Parallele Programmierung
Gefahren der Nebenläufigkeit

Vorlesung 4

Prof. Dr. Luc Bläser

Letzte Vorlesung - Quiz

- Erklären Sie die Unterschiede

Java Monitor ⇔ Lock & Conditions

Semaphore ⇔ CountDownLatch

CountDownLatch ⇔ CyclicBarrier

Gefahren der Nebenläufigkeit

- Nebenläufige Programmierung birgt das Risiko neuer Arten von Programmierfehler
  - Fehler, die es bei Single-Threading so nicht gibt
  - Können sporadisch oder selten auftreten
  - Sehr schwierig durch Tests zu finden

Welchen Fall haben wir bereits gesehen?

Inhalt Heute

- Fehler der Nebenläufigkeit
  - Race Conditions
  - Deadlocks
  - Starvation
  - Erkennung und Vermeidung
- Design Aspekte
  - Immutability und Confinement
  - Collection und Thread-Sicherheit

Lernziele

- Race Conditions, Deadlocks und Starvation verstehen, erkennen und verhindern
- Korrektheitsbedingungen für Nebenläufigkeit kennen
- Möglichkeiten zum sicheren Verzicht auf Synchronisation erfassen
- Thread-Sicherheit bei Collections überblicken

Fehler der Nebenläufigkeit

- Race Conditions
  - Ungenügend synchronisierte Zugriffe auf gemeinsame Ressourcen
- Deadlocks
  - Gegenseitiges Aussperren von Threads
- Starvation
  - Kontinuierliche Fortschrittsbehinderung von Threads wegen Fairness-Problemen

Race Condition

- Mehrere Threads greifen auf gemeinsame Ressourcen ohne genügende Synchronisation zu
  - Mögliche falsche Resultate oder falsches Verhalten
  - Je nach Thread-Verzahnung und zeitlicher Ausführung

- Ursache ist oft ein Data Race, aber nicht immer!

Data Race

- Unsynchronisierter Zugriff auf gleichen Speicher
  - Selbe Variable oder Array-Element
  - Mindestens ein schreibender Zugriff von einem Thread (Read-Write, Write-Read, Write-Write)

```
int balance;
```

Thread 1

```
balance = 100;
```

Thread 2

```
return balance;
```

Formaler Fehler

Data Races: Weitere Beispiele

Thread 1

```
balance += 100;
```

Thread 2

```
balance += 50;
```

Thread 1

```
if (balance > amount) {
  balance -= amount;
}
```

Thread 2

```
if (balance > amount) {
  balance -= amount;
}
```

Race Condition ohne Data Race

```
class BankAccount {
  int balance = 0;
  synchronized int getBalance() { return balance; }
  synchronized void setBalance(int x) { balance = x; }
}
```

Mehrere Threads

```
account.setBalance(account.getBalance() + 100);
```

Kein atomares Inkrementieren
(Lost Update möglich)

Race Condition ohne Data Race

- Critical Sections nicht geschützt
  - Data Races mit Synchronisation eliminiert
  - Aber nicht genügend grosse synchronisierte Blöcke

Kombinationen

	Race Condition	keine Race Condition
Data Race	Fehlerhaftes Programmverhalten	Programm verhält sich zwar korrekt, dennoch formal falsch
kein Data Race	Fehlerhaftes Programmverhalten	**Richtig**

Einfach alles synchronisieren?

- Hilft nichts
  - Race Conditions auch mit Synchronisation möglich
  - Weitere Nebenläufigkeitsfehler (später)
- Synchronisationskosten
  - Synchronisation ist relativ teuer
  - Cache-Kohärenz, verhindert Optimierung etc.

Wann kann man auf Synchronisation verzichten?

