TEST
CONCURRENT PROGRAMMING

code: **201400537**
date: **19 June 2015**
time: **13.45–16.45**

- This is an 'open book' examination. You are allowed to have a copy of *Java Concurrency in Practice*, and a copy of the (unannotated) lecture *slides*. You are *not* allowed to take personal notes 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.
  Students for the **Concurrent & Distributed Programming** course also need to obtain at least a 5.0 for the test. This test result is 80 % of your final grade (the other 20 % is given by the distributed programming homework).
  Students that attended at least 6 out of 7 exercise sessions obtain a 1.0 bonus.

ANSWERS

Question 1 (15 points)

Suppose we have an application with N producers and N consumers that share information over a bounded queue. For performance reasons, each producer has its own queue where it can store the data it produces (thus, there are N queues in total).

Discuss *three different* possible ways to implement the consumers for this application. For each possibility, discuss the advantages and disadvantages.

Solution to Question 1

5 points per option (with good motivation). Subtract points if solutions are not essentially different (i.e. check queues in fixed or random order is not really different).

- One queue per consumer
  - Advantages: simple, not much synchronisation needed
  - Disadvantages: if workload not evenly distributed, one can have consumers that are idle for a long time
- Consumers systematically poll all the queues in some fixed (or random) order.
  - Advantages: all queues get elements taken from
  - Disadvantage: if one producer is much more active than others, his queue might fill up quickly.
- Consumers take an element from the queue with the most elements.
  - Advantages: best chance not to fill up queues
  - Disadvantages: possibility of starvation of a single element in a queue, complicated queue management
- Workstealing solution, if a consumer is idle, it looks at its neighbour (or other consumers, in some fixed or random order) and steals work from them.
  - Advantages: less waiting time for tasks in queue, less idle time for consumers.
  - Disadvantage: complex to manage.

Question 2 (10 points)

A `CountdownLatch` is a synchronizer 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. It has two methods:

- **`void`** `await()`, and
- **`void`** `countDown()`.

The `await` method blocks until the current count reaches zero. Each invocation of the countDown() method lowers this count by one. When count reaches 0, all waiting threads are released and any subsequent invocations of `await` return immediately. The `CountDownLatch` is a one-shot phenomenon – the count cannot be reset.

a. (*7 pnts.*) Implement a `CountDownLatch` using locks.

b. (*3 pnts.*) Discuss possible optimisations to improve the performance of your countdown latch implementation.

Solution to Question 2

a. (*7 pnts.*)

```
1  public class CountDownLatch {
2
3      private int counter;
4
5      public CountDownLatch(int n) {
6         counter = n;
7      }
8
9      public void await() {
10        synchronized (this) {
11            if (counter > 0) {
12               try {
13                   wait();
14               }
15               catch (InterruptedException e) {
16               }
17            }
18        }
19     }
20
21     public void countDown() {
22        synchronized (this) {
23            if (counter>0){
24               counter--;
25               if (counter==0) {
26                   notifyAll();
27               }
28            }
29        }
30     }
31 }
```

b. (*3 pnts.*) Make it lock free. Use an AtomicInteger to maintain `count`; `await`: spin as long as `count` is not 0; and `countdown`: use atomicDecrement to lower count by 1.

Question 3 (20 points)

Consider an application that implements a simple flight management system. There are several classes used in this system: `Passenger`, `City`, `Pilot`, and `Flight`. The class `Flight` implements all kinds of methods to manage everything around the flight. Some implementation details are left out, as they are irrelevant for this assignment.

