LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Nebenläufigkeit mit Java

Einheit 03: Synchronisation

Lorenz Schauer
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Synchronisation von Threads

- Locks
- Java Monitor-Konzept
  - Lock Freigabe
- Zusammenspiel zwischen `wait()`, `notify()` bzw. `notifyAll()`

Weitere Aspekte

- Monitore sind reentrant
- Warum nicht alles Synchronisieren?
- Deadlocks

Praxis:

- Parkhaus
- Philosophen-Problem

Lernziele

- Das Java Thread-Konzept kennenlernen und verstehen
- Threads mittels Monitoren synchronisieren können
- Java-Threads in weiteren Übungen vertiefen

Beispiel zu ReentrantLock (Letzte Stunde)


```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Test {
   public static void main(String[] args) {
      final Lock lock = new ReentrantLock();

      Runnable r = new Runnable()
      {
        int x=0;
        int y=2;
        boolean flag = true;

        @Override
         public void run()
        {
          while ( flag )
          {
            lock.lock();

            x=x+2;
            y++;

            lock.unlock();

            if ( x==y){
              System.out.println( "Fertig");
             flag=false;
            }
          }
        }
      };
      new Thread( r ).start();
      new Thread( r ).start();
   }
}
```

ReentrantLock implementiert Iface Lock

Mittels Lock lässt sich ein krit. Bereich markieren

Beginn mit: lock()

Ende mit: unlock()

Das Java Monitor-Konzept

Monitore:

- Werden implizit durch die JVM erstellt
- Werden durch eine automatisch verwaltete Sperre realisiert
- Jedes Objekt verfügt über eine Sperre (Lock)
- Eintrittspunkte der Monitore müssen mit dem Schlüsselwort `synchronized` markiert sein
  - Für Klassenmethoden
    - Bsp.: `public synchronized static void doIt(){…}`
  - Für Objektmethoden
    - Bsp.: `public synchronized void makeIt(){…}`
  - Für Blöcke
    - Bsp.: `synchronized (objMitMonitor){…}`
- Ein solcher Eintrittspunkt kann nur betreten werden, wenn das Lock verfügbar ist.
  - Ansonsten muss der Thread warten, solange bis das Lock verfügbar ist.

Das Java Monitor-Konzept

- Ein freies Lock wird beim Betreten einer `synchronized` Methode/Block durch den aufrufenden Thread belegt (oder gehalten)
  - Daraufhin kann kein anderer Thread mehr eine synchronisierte Methode des Objekts betreten (solange bis das Lock wieder freigegeben wird)

- Die statische Methode `Thread.holdsLock()` zeigt an, ob der aktuelle Thread das Lock hält.

- Ein gehaltenes Lock wird freigegeben, wenn:
  - … die synchronisierte Methode verlassen wird
  - … eine Ausnahme erfolgt
  - … ein `wait()` Aufruf getätigt wird.

Lock Freigabe

Durch Ausnahme (Exception)

- Lock wird bei einer unbehandelten RuntimeException in einer `synchronized` Methode/Block automatisch durch JVM freigegeben
  - Da bei einer Exception der Block automatisch verlassen wird

Durch Aufruf von `wait()`

- Thread beendet die Abarbeitung
- Geht in den „Blocked“ Zustand
  - Reiht sich in die Warteschlange des Objekts ein
- Das Lock wird freigegeben
- Der Thread wartet nun auf eine Benachrichtigung durch (`notify()` oder `notifyAll()`), dass er wieder weiterabreiten darf
- Wartende Threads können auch durch einen Interrupt unterbrochen werden
  - Daher: `throws InterruptedException` (Muss abgefangen werden!)
- Aufruf von `wait()`, `notify()` oder `notifyAll()` nur möglich, wenn Lock gehalten wird!
  - Ansonsten Laufzeitfehler: `IllegalMonitorStateException`

Zusammenspiel zwischen `wait()`, `notify()` und `notifyAll()`

Zusammenfassung der Methoden:

- `void wait() throws InterruptedException`
  - Thread wartet an dem aufrufenden Objekt darauf, dass er nach einem notify() bzw. notifyAll() weiterarbeiten kann.
- `void wait( long timeout ) throws InterruptedException`
  - Wartet auf ein notify()/notifyAll() maximal aber eine gegebene Anzahl von Millisekunden. Nach Ablauf dieser Zeit ist er wieder rechenbereit.
- `void wait( long timeout, int nanos ) throws InterruptedException`
  - Etwas spezifischer als vorher
- `void notify()`
  - Weckt einen beliebigen Thread auf, der an diesem Objekt wartet und sich wieder um das Lock bemühen kann.
    - Erhält er das Lock, kann er die Bearbeitung fortführen.
- `void notifyAll()`
  - Benachrichtigt alle Threads, die auf dieses Objekt warten.

