HAW Hamburg
Fakultät TI
Technik und Informatik

Synchronisation

Programmiermethodik 2

Wiederholung

- Einführung
- Erzeugen
- Beenden
- Weitere Methoden
- Timer
- Probleme

Ausblick

Use Cases

- Ich möchte sicherstellen, dass mehrere Threads ...
  - parallel arbeiten.
  - und gleichzeitig gewisse Reihenfolgebedingungen einhalten.

Agenda

- Kritischer Abschnitt
- Monitor-Mechanismus
- Reihenfolge-Beschränkungen
- Deadlocks

Zum Nachlesen

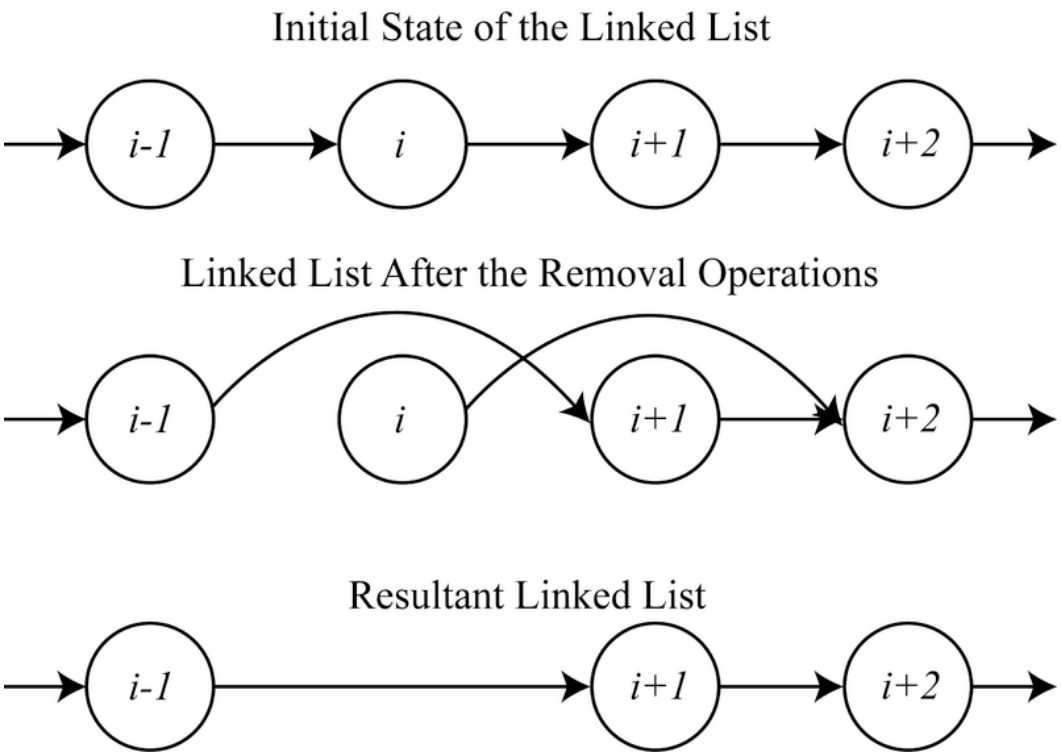
- Kathy Sierra, Bert Bates: Java von Kopf bis Fuß, Kapitel 15 ab Abschnitt "Multithreading", O'Reilly-Verlag
- zu `wait()/notify()`: Oracle Java Tutorial "Guarded Blocks": *http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html*, abgerufen am 11.04.2014
- Christian Ullenboom: Java ist auch eine Insel, 9., aktualisierte Auflage 2011, Galileo Computing, Kapitel 14.6: Synchronisation über Warten und Benachrichtigen, auch online verfügbar

Kritischer Abschnitt

Thread-Synchronisation

- Thread-Synchronisation
  - Herstellen einer zeitlichen Reihenfolge zwischen parallel ablaufenden Threads
- grundsätzliche Probleme
  - Zugriff auf gemeinsam benutzte Objekte
    - Wechselseitiger Ausschluss
  - Einhalten von notwendigen Reihenfolgebedingungen
    - Ablaufsteuerung durch Reihenfolgebeschränkungen

Erinnerung: parallele Operationen auf verlinkter Liste



Quelle: [2]

