TEST
CONCURRENT PROGRAMMING

code: **201400537**
date: **1 July 2015**
time: **13.45–16.45**

- This is an 'open book' examination. You are allowed to have a copy of Java Concurrency in Practice, and a copy of the (unannotated) lecture slides and exercises. You are also allowed to bring the material related to Block 7 (the parallel Haskell tutorial, and an OpenCL tutorial). You are *not* allowed to take personal notes and (answers to) previous examinations with you.
- You can earn 100 points with the following 7 questions. The final grade is computed as the number of points, divided by 10.
  Students in the **Programming Paradigms** module need to obtain at least a 5.0 for the test.
  Students for the **Concurrent & Distributed Programming** course also need to obtain at least a 5.0 for the test. This test result is 80 % of your final grade (the other 20 % is given by the distributed programming homework).
  Students that attended at least 6 out of 7 exercise sessions obtain a 1.0 bonus.

Question 1 (15 points)

Some single producer, single consumer applications use the following approach to handle buffered data:

- Two bounded buffers `inbuf` and `outbuf` are allocated (with the same size).
- A producer continuously produces a new data element, and adds this to `inbuf`, until `inbuf` is full.
- A consumer continuously takes an element from `outbuf` and processes this, until `outbuf` is empty.
- When `inbuf` is full, and `outbuf` is empty, the two buffers are swapped.

a. (*4 pnts.*) What is the main advantage of using separate input and output buffers?

b. (*8 pnts.* If the producer and consumer work at irregular speed, they might end up being blocked for a long time when they want to swap the buffers. A solution for this is to use a buffer pool, so that full buffers do not have to be immediately processed. Sketch how such a buffer pool would work, and where synchronisation is needed. You solution should ensure that the producer and the consumer *never* directly synchronise with each other.

c. (*3 pnts.*) Discuss whether there is a risk of memory overflow for your solution. Why or why not?

Question 2 (20 points)

Consider an application that implements a simple hospital emergency management system. There are several classes used in this system: `Hospital`, `Ambulance`, `Patient`, `AmbulanceDatabase`, and `PatientDatabase`. As problems in this application can be life-threatening, the application developers have also inserted some specifications, with the idea that later they should be able to verify their implementation w.r.t. this specification.

Some implementation details are left out, as they are irrelevant for this assignment.

This application (both implementation and specification) contains several concurrency problems. Discuss *five different* problems, and explain *why* they are a problem, for example by discussing an execution that illustrates the problem.

```java
class Hospital {

    private String name;
    private PatientDatabase patients;
    private AmbulanceDatabase ambulances;

    public PatientDatabase getPatientsDB() {
        return patients;
    }

    public AmbulanceDatabase getAmbulanceService() {
        return ambulances;
    }
}
```

```java
class Ambulance {

    private int number;
    private Patient patient;

    public Ambulance(int n, Patient p) {
        number = n;
        patient = p;
    }

    public int getNumber() { return number; }

    public Patient getPatient() { return patient; }
}
```

```java
import java.util.*;
class Patient {

    private int insuranceId;
    private Date admitted;
    // other relevant patient info

    public Patient(int id, Date admitted) {
        this.insuranceId = id;
        this.admitted = admitted;
    }

    public void newVisit(Date newD) { admitted = newD; }

    public int getInsurance() { return insuranceId; }
}
```

```java
import java.util.*;
class AmbulanceDatabase {

    private Hospital h;
    //@ private invariant h.getAmbulanceService() == this;

    /*@ private invariant
          (\forall int i, Ambulance a;
             free.contains(i) && occupied.contains(a); a.getNumber() != i);
    */
    private List<Integer> free = new LinkedList<Integer>();
    private List<Ambulance> occupied = new LinkedList<Ambulance>();

    public synchronized void assignAmbulance(Patient p) {
        if (free.isEmpty()) {
            try {
                wait();
            }
            catch (InterruptedException e) {
                // do something appropriate
            }
        }
        int i = free.remove(0);
        occupied.add(new Ambulance(i, p));
    }

    //@ ensures free.contains(a.getNumber());
    public synchronized void freeAmbulance(Ambulance a) {
        occupied.remove(a);
        free.add(a.getNumber());
        notify();
    }

    public synchronized void announcementToAmbulanceDrivers() {
        if (!occupied.isEmpty()) {
            try {
                wait();
            }
            catch (InterruptedException e) {
                // do something appropriate
            }
        }
        // make an announcement to all ambulance drivers, as they are all waiting in the hospital
    }

    public void waitingAmbulances() {
        for (Integer i : free) {
            System.out.println("Ambulance␣" + i);
        }
    }

    public synchronized void emergencyPickUp(Ambulance a) {
        h.getPatientsDB().admitted(a.getPatient().getInsurance());
    }
}
```

```
import java.util.*;
import java.util.concurrent.*;
class PatientDatabase {

    private Map<Integer, Patient> patients = new ConcurrentHashMap<Integer, Patient>();
    private Hospital h; // all objects initialised to belong to the same hospital

    // private invariant h.getPatientsDB == this;

    public synchronized void admitted(int id) {
        Patient p = patients.get(id);
        Date today = new Date(); // Date constructor initialised date with current date
        if (p == null) {
            p = new Patient(id, today);
        } else {
            p.newVisit(today);
        }
        patients.put(id, p);
    }

    public synchronized void emergencyAdmittance(int id) {
        admitted(id);
        h.getAmbulanceService().assignAmbulance(patients.get(id));
    }
}
```

