

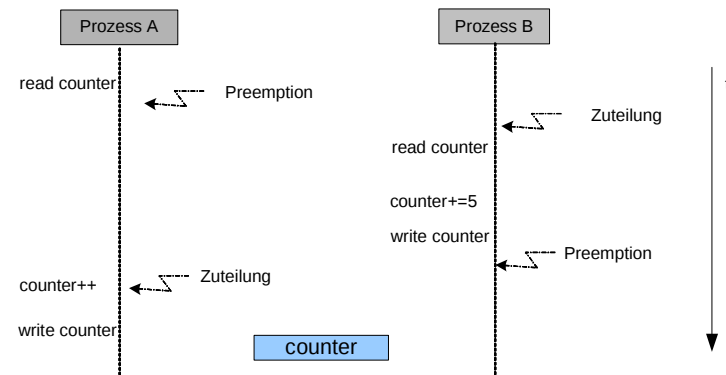
MAS: Betriebssysteme

Koordination und Synchronisation:
Monitore, Java-Synchronisation und Verklemmungen

T. Pospíšek

Gesamtüberblick

1. Einführung in Computersysteme
2. Entwicklung von Betriebssystemen
3. Architekturansätze
4. Interruptverarbeitung in Betriebssystemen
5. Prozesse und Threads
6. CPU-Scheduling
- 7. Synchronisation und Kommunikation**
8. Speicherverwaltung
9. Geräte- und Dateiverwaltung
10. Betriebssystemvirtualisierung



Zielsetzung

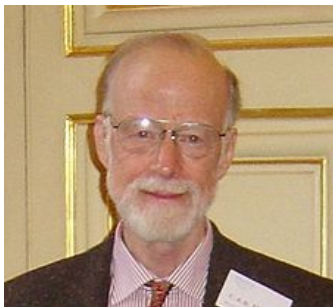
- Der Studierende soll verstehen, wie Monitore funktionieren und welche Vorteile sie gegenüber Semaphoren haben
- Der Studierende soll Java-Monitore und deren Anwendung verstehen
- Der Studierende soll wissen, was Deadlocks sind und wie man sie vermeiden bzw. erkennen und beheben kann

Überblick

1. **Monitore**
2. Java-Synchronisation
3. Deadlocks



1959



Sir Charles Antony Richard Hoare (11.01.1934)
Britischer Computerwissenschaftler



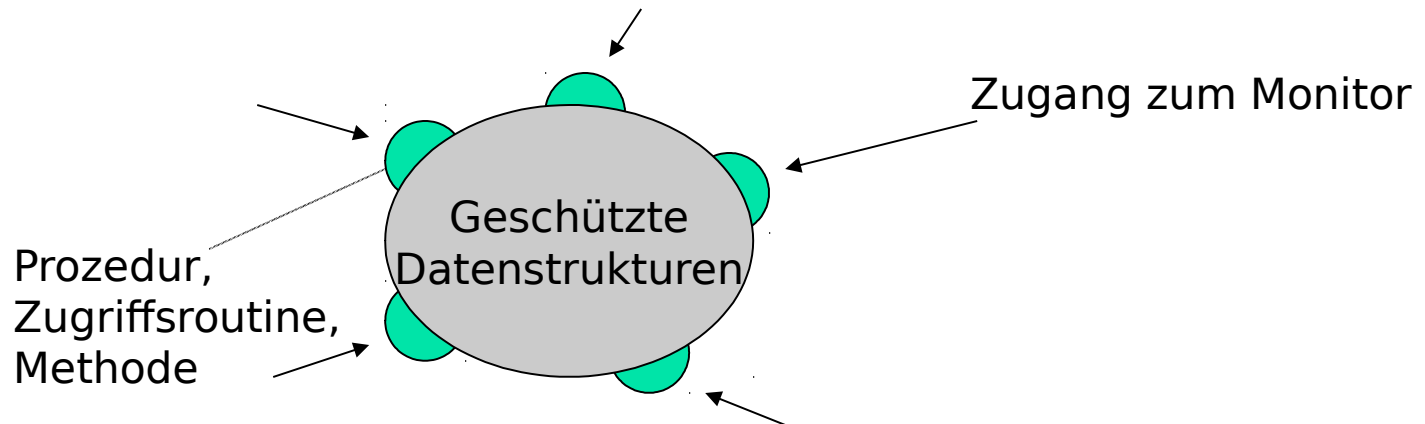
Per Brinch Hansen (13.11.1938 – 31.07.2007),
Dänisch-Amerikanischer Computer-
Wissenschaftler

Monitore

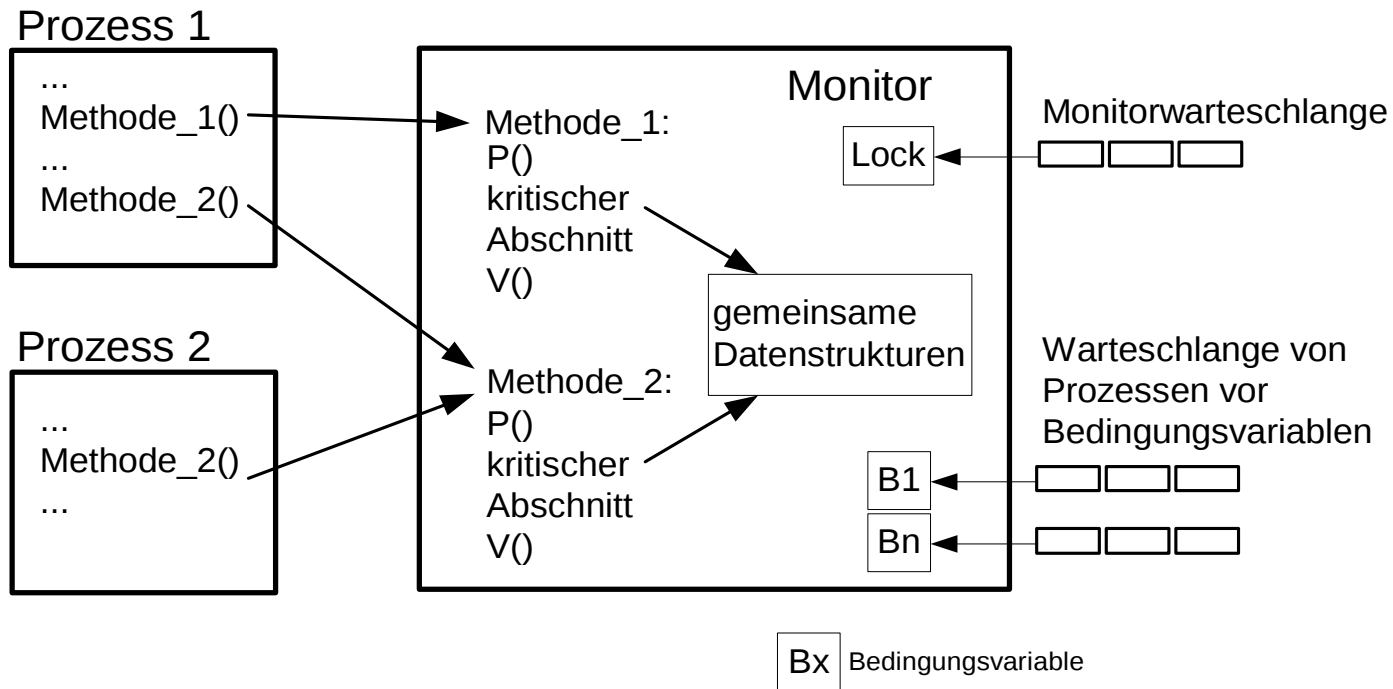
- Die Nutzung von Semaphoren ist zwar eine Erleichterung, aber nicht unproblematisch
 - Man kann als Programmierer eine Operation vergessen
 - Man kann sie versehentlich verwechseln
 - **V()**; ... Kritischer Abschnitt ...; **P()**;
 - führt z.B. dazu, dass alle Prozesse im kritischen Abschnitt zugelassen sind
- **Hoare** und **Brinch Hansen** schlugen daher vor, die Erzeugung und Anordnung der Semaphoroperationen 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

- Nach Richter versteht man unter einem Monitor
 - eine Menge von Prozeduren und Datenstrukturen, die als Betriebsmittel betrachtet werden
 - und mehreren Prozessen zugänglich sind,
 - aber nur von einem Prozess/Thread zu einer Zeit benutzt werden können



Monitore: Grundstruktur



Monitore, Beispielnutzung

Producer-Consumer (1)

Monitor ProducerConsumer

```
{
    final static int N = 5;           // Maximale Puffergröße
    static int count = 0;             // Anzahl gefüllte Pufferbereiche
    condition not_full;               // Signal -> Puffer nicht voll
    condition not_empty;             // Signal -> Puffer nicht leer
    ...

    void insert(item: integer) {
        if (count == N) wait(not_full); // Warten bis Puffer
                                         // nicht mehr voll ist

        // Insert item
        count+=1;
        if (count == 1) signal(not_empty);

    }
    item remove() {
        if (count == 0) wait(not_empty); // Warten bis Puffer
                                         // nicht mehr leer ist

        // Remove item
        count--;
        if (count == (N-1)) then signal(not_full);
        return (item);
    }
}
```

Die condition-Variablen werden in zwei Operationen genutzt:

- **signal**
- **wait**

Bei Aufruf von **wait** wird der Monitor verlassen.
Zwei Ansätze:

- **signal-and-continue**
- **signal-and-wait**

Ein anderer Prozess kann ihn betreten → wichtig!

