Multithreading

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

---

Steffen Helke
Technische Universität Berlin
Fachgebiet Softwaretechnik

10. Juni 2013

Übersicht

- Rückblick: Producer-Consumer-Problem
- Synchronisationsmechanismen
- Thread-Pools

Teil IV der Vorlesung PROG 2

Multithreading

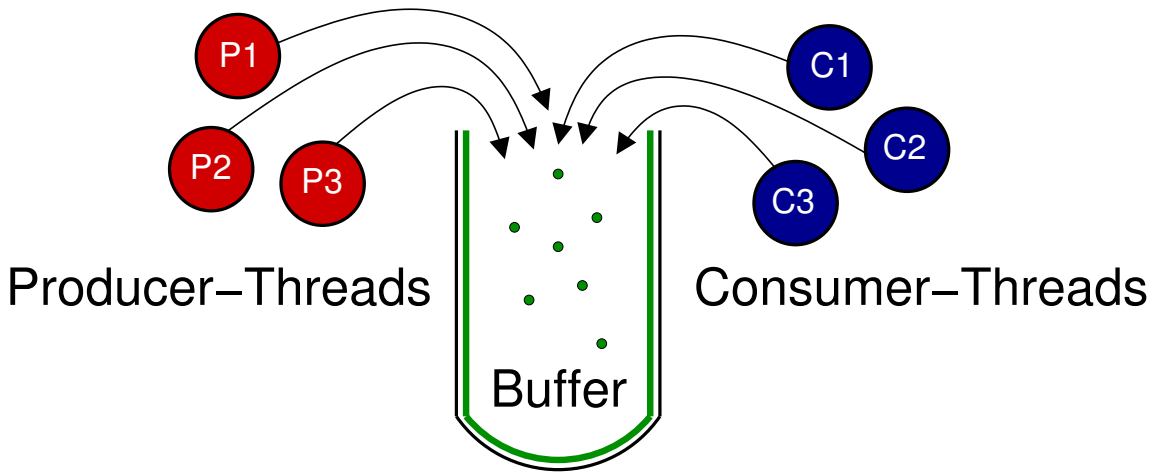
Producer-Consumer-Problem

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Andreas Rausch, TU Clausthal, Programmierkurs SS 2011*

Producer-Consumer-Problem

Problembereich

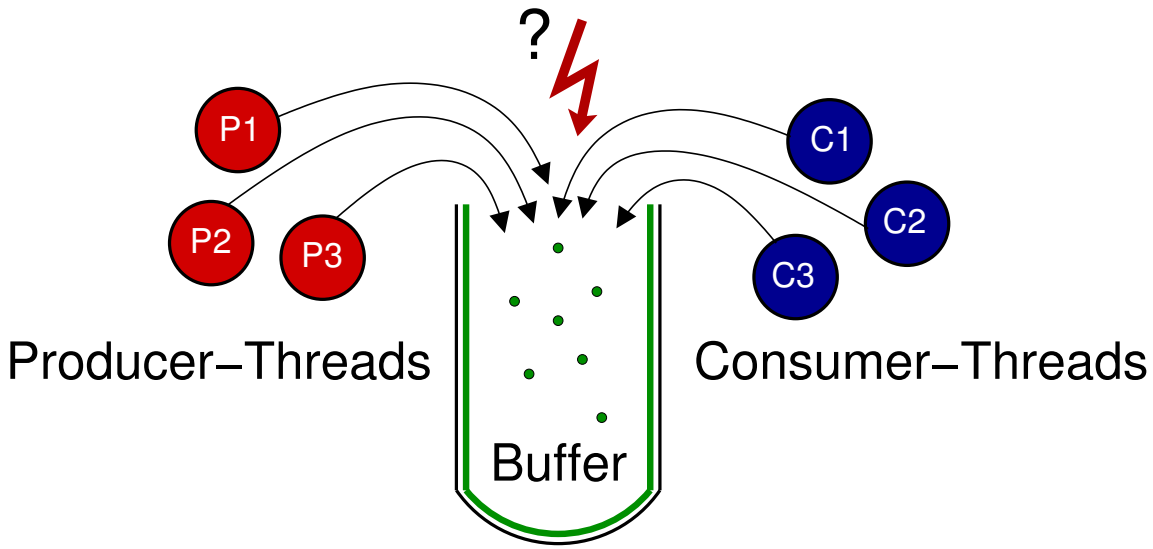
- eine feste Anzahl von Threads (Producer) erzeugen Elemente für gemeinsame Datenstruktur (Buffer)
- eine feste Anzahl von Threads (Consumer) entnehmen Elemente aus der gemeinsam genutzten Datenstruktur
- die Datenstruktur hat eine beschränkte Kapazität



