

## Telekommunikation / Elektrotechnik Verteilte System (TeV Sys)

### Nebenläufigkeit – Übung 2


---

#### Einführung

Weitere Informationen zum Thema Synchronisation von Threads mittels *Monitoren* finden Sie im Java Tutorial von Sun zum Thema Concurrency (<http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>).

- Synchronization (<http://java.sun.com/docs/books/tutorial/essential/concurrency/sync.html>)  
Wie benutzt man das Schlüsselwort *synchronized*.
- Guarded Blocks (<http://java.sun.com/docs/books/tutorial/essential/concurrency/guardmeth.html>)  
Wie benutzt man die Schlüsselwörter *wait* und *notify*.

#### Aufgaben

1.  **Multithreaded-Zähler.** Lösen Sie die Probleme des Counters (Zähler) mit Hilfe von Monitoren.
2. **CircularBuffer.** Folgender Code implementiert einen zirkulären Buffer. Ein zirkulärer Buffer ist eine effiziente Implementierung einer FIFO -Queue.

```
public class CircularBuffer {
    int[] buffer;
    int capacity;
    int insert;      // next position to put an element
    int remove;      // next position to get an element
    int size;

    CircularBuffer(int capacity) {
        this.buffer = new int[capacity];
        this.capacity = capacity;
        this.size = 0;
        this.insert = 0;
        this.remove = 0;
    }

    boolean isEmpty() {
        return size == 0;
    }

    boolean isFull() {
        return size == capacity;
    }


    int size() {
        return size;
    }

    int get() {
        if (isEmpty()) {
            throw new IllegalStateException();
        }
        size--;
        int result = buffer[remove];
        remove = (remove + 1) % capacity;
        return result;
    }

    void put(int value) {
        if (isFull()) {
            throw new IllegalStateException();
        }
        size++;
        buffer[insert] = value;
        insert = (insert + 1) % capacity;
    }
}
```

---

Die Implementation ist korrekt, zumindest solange die Nebenläufigkeit nicht ins Spiel kommt. Angenommen zwei Threads teilen sich eine Instanz eines zirkulären Buffers und benutzen beide die Methode *put*.

- a) Skizzieren Sie ein Szenario, in dem es durch nebenläufiges Ausführen der Methode *put* zu Problemen kommt.
- b) Welches sind die kritischen Abschnitte in der Klasse *CircularBuffer*?
- c)  Wie lösen Sie das Problem in Java?

3. **Collections in Java.** In Aufgabe 2 haben Sie einen zirkulären Buffer thread-safe gemacht. In Java gibt es viele vordefinierte abstrakte Datentypen die sogenannten Collections (siehe <http://java.sun.com/javase/6/docs/api/>), beispielsweise die Klasse *ArrayList*, die eine verlinkte Liste implementiert.

- a) Ist die Klasse *ArrayList* thread-safe?

Geben Sie den entsprechenden Abschnitt aus der JavaDoc zur Klasse *ArrayList* an.

- b) Falls die Klasse *ArrayList* nicht thread-safe ist, wie müsste Sie thread-safe gemacht werden?


4. **Producer/Consumer-Problem.** Beim Producer/Consumer-Problem gibt es einen Erzeuger (Producer) und einen Verbraucher (Consumer). Der Erzeuger schreibt Daten in einen Puffer fixer Grösse, der Verbraucher liest Daten aus diesem Puffer.

Auf dem Claroline finden Sie eine Implementation zu diesem Problem.

- a) Folgenden Fehler erhalten Sie nach einer gewissen Zeit:

```
22
Exception in thread "Thread-0" java.lang.IllegalStateException
    at CircularBuffer.put(CircularBuffer.java:86)
    at Producer.run(Producer.java:17)
23
```

Was läuft da schief? Korrigieren Sie den Fehler.

- b)  Der Consumer implementiert ein busy waiting, was sich vor allem problematisch auswirkt, wenn seine `sleepTime == -1` ist, er also dauernd versucht, Werte zu lesen, der Producer aber nur alle 100 ms neue Werte generiert (`sleepTime == 100`). Konsequenz: die CPU-Auslastung beträgt 100%.

```
while (true) {
    if (!buffer.isEmpty()) {
        int value = buffer.get();
        System.out.println(value);
    }
    //
    if (sleepTime >= 0) {
        // wait for a random time between 0 and sleepTime
        try {
            Thread.sleep(random.nextInt(sleepTime));
        } catch (InterruptedException e) {
        }
    }
}
```

Das muss nicht sein!

Benutzen Sie die Synchronisationsmöglichkeiten von Monitoren, um die Implementation zu verbessern. Verbessern Sie den Consumer und den Producer.