Kritischer Abschnitt

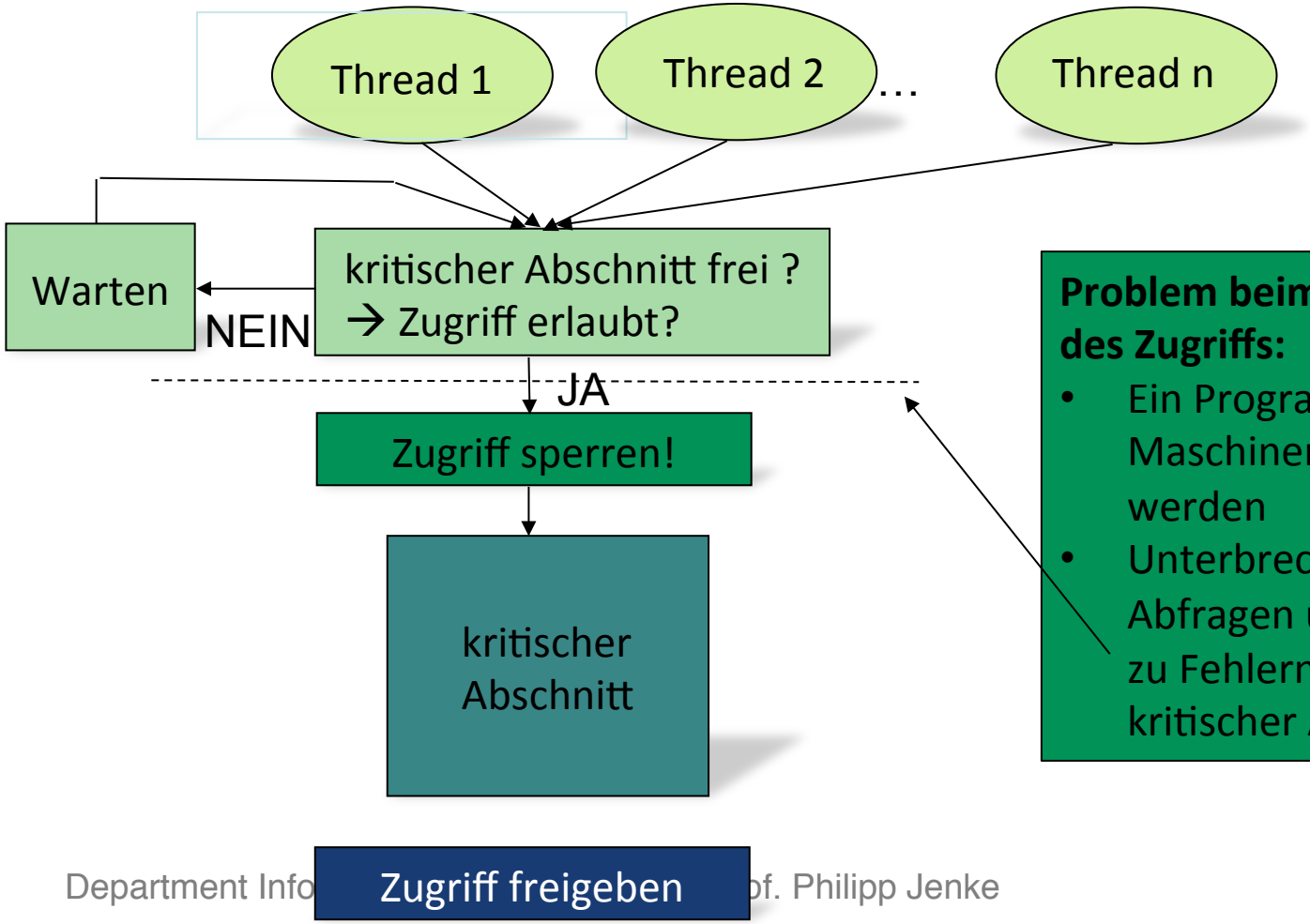
- Nebenläufigkeit führt zu der Notwendigkeit, Zugriff auf gemeinsam verwendete Objekte zu reglementieren
- Bereiche in denen Konflike durch parallelen Zugriff auftreten können nennt man "Kritische Abschnitte" (engl. *critical sections*)
  - problematisch nur, wenn dort der Zustand verändert werden kann
  - kein Problem bei unveränderlichen Objekten
- Konsequenz
  - Befehlsfolgen in kritischen Abschnitten dürfen nicht unterbrochen werden
  - nur ein Thread zur Zeit darf einen kritischen Abschnitt betreten
    - z.B. Veränderung einer Liste oder eines Zählers

Lösung: Wechselseitiger Auschluss

- engl. m*utual exclusion*
- Anforderung, dass keine zwei Prozesse oder Threads parallel eine kritische Sektion betreten
- Problemstellung wurde 1965 von Edsger W. Dijkstra beschrieben [3]

Thread-Synchronisation

- Allgemeine Synchronisationslösung für wechselseitigen Ausschluss



Problem beim Abfragen / Sperren des Zugriffs:

- Ein Programm kann nach **jedem** Maschinenbefehl unterbrochen werden
- Unterbrechung zwischen Abfragen und Sperren führt evtl. zu Fehlern → ist eigener kritischer Abschnitt!

Aktives Warten

- eigene Lösung für wechselseitigen Ausschluss
- engl. *busy waiting*
  - ein Thread prüft ständig, ob er einen kritischen Abschnitt betreten darf
  - *z.B. in einer "while"-Schleife*
  - Beispiel:

```
while (istBesetzt) {
}; // Warten (leerer Block)
istBesetzt = true; // Selbst Sperre setzen
... // Code für kritischen Abschnitt
istBesetzt = false; // Sperre freigeben
```

Beispiel

- zwei Threads, die gemeinsame Zählervariable verändern
- eigentlich: +1
- Vorgehen
  - +0.5
  - Pause
  - +0.5
  - Ausgabe
- Wunsch: Ausgabe immer ganzzahlig
- aber: klappt nicht
  - immer wieder x.5-Werte

```
public class ZaehlerOhneSynchronisation extends Thread {
  /**
   * Klassenvariable (gemeinsamerZugriff!)
   */
  private static double zaehler = 0.0;

  @Override
  public void run() {
    while (zaehler < 100.0) {
      zaehler = zaehler + 0.5;
      try {
        Thread.sleep((int) (10 * Math.random()));
      } catch (InterruptedException e) {
      }
      zaehler = zaehler + 0.5;
      System.err.println("Aktueller Zählerwert: " + zaehler);
    }
  }

  /**
   * Eintrittspunkt.
   */
  public static void main(String[] args) {
    new ZaehlerOhneSynchronisation().start();
    new ZaehlerOhneSynchronisation().start();
  }
}
```

Übung: Aktives Warten

- Verändern Sie den Beispielcode so, dass aktives Warten verwendet wird
- Die Ausgabe des Zählerstandes soll dann immer ganzzahlig sein.

```
public class ZaehlerOhneSynchronisation extends Thread {
  /**
   * Klassenvariable (gemeinsamerZugriff!)
   */
  private static double zaehler = 0.0;

  @Override
  public void run() {
    while (zaehler < 100.0) {
      zaehler = zaehler + 0.5;
      try {
        Thread.sleep((int) (10 * Math.random()));
      } catch (InterruptedException e) {
      }
      zaehler = zaehler + 0.5;
      System.err.println("Aktueller Zählerwert: " + zaehler);
    }
  }

  /**
   * Eintrittspunkt.
   */
  public static void main(String[] args) {
    new ZaehlerOhneSynchronisation().start();
    new ZaehlerOhneSynchronisation().start();
  }
}
```

Aktives Warten

- Probleme
  - mögliche Unterbrechung zwischen Abfrage und Sperren
  - Synchronisation klappt manchmal nicht!
  - eine saubere Programmierlösung mit aktivem Warten ist aufwändig
  - in der Praxis häufig: zweiter Thread kommt gar nicht zum Zug
  - wartender Thread (in `while`-Schleife) verbraucht CPU-Zeit!