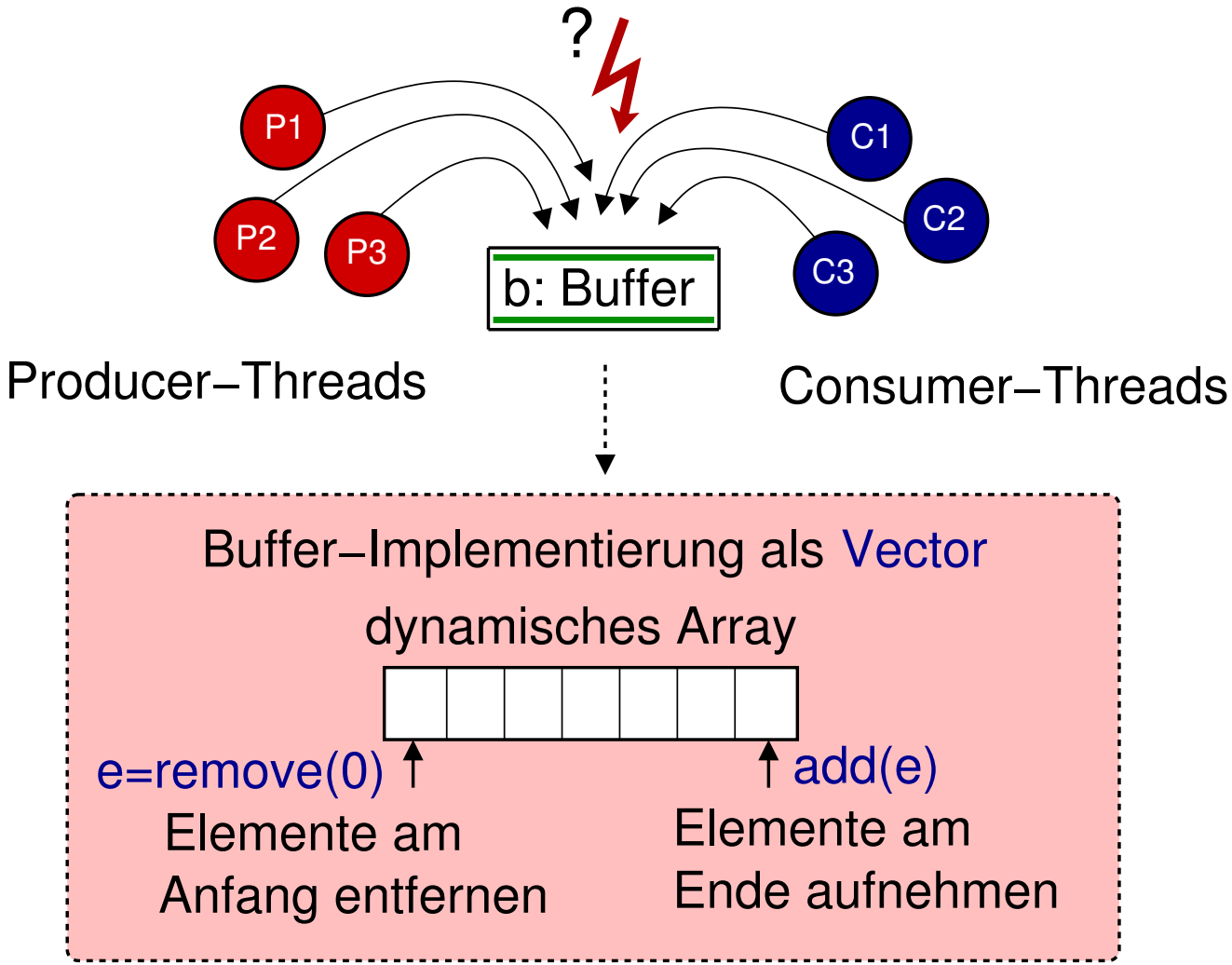
Konfliktvermeidung

Synchronisationsbedarf

- der Lese- bzw. Schreibvorgang eines Threads darf nicht durch andere Threads unterbrochen werden (Inkonsistenzen)
- zugreifende Consumer-Threads werden blockiert, wenn die Datenstruktur leer ist
- zugreifende Producer-Threads werden blockiert, wenn die Datenstrukur voll ist



Implementierung der Datenstruktur (Buffer)



Java-Code Datenstruktur (Buffer)

```
public class Buffer {
  private Vector<String> queue;
  final int MAX = 3;

  public Buffer() {
    queue = new Vector<String>(); }

  public void transferElementToBuffer (String element) {
      if (queue.size() >= MAX) {
        throw new RuntimeException ("Kein Platz"); }
     queue.add(element);
     System.out.println(element + " wurde abgelegt");
  }
  public String takeElementFromBuffer () {
      if (queue.isEmpty()) {
        throw new RuntimeException ("Puffer leer"); }
     String element = queue.remove(0);
     System.out.println(element + " wurde entnommen");
     return element;
  }
}
```

Producer-Klasse zum Auffüllen des Buffers

```
public class Producer extends Thread {
 private Buffer buffer;
 private int sleepTime;

 public Producer (Buffer buffer, int sleepTime) {
   this.buffer = buffer;
   this.sleepTime = sleepTime; }

 public void run() {
   for (int i = 0; i < 10; i++) {
      try {
            Thread.sleep(sleepTime);
            String element = new String("Element " + i);
             buffer.transferElementToBuffer(element);
      } catch (InterruptedException e) {
              e.printStackTrace(); }
   }
  }
}
```

Consumer-Klasse zum Reduzieren des Buffers

```
public class Consumer extends Thread {
  private Buffer buffer;
  private int sleepTime;

  public Consumer (Buffer buffer, int sleepTime) {
    this.buffer = buffer;
    this.sleepTime = sleepTime; }

    public void run() {
      for (int i = 0; i < 10; i++) {
        try {
            Thread.sleep(sleepTime);
             buffer.takeElementFromBuffer();
        } catch (InterruptedException e) {
             e.printStackTrace();}
      }
   }
}
```

Consumer-Producer-Klasse

```
public class ConsumerProducerProblem {
  public static void main (String[] args) {
     Buffer buffer = new Buffer();
     Producer producer = new Producer(buffer, 200);
     Consumer consumer = new Consumer(buffer, 1000);

     producer.start();
     consumer.start();
  }
}
```

Probleme

- Exceptions werden sowohl beim Entnehmen, als auch beim Einfügen geworfen
- Zugriffe sind nicht atomar (Gefahr von Inkonsistenzen)

➜ zusätzliche Synchronisation nötig

Lösungsansatz: Einsatz von Semaphoren

Idee

- Mittel zur Synchronisation von nebenläufig ausgeführten Threads mit gemeinsamen Speicher
- Idee bereits 1965 von Edsger W. Dijkstra veröffentlicht

Bestandteile einer Semaphore

- Zähler
- Warteschlange
- Methode *P()* zum Dekrementieren des Zählers
- Methode *V()* zum Inkrementieren des Zählers

Abarbeitung beim Einsatz von Semaphoren

Vorgehen

1. Initialisierung des Zählers (maximal zugelassene Anzahl an Threads im kritischen Bereich)
2. vor Zugriff auf kritischen Bereich muss ein Thread t die Methode *P()* aufrufen
3. ist Zähler bereits 0, wird Thread t in Warteschlange eingefügt, sonst bekommt er Zugriff auf kritischen Bereich
4. ist Thread t fertig, so ruft er Methode *V()* auf
5. falls Threads in der Warteschlange, erhält der nächste Zugriff

Semaphoren in Java

Notation

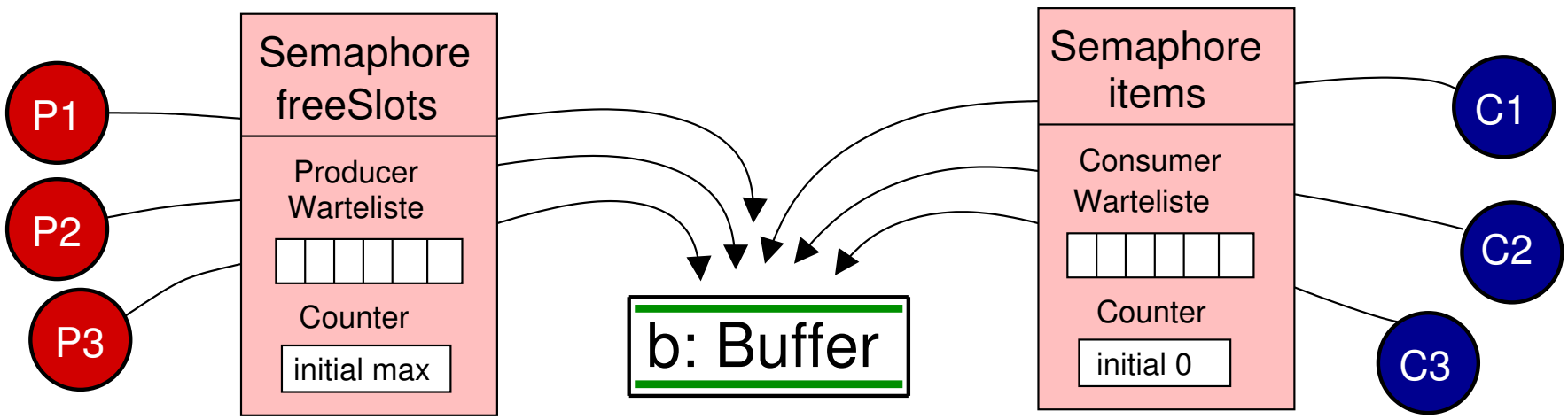
- Methoden *P()* und *V()* werden umbenannt
- zusätzliche Einführung von Methoden, zum Inkrementieren oder Dekrementieren um mehr als 1 vornehmen zu können

```
// Dekrementieren des internen Zählers
void acquire();

// Inkrementieren des internen Zählers
void release();

// Dekrementieren um mehrere Schritte
void acquire(int permits);

// Inkrementieren um mehrere Schritte
void release(int permits);
```

