Prof. Dr. Th. Letschert

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

NVP – Nebenläufige und Verteilte Programme

Aufgabenblatt 6

Aufgabe 1

1. Was ist der Unterschied zwischen einer synchronisierten und einer nebenläufigen Kollektion?
2. Was ist eine nebenläufige Kollektion mit Benachrichtigung?
3. Stellt die Java-API eine blockierende Warteschlange mit beschränkter Kapazität zur Verfügung?
4. Wann setzt man besser explizite Locks und explizite Bedingungsvariablen ein statt die impliziten Locks und Bedingungsvariablen zu verwenden? – Erläutern Sie allgemein und am Beispiel eines Synchronisationsproblems.
5. Warum werden Bedingungsvariablen (*java.util.concurrent.locks Interface Condition*) mit einer nicht–statischen Factory–Methode von *java.util.concurrent.locks.Lock* erzeugt? Warum gibt es nicht eine *Condition*–Klasse mit Konstruktor oder wenigsten eine statische Factory–Methode?

Aufgabe 2

Was halten Sie von folgender Lösung des Produzenten–Konsumenten Problems:

```
package blatt_03.aufgabe_02.version_01

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

trait MyQueue[T] {
  def put(item: T): Unit
  def get: T
}

class Producer(q: MyQueue[Int]) extends Thread {
  var i = 0;

  override def run(): Unit = {
    while (true) {
      val item = produce
      q.put(item)
    }
  }

  private def produce: Int = {
    Thread.sleep((Math.random()*1000).asInstanceOf[Long])
    i = i+1
    i
  }

}
```

```
class Consumer(q: MyQueue[Int]) extends Thread {

 override def run(): Unit = {
   while (true) {
     val item = q.get
     consume(item);
   }
  }

  private def consume(i: Int): Unit = {
    println(i)
    Thread.sleep((Math.random()*1000).asInstanceOf[Long])
  }

}

object Aufgabe_2_Main extends App {

  object queue extends MyQueue[Int] {
    val l = Collections.synchronizedList(new LinkedList[Int]());
    override def put(item: Int): Unit = l.add(item)
    override def get: Int = l.remove(0)
  };

  new Producer(queue).start();
  new Producer(queue).start();
  new Consumer(queue).start();

}
```

Ist das OK? Kann das so funktionieren?

Was ist von folgender Verbesserung zu halten:

```
   ...
object Aufgabe_2_Main extends App {

  object queue extends MyQueue[Int] {
    val l = Collections.synchronizedList(new LinkedList[Int]());

    override def put(item: Int): Unit = {
      while (l.size() <10) wait()
      l.add(item)
      notify()
    }
    override def get: Int = {
      while (l.size() == 0) wait()
      val res = l.remove(0)
      notify()
      res
    }
  };

  new Producer(queue).start();
  new Producer(queue).start();
  new Consumer(queue).start();

}
```

Korrigieren Sie.

Durch welchen Helfer aus JUC sollte die selbst gestrickte `MyQueue`–Implementierung ersetzt werden?

Aufgabe 3

Eine Basis–Implementierung für einen zyklischen Puffer ist:

```
trait BoundedBuf[T] {
  def put(v: T): Unit
  def get: T
}

abstract class CyclicBufferBase[T](capacity: Int)(implicit val mt: Manifest[T]) extends
    BoundedBuf[T] {
  private val buf : Array[T] = new Array[T](capacity)
  private var tail = 0
  private var head = 0
  private var count = 0

  protected final def putImpl(v: T): Unit = {
    buf(tail) = v;
    tail = tail + 1
    if (tail == capacity) {
      tail = 0;
    }
    count = count+1
  }


  protected final def getImpl: T = {
    val v = buf(head)
    head = head+1
    if (head == capacity) {
      head = 0
    }
    count = count -1
    v
  }

  def full: Boolean = count == buf.length

  def empty: Boolean = count == 0
}
```

Daraus können Puffer konstruiert werden, die auf unterschiedliche Art gegenseitigen Ausschluss und Bedingungssynchronisation realisieren.

1. Konstruieren Sie als Ableitung `CyclicBufferBase` einen blockierenden Puffer `StandardBuffer` der mit impliziten Bedingungsvariablen und impliziten Mutexen arbeitet.
2. Konstruieren Sie als Ableitung einen Puffer mit Benachrichtigung `LockCondBuffer` der expliziten Bedingungsvariablen und Mutexe verwendet.
3. Konstruieren Sie als Ableitung einen Puffer mit Benachrichtigung `SemaphoreBuffer` der `java.util.concurrent Class Semaphore` verwendet.
4. Konstruieren Sie einen Puffer `SleepyBuffer`, der mit *Polling* versucht durch Warten in einen geeigneten Zustand zu kommen: Ist der Puffer voll oder leer, dann wird ein wenig geschlafen und es noch einmal versucht. Achten Sie darauf, dass *sleep* keinen Mutex freigibt.

Testen Sie beispielsweise etwa wie folgt:

```
class StandardBuffer[T](capacity: Int)(implicit val mtt: Manifest[T]) extends
    CyclicBufferBase[T](capacity)(mtt) {
  ???
}
class LockCondBuffer[T](capacity: Int)(implicit val mtt: Manifest[T]) extends
    CyclicBufferBase[T](capacity)(mtt) {
  ???
}
class SemaphoreBuffer[T](capacity: Int)(implicit val mtt: Manifest[T]) extends
    CyclicBufferBase[T](capacity)(mtt) {
  ???
}

class SleepyBuffer[T](capacity: Int)(implicit val mtt: Manifest[T]) extends
    CyclicBufferBase[T](capacity)(mtt) {
  ???
}


object Aufgabe_3_Main extends App {
   val buffer1 = new StandardBuffer[Int](3);
   val buffer2 = new LockCondBuffer[Int](3);
   val buffer3 = new SemaphoreBuffer[Int](3);
   val buffer4 = new SleepyBuffer[Int](3);

   val buffer = // der Puffer der getestet werden soll ...

   def thread(runCode: => Unit) : Thread =
     new Thread(new Runnable{
       override def run() : Unit = runCode
     }
   )

   val p1 = thread({
     var i = 0
     while (true) {
       buffer.put(i)
       i = i+1
     }
   })


   val p2 = thread({
     var i = 10000
     while (true) {
       buffer.put(i)
       i = i-1
     }
   })

   val c = thread({
     while (true) {
       println(" got: " + buffer.get)
     }
   })

   p1.start()
   p2.start()
   c.start()

}
```