- also: Aktives Warten ist keine Lösung!

Monitor-Mechanismus

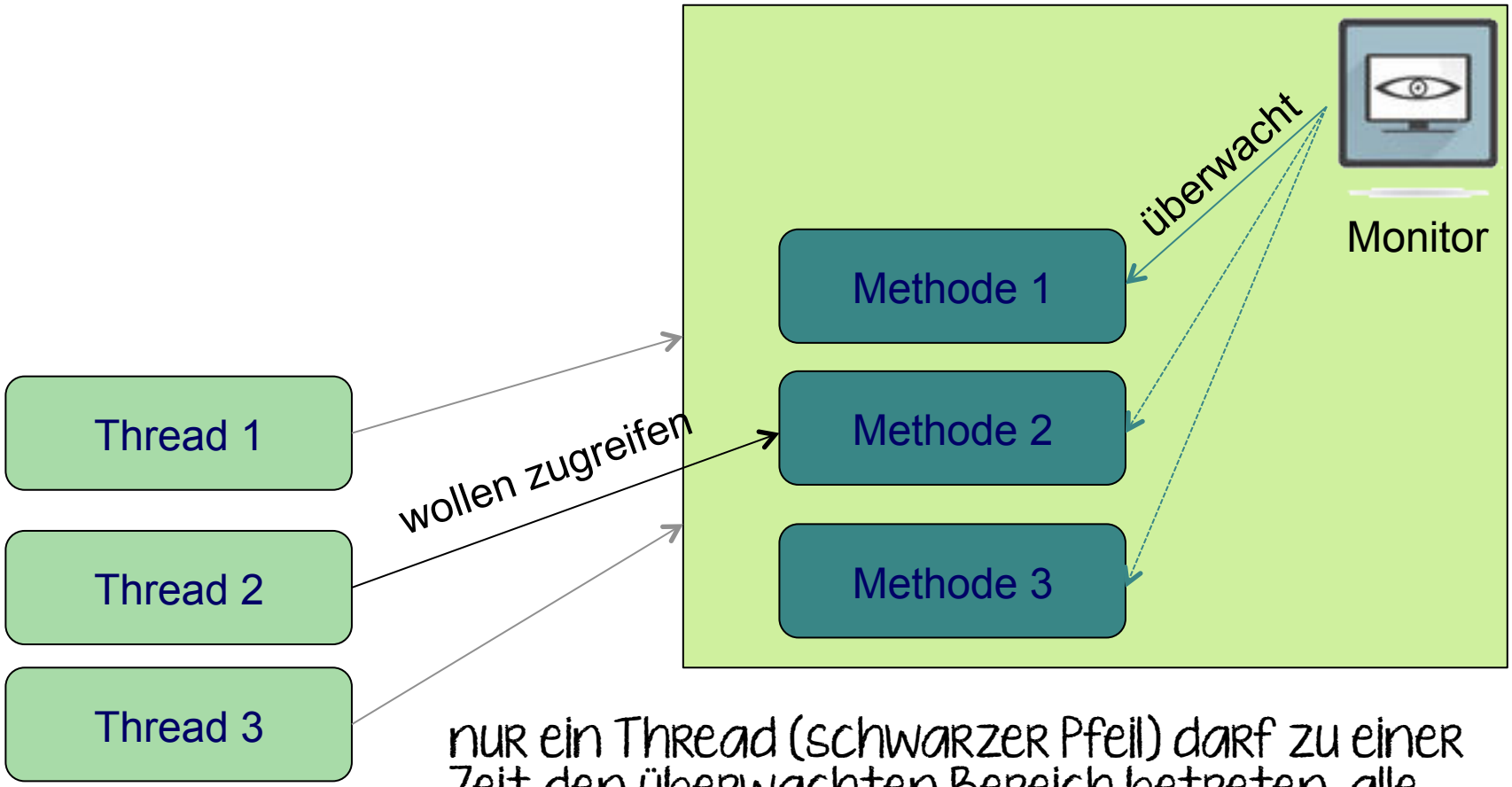
Synchronisation in Java

- Java bietet verschiedene Mechanismen zur Synchronisation
  - Monitor
  - Semaphore

Monitor-Mechanismus

- Ein Monitor überwacht den Aufruf bestimmter Methoden
- Ein Thread „betritt“ den überwachten Monitorbereich durch den Aufruf einer der Methoden und „verlässt“ ihn mit dem Ende dieser Methode
- Nur ein Thread zur Zeit kann sich innerhalb der überwachten Monitor-Methoden aufhalten (Sperre des kritischen Abschnitts), die übrigen müssen warten
- Die wartenden Threads werden von dem Monitor in einer Monitor-Warteschlange verwaltet (sind blockiert)
- Es gibt zusätzliche Synchronisationsfunktionen innerhalb des Monitors
- falls ein Monitor mehrere Methoden überwacht, darf nur eine zur Zeit betreten werden

Monitor-Mechanismus



nur ein Thread (schwarzer Pfeil) darf zu einer Zeit den überwachten Bereich betreten, alle anderen müssen warten (graue Pfeile)

Monitore in Java

- jedes Java-Objekt besitzt einen eigenen Monitor
  - ist sein eigener Monitor
- falls Monitorbereich eines Objektes gesperrt
  - kein anderer Thread eine synchronisierte Methode dieses Objektes ausführen
    - ab in die Monitor-Warteschlange!
  - jede unsynchronisierte Methode lässt sich dagegen ausführen!

