EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

PROGRAMMING PARADIGMS
CONCURRENT PROGRAMMING

code: **201400537**
date: **23 June 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 **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. 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 (15 points)

An *Exchanger* is a synchronization mechanism where two threads participate and swap elements with each other. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return.

```
interface ExchangeInterface<V> {

    public V exchange(V x) throws InterruptedException;

}
```

Provide an implementation of this `ExchangeInterface` using a `ReentrantLock` in combination with one or more `Condition` objects. Make sure that there always exactly **two** parties involved in the exchange.

Solution to Question 1

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

class Exchanger<V> implements ExchangeInterface<V> {

    private V buffer;
    private boolean reserved;

    final Lock l = new ReentrantLock();
    final Condition exchange = l.newCondition();
    final Condition newRound = l.newCondition();

    public V exchange(V x) throws InterruptedException {
    l.lock();
    try {
        if (buffer == null) {
        buffer = x;
        exchange.await();
        V res = buffer;
        buffer = null;
        reserved = false;
        newRound.signalAll();
        return res;
        } else {
        while (reserved) {
            newRound.await();
        }
        reserved = true;
        V res = buffer;
        buffer = x;
        exchange.signalAll();
        return res;
        }
    }
    finally {
        l.unlock();
    }
    }
}
```

Points subtracted for:

- If more than two parties can be involved in the exchange, i.e if the following scenario is possible:
  A writes x
  B writes y, returns x
  C writes z, return y
  A returns z
- Unlock not in finally clause

Question 2 (20 points)

Consider the following class `BoundedMap`:

```
class BoundedMap {

    private int[] contents;

    public BoundedMap(int size) {
        contents = new int[size];
    }

    public int lookup(int i) {
        return contents[i];
    }

    public void update(int i, int x) {
        contents[i] = x;
    }

    public void triple(int i) {
        contents[i] = 3 * contents[i];
    }
}
```

This gives a sequential implementation to map values 0 to `size` to some other integer. It contains functions to lookup the value stored at position `i`, update the value stored at position `i`, and to multiply the value stored at position `i` by 3.

a. (*4 pnts.*) Discuss how to make a thread safe version of `BoundedMap` using locks. (There is no need to repeat all code, as long as you clearly indicate where the changes are.)

b. (*3 pnts.*) Give a suitable `guardedby` annotation for your program.

c. (*3 pnts.*) Discuss what would be suitable pre- and postconditions for the methods `update` and `triple` in your thread safe version of `BoundedMap`. Motivate your answer.

d. (*10 pnts.*) Give a lock-free version of the bounded map.

Solution to Question 2

a. (*4 pnts.*)

```
class LockedBoundedMap {

    private int[] contents;

    public LockedBoundedMap(int size) {
        contents = new int[size];
    }

    public synchronized int lookup(int i) {
        return contents[i];
    }

    public synchronized void update(int i, int x) {
        contents[i] = x;
    }

    public synchronized void triple(int i) {
```

```
            contents[i] = 3 * contents[i];
        }
    }
```

b. (*3 pnts.*) `contents guardedby this`

c. (*3 pnts.*) Precondition: i should be between bounds of the array: `0 <= i && i < contents.length` (max 1 pnts.) You can only specificy `true` as postconditions (2 pnts), because other threads might change the state of the bounded map.

d. (*10 pnts.*) Division of points: 3 for correct use of `AtomicIntegerArray`, 2 points for methods `get` and `update`, 5 points for method `triple`. If `update` methods uses `compareAndSet` point subtracted for unnecessary complexity.

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

class LockFreeBoundedMap {

    private AtomicIntegerArray contents;

    public LockFreeBoundedMap(int size) {
        contents = new AtomicIntegerArray(size);
    }

    public int lookup(int i) {
        return contents.get(i);
    }

    public void update(int i, int x) {
        contents.set(i, x);
    }

    public void triple(int i) {
        int oldVal;
        int newVal;
        do {
            oldVal = contents.get(i);
        newVal = 3 * oldVal;
        } while (! contents.compareAndSet(i, oldVal, newVal));
    }
}
```

Question 3 (10 points)

A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

a. (*6 pnts.*) Many implementations of a ReadWriteLock (for example, Java's `ReentrantReadWriteLock` does not allow locks to be upgraded, i.e., when you hold a read lock, you cannot transform this into a write lock, without releasing the lock first. The reason that this is disallowed is because it causes a deadlock. Explain how such a deadlock could occur.

b. (*4 pnts.*) When using a ReadWriteLock in a fair mode, different strategies could be implemented to decide which thread trying to acquire one of the two locks can go first. Discuss under which circumstances the strategy where threads trying to acquire the write lock always go first would be suitable.

