EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

PROGRAMMING PARADIGMS
CONCURRENT PROGRAMMING

code: **201400537**
date: **17 June 2016**
time: **13.45–16.45**

- This is an 'open book' examination. You are allowed to have a copy of *Java Concurrency in Practice*, the two additional papers on 'Parallel Haskell' and OpenCL which are available from Blackboard, and a copy of the (unannotated) lecture (hoorcollege) *slides*. You are *not* allowed to take the slides of the exercises sessions with you. Furthermore, you are *not* allowed to take personal notes and (answers to) previous examinations with you.

- You can earn 100 points with the following 8 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 that attended at least 6 out of 7 exercise sessions obtain a 1.0 bonus.

VEEL SUCCES!

Question 1 (10 points)

In this question you are asked to determine the possible outcomes of a simple, concurrent Java program. Apart from normal output, the following outcomes might be possible:

- *compile-error*: the program is not a correct Java program;
- *runtime-error*: an Exception will be thrown;
- *no output*: e.g., due to a deadlock or livelock.

You do not have to motivate your answers.
The questions after a. are variations on the original Java program; if you think that the variation does not change the outcome of the original program, you can answer "same as a.".

Consider the following Java-program Grow.

```java
public class Grow {
    private String s = "#";

    public String    getString()        { return s;       }
    public void       add(String sfix)   { s = s + sfix; }

    static final Grow g = new Grow();

    public static void main(String[] args) {
        Thread ta = new Thread() { public void run() { g.add("a"); }; };
        Thread tb = new Thread() { public void run() { g.add("b"); }; };
        ta.start(); tb.start();

        try { ta.join(); tb.join(); }
        catch (InterruptedException e) {}

        System.out.println(g.getString());
    }
}
```

a. (*2 pnts.*) What are the possible outcomes of program `Grow`?

b. (*1 pnt.*) In the original program, we remove the `try-catch`-block (i.e., two lines) from the method `main`. What are now the possible outcomes of the program?

c. (*1 pnt.*) In the original program, we remove the `try-catch`-block (i.e., two lines) from the method `main`. Furthermore, the calls `ta.start()` and `tb.start()` are replaced by `ta.run()` and `tb.run()`, respectively. What are now the possible outcomes of the program?

d. (*1 pnt.*) In the original program, the methods `run` of both `Threads` `ta` and `tb` are defined as `synchronized` methods. What are now the possible outcomes of the program?

e. (*1 pnt.*) In the original program, the method `getString` is defined as a `synchronized` method. What are now the possible outcomes of the program?

f. (*1 pnt.*) In the original program, the method `add` is defined as a `synchronized` method. What are now the possible outcomes of the program?

g. (*1 pnt.*) In the original program, the method `add` is changed to:

```
public void add(String sfix) {
    s = s + sfix;
    notifyAll();
}
```

What are now the possible outcomes of the program?

h. (*2 pnts.*) In the original program, the method `add` is changed to:

```
public synchronized void add(String sfix) {
    if (s.equals("#") && sfix.equals("a"))
        try { wait(); }
        catch (InterruptedException e) {}
    notify();
    s = s + sfix;
}
```

What are now the possible outcomes of the program?

Question 2 (10 points)

Indicate for each of the statements below whether the statement is *true* or *false*. You do not have to motivate your answers. The total number of points for this question is 10 points. For each *wrong* answer, 2 points are subtracted. For each *open* answer – for which you do not specify either true or false – 1 point is subtracted.

a. Dekker's algorithm is a lock-free algorithm for mutual exclusion.

b. The happens-before relation is a partial order. Thus not all pairs of statements in a concurrent Java program might be related by the happens-before relation.

c. When you call the method `interrupt` of a `Thread` object, the method that is being executed in the `Thread` will be interrupted and the method will throw an `InterruptedException`.

d. An object of the class `Timer` uses a single thread to execute its `TimerTask` objects.

e. Locks that are mostly uncontended can be found with thread dumps.

f. If a class invariant involves two variables, at least two locks are needed to protect such a class invariant.

g. Most implementations of `ExecutorService` support work-stealing: threads in the thread pool attempt to find and execute subtasks created by other active tasks.

h. An explicit lock (e.g., `ReentrantLock`) which uses an unfair locking policy usually outperforms an explicit lock which uses a fair locking policy.

i. A `Condition` object provides the methods `await` and `signal`. You cannot call `wait` or `notify` on `Condition` objects.

j. Test code can introduce timing or synchronization artifacts that can mask bugs that might otherwise manifest themselves.

