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!

ANSWERS

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`.

```
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 `Thread`s `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?

Solution to Question 1

This is a question from an 'old' Programming 2 examination, i.e., of April 2004. It tests the knowledge of the basic built-in concurrency constructs of Java, and should not cause problems for students of the 'Concurrent Programming' submodule.

a.	original program	`#ab`, `#ba`, `#a` en `#b`
b.	`try-catch` block removed	`#ab`, `#ba`, `#a`, `#b`, and `#`
c.	`run` instead of `start`	`#ab`
d.	synchronized `run` methods	`#ab`, `#ba`, `#a` and `#b` (= same as a.)
e.	synchronized `getString`	`#ab`, `#ba`, `#a` and `#b` (= same as a.)
f.	synchronized `add`	`#ab` and `#ba`
g.	`notifyAll` (no `synchronized`)	runtime-error (i.e., `IllegalMonitorStateException`)
h.	`wait` and `notify`	`#ba`

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.

Solution to Question 2

a.	`true`	Dekker's algorithm does not use a lock.
b.	`true`	That is the idea of a partial order.
c.	`false`	When non-blocked, the thread may just ignore the interrupt.
d.	`true`	As observed in exercise session 1.
e.	`false`	Contended locks show up in thread dumps.
f.	`false`	A single lock is needed to protect both variables.
g.	`false`	Only `ForkJoinPool` supports work-stealing.
h.	`true`	Unfair locking permits barging for uncontended locking.
i.	`false`	Any class defines the methods `wait` and `notify`.
j.	`true`	Such bugs are called 'Heisenbugs'.

Mostly uncontended locks rarely show up in thread dumps, so thread dumps are not the means to find these locks.

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?

Solution to Question 3

a. (*2 pnts.*) Java's `Exchanger` class is ideal for synchronizing two threads and exchanging objects.

b. (*3 pnts.*) The producer and consumer threads can continue *without synchronisation* until the buffer is full/empty, respectively. The only synchronisation point is when the buffers need to be swapped.

c. (*3 pnts.*) Use a pool of *empty* buffers and a pool of *full* buffers. When the producer has a full buffer, the producer adds this buffer to the full buffer pool, and then takes a free buffer from the empty buffer pool. When the consumer is ready to read a new buffer, the consumer takes a buffer from full buffer pool. After processing it, the now empty buffer is put back in the empty buffer pool.

d. (*2 pnts.*) Yes, a buffer pool is also a scalable solution 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  }
```

```
class PizzaBoy {
    private String  name;
    private Order   order;

    public PizzaBoy(String name, Order order) {
        this.name  = name;
        this.order = order;
    }

    public String   getName()   { return name;  }
    public Order    getOrder()  { return order; }
}
```

```
class PizzaBoyDatabase {
    private PizzaCompany    company;
    private List<String>    free     = new LinkedList<String>();
    private List<PizzaBoy>  occupied = new LinkedList<PizzaBoy>();

    // private invariant company.getPizzaBoysDB() == this;

    /*@ private invariant
       (\forall String name, Pizzaboy boy;
            free.contains(name) && occupied.contains(boy); boy.getName() != name);
    */

    public synchronized void assignOrder(Order order) {
        if (free.isEmpty()) {
            try { wait(); }
            catch (InterruptedException e) { /* do something useful */ }
        }
        String name = free.remove(0);
        occupied.add(new PizzaBoy(name, order));
    }

    //@ ensures free.contains(boy.getNumber());
    public synchronized void orderDelivered(PizzaBoy boy) {
        occupied.remove(boy);
        free.add(boy.getName());
        notify();
    }

    public synchronized void giveInstructionsToAllBoys() {
        if (! occupied.isEmpty()) {
            try { wait(); }
            catch (InterruptedException e) { /* do something useful */ }
        }
        // Give instructions to all pizza boys, now that none of them is occupied.
    }

    public void printWaitingPizzaBoys() {
        System.out.println("Boys␣waiting␣for␣an␣order:");
        for (String name : free)
            System.out.println(name);
    }

    // A client can have a favorite Pizza Boy.
    public synchronized void assignFavorite(PizzaBoy boy, String client_name) {
        Client c = company.getClientsDB().getClient(client_name);
        if (c != null)
```

should be: boy.getName();

```
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  }
```

The assignment of favorite_boy may be cashed, so it is better to declare favorite_boy as volatile variable.