Solution to Question 3

a. (*6 pnts.*) Thread 1 and 2 both hold a read lock, and try to upgrade it to a write lock. They both keep their read lock, so the write lock can never be acquired, because there always is another thread still reading.

b. (*4 pnts.*) This is suitable if there are not too many write threads, so that there are moments where threads that only wish to read would get access. Having the possibility to read always an up-to-date value is important, but only if one observes that readers might actually starve, if there are too many writes.

Question 4 (16 points)

Below are 4 multiple choice questions. For each question, motivate your answer. Points are only given if the motivation is correct.

a. (*4 pnts.*) Suppose `x` and `y` are shared variables, while `r1` and `r2` are local variables. If initially `x` = 3 and `y` = 4, what can we say about the possible final values of `r1` and `r2`.

Thread 1	Thread 2
`x = 6`	`y = 7`
`int r1 = y`	`int r2 = x`

A. `r1` can never be 7.

B. `r2` can never be 6.

C. `r1` may be 13.

D. `r1` may be 4, `r2` may be 3.

b. (*4 pnts.*) Consider the following Java program.

```
class DR {
    public static int x = 0;
    public static int y = 0;

    public static void main(String [] args) {
        (new Thread1()).start();
        (new Thread2()).start();
    }
}

class Thread1 extends Thread {
    public void run() {
        int r1 = DR.x;
        if (r1 > 0) {
            DR.y = 1;
        }
    }
}

class Thread2 extends Thread {
    public void run() {
        int r2 = DR.y;
        if (r2 > 0) {
            DR.x = 1;
        }
    }
}
```

Does this have data races or race conditions?

A. None

B. Only data races

C. Only race conditions

D. Both

c. (*4 pnts.*) Consider the following Java program.

```
class DR2 {
    public static int x = 0;

    public static void main(String [] args) {
        (new Thread1()).start();
        (new Thread2()).start();
    }
}

class Thread1 extends Thread {
    public void run() {
        DR2.x = DR2.x + 2;
    }
}

class Thread2 extends Thread {
    public void run() {
        DR2.x = DR2.x * 2;
    }
}
```

Does this have data races or race conditions?

A. None

B. Only data races

C. Only race conditions

D. Both

d. (*4 pnts.*) Consider the following Java program.

```
import java.util.concurrent.locks.*;
public class Bucket {

    //@ invariant elements.length == locks.length;
    Object[] elements;
    Lock[] locks;
    volatile int nrElements;

    public Bucket(int n) {
        elements = new Object[n];
        locks = new Lock[n];
        for (int i = 0; i < n; i++) {
            elements[i] = new Object();
            locks[i] = new ReentrantLock();
        }
        nrElements = 0;
    }

    //@ requires i < elements.length;
    public void add(int i, Object o) {
        locks[i].lock();
        elements[i] = o;
        locks[i].unlock();
        nrElements++;
    }

    //@ requires i < elements.length;
    public void remove(int i) {
        locks[i].lock();
        elements[i] = null;
        locks[i].unlock();
        nrElements--;
    }
}
```

Where is the performance bottleneck in this code?

A. Too many locks

B. `nrElements` accessed by all mutating operations

C. Locks maintained in an array

D. Locks protect too much data

Solution to Question 4

a. (*4 pnts.*) Correct answer d, due to data races, reordering is allowed.

b. (*4 pnts.*) Correct answer a, the branch conditions will always fail, so the then-part is never executed. Therefore no data races, and also no race conditions, as the behaviour is fully deterministic now.

c. (*4 pnts.*) Correct answer d. Data race on `DR2.x` and final value of `DR2.x` depends on the actual interleaving that is executed.

d. (*4 pnts.*) Correct answer b. `nrElements` is a hot field.