Question 3 (10 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).
- The producer continuously produces a new data element, and adds this to `inbuf`, until `inbuf` is full.
- The 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. (*2 pnts.*) Which class of Java's synchronizer classes could be used to swap the two buffers?

b. (*3 pnts.*) What is the main advantage of using separate input and output buffers?

c. (*3 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. Your solution should ensure that the producer and the consumer *never* directly synchronise with each other.

d. (*2 pnts*) Can a buffer pool implementation be used when there are multiple producers and/or consumers?

Question 4 (20 points)

Consider an application that implements a simple management system for a *pizzza delivery* company. There are several classes used in this system: `PizzaCompany`, `Order`, `PizzaBoy`, `PizzaBoyDatabase`, `Client`, and `ClientDatabase`. The application developers have also inserted some specifications (i.e., invariants and postconditions), with the idea that later they might be able to verify their implementation w.r.t. these specifications.

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

Note that several implementation details have been left out, as they are irrelevant for this question. Furthermore, several design and implementations decisions are questionable, but are also irrelevant for this question; we are only interested in the *concurrency* problems.

```
1  class PizzaCompany {
2      private String            name;
3      private PizzaBoyDatabase   boys;
4      private ClientDatabase     clients;
5
6      public PizzaBoyDatabase   getPizzaBoysDB()   { return boys;      }
7      public ClientDatabase     getClientsDB()     { return clients;   }
8  }
```

```
1  class Order {
2      private int      id;
3      private String   description;
4
```

```
5    public Order(int id, String description) {
6        this.id          = id;
7        this.description = description;
8    }
9  }
```

```
1  class PizzaBoy {
2      private String  name;
3      private Order   order;
4
5      public PizzaBoy(String name, Order order) {
6          this.name  = name;
7          this.order = order;
8      }
9
10     public String   getName()   { return name;  }
11     public Order    getOrder()  { return order; }
12 }
```

```
1  class PizzaBoyDatabase {
2      private PizzaCompany    company;
3      private List<String>    free     = new LinkedList<String>();
4      private List<PizzaBoy>  occupied = new LinkedList<PizzaBoy>();
5
6      // private invariant company.getPizzaBoysDB() == this;
7
8      /*@ private invariant
9          (\forall String name, Pizzaboy boy;
10             free.contains(name) && occupied.contains(boy); boy.getName() != name);
11     */
12
13     public synchronized void assignOrder(Order order) {
14         if (free.isEmpty()) {
15             try { wait(); }
16             catch (InterruptedException e) { /* do something useful */ }
17         }
18         String name = free.remove(0);
19         occupied.add(new PizzaBoy(name, order));
20     }
21
22     //@ ensures free.contains(boy.getNumber());
23     public synchronized void orderDelivered(PizzaBoy boy) {
24         occupied.remove(boy);
25         free.add(boy.getName());
26         notify();
27     }
28
29     public synchronized void giveInstructionsToAllBoys() {
30         if (! occupied.isEmpty()) {
31             try { wait(); }
32             catch (InterruptedException e) { /* do something useful */ }
33         }
34         // Give instructions to all pizza boys, now that none of them is occupied.
35     }
36
37     public void printWaitingPizzaBoys() {
38         System.out.println("Boys waiting for an order:");
39         for (String name : free)
40             System.out.println(name);
```

```
41        }
42
43        // A client can have a favorite Pizza Boy.
44        public synchronized void assignFavorite(PizzaBoy boy, String client_name) {
45            Client c = company.getClientsDB().getClient(client_name);
46            if (c != null)
47                c.setFavorite(boy);
48        }
49   }
```

```
1    class Client {
2        private String     name;
3        private String     address;
4        private PizzaBoy   favorite_boy;
5        private List<Order> previous_orders = new LinkedList<Order>();
6
7        public Client(String name, String address) {
8            this.name         = name;
9            this.address      = address;
10           this.favorite_boy = null;
11       }
12
13       public String  getName()    { return name;    }
14       public String  getAddress() { return address; }
15
16       public synchronized void addOrder(Order order) {
17           previous_orders.add(order);
18       }
19
20       public void setFavorite(PizzaBoy boy) { favorite_boy = boy; }
21   }
```

```
1    class ClientDatabase {
2        private PizzaCompany company;
3        private Map<String, Client> clients = new HashMap<String, Client>();
4
5        // private invariant company.getClientsDB() == this;
6
7        public synchronized void addClient(String name, String address) {
8            if (getClient(name) == null)
9                clients.put(name, new Client(name, address));
10       }
11
12       public synchronized Client getClient(String name) {
13           return clients.get(name);
14       }
15
16       public synchronized void placeOrder(Client client, Order order) {
17           client.addOrder(order);
18           company.getPizzaBoysDB().assignOrder(order);
19       }
20   }
```

Question 5 (10 points)

A CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

A `CountDownLatch` is initialized with a given `count`. The `await` methods block until the current count reaches zero due to invocations of the `countDown` method, after which all waiting threads are released and any subsequent invocations of `await` return immediately. This is a one-shot phenomenon – the `count` cannot be reset.

