CPSC 3220 – Quiz #5 February 28, 2018 Name: __**Answer Key**__

1.(2 points) True / False - If a multi-threaded program runs correctly in all cases on a single time-sliced processor, then it will run correctly if each thread is run on a separate processor of a shared-memory multiprocessor. Justify your answer.

True
To run correctly in all cases on a single processor implies that the program's threads can execute in any interleaving — that is, they are independent of the order of instruction execution.

```
1:      void RWLock::donwRead() {
2:              lock.acquire();
3:              activeReaders--;
4:              if (activeReaders == 0 && waitingWriters > 0) {
5:                      writeGo.signal();
6:              }
7:              lock.release();
8:      }
```

2. (2 points) In the readers / writers lock example for the function RWLock:doneRead (above), why do we use writeGo.Signal() rather