Question 5 (9 points)

One common problem in concurrent programming is the management of shared resources. In this exercise, you are asked to write a small resource management system in Haskell, using software transactional memory. The resource management system uses a counter to keep track of the amount of resources available. Processes may ask for an arbitrary amount of resources and block if they're not available.

Let us define the type of resources as follows: `type Resource = TVar Int`.

Define operations to:

- initialize availability of a resource,
- `acquire` a certain number of resources, and
- `release` a certain number of resources.

Solution to Question 5

(okay to do this atomically)

```
type Resource = TVar Int

init:: -> Int -> STM()
init nr = newTVar res nr

acquire :: Resource -> Int -> STM ()
acquire res nr = do n <- readTVar res
                    if n < nr
                    then retry
                    else writeTVar res (n - nr)

release :: Resource -> Int -> STM ()
release res nr = do n <- readTVar res
                    writeTVar res (n + nr)
```

Question 6 (15 points)

Consider the Rust program in Listing 1. The idea of this program is that a long list of sampled data is produced using a function `sample`. This sampled data is processed in multiple ways (in process A and process B), and the outcome of the various processors is printed.

Unfortunately, the current program does not compile.

a. (*2 pnts.*) The first error message of the compiler reads as follows:

```
error: cannot borrow immutable local variable `data_a` as mutable
  --> plotter.rs:15:12
   |
13 |        let data_a = vec![0;100];
   |            ------ use `mut data_a` here to make mutable
14 |        for i in 0 .. 100 {
15 |            data_a[i] = data[i] * 2;
   |            ^^^^^^ cannot borrow mutably
```

(a similar error message is given for `data_b`). Explain the reason of this error message, and how it can be solved.

b. (*8 pnts.*) The second error message of the compiler reads:

```
error[E0373]: closure may outlive the current function, but it borrows
 `data`, which is owned by the current function
  --> plotter.rs:20:34
   |
20 |    let process_b = thread::spawn(|| {
   |                                  ^^ may outlive borrowed value `data`
...
23 |            data_b[i] = data[i] * 3;
   |                        ---- `data` is borrowed here
   |
```

Discuss **two** different ways to solve this problem.

c. (*5 pnts.*) Adapt the program to use message passing instead of shared data.

Solution to Question 6

a. (*2 pnts.*) The variables need to be declared mutable, using keyword `mut`.

b. (*8 pnts.*) Making a clone of data, move data to process A, and the clone of data to process B. Mentioning only move, or only cloning gives half points.

```
use std::thread;


fn sample() -> Vec<i32> {
   return vec![6;100];
}

fn main () {

   let data = sample();

   let data_clone = data.clone();
```

```
use std::thread;

fn sample() -> Vec<i32> {
    // generate a sequence of data
}

fn main () {

    let data = sample();

    let process_a = thread::spawn(|| {
        let data_a = vec![0;100];
        for i in 0 .. 100 {
            data_a[i] = data[i] * 2;
        }
        return data_a;
    });

    let process_b = thread::spawn(|| {
        let data_b = vec![0;100];
        for i in 0 .. 100 {
            data_b[i] = data[i] * 3;
        }
        return data_b;
    });

    let res_a = process_a.join().unwrap();
    let res_b = process_b.join().unwrap();

    for i in 0 .. 100 {
        println!("({},{})",res_a[i], res_b[i]);
    }
}
```

Listing 1: Plotter Algorithm in Rust

```
    let process_a = thread::spawn(move || {
        let mut data_a = vec![0;100];
        for i in 0 .. 100 {
            data_a[i] = data[i] * 2;
        }
        return data_a;
    });

    let process_b = thread::spawn(move || {
        let mut data_b = vec![0;100];
        for i in 0 .. 100 {
            data_b[i] = data_clone[i] * 3;
        }
        return data_b;
    });

    let res_a = process_a.join().unwrap();
    let res_b = process_b.join().unwrap();

    for i in 0 .. 100 {
        println!("({},{})",res_a[i], res_b[i]);
    }
}
```