In this question you are asked to develop a `CountDownLatch` class. The implementation of your `CountDown-Latch` should use a <u>`ReentrantLock`</u> in combination with a <u>`Condition`</u> object. Your `CountDownLatch` should implement the following interface:

```
interface Latch {
    // Causes the current thread to wait until the latch has counted down
    // to zero,  unless the thread is interrupted.
    public void await() throws InterruptedException;

    // Decrements the count of the latch, releasing all waiting threads
    // if the count reaches zero.
    public void countDown();
}
```

Question 6 (15 points)

Consider the interface `BasicLock` interface which offers the methods `lock` and `unlock` of Java's `Lock` interface:

```
public interface BasicLock {
    // Acquires the lock.
    public void lock();

    // Releases the lock.
    // Throws IllegalMonitorStateException when the lock is not held.
    public void unlock() throws IllegalMonitorStateException;
}
```

In this question you are asked to create a class `ReentrantCASLock` which implements the interface `BasicLock`. Your implementation should use an <u>`AtomicLong`</u> variable and use the *compare-and-set* mechanism to ensure mutual exclusion.

Your implementation of `ReentrantCASLock` should have the following properties:

- *(7 pnts.)* The class `ReentrantCASLock` should implement the mutual exclusion property correctly using the *compare-and-set* mechanism.

- *(6 pnts.)* The class `ReentrantCASLock` should accomodate reentering: if a thread holds the lock and requests it again with `lock` this should be allowed.

- *(2 pnts.)* The method `unlock` should throw an `IllegalMonitorStateException` when a thread which calls `unlock` does not have the lock.

Solutions that do *not* use the *compare-and-set* mechanism to ensure mutual exclusion will not rewarded with any points.

The following methods of the class `Thread` might be useful when implementing `ReentrantCASLock`:

```
class java.lang.Thread {
    // Returns a reference to the currently executing thread object.
    public static Thread currentThread() { ... }

    // Returns the identifier of this Thread: a positive number.
    public long getId() { ... }
    ...
}
```

Question 7 (10 points)

a. (*3 pnts.*) Explain whether it is possible to have `r1==r2==1` at the end of an execution. Motivate your answer.

initially: `x==y==0, r1==r2==1`

r1 = x;	r2 = y;
y = r1;	x = r2;

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

initially: `answer==2, ready==false, x==0`

r1 = x;	answer = 3;
if (ready) then	ready = true;
r1 = answer;	

c. (*2 pnts.*) What changes if the variable `answer` is defined `volatile`? Motivate your answer.

d. (*2 pnts.*) What changes if the variable `ready` is defined `volatile`? Motivate your answer.

Question 8 (15 points)

a. (*2 pnts.*) Consider the following algebraic data type, which models a binary tree:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Implement a recursive function `ptreesum1 :: Tree Int -> Int` that sums up all elements in the tree by using *semi-explicit* parallelism. A sequential version is given below.

```
treesum :: Tree Int -> Int
treesum (Leaf v)     = v
treesum (Node v l r) = v + (treesum l) + (treesum r)
```

Explain in detail how your implementation achieves parallelism.

b. (*7 pnts.*) Implement a second variant: `ptreesum2 :: Tree Int -> IO Int` that uses *fork/join parallelism*, where a new thread is forked for every recursive call.

c. (*6 pnts.*) Consider the OpenCL kernel given below, which performs some operations on an integer array `arr`, depending on whether `index` is even or odd:

```
1   __kernel void incorrect(__global int* arr, int size) {
2       int index = get_global_id(0);
3
4       if (index % 2 == 0) {
5           arr[index] *= 2;
6           barrier(CLK_GLOBAL_MEM_FENCE);
7           arr[index] += arr[index - 4 % size];
8       }
9       else {
10          arr[index] += arr[index + (size / 2) % size];
11          barrier(CLK_GLOBAL_MEM_FENCE);
12          arr[index] *= 3;
13      }
14
15      barrier(CLK_GLOBAL_MEM_FENCE);
16  }
```

This kernel contains bugs: a parallel execution may not give the same result as a sequential execution of this kernel. Identify and explain all bugs in this kernel. Also write a bug-free version of this kernel.