Quelle: Tanenbaum, A. S.: Moderne Betriebssysteme, 3. aktualisierte Auflage, Pearson Studium, 2009

Monitore, Beispielnutzung

Producer-Consumer (2)

```
class UseMonitor
{
    ProducerConsumer mon = new
    ProducerConsumer();
    ...
    void producer() {
        while (true) {
            // produce item
            mon.insert(item);
        }
    }

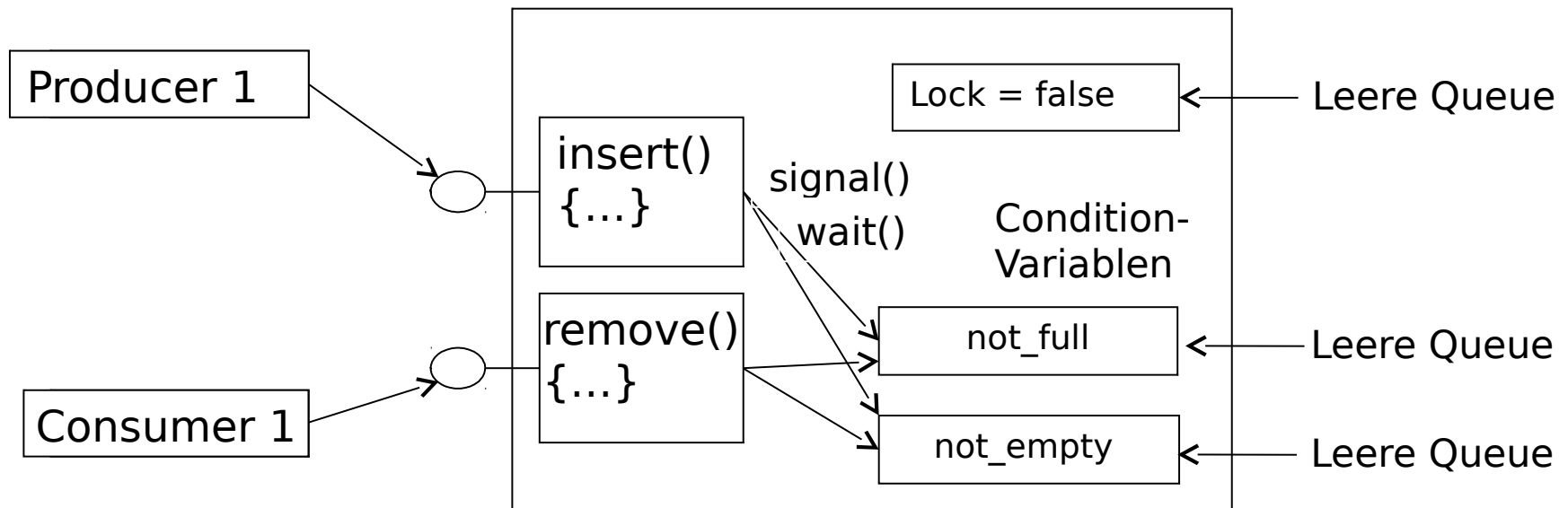
    void consumer() {
        while (true) {
            item = mon.remove();
            // consume item
        }
    }
}
```

Quelle: *Tanenbaum, A. S.*: Moderne Betriebssysteme, 3. aktualisierte Auflage, Pearson Studium, 2009

Monitore, Producer-Consumer, Szenario 1 (1)

- Initialzustand: Nichts produziert, nichts konsumiert

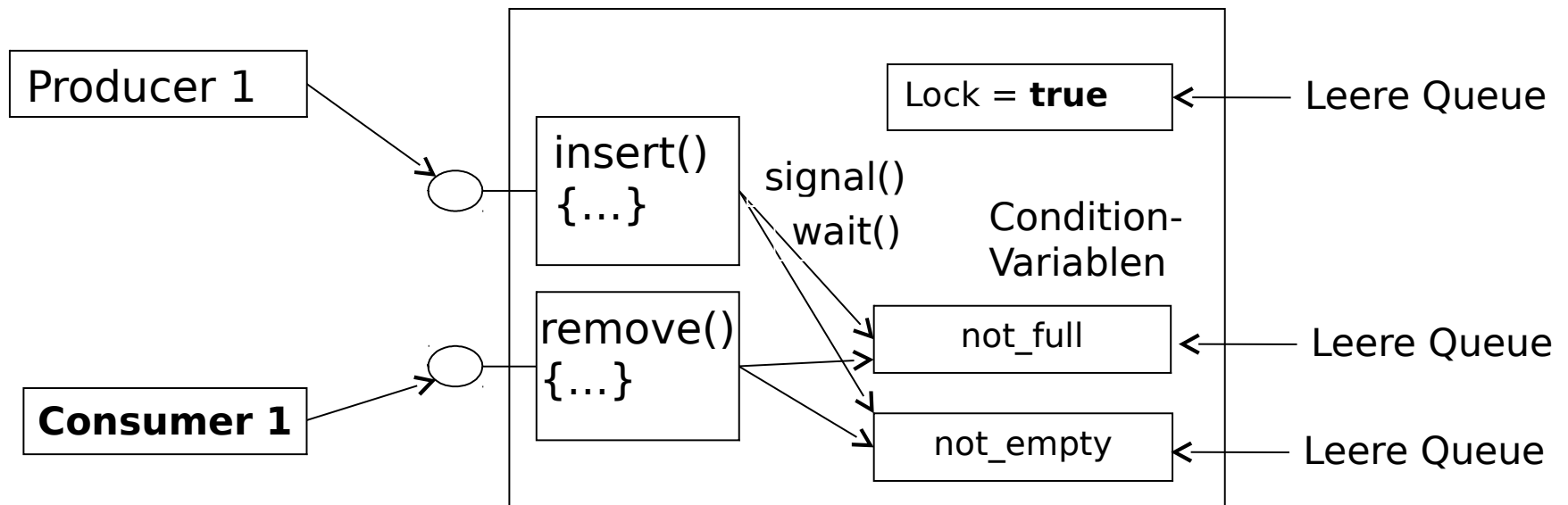
Monitor ProducerConsumer



Monitore, Producer-Consumer, Szenario 1 (2)

