Übungen zu Verteilte Systeme

Hochschule Regensburg

21.10.2013, Übung 3

Universitätsstraße 31, 93053 Regensburg

Prof. Dr. Jan Dünnweber

Producer-Consumer

- In der heutigen Übung implementieren wir das "klassische" *Producer/Consumer*-Modell in Java
  - Programmieren Sie dazu die Klasse `KitchenCounter`, als Serviertheke auf der nur eine bestimmte (durch den Konstruktor festgelegte) Anzahl von Leberkässemmeln Platz hat.
    Für das Ablegen und Entnehmen von **jeweils einer** Leberkässemmel sollen die *parametermeterlosen*, *öffentlichen* Methoden `put` & `take` bereit gestellt werden
  - Implementieren Sie das interface `Runnable` in der Klasse `CafeWaitress`, so dass in der `run()`-Methode pausenlos Leberkässemmeln hergestellt und auf der Serviertheke abgelegt werden
  - Implementieren Sie das interface `Runnable` zusätzlich in der Klasse `Student`, so dass in der `run()`-Methode ununterbrochen Leberkässemmeln von der Serviertheke entnommen und verspeißt werden
- Alle Methoden sollen Statusmeldungen ausgeben.

Thread-Safety

- Simulieren Sie die Vorgänge in der Cafeteria durch ein Testprogramm mit 8 Studenten, 2 Serviererinnen und einer Theke für bis zu 4 Leberkässemmeln
- Stellen Sie durch die Verwendung eines `ReentrantLock` sicher, dass für die Methoden `put` & `take` *Thread Safety* gewährleistet ist
- Verwenden Sie unterschiedliche `Conditions` für eine *leere* und eine *volle* Theke

Eine Zugsimulation durch Threads

Die Folgende von `Thread` abgeleitete Klasse ist unser Modell für Züge:

```
public class Train extends Thread {

  private String label;
  private SingleTrack track;

  public Train (String label, SingleTrack track) {
    this.label = label;
    this.track = track;
  }

// ...
```

- **Hinweis:** Den Code gibt es auch online:
  `http://www.dpunkt.de/buecher/3213/`

  `fortgeschrittene-programmierung-mit-java-5`

Train.java

Die `run()`-Methode

```
  public void run () {
    try {
      for (int i = 0; i < 2; i++) {
        System.out.println (this.label + "running");
        Thread.sleep (1000);  }
      System.out.println (this.label + "try to enter...");
      this.track.enter ( );
      System.out.println (this.label + "entered!");
      for (int i = 0; i < 3; i++) {
        System.out.println (this.label + "running on SingleTrack");
        Thread.sleep (1000);  }
      System.out.println (this.label + "exiting");
      this.track.leave ();
      for (int i = 0; i < 2; i++) {
        System.out.println (this.label + "running");
        Thread.sleep (1000);  }  }
    catch (InterruptedException e) {
      System.out.println (e);  }
  }
}
```

Eine eingleisige Bahnstrecke

Ein Gleis kann zu jedem Zeitpunkt nur von einem Zug benutzt werden. Zusammenstösse werden mittels `wait` und `notify` verhindert

```
public class SingleTrack {

  private boolean isFree = true;

  public void enter () throws InterruptedException {
    synchronized (this) {
      while (! this.isFree)
        this.wait ();
      this.isFree = false;
    }
  }

  public void leave () throws InterruptedException {
    synchronized (this) {
      this.isFree = true;
      this.notifyAll ();
    }
  }
```

Single Track Simulation

Die Testanwendung:

```
public class Application {

  public static void main (String [] args) {
    SingleTrack track = new SingleTrack  ();

    Train t0 = new Train ("", track);
    Train t1 = new Train ("\t\t", track);

    t0.start ();
    try { Thread.sleep (500); } catch (InterruptedException ignored) { }
    t1.start ();
  }
}
```

- Was muss geändert werden, wenn `Locks` und `Conditions` anstelle von `wait` & `notify` verwendet werden sollen?
- Kann der wechselseitige Ausschluß auch mittels der Klasse `java.util.concurrent.Semaphore` implementiert werden?