Lecture: Concurrency Theory and Practise
Exercise 1
http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2010ws/

I Theory

I.1 Amdahl's Law

You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a 8-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's Law, explain how you would decide which to buy for a particular application.

I.2 Locks

Programmers at the Flaky Computer Corporation designed the following protocol to achieve n-thread mutual exclusion.

```
1  class Flaky implements Lock {
2     private volatile int turn;
3     private volatile boolean busy = false;
4     public void lock() {
5       int me = ThreadID.get();
6       do {
7          do {
8            turn = me;
9          } while (busy);
10         busy = true;
11      } while (turn != me);
12    }
13    public void unlock() {
14      busy = false;
15    }
16 }
```

For each question, either sketch a proof, or display an execution where it fails.

- Does this protocol satisfy mutual exclusion?
- Is this protocol starvation-free?
- Is this protocol deadlock-free?

I.3 Uncontended Locks

In practise, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock.

Scientists at Cantaloupe-Melon University have devised the following wrapper for an arbitrary lock:

```
1  class FastPath implements Lock {
2     private static ThreadLocal<Integer> myIndex;
3     private Lock lock;
4     private volatile int x,y = -1;
5
6     public void lock() {
7       int i = myIndex.get();
8       x = i;                  // I'm here
9       while (y != -1) {}      // is the lock free?
10      y = i;                  // me again?
11      if (x != i)             // Am I still here?
12         lock.lock();         // slow path
13    }
14
15    public void unlock() {
16      y = -1;
17      lock.unlock;
18    }
19 }
```

They claim that if the base Lock class provides mutual exclusion and is starvation-free, so does the FastPath lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

I.4 Alternative Peterson lock

A way to generalize the two-thread Peterson lock to n threads is to arrange a number of two-thread Peterson locks in a binary tree.

Suppose $n = 2^k$ for some k. Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1.

In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method unlocks each fo the two-thread Peterson locks which the thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration. (This means, threads can take naps, or even vacations, but they do not drop dead.)

For each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated:

- mutual exclusion,
- freedom from deadlock, and
- freedom from starvation.

Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?

I.5 Consistency

Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

I.6 Volatiles

Consider the following code snippet.

```
1  class VolatileExample{
2    int x = 0;
3    volatile boolean v = false;
4    public void writer() {
5      x = 42;
6      v = true;
7    }
8    public void reader() {
9      if (v == true) {
10       int y = 100/x;
11     }
12   }
13 }
```

According to the Java memory model, will the reader method ever divide by zero?

II Practise

II.1 Warm-up

- Write a Java program that spawns n threads, where n is a program argument. These threads access a shared counter (initialized as 0) in a loop. In each iteration, they read the counter to a local (stack) variable, increment it, and store it back to the counter.

  When all threads complete 10000 iterations each, the program stops and prints the value of the shared counter. Note that the final value may be smaller than the total number of iterations.

  Run the program on 1, 4, 8 and 16 threads and report the results in a textual table in some text file.

  Include the run time in miliseconds. You can measure the run time by calling the System.currentTimeMillis() before and after the execution.

- The Java concurrency package, java.util.concurrent, is a library that provides synchronization tools for Java programs. This library includes a built-in mutual exclusion primitive called ReentrantLock.

  Write a Java program doing the same task as in the first part, but protect the shared counter using java.util.concurrent.ReentrantLock, so no two threads modify the counter at the same time.

  Run the program on 1, 4, 8 and 16 threads and report the results in a textual table in some text file. In your report, include the run time in miliseconds.

What can you deduce from the results?

This warm-up exercise will not be corrected, therefore do not submit a solution.

II.2 Dining Philosophers

The dining philosophers problem was invented by E.W. Dijkstra, a concurrency pioneer, to clarify the notions of deadlock and starvation freedom.

Imagine n philosophers who spend their lives just thinking and feasting. They sit around a circular table, each in front of his own plate. However, there are only n chopsticks available. Each philosopher thinks for some time. When he gets hungry, he tries to pick up the two chopsticks that are closest to him. If both chopsticks are available, he takes them and eats for a while. After a philospher finishes eating, he puts down the chopsticks and again starts to think. To not disrupt each others thoughts, philosophers do not communicate with each other and try to chew silently.

Write a program to simulate the behavior of the philosophers, where each philosopher is a thread and the chopsticks are shared objects. Notice:

- You must prevent a situation where two philosophers hold the same chopstick at the same time.
- It must never be the case that a philosopher hold one chopstick and is stuck waiting for another to get the second chopstick, i.e. your program should be deadlock free.

Extend your program in such a way that philosophers never starve.
Details for submission:

- Submit an executable .jar file named philosopher.jar which includes the source files.
- The program should take as parameter the number of philosophers. Calling java −jar philosopher.jar 7 shall simulate seven philosophers sitting around the table.

Submission

- Deadline: 02.11.2010, 13:00
- Send your solutions via email to `heidegger@informatik.uni-freiburg.de`.
- Please submit solutions in teams of 2 or 3 people.
- The solutions to the theory part are to be submitted as .pdf files.
- The solutions to the practical part should be an executable .jar file for each exercise. Make sure that you include all source files and libraries you use. Sources should always be documented!