Synchronized

- Monitor eines Objekts überwacht alle Methoden / Blöcke des Objekts, die mit `synchronized` bezeichnet sind
- Eintritt in den Monitorbereich über Aufruf irgendeiner `synchronized` – Methode des Objekts
- Eintritt in eine `synchronized`-Methode
  - Monitorbereich des Objekts für andere Threads gesperrt
  - nach dem Austritt wieder freigegeben
- Syntax:

```
<Sichtbarkeit> synchronized <Rückgabetyp> <Bezeichner> (<Paramater>) {
        ...
}
```

- Beispiel:

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

Beispiel


```
public class ZaehlerMitSynchronisation extends Thread {
  /**
   * Gemeinsamer Zählerstand aller Threads
   */
  private static double zaehler = 0.0;

  @Override
  public void run() {
    while (zaehler < 100.0) {
      erhoeheZaehler();
      System.err
          .println("Aktueller Zählerwert (" + getName() + "): " + zaehler);
    }
  }

  /**
   * Synchronisiertes Erhöhen des Werts
   */
  public synchronized void erhoeheZaehler() {
    zaehler = zaehler + 0.5;
    try {
      Thread.sleep((int) (10 * Math.random()));
    } catch (InterruptedException e) {
    }
    zaehler = zaehler + 0.5;
  }

  public static void main(String[] args) {
    new ZaehlerMitSynchronisation().start();
    new ZaehlerMitSynchronisation().start();
  }
}
```

Geschachtelte Aufrufe

- Frage: Kann man aus einer `synchronized`-Methode heraus eine andere, ebenfalls als `synchronized` gekennzeichnete Methode des gleichen Objekts aufrufen?
- Beispiel:

```
class Termin {
        synchronized void aendereTermin(...) {
                ...  schreibeTermin(...);    ...
        }
        synchronized void schreibeTermin(...) {
                ...
        }
}
```

Synchronisations-Varianten

- Synchronisation von Methoden einer Klasse
  - Schlüsselwort `synchronized` im Methodenkopf angeben
  - Wirkung ist identisch mit `synchronized(this) { ... }` am Anfang der Methode
- Synchronisation von Blöcken
  - Es können beliebige Code-Blöcke synchronisiert werden
  - Angabe eines Synchronisationsobjekts nötig
  - Syntax: `synchronized ( <Synchr.-Objekt> ) { ... }`
    - meist `getClass()` als Monitor verwendet
- Synchronisation über Klassen
  - Wie Objekte, besitzt auch jede Klasse genau einen Monitor
  - Eine Klassenmethode, die die Attribute `static synchronized` trägt, fordert somit den Monitor der Klasse an

Beispiel


```
public void run() {
  while (zaehler < 100.0) {
    // Alternative: synchronized (getClass()) {
    synchronized (this) {
      zaehler = zaehler + 0.5;
      try {
        Thread.sleep((int) (10 * Math.random()));
      } catch (InterruptedException e) {
      }
      zaehler = zaehler + 0.5;
    }
    System.err
        .println("Aktueller Zählerwert (" + getName() + "): " + zaehler);
  }
}
```

Übung: Synchronisierter Fußball

- Identifizieren Sie die kritische Methode in der Fußballsimulation.
- Verändern Sie den Quellcode so, dass der kritische Bereich synchronisiert wird
  - also nur von einem Thread gleichzeitig besucht wird

```
public class Spieler extends Thread {

  private final Keeper keeper;

  public Spieler(Keeper keeper, String name) {
    super(name);
    this.keeper = keeper;
  }

  @Override
  public void run() {
    for (int i = 0; i < 10; i++) {
      keeper.score();
      System.err.println(getName() + " hat ein Tor geschossen.");
    }
  }
}
```

```
public class Keeper {

  /**
   * Anzahl der kassierten Tore.
   */
  protected int anzahlTore = 0;

  /**
   * Ein Spieler schiesst auf das Tor (und trifft)
   */
  public void score() {
    anzahlTore++;
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  @Override
  public String toString() {
    return "Anzahl Tore: " + anzahlTore;
  }
```

Reihenfolge-beschränkungen

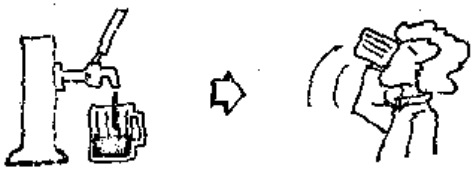
Synchronisation im Monitorbereich

- bisher gelöst: Wechselseitiger Ausschluss
- neue Problemstellung: Einhalten von Reihenfolgebedingungen
  - Ein Thread X befindet sich im kritischen Abschnitt
    - in einem `synchronized`-Block/ -Methode
    - im Monitorbereich
  - Er kann den kritischen Abschnitt erst verlassen, nachdem ein anderer Thread im selben kritischen Abschnitt ein Ereignis ausgelöst hat

Beispiel: Erzeuger-Verbraucher-Problem

- ein oder mehrere Erzeuger-Threads generieren einzelne Datenpakete und speichern diese in einem Puffer
- ein oder mehrere Verbraucher-Threads entnehmen einzelne Datenpakete aus dem Puffer und verbrauchen diese
- Zu jedem Zeitpunkt darf nur ein Thread (Erzeuger oder Verbraucher) auf den Puffer zugreifen
  - Kritischer Abschnitt
- Erzeuger-Threads müssen auf einen Verbraucher warten, wenn der Puffer voll ist
- Verbraucher-Threads müssen auf einen Erzeuger warten, wenn der Puffer leer ist

Beispiel

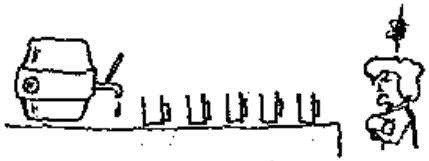


Verbraucher = Gast
Erzeuger = Zapfhahn

Idee: beschränkte Ressourcen:
5 Gläser

Produktion zu schnell:
Stress für den Gast!

