hinreichende Bedingungen für einen Deadlock:
  - **(1) Mutual exclusion** (notwendig)
    - Jedes beteiligte Betriebsmittel ist entweder exklusiv belegt oder frei
  - **(2) Hold-and-wait** (notwendig)
    - Prozesse belegen bereits exklusiv Betriebsmittel (mind. eines) und fordern noch weitere an: Die Anforderung wird also nicht auf einmal getätigt
  - **(3) No preemption** (notwendig)
    - Es ist kein Entzug eines Betriebsmittels möglich, Prozesse müssen sie selbst wieder zurückgeben
  - **(4) Circular waiting** (notwendig und hinreichend)
    - Zwei oder mehr Prozesse müssen in einer geschlossenen Kette auf Betriebsmittel warten, die der nächste reserviert hat

Exkurs: Notwendig und hinreichend (Aussagenlogik)

- **Notwendige** Bedingung (*conditio sine qua non*)
  - Bedingung B, die erfüllt sein muss, wenn das Ereignis E eintreten soll
  - Man schreibt E → B (aus E folgt B)
  - Mehrere notwendige Bedingungen B1, B2, B3, … möglich
  - E → B1 and B2 and B3
- **Hinreichende** Bedingung
  - Wenn Bedingung B erfüllt ist, tritt Ereignis E zwangsläufig ein.
  - Es gibt auch andere hinreichende Bedingungen
  - Man schreibt B → E (aus B folgt E)
- **Notwendige und hinreichende** Bedingung
  - Äquivalente Bedingung
  - Wenn Bedingung B erfüllt ist, tritt Ereignis E in jedem Fall ein
  - Hinreichende Bedingung ist auch notwendig
  - iff (if and only iff, genau dann, wenn)
  - Man schreibt E <-> B (Aquivalenz)

Java-Beispiel: Deadlocksituation

```
public class myDeadlock  {
    public static void main(String[] args) {
          final Object resource1 = new Object(); // Dummy-Objekte
          final Object resource2 = new Object();
                Thread t1 = new Thread( new Runnable()  {
             public void run()  {
               synchronized (resource1)  {
                   //  mach etwas
                   synchronized (resource2)  {
                        //  mach etwas
                       } } } }
               );
               Thread t2 = new Thread(  new Runnable()  {
             public void run()  {
               synchronized (resource2)  {
                   //  mach etwas
                   synchronized (resource1)  {
                        //  mach etwas
                       } } } }
            );
          t1.start(); t2.start();
    }
}
```

Zum ausprobieren!

Deadlock: Behandlung

- Es gibt vier verschiedene Strategien zur Deadlock-Behandlung:
  - **Ignorieren**
    - Vogel-Strauß-Strategie: „Kopf in den Sand“
  - **Erkennen und beheben**
    - Deadlocks sind grundsätzlich zugelassen
    - Deadlock-Erkennung über Betriebsmittelbelegungsgraphen
    - Beheben durch Entzug von Betriebsmitteln (Bedingung 3)
      - Rollback
      - Prozessabbruch
      - Transaktionsabbruch
  - **Dynamisches Verhindern**
    - Ressourcen vorsichtig zuteilen
  - **Vermeiden**
    - Eine der vier Bedingungen muss unerfüllt bleiben

Synchronisation in Java

Java: Die Synchronisationsprimitive synchronized

- Der Modifier *synchronized* dient der Festlegung kritischer Abschnitte:
  - einzelne Codeblöcke
  - ganze Methoden von Objekten
- Wichtiger Begriff:
  - **Thread-safe** (Syn.: eintrittsinvariant, reentrant, wiedereintrittsfähig) heißt eine Klasse oder Methode, wenn sie bedenkenlos in einer nebenläufigen Thread-Umgebung genutzt werden kann
  - Beispiel: Klasse *java.util.concurrent.ConcurrentHashMap*
    https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html
  - Problematisch: Klassenvariablen, globale Variablen

