EXAMINATION

Formal Methods and Tools
Faculty of EEMCS
University of Twente

PROGRAMMING PARADIGMS
CONCURRENT PROGRAMMING

code: **201400537**
date: **4 July 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 **5 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!

ANSWERS

Question 1 (30 points)

a. (*11 pnts.*) Suppose you have an existing reentrant lock implementation, providing methods `lock` and `unlock`. Use this lock implementation to implement a (reentrant) read-write-lock. Your implementation should provide the methods `acquireReadLock`, `acquireWriteLock`, `releaseReadLock`, and `releaseWriteLock`. Remember to indicate which fields are protected by which lock. You may assume that methods will be called according to the lock protocol, i.e., a release method will only be called if the lock is actually held.

b. (*6 pnts.*) Sometimes read-write-lock implementations provide an *upgrade* feature: a thread holding a read lock, can upgrade its lock to a write lock. Discuss one advantage, and one disadvantage of such a feature.

c. (*5 pnts.*) The use of read-write-locks could lead to starvation. Discuss why, and suggest (in words) a change to your implementation that would reduce this risk.

d. (*8 pnts.*) Provide a lock-free implementation of a read-write-lock.

Solution to Question 1

a. (*11 pnts.*)

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

public class ReadWriteLock {

    private final Lock l = new ReentrantLock();
    private int counter = 0; // guardedby l;
    private boolean writerActive = false;  // guardedby l;

    // readLock held if counter > 0
    // writeLock held if writerActive
    // lock free if counter == 0 && ! writerActive
    //@ invariant ! ((counter > 0) & writerActive)

    private final Condition lockFree = l.newCondition();

    public void acquireReadLock() throws InterruptedException {
    l.lock();
    try {
        while (writerActive) {
        lockFree.await();
        }
        counter = counter + 1;
    }
    finally {
        l.unlock();
    }
    }

    public void acquireWriteLock() throws InterruptedException{
    l.lock();
    try {
        while (counter > 0 || writerActive) {
        lockFree.await();
        }
        writerActive = true;
    }
    finally {
        l.unlock();
    }
    }

    public void releaseReadLock() {
    l.lock();
    try {
        counter = counter - 1;
        if (counter == 0) {
        lockFree.signalAll();
        }
    }
    finally {
        l.unlock();
    }
    }

    public void releaseWriteLock() {
    l.lock();
    try {
        writerActive = false;
        lockFree.signalAll();
    }
    finally {
        l.unlock();
    }
    }
}
```

Points subtracted: no final (-1), no wait/notify (or await/signal) used: -5, no guardedby (or something equivalent): -2.

b. (*6 pnts.*)

- Advantage (3 pnts): if you upgrade from read to write lock, you know that the data protected by the lock is not changed in between. If you first have to release the read lock, and then acquire the write lock, the data might be changed.
- Disadvantage (3 pnts): risk of deadlock. If two threads hold a read lock try to upgrade to a write lock, they are waiting for the other threads to release their read lock, but will never give up their own read lock.

Argument that performance increases: 1 point only, as you still have to wait to get exclusive write access.

c. (*5 pnts.*) If there are always a few threads holding the read lock, a thread wanting to acquire the write lock might never get a chance to proceed (2 pnts). Possible solution: disallow new threads to obtain the readLock if there is a thread waiting for the writeLock (3 pnts).

d. (*8 pnts.*)

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

public class ReadWriteLockLockFree {

    private AtomicInteger counter = new AtomicInteger(0);

    // readLock held if counter > 0
    // writeLock held if counter == - 1
    // lock free if counter == 0

    public void acquireReadLock() {
    int c;
    do {
        c = counter.get();
        if (c < 0) {
        continue;
        }
    }
    while (!counter.compareAndSet(c, c + 1));
    }

    public void acquireWriteLock() {
    int c;
    do {
        c = counter.get();
        if (c != 0) {
        continue;
        }
    }
    while (!counter.compareAndSet(c, -1));
    }
    public void releaseReadLock() {
    int c;
    do {
        c = counter.get();
    }
    while (!counter.compareAndSet(c, c - 1));
    }

    public void releaseWriteLock() {
    counter.set(0);
    }
}
```

Two atomic integers used: -4 points.