Question 3 (15 points)

a. (*10 pnts.*) Consider the class `LockFreeExchanger` (from *The Art of Multiprocessor Programming*) on the next page. It is a lock-free implementation of an exchanger, allowing to threads to exchange information. The exchanger's main loop continues until the `timeout` limit passes, and then throws an exception. Explain the behaviour of a thread in the meantime, i.e., explain what happens in the three different cases.

b. (*5 pnts.*) Explain how to adapt the program for a three-way exchanger, *i.e.*, where Thread 1 passes a value to Thread 2, Thread 2 passes a value to Thread 3, and Thread 3 passes a value to Thread 1.

```java
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicStampedReference;
import java.util.concurrent.TimeoutException;

public class LockFreeExchanger<T> {

    static final int EMPTY = 0, WAITING = 1, BUSY = 2;
    private AtomicStampedReference<T> slot;

    public LockFreeExchanger() {
        this.slot = new AtomicStampedReference<T>(null, 0);
    }

    public T exchange(T myItem, long timeout, TimeUnit unit)
        throws TimeoutException {

        long nanos = unit.toNanos(timeout);
        long timeBound = System.nanoTime() + nanos;
        int[] stampHolder = {EMPTY};
        while (true) {
            if (System.nanoTime() > timeBound) throw new TimeoutException();

            T yrItem = slot.get(stampHolder);
            int stamp = stampHolder[0];

            switch(stamp) {

            case EMPTY:

                if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
                    while (System.nanoTime() < timeBound) {
                        yrItem = slot.get(stampHolder);
                        if (stampHolder[0] == BUSY) {
                            slot.set(null, EMPTY);
                            return yrItem;

                        }
                    }

                    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
                        throw new TimeoutException();
                    } else {
                        yrItem = slot.get(stampHolder);
                        slot.set(null, EMPTY);
                        return yrItem;
                    }
                }

                break;

            case WAITING:
                if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
                    return yrItem;
                break;

            case BUSY:
                break;
            default:
                break;
            }
        }

    }
}
```

Question 4 (10 points)

a. (*3 pnts.*) Explain whether it is possible to have `r1 = r2 = 8` at the end of an execution. Motivate your answer.

```
initially x = y = 6
r1 := x          | r2 := y
y := r1          | x := r2
```

b. (*3 pnts.*) Explain what are the possible values of `r1` at the end of the execution. Motivate your answer.

```
initially answer = 21, ready = false, x = 0
answer = 42          | r1 = x
ready = true         | if (ready) then
                     |     r1 = answer
```

c. (*2 pnts.*) What changes if you make `answer` volatile? Motivate your answer.

d. (*2 pnts.*) What changes if you make `ready` volatile? Motivate your answer.