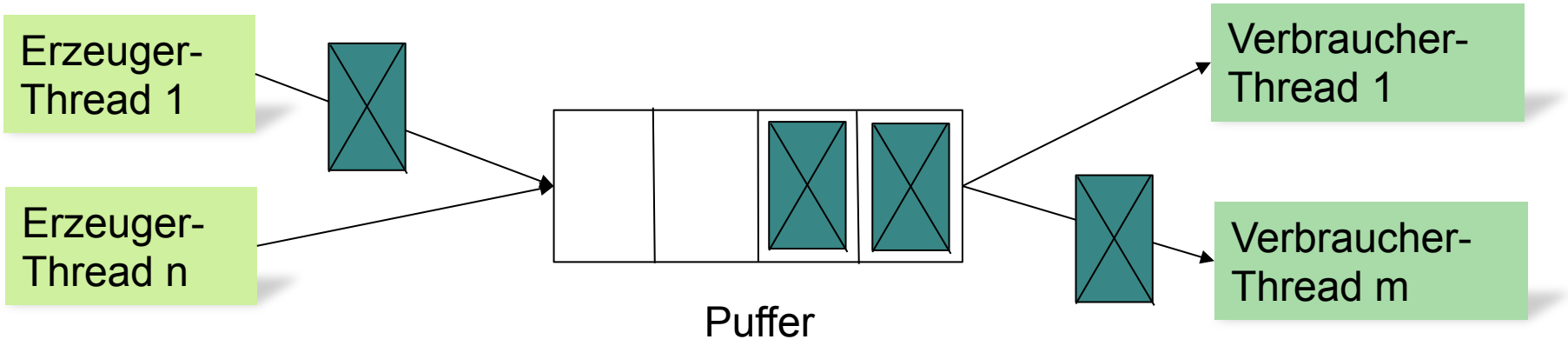
- dann
  - Zapfhahn füllt nur, wenn mindestens ein Glas leer
  - Gast trinkt nur, wenn mindestens ein Glas voll



Produktion zu langsam:
Gast verdurstet!

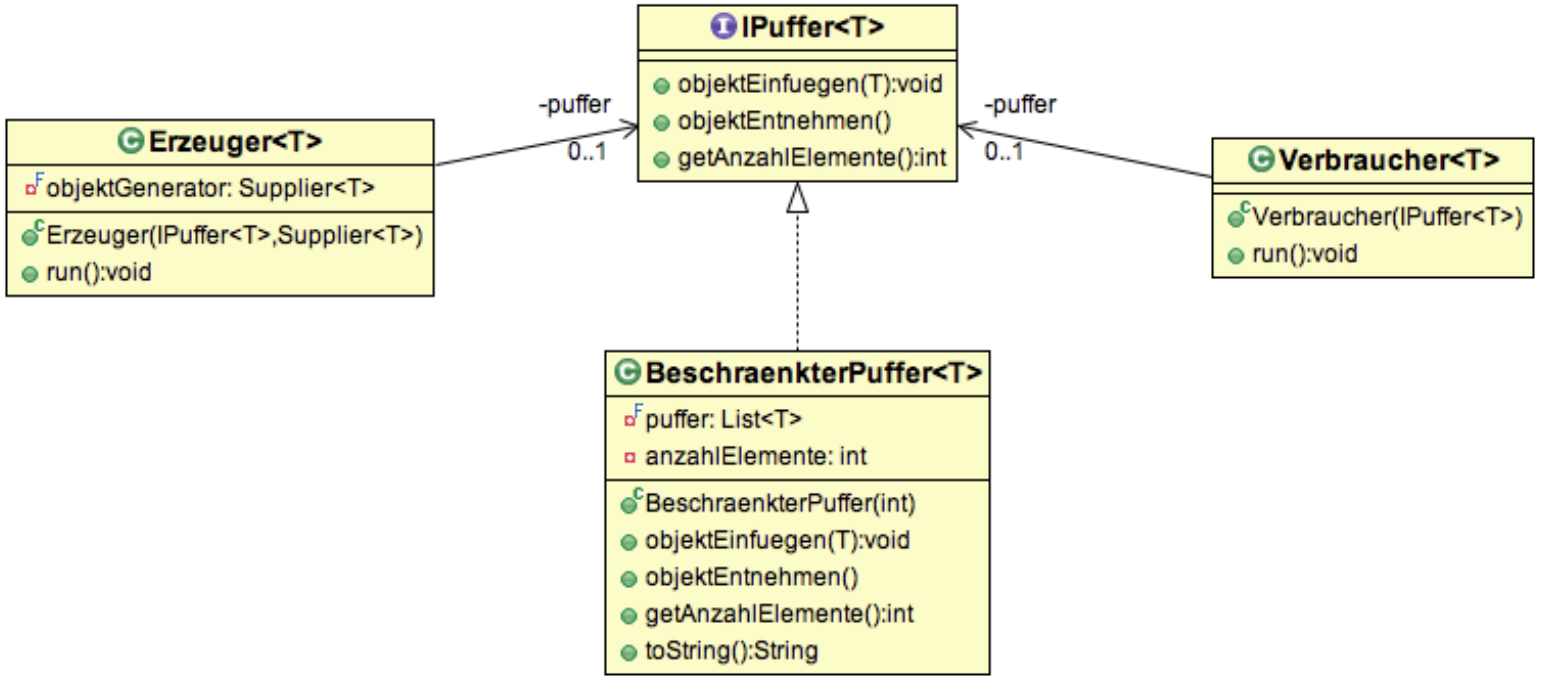
oder:
Gast entfernt halbvolles Glas!

