EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

PROGRAMMING PARADIGMS
CONCURRENT PROGRAMMING

code: **201400537**
date: **5 July 2017**
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. Students in the *Programming Paradigms* module need to obtain at least a 5.0 for the test. 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!

ANSWERS

Question 1 (25 points)

One popular synchronization construct is a barrier. A barrier for a group of threads ensures that all threads stop at this point, and cannot proceed until all other threads have reached this barrier. In this exercise, we look at different mechanisms to implement barriers.

A barrier implementation should support the following operations:

- initialise the barrier, setting the number of threads that will be involved in the synchronisation; and
- `await`: a thread that calls this operation blocks until all threads involved in the synchronisation have reached the barrier.

a. (*15 pnts.*) Implement a barrier in Java **using a count down latch**. Make sure that your barrier is reusable, i.e., it can be reused multiple times to synchronise a group of threads.

- No latch used: -10
- Latch (re)creation by multiple threads: -5
- Unclear who re-initialises the latch (barrier for multiple uses): -4
- Re-use of latch: -8

b. (*10 pnts.*) Use transactional variables to give a Haskell implementation of a barrier.

Solution to Question 1

a. (*15 pnts.*)

```
import java.util.concurrent.CountDownLatch;

public class MyBarrier {

    private int parties;
    private CountDownLatch latch;

    public MyBarrier(int n) {
    parties = n;
    Thread t = new MyBarrierControl();
    t.start();
    }

    public void await() throws InterruptedException {
        latch.await();
    }

    class MyBarrierControl extends Thread {

    public void run () {
        while (true) {
        latch = new CountDownLatch(parties + 1);
        try {
            latch.await();
        }
        catch (InterruptedException e) {
        }
        }
    }
    }
}
```

b. (*10 pnts.*)

```
module Main where
import Control.Concurrent
import Control.Concurrent.STM

data Barrier = Barrier {
  parties :: TVar Int,
  joined :: TVar Int
}

init:: Int -> IO Barrier
init p = atomically $ do
     parties <- newTVar p
     joined <- newTVar 0
     return (Barrier parties joined)

await :: Barrier -> STM()
await bar = do
        n <- readTVar (joined bar);
        writeTVar (joined bar) (n + 1);
        tryLeave bar

tryLeave :: Barrier -> STM()
```

```
tryLeave bar = do
        p <- readTVar (parties bar);
        j <- readTVar (joined bar);
        if (j < p)
        then retry
        else do skip
```

- No initalisation: -3

Question 2 (35 points)

Consider the class `Delivery`.

```
class Delivery {

    private int[] box;

    public Delivery(int n) {
        box = new int[n];
        for (int i = 0; i < n; i++) {
             box[i] = 0;
        }
    }

    public void deliver(int parcels, int i) {
        box[i] = box[i] + parcels;
    }

    public int empty(int i) {
        int res = box[i];
        box[i] = 0;
        return res;
    }

}
```

This implements a sequence of message boxes. Delivery men can add a number of parcels to a box. The owner of the box can empty the box. For simplicity, we only record the number of items in each box.

a. (*6 pnts.*) Make a thread safe version of class `Delivery` in such a way that delivery men that want to deliver parcels in disjoint boxes simultaneously do not block each other.

b. (*2 pnts.*) Specify a suitable postcondition for method `deliver` in your thread safe version of `Deliver`. Explain your answer.

c. (*5 pnts.*) Suppose we want to add a transfer function, which transfer parcels from one box to another. Discuss what the main concurrency-related risk is, when adding this to your thread safe version, and how this risk can be avoided.

d. (*4 pnts.*) Naturally, parcels should not be lost. What sort of fairness is needed to ensure that every parcel will eventually be delivered? Motivate your answer.

e. (*10 pnts.*) Give a thread safe lock-free version of `Delivery`.

f. (*8 pnts.*) Give a message passing implementation for the `delivery` operation for a **single** delivery box. When a delivery man arrives, he tries to send a parcel to the box, and continues trying this, until he succeeds. As multiple delivery men can enter parcels in the box, you have to make sure that the message to confirm the receipt of the parcel is send to the right delivery man. (It is recommended to give an answer in Rust. However, points will not be subtracted for the concrete syntax, as long as the message passing schema is clear.)

