EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

PROGRAMMING PARADIGMS
CONCURRENT PROGRAMMING

code: **201400537**
date: **4 July 2018**
time: **13.45–16.45**

- This is an 'open book' examination. You are allowed to have a copy of *Java Concurrency in Practice*, an (unannotated) copy of the course manual, a copy of the (unannotated) lecture (hoorcollege) *slides*, and print outs of the following additional material:
  - A gentle introduction to OpenCL - DrDobbs, by M. Scarpino.
  - B. Chapman, G. Jost and R. van der Pas. Using OpenMP - The Book.
  - Simon Peyton Jones and Satnam Singh. A Tutorial on Parallel and Concurrent Programming in Haskell. Advanced Functional Progamming Summer School.
  - The Rust Programming Language, second edition.

  You are *not* allowed to take personal notes, solutions to the exercises, and (answers to) previous examinations with you.
- You can earn 100 points with the following **5 questions**. The final grade is computed as the number of points, divided by 10. The bonus points that were obtained by participating in the quiz during the tutorial sessions and the first lecture will be added to the final result.

GOOD LUCK!

Question 1 (30 points)

a. (*11 pnts.*) Suppose you have an existing reentrant lock implementation, providing methods `lock` and `unlock`. Use this lock implementation to implement a (reentrant) read-write-lock. Your implementation should provide the methods `acquireReadLock`, `acquireWriteLock`, `releaseReadLock`, and `releaseWriteLock`. Remember to indicate which fields are protected by which lock. You may assume that methods will be called according to the lock protocol, i.e., a release method will only be called if the lock is actually held.

b. (*6 pnts.*) Sometimes read-write-lock implementations provide an *upgrade* feature: a thread holding a read lock, can upgrade its lock to a write lock. Discuss one advantage, and one disadvantage of such a feature.

c. (*5 pnts.*) The use of read-write-locks could lead to starvation. Discuss why, and suggest (in words) a change to your implementation that would reduce this risk.

d. (*8 pnts.*) Provide a lock-free implementation of a read-write-lock.

Question 2 (30 points)

In this exercise, we will study the reference implementation for the `LinkedBlockingQueue` as available in Java. The class starts with the following declarations and constructor.

```
    static class Node<E> {
  /** The item, volatile to ensure barrier separating write and read */
      volatile E item;
      Node<E> next;
      Node(E x) { item = x; }
  }

 /** The capacity bound, or Integer.MAX_VALUE if none */
  private final int capacity;

  /** Current number of elements */
  private final AtomicInteger count = new AtomicInteger(0);

  /** Head of linked list */
  private transient Node<E> head;

  /** Tail of linked list */
  private transient Node<E> last;

  /** Lock held by take, poll, etc */
  private final ReentrantLock takeLock = new ReentrantLock();

  /** Wait queue for waiting takes */
  private final Condition notEmpty = takeLock.newCondition();

  /** Lock held by put, offer, etc */
 private final ReentrantLock putLock = new ReentrantLock();

  /** Wait queue for waiting puts */
  private final Condition notFull = putLock.newCondition();

 public LinkedBlockingQueue(int capacity) {
      if (capacity <= 0) throw new IllegalArgumentException();
      this.capacity = capacity;
      last = head = new Node<E>(null);
 }
```

a. (*2 pnts.*) Explain why the locks `putLock` and `takeLock` are declared `final`.

b. (*13 pnts.*) Figure 1 contains the code for the methods `put` and `take` (with helper methods). Discuss what these methods do, and why they do this correctly. In particular, discuss the following:
   - How do these methods guarantee that concurrent execution of `put` and `take` is possible?
   - Which fields are protected by which locks?
   - For all fields that are not protected by a lock, why it is okay that they are not protected by a lock?
   - Why is it okay to use count in the wait guard, even though it is not protected by a lock?

c. (*5 pnts.*) Provide an implementation for the method `signalNotEmpty`, which signals that there is an element available in the queue.

```
private void signalNotEmpty() {
      // body
  }
```