Java-Monitore: Synchronisationsprimitive synchronized

- Zugriffsserialisierte Methode

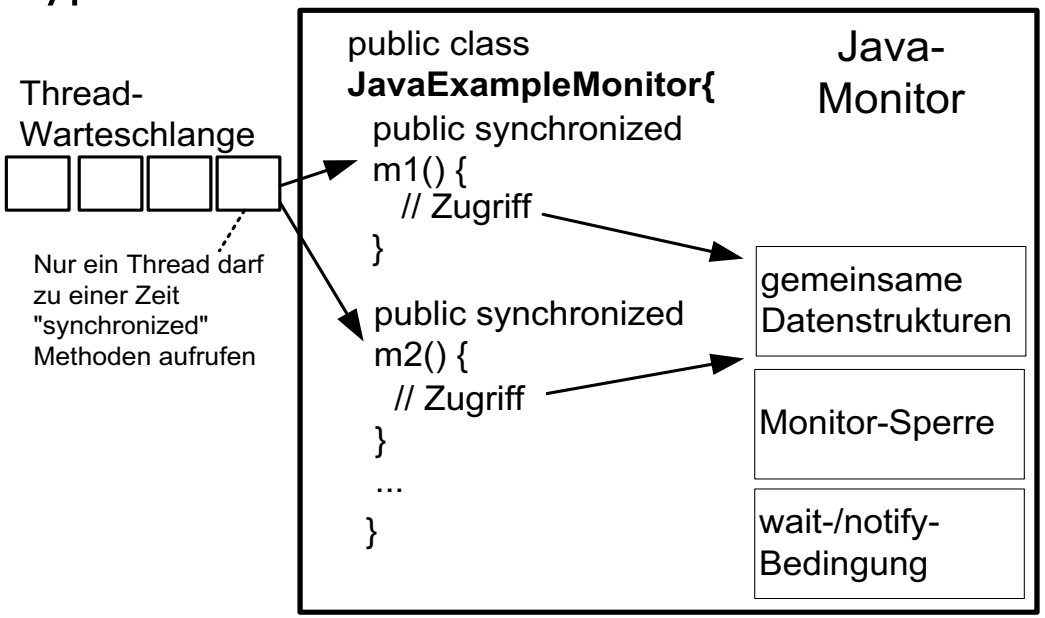
```
public synchronized void method1()
{
    // geschützter Codebereich
}
```

- Zugriffsserialisierter Anweisungsblock

```
...
XyObject object1 = new XyObject(…);

synchronized (object1)
{
    // geschützter Codebereich
}
```

Monitore: Java-Monitor

- Java-Monitor implementiert **signal-and-continue**
  - Signalisierender Thread wird nicht sofort blockiert
  - Auch als Mesa-Typ bezeichnet



- Anm: Alternativer Monitor-Ansatz:
  - **signal-and-wait**, signalisierender Thread wird sofort blockiert → Hoare-Typ

Java-Monitore: Synchronisationsprimitive synchronized

- Anwendung auf ganze Methoden
  - Das betroffene Objekt wird vor Zugriffen anderer Instanzen gesperrt
  - Sperre wird gehalten, bis die Methode abgearbeitet ist
  - Wird darüber hinaus der Modifier *static* bei der Methodendefinition benutzt, so wird die ganze Klasse gesperrt, bis die Methode abgearbeitet ist

Java-Monitore: „Implizite Monitore“

- *Synchronized* wird in der JVM über eine Monitor-Variante mit **einer** Sperre und **einer** Condition-Variable implementiert.
- Für jedes Objekt, das mind. eine *synchronized*-Methode hat, wird von der JVM ein eigener Monitor ergänzt
- Der Monitor realisiert das Sperren und Warten, wenn ein Thread auf *synchronized*-Methode zugreift
- Sperre wird aufgehoben, wenn Thread die Methode verlässt

Threads in Java (Wdl.)