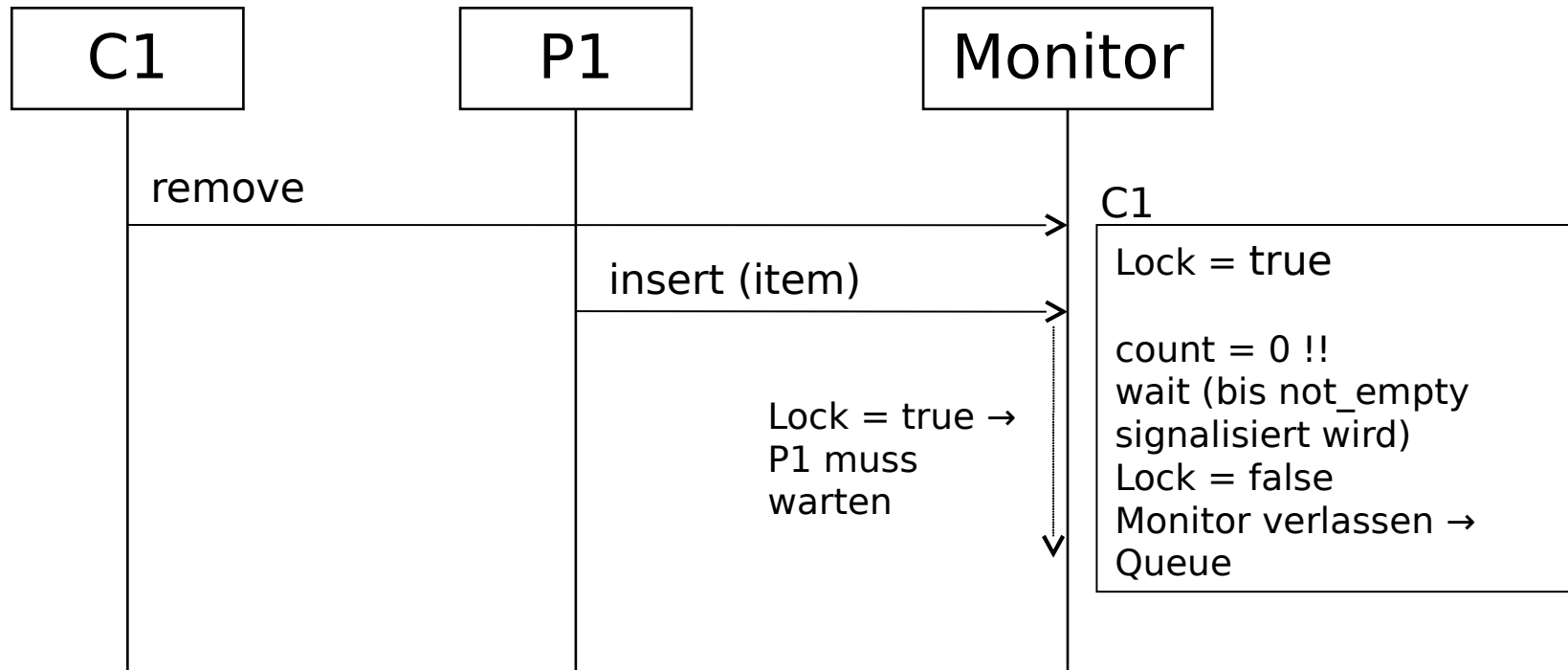
- Consumer 1 möchte als erster konsumieren

Monitor ProducerConsumer



Monitore, Producer-Consumer, Szenario 1 (3)

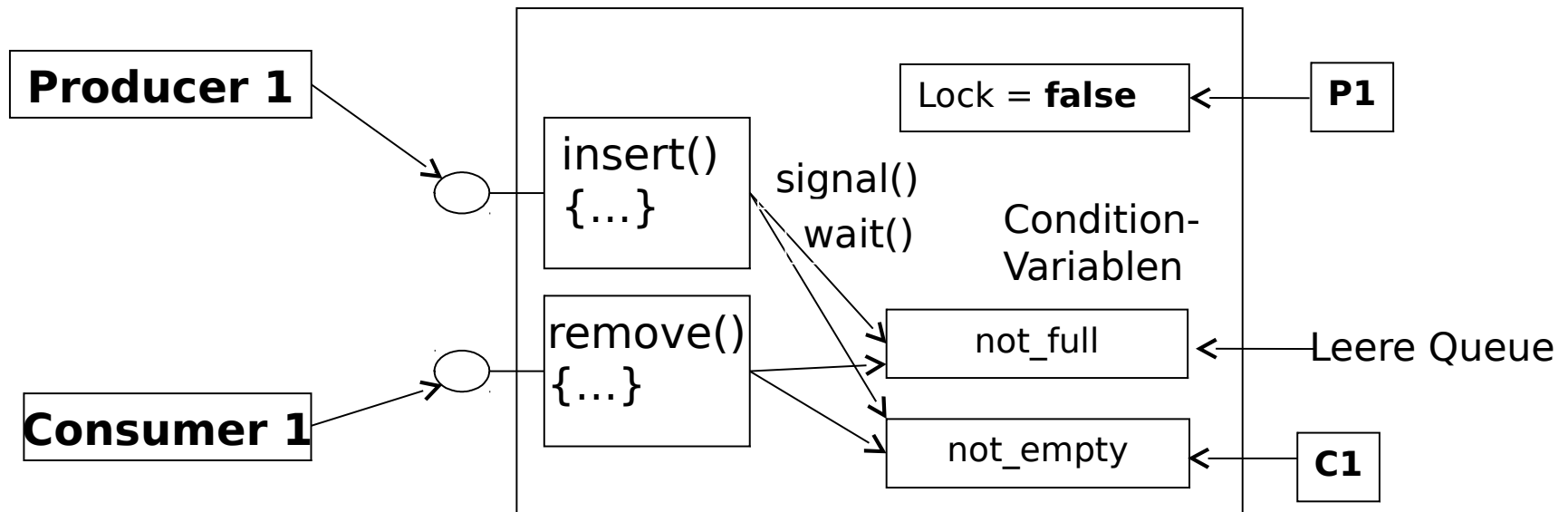
- Consumer 1 möchte als erster konsumieren, Producer 1 kommt später und muss warten



Monitore, Producer-Consumer, Szenario 1 (4)

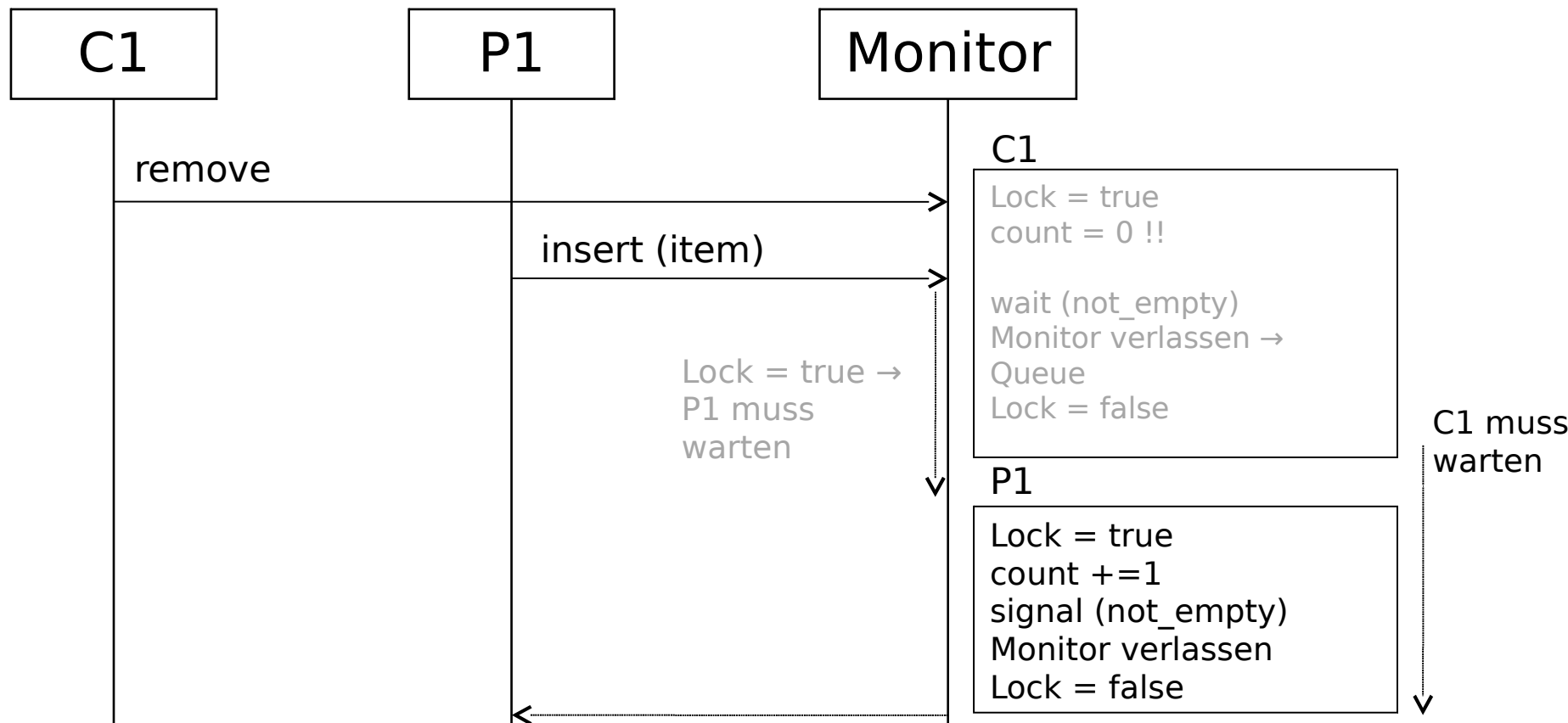
- Consumer 1 kann nichts vorfinden (kein Item)
- Producer 1 wartet bis Consumer 1 den Monitor verlassen hat und betritt ihn dann