Semaphoren für Producer-Consumer-Beispiel

Idee

- Zugriffskontrolle mit Hilfe von zwei Semaphoren
- Producer-Kontrolle mit *freeSlots* (initial *max*)
- Consumer-Kontrolle mit *items* (initial 0)



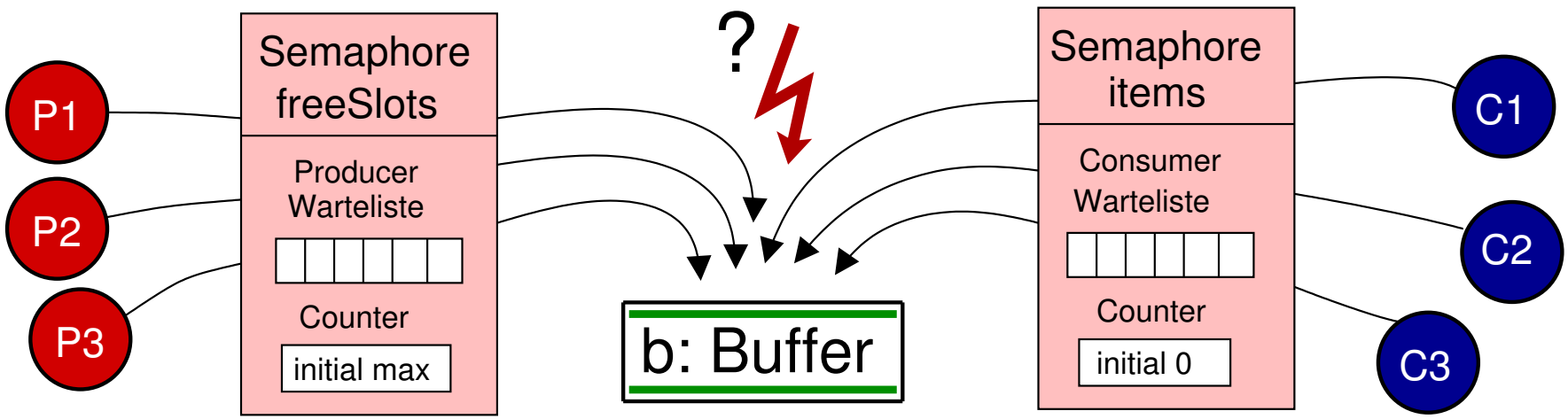
Datenstruktur mit Semphore (Buffer)

```
public class SemaphoreBuffer {
  private Vector<String> queue;
  private Semaphore items;
  private Semaphore freeSlots;
  final int MAX = 2;

  public SemaphoreBuffer() {
    queue = new Vector<String>();
    this.items = new Semaphore(0);
    this.freeSlots = new Semaphore(MAX);
  }
  public void transferElementToBuffer (String element) {
   try {
       freeSlots.acquire();
   } catch (InterruptedException e) {
       e.printStackTrace(); }
   queue.add(element);
   System.out.println(element + " wurde abgelegt");
   items.release();
  }
  public String takeElementFromBuffer () { ... }
}
```

Motivation für zusätzliches Monitoring

Problem

- Semaphoren verhindern hier nur Producer-Zugriff bei vollem bzw. Consumer-Zugriff bei leerem Puffer
- gleichzeitiger Zugriff von zwei Threads kann weiter zu Lese- oder Schreibkonflikten führen



Teil IV der Vorlesung PROG 2

Multithreading

Monitoring

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Andreas Rausch, TU Clausthal, Programmierkurs SS 2011*