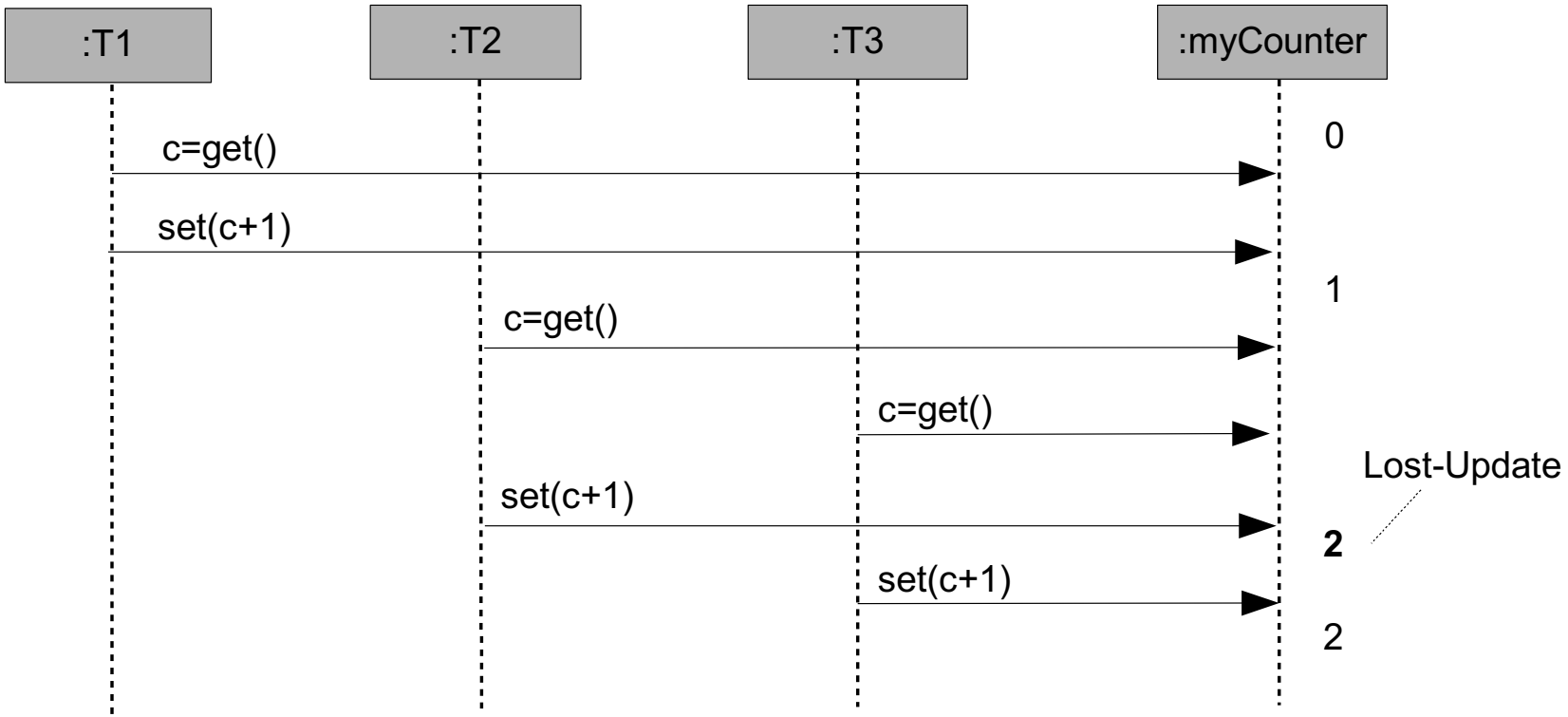
- Nebenläufigkeit wird durch die Klasse *Thread* aus dem Package java.lang unterstützt
- Realisiertes Basiskonzept in der JVM: Monitore
- Eigene Klasse definieren, die von Thread abgeleitet ist und die Methode run() überschreiben
- Alternative: In einer Klasse das Interface *Runnable* implementieren und die Methode run() schreiben
  - Hat Vorteile wegen fehlender Mehrfachvererbung in Java
  - Wird daher meistens verwendet