Monitor ProducerConsumer



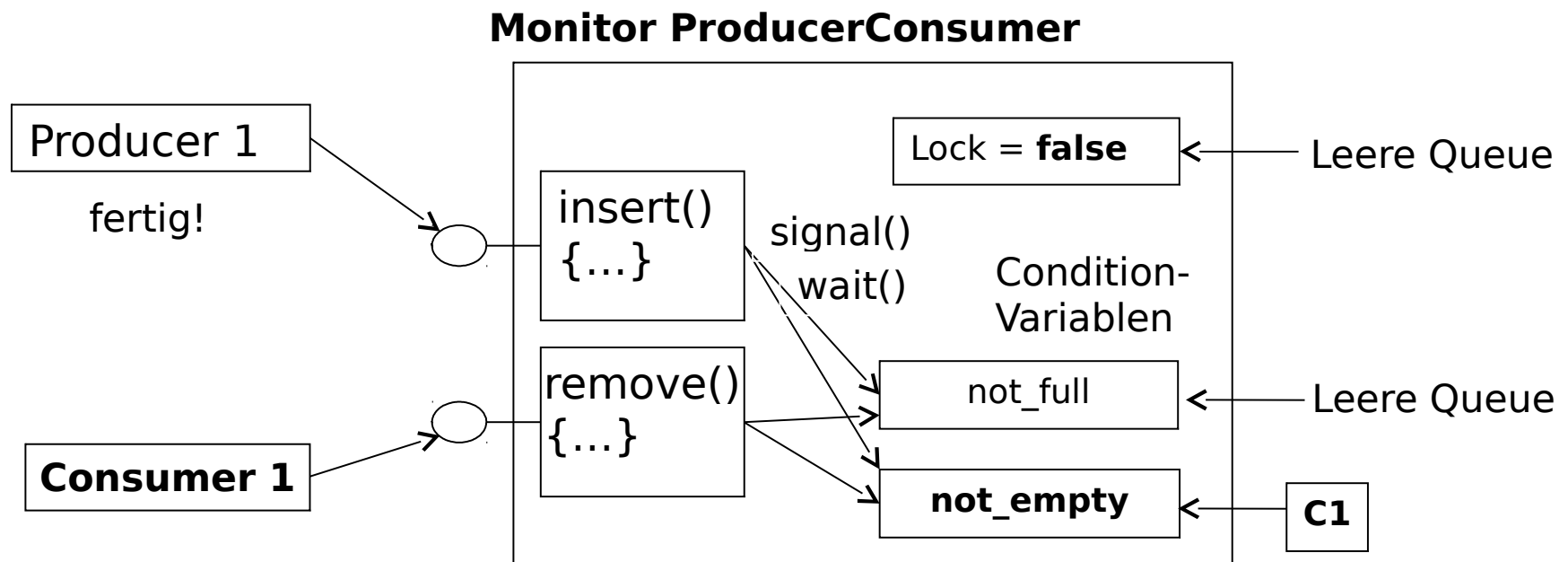
Monitore, Producer-Consumer, Szenario 1 (5)

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



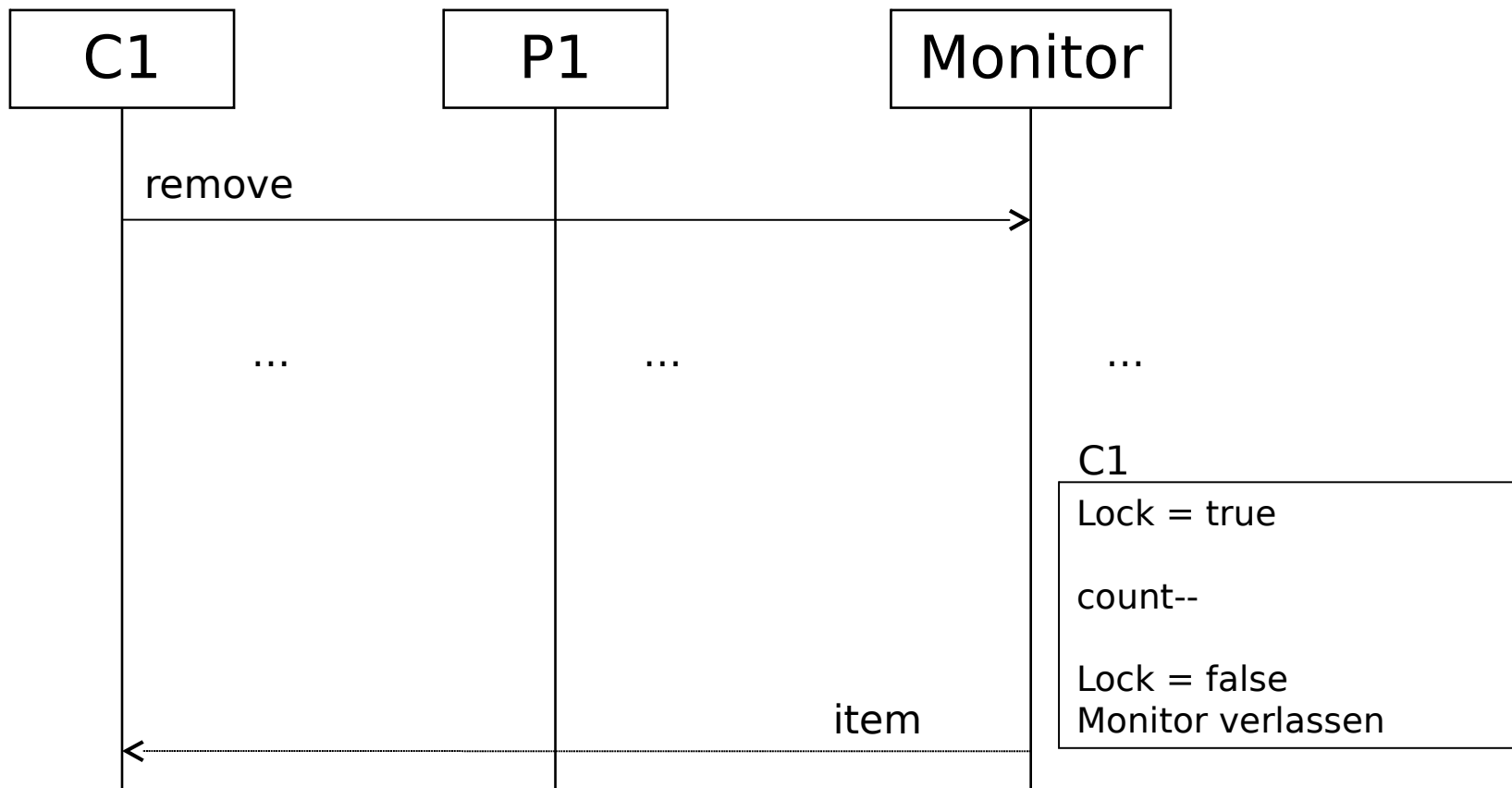
Monitore, Producer-Consumer, Szenario 1 (6)