Question 2 (30 points)

In this exercise, we will study the reference implementation for the `LinkedBlockingQueue` as available in Java. The class starts with the following declarations and constructor.

```
    static class Node<E> {
  /** The item, volatile to ensure barrier separating write and read */
      volatile E item;
      Node<E> next;
      Node(E x) { item = x; }
  }

 /** The capacity bound, or Integer.MAX_VALUE if none */
  private final int capacity;

  /** Current number of elements */
  private final AtomicInteger count = new AtomicInteger(0);

  /** Head of linked list */
  private transient Node<E> head;

  /** Tail of linked list */
  private transient Node<E> last;

  /** Lock held by take, poll, etc */
  private final ReentrantLock takeLock = new ReentrantLock();

  /** Wait queue for waiting takes */
  private final Condition notEmpty = takeLock.newCondition();

  /** Lock held by put, offer, etc */
 private final ReentrantLock putLock = new ReentrantLock();

  /** Wait queue for waiting puts */
  private final Condition notFull = putLock.newCondition();

 public LinkedBlockingQueue(int capacity) {
      if (capacity <= 0) throw new IllegalArgumentException();
      this.capacity = capacity;
      last = head = new Node<E>(null);
 }
```

a. (*2 pnts.*) Explain why the locks `putLock` and `takeLock` are declared `final`.

b. (*13 pnts.*) Figure 1 contains the code for the methods `put` and `take` (with helper methods). Discuss what these methods do, and why they do this correctly. In particular, discuss the following:
   - How do these methods guarantee that concurrent execution of `put` and `take` is possible?
   - Which fields are protected by which locks?
   - For all fields that are not protected by a lock, why it is okay that they are not protected by a lock?
   - Why is it okay to use count in the wait guard, even though it is not protected by a lock?

c. (*5 pnts.*) Provide an implementation for the method `signalNotEmpty`, which signals that there is an element available in the queue.

```
private void signalNotEmpty() {
      // body
}
```

Question continued on page 4!

```
private void insert(E x) {
    last = last.next = new Node<E>(x);
}

private E extract() {
    Node<E> first = head.next;
    head = first;
    E x = first.item;
    first.item = null;
    return x;
}

public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    // Note: convention in all put/take/etc is to preset
    // local var holding count  negative to indicate failure unless set.
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly();
    try {
        try {
            while (count.get() == capacity)
                notFull.await();
        } catch (InterruptedException ie) {
            notFull.signal(); // propagate to a non-interrupted thread
            throw ie;
        }
        insert(e);
        c = count.getAndIncrement();
        if (c + 1 < capacity)
            notFull.signal();
    } finally { putLock.unlock(); }
    if (c == 0)
        signalNotEmpty();
}

public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        try {
            while (count.get() == 0)
                notEmpty.await();
        } catch (InterruptedException ie) {
            notEmpty.signal(); // propagate to a non-interrupted thread
            throw ie;
        }
         x = extract();
        c = count.getAndDecrement();
        if (c > 1)
           notEmpty.signal();
    } finally { takeLock.unlock(); }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

Figure 1: Methods `put` and `take` as provided in reference implementation `java.util.concurrent`

d. (*5 pnts.*) Provide an implementation for the method `toArray`, which returns an array with all elements currently stored in the queue.