Übungsbeispiel: Zähler durch mehrere Threads hoch zählen (1)

- Code für Counter

```
01: package Threads;
02: import java.io.*;
03:
04: class CounterObject {
05:  private int count = 0;
06:
07:  CounterObject() {
08:  // Nichts zu konstruieren
09:  }
10:
11:  void set(int newCount) {
12:       count = newCount;
13:  }
14:
15:  int get() {
16:       return count;
17:  }
18:}
```

Übungsbeispiel:
Zähler durch mehrere Threads hoch zählen (2)



Übungsbeispiel: Zähler durch mehrere Threads hoch zählen (3)

- Code für Thread-Klasse (nicht synchronisiert)

```
01: class CountThread1 extends Thread {
02:  private CounterObject myCounter;
03:  private int myMaxCount;
04:  CountThread1(CounterObject c, int maxCount)
05:  {
06:        myCounter = c;
07:        myMaxCount = maxCount;
08:  }
09:  public void run() {
10:    System.out.println("Thread "+getName()+" gestartet");
11:    for (int i=0;i<myMaxCount;i++){
12:
13:        // Kritischer Bereich
14:        int c = myCounter.get();
15:        c++;
16:        myCounter.set(c);
17:        // Ende des kritischen Bereichs
18:    }
19:  }
20:}
```

Übungsbeispiel: Zähler durch mehrere Threads hoch zählen (4)

- Code für Thread-Klasse (synchronisiert)

```
01: class CountThread1 extends Thread {
02:  private CounterObject myCounter;
03:  private int myMaxCount;
04:  CountThread1(CounterObject c, int maxCount)
05:  {
06:        myCounter = c;
07:        myMaxCount = maxCount;
08:  }
09:  public void run() {
10:    System.out.println("Thread "+getName()+" gestartet");
11:    for (int i=0;i<myMaxCount;i++){
12:
13:        synchronized (myCounter) {
14:            int c = myCounter.get();
15:            c++;
16:            myCounter.set(c);
17:        }
18:    }
19:  }
20:}
```

Einschub: Philosophenproblem Lösung mit Java-Monitoren

- Zieschke et al. beschreiben eine Lösung mit Nutzung von **synchronized** in Java
- Stäbchen-Klasse mit **nummerierten Stäbchen**
- Philosoph wird als Thread modelliert
  - Zwei verschachtelte synchronized-Blöcke für das Stäbchen-Aufnehmen
  - Philosoph nimmt immer zunächst das Stäbchen mit der kleineren Nummer
  - Situation, dass alle Philosophen ein Stäbchen haben, kann nicht auftreten
  - Dadurch ist keine Verklemmung nicht möglich (später)
  - Siehe Beispielcode -→ Quelle unten

Quelle: Ziesche, P.; Arinir, D.: Java: Nebenläufige und verteilte Programmierung. Konzepte, UML2-Modellierung, Realisierung in Java, 2. Auflage. W3L-Verlag, Herdecke, Witten. 2010

Methoden zur expliziten Synchronisierung (1)

- *wait()*
  - Thread geht in Wartezustand bis ein notify()- oder notifyAll()-Aufruf eines anderen Threads abgesetzt wird, der zum gleichen kritischen Abschnitt passt
- *wait(long timeout)*
  - Thread geht max „timeout“ ms in einen Wartezustand (sonst wie wait())
- *notify()*
  - Weckt (mind.) einen wartenden Thread auf
- *notifyAll()*
  - Weckt alle wartenden Thread auf

Methoden zur expliziten Synchronisierung (2)