- Consumer 1 kann nun konsumieren



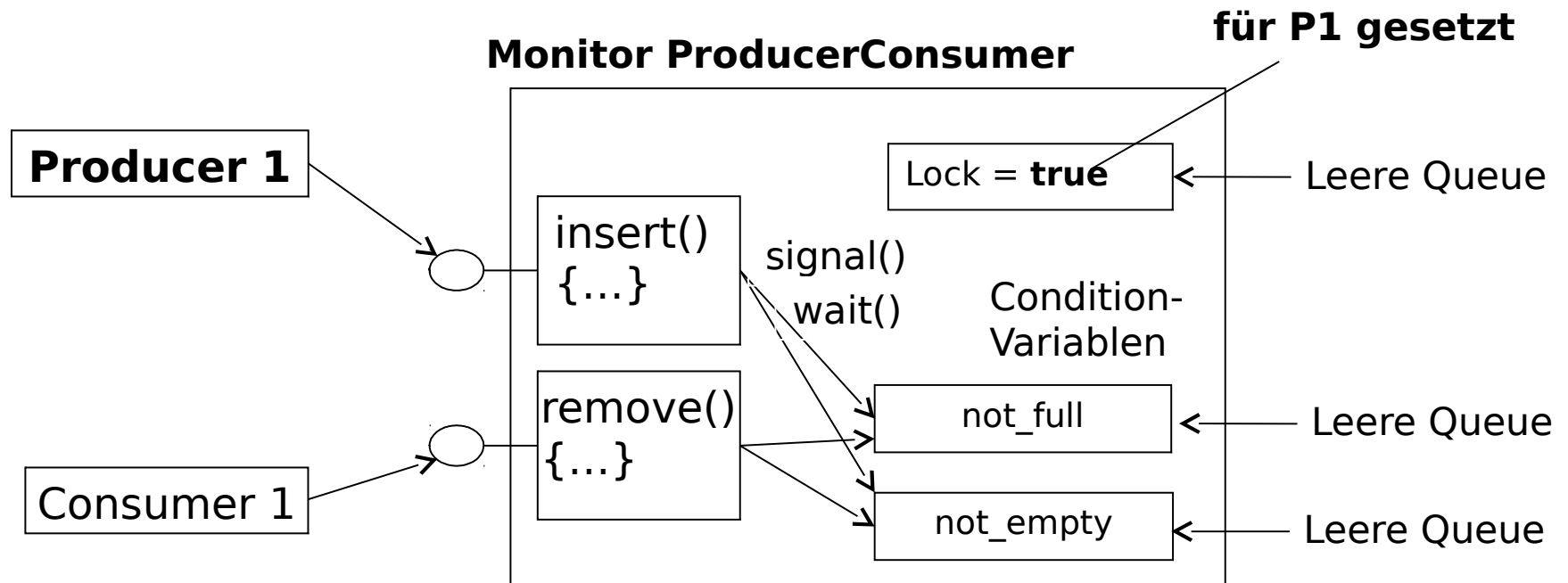
Monitore, Producer-Consumer, Szenario 1 (7)

- Consumer 1 konsumiert



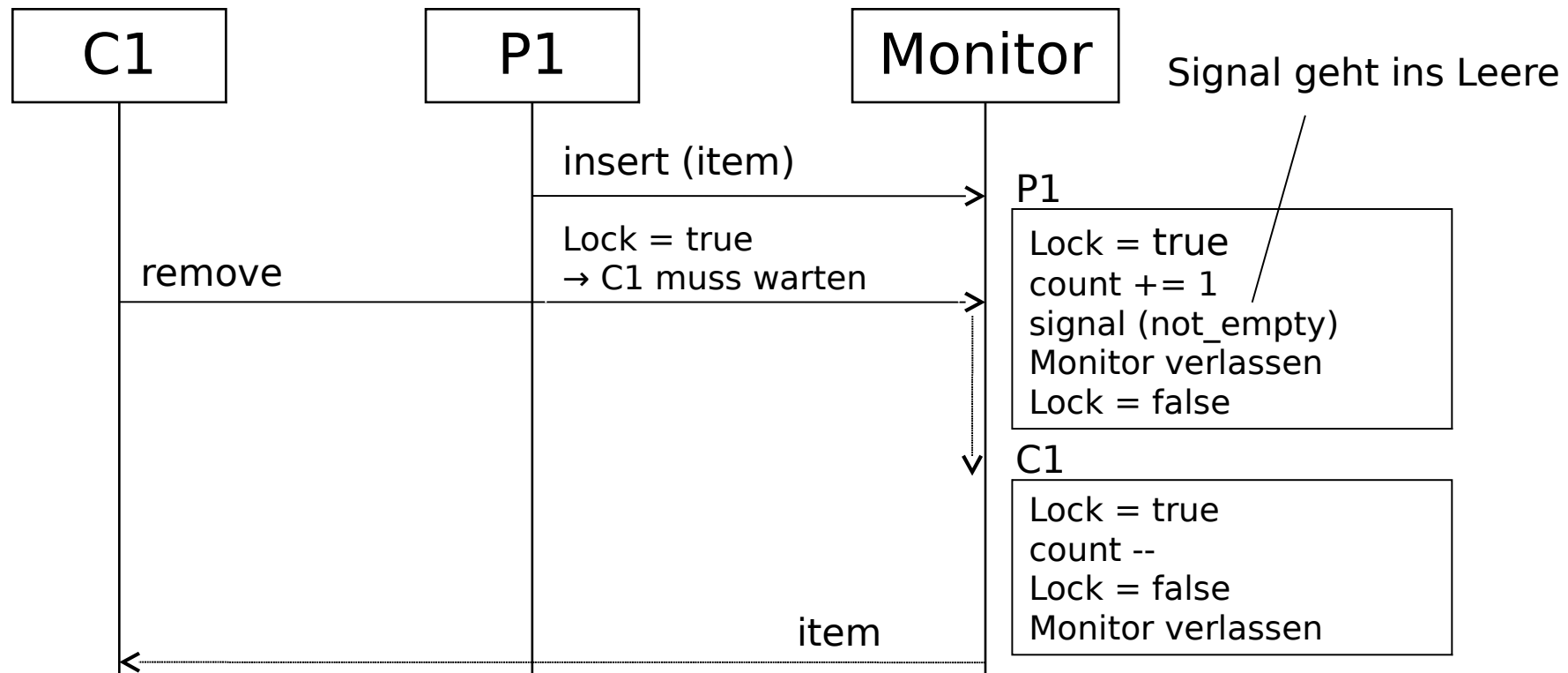
Monitore, Producer-Consumer, Szenario 2 (1)

- Anderes Szenario: Producer 1 möchte als erster produzieren



Monitore, Producer-Consumer, Szenario 2 (2)

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



Überblick

1. Monitore
- 2. Java-Synchronisation**
3. Deadlocks

Java:

Die Synchronisationsprimitive `synchronized`

- Der Modifier *synchronized* dient der Festlegung kritischer Abschnitte:
 - einzelne Codeblöcke
 - Methoden eines Objekts
- Anmerkung:
 - **Thread-safe** (Syn.: eintrittsinvariant, reentrant, wiedereintrittsfähig) heißt eine Klasse oder Methode, wenn sie bedenkenlos in einer nebenläufigen Thread-Umgebung genutzt werden kann
 - Problematisch: Klassenvariablen, globale Variablen

Java-Monitore:

Synchronisationsprimitive synchronized

- Zugriffsserialisierte Methode

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

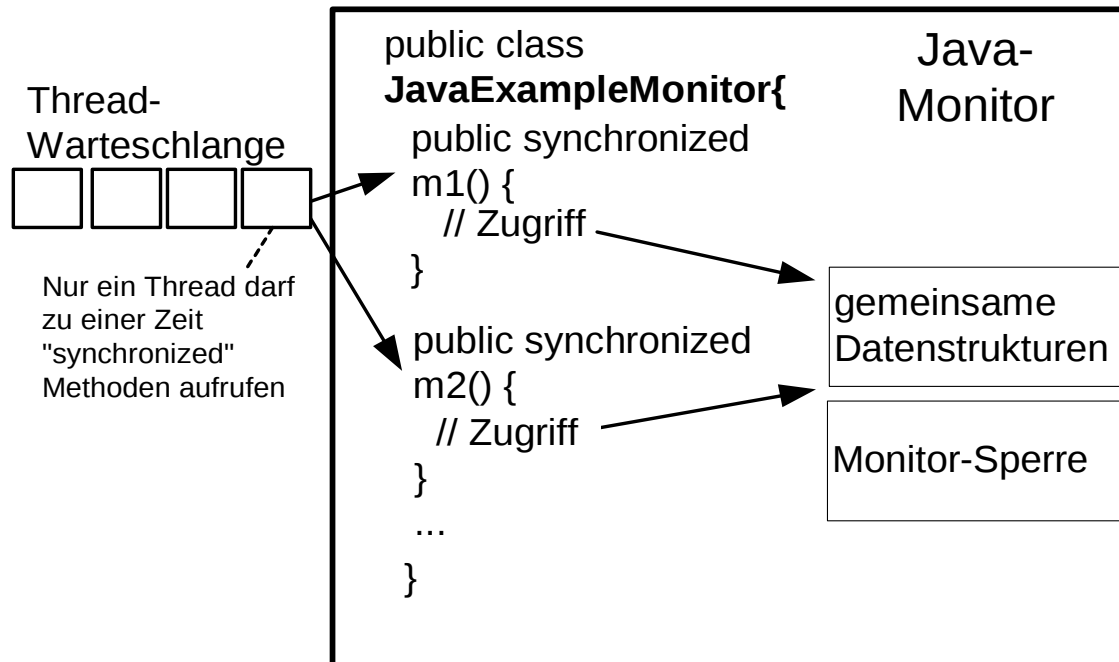
- Zugriffsserialisierter Anweisungsblock

