EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

PROGRAMMING PARADIGMS
CONCURRENT PROGRAMMING

code: **201400537**
date: **22 June 2018**
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 **6 questions**. The final grade is computed as the number of points, divided by 10. 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!

Question 1 (10 points)

Suppose you are implementing a webshop application. Performance is very important for such an application: by reducing the time spent on a transaction for a single client, the profit for the webshop can be optimised.

The webshop application will have the following components:

- a database that keeps track of the items on sale, and for each item, the number of items available in the store;
- the client administration; and
- the payment system.

a. (*6 pnts.*) Discuss 3 different techniques that can be used when implementing the application to improve the performance, while keeping it thread-safe. For each technique, discuss how to use it for this application, and why you expect it will improve the application's performance.

b. (*2 pnts.*) What sort of fairness guarantee will you need for this application? Explain your answer.

c. (*2 pnts.*) Will this guarantee always be sufficient to ensure that your clients are satisfied with the responsiveness of your webshop? Explain your answer.

Question 2 (35 points)

In this exercise, we will consider different implements of the webshop's store house. To make it easier to find items in the store house, all items have a location attached to it. The store house is divided in 400 rows, and within each row there are 40 different shelves. Thus, the location of an item consists of a row number and a shelf number. Assuming that we have a type `Item` to represent all possible items sold in the webshop, the following code fragment declares the datastructures used to model the store house.

```
import java.util.Map;
class Store<Item> {
    public static final int ROWS = 400; //number of rows in the store house
    public static final int SHELVES = 40; // number of shelves per row

    private class Location {   // location: row + shelf number
        private int row;
        private int shelf;
    }

    // items maps each available item to its location
    private Map<Item, Location> items;

    // number of items stored at each location
    //@ (\forall int i, int j; 0 <= i && i < ROWS; 0 <= j && j < SHELVES; storage[i][j] >= 0);
    private int[][] storage = new int [ROWS][SHELVES];
}
```

The store house application implements the following (thread-safe) operations:

- `int lookup(Item i)`: check how many items `i` are still available;
- `void take(Item i, int n)`: take `n` items `i` from the storage, in case sufficient items are available; otherwise wait until the store has been refilled and sufficient items are available again; and
- `void put(Item i, int n)`: add `n` items `i` to the storage. For simplicity, we assume there is no maximum number of items.

a. (*3 pnts.*) The field `items` will not be protected by a lock. Discuss why this choice can be made, while preserving thread-safety, and what are the consequences of this choice.

b. (*12 pnts.*) Give a thread-safe implementation for the operations `lookup`, `take`, and `put` using a *single* lock, which will still provide a reasonable performance. Explain your choice of lock, and indicate your locking policy.

c. (*8 pnts.*) Adapt your solution to have a separate lock for each row. (It is okay to indicate which parts of the code may be copied from your previous answer, and which parts are changed).

d. (*4 pnts.*) The webshop owner has some pecularities. Therefore, he asks you to change the implementation such that the total number of items in a row is increasing for every row. As a JML property, the following invariant should hold:

```
/*@ invariant (\forall int i, int j; 0 <= i && i < j && j < ROWS;
                              (\sum int k; 0 <= k && k < SHELVES; row[i][k]) <=
                              (\sum int k; 0 <= k && k < SHELVES; row[j][k]));
```

or in words: if $i < j$, then the total number of items in `row[i]` should be less or equal than the total number of items in `row[j]`.
Discuss for both of your solutions (single lock, and locks per row) whether it is possible to adapt the code to preserve this property without changing the locking strategy. Motivate your answer. (There is no need to spell out the solution, an answer at conceptual level is sufficient.)

e. (*8 pnts.*) The performance of the webshop database is still not optimal: when many customers are active, the database sometimes reacts very slow. Therefore, make an alternative, *lock-free* implementation of the methods `lookup`, `take` and `put`. Discuss which changes are necessary to the datastructures used to model the store house.

Question 3 (20 points)

On Mondays, the webshop always receive a resupply. For every item in the store house, a 100 new instances are received. When this happens, the webshop application is stopped for a moment, and several operations are done to compute some statistics about the sales of last week, and to update the store. Naturally, we want the moment that the application is stopped to be as short as possible, so try to write operations that are as fast as possible.

a. (*5 pnts.*) Write a sequential operation to update the storage of every item (you can assume that every location has an item assigned to it, so there is no need to use the `items` map here), and then use OpenMP-like annotations to parallellise your code. Discuss two possible parallellisations. Which one would you expect to have the best performance.

b. (*5 pnts.*) As said, the webshop owner has its pecularities, and before reopening the webshop again, he wants to be sure that the number of items in a row is still increasing. Write a sequential operation that computes the sums of items in a row, for every row. The result should be stored in an array `int [] itemsPerRow`. Use OpenMP-like annotations to parallellise the loops in your program, and explain your answer.

c. (*5 pnts.*) The webshop owner also would like to know which product sells best. He has a few GPUs, so he asks you to write an OpenCL kernel that computes for each row the item that has the smallest number of items stored. Write a kernel that takes an array as input and computes the smallest number stored in that array in a *logarithmic* number of steps. It is okay to overwrite the contents of the array, and it is *not* necessary to return the index of the array where the minimal value was originally stored. You may assume that the length of the array is less or equal to 2 times the number of threads in a single workgroup. You may also assume that the length of the array is a power of 2, i.e., in every step the number of items that need to be multiplied is even.

d. (*5 pnts.*) Suppose that your array is larger than can be handled with a single workgroup. Discuss how to adapt the approach to make it work correctly in that case. You do not have to give precise code, it is sufficient to sketch your solution.

