TU Berlin
Software and Business Engineering
Prof. Dr. Ingo Weber
Dr. Tobias Heindel
Richard Hobeck

WiSe 2021/22
Aufgabe 11
Block 4
17. Januar 2022

Abgabetermin: 23. Januar 2022

Aufgabe 11
Programmieren I

Hinweise

- Die Abgabe dieser Übungsaufgaben muss bis spätestens Sonntag, den 23. Januar 2022 um 23:59 Uhr im ISIS-Kurs erfolgt sein. Es gelten die Ihnen bekannten Übungsbedingungen.
- Lösungen zu diesen Aufgaben sind als gezippter Projektordner abzugeben. Eine Anleitung zum Zippen von Projekten finden Sie auf der Seite des ISIS-Kurses. *Bitte benutzen Sie einen Dateinamen der Form* `VornameNachname.zip`.
- Bitte beachten Sie, dass Abgaben im Rahmen der Übungsleistung für die Zulassung zur Klausur relevant sind. Durch Plagiieren verwirken Sie sich die Möglichkeit zur Zulassung zur Klausur in diesem Semester.

In der Vorlesung haben Sie das „Consumer-Producer“-Problem kennengelernt, für den Spezialfall eines Puffers, eines „Consumer“s und eines „Producer“s. In der ersten Aufgabe gibt es zwei Variationen. Zum einen geht es um das „Einfügen“ und “Löschen“ *innerhalb* einer *Liste* (anstatt des „Schreibens“ und „Lesens“ in einen *Puffer*) und zum anderen gibt es für das Einfügen und Einsortieren jedes einzelnen Elements in eine fixe Liste einen eigenen Thread, sodass es eine Vielzahl von Threads gibt, die geeignet miteinander interagieren sollten.

```java
public class List<T> {
    private volatile ListEl<T> head;

    public ListEl<T> getHead() {return head;}

    public void setHead(ListEl<T> head) {
        this.head = head;
    }

    public List(){
        head = null;
    }
}
```

```java
class ListEl<T>{
    T val;
    volatile ListEl<T> next;

    ListEl(T v, ListEl<T> n){
        val = v;
        next = n;
    }
}
```

Abbildung 1: Die Klasse für Listen

Aufgabe 11.1 *Nebenläufige Operationen auf Listen* (3 Punkte)

- Schreiben Sie eine Klasse `ListInserter`, die das `Runnable`-Interface implementiert und deren Konstruktor eine Liste vom Typ `List<Integer>` und einen Thread als Formalparameter hat. Am Anfang soll die `run`-Methode auf jeden Fall ein kurze Zeit warten (per Aufruf der `sleep`-Methode). Danach soll dieser Thread einen beliebigen Wert an das Ende der Liste einfügen. Der übergebene Thread des Konstruktors dient der Möglichkeit, mit einem weiteren Thread zu „kommunizieren“ (über Methoden wie `wait()`, `notify()`, `join()`, etc.).

- Schreiben Sie eine Klasse `ListNibbler`, die das `Runnable`-Interface implementiert und deren Konstruktor eine Liste vom Typ `List<Integer>` und einen Thread als Formalparameter hat. Die `run`-Methode durchläuft die Liste und löscht ein zufälliges Element der Liste, also ein `ListEl`-Objekt. Der übergebene Thread des Konstruktors dient der Möglichkeit, mit einem weiteren Thread zu „kommunizieren“ (über Methoden wie `wait()`, `notify()`, `join()`, etc.).
- Schreiben Sie eine `main`-Methode in der Klasse `List`, die eine neue, leere Liste erzeugt, dann geeignet je 20 Threads von neuen `ListInserter`- und `ListNibbler`-Objekten erzeugt, startet und die Liste ausgibt, nachdem alle `ListInserter`-Threads ihre `run`-Methode verlassen haben (und bevor alle `ListNibbler` fertig sind).

Am Ende sollte die Liste immer wieder leer sein. Eine typische Ausgabe sehen Sie in Abbildung 2.

Hinweis Die Hauptschwierigkeit der Aufgabe liegt darin, die Threads *geeignet* aufeinander warten zu lassen, sodass die Liste am Ende wieder leer ist.

Ein Problem bei der Verwendung von (verschachtelten) `synchronized`-Blöcken sind „deadlocks“.[1] Dieses Problem kann z.B. auch bei einer Ringbahn auftreten, wenn Züge im Zyklus auf den nächsten Zug warten. Der Einfachheit halber betrachten wir eine Ringbahn, die nur in eine Richtung fährt.