Erzeuger-Verbraucher-Problem



Umsetzung

- Wir versuchen, das Problem programmatisch zu lösen



Puffer

- Puffer kann Elemente aufnehmen
- und wieder abgeben
- gemeinsames Interface: `IPuffer`

```
public interface IPuffer<T> {

  /**
   * Einfügen eines Elements in den Puffer.
   *
   * @param objekt
   *          Objekt, das in den Puffer eingefügt werden soll.
   */
  public void objektEinfuegen(T objekt);

  /**
   * ZUrückgeben eines Elements aus dem Puffer.
   *
   * @return Objekt, das der Puffer zurückliefert.
   */
  public T objektEntnehmen();

  /**
   * Liefert die aktuelle Anzahl der Elemente im Puffer.
   *
   * @return Anzahl der Elemente im Puffer.
   */
  public int getAnzahlElemente();

}
```

Erzeuger

- Erzeuger ist ein `Thread`
- fügt ein Objekt (generischer Typ `T`) in einen Puffer ein
- Objekterzeugung über Lambda-Ausdruck (SAM `Supplier<T>`)

```
public class Erzeuger<T> extends Thread {
  /**
   * Referenz auf den gemeinsamen Puffer.
   */
  private IPuffer<T> puffer;

  /**
   * Erzeugt das Objekt.
   */
  private final Supplier<T> objektGenerator;

  /**
   * Konstruktor mit Übergabe des Puffers
   * */
  public Erzeuger(IPuffer<T> puffer, Supplier<T> objektGenerator) {
    this.puffer = puffer;
    this.objektGenerator = objektGenerator;
  }

  @Override
  public void run() {
    T objekt = objektGenerator.get();
    puffer.objektEinfuegen(objekt);
    System.err.println("Erzeuger hat Objekt " + objekt
        + " erzeugt und in den Puffer gelegt. ");
  }
}
```

Übung: Verbraucher

- Verbraucher
  - ist ein `Thread`
  - nimmt ein Element aus einem Puffer (`T objektEntnehmen()`)
  - gibt Objekt auf Konsole aus (durch `toString()`)
- Schreiben Sie die Klasse `Verbraucher`

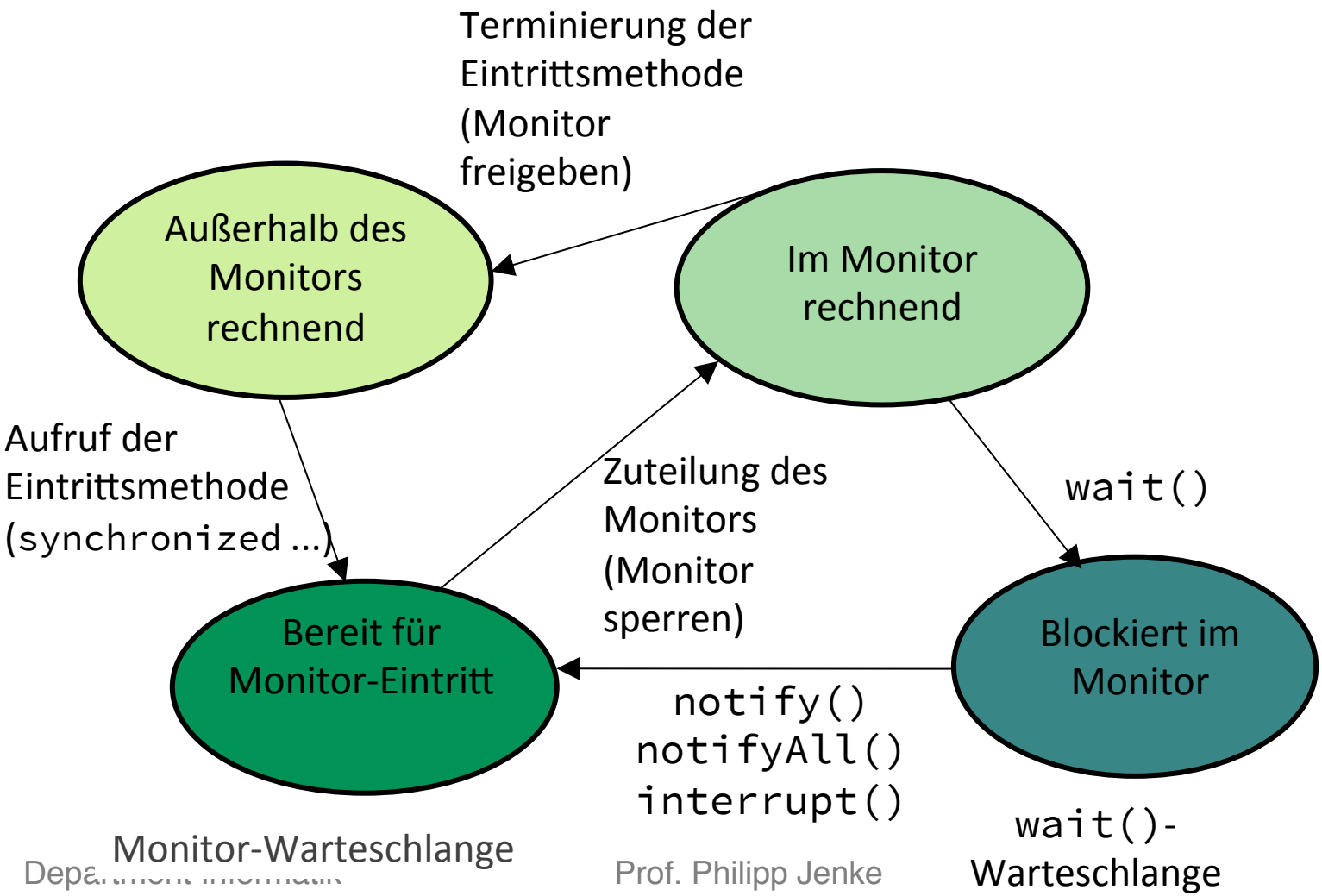
Mechanismen zur Reihenfolgensteuerung

- Ziel
  - nur ein Element einfügen, wenn Puffer nicht voll
  - nur ein Element entfernen, wenn mindestens eins im Puffer ist
- Umsetzung mit Threads (Wunsch)
  - Einfügen und Entfernen synchronisiert
  - dann
    - Methode betreten
    - Prüfen, ob Bedingung erfüllt
    - falls ja: machen, Methode verlassen
    - falls nein: Warten (Parken), Monitor freigeben, später zurückkommen und wieder Bedingung prüfen

wait() und notify()