Solution to Question 2

a. (*6 pnts.*

```
import java.util.concurrent.locks.*;
```

```
class DeliveryLocked {

    private int[] box;
    private Lock[] locks;

    public DeliveryLocked(int n) {
    box = new int[n];
    for (int i = 0; i < n; i++) {
        box[i] = 0;
        locks[i] = new ReentrantLock();
    }
    }

    public void deliver(int parcels, int i) {
    locks[i].lock();
    try {
        box[i] = box[i] + parcels;
    }
    finally {
        locks[i].unlock();
    }
    }

    public int empty(int i) {
    locks[i].lock();
    try {
        int res = box[i];
        box[i] = 0;
        return res;
    }
    finally {
        locks[i].unlock();
    }
    }


}
```

- No finally: -2

b. (*2 pnts.* Nothing can be guaranteed about the contents of the box after the method has finished, because more parcels might be added, or the parcels might be removed by other threads. (One could argue that `box[i]>= 0` is a postcondition (actually an invariant of the class)).

c. (*5 pnts*) Suppose we wish to transfer from box i to j. If the method first locks lock i, and then lock j, then two calls `transfer(i, j, ..)` and `transfer(j, i, ...)` might deadlock. This can be avoided by always locking the smallest box first.

- Only problem, but no solution: -2
- Splitting the locks will not work, because then one might not transfer all packets that have been delivered in i to j.

d. (*4 pnts.* Weak fairness will be sufficient. This guarantees that every delivery man that attempts to obtain the lock, will eventually get it. Once the delivery man has the lock, he will be able to deliver the package. Weak fairness is sufficient, as the delivery man is continuously enabled to acquire the lock. Points will be deducted is strong fairness is answered.

e. (*10 pnts.*) 4 points for correct use of `AtomicIntegerArray`. 3 points for each correctly implemented method.

```
import java.util.concurrent.atomic.*;

class DeliveryLockFree {

    private AtomicIntegerArray box;

    public DeliveryLockFree(int n) {
    box = new AtomicIntegerArray(n);
    for (int i = 0; i < n; i++) {
        box.set(i, 0);
    }
    }

    public void deliver(int parcels, int i) {
    int oldBox;
    int newBox;
    do {
        oldBox = box.get(i);
        newBox = oldBox + parcels;
    }
    while (!box.compareAndSet(i, oldBox, newBox));
    }

    public int empty(int i) {
    int oldBox;
    do {
        oldBox = box.get(i);
    }
    while (!box.compareAndSet(i, oldBox, 0));
    return oldBox;
    }
}
```

f. (*8 pnts.*)

```
use std::sync::mpsc;
use std::sync::mpsc::Sender;
use std::thread;

struct Message {
  parcel : i32,
  address : Sender<bool>,
}

fn main () {

   let (deliver, receive) = mpsc::channel();
   let (answer, response) = mpsc::channel();

   // delivery man
   thread::spawn(move || {
     let mut send = false;
     while !send {
       let answer_clone = answer.clone();
       deliver.send(Message{parcel: 1, address: answer_clone}).unwrap();
       send = response.recv().unwrap();
     }
   });

   // box
```

```
    thread::spawn ( move || {
      loop {
        let chan = receive.recv().unwrap().address;
        chan.send(true).unwrap();
      }
    });;

}
```

Question 3 (15 points)

a. (*3 pnts.*) Consider the following fragment of a light handling system.

```
// nrLights is length of lights array
// DIMMODE is some boolean condition

# pragma omp parallel for shared (lights, MAX, DIMMODE) private (i)
for (i = 0; i < nrLights; ++i) {
  lights[i] = MAX;
  if (DIMMODE && ((i % 2) == 0) {
    lights[i] = MAX/2;
  }
}
```

Describe what will be the effect of the OpenMP pragma on the behaviour of this fragment.

b. (*4 pnts.*) Consider the following fragment of a light handling system.

```
// nrLights is length of lights array
// length of init is the same as length of lights

# pragma omp parallel for shared (lights, init, MAX, DIMMODE) private (i)
for (i = 1; i < nrLights; ++i) {
  lights[i] = init[i];
  if (lights[i] < lights[i - 1])
    lights[i] = lights[i - 1];
 }
}
```