Synchronisation: Verzichtbare Fälle

- Immutability (Unveränderlichkeit)
  - Objekte mit nur lesendem Zugriff
- Confinement (Einsperrung)
  - Objekt gehört nur einem Thread zu einer Zeit

Immutable Objects

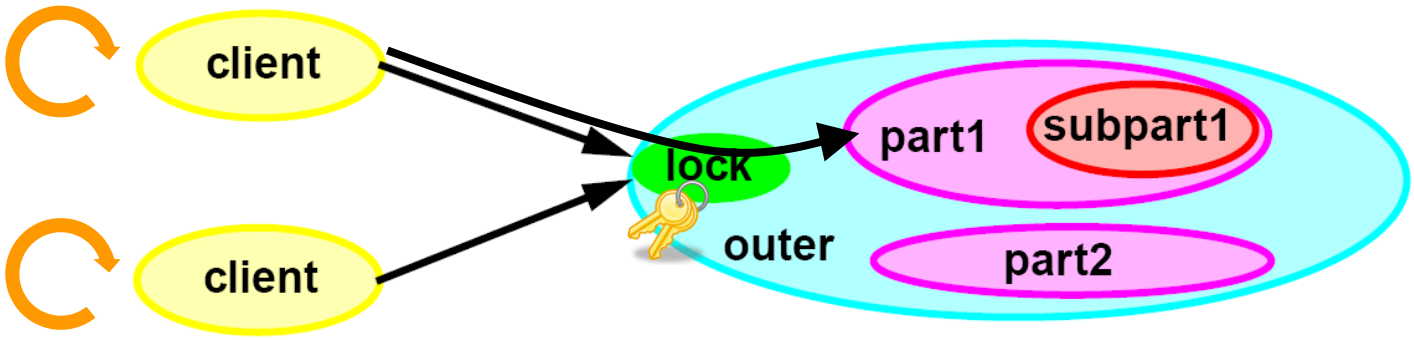
- Instanzvariablen sind alle final
  - Primitive Datentypen
  - Referenzen wiederum auf Immutable Objekte
- Methoden mit nur lesendem Zugriff
  - Konstruktor initialisiert die Instanzvariablen
- Nach Konstruktor kann Objekt ohne Synchronisation von Threads verwendet werden

Confinement

- Struktur garantiert, dass auf ein Objekt nur durch einen Thread zur gleichen Zeit zugegriffen wird
  - Thread Confinement
    - Objekt gehört nur einem Thread und wird nur von demjenigen verwendet
  - Object Confinement
    - Objekt in anderem bereits synchronisierten Objekt eingekapselt

Object Confinement

- Einkapselung in synchronisiertes Objekt
  - Kann auf innere Objekte nur via synchronisierte Methoden des äusseren zugreifen
- Keine Synchronisation für innere nötig



© D. Lea. Concurrent Programming in Java, Slides

Object Confinement: Beispiel

```
class ProductDatabase {
  private HashMap<String, Product> productMap = new HashMap<>();

  public synchronized void addProduct(String name, String details) {
    productMap.put(name, new Product(details));
  }

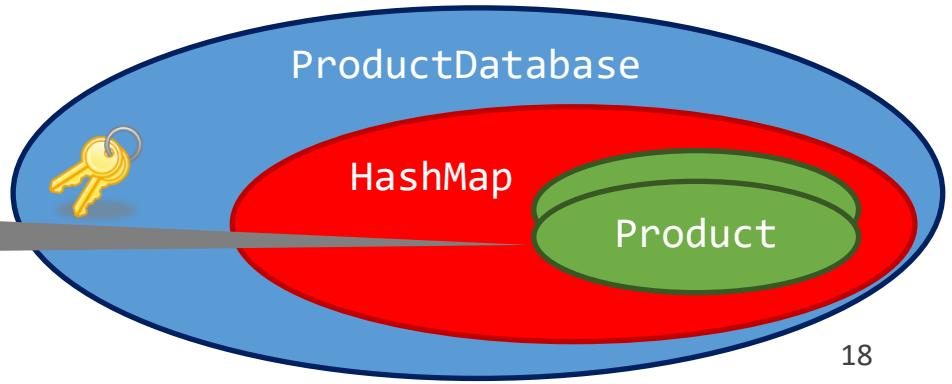
  public synchronized String getProductDetails(String name) {
    return productMap.get(name).getDetails();
  }

  public synchronized void notifySale(String name) {
    productMap.get(name).increaseSales();
  }
}
```