Question 4 (10 points)

For the payment system, a Haskell solution is chosen.

a. (*5 pnts.*) For each item, we use a mutable variable to store the price of the item. We define a list `prices :: [(Item, Price)]`, where `Price = MVar int`. Define the following operations to update and inspect the list of prices.

- `create prices i p :: [(Item, Price)]`: to add item `i` to the pricelist with initial prize `p`, you may assume `i` is not already in the list.
- `lookup prices i :: int`: return the current price of item `i`. You may assume `i` is in the list.
- `update prices i p :: [(Item, Price)]`: update the price of item `i` to `p`. You may assume `i` is in the list.

b. (*5 pnts.*) For the payment system itself, a Software Transactional Memory solution is chosen, where bank accounts are modelled as transactional variables. Given a bank account number and an item, the amount is taken from the client's bank account and added to the webshop owner's bankaccount. Define the following transactions:

- `pay acc i prices`: lookup the price for item `i` and take this amount from the bank account. If there is insufficient money on the account, this fails.
- `receive acc n`: receive the amount `n` on the bankaccount.
- `transfer acc1 acc2 i prices`: transfer the price for item i from account `acc1` to account `acc2`.

Question 5 (10 points)

For the client administration, privacy of the data is very important, and the webshop owner has heard that ownership is built into Rust, therefore he insists that you use Rust to implement this part of the application.

The addresses of all clients are registered using a struct `Address`. Initially, not all provinces have been entered, but an `updateAddress` function is defined that looks up which postal code is related to which province.

a. (*6 pnts.*) Consider this attempt to implement and use `updateAddress`.

```
struct Address {
   street : String,
   postal_code_number : i32,
   postal_code_letters : String,
   province : String
}

// Only partial implementation shown. This should be a big case-expression.
fn lookup_province(postal_code_number : i32) -> String {
   String::from("Overijssel")
}

fn update_address(address : Address) {
   address.province = lookup_province(address.postal_code_number)
}

fn register(address : Address) {
   // some suitable implementation
}

fn main() {
  let address = Address {street : String::from("Hengeloseweg"),
                         postal_code_number : 7521,
                         postal_code_letters: String::from("AD"),
                         province: String::from("")};
  update_address(address);
  register(address)
}
```

This program does not compile. Explain which two error messages are given by the compiler, and why. What should be done to solve these error messages?

b. (*4 pnts.*) The webowner would like to do some statistics on the origins of the clients. The following program counts the number of clients in Overijssel, and in parallel prints all postal codes of all customers.

```
use std::thread;

// Only partial implementation shown. This should be a big case-expression.
fn lookup_province(postal_code_number : i32) -> String {
   String::from("Overijssel")
}

fn main() {
   let postal_codes = [7521, 7522, 7523];
   let handle = thread::spawn(move || {
         let mut count = 0;
         for i in 1 .. 3 {
           if lookup_province(postal_codes[i]) == "Overijssel" {
           count = count + 1; }
     }
     count
  });
  for i in 1 .. 3 {
    println!("␣{}␣", postal_codes[i]);
  }
  handle.join();
}
```

Even though this program does not use locking, the Rust compiler accepts it. Explain why Rust considers this program to be safe for concurrent use.

Question 6 (15 points)

Before the webshop owner is willing to pay you for all your work on the webshop application, he wishes to test your knowledge about concurrent programming. Therefore, you are asked to answer the following random questions.

a. (*4 pnts.*) Consider the classes `Plane` and `Passenger`. Explain what concurrency error is exposed by these classes.

```
import java.util.*;
public class Plane {
    private List<Passenger> passengers;
    private int flightnumber;
    private String destination;

    public synchronized int getFlightnumber() {
        return flightnumber;
    }

    public synchronized List<String> passengerList() {
        List<String> passengerList = new ArrayList<String>();
        for (Passenger p: passengers) {
            passengerList.add(p.getName());
        }
        return passengerList;
    }
}

public class Passenger {
    private Plane flight;
    private String name;

    public synchronized String getName() {
        return name;
    }

    public synchronized int getFlightnumber() {
        return flight.getFlightnumber();
    }
}
```

b. (*7 pnts.*) A countdown latch is a synchronisation mechanism that allows a number of threads to wait until a set of operations being performed in other threads completes.
A CountDownLatch is initialized with a given count. It has an `await` method and a `countDown` method. The `await` method blocks until the current count reaches zero. The `countDown` method decreases the count by 1. Once count reaches 0, all waiting threads are released, and any subsequent invocations of `await` return immediately.
Provide a *lock-free* implementation of a countdown latch.

c. (*4 pnts.*) Suppose we have the following two threads (where initially `x`, `y` and `z` are equal to 0 and `t` is a shared lock):

```
// Thread 1
z = 3;
y = (z + 3);
t.lock();
r2 = x;
x = 32;
t.unlock();
```

```
// Thread 2
t.lock();
r1 = z;
x = 64;
t.unlock();
```

What are the possible final values of `r1` and `r2`. Use the notions of *program order* and *synchronization order* to explain your answer.