This application contains several concurrency problems. Discuss *five different* problems, and explain *why* they are a problem, for example by discussing an execution that illustrates the problem.

```
1  public class Passenger {
2
3      private String name;
4      private int age;
5      private int miles;
6      private boolean wantsTaxi;
7
8      public synchronized boolean wantsTaxi() {
9          return wantsTaxi;
10     }
11
12     public synchronized String getName() {
13         return name;
14     }
15
16     public synchronized void updateMiles(City from, City to) {
17         miles = miles + from.distanceTo(to);
18     }
19 }
```

```
1  public class City {
2
3      String name;
4
5      public synchronized int distanceTo(City c) {
6          int result = 0;
7          //lookup distance to c in table
8          return result;
9      }
10
11     public synchronized void bookTaxi(Passenger p) {
12         String name = p.getName();
13         // send message to taxi company
14     }
15 }
```

```
1  public class Pilot {
2
3      Flight flight;
4      String name;
5
6      public void assignedToFlight(Flight flight) {
7          this.flight = flight;
8      }
9  }
```

```
1  public enum Status {
2
3          FLYING, TAXYING, LANDED, READYFORTAKEOFF, BOARDING, EMPTY
4
5  }
```

```
1  import java.util.concurrent.locks.Lock;
2  import java.util.concurrent.locks.ReentrantLock;
3
4  public class Flight {
5      Lock pilotLock = new ReentrantLock();
6      private Status status;
7      final int seats;
8      final City from, to;
9      private volatile Passenger reservation[];
10
11     public Flight(int seats, City from, City to){
12         this.seats = seats;
13         this.from = from;
14         this.to = to;
15         this.status = Status.EMPTY;
16         reservation = new Passenger[seats]; }
17
18     //@ requires status == Status.BOARDING;
19     public void boardingCompleted() {
20         status = Status.READYFORTAKEOFF; }
21
22     //@ requires status == Status.READYFORTAKEOFF;
23     public void permissionToLeave() {
24         status = Status.TAXYING; }
25
26     //@ requires status == Status.TAXYING;
27     public void permissionToFly() {
28         status = Status.FLYING; }
29
30     public int available(){
31         int count = 0;
32         for (int i=0; i < seats; i++){
33             if (reservation [i] == null) count++;
34         }
35         return count; }
36
37     public void book(Passenger p){
38         for (int i = 0; i < seats; i++){
39             if (reservation[i] == null) reservation[i] = p;
40             } }
41
42     public void taxiService() {
43         for (int i = 0; i < seats; i++) {
44             if (reservation[i].wantsTaxi()) {
45                 to.bookTaxi(reservation[i]);
46             } } }
47
48     public void updateMiles() {
49         for (int i = 0; i < seats; i++) {
50             reservation[i].updateMiles(from, to);
51         } }
52
53     public void assignPilot(Pilot p) {
54            pilotLock.lock();
55            p.assignedToFlight(this);
56            pilotLock.unlock();
57     } }
```

Solution to Question 3

4 points per correct answer, with a good motivation (max. 20 points). These following 5 problems were knowingly inserted in the program:

- Updates to the elements in the array `reservation` are not atomic, only updates to the reference. Problem for example between lines 29 and 36.
- Deadlock between Passenger and City with calls to bookTaxi and updateMiles in Flight.
- Assigning pilot to flight not protected by lock (only lock in flight acquired)
- Data race on status
- Unlock not in finally clause: if pilot argument is null, an exception will be thrown, and the lock will never be released (the nullpointer exception is a sequential bug, however, the fact that the lock will never be released is a concurrency problem).

Students also correctly identified the following problems:

- Data races between `available`, `book`, `taxiService`, `updateMiles` on `reservations[i]`.
- In `taxiService`, there is an unsynchronised test-and-set pattern: `wantsTaxi` could have been updated before `bookTaxi` is actually called.
- `available` might return wrong answer.

Admittedly: the fields in `Pilot` and `City` should have been private, and the implementation of `book` is broken: it should stop when a passenger has been assigned a seat.

Question 4 (20 points)

Consider the following recursive `Tree` implementation.

```
1  public class Tree {
2
3      private Tree left, right;
4      private int value;
5
6      public Tree(int v, Tree l, Tree r) {
7          value = v;
8          left = l;
9          right = r;
10     }
11
12     public synchronized void insert(int v) {
13         if (left == null) {
14             Tree node = new Tree(v, null, null);
15             left = node;
16         }
17         else {
18             left.insert(v);
19         }
20     }
21
22     public synchronized boolean swapSubtrees(int v) {
23             if (value == v) {
24                 Tree temp = left;
25                 left = right;
26                 right = temp;
27                 return true;
28             }
29             else {
30                 if (left == null && right == null) {
31                     return false;
32                 }
33                 else {
34                     if (left == null) {
35                         return right.swapSubtrees(v);
36                     }
37                     else {
38                         return left.swapSubtrees(v) || right.swapSubtrees(v);
39                     }
40                 }
41             }
42         }
43
44
45     public String toString() {
```

It contains a method `insert`, which inserts a new element as the leftmost element of the tree, and a method `swapSubtrees(V v)`, which swaps the left and right subtree of a node containing the value `v`.

a. (*3 pnts.*) To make the class suited for concurrent usage, all methods are **`synchronized`**. Explain why this is done.

b. (*5 pnts.*) The use of **`synchronized`** in this case can have a negative impact on performance, in particular for large trees. Explain why this is the case, and discuss a lock-based solution that at least partially addresses this problem.
Remark: the intention was to find a solution that would still not allow parallel changes to the tree, but during

correction it was realised that this question was not clear enough. Therefore, this question has been changed into a bonus exercise.

c. (*6 pnts.*) Another option to improve performance is to use lock-free operations. Is it possible to make `insert` lock-free? If so, show how is this done. If not, discuss why not.

d. (*6 pnts.*) Is it possible to make `swapSubtrees` lock-free? If so, show how is this done. If not, discuss why not.