Question 5 (15 points)

a. (*4 pnts.*) OpenCL also contains several atomic instructions. The following kernel uses an `atomic_add(x, y)` instruction, which *atomically* adds the value of `y` to `x`.

```
__kernel void gpadd(__local int *x, __local int *Values) {
  uint tid = get_local_id(0);

  atomic_add(x, Values[tid]);

}
```

Explain what is the intended behaviour of this kernel.

b. (*4 pnts.*) Write a kernel that gets two arrays `a` and `b` as input elements, and then does the following:

- first it doubles all elements in array `a`; and
- then it adds the new value of `a[i]` to `b[i]`.

c. (*7 pnts.*) Implement a synchronous exchanger using `MVars`. That is, a thread wishing to exchange its value blocks until there is another thread that wishes to exchange a value. Once the other thread is there, the two values are exchanged, and the two threads proceed their own execution. There is no need to worry about time outs. You can use two mutable variables for the exchange.

Implement Haskell functions `swapMVar1` and `swapMVar2` that implement this exchange. A `main` method using this exchanger could look as follows.

```
main = do
    a <- newEmptyMVar
    b <- newEmptyMVar
    forkIO $ do
        r <- swapMVar1 a 1 b
        putStrLn $ "Thread␣1:␣" ++ (show r)
    forkIO $ do
        r <- swapMVar2 a 2 b
        putStrLn $ "Thread␣2:␣" ++ (show r)
    threadDelay 1000
```

Question 6 (15 points)

For performance reasons, in linked list implementations, sometimes each node is protected by its own lock, instead of using a single lock to protect the complete list.

Consider the following (stripped-down) implementation of such a list. It defines a class `LockNode`, where the node holds a lock. Further, in class `LockCouplingList`, the `insert(e, pos)` method, adds a new node containing item `e` at position `pos` in the list.

```
1  class LockNode<E> {
2      private Lock nodeLock;
3      E item;
4      LockNode<E> next;
5
6      public LockNode(E val, LockNode<E> nxt) {
7          item = val;
8          next = nxt;
9          nodeLock = new ReentrantLock();
10     }
11
12
13     public void lock() {
14         nodeLock.lock();
15     }
16
17     public void unlock() {
18         nodeLock.unlock();
19     }
20
21 }
22
23 public class LockCouplingList<E> {
24     private LockNode<E> head = new LockNode<E>(null, null);
25
26     public void insert(int pos, E val) {
27         LockNode<E> current = head;
28         LockNode<E> prev;
29         LockNode<E> newNode;
30         current.lock();
31         while (current.next != null && pos > 0) {
32             prev = current;
33             current = current.next;
34             pos--;
35             current.lock();
36             prev.unlock();
37         }
38         newNode = new LockNode<E>(val, current.next);
39         current.next = newNode;
40         current.unlock();
41     }
42 }
```

a. *(6 pnts.)* Discuss one major advantage of this implementation, and one major disadvantage.

b. *(3 pnts.)* Which fields are protected by the lock `nodeLock`?

c. *(6 pnts.)* Suppose you swap lines 35 and 36 in the method `insert`, i.e., the lock acquire and the lock release. Would this change the behaviour of the program. Explain why, or why not?

Question 7 (10 points)

A `CopyOnWriteArrayList` is a thread-safe variant of `ArrayList` in which all operations that update the list are implemented by making a fresh copy of the underlying array, while lookup operations are done over a snapshot of the state of the array, at the moment the loopup operation started. This may be efficient when read operations vastly outnumber changes, because read operations do not have to be synchronised.

Below is a skeleton of a class implementing such a `CopyOnWriteArrayList`. Note that the synchronized `getArray` method returns you a copy of the current list.

```
class CopyOnWriteArrayList<E> {
  protected Object[] array = new Object[0];

  protected synchronized Object[] getArray() { return array; }

  public void add(E element) {
    // to do
  }

  public Iterator<E> iterator() {
    return new Iterator<E>() {
      // iterator fields to be declared

        public boolean hasNext() {
          // to implement
        }

        public E next() {
          // to implement
        }

    };
  }
}
```

a. (*5 pnts.*) Provide an implementation for method `add`, which updates the array. *Hint*: a call `System.arraycopy(arrayA, 0` copies *all* the elements from `arrayA` to `arrayB`, provided `arrayB` is long enough.

b. (*5 pnts.*) Fill in the missing parts for the `iterator` method, which works on a snapshot of the array.

, arrayB, 0, arrayA.length)