This is rather useless as the application keeps creating new PizzaBoy objects (PizzaBoyDatabase, line 19). Still, this is not a concurrency error.

```
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  }
```

If the order cannot be assigned (because all PizzaBoys are occupied), the thread will start waiting (in assignOrder), and will release the lock on the PizzaBoysDatabase. The lock on this ClientDatabase is not released, though: another potential deadlock.

Solution to Question 4

There are at least five concurrency problems within this application. For each correct answer with a good motivation `4` points should be rewarded.

- The calls of `wait` on `free` and `occupied` are not done in a loop (and thus the list might have become empty, respectively non-empty again, before the waiting thread continues).
- The JML postcondition of `orderDelivered` is not correct. This condition might have been made invalid by another thread as soon as the lock is released.

- In the method `orderDelivered`, the method `notify` is called (instead of `notifyAll`). This might wake up a wrong thread (namely one waiting in `giveInstructionToAllBoys`).
- The method `printWaitingPizzaBoys` is not `synchronized`. Therefore access to `free` is *not* `synchronized`. This might cause a data race!
- Potential deadlock between `assignFavorite` (acquires lock on `PizzaBoysDatabase`, and then `ClientsDatabase`), and `placeOrder` (acquires lock on `ClientsDatabase`, and then `PizzaBoysDatabase`).

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 `CountDownLatch` should use a `ReentrantLock` in combination with a `Condition` 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();
}
```

Solution to Question 5

A possible implementation of `CountDownLatch` which uses a `ReentrantLock` is the following:

```
public class CountDownLatch implements Latch {
    private int         count;
    private boolean     open;
    private Lock        lock;
    private Condition   barrierCond;

    public CountDownLatch(int count) {
        this.count       = count;
        this.open        = (count == 0);
        this.lock        = new ReentrantLock();
        this.barrierCond = lock.newCondition();
    }

    public void await() throws InterruptedException {
        if (!open) {
            lock.lock();
            try {
                if (count > 0)
                    barrierCond.await();
            } finally {
                lock.unlock();
            }
```

(count > 0) is also correct

```
        }
    }

    public void countDown() {
        lock.lock();
        try {
            if (count > 0) {
                if (--count == 0) {
                    barrierCond.signalAll();
                    open = true;
                }
            }
        } finally {
            lock.unlock();
        }
    }
}
```

Some notes for the correction:

- No points should be rewarded for a solution which does not use a `ReentrantLock`.
- An additional layer of `synchronized` methods has been used: `-4` points.
- No `Condition` object has been used: `-4` points.
- The methods `wait` and/or `notify` are called on the `Condition` object: `-4` points.
- The lock is not unlocked in a `finally` clause: **`-2`** points.
- The latch is not left 'open' after the counter has counted to zero: **`-2`** points.
- The latch is left 'open' after the counter has reached zero, but is still protected by the lock: `-2` points.

* *Calls to 'await' or 'signalAll' not protected by the lock: -4 points.*

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 `AtomicLong` 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() { ... }
    ...
}
```

Solution to Question 6

This question is Exercise 4-CP.1 of block 4 of CP. A possible solution is the following class:

```
public class ReentrantCASLock implements BasicLock {
    // value of -1 indicates that there is no thread in the critical section
    private AtomicLong lockTakenByThread;
    private int hold_count; // for reentrance

    public ReentrantCASLock() {
        this.lockTakenByThread = new AtomicLong(-1);
        this.hold_count = 0;
    }

    @Override
    public void lock() {
        long thread_nr = Thread.currentThread().getId();
        // reentrance
        if (this.lockTakenByThread.get() == thread_nr) {
            hold_count++;
            return;
        }

        // try to set the lock to thread_nr; repeat until successful
        while (! this.lockTakenByThread.compareAndSet(-1, thread_nr))
            ;

        // we acquired the lock through compare-and-set
        hold_count = 1;
        return;   not necessary!
    }

    @Override
    public void unlock() throws IllegalMonitorStateException {
        long thread_nr = Thread.currentThread().getId();
        // only when the hold_count == 0, we release the lock
        if (this.lockTakenByThread.get() == thread_nr) {
            if (--hold_count == 0)
                this.lockTakenByThread.set(-1);
        }
        else
            throw new IllegalMonitorStateException();
    }
}
```

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.