- wait() und notify() dürfen **nur innerhalb eines mit synchronized geschützten Abschnitts** aufgerufen werden
- wait() blockiert den Thread selbst, **die Sperre für den kritischen Abschnitt wird freigegeben**
- wait()/notify() ist **laufzeitkritisch**, Signalisierung darf nicht zu früh kommen, sonst geht sie ins Leere
- notify() **garantiert nicht**, dass genau ein Thread aufgeweckt wird → Bedingung erneut prüfen
  - wait() in while-Schleife aufrufen, Bedingung erneut prüfen
  - Evtl. alle wecken, damit richtiger dabei ist (notifyAll())
- **Kein Warteschlangenmechanismus** implementiert, also nicht absolut fair

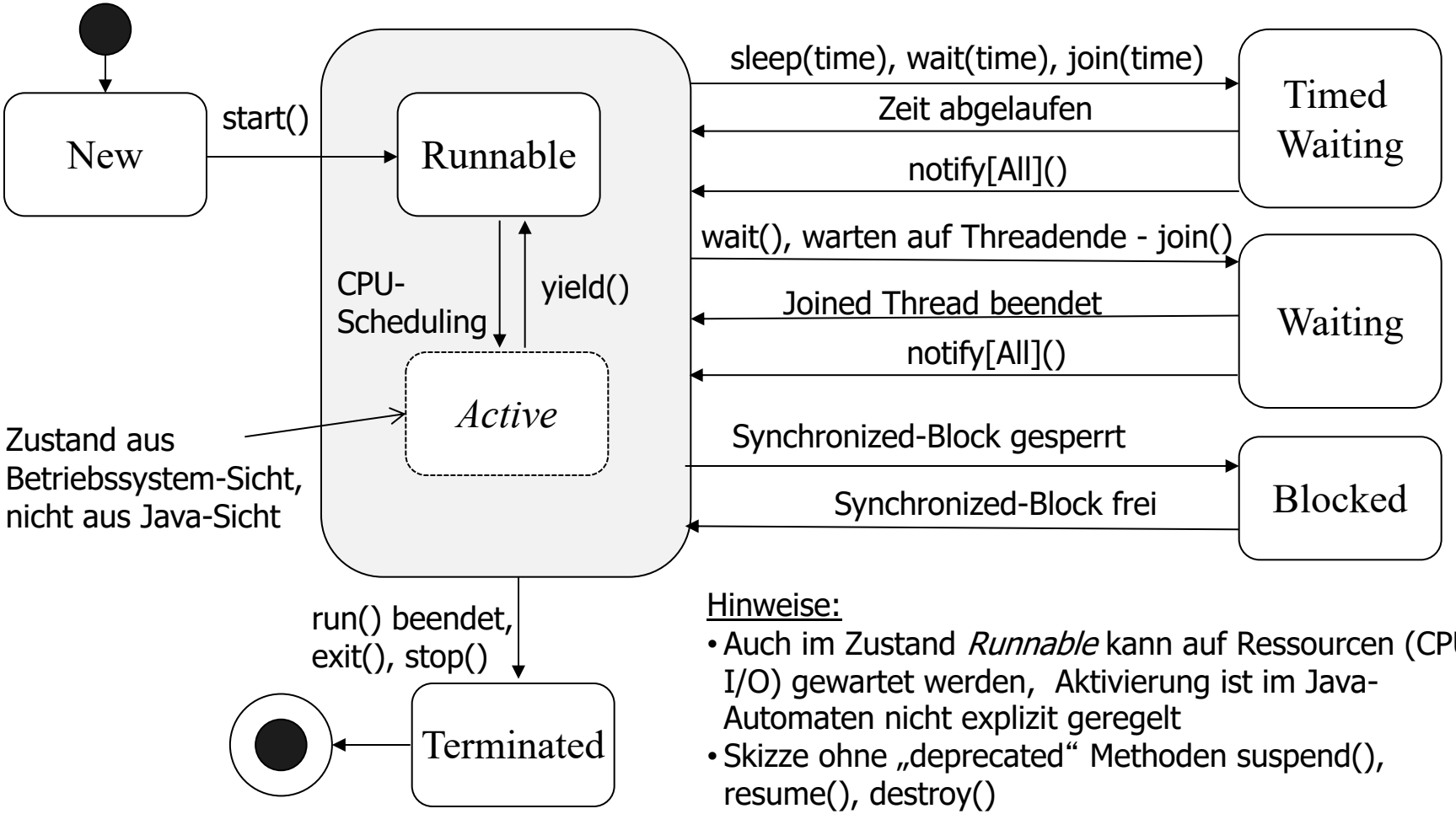
Java-Thread-Zustände gemäß API (1) (siehe Enum Thread.State)