- Thread parken: `wait()`
  - Methode nicht weiter bearbeiten
  - Monitor freigeben für anderer Threads
  - Thread in einer Warteschlange parken
- Threads aus der Warteschlange zurückholen: `notifyAll()`
  - ein Thread aus der Warteschlange holen
  - Monitor zugriff erteilen
  - Thread darf weitermachen, wo er zuvor geparkt wurde

Thread-Zustandsdiagramm



wait() und notify()

- Monitor freigeben und in zusätzlicher `wait()`-Warteschlange warten

  `wait()`

  - kann eine `InterruptedException` werfen
- einen (zufälligen) Thread in der `wait()`-Warteschlange wecken

  `notify()`
- alle Threads in `wait()`-Warteschlange wecken

  `notifyAll()`
- Der Aufruf dieser Methoden muss aus dem Monitor heraus erfolgen
  - innerhalb einer `synchronized`-Methode

Beschränkter Puffer

- Umsetzung in beschränktem Puffer
- Sicherstellung der Reihenfolgeanforderungen

```
public class BeschraenkterPuffer<T> implements IPuffer<T> {

  /**
   * Liste als Speicher
   */
  private final List<T> puffer;

  /**
   * Aktuelle Anzahl von ELementen im Puffer.
   */
  private int anzahlElemente = 0;

  /**
   * Konstruktor
   */
  public BeschraenkterPuffer(int pufferGroesse) {
    puffer = new ArrayList<T>();
    anzahlElemente = 0;
    for (int i = 0; i < pufferGroesse; i++) {
      puffer.add(null);
    }

  }
```

Puffer-Methode: Objekt einfügen


```
@Override
public synchronized void objektEinfuegen(T objekt) {
  while (anzahlElemente == puffer.size()) {
    try {
      this.wait();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      return;
    }
  }
  puffer.set(anzahlElemente, objekt);
  anzahlElemente++;
  try {
    Thread.sleep(500);
  } catch (InterruptedException e) {
  }
  System.err.println("---\nNeuer Pufferinhalt: " + this);
  this.notifyAll();
}
```

solange "Puffer voll"

einfügenden Thread parken

erst wenn Platz im Puffer

Element einfügen

geparkte Threads aufwecken

Übung: Objekt aus Puffer entnehmen

- Implementieren Sie die Methode `T objektEntnehmen()` für die Klasse `BeschraenkterPuffer:`
  - Warten bis mindestens ein Element vorhanden
  - Element entnehmen
  - geparkte Threads informieren
  - Element zurückgeben

Weiteres Beispiel

- Zwei Zähler zählen abwechseln gemeinsamen Wert hoch

```
public class ZaehlerAbwechselnd extends Thread {
  /**
   * Klassenvariable (gemeinsamer Zähler)
   */
  private static double zaehler = 0.0;

  @Override
  public void run() {
    // Gemeinsamer Monitor: Klassen-Monitor
    synchronized (getClass()) {
      while (zaehler < 20.0) {
        zaehler = zaehler + 0.5;
        try {
          Thread.sleep((int) (10 * Math.random()));
        } catch (InterruptedException e) {
          return;
        }
        zaehler = zaehler + 0.5;
        System.err.format("%s: %.1f\n",
            Thread.currentThread().getName() + ": ", zaehler);
        // Anderen Thread aufwecken
        getClass().notify();
        // Eigenen Thread parken
        try {
          getClass().wait();
        } catch (InterruptedException e) {
          return;
        }
      }
      // Ganz am Ende noch einmal den Partner aufwecken
      getClass().notify();
    }
  }
```

Ausgabe:

Thread-0: : 1,0
Thread-1: : 2,0
Thread-0: : 3,0
Thread-1: : 4,0
Thread-0: : 5,0
Thread-1: : 6,0
Thread-0: : 7,0

...

Thread-1: : 18,0
Thread-0: : 19,0
Thread-1: : 20,0

Deadlocks

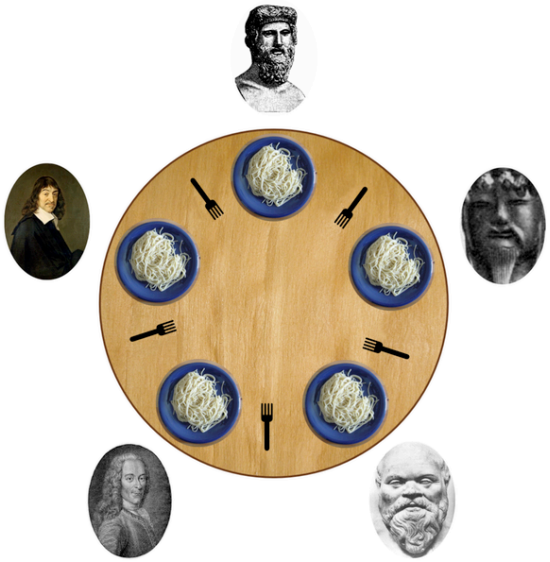
Deadlocks

- mehrere Threads hängen voneinander ab
- es kann Situation entstehen in der kein Thread weitermachen kann
  - weil er auf einen anderen Thread wartet
- Deadlock!