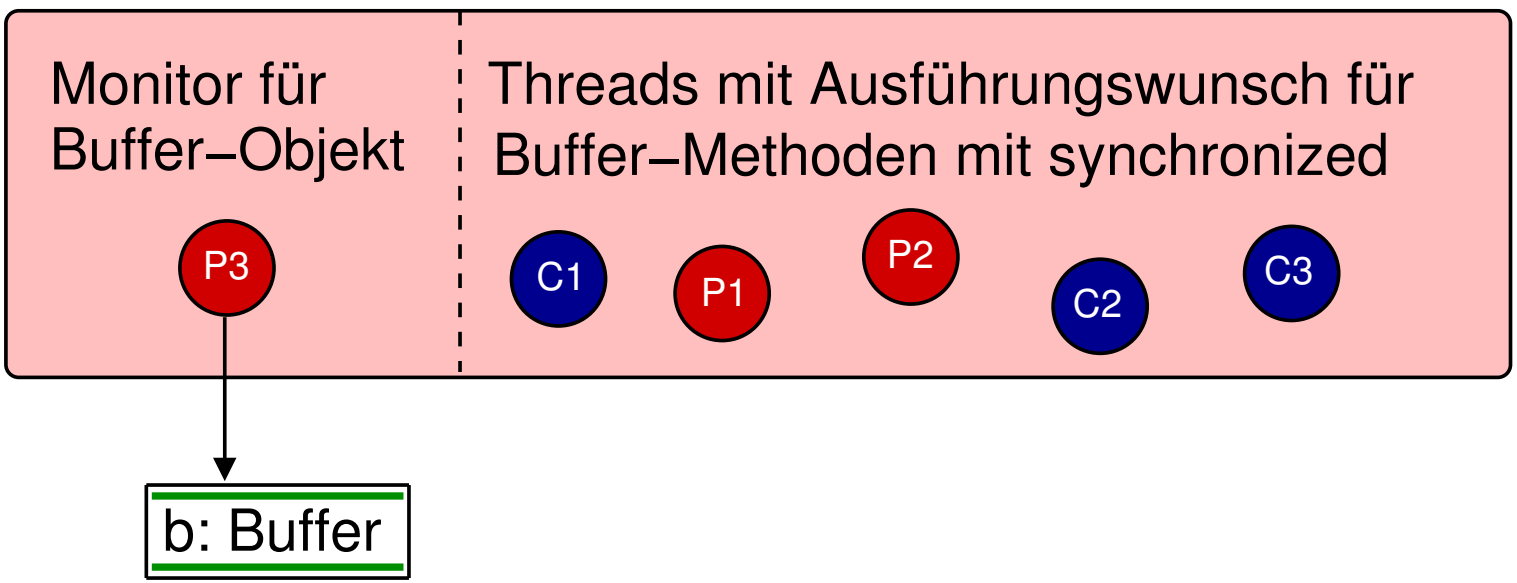
Monitore in Java

- auf höherem Abstraktionsniveau als Semaphoren-Konzept angesiedeltes Synchronisationsmittel
- von C.A.R. Hoare (1974) und B. Hansen (1975) entwickelt
- Idee: kritischer Programmteil darf nur von einem Thread zu einem Zeitpunkt betreten werden
- Setzen einer Sperre beim Betreten des kritischen Bereichs, wollen weitere Threads in den Bereich, so müssen sie warten
- Umsetzung in Java: Schlüsselwort *synchronized* vor eine Methode oder einen Programmblock setzen
- Hinweis: Sind mehrere Methoden mit *synchronized* markiert, so kann nur genau ein Thread gleichzeitig nur genau eine dieser Methoden zu einem Zeitpunkt ausführen

Monitor-Konzept in Java mit synchronized

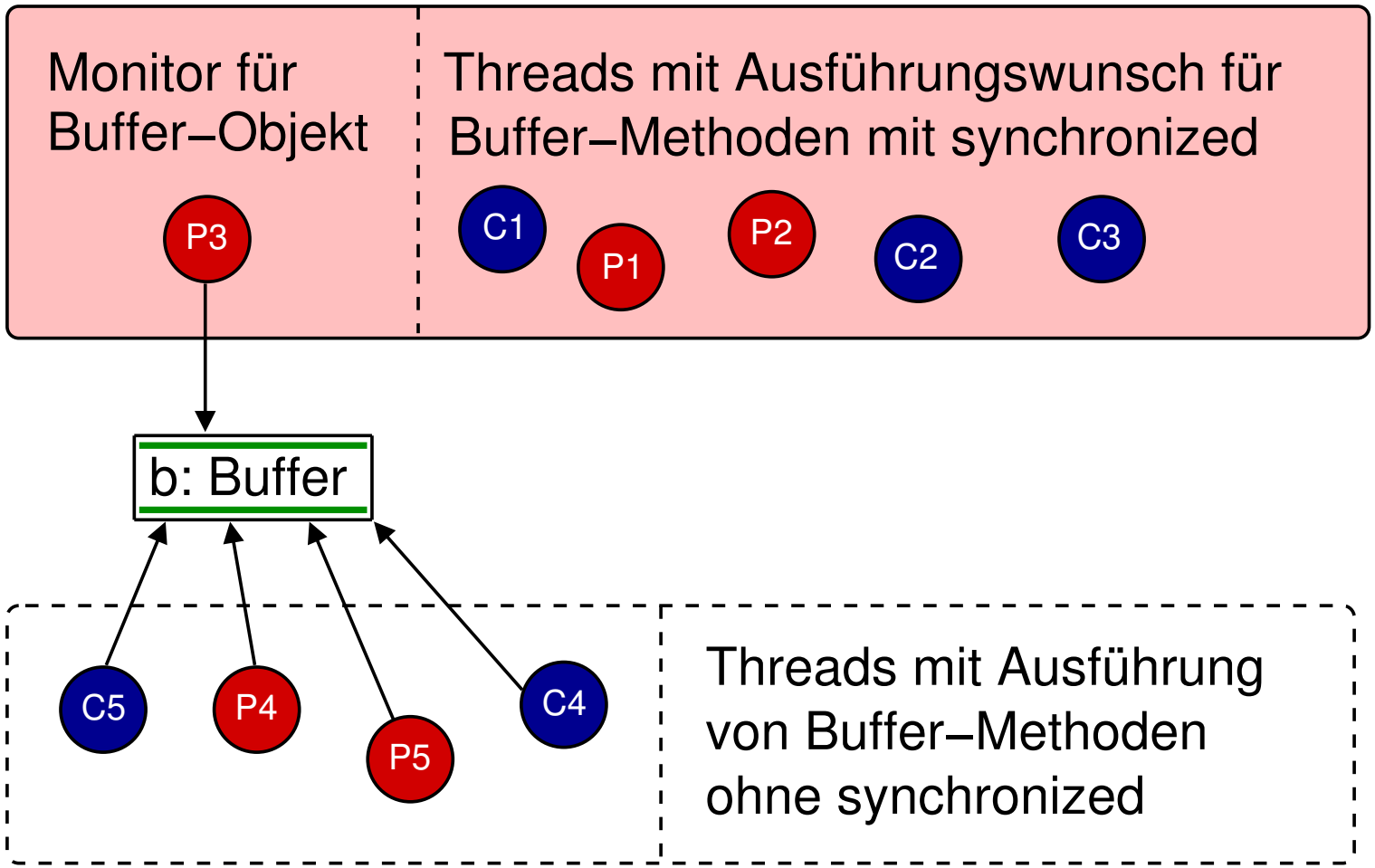
Definition

- jedes Objekt ist durch einen eigenen Monitor überwachbar
- ein Objekt wird durch seinen Monitor vollständig überwacht, wenn alle seine Methoden mit *synchronized* markiert sind und es nur über private Zustandsvariablen verfügt



Unvollständiger Monitor mit synchronized

➜ Methoden *ohne synchronized* sind von allen Threads jederzeit auf dem überwachten Objekt ausführbar



Monitor-Notation in Java mit synchronized

1. Deklaration von Methoden

```
public synchronized
          void transferElementToBuffer (String element) {
   ...
   queue.add(element); ... }
```

2. Deklaration von einzelnen Programmblöcken

```
public void transferElementToBuffer (String element) {
   ...
   synchronized (queue) {
        queue.add(element); } ... }
```

➜ überwachte Variablen (im Beispiel *queue*) müssen „echte" Java-Objekte sein, d.h. eine Variable vom Typ *long* würde vom *synchronized*-Block z.B. ignoriert