Solution to Question 4

a. (*3 pnts.*) To protect against concurrent accesses and updates in the same tree, to avoid data races.

b. (*5 pnts.*) Tree access might be blocked a long time. Each level of the tree holds a different lock, thus during the recursive calls, each time a new lock is acquired. Solution: one single lock for the tree. Exotic option: variant of lock-coupling list: release lock in parent node when traversing children.
Remark: the solution with holding the lock only locally only works as long as there are no remove operations, unless you use a pattern that is similar to the lock-coupling list. Therefore, no points are awarded for this solution, but instead this exercise has been changed into a bonus question.

c. (*6 pnts.*) Change the references to atomics, remove the synchronized keyword, try to update left by CAS operation with **`null`** as original value. If this fails, this is not the left-most element, make a recursive call `left.insert(v)`.

d. (*6 pnts.*) This is not possible (without introducing an extra protection around the update), as both the left and right pointer should be updated, thus other threads might observe the tree in an inconsistent state.

Question 5 (15 points)

a. (*7 pnts.*) In hardware, stencil operations update the elements of an array by inspecting the neighbouring elements, and computing a new value based on the neigbhouring values, and the current value of the element itself.

Write a kernel that implements a one-dimensional stencil operation on an array `input`, such that a matrix `output` is produced, containing the result of the first 10 iterations of the stencil computation. In each iteration, the new value of `input[i]` is computed as the result of `f(input[i - 1], input[i], input[i + 1])`.

Make sure your kernel is free of data races.

In the boundary cases, the missing argument may be set to 0. Thus, the next value of `input[0]` is computed as `f(0, input[0], input[1])`.

Note: you are free to write your kernel in pseudocode notation, it is not required to use the precise OpenCL syntax. Further, you may assume both `input` and `output` are stored in global memory.

b. (*8 pnts.*) Implement the `takeMVar` and `putMVar` operations for the Haskell type `MVar` using Software Transactional Memory (STM).

```
1  type MVar a = TVar (Maybe a)
2
3  newEmptyMVar :: IO (MVar a)
4  newEmptyMVar = atomically $ do
5  newTVar Nothing
6
7  takeMVar :: MVar a -> IO a
8  takeMVar mvar = -- to be implemented
9
10 putMVar :: MVar a -> a -> IO ()
11 putMVar mvar = -- to be implemented
```

Solution to Question 5

a. (*5 pnts.*)

```
__kernel void exercise(__global int *input, __global int **output) {
   uint tid = get_global_id();

   for (int i = 0; i < 10; i++) {
     int prev = tid > 0 ? input[tid - 1]: 0
     int cur = input[tid];
     int next = tid < get_local_size() ? input[tid + 1]: 0;
     barrier(CLK_GLOBAL_MEM_FENCE);
     input[tid] = f(prev, cur, next);
     output[tid, i] = input[tid];
     barrier(CLK_GLOBAL_MEM_FENCE);
   }
}
```

b. (*8 pnts.*)

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

type MVar a = TVar (Maybe a)

newEmptyMVar :: IO (MVar a)
newEmptyMVar = atomically $ do
  newTVar Nothing

takeMVar :: MVar a -> IO a
takeMVar mvar = atomically $ do
  val <- readTVar mvar
  case val of
    Nothing -> retry
    Just v -> do
                writeTVar mvar Nothing
                return v

putMVar :: MVar a -> a -> IO ()
putMVar mvar v = atomically $ do
  val <- readTVar mvar
  case val of
    Nothing -> writeTVar mvar (Just v)
    Just v -> retry
```

Question 6 (10 points)

For the following cases, explain whether it is possible to have `r1 = r2 = 4` at the end of an execution. Motivate your answer.

a. (*3 pnts.*)

initially `x = y = 3`	
`r1 := x`	`r2 := y`
`y := 4`	`x := 4`

b. (*3 pnts.*)

initially `x = y = 3`	
`lock(m1)`	`lock(m2)`
`r1 := x`	`r2 := y`
`unlock(m1)`	`unlock(m2)`
`lock(m2)`	`lock(m1)`
`y := 4`	`x := 4`
`unlock(m2)`	`unlock(m1)`

c. (*4 pnts.*) Consider two neighbours that have a shared garden. One neighbour has a cat, and the other neighbour has a dog. Whenever the cat and the dog are in the garden together, they end up in fight. Therefore, the two neighbours have agreed on an alternating system, where the cat and the dog are left out in turns. When it is the dog's turn, he is free to enter the garden. When the dog is called in again, it will become the cat's turn, *etc.* Here's an implementation of their system.