Gekapselt

Gekapselt

Braucht keine innere Synchronisation



Achtung: Kapselungsbrüche

- Inneres Objekt ist aussen zugreifbar
  - `HashMap<String, Product> productMap`
- Rückgabe einer Referenz auf inneres Objekt
  - ```
    Product getLatestProduct() {
      return productMap.get("...");
    }
    ```
- Holder installiert selber Referenz ausserhalb
  - ```
    public void fillCatalog(List<Product> list) {
      list.add(productMap.get("..."));
    }
    ```
- Inneres Objekt gibt selber this raus
  - ```
    public void increaseSales() {
      sales++; globalLastSale = this;
    }
    ```
    ```

Begriff «Thread Safety»

- Klassen / Methoden, die intern synchronisiert sind
  - Keine Race Conditions innerhalb dieses Codes
  - Kritischer Abschnitt nur pro Methode erfüllt
- Aber
  - Kein kritischer Abschnitt über mehrere Methodenaufrufe
  - Andere Nebenläufigkeitsfehler auch möglich
- Keine durchgängige Definition
  - Spezifikation überprüfen

Java Collections

Sind die Java Collections Thread-safe?

Java Collections und Thread Safety

Version	Beispiele	Thread-sicher
Alte Java 1.0 Collections	Vector, Stack, Hashtable	JA
Moderne Collections (java.util, Java > 1.0)	HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap	NEIN
Concurrent Collections (java.util.concurrent)	ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList, …	JA

Wieso sind die modernen Collections nicht mehr Thread-sicher?

Thread-Safe Collections: Diskussion

- Oft ist Synchronisation nicht nötig
  - Confinement => Unnötige Synchronisationskosten
- Synchronisation meist ungenügend
  - Elemente sind nicht synchronisiert
  - Iteration der Elemente ist nicht synchronisiert
- Alte Java 1.0 Collections sind historisch Thread-sicher
  - Wegen Rückwärtskompatibilität immer noch so

Concurrent Collections

- Effiziente Thread-sichere Collections
  - `java.util.concurrent`
  - Geeignet für starke Nebenläufigkeit («Contention»)
- Schwach konsistente Iteratoren
  - Keine ConcurrentModificationExceptions
  - Sehe nebenläufige Updates bei Iteration vielleicht nicht

Achtung: Verstecktes Multi-Threading

- Finalizers
  - Laufen über separaten Finalizer-Thread
- Timers
  - Handler durch separaten Thread ausgeführt (ausser GUI)
- Externe Libraries & Frameworks
  - Z.B. Abarbeitung von Web-Service Aufrufen

Gefahr von Race Conditions

Nested Locks

Thread 1

```
synchronized(listA) {
  synchronized(listB) {
    listB.addAll(listA);
  }
}
```

Thread 2

```
synchronized(listB) {
  synchronized(listA) {
    listA.addAll(listB);
  }
}
```

Welches Problem kann hier auftreten?

Deadlock Szenario

Thread 1

```
synchronized(listA) {
  synchronized(listB) {
    listB.addAll(listA);
  }
}
```

Thread 2

```
synchronized(listB) {
  synchronized(listA) {
    listA.addAll(listB);
  }
}
```

Thread 1	Thread 2	Gesperrte Objekte
synchronized(listA)		listA
	synchronized(listB)	listA, listB
synchronized(listB) => blockiert		listA, listB
	synchronized(listA) => blockiert	listA, listB

Beide Threads haben sich gegenseitig ausgesperrt

Anderes Deadlock Beispiel

```
class BankAccount {
  private int balance;

  public synchronized void transfer(BankAccount to, int amount) {
    balance -= amount;
    to.deposit(amount);
  }

  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

Thread 1

```
a.transfer(b, 20);
```

Thread 2

```
b.transfer(a, 50);
```

Sehen Sie das Deadlock-Problem hier?