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

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

Monitore: Java-Monitor

signal-and-continue



Java-Monitore:

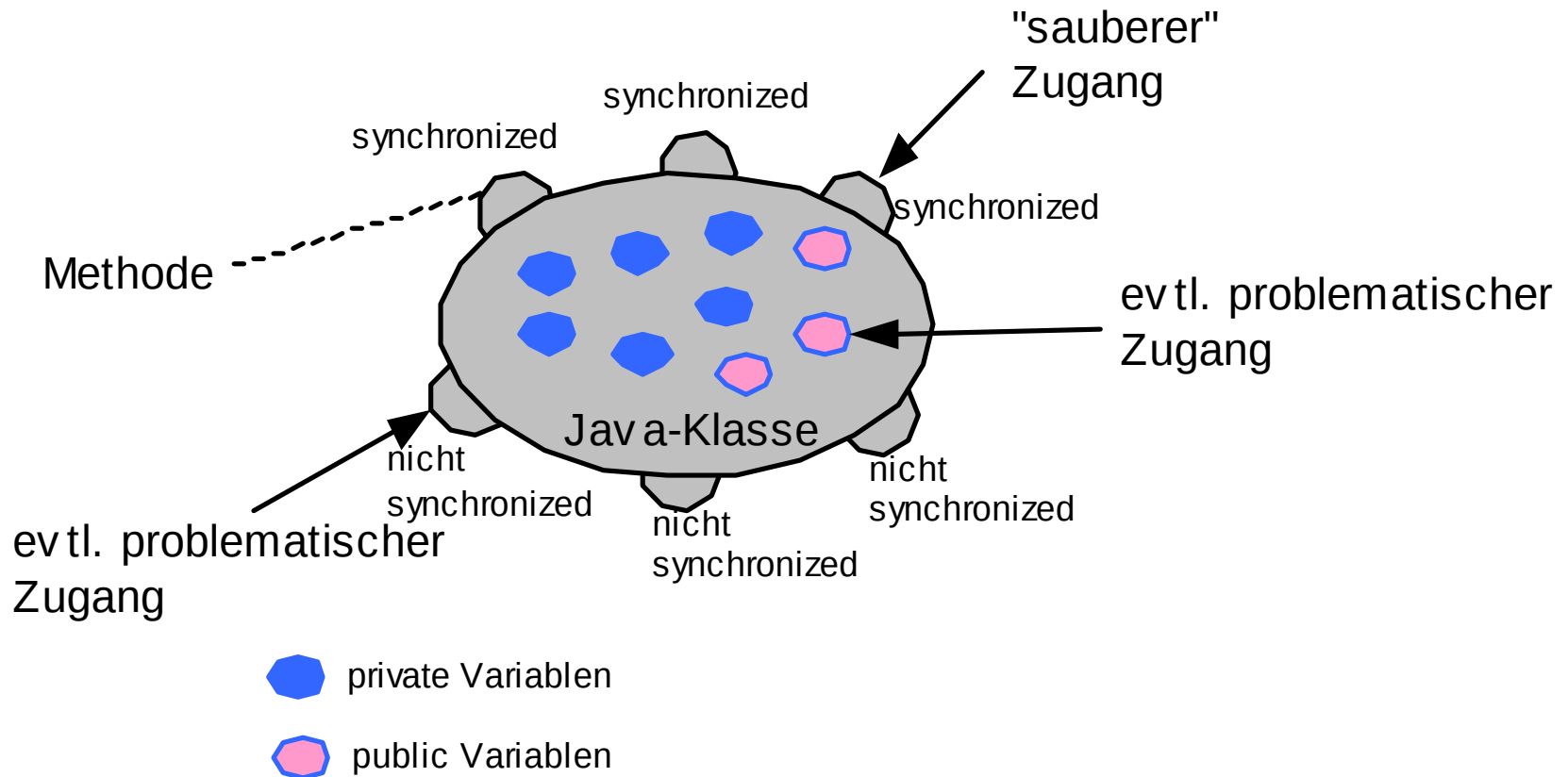
Synchronisationsprimitive `synchronized`

- 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 eine *synchronized*-Methode zugreift
- Sperre wird aufgehoben, wenn Thread die Methode verlässt

Java-Monitore: Kritikpunkte



Threads in Java (Wiederholung)

- 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

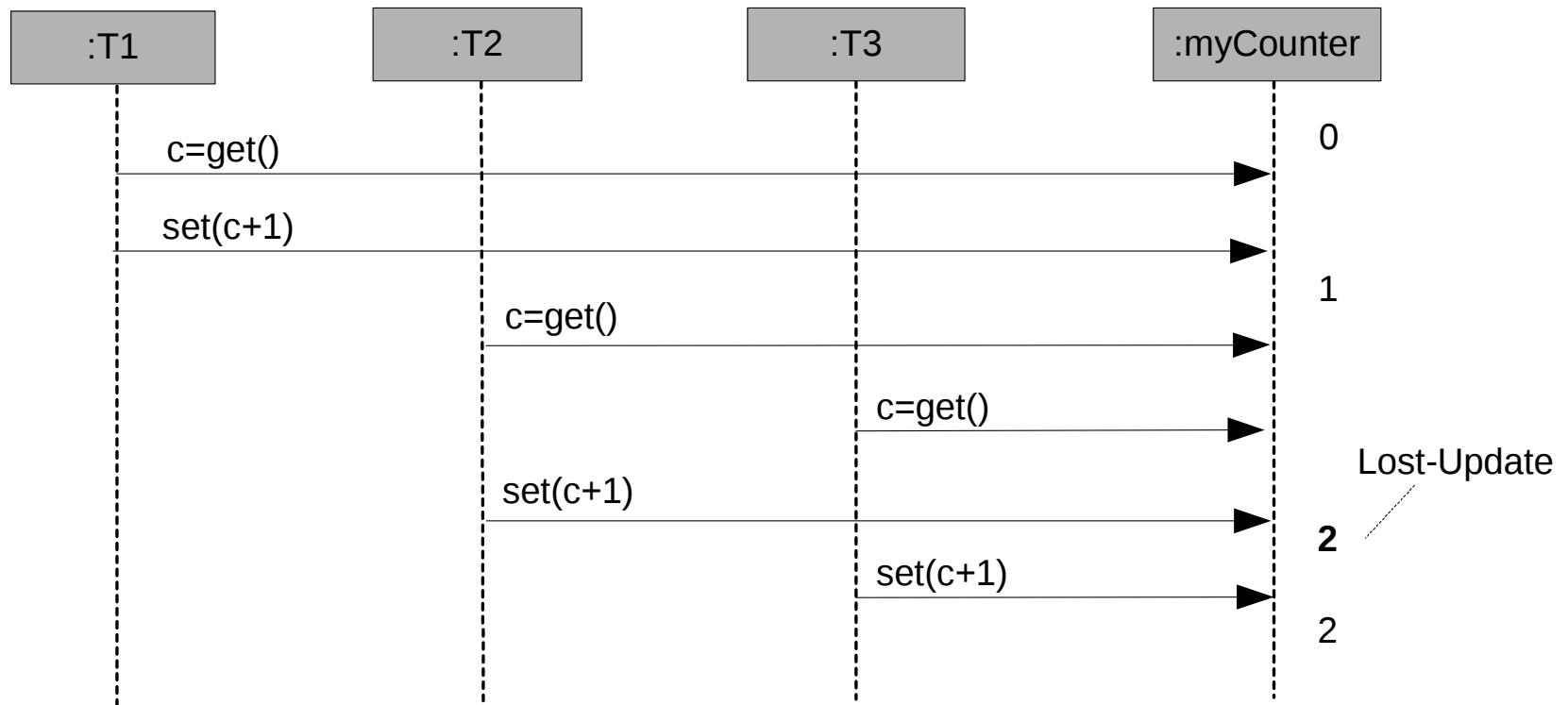
Beispiel:

Zähler durch mehrere Threads hochzählen

- Eine einfache Objektklasse verwaltet einen Zähler
- Mehrere Threads greifen über die Methoden `get()` und `set()` auf den Zähler zu
- Zunächst wird das Problem ohne Synchronisierung gelöst
 - Was passiert?
- Dann wird der kritische Abschnitt synchronisiert
 - Was passiert nun?
- Bringen Sie die Anwendung unter Windows zum Laufen und testen sie diese mit verschiedenen Parametern

Beispiel:

Zähler durch mehrere Threads hochzählen



Beispiel: Zähler durch mehrere Threads hoch zählen

■ Code für Counter

```
package Threads;
import java.io.*;

class CounterObject {
    private int count = 0;

    CounterObject() {
        // Nichts zu konstruieren
    }

    void set(int newCount) {
        count = newCount;
    }

    int get() {
        return count;
    }
}
```

Beispiel: Zähler durch mehrere Threads hochzählen

- Code für Thread-Klasse (nicht synchronisiert)

```
class CountThread1 extends Thread {
    private CounterObject myCounter;
    private int myMaxCount;
    CountThread1(CounterObject c, int maxCount)
    {
        myCounter = c;
        myMaxCount = maxCount;
    }
    public void run() {
        System.out.println("Thread " + getName() + " gestartet");
        for (int i=0;i<myMaxCount;i++){

            // Kritischer Bereich
            int c = myCounter.get();
            c++;
            myCounter.set(c);
            // Ende des kritischen Bereichs

        }
    }
}
```

Beispiel: Zähler durch mehrere Threads hochzählen

- Code für Thread-Klasse (synchronisiert)

```
class CountThread2 extends Thread {  
    private CounterObject myCounter;  
    private int myMaxCount;  
  
    CountThread2(CounterObject c, int maxCount) {  
        myCounter = c;  
        myMaxCount = maxCount;  
    }  
    public void run() {  
        System.out.println("Thread " + getName() + " gestartet");  
        for (int i=0;i<myMaxCount;i++){  
            synchronized (myCounter) {  
                int c = myCounter.get();  
                c++;  
                myCounter.set(c);  
            }  
        }  
    }  
}
```

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
 - Lösung: wait() in while-Schleife aufrufen
- notify() **garantiert nicht**, dass genau ein Thread aufgeweckt wird
- **Kein Warteschlangenmechanismus** implementiert, also nicht absolut fair

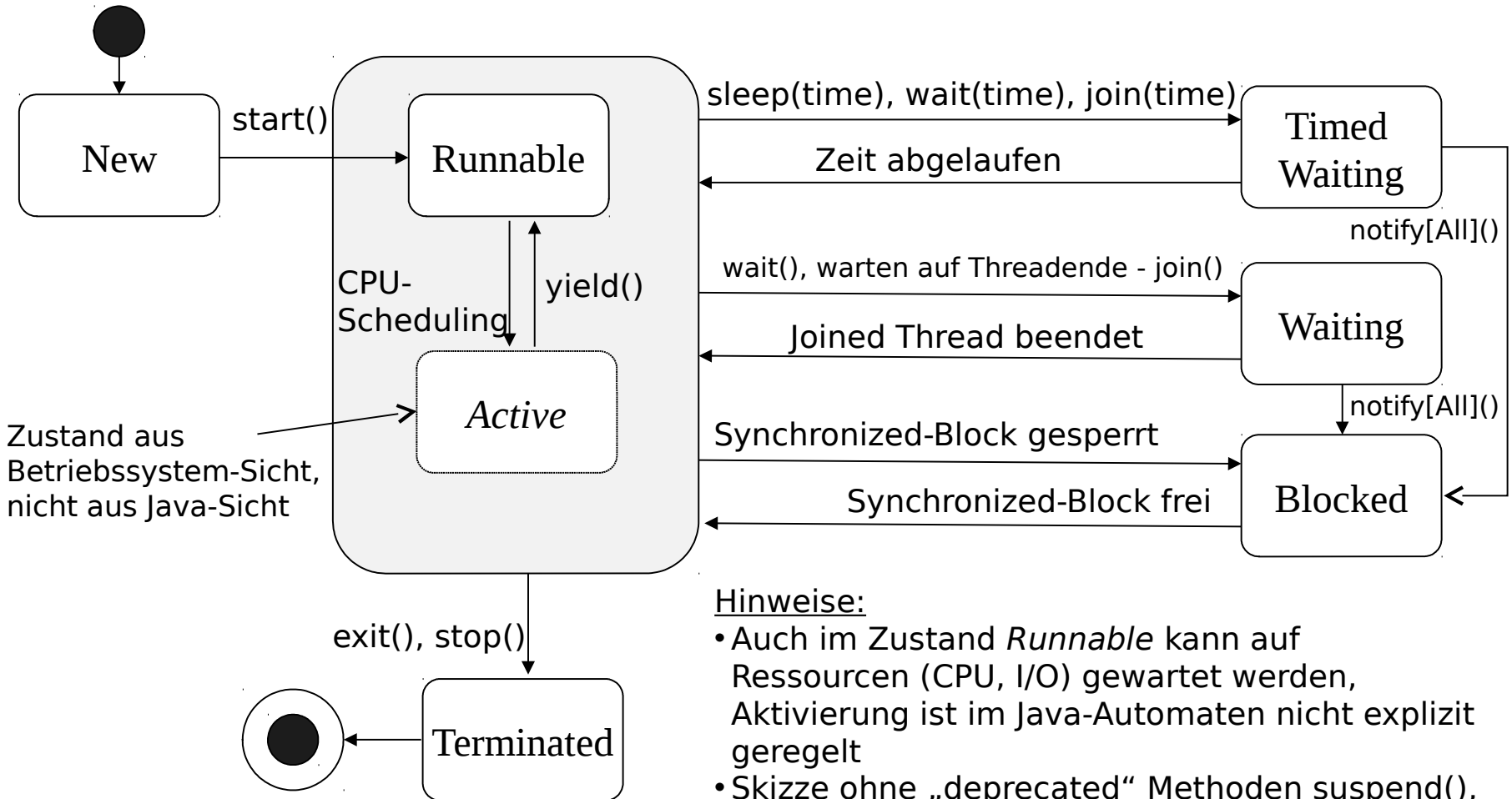
Java-Thread-Zustände gemäß API (1) (siehe Enum Thread.State)

- **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



Beispiel zur expliziten Synchronisation

Eigene Semaphor-Implementierung

```
class MySemaphore {  
    private int max;           // Anzahl der maximal möglichen Threads im kritischen Abschnitt  
    private int free;         // Anzahl der verfügbaren Plätze im kritischen Abschnitt (so viele  
                                // Threads dürfen noch in den kritischen Abschnitt rein)  
    private int waiting;      // Anzahl der in der P-Operation mit wait wartenden Threads (so  
                                // viele Threads möchten aktuell in den kritischen Abschnitt rein)  
  
    public MySemaphore() {  
        this(0);  
    }  
    public MySemaphore(int i) {  
        if (i >= 0) {  
            max = i;  
        } else {  
            max = 0;  
        }  
        free = max;  
        waiting = 0;  
    }  
  
    protected void finalize() throws Throwable {...}
```

Beispiel zur expliziten Synchronisation Semaphor-Implementierung

```
/**
 * P-Operation
 */
public synchronized void P()
{
    while (free <= 0)
    {
        waiting++;          // Anzahl der wartenden Threads erhöhen
        try {
            this.wait();
        } catch (InterruptedException e) {}

        waiting--;          // Anzahl der wartenden Threads vermindern
    }
    free--;                  // Jetzt erst Semaphorzähler vermindern
}
```

Beispiel zur expliziten Synchronisation Semaphor-Implementierung

```
/**
 * V-Operation
 */
public synchronized void V()
{
    free++; // Semaphorzähler erhöhen

    if (waiting > 0)
    {
        // Nur wenn ein anderer Thread wartet,
        // diesen mit notify benachrichtigen
        this.notify();
    }
}
```

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!