Äquivalenz der Ausdrucksmittel

Hinweis

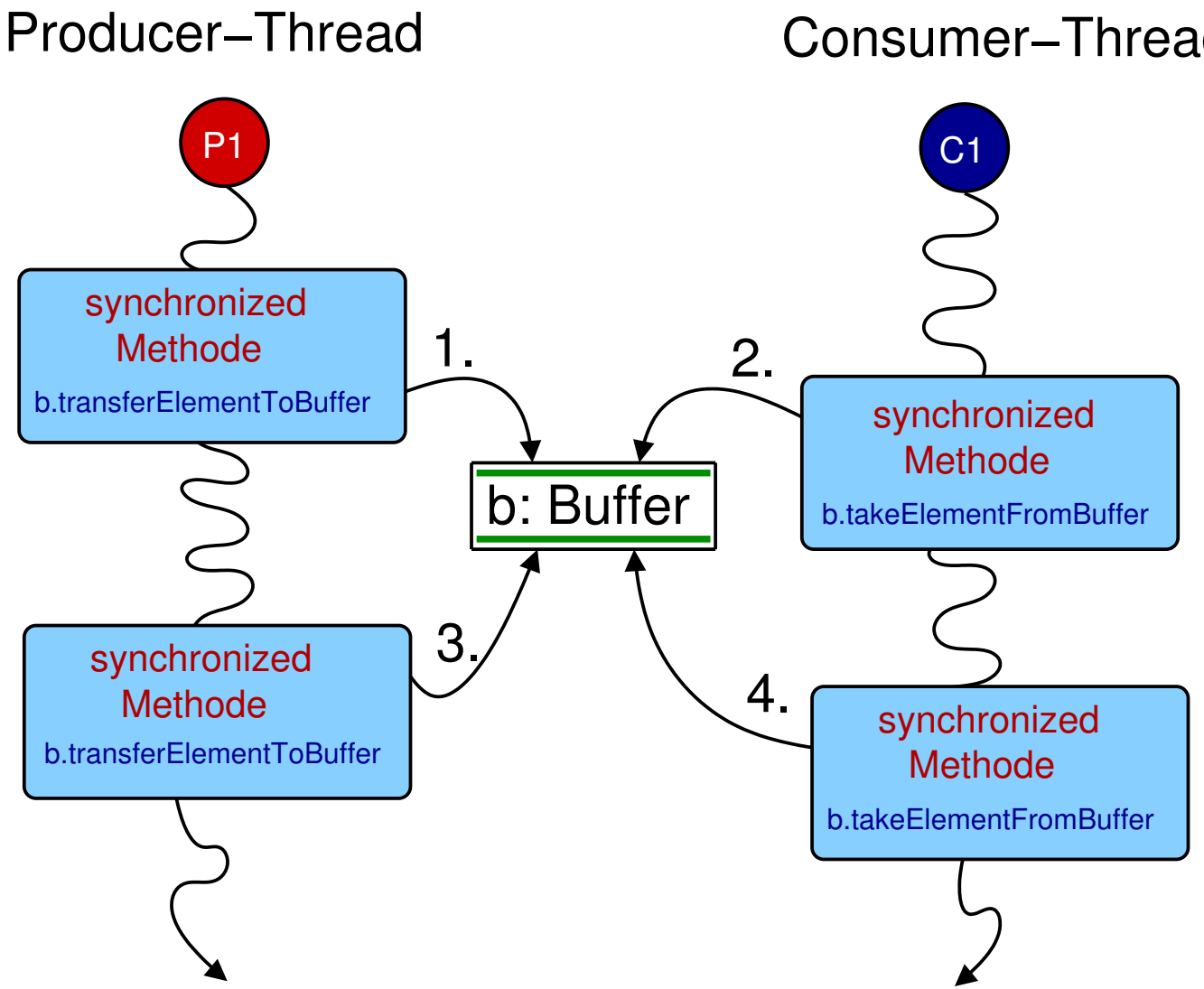
- Wirkung einer *synchronized*-Methode kann auch durch einen *synchronized*-Block formuliert werden
- Realisierung: Synchronisation über *this*-Objekt

```
public synchronized void foo()
 {
   ....
 }
```

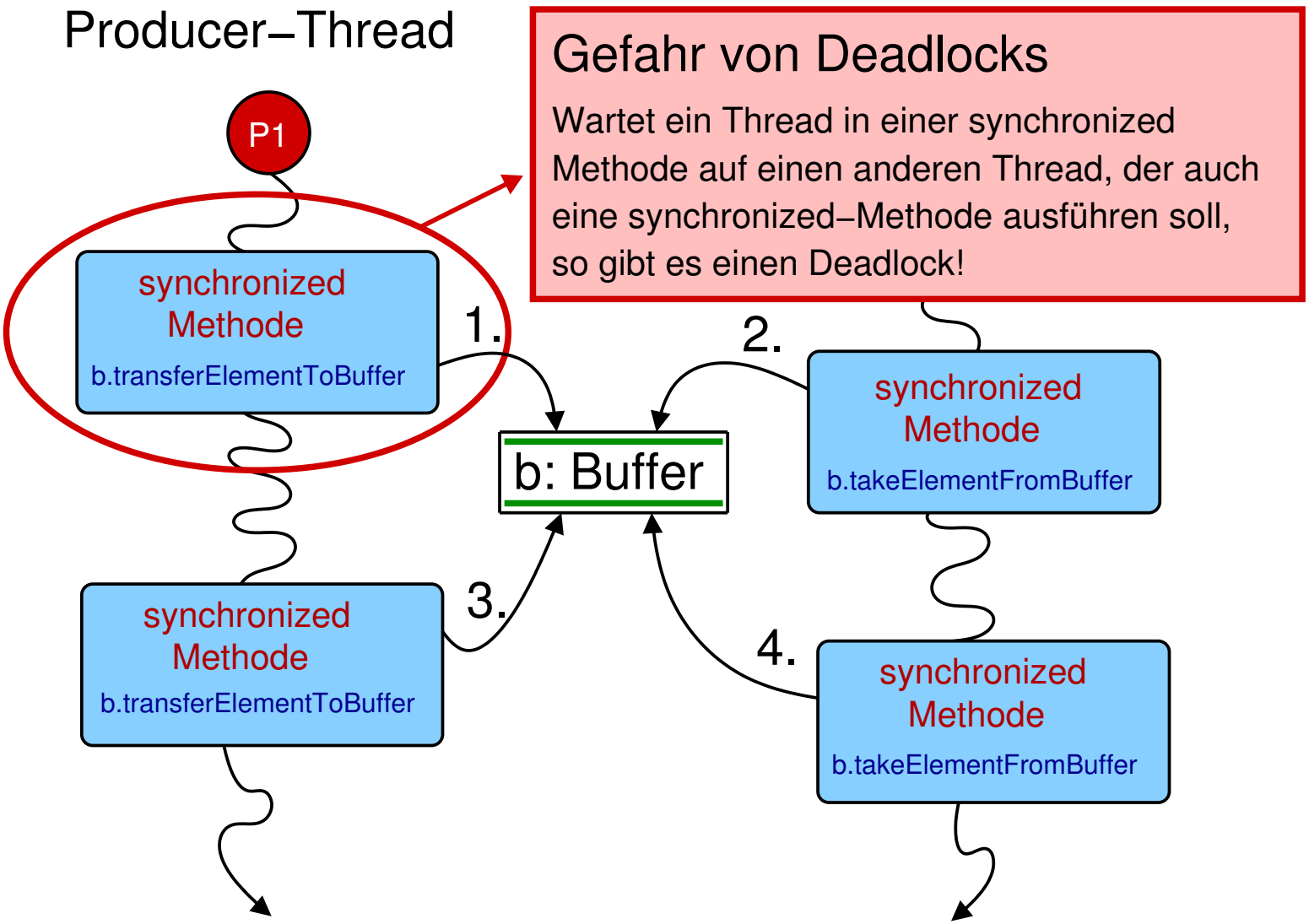
```
public void foo()
 {
    synchronized(this)
      {
          ...
      }
 }
```