```
public Object[] toArray() {
     // body
}
```

e. (*5 pnts.*) Give a realistic example how careless use of the two locks could lead to a deadlock.

Solution to Question 2

a. (*2 pnts.*) So that the locks cannot be changed during program execution. This is necessary for thread-safety: correct protection of data, and to make sure that the lock that was acquired, will also be released again.

b. (*13 pnts.*)

- (2 pnts) `put` tries to acquire the `putLock`. If that succeeds, and the buffer is not full, a new node is created, and the the `next` pointer from the old `last` field, and the `last` pointer are set to point to this new node, and the lock is released. At the same time, count (counting the number of elements in the queue) is increased. If there is still space in the buffer, signal a notFull. If the buffer was empty before, signal a notEmpty. If the buffer was full, the thread goes waiting until it receives a notFull signal.
- (2 pnts) `take` tries to acquire the `takeLock`. If that succeeds, and if the buffer is not empty head is moved one forward, and head.item is returned. At the same time, count is decreased. If the buffer is not empty, signal that. If the buffer was full before, signal a notFull.
- (2 pnts) Methods work on disjoint parts of the memory (even for empty list), and therefore can be safely executed in parallel
- (2 pnts) putLock guards last and last.next. takeLock guards head and head.item. Thus they are disjoint, and can be acquired simultaneously.
- (2 pnt) count is accessed by both threads, but this is an atomic field, that is why this is okay.
- (1 pnt) capacity is a read-only field, therefore it does not have to be protected by a lock.
- (2 pnts.) Count in condition works because in the put method, count can only decrease at this point (all other puts are shut out by lock), and the thread trying to execute this put (or some other waiting put) are signalled if there is a change in capacity. Similar, but inverse argument for take.

c. (*5 pnts.*) Essential that it acquires the `takeLock`, otherwise 0 points.

```
/**
 * Signals a waiting take. Called only from put/offer (which do not
 * otherwise ordinarily lock takeLock.)
 */
private void signalNotEmpty() {
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lock();
    try {
        notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
}
```

d. (*5 pnts.*) Essential that both locks are acquired, otherwise 0 points.

```
private void fullyLock() {
   putLock.lock();
   takeLock.lock();
}
```

```
public Object[] toArray() {
    fullyLock();
    try {
        int size = count.get();
        Object[] a = new Object[size];
        int k = 0;
        for (Node<E> p = head.next; p != null; p = p.next)
            a[k++] = p.item;
        return a;
    } finally {
        fullyUnlock();
    }
}
```

e. (*5 pnts.*) There are multiple methods that require both locks (toArray, but also remove from the middle of the queue). If one acquire `putLock` first, then `takeLock` and the other does it in the other order, there is a big risk of deadlock.

Question 3 (10 points)

a. (*6 pnts.*) Consider the following parallel *quicksort* algorithm, implemented in Haskell:

```
force :: [a] -> ()
force xs = go xs `pseq` () where
    go []     = 1
    go (_:xs) = go xs

par_qsort :: (Ord a) => [a] -> [a]
par_qsort []     = []
par_qsort (x:xs) = (force h) `par` ((force l) `par` (l ++ x : h)) where
    l = par_qsort [y | y <- xs, y < x]
    h = par_qsort [y | y <- xs, y >= x]
```

Implement a parallel version of *quicksort* in Haskell using *fork/join*-parallelism. Your implementation should have the following type signature: `fj_qsort :: Ord a => [a] -> IO [a]`, and forks a new concurrent thread for every recursive call.

b. (*4 pnts.*) Explain how the performance and scalability of `fj_qsort` differs from `par_qsort`. Which one is more efficient, and why?

Solution to Question 3

a. (*6 pnts.*) This question is very similar to the `ptreesum2` implementation of the test of June 17, 2016. ForkIO but no joins: 2 points.

```
performwork :: Ord a => [a] -> MVar [a] -> IO ()
performwork []     join = do putMVar join []
performwork (x:xs) join = do
    joinl <- newEmptyMVar
    joinr <- newEmptyMVar
    forkIO (performwork [y | y <- xs, y < x]  joinl)
    forkIO (performwork [y | y <- xs, y >= x] joinr)
    left <- takeMVar joinl
    right <- takeMVar joinr
    putMVar join (left ++ x : right)

fj_qsort :: Ord a => [a] -> IO [a]
fj_qsort xs = do
    join <- newEmptyMVar
    forkIO (performwork xs join)
    result <- takeMVar join
    return (result)
```

b. (*4 pnts.*) The `par_qsort` implementation is *much* faster, as it generates sparks. By using `forkIO`, the `fj_qsort` algorithm is *forced* to construct and destruct threads on every recursive call, but `par` and `pseq` only generate a spark: an opportunity for parallelism. Most sparks are discarded, leaving a more efficient implementation. Any explanation that does not reason about sparks in a similar way is incorrect.

Question 4 (20 points)

Consider the following class `Loop` and its method `shiftLeft`.