Or change the vector into an array, so it becomes copyable.

```
use std::thread;


fn sample() -> [i32;100] {
    return [6;100];
}

fn main () {

    let data = sample();

    let process_a = thread::spawn(move || {
        let mut data_a = vec![0;100];
        for i in 0 .. 100 {
            data_a[i] = data[i] * 2;
        }
        return data_a;
    });

    let process_b = thread::spawn(move || {
        let mut data_b = vec![0;100];
        for i in 0 .. 100 {
            data_b[i] = data[i] * 3;
        }
        return data_b;
    });

    let res_a = process_a.join().unwrap();
    let res_b = process_b.join().unwrap();

    for i in 0 .. 100 {
        println!("({},{})",res_a[i], res_b[i]);
    }
}
```

Use of message passing is also an option to solve this (but still requires move and cloning, so more involved than suggested).

c. Points subtracted if the solution is actually making the computations sequential, or if the results of the computations is not returned.

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


fn sample() -> Vec<i32> {
   return vec![6;100];
}

fn main () {

   let (tx_a, rx_a) = mpsc::channel();
   let (tx_b, rx_b) = mpsc::channel();
   let (res_a, plot_a) = mpsc::channel();
   let (res_b, plot_b) = mpsc::channel();


   thread::spawn(move || {
      let data = sample();
      let data_clone = data.clone();
      tx_a.send(data).unwrap(); // unwrap for warning unused variable
      tx_b.send(data_clone).unwrap(); // unwrap for warning unused variable
      });

   thread::spawn(move || {
       let data = rx_a.recv().unwrap();
       let mut data_a = vec![0;100];
       for i in 0 .. 100 {
           data_a[i] = data[i] * 2;
       }
       res_a.send(data_a).unwrap(); // unwrap for warning unused variable
   });

   thread::spawn(move || {
       let data = rx_b.recv().unwrap();
       let mut data_b = vec![0;100];
       for i in 0 .. 100 {
           data_b[i] = data[i] * 3;
       }
       res_b.send(data_b).unwrap(); // unwrap for warning unused variable
   });

   let res_a = plot_a.recv().unwrap();
   let res_b = plot_b.recv().unwrap();

   for i in 0 .. 100 {
       println!("({},{})",res_a[i], res_b[i]);
   }
}
```

Question 7 (15 points)

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given graph, where all edges have a weight.

Listing 2 shows a C implementation of this algorithm. In this algorithm, the graph is represented by a matrix, such that `graph[i][j]` returns the weight of the edge between nodes `i` and `j`. Note that `graph[i][j]` is 0 if `i` is equal to `j`, and `graph[i][j]` is `INF` (infinite) if there is no edge from node `i` to `j`.

The code in lines 1 - 14 does initialisation, the loop in lines 21 - 36 is the main part of the algorithm. The complexity of this algorithm is n^3, where n is the number of nodes. Clearly, parallellism could be used to improve the performance of the algorithm.

a. (*7 pnts*) Discuss 3 different ways in which the main loop could be parallellized, and give suitable OpenMP pragma's for each of them.

b. (*8 pnts*) Implement an OpenCL kernel that executes this algorithm on a GPU. (No points will be subtracted for incorrect syntax). Explain how you get to your solution.

Solution to Question 7

a. (*7 pnts*) This question was actually more involved than foreseen. The inner loop can be parallellized with a parallel for (but the other loops have dependencies). Full points have been awarded for any solution that noticed this issue.

b. (*8 pnts*) Also this question was more involved than foreseen. The two inner loops (on i and j) can be removed. The kernel uses a matrix (uses two thread ideas). The body should do the for loop on k. All necessary variables first have to be read and stored in a local variable, then a barrier is needed for synchronization. Then dist[i][j] can be updated, and a barrier needs to be executed before the threads can proceed. Also a barrier is needed after initialisation. Full points have been awarded for any solution in this direction. Points have been subtracted if the computation was done only once, without a repeat.

```
// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
      ---> Before start of a iteration, we have shortest distances between all
      pairs of vertices such that the shortest distances consider only the
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the set of
      intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

}
```

Listing 2: Floyd Warshall Algoritm