- New
  - Noch nicht gestarteter Thread
- Runnable
  - Thread läuft in JVM, wartet aber evtl. auf Ressourcen des Betriebssystems (Prozessorzuteilung); Thread in diesem Zustand muss also nicht unbedingt eine CPU nutzen (Zuteilung erfolgt über Betriebssystem)
- Blocked
  - Thread ist blockiert und wartet auf einen Monitor-Lock, um in einen Synchronized-Block zu gelangen oder kann nach einem wait()-Aufruf wieder in einen kritischen Abschnitt eintreten

Java-Thread-Zustände gemäß API (2) (siehe Enum Thread.State)

- Waiting
  - Thread wartet auf einen anderen Thread, um eine Aktion ausführen zu können. Thread hat wait() oder join() ohne Zeitangabe aufgerufen
- Timed Waiting
  - Thread wartet auf das Auslaufen einer vorgegebenen Zeitspanne. Thread hat wait() oder join() mit Zeitangabe oder sleep() aufgerufen
- Terminated
  - Thread hat seine Arbeit abgeschlossen

Verfeinerter Zustandsautomat für einen Java-Thread



Hinweise:

- Auch im Zustand *Runnable* kann auf Ressourcen (CPU, I/O) gewartet werden, Aktivierung ist im Java-Automaten nicht explizit geregelt
- Skizze ohne „deprecated“ Methoden suspend(), resume(), destroy()
- Thread.yield()-Aufruf bewirkt, dass CPU abgegeben wird

Übungsbeispiel zur expliziten Synchronisation
Eigene Semaphor-Implementierung

```
01: class MySemaphor {
02:   private int max;     // Anzahl der maximal möglichen Threads im
03:                        // kritischen Abschnitt
04:   private int free;    // Anzahl der verfügbaren Plätze im kritischen
05:                        // Abschnitt (so viele Threads dürfen noch in
06:                        // den kritischen Abschnitt rein)
07:   private int waiting; // Anzahl der in der P-Operation mit wait
08:                        // wartenden Threads (so viele Threads möchten
09:                        // aktuell in den kritischen Abschnitt rein)
10:   public MySemaphor() {
11:         this(0);
12:   }
13:   public MySemaphor(int i) {
14:         if (i >=0) {
15:           max = i;
16:         } else {
17:           max = 0;
18:         }
19:         free = max;
20:         waiting = 0;
21:   }
...
```

Übungsbeispiel zur expliziten Synchronisation Semaphor-Implementierung

```
01: /**
02:  * P-Operation
03:  */
04: public synchronized void P()
05: {
06:   while (free <= 0)
07:   {
08:        waiting++;  // Anzahl der wartenden Threads erhöhen
09:        try {
10:              this.wait();
11:        } catch (InterruptedException e) {}
12:
13:        waiting--;   // Anzahl der wartenden Threads vermindern
14:   }
15:   free--; // Jetzt erst Semaphorzähler vermindern
16: }
```