```
1  class Garden {
2      private boolean dogsTurn = false;
3      private boolean catsTurn = true;
4      private boolean catInGarden = false;
5      private boolean dogInGarden = false;
6
7      public void letDogInGarden() {
8          while (! dogsTurn);
9          dogInGarden = true;
10     }
11
12     public void callDog() {
13         dogInGarden = false;
14         dogsTurn = false;
15         catsTurn = true;
16     }
17
18     public void letCatInGarden() {
19         while (! catsTurn);
20         catInGarden = true;
21     }
22
23     public void callCat() {
24         catInGarden = false;
25         catsTurn = false;
26         dogsTurn = true;
27     }
28 }
```

Suppose neighbour 1 is implemented as a thread continuously leaving the cat out in the garden, and then calling it in again.

```
1  class Neighbour1 extends Thread {
2
3      Garden g;
4
5      public void run() {
6          while(true) {
7              g.letCatInGarden();
8              // do something else
9              g.callCat();
10         }
11     }
12 }
```

Neighbour 2 is implemented similarly, but leaving the dog out, and later calling it in again.

Discuss, using the notion of memory model, whether the two neighbours succeeded in their effort to make sure that the cat and the dog are never in the garden together. If you think they did not succeed, what would be a way to solve it?

Solution to Question 6

a. (*3 pnts.*) Yes, data race, reordering possible.

b. (*3 pnts.*) No, because of synchronisation.

c. (*4 pnts.*) There are data races on `dogsTurn` and `catsTurn`, thus the compiler might actually set `catInGarden` and `dogInGarden` before. Updates to `catsTurn` and `dogsTurn` might not be visible in the other thread. Solution: make `catsTurn` and `dogsTurn` volatile.

Question 7 (10 points)

Consider the class `ThreeStoreRegister` that defines three registers to store shared data. The `write` operation writes in the first empty slot, the `read` operations reads from the first non-empty slot, and empties it.

```
1  public class ThreeStoreRegister<V> {
2
3      private V r1, r2, r3;
4
5      //@ requires v != null;
6      public synchronized void write(V v) throws RegisterFullException {
7          if (r1 == null) {
8          r1 = v;
9          }
10         else {
11         if (r2 == null) {
12             r2 = v;
13         }
14         else {
15             if (r3 == null) {
16             r3 = v;
17             }
18             else {
19             throw new RegisterFullException();
20             }
21         }
22         }
23     }
24
25     public synchronized V read() throws RegisterEmptyException {
26         V result;
27         if (r1 != null) {
28         result = r1;
29         r1 = null;
30         }
31         else {
32         if (r2 != null) {
33             result = r2;
34             r2 = null;
35         }
36         else {
37             if (r3 != null) {
38             result = r3;
39             r3 = null;
40             }
41             else throw new RegisterEmptyException();
42         }
43         }
44         return result;
45     }
46 }
```

a. (*2 pnts.*) Specify an appropriate `guardedBy` annotation for this class.

b. (*5 pnts.*) Make this class blocking, such that there is no more need to throw the `RegisterFullException` and `RegisterEmptyException`. *Note:* it is not necessary to copy all the code - you can just refer indicate the changes, and refer to the line numbers in the original code, where appropriate.

c. (*3 pnts.*) Discuss a more fine-grained approach to your solution. What would be the advantage of this more fine-grained approach?

Solution to Question 7

a. (*2 pnts.*) `v1, v2, v3 guardedBy this;`

b. (*5 pnts.*) Use wait/notify mechanism.

```
1  public class ThreeStoreRegisterWait<V> {
2
3      private V r1, r2, r3;
4
5      public synchronized void write(V v)  {
6          while (r1 == null && r2 == null && r3 == null) {
7          try {
8              wait();
9          }
10         catch (InterruptedException e) {
11         }
12         }
13         if (r1 == null) {
14         r1 = v;
15         }
16         else {
17         if (r2 == null) {
18             r2 = v;
19         }
20         else {
21             if (r3 == null) {
22             r3 = v;
23             }
24         }
25         }
26         notifyAll();
27     }
28
29     public synchronized V read()  {
30         V result = r1; //default initialisation, because compiler does not see that it always ge
31         while (r1 != null && r2 != null && r3 != null) {
32         try {
33             wait();
34         }
35         catch (InterruptedException e) {
36         }
37         }
38         if (r1 != null) {
39         result = r1;
40         r1 = null;
41         }
42         else {
43         if (r2 != null) {
44             result = r2;
45             r2 = null;
46         }
47         else {
48             if (r3 != null) {
49             result = r3;
50             r3 = null;
51             }
52         }
53         }
54         notifyAll();
```

```
55              return result;
56          }
57      }
```

c. (*3 pnts.*) Use extrinsic lock, and associate two conditions with it: registerNotFull and registerNotEmpty. Then the write operation can wait on registerNotFull, and notify registerNotEmpty, while the read operation wait on registerNotEmpty, and notifies registerNotFull. The advantage is that you do not unnecessarily wake up threads that are waiting for a different condition.