Solution to Question 7

a. (*3 pnts.*) No, because program order should be respected.

b. (*3 pnts.*)

`0`: First thread 1, then thread 2 is executed in order.

`3`: First thread 2, then thread 1 is executed in order.

`2`: Thread 2 is reorderded, `ready = true;` is executed, and then thread 1 is executed.

The assignment 'ready=true' and the test 'if (ready)' (and even the assignment 'r1=answer') are not related by the happens-before relation, so 'ready=true' might still be reordered.

c. (*2 pnts.*) Nothing. The reordering is still possible as only the writes before the assignment to the volatile variable `answer` should happen before this assignment.

d. (*2 pnts.*) Because `ready` is now `volatile`, the assignment to `answer` cannot come *after* the assignment to `ready`. Hence, reordering of thread 2 is not longer possible. So the possible end-value of `r1` is now either `0` or `3`.

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.

Solution to Question 8

a. (*2 pnts.*) This question is (almost) completely trivial; just apply `par` and `pseq` like done in every example presented in both the lectures and in the paper. The student needs some insight in `par` and `pseq` to explain the code. An implementation that applies an extra `par` instead of `pseq` is also correct.

b. (*7 pnts.*) Here the the *join* functionality from the execise session is needed. Every invocation generates two empty MVar variables: `joinl` and `joinr`, and forks two concurrent threads that write their result to `joinl` and `joinr`, respectively.

The listing below presents possible implementations for `ptreesum1` and `ptreesum2`:

```
module PTreesum where

import Control.Concurrent
import Control.Parallel
import Treesum

-- parallel version 1 (solution)
ptreesum1 :: Tree Int -> Int
ptreesum1 (Leaf v) = v
ptreesum1 (Node v l r) = a `par` (b `pseq` (v + a + b)) where
    a = ptreesum1 l
    b = ptreesum1 r

-- parallel version 2 (solution)
performwork :: Tree Int -> MVar Int -> IO ()
performwork (Leaf v) join     = do putMVar join v
performwork (Node v l r) join = do
    joinl <- newEmptyMVar
    joinr <- newEmptyMVar
    forkIO (performwork l joinl)
    forkIO (performwork r joinr)
    resl <- takeMVar joinl
    resr <- takeMVar joinr
    putMVar join (v + resl + resr)

ptreesum2 :: Tree Int -> IO Int
ptreesum2 t = do
```

```
    join <- newEmptyMVar
    forkIO (performwork t join)
    result <- takeMVar join
    return result
```

c. (*6 pnts.*) This kernel contains two major errors, namely: *data races* (lines 7 and 10), and *barrier divergence* (the two barriers). Note that barrier divergence leads to undefined behaviour; it may work or it might not. Solution: split computation up in reading- and writing phases, separated with barriers.

A corrected version of the kernel is the following:

```
__kernel void correct(__global int* arr, int size) {
    int index = get_global_id(0);

    int val = 0;

    // first reading phase
    if (index % 2 == 0) {
        val = 2;
    } else {
        val = arr[index + (size / 2) % size];
    }

    barrier(CLK_GLOBAL_MEM_FENCE);

    // first writing phase
    if (index % 2 == 0) {
        arr[index] *= val;
    } else {
        arr[index] += val;
    }

    barrier(CLK_GLOBAL_MEM_FENCE);

    // second reading phase
    if (index % 2 == 0) {
        val = arr[index - 4 % size];
    } else {
        val = 3;
    }

    barrier(CLK_GLOBAL_MEM_FENCE);

    // second writing phase
    if (index % 2 == 0) {
        arr[index] += val;
    } else {
        arr[index] *= val;
    }

    barrier(CLK_GLOBAL_MEM_FENCE);
}
```