Überblick

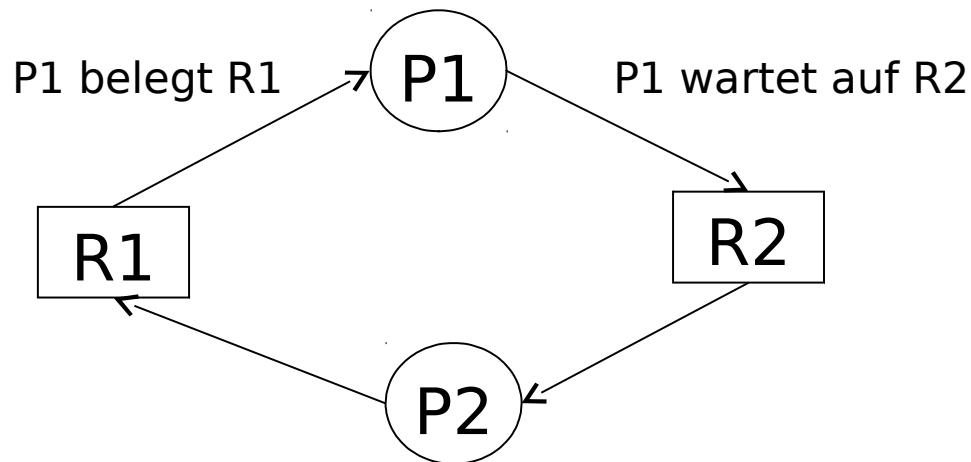
1. Monitore
2. Java-Synchronisation
- 3. Deadlocks**

Deadlocks: Definition

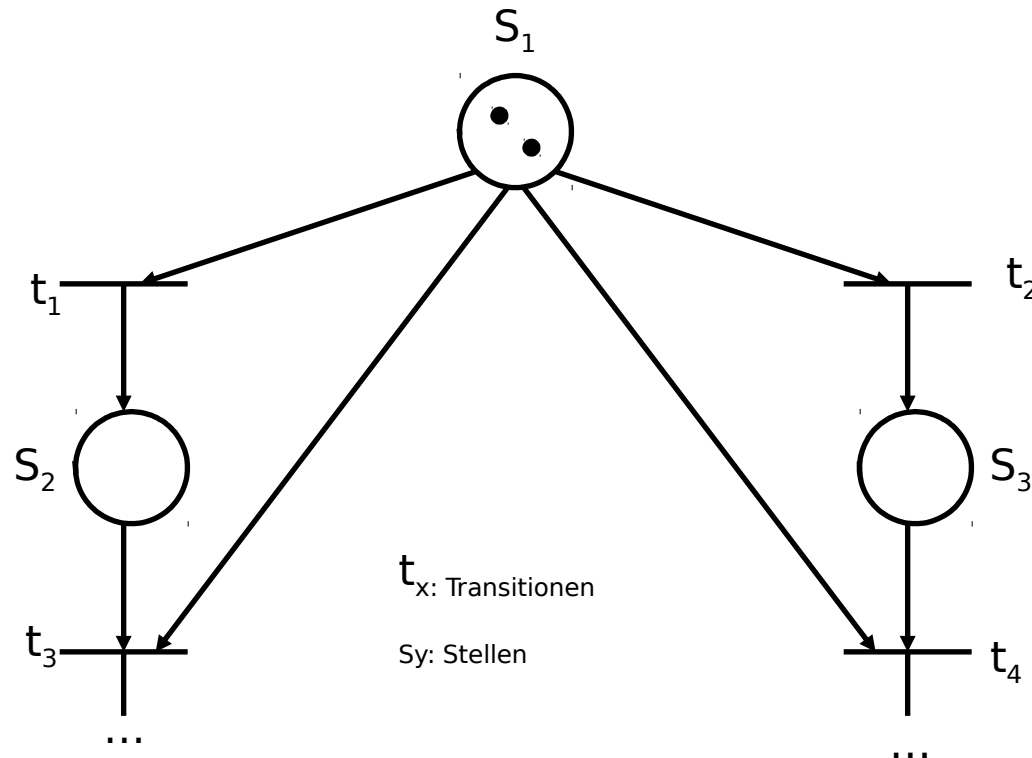
- Prozesse bzw. Threads können sich gegenseitig behindern oder sogar blockieren
 - Programme können nicht mehr ausgeführt werden
- Das typische Problem:
 - Prozess P1 hat das Betriebsmittel B1 reserviert und wartet auf B2, dass aber von P2 reserviert ist
 - P2 wiederum wartet auf B1
 - Beide Prozesse 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

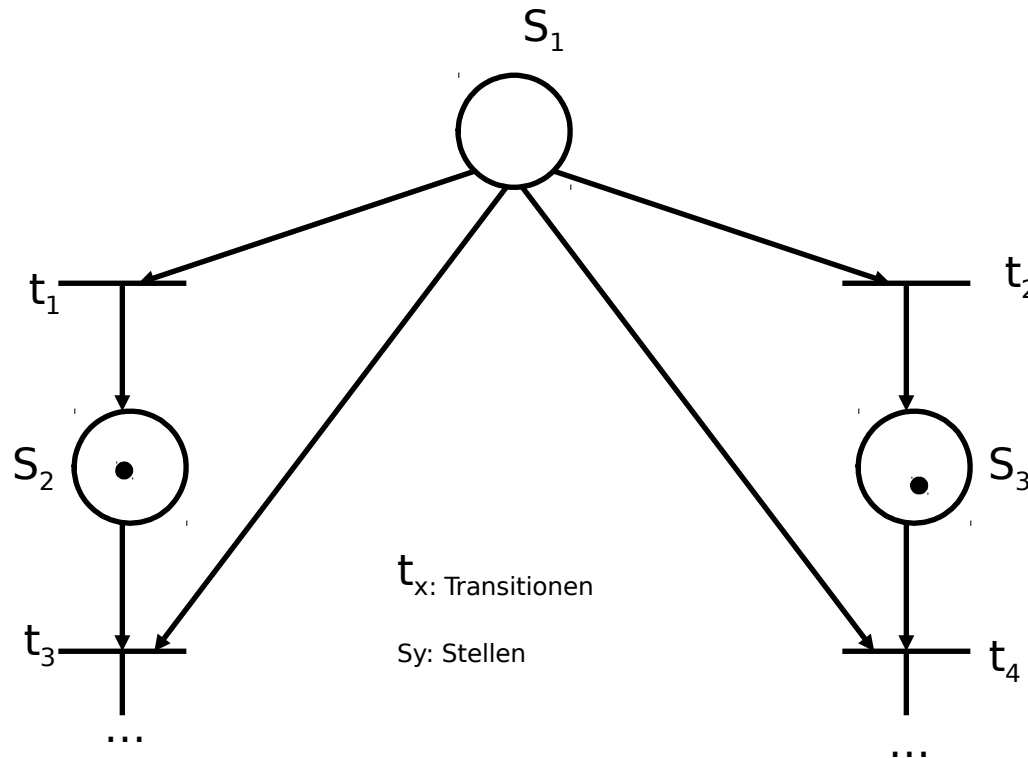


Deadlock als Petrinetz: vor der Deadlocksituation



- Wenn t_1 und t_2 gleichzeitig schalten, liegt ein Deadlock vor, da S_1 nicht mehr feuern kann

Deadlock als Petrinetz: Deadlocksituation



- Wenn t_1 und t_2 gleichzeitig schalten, liegt ein Deadlock vor, da S_1 nicht mehr feuern kann

Deadlock: Bedingungen

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

Java-Beispiel: Deadlocksituation

```
public class myDeadlock {  
    public static void main(String[] args) {  
        final Object resource1 = new Object(); // Dummy-Objekte nur zur Demonstration  
        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, aber erkennen
 - Deadlockerkennung über Betriebsmittelbelegungsgraphen
 - Beheben durch:
 - Rollback
 - Prozessabbruch
 - Transaktionsabbruch
 - **Dynamisches Verhindern**
 - Ressourcen vorsichtig zuteilen
 - **Vermeiden**
 - Eine der vier Bedingungen muss unerfüllt bleiben

Überblick und Zusammenfassung

- ✓ Monitore
- ✓ Java-Synchronisation
- ✓ Deadlocks