Anderes Deadlock Beispiel

```
class BankAccount {
  private int balance;

  public synchronized void transfer(BankAccount to, int amount) {
    balance -= amount;
    to.deposit(amount);
  }

  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

Implizit geschachtelter Lock

Thread 1

```
a.transfer(b, 20);
```

Äquivalent zu:

```
synchronized(a) {
  synchronized(b) { … }
}
```

Thread 2

```
b.transfer(a, 50);
```

Äquivalent zu:

```
synchronized(b) {
  synchronized(a) { … }
}
```

Deadlocks

- Einige Threads sperren sich gegenseitig so, dass keiner von denen weitermachen kann
- Programm mit potentiellem Deadlock ist inkorrekt
  - Threads können sich plötzlich gegenseitig blockieren

Spezialfall Livelocks

- Threads haben sich gegenseitig permanent blockiert
  - Führen aber noch Warteinstruktionen aus
  - Verbrauchen CPU während Deadlock

Thread 1

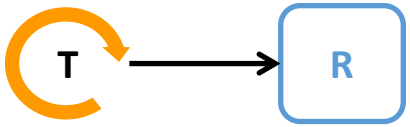
```
b = false;
while (!a) { }
…
b = true;
```

Thread 2

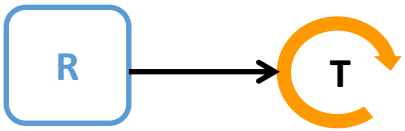
```
a = false;
while (!b) { }
…
a = true;
