Nebenläufige Programmierung in Java

Kurzüberblick im WS17/18

Georg Ringwelski

Agenda

- Nebenläufige Threads in Java
- Leser-Schreiber Probleme
- Ansatz 1: Unveränderbare Objekte
- Ansatz 2: Synchronisierung
  - Schlossvariablen (Keyword „synchronized“)
  - Semaphore (Klasse java.util.concurrent.Semaphore)
  - Probleme: Deadlock, Livelock, Aushungern
- Best Practices

Nebenläufige Threads in Java

Einzelübung (15 min): Schreiben Sie ein ausführbares Java Programm, das zwei nebenläufige Threads startet, die jeweils für 5 Sekunden in zufälligen Zeitabständen eine Ausgabe an der Konsole machen. Wenn beide Threads fertig sind wird das vom Haupt-Thread ausgegeben

Wählen Sie einen folgender Ansätze:

a) Ihre Thread-Klasse erweitert java.lang.Thread z.B. als anonyme innere Klasse

b) Ihre Klasse implementiert java.lang.Runnable und wird einem Thread-Objekt im Konstruktor mitgegeben

c) Ihre Klasse implementiert java.lang.Runnable und wird von einem java.util.concurrent.Executor-Objekt nebenläufig gestartet.

Agenda

- Nebenläufige Threads in Java
- Leser-Schreiber Probleme
- Ansatz 1: Unveränderbare Objekte
- Ansatz 2: Synchronisierung
  - Schlossvariablen (Keyword „synchronized“)
  - Semaphore (Klasse java.util.concurrent.Semaphore)
  - Probleme: Deadlock, Livelock, Aushungern
- Best Practices

Leser Schreiber Probleme

- Wenn nebenläufige Threads auf gemeinsame Variablen zugreifen entstehen Probleme
- Bsp: (c=c+1; | c=c-1;) kann zu folgendem werden:

Thread A: Retrieve c.
Thread B: Retrieve c.
Thread A: Increment retrieved value; result is 1.
Thread B: Decrement retrieved value; result is -1.
Thread A: Store result in c; c is now 1.
Thread B: Store result in c; c is now -1.

- Demo: CounterApp.java

Ansatz 1: Unveränderbare (immutable) Objekte

Der Zustand eines „Immutable Object“ kann nach der Erzeugung nicht mehr verändert werden, dazu sollten folgende Regeln eingehalten werden

→Alle Attribute „private final“

→Keine setter

→Überschreiben in Subklassen verbieten (Klasse „final“ deklarieren)

→Wenn änderbare Objekte in Attributen referenziert werden, dann dürfen keine Methoden zur Verfügung gestellt werden, die diese Objekte verändern.

→Wenn änderbare Objekte in Attributen referenziert werden, dann dürfen diese nicht als Parameter an andere weitergegeben werden (vorher clone()) wg. „Call-by-reference-Effekt“

In immuatble objects gibt es keine Leser-Schreiber Probleme!

Ansatz 1: Unveränderbare (immutable) Objekte

Diskussion im Plenum:

1. Kann man Counter.java für immutable objects realisieren?

2. Welche Klassen kann man generell für immutable objects realisieren, welche nicht?

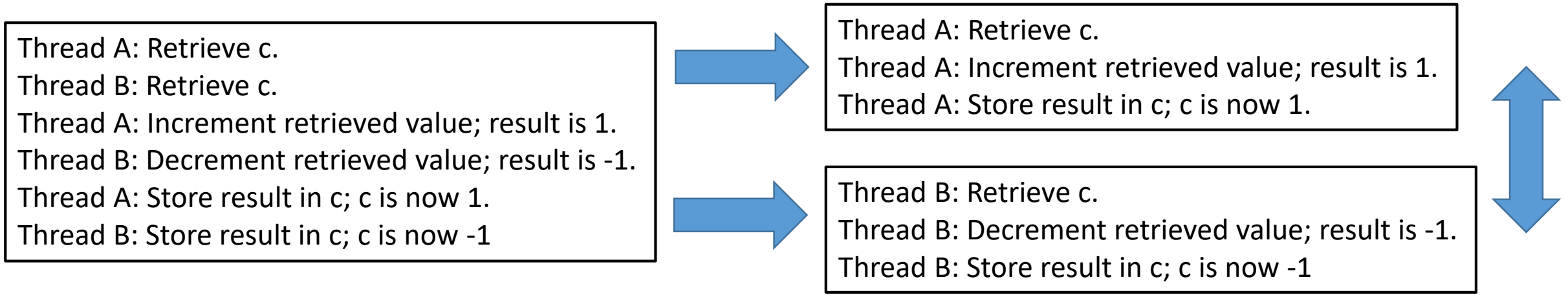
3. Kann man die Steuerung eines mobilen Roboters als immutable object realisieren?

Agenda

- Leser-Schreiber Probleme
- Ansatz 1: Unveränderbare Objekte
- Ansatz 2: Synchronisierung
  - Schlossvariablen (Keyword „synchronized“)
  - Semaphore (Klasse java.util.concurrent.Semaphore)
  - Probleme: Deadlock, Livelock, Aushungern
- Best Practices

Ansatz 2: Synchronisierung

„Synchronisierung“ werden Techniken genannt, mit denen Leser-Schreiber-Probleme durch Verhindern von Thread-Wechseln an den kritischen Stellen verhindert wird:



Synchronisierung mit Schlossvariablen

Eine Schlossvariable ist eine gemeinsame Variable (mutex), die Thread-Wechsel für einen bestimmten Bereich (kritischer Abschnitt) verbietet

```
mutex.lock()
        Thread A: Retrieve c.
        Thread A: Increment retrieved value; result is 1.
        Thread A: Store result in c; c is now 1.
mutex.release()
```

- Wenn Thread A mutex.lock() aufgerufen hat wartet der nächste Thread (zB B), bis A mutex wieder frei gibt
- In Java: synchronized methods oder synchronized statements

Synchronized Methodes

Durch den Modifikator „synchronized“ können Methoden eines Objekts miteinander synchronisiert werden

- Jedes Objekt erhält **genau eine** Schlossvariable
- Alle synchronized-Methoden der Klasse nutzen diese gemeinsam
- Fallstudie Counter.java
  - Ist das Leser-Schreiber-Problem gelöst?
  - Wie verhält sich die Laufzeit?
- Das funktioniert auch bei statischen Methoden (hier wird die Schlossvariable im Class-Objekt der Klasse gehalten

Synchronized Statements

Synchronized Methodes sind nicht immer geeignet

- Es findet zu viel unnötige Synchronisierung statt, die das System verlangsamt
- Eigentlich müssen oft nicht alle Methoden miteinander synchronisiert werden, so dass man mehrere Schlossvariablen bräuchte (für jede separate Funktionalität eine)

Synchronized statements erlauben die explizite Definition des synchronisierenden Objekts für Sequenzen von Statements

Bsp:

```
Object lock_1 = new Object();
public void addName(String name) {
        synchronized(lock_1) { lastName = name; }
        nameList.add(name);
}
```

Beispiel Synchronized Statements

Übung in Kleingruppen Realisieren Sie einen Counter mit 2 Zählvariablen, die in 2 Threads verändert werden und messen Sie die Performanz

1. Synchronisieren alle Methoden
2. Synchronisieren Sie die kritischen Abschnitte über das Counter-Objekt
3. Erstellen Sie zwei Objekte und synchronisieren jeweils zwei kritische Abschnitte, so dass sie für jeden einzelnen Counter eine Schlossvariable einsetzen

Agenda

- Leser-Schreiber Probleme
- Ansatz 1: Unveränderbare Objekte
- Ansatz 2: Synchronisierung
  - Schlossvariablen (Keyword „synchronized“)
  - Semaphore (Klasse java.util.concurrent.Semaphore)
  - Probleme: Deadlock, Livelock, Aushungern
- Best Practices