Quelle: [4]

Beispiel: Philosophen-Problem

- fünf Philosophen
  - entweder denken oder essen
- fünf Gabeln je zwischen zwei Philosophen
- zum Essen zwei Gabeln benötigt



Quelle: [5]

Beispiel: Philosophen-Problem

- Philosoph = `Thread`
- entweder denken (= Warten)
- oder Essen
  - linke Gabel aufnehmen
  - rechte Gabel aufnehmen
  - Warten
  - linke Gabel zurücklegen
  - rechte Gabel zurücklegen
- möglicher Deadlock
  - jeder Philosoph hat eine Gabel aufgenommen
  - wartet, dass er zweite Gabel aufnehmen kann

- Auszug aus `Gabel`:

```
/**
 * Die Gabel wird von einem Philisophen aufgenommen.
 */
public synchronized void nimmAuf(Philosoph philosoph) {
  while (hatGabel != null) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }
  System.err.println("Philosoph " + Thread.currentThread().getName()
      + " nimmt " + name + " auf");
  hatGabel = philosoph;
}

/**
 * Diue Gabel wird zurückgelegt.
 */
public synchronized void legeZurueck() {
  System.err.println("Philosoph " + Thread.currentThread().getName()
      + " legt Gabel " + name + " zurück.");
  hatGabel = null;
  notifyAll();
}
```

Fork-Join

Teile und Herrsche

- Viele Probleme sind sehr aufwändig, wenn man sie von "vorne nach hinten" löst
- dann häufig sinnvoll: "teile und herrsche"
- teile Problem in Teilprobleme und löse diese (siehe Rekursion!)
- Struktur:

```
löse Problem:
        ist Problem klein:
                löse Problem direkt
        andernfalls:
                zerlege das Problem in Teilprobleme
                löse die Teilprobleme
                setze Problemlösung aus den Teillösungen zusammen
```

Fork-Join-Framework

- seit Java 7 enthalten
- Umsetzung
  - `ForkJoinPool` (zentrale Verwaltung)
  - `Task` (Problem, kann Teiltasks generieren)
    - mit Rückgabewert: `RecursiveAction`
    - ohne Rückgabewert: `RecursiveTask<T>`

Beispiel: Fibonacci-Zahlen

- Erinnerung: rekursive Berechnung

```
public static long berechne(int index) {
  if (index == 0) {
    return 0;
  } else if (index == 1) {
    return 1;
  } else {
    return berechne(index - 1) + berechne(index - 2);
  }
 }
}
```

- mit Fork-Join
  - falls `index` klein (z.B. <= 10): Verwende rekursive Berechnung
  - ansonsten
    - starte parallele Berechnung für `index-1` (`fork()`)
    - berechne für `index-2` (`compute()`)
    - warte auf Ergebnis für `index-1` (`join()`)
    - summiere Teilergebnisse

Beispiel: Fibonacci-Zahlen

- starte parallele Berechnung für index-1 (`fork()`)
- berechne für `index-2` (`compute()`)
- warte auf Ergebnis für `index-1` (`join()`)
- summiere Teilergebnisse

```
public class FibonacciTask extends RecursiveTask<Long> {
  /**
   * Index der zu berechnen Zahl.
   */
  private int index;

  public FibonacciTask(int index) {
    this.index = index;
  }

  protected Long compute() {
    if (index < 10) {
      return FibonacciRekursiv.berechne(index);
    } else {
      // Zwei Teil-Tasks erstellen
      FibonacciTask left = new FibonacciTask(index - 1);
      FibonacciTask right = new FibonacciTask(index - 2);
      // Problem index - 1 parallel berechnen lassen
      left.fork();
      // Problem index - 2 im gleichen Thread berechnen
      long rightAns = right.compute();
      // Berechnung der Lösung aus dem zweiten Thread abwarten
      long leftAns = left.join();
      // Summe berechnen
      return leftAns + rightAns;
    }
  }
```

- Starten den Berechnung als Pool

```
ForkJoinPool pool = new ForkJoinPool();
return pool.invoke(new FibonacciTask(index));
```

Alternative zu Monitor: Semaphoren/Lock

- synchronized

```
public class CounterSynchronized {
    private int count = 0;
    public int inc() {
        synchronized (this) {
            return ++count;
        }
    }
}
```

- Vorteil: sehr einfache Verwendung

- Lock

```
public class CounterLock {

    private Lock lock = new ReentrantLock();
    private int count = 0;

    public int inc() {
        lock.lock();
        int newCount = ++count;
        lock.unlock();
        return newCount;
    }

}
```

- Vorteil: mehr Freiheiten bei Verwendung

Zusammenfassung

- Kritischer Abschnitt
- Monitor-Mechanismus
- Reihenfolge-Beschränkungen
- Deadlocks
- Fork-Join

Quellen

- Die Folien basieren zum großen Teil auf den Folien von Prof. Dr. Martin Hübner, Hochschule für Angewandte Wissenschaften Hamburg und folgendem Buch: Elisabeth Freeman, Eric Freeman, Kathy Sierra, Bert Bates: *Head First Design Patterns*, O'Reilly Media, 2004
- [1] Valerijs Kostreckis, *123rf.com/*, Bild-Nummer : 14007058, abgerufen: 24.10.2013
- [2] Wikipedia: Mutual Exclusion: *http://en.wikipedia.org/wiki/Mutual_exclusion*, abgerufen am 31.10.2013
- [3] Dijkstra, E. W.: *"Solution of a problem in concurrent programming control"*. Communications of the ACM 8 (9): 569
- [4] Christian Ullenboom: Java ist auch eine Insel, Galileo Computing, ISBN 978-3-8362-1506-0
- [5] Wikipedia: Philosophenproblem, *http://de.wikipedia.org/wiki/Philosophenproblem*, abgerufen am 22.3.2014
- [6] Michael Vigneau, http://www.ccs.neu.edu/home/kenb/synchronize.html, abgerufen am 19.06.2015 (Texte überarbeitet)