Describe what will be the effect of the OpenMP pragma on the behaviour of this fragment.

c. (*4 pnts.*) Give an OpenCL kernel that has the same effect as the sequential version of this second fragment.

d. (*4 pnts.*) Consider the following OpenCL kernel.

```
   //size is length of lights and init array
2
   __kernel lights (__global float* lights, __global float* init, int size) {
4      int index = get_global_id(0);
       lights[index] = 0;
6      for (int i = 0; i < size; i++) {
           if (i % (index + 1) == 0) {
8              lights[index] = lights[index] + init[i];
           }
10     }
   }
```

Would adding a barrier before the `for`-loop in line 6 change the final value of the `lights` array?

Solution to Question 3

a. (*3 pnts.*) It will parallellize its execution, which is fine because all iterations are independent.

b. (*4 pnts.*) The parallellization will mess up behaviour, because there is a dependence between the iterations of the loop.

c. (*4 pnts.*) Needs two barriers (-2 points for each missing barrier):
light[i] = init[i]
BARRIER

read light[i - 1], store in local variable x
BARRIER compare light[i] and x, and if necessary update x

d. (*4 pnts.*) No, because every thread only considers a single element of the light array. Initalizes lights array. Each thread iterates through init array (ok, because read-only) and sums up a selected number of initialisation values.

Question 4 (15 points)

a. (*3 pnts.*) Suppose we transform the following program fragment:

```
l.lock();
try {
  x = 3;
  y = 4;
}
finally {
  l.unlock();
}
```

into the following program fragment

```
l.lock();
try {
  y = 4;
}
finally {
  l.unlock();
}
x = 3;
```

Suppose this is used in a context with other threads that need up-to-date values of variables `x` and `y`. Explain whether this program transformation is okay. Motivate your answer.

b. (*3 pnts.*) Suppose we have the following 3 threads, where initally `x = 0`, `y = 0`, and `z = 0`, and `r1`, `r2` and `r3` are local variables.

Thread 1	Thread 2	Thread 3
`x = 10;`	`y = 8;`	`z = 6;`
`int r1 = z;`	`int r2 = x;`	`int r3 = y;`

Give **all** possible combinations of the final values of `r1`, `r2` and `r3` under a relaxed memory model.

c. (*3 pnts.*) Consider the following program.

```
class Hotel {
    volatile String[] rooms = new String[34];

    //@ requires 0 <= number && number < 34;
    void checkIn(Guest name, int number) {
        rooms[number] = name;
    }

    //@ requires 0 <= number && number < 34;
    void checkOut(int number) {
        rooms[number] = null;
    }
}
```

When a single instance of this class is used by multiple threads, explain whether this program has data races. Motivate your answer.

d. (*6 pnts.*) Suppose two neighbours, Bob and John, share a garden. Bob has a cat, and John has a dog. Both animals like to be outside in the garden, but they cannot be outside at the same time, as they will start fighting. They agree to use a flag system that signals when their animals are outside, in order to avoid fights. An attempt to implement this system is made in the classes `Garden`, `Neighbour`, and `AnimalControl` (see Figure 1).

Discuss at least two reasons why this implementation will not prevent that the cat and the dog are outside simultaneously.

```
public class Garden {

    public static final int EMPTY = 0;
    public static final int CAT = 1;
    public static final int DOG = 2;
    private boolean flag = false;

    //@ invariant animal == EMPTY || animal == CAT || animal == DOG;
    private int animal = EMPTY;

    //@ requires an == CAT || an = DOG;
    public void enter(int an) {
    animal = an;
    }

    //@ requires animal = CAT || animal = DOG;
    public void leave() {
        animal = EMPTY;
    }

    public void raiseFlag() {
        flag = true;
    }

    public void lowerFlag() {
        flag = false;
    }

    public boolean inspectFlag() {
        return flag;
    }
}

public class Neighbour extends Thread {

    private final int animal;
    private Garden garden;

    public Neighbour(int an, Garden g) {
        animal = an;
        garden = g;
    }

    public void run () {
        while (garden.inspectFlag());
        garden.raiseFlag();
        garden.enter(animal);
        garden.leave();
        garden.lowerFlag();
    }
}

public class AnimalControl {

    public static void main(String [] args) {
        Garden g = new Garden();
        Neighbour bob = new Neighbour(Garden.CAT, g);
        Neighbour john = new Neighbour(Garden.DOG, g);
        bob.start();
        john.start();
    }
}
```