```

Deadlock Erkennung

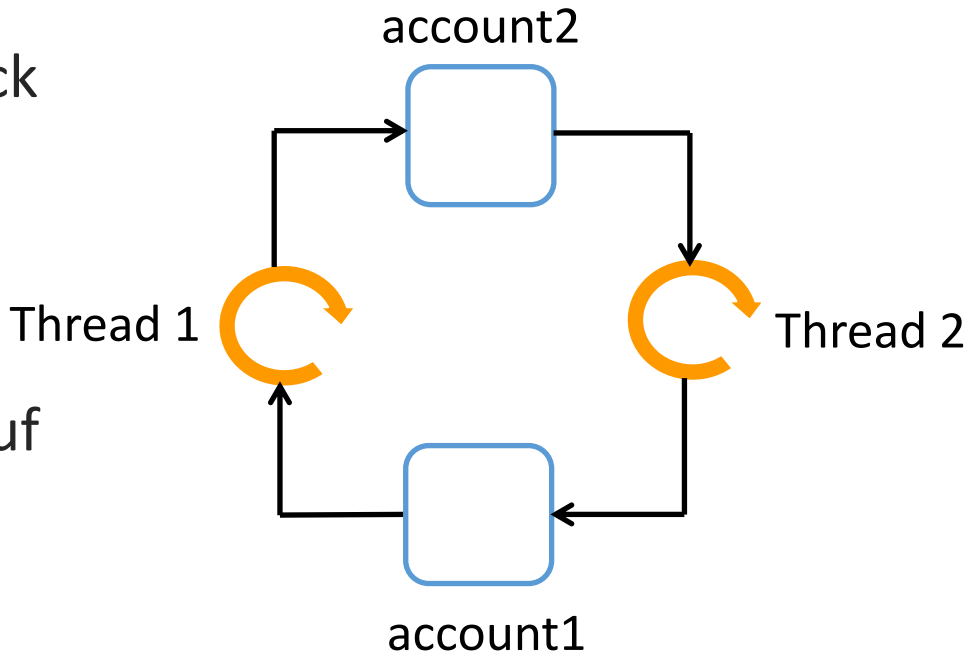
- Betriebsmittelgraph
  - Thread T wartet auf Lock von Ressource R



  - Thread T besitzt Lock auf Ressource R



R. Holt 1970



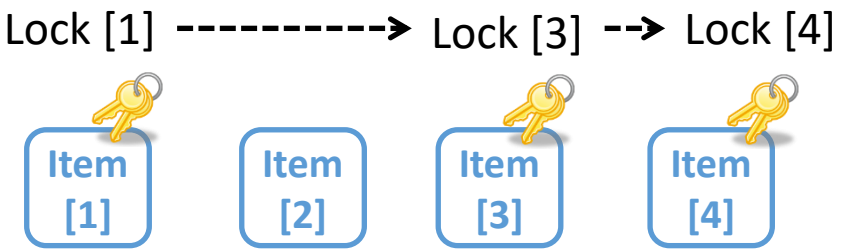
Deadlock ⇔ Zyklus im Betriebsmittelgraph

Deadlock Voraussetzungen

- Geschachtelte Locks
- Zyklische Warteabhängigkeiten

Deadlock Vermeidung (1)

- Lineare Sperrordnung der Ressourcen einführen
  - Nur geschachtelt in aufsteigender Reihenfolge sperren

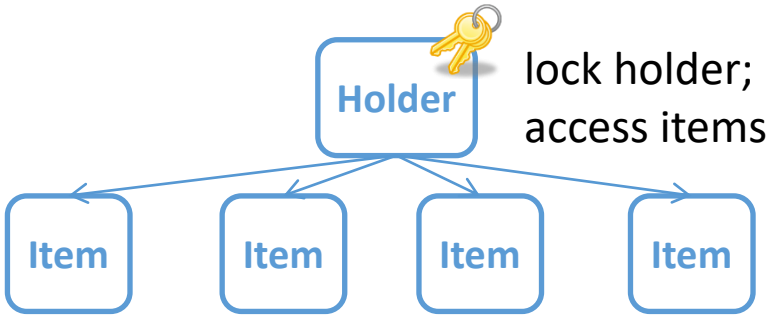


Konten nur nach aufsteigender Nummer sperren

Eliminiert zyklische Warteabhängigkeiten

Deadlock Vermeidung (2)

- Grobgranulare Locks wählen
  - Wenn lineare Sperrordnung nicht möglich/sinnvoll



Sperre gesamte Bank
bei Kontenzugriff

Eliminiert Schachtelung von Locks

Starvation

- Ein Thread kriegt nie die Chance, auf eine Ressource zuzugreifen
  - Obwohl die Ressource immer wieder frei wird (kein Deadlock oder Livelock)
  - Andere Threads können ihn dauernd überholen und die Ressource wegschnappen

