CPSC 561
Fall 2017

4. October 2017
University of Calgary

Homework Assignment # 1

Due: 16. October 2017, 12:59pm

The Rules: You are allowed to discuss the problems with other students currently enrolled in CPSC 561, or with your TA or instructor. But you are not allowed to use the help of any other person. If you discuss possible solutions with other students enrolled in this course, you must list the names of those students on the cover sheet (see the last page of this assignment). You are also **not allowed to take away any notes made during these discussions**. Instead, you have to write up the solutions entirely on your own.

You can use published literature, as long as you cite that literature in a scientific way, including page numbers, URLs, etc. (whether you use published literature or not will not affect your grade). However, in general it should not be necessary to consult literature other than the course textbook or the lecture notes. In any case, **your solutions must be self-contained**, meaning that a person with knowledge of the course material must be able to follow all your reasoning without referring to any external literature. For example, if you use a (not well-known) statement, say from Wikipedia, which was not covered in the lecture, then you have to provide a proof of that statement as part of your solution.

If you are unsure whether a certain source can be used, **please ask your instructor**.

Please fill out the cover sheet (see the last page) and hand it in with your solutions.

Question 1

Some time ago, the following algorithm was published in a scientific journal as a proposed solution to the mutual exclusion problem for a system with two processes with IDs 0 and 1:

Class `FunnyLock`

shared:

`boolean` $flag[0 \dots 1] = (\texttt{False}, \texttt{False})$

`int` $turn = 0$

Method $\texttt{lock}_i\texttt{()}$

```
1 flag[i].Write(True)
2 while turn.Read() ≠ i do
3 |   while flag[1 − i].Read() = True do
  |   |   /* do nothing                                    */
4 |   end
5 |   turn.Write(i)
6 end
```

Method $\texttt{release}_i\texttt{()}$

```
7 flag.Write(False)
```

(a) Show that the object `FunnyLock` does not satisfy mutual exclusion.

(b) Does it satisfy deadlock-freedom? Either give a proof or a counter example (i.e., an execution in which deadlock-freedom is violated).

Question 2
Consider the following mutual exclusion object for n processes:

Class `LousyLock`

shared:
`int` $flag[0 \ldots n-1] = (0, \ldots, 0)$
`int` $turn = 0$

Method `lock`$_i$`()`

```
1 flag[i].Write(1)
2 while turn.Read() ≠ i do
3 |   if flag[turn].Read() = 0 then turn.Write(i)
4 end
5 flag[i].Write(2)
6 for j = 0, ..., n − 1 do
7 |   if j ≠ i ∧ flag[j].Read() = 2 then goto line 1
8 end
```

Method `release`$_i$`()`

```
9 flag[i].Write(0)
```

(a) Prove that `LousyLock` satisfies mutual exclusion.

(b) Prove that `LousyLock` satisfies deadlock-freedom.

(c) Does `LousyLock` satisfy starvation-freedom? Justify your answer.

Question 3

(a) Implement a deadlock-free long-lived lock in a Java class `LLLock`, which supports the methods `lock()` and `unlock()` for n processors, where n is any power of 2.

You cannot use objects from the `java.util.concurrent.atomic` package.

(b) Implement a deadlock-free one-time lock in a Java class `OTLock`, which supports only the method `lock()` for n processors, where n is any power of 2. It must satisfy the following safety-property: If all `lock()` calls terminate, then exactly one of the calls returns `True`, and all other calls return `False`. In addition to being deadlock-free your implementation should have the following progress-property: Any `lock()` method call is wait-free if it gets invoked after some other `lock()` method call has already finished.

You cannot use objects from `java.util.concurrent.atomic` package.

(c) The goal is to implement several Java programs, as specified below, which take as input two integers L and n, and output all primes in $\{\lfloor L/2 \rfloor, \ldots, L\}$ using n threads, where n is a power of 2. The algorithms should follow the outline discussed in the lectures (see also Figure 1.2 on page 4 of the textbook). To test whether an integer is a prime, use the Java class `isPrime`, provided with this assignment.

- **Variant 1** uses a shared counter implemented from a register and protected by your long-lived lock from Question 1.
- **Variant 2** uses an instance of your one-time lock from Question 2 for each number in $\{\lfloor L/2 \rfloor, \ldots, L\}$. A process runs the primality test only after locking the number.
- **Variant 3** is lock-free and uses the Java class `AtomicInteger` to realize a shared counter.

(d) Run several experiments with the three algorithms implemented, and time how long it takes until all numbers have been tested for primaility. Run each variant for all values of $n \in \{1, 4, 8, 16, 32\}$. Choose L large enough to obtain expressive results (e.g., Variant 3 with one thread should take 10 to 60 seconds to complete, depending on how patient you are). For such large values of L, Variant 2 may not complete—why?

Fill in the spreadsheet provided with this assignment with the results of your experiments and briefly interpret your results.

You can get the current time of the system in milliseconds using `System.currentTimeMillis()` and in nanoseconds using `System.nanoTime()`.

Cover Page for CPSC 561
Fall 2017
Homework Assignment # 1

Name: _____

Course Section: _____

Collaborators:

Question 1: _____

Question 2: _____

Question 3: _____

Question 4: _____

Other Sources:

Question 1: _____

Question 2: _____

Question 3: _____

Question 4: _____

Declaration:

I have written this assignment myself. I have not copied or used the notes of any other student.

Date/Signature: _____