```
public class Loop {

    private int[] a = new int[1000000];

    public void shiftLeft () {
        for (int i = 1; i < a.length; i++) {
            int tmp = a[i];
            a[i - 1] = tmp;
        }
    }
}
```

a. (*3 pnts.*) Discuss which dependencies there are between the iterations of the loop (if any). Motivate your answer.

b. (*3 pnts.*) Discuss what sort of OpenMP-like annotations could be used to parallellise the execution of this program.

c. (*5 pnts.*) Write a GPU kernel that has the same behaviour as the method `shiftLeft`.

Now consider the class `Count` and its method `countZeroes`.

```
public class Count {

    private int[] a = new int[100000];

    public int countZeroes() {
        int c = 0;
        for (int i = 0; i < a.length; i++) {
            c = c + (a[i] == 0? 1 : 0);
        }
        return c;
    }
}
```

d. (*4 pnts.*) Discuss how this method could be parallellised by using OpenMP-like annotations.

e. (*5 pnts.*) Write an OpenCL kernel that has the same behaviour as the method `countZeroes`.

Solution to Question 4

a. (*3 pnts.*) Forward loop carried dependency: read to a[i] in line 1 of loop body should happen before write to a[i - 1] in line 2 of loop body of next iteration.

b. (*3 pnts.*) Simd annotation needed

c. (*5 pnts.*) Special care needed for `tid` == 0 (-1 points if no special case for tid=0, missing barrier: -3)

```
__kernel shiftLeft(int [] a) {
  int tid = get_global_id();
  int tmp;
  if (tid > 0) {
    tmp = a[tid];
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
  if (tid > 0) {
    a[tid - 1] = tmp;
  }
}
```

. Alternative: replace tid by tid + 1.

d. (*4 pnts.*) Reduction: all threads share access to c

e. (*5 pnts.*) No atomic_add: -3 points, unnecessary barrier: - 1 points.

```
__kernel countZeroes (int [] a) {

  int tid = get_global_id();
  global int c;

  int tmp = a[tid] == 0 ? 1 : 0;
  c.atomic_add(tmp);

}
```

Question 5 **(10 points)**

a. (*4 pnts.*) Consider the following 2 Rust programs.

Program 1:

```
use std::thread;

fn main () {
    let x = vec![5, 8, 10];
    let t = thread::spawn (move || {
        let y = x;
        return y;
    });
    let y = t.join().unwrap();
    println!("{}", x[0] + y[0]);
}
```

Program 2:

```
use std::thread;

fn main () {
    let x = 5;
    let t = thread::spawn (move || {
        let y = x;
        return y;
    });
    let y = t.join().unwrap();
    println!("{}", x + y);
}
```

One of these two programs will not compile, the other one will. Explain which program will not compile, and why, and for the other program, explain its effect.

b. (*6 pnts.*) Use the message passing mechanism of Rust to implement a distributed counter: the main thread continuously sends messages, while in a separate thread, the receiver counts how many times it has received a message.

Solution to Question 5

a. (*4 pnts.*) Program 1 will not compile. x is a vector, which is not copyable. Ownership of x is passed to the new thread, thus the print statement in the main thread will fail (2 points). Program 2 will execute, and it will print 10. Both x and y are integers, and thus ownership to them is freely copyable (2 points).

b. (*6 pnts.*)

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

fn main () {
  let (tx, rx) = mpsc::channel();
  thread::spawn(move ||{
    let mut c = 0;
    loop {
        rx.recv().unwrap();
        c = c + 1;
        }
    });
```

```
 loop {
   tx.send(1).unwrap();
 }
}
```

No mut: -1 points, no loop: -2 points.