Hinweis: `notify()` bzw. `notifyAll()` i.d.R., wenn aufrufender Thread dann auch das Lock freigibt (also am Ende eines `synchronized` Blocks)

- Sonst werden die Threads zwar aufgeweckt, aber das Lock ist immer noch vom aktuellen Thread belegt („signal and continue“-Prinzip)

Zusammenspiel zwischen wait(), notify() und notifyAll()


```
Allgemeines Beispiel zum Zusammenspiel zwischen wait() und notify()

public class MeinKlasse{

   private Data state;

   public synchronized void op1() throws InterruptedException {
      while (!cond1) wait();
      // modify monitor state
      notify();
      // or notifyAll();
   }

   public synchronized void op2() throws InterruptedException {
      while (!cond2) wait();
      // modify monitor state
      notify();
      // or notifyAll();
   }
}
```

Die `while`-Schleife wird deshalb gebraucht, da bei Fortführung die Synchronisationsbedingung nicht notwendigerweise gelten muss!
Ein einfaches `if` kann evtl. nicht ausreichen

Konkretes Beispiel: Erzeuger-Verbraucher


```
public class Speicher {

    private double[] speicher;
    private int pointer;
    private int size;

    private Random r = new Random();

    public Speicher(double[] speicher){
        this.speicher = speicher;
        this.size = speicher.length;
        this.pointer=0;
    }

    public synchronized void produce() throws InterruptedException {

        while(pointer>=size) this.wait();

        //modify monitor state
        double element = r.nextDouble();
        speicher[pointer]= element;
        System.out.println("Producer produziert: "+element+" und legt es auf Poisiton "+pointer);
        pointer++;

        notify();
    }

    public synchronized void consume() throws InterruptedException{

        while(pointer<=0) wait();

        //modify monitor state
        pointer--;
        System.out.println("Consumer konsumiert element: "+speicher[pointer]+" an Position "+pointer);

        notify();
    }
```

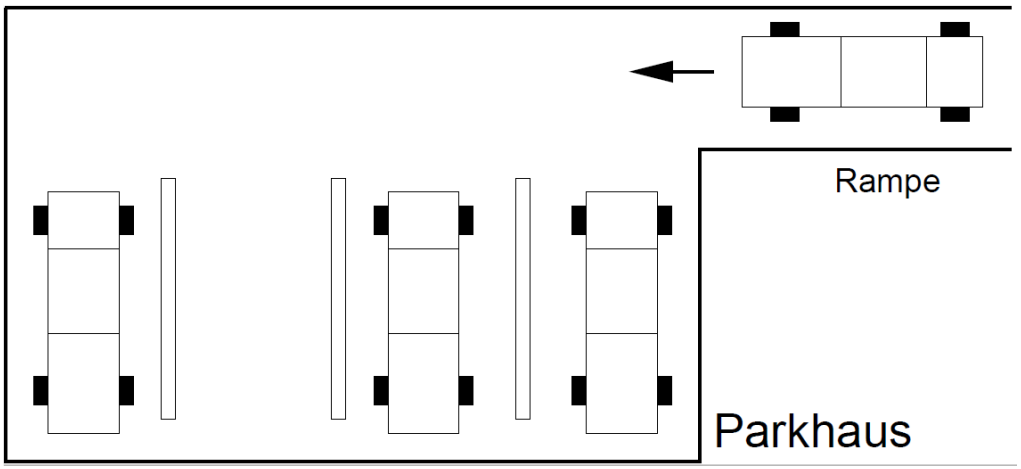
Konkretes Beispiel: Semaphore


```
class Semaphore {

   private int value;

   public Semaphore (int initial) {
      value = initial;
   }

   public synchronized void down() throws InterruptedException {
      while (value==0) wait();
      value--;
   }

   public synchronized void up() {
      value++;
      notify();
   }
}
```

Programmieraufgabe

Schreiben Sie eine kleine Simulation in Java, die den folgenden Anforderungen gerecht wird:

- In einem Parkhaus gibt es MAX Stellen an Parkplätzen.
- Zudem wollen Autos über eine Rampe in und aus dem Parkhaus fahren, welche zur gleichen Zeit immer nur von maximal einem Auto benutzt werden kann.
- Ein Auto kann also nur in das Parkhaus einfahren, wenn die Rampe frei ist und sich mindestens noch ein freier Platz im Parkhaus befindet.
- Ein Auto kann aus dem Parkhaus ausfahren, wenn die Rampe frei ist.
- Simulieren Sie Ihre Implementierung mit mind. 10 Autos, die alle ein- und ausfahren wollen.
- Sorgen Sie für ausreichende Ausgaben, um Ihre Simulation zu testen



Das Java Monitor-Konzept

Monitore sind *reentrant*

- D.h.: Ein Thread der einen synchronisierten Block betritt bekommt den Monitor des Objekts (also hält das Lock).
- Ruft diese Methode eine andere auf, die am gleichen Objekt synchronisiert ist, kann sie sofort eintreten und muss nicht warten!
- Ohne diese Möglichkeit würde Rekursion nicht funktionieren!
- Kann Geschwindigkeitsvorteile bringen, wenn viele synchronisierte Methoden nacheinander aufgerufen werden müssen:

```
StringBuffer sb = new StringBuffer();
synchronized( sb )
{
  sb.append( „Hallo " );
  sb.append( „und Servus, " );
  sb.append( „wie geht’s denn " );
  sb.append( „allerwei?" );
}
```

In `StringBuffer` sind viele Methoden synchronisiert, was dazu führt, das bei jedem Aufruf der Monitor reserviert werden muss, was Zeit kostet.
Daher: Bündelung der Methoden in einem Synchronized Block

Warum nicht gleich alles synchronisieren?

Unerwünschte Nebeneffekte können also durch Markieren der kritischen Abschnitte mittels `synchronized` verhindert werden!

Warum dann nicht gleich jede Methode synchronisieren?

Antwort: Führt zu anderen Problemen:

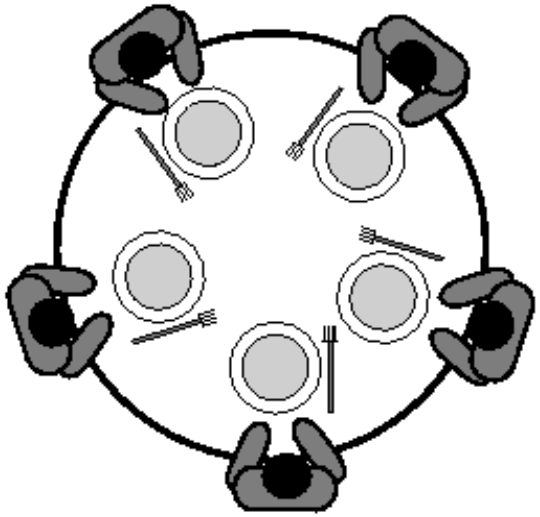
- Synchronisierte Methoden müssen von JVM verwaltet werden, um wechselseitigen Ausschluss zu ermöglichen.
  - Threads müssen auf andere warten können
  - Das erfordert eine Datenstruktur, in der wartende Threads eingetragen und ausgewählt werden
    => Kostet Zeit und Ressourcen!
- Unnötig und falsch synchronisierte Blöcke machen die Vorteile von Mehrprozessormaschinen zunichte.
  - Lange Methodenrümpfe erhöhen die Wartezeit für die anderen!
- Deadlock-Gefahr steigt!

Quelle: http://openbook.rheinwerk-verlag.de/javainsel9/bilder/365_java_09_004.gif

Das Philosophen-Problem

Implementieren Sie das bekannte Philosophen-Problem in Java (zunächst ohne Deadlock-Vermeidung):

- 5 Philosophen hocken an einem Tisch
- Es gibt genau 5 Gabeln
- Jeder Philosoph soll abwechselnd schlafen und essen.
- Will ein Philosoph essen, so versucht er zunächst seine linke Gabel zu nehmen und dann seine rechte.
  - Wenn das klappt, dann isst er und legt anschließend das Besteck wieder zurück
  - Wenn die Gabel nicht vorhanden ist, dann wartet er auf die Gabel, bis sie wieder greifbar ist.
- Das Philosophen-Problem führt höchstwahrscheinlich irgendwann zu einem Deadlock. Wie könnte dieser vermieden werden?
  - Implementieren Sie nun eine Deadlockfreie Variante



Quelle: https://www.dpunkt.de/java/Programmieren_mit_Java/Multithreading/11.html