**Telekommunikation / Elektrotechnik**
**Verteilte System (TeVSys)**

# Nebenläufigkeit – Lösung zu Übung 2

1. **Multithreaded-Zähler**.
   (siehe separater Code)

2. **CircularBuffer**.
   a) Szenario: Angenommen Thread 1 wird zwischen Zeile 6 und Zeile 7 unterbrochen und Thread 2 läuft durch, dann werden die Werte von Thread 1 und Thread 2 an dieselbe Stelle im Ringbuffer geschrieben.

```
1      void put(int value) {
2             if (isFull()) {
3                    throw new IllegalStateException();
4             }
5             size++;
6             buffer[insert] = value;
7             insert = (insert + 1) % capacity;
8      }
```

   Es sind noch viele andere Szenarien denkbar.

   b) Kritische Abschnitte sind jene Bereiche, wo sich nicht zwei Threads gleichzeitig befinden dürfen. Im Ringbuffer sind dies die Methoden get und put. Teilaufgabe a) zeigt, was bei der Methode put bei gleichzeitigem Zugriff schiefgehen kann. Ein analoges Beispiel kann man für die Methode get konstruieren.

   c) Die kritischen Bereiche müssen durch einen Monitor geschützt werden. Da die ganze Methode geschützt werden muss am besten **synchronized** vor die Methoden schreiben.

```
public class CircularBuffer {
       // …
       synchronized int get() {
              // …
       }

       synchronized void put(int value) {
              // …
       }
}
```

   **Bemerkungen**

   Mit obigen Massnahmen ist der Circular Buffer soweit geschützt, dass er nicht korrupt wird. Für eine konkrete Implementation sind ev. noch weitere Massnahmen notwendig. Beispiel:

   Reader-Threads sind wie folgt programmiert:

```
1      CircularBuffer buffer = …;// der gemeinsam genutzte Circular Buffer
2      while (true) {
3             if (!buffer.isEmpty()) {
4                    System.out.println(buffer.get());
5             }
6             Thread.Sleep(100);
7      }
```

   Der Test auf Zeile 3 prüft, ob der Ringbuffer Elmente enthält. Nur wenn er Elemente enthält, soll auf Zeile 4 mit der Methode get ein Wert rausgenommen werden.

   Trotz synchronizisierten Methoden der Klasse CircularBuffer kann noch folgendes Szenario eintreten:

Angenommen, im Ringbuffer steckt genau 1 Element. Reader-Thread 1 wird zwischen Zeile 3 und 4 unterbrochen.

Reader-Thread 2 prüft auf Zeile 3, ob der Ringbuffer leer ist. Das Thread 1 das Element noch nicht herausgeholt hat, ist der Puffer noch nicht leer. Thread 2 läuft also weiter und nimmt auf Zeile 4 das Element raus. Danach schläft er.

Reader-Thread 2 läuft weiter und führt Zeile 4 aus. Der Ringbuffer ist jetzt leer, als wird eine IllegalStateException geniert.

Es gibt also einen weiteren kritischen Abschnitt, nämlich die Zeilen 3 und 4. Lösung: eine zusätzliche Synchronisierung:

```
1       CircularBuffer buffer = …; // der gemeinsam genutzte Circular Buffer
2       while (true) {
                synchronized(buffer) {                                  //
3                       if (!buffer.isEmpty()) {
4                               System.out.println(buffer.get());
5                       }
                }                                                       //
6               Thread.Sleep(100);
7       }
```

Siehe auch Teilaufgabe 3. Die entsprechende Synchronisation ist in der Klasse CircularBuffer gemacht.

**3. Producer/Consumer-Problem.**

a) Der Producer überprüft nicht, ob der Ringbuffer voll ist. Korrekt wäre:

```
while (true) {
        if (!data.isFull()) {
                data.put(value);
                value++;
        }
        if (sleepTime >= 0) {
```

b) (siehe separater Code)