Semaphore

Semaphore erlauben den gleichzeitigen Zugriff mehrerer Objekte bis zu einer definiert Maximalzahl auf einen kritischen Abschnitt

- Schlossvariablen sind Semaphore mit dem Wert 1
- In Java als Klasse java.util.concurrent.Semaphore realisiert (vgl. Java API)
- Es bietet sich ein Ansatzpunkt für komplexere Synchronisierungen wie
  - Threads benötigen unterschiedlich viele „permits“
  - Fairness-Regeln bei der Vergabe des nächsten freien permit
  - …

Wiederholung

- Wie werden in Java nebenläufige Threads gestartet?
- Wie kann man in Java einen Thread anhalten und warten lassen?
- Wie kann man einen Thread auf einen anderen warten lassen?

- Was sind immutable objects, warum sind sie beim Multithreading interessant?
- Was ist das Leser-Schreiber-Problem, wann kann es auftreten?
- Wie kann man es verhindern?

Agenda

- Leser-Schreiber Probleme
- Ansatz 1: Unveränderbare Objekte
- Ansatz 2: Synchronisierung
  - Schlossvariablen (Keyword „synchronized“)
  - Semaphore (Klasse java.util.concurrent.Semaphore)
  - Probleme: Deadlock, Livelock, Aushungern
- Best Practices

Probleme die durch Synchronisierung bzw. Nebenläufigkeit entstehen

- **Deadlock**: z.B. A wartet auf B und B wartet auf A, oder mit mehr Beteiligten im Zyklus (Bsp: speisende Philosophen)

  → Threads blockieren sich gegenseitig

- **Livelock**: z.B. A reagiert auf B und B reagiert auf die Reaktion von A und B reagiert auf die Reaktion von A …

  → Threads sind nicht blockiert, aber zu beschäftigt um ihre Aufgaben wirklich abzuschließen

- **Aushungern** (Starvation): z.B. A muss immer warten, weil andere Threads aus der Warteschlange an der Semaphore/Schlossvariable immer Vorrang bekommen

  →Threads kommen nie dazu ihre Arbeit zu machen

Probleme die durch Synchronisierung entstehen

Übung in drei Kleingruppen (15 min)

Konstruieren Sie ein Beispiel in Java in dem ein deadlock/livelock/starvation so auftritt, dass man das an der Konsolenausgabe erkennen kann.

Präsentieren Sie das Beispiel im Plenum und…

- erklären Sie, welcher Thread dabei seine Arbeit nicht machen kann.
- erklären Sie, wie man das Problem lösen könnte

Probleme die durch Synchronisierung entstehen

Diskussion im Plenum

Welche der folgenden Probleme können bei der Steuerung eines mobilen Roboters auftreten und müssen in der Software adressiert werden (wir gehen davon aus, dass Sensoren und Aktoren nebenläufig gesteuert werden)

- Leser-Schreiber Problem
- Aushungern
- Verklemmungen
- Livelocks

Agenda

- Leser-Schreiber Probleme
- Ansatz 1: Unveränderbare Objekte
- Ansatz 2: Synchronisierung
  - Schlossvariablen (Keyword „synchronized“)
  - Semaphore (Klasse java.util.concurrent.Semaphore)
  - Probleme: Deadlock, Livelock, Aushungern
- Best Practices

Zusammenfassung,
Best Practices in der nebenläufigen Programmierung

1. Benutzen Sie, wenn möglich, immer immutable objects
2. Synchronisieren Sie alle Variablen, die von nebenläufigen Threads verändert werden könnten (auch wenn sie glauben, dass es keine Probleme geben kann)
3. Synchronisieren Sie möglichst wenig, also nur das Schreiben von Variablen, die laut Regel 2 synchronisiert werden müssen. Halten Sie die kritischen Abschnitte möglichst klein.
4. Überlegen Sie genau, ob es zu deadlocks, livelocks oder starvation kommen kann und verhindern Sie das durch geeignete Abhängigkeiten und Scheduling der wartenden Threads.