Aufgabe 11.2 *Synchronization of Threads* **(2 Punkte)**

Betrachten Sie den Code in Abbildung 3. Dort werden drei Threads erzeugt, die für Züge auf einer Ringbahn (mit nur einer Richtung) fahren. Der Ring—wie gesagt, es gibt nur ein Gleis—ist in sechs Abschnitte unterteilt. Sie sollen sicherstellen, dass auf jedem Abschnitt immer höchstens ein Zug ist, und zwar durch geeignete `synchronized`-Blöcke in der `run`-methode (siehe unten) auf Objekten, die bereits im Quellcode existieren. **(Führen Sie keine neuen Variablen oder Objekte ein!)**
Begründen Sie, warum Ihre Implementierung korrekt ist, also

- warum immer wieder ein Zug weiterfährt,
- warum niemals zwei Züge im gleichen Abschnitt sind,
- es möglich ist, dass die Züge aller drei Threads in der Tat auch *gleichzeitig* den Code von `synchronized`-Blöcken ausführen.

Schließlich erweitern Sie den Code, dass eine beliebige Anzahl von Zügen auf einem Ring beliebiger Länge fahren kann.

Aufgabe 11.3 *“Maximale” Anzahl von* Inserter*n und* Nibbler*n* **(2 Zusatzpunkte)**

- Ermöglichen Sie es, dass mehrere *Inserter* gleichzeitig die Liste durchlaufen.
- Ermöglichen Sie es, dass mehrere *Nibbler* gleichzeitig die Liste durchlaufen und Elemente “gleichzeitig” löschen können.

Jeweils dürfen Sie keine deadlocks verursachen, die Struktur der Liste zerstören, o.ä. Fehler verursachen.

[1] Das klassische Beispiel sind die speisenden Philosoph*innen (Dining Philosophers Problem).

```
ListInserter@7b2c1915 insterted 13.
ListInserter@76f65654 insterted 96.
ListInserter@2eeb35f7 insterted 49.
ListInserter@6bdfdec0 insterted 24.
ListInserter@2ec5f578 insterted 32.
ListInserter@7f80b23d insterted 15.
ListInserter@2190b02f insterted 44.
ListInserter@52057e44 insterted 7.
ListInserter@26f7e68e insterted 27.
ListInserter@366697ab insterted 80.
ListInserter@7cde8e41 insterted 64.
ListInserter@108b3bb8 insterted 24.
ListInserter@1e766eff insterted 70.
ListInserter@6c8ceecf insterted 12.
ListInserter@364f4a0 insterted 22.
ListInserter@56b8fc93 insterted 32.
ListInserter@4f8e93d4 insterted 2.
ListNibbler@db7e09 eating 44
ListInserter@1d14dddd insterted 54.
ListNibbler@7f8cc906 eating 24
ListInserter@1bd9dcdd insterted 86.
ListInserter@2c680bcc insterted 75.
ListNibbler@759d72b2 eating 32
ListNibbler@4c73fbe7 eating 13
ListNibbler@3138cf7f eating 64
ListNibbler@37b87480 eating 70
ListNibbler@6351ee7d eating 96
ListNibbler@684d8ab eating 2
ListNibbler@30b11b29 eating 22
ListNibbler@184e6c9b eating 54
ListNibbler@6cc1bfc7 eating 7
ListNibbler@6cc912f eating 80
ListNibbler@6fe70e0e eating 27
ListNibbler@501e0f00 eating 24
ListNibbler@2b530204 eating 86
ListNibbler@74774934 eating 49
[12, 32, 75, 15]
ListNibbler@7610c203 eating 12
ListNibbler@64daeaf8 eating 75
ListNibbler@6d22af49 eating 32
[]

Process finished with exit code 0
```

Abbildung 2: Ausgabe der `ListInserter` und `ListNibbler` Threads.

```
class Zug extends Thread {
  int abschnitt; // der Abschnitt in dem sich der Zug befindet

  public void run() {
    while (true) {
      int next = (abschnitt + 1) % RingBahn.freieAbschnitte.length;
      if (RingBahn.freieAbschnitte[next]) {
        RingBahn.freieAbschnitte[next] = false;
        System.out.println(this + " fährt weiter nach Abschnitt " + next + ".");
        RingBahn.freieAbschnitte[abschnitt] = true;
        abschnitt = next;
      }

      try {
        Thread.sleep((long) (Math.random() * 1000));
      } catch (InterruptedException e) {}
    }
  }

  public Zug(int a, String name) {
    abschnitt = a;
    super.setName(name);
  }
}

public class RingBahn {
  private static final Zug x = new Zug(1, "Zug x");
  private static final Zug y = new Zug(3, "Zug y");
  private static final Zug y = new Zug(5, "Zug z");
  static volatile Boolean[] freieAbschnitte = {true, false, true, false, true, false};

  public static void main(String[] args) {
    y.start();
    x.start();
    z.start();
  }
}
```

Abbildung 3: Ringbahn Quellcode