```
do {
  success = account.withdraw(100);
} while (!success);
```

Starvation möglich

Starvation: Diskussion

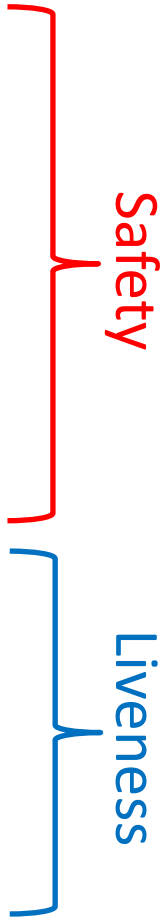
- Starvation ist ein Liveness/Fairness Problem
  - Threads können für unbestimmte Zeit aufgehalten werden (abhängig von Scheduling)

Starvation Vermeidung

- Faire Synchronisationskonstrukte
  - Länger wartende Threads mit erfüllter Bedingung haben Vortritt
  - Fairness einschalten in Java Semaphore, Lock & Condition, Read-Write Lock
- Java Monitor hat ein Fairness-Problem
  - Starvation-anfällig, vor allem bei vielen Threads

Parallelität Korrektheitskriterien

- Keine Race Conditions
  - Kritische Abschnitte auf gemeinsame Ressourcen sind genügend synchronisiert
- Keine Deadlocks
  - Threads können sich nicht gegenseitig für unbeschränkte Zeit sperren
- Keine Starvation
  - Wenn ein Thread auf eine Bedingung wartet, soll er nach einer bestimmten Zeit fortschreiten können, sofern die Bedingung genügend oft erfüllt wird



Rückblick: Lernziele

- Race Conditions, Deadlocks und Starvation verstehen, erkennen und verhindern
- Korrektheitsbedingungen für Nebenläufigkeit kennen
- Möglichkeiten zum sicheren Verzicht auf Synchronisation erfassen
- Thread-Sicherheit bei Collections überblicken

Weitere Lektüre (fakultativ)

- B. Goetz et al. Java Concurrency in Practice, Addison Wesley, 2006
  - Kapitel 2: Sharing Objects (Immutability, Confinement)
  - Kapitel 5: Building Blocks (Collections & Thread Safety)
  - Kapitel 10: Avoiding Liveness Hazards
- Historische Paper (bei Interesse)
  - Dijkstra, E. W. (1971, June). Hierarchical ordering of sequential processes. Acta Informatica 1(2): 115–138.
    - Deadlocks am Beispiel von Dining Philosophers

Anhang (Fakultatives Selbststudium)

Thread Prioritäten & Starvation

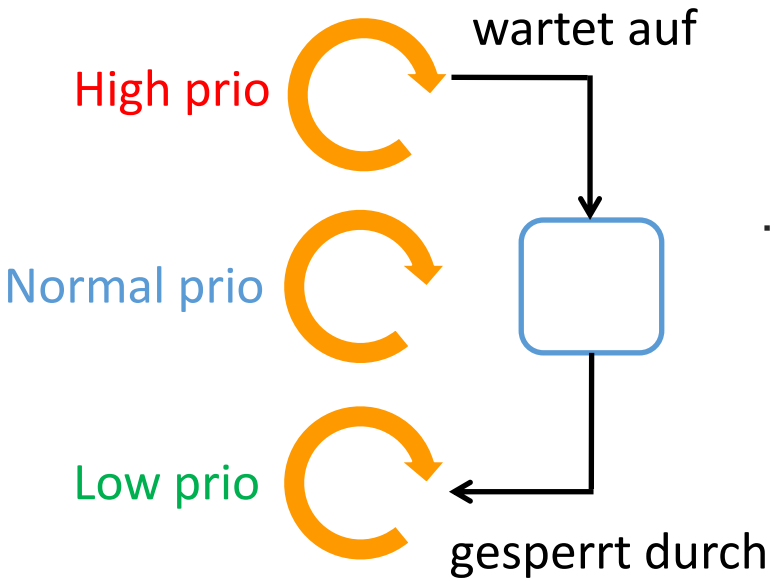
Thread Prioritäten & Starvation

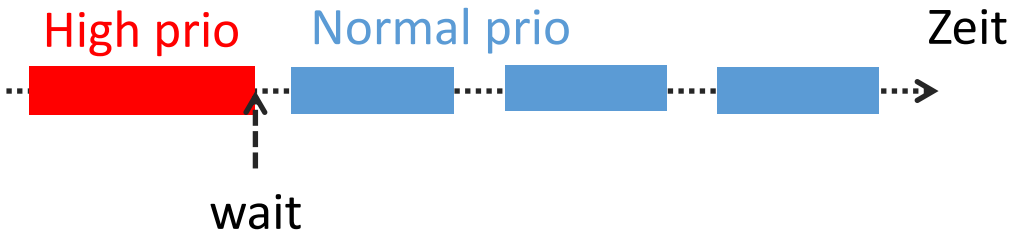
- Thread Prioritäten
  - `myThread.setPriority(priority)`
  - 1: MIN_PRIORITY, 5: NORM_PRIORITY, 10: HIGH_PRIORITY
- Scheduling vom Betriebssystem abhängig
  - Meist haben hochprioritäre Threads immer Vorrang
  - Preemption unter Threads mit gleicher Priorität
  - Windows hat nur 6 Prioritäten

Starvation mit Thread Prioritäten möglich

Priority Inversion

- Hoch prioritärer Thread wartet auf Bedingung von tief prioritärem Thread
  - Normale Threads können dazwischen laufen
  - Tief und hoch prioritärer Thread verhungern





Letzte Vorlesung – Quiz

```
var exchanger = new Exchanger<Integer>();
for (int count = 0; count < 2; count++) {
  new Thread(() -> {
    for (int in = 0; in < 5; in++) {
      try {
        int out = exchanger.exchange(in);
        System.out.println(
          Thread.currentThread().getName() + " got " + out);
      } catch (InterruptedException e) { }
    }
  }).start();
}
```

Welche Ausgabe erwarten Sie?