Figure 1: The Animal Control System

Solution to Question 4

a. (*3 pnts.*) This program transformation changes when the update to `x` becomes visible to other threads (and thus usually will change the behaviour of the program).

b. (*3 pnts.*) (0, 0, 0) (because of reordering, 2 pnts)
(10, 0, 0)
(0, 8, 0)
(0, 0, 6)
(10, 8, 0)
(0, 8, 6)
(10, 0, 6)
(10, 8, 6) (all because of different possible interleavings)

c. (*3 pnts.*) This program has a data race on the writes to `rooms[number]`, because `volatile` only affects the reference to the array, not the access to the elements of the array.

d. (*6 pnts.*) flag is not volatile, thus updates are not immediately visible. In particular, if one sets the flag to true, the other might not see it, and also release its animal. Moreover, the change from seeing that the flag is down, to raising it should be done in a single atomic step.

Question 5 (10 points)

Consider the following Rust program:

```
use std::thread;
use std::sync::Mutex;
use std::sync::Arc;

fn apply(x : usize) -> usize {
  // apply computation to this job
}

fn main () {


    let n = // number of jobs
    let mut jobs = Vec::new();
    for i in 0 .. n {
        // initialise jobs
    }
    let jobs_copy = jobs.clone();

    let results = Arc::new(Mutex::new(vec![0;n]));
    let results_copy = results.clone();

    let worker1 = thread::spawn(move || {
      for i in 0 .. n/2 {
        results.lock().unwrap()[i] = apply(jobs[i])
      }
    });

    let worker2 = thread::spawn(move || {
      for i in n/2 .. n {
        results_copy.lock().unwrap()[i] = apply(jobs_copy[i]);
      }
    });
```

```
34    worker1.join().unwrap();
      worker2.join().unwrap();
36  }
```

This program has a list of jobs to which a function `apply` should be applied. It does this in two parallel threads. The first thread applies this function to the first half of the jobs, the second thread applies this function to the second half of the jobs.

a. (*3 pnts.*) Explain why `jobs` and `results` are cloned.

b. (*4 pnts.*) The variable `results` has a far more complicated initialisation than `jobs`. Explain what this initialisation does, and why this is necessary.

c. (*3 pnts.*)Modify the program, such that the main thread after joining `worker1` and `worker2` can print out the final values of `result`.

Solution to Question 5

a. (*3 pnts.*) When the thread is created, ownership of jobs and results is moved from the main thread to the worker1 thread. In order to pass these variables also to the second thread, a clone is created. As the main thread no longer owns the variables, it can not transfer them to worker2 anymore.

b. (*4 pnts.*) The threads update the contents of the `results` array. This has to be protected by a lock. Moreover, Arc is used to allow multiple owners of the variables (in a thread safe way).

c. (*3 pnts.*)

```
use std::thread;
use std::sync::Mutex;
use std::sync::Arc;

fn apply(x : usize) -> usize {
  return x;
}

fn main () {


    let n = 100;
    let mut jobs = Vec::new();
    for i in 0 .. n {
        jobs.push(i);
    }
    let jobs_copy = jobs.clone();

    let results = Arc::new(Mutex::new(vec![0;n]));
    let results_copy = results.clone(); // added
    let results_final = results.clone();

    let worker1 = thread::spawn(move || {
      for i in 0 .. n/2 {
        results.lock().unwrap()[i] = apply(jobs[i])
      }
    });

    let worker2 = thread::spawn(move || {
      for i in n/2 .. n {
        results_copy.lock().unwrap()[i] = apply(jobs_copy[i]);
      }
```

```
        });

    worker1.join().unwrap();
    worker2.join().unwrap();
    // added
    for i in 0 .. n {
        println!("{}",results_final.lock().unwrap()[i]);
    }
}
```