Question continued on page 4!

```
private void insert(E x) {
    last = last.next = new Node<E>(x);
}

private E extract() {
    Node<E> first = head.next;
    head = first;
    E x = first.item;
    first.item = null;
    return x;
}

public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // Note: convention in all put/take/etc is to preset
    // local var holding count  negative to indicate failure unless set.
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        try {
            while (count.get() == capacity)
                notFull.await();
        } catch (InterruptedException ie) {
            notFull.signal(); // propagate to a non-interrupted thread
            throw ie;
        }
        insert(e);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally { putLock.unlock(); }
    if (c == 0)
        signalNotEmpty();
}

public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        try {
            while (count.get() == 0)
                notEmpty.await();
        } catch (InterruptedException ie) {
            notEmpty.signal(); // propagate to a non-interrupted thread
            throw ie;
        }
         x = extract();
        c = count.getAndDecrement();
        if (c > 1)
           notEmpty.signal();
    } finally { takeLock.unlock(); }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

Figure 1: Methods `put` and `take` as provided in reference implementation `java.util.concurrent`

d. (*5 pnts.*) Provide an implementation for the method `toArray`, which returns an array with all elements currently stored in the queue.

```
public Object[] toArray() {
    // body
}
```

e. (*5 pnts.*) Give a realistic example how careless use of the two locks could lead to a deadlock.

Question 3 (10 points)

a. (*6 pnts.*) Consider the following parallel *quicksort* algorithm, implemented in Haskell:

```
force :: [a] -> ()
force xs = go xs `pseq` () where
    go []     = 1
    go (_:xs) = go xs

par_qsort :: (Ord a) => [a] -> [a]
par_qsort []     = []
par_qsort (x:xs) = (force h) `par` ((force l) `par` (l ++ x : h)) where
    l = par_qsort [y | y <- xs, y < x]
    h = par_qsort [y | y <- xs, y >= x]
```

Implement a parallel version of *quicksort* in Haskell using *fork/join*-parallelism. Your implementation should have the following type signature: `fj_qsort :: Ord a => [a] -> IO [a]`, and forks a new concurrent thread for every recursive call.

b. (*4 pnts.*) Explain how the performance and scalability of `fj_qsort` differs from `par_qsort`. Which one is more efficient, and why?

Question 4 (20 points)

Consider the following class `Loop` and its method `shiftLeft`.

```
public class Loop {

    private int[] a = new int[1000000];

    public void shiftLeft () {
        for (int i = 1; i < a.length; i++) {
            int tmp = a[i];
            a[i - 1] = tmp;
        }
    }
}
```

a. *(3 pnts.)* Discuss which dependencies there are between the iterations of the loop (if any). Motivate your answer.

b. *(3 pnts.)* Discuss what sort of OpenMP-like annotations could be used to parallellise the execution of this program.

c. *(5 pnts.)* Write a GPU kernel that has the same behaviour as the method `shiftLeft`.

Now consider the class `Count` and its method `countZeroes`.

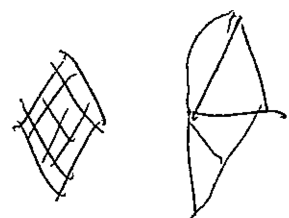
```
public class Count {

    private int[] a = new int[100000];

    public int countZeroes() {
        int c = 0;
        for (int i = 0; i < a.length; i++) {
            c = c + (a[i] == 0? 1 : 0);
        }
        return c;
    }
}
```

d. *(4 pnts.)* Discuss how this method could be parallellised by using OpenMP-like annotations.

e. *(5 pnts.)* Write an OpenCL kernel that has the same behaviour as the method `countZeroes`.



Question 5 (10 points)

a. *(4 pnts.)* Consider the following 2 Rust programs.

Program 1:

```
use std::thread;

fn main () {
    let x = vec![5, 8, 10];
    let t = thread::spawn (move || {
        let y = x;
        return y;
    });
    let y = t.join().unwrap();
    println!("{}", x[0] + y[0]);
}
```

Program 2:

```
use std::thread;

fn main () {
    let x = 5;
    let t = thread::spawn (move || {
        let y = x;
        return y;
    });
    let y = t.join().unwrap();
    println!("{}", x + y);
}
```

One of these two programs will not compile, the other one will. Explain which program will not compile, and why, and for the other program, explain its effect.

b. *(6 pnts.)* Use the message passing mechanism of Rust to implement a distributed counter: the main thread continuously sends messages, while in a separate thread, the receiver counts how many times it has received a message.