Übungsbeispiel zur expliziten Synchronisation Semaphor-Implementierung

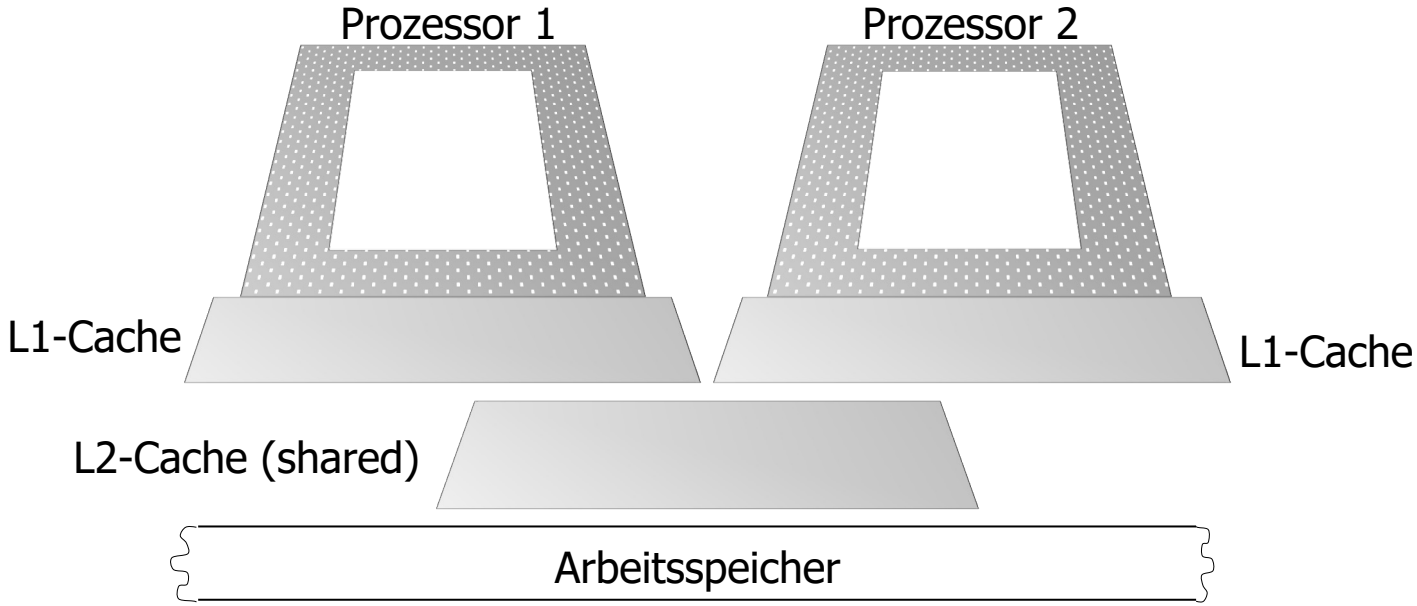
```
01: /**
02:   * V-Operation
03:   */
04: public synchronized void V()
05: {
06:   free++;  // Semaphorzähler erhöhen
07:   if (waiting > 0)
08:   {
09:        // Nur wenn ein anderer Thread wartet,
10:        // diesen mit notify benachrichtigen
11:        this.notify();
12:   }
13: }
```

Java-Synchronisation: Abschließende Anmerkungen

- Die Synchronisation von Threads mit „synchronized“ kann problematisch für die Performance sein
  - Monitor-Verwaltung kostet etwas
  - Man muss aufpassen, da zu große kritische Abschnitte zu Leistungseinbußen führen
- Programmierung von Threads ist gefährlich, da man Synchronisationsprobleme leicht übersehen kann
  - Ist ein Codestück, das zu serialisieren ist, nicht mit „*synchronized*“ serialisiert, dann wird es irgendwann mal ein Problem geben, auch wenn man es nicht gleich bemerkt!

Exkurs: Java-Speichermodell

- Der Prozessor optimiert die Cache-Nutzung, was zu Inkonsistenzen führen kann
- Nutzung von **volatile**: push beim Schreibzugriff und refresh vor jedem Lesezugriff
- Auch synchronized macht push und refresh bei jedem Zugriff



Quelle: Ziesche, P.; Arinir, D.: Java: Nebenläufige und verteilte Programmierung. Konzepte, UML2-Modellierung, Realisierung in Java, 2. Auflage. W3L-Verlag, Herdecke, Witten. 2010