➜ eine Methodendeklaration mit *synchronized* schützt im Consumer-Producer-Beispiel also das gesamte Buffer-Objekt

Abarbeitung von synchronized-Methoden



Deadlocks bei Verwendung von synchronized



Deadlock im Producer-Consumer-Beispiel

```
public class MonitoredBuffer {
  private Vector<String> queue;
  private Semaphore freeSlots;
  ...
  public MonitoredBuffer() {
    queue = new Vector<String>();
    this.items = new Semaphore(0);
    ... }

  public synchronized void
                transferElementToBuffer (String element) {
// Ist der Aufruf von acquire() erfolglos, so wartet der
// Thread darauf, dass jemand den Buffer reduziert. Während
// der Thread wartet, gibt er den Buffer leider nicht frei.
    try {
          freeSlots.acquire();
     } catch (InterruptedException e) { ... }
     queue.add(element); ...
  }
// Kann nicht ausgeführt werden, da der Buffer gesperrt ist.
  public synchronized String
                takeElementFromBuffer () { ... }
```

Bedingte Synchronisation mit synchronized

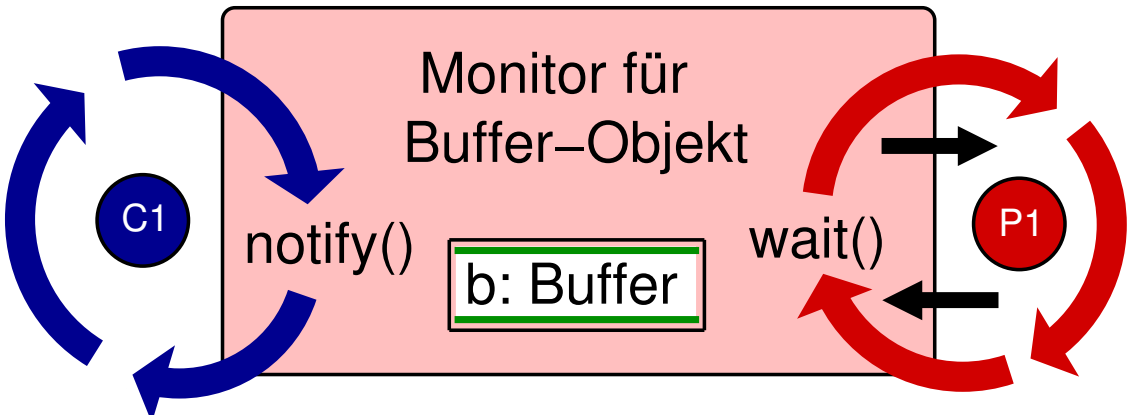
Schwächen einer einfachen Synchronisation

- Thread kann erst nach erfolgreicher Abarbeitung einer *synchronized*-Methode den Monitor verlassen
- Beispiel: *Producer* braucht keinen Puffer-Zugriff, wenn dieser leer ist, kann diesen aber erst freigegeben, wenn *transferElementToBuffer* komplett abgearbeitet ist

Lösung: Bedingte Synchronisation

- Thread wartet auf eine Bedingung, die er selbst nicht „herstellen" kann und verlässt mit *wait()* den Monitor
- konkurrierender Thread erlangt Zugriff und informiert nach Ausführung seiner Methode mit Hilfe von *notify()* oder *notifyAll()* einen oder mehrere wartende Threads

Bedingte Synchronisation im Java-Monitor



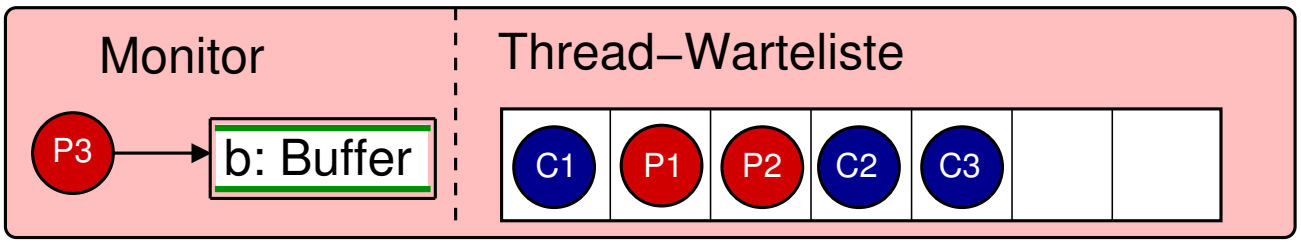
Vorgehen

- Es darf immer nur ein Thread in den Monitor
- Ausführung von *wait()*, um Monitor zu verlassen
- Ausführung von *notify()*, um wartende Threads aufzuwecken

Beispiel: Buffer mit bedingter Synchronisation

```
public class MonitoredBuffer {
  private Vector<String> queue;
  final int MAX = 2;
  public MonitoredBuffer() {
    queue = new Vector<String>(); }

  public synchronized void
                  transferElementToBuffer (String element) {
    while (queue.size() == MAX) {
  // Warten auf notify-Aufruf eines anderen Threads
  // in dieser Wartezeit dürfen andere Threads
  // die synchronized-Methode ausführen
      try {
          wait();
      } catch (InterruptedException e) { ... }
   }
   queue.add(element);
   System.out.println(element + " wurde abgelegt");
  // Wenn Element hinzugefügt, anderen Thread aufwecken
   notify();
  }
  public String takeElementFromBuffer () { ... }}
```

Semantische Fragen bei der Abarbeitung



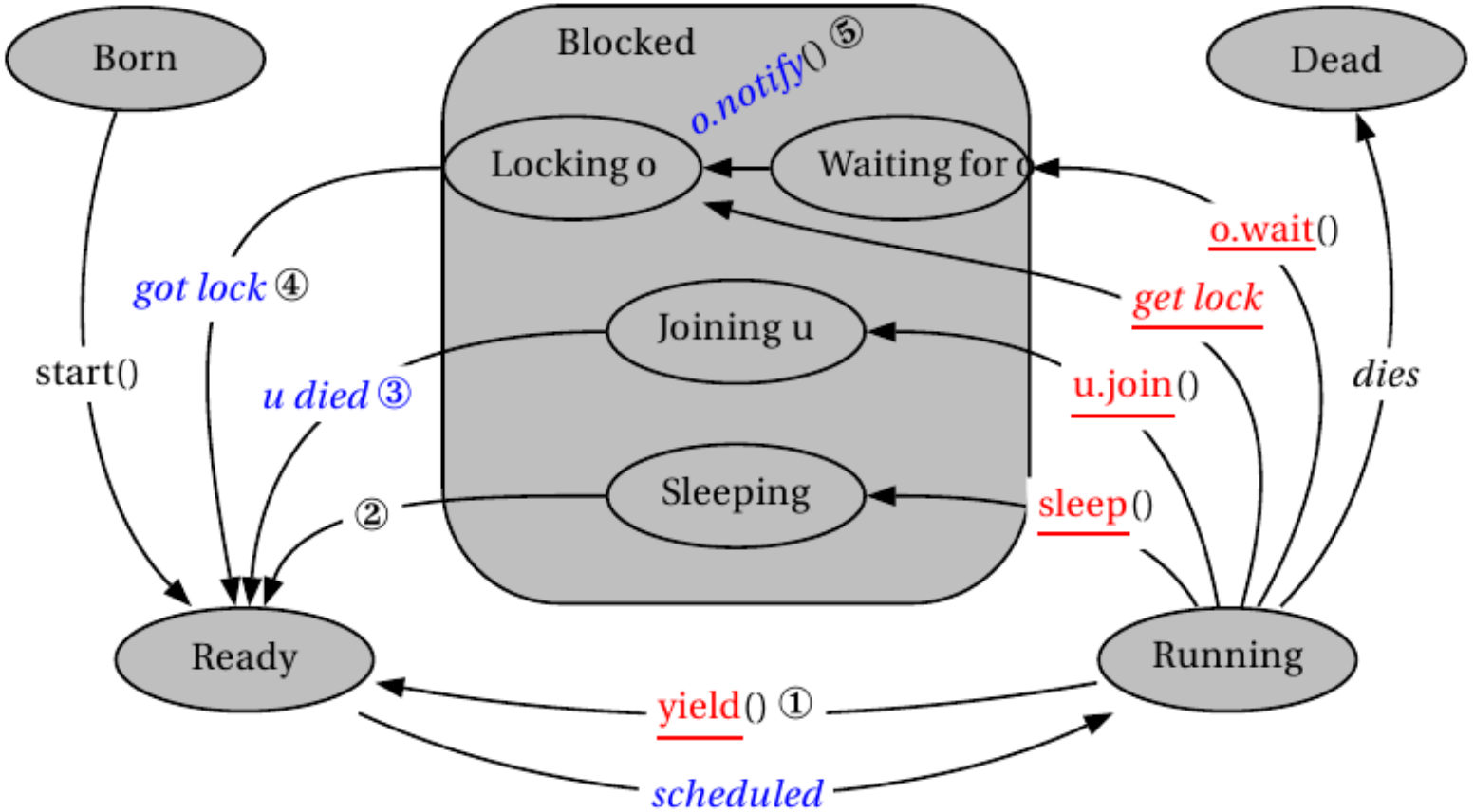
Warteliste

- nur *eine* Warte-(liste/menge) pro Objekt bzw. pro Klasse bei statischen Variablen
- egal, ob warten auf Eintritt in den Monitor (*synchronized*) oder warten auf Erfüllung einer Bedingung (*wait*)

Reihenfolge der Abarbeitung

- keine Annahmen möglich (wie z.B. Threads, die lange warten)
- *notify* ist nicht atomar, d.h. ein bereits aufgeweckter Thread kann z.B. den Lock gegen einen neuen Thread noch verlieren

Die Zustände eines Threads



rot = vom Thread selbst ausgeführt

blau = von einem anderen Thread ausgeführt

Alternativen zum synchronized-Statement

Explizites Locking

- seit Java 5 möglich (*java.util.concurrent.lock*)
- Hinweis: Lock wird beim Verlassen von Methoden nicht automatisch freigegeben, d.h. ist im *finally*-Block zu setzen

```
public class LockedBuffer {
  // Erzeugt neues Lock-Objekt
  private Lock accessLock = new ReentrantLock(); ...

public void transferElementToBuffer (String element) {
  accessLock.lock(); // Buffer sperren
  try {
        queue.add(element); ...
  }
  catch (InterruptedException e) { ... }
  finally {
        accessLock.unlock(); // Buffer freigeben
  }
}}
```

Bedingtes explizites Locking

Vorteile

- faire Abarbeitung (lang Wartende bevorzugen) konfigurierbar
- Freigabe in Abhängigkeit von Bedingungen auch möglich

```
public class LockedBuffer {
// faire Abarbeitung
  private Lock accessLock = new ReentrantLock(true);
// Bedingung fuer wartende Producer
  private Condition canTransfer = accessLock.newCondition();
// Bedingung fuer wartende Consumer
  private Condition canTake = accessLock.newCondition(); ...

public void transferElementToBuffer (String element) {
  accessLock.lock(); // Buffer sperren
  try { ... }
  catch ( ... ) { ... }
  finally { accessLock.unlock(); } // Buffer freigeben
}}
```

Beispiel: Bedingtes explizites Locking

```
public class LockedBuffer {
// faire Abarbeitung
  private Lock accessLock = new ReentrantLock(true);
// Bedingung fuer wartende Producer
  private Condition canTransfer = accessLock.newCondition();
// Bedingung fuer wartende Consumer
  private Condition canTake = accessLock.newCondition(); ...

public void transferElementToBuffer (String element) {
 accessLock.lock(); // Buffer sperren
 try {
  while(queue.size() == MAX) {
    System.out.println( "Buffer voll" );
    canTransfer.await(); // anderen Threads Vortritt lassen
  }
  queue.add(element); ...
  canTake.signal(); // Consumer informieren
 }
 catch (InterruptedException e) { ... }
 finally { accessLock.unlock();} // Buffer freigeben
}}
```

Schlüsselwort: volatile

Funktionsweise

- Optimierungen der JVM haben oft Seiteneffekte und können die Datenintegrität bei Multithreading beeinträchtigen
- Idee: Objekte kennzeichnen, die asynchron von außerhalb des aktuellen Threads verändert worden sein könnten

➜ bei Zugriff wird Objekt stets neu gelesen, anstatt nur auf einen Puffer zu referenzieren

Beispiel

```
// Zugriffe auf Objektvariablen werden durch die JVM
// häufig optimiert und auf lokalen Variablen vorberechnet
// Erst das lokal errechnete Ergebnis wird in einem Schritt
// an die Objektvariable zugewiesen
 public volatile Integer number;
 for (int i = 0; i < 1000000; i++)
  number++; // Ergebnis könnte sonst lokal errechnet werden
```

Ein Experiment als Wettlauf ...

➜ Was könnte es für ein Ergebnis geben?

```
public class Number {
    public long n;
    public Number (long n) { this.n = n; }
}
```

```
public class RaceCondition {
  static  Number s;
  public static void main( String[] args ) {
   s = new Number(0);
   Thread increment = new Thread (new IncrementRunnable(s));
   Thread decrement = new Thread (new DecrementRunnable(s));
   increment.start();
   decrement.start();
   try {
         increment.join();
         decrement.join();
   } catch (InterruptedException e) { ... }
   System.out.println("Der Wert ist " + s.n );
}}
```

Zwei konkurrierende Runnables im Wettlauf ...

```
public class DecrementRunnable implements Runnable {
    private Number s;
    public DecrementRunnable(Number s) {
        this.s = s; }
    public void run() {
        for (long i=0; i < 1000000; i++)
          s.n--; }}
```

```
public class IncrementRunnable implements Runnable {
    private Number s;
    public IncrementRunnable(Number s) {
        this.s = s; }
    public void run() {
        for (long i=0; i < 1000000; i++)
          s.n++; }}
```

➜ Was müssen wir machen, um das gewünschte Ergebnis zu erhalten?

Lösung Runnables im Wettlauf

- Es gibt natürlich mehrere Lösungsmöglichkeiten!
- Variante 1: Verwendung von *synchronized*
- Variante 2: Einsatz von *AtomicLong*, definiert in *java.util.concurrent.atomic.AtomicLong*
- ... und noch weitere ...

```
public class Number {
    public AtomicLong n;
    public Number (long n) { this.n = new AtomicLong(n); }
}
```

```
public class IncrementRunnable implements Runnable {
    private Number s;
    public IncrementRunnable(Number s) {
        this.s = s; }
    public void run() {
        for (long i=0; i < 1000000; i++)
          s.n.incrementAndGet(); }}
```

Teil IV der Vorlesung PROG 2

Multithreading

Thread-Pools und Thread-Gruppen

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Warum Thread-Pools einsetzen?

Grenzen „normaler" Threads

- Ausführung eines neuen Runnables erfordert den Aufbau eines neuen Threads
- mehrfache Ausführung eines Runnables durch einen Thread ist nicht so einfach möglich

Lösung: Einsatz von Thread-Pools

➜ Erzeugung mit *java.util.concurrent.Executors*

```
// Erzeugt einen Thread-Pool mit wachsender Grösse
static ExecutorService newCachedThreadPool()
// Erzeugt einen Thread-Pool mit maximal n Threads
static ExecutorService newFixedThreadPool(int n)
// mehrfache Ausfuehrung von Runnables
static ScheduledExecutorService
            newSingleThreadScheduledExecutor()
```

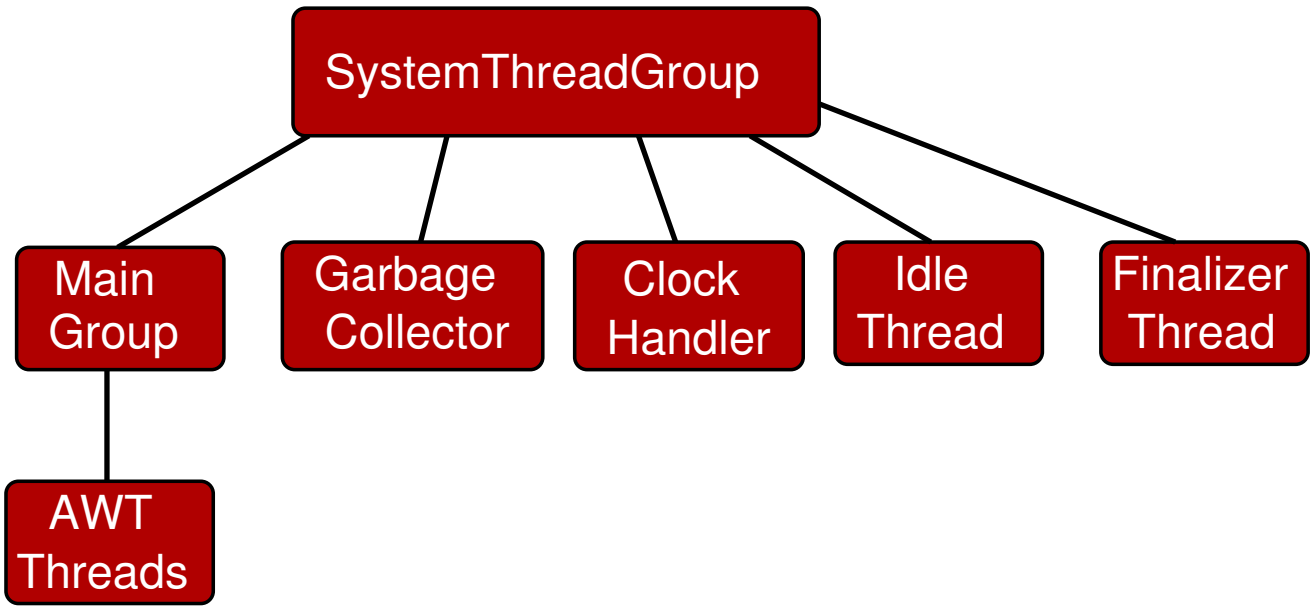
Erzeugung von Threads mit Executor

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class RunnableThread implements Runnable {
  private int sleepTime;
  private Random generator = new Random();

  public RunnableThread() { ... }

  public static void main(String[] args) {
    // Erzeuge neue RunnableThreads
    RunnableThread thread1 = new RunnableThread();
    RunnableThread thread2 = new RunnableThread();
    // Erzeuge Manager fuer die drei Threads
    ExecutorService
        threadExecutor = Executors.newCachedThreadPool();
    // Starte neue Threads
    threadExecutor.execute(thread1);  ... }
    // Beende den Starter-Thread
    threadExecutor.shutdown();
}
```

Verwendung von Thread-Gruppen

Intention

- oft einfacher, wenn nicht jeder Thread einzeln, sondern als Gruppe angesprochen werden kann
- z.B. auf allen Producer- und/oder Consumer-Threads ein *interrupt* auslösen oder System-Thread-Gruppe in Java



Was gibt es noch?

Fairness

- Prioritäten von Threads mit *getPriority()* oder *setPriority()* steuerbar
- Kooperatives Arbeiten: Freigabe des Prozessors durch die Methode *yield()* durch einen Thread auf freiwilliger Basis

Threads im Hintergrund

- Hintergrund-Threads werden auch als Daemonen bezeichnet
- Methode *setDaemon()* macht einen Thread zu einem Daemonen
- Daemonen sterben, wenn es im System nur noch Daemonen gibt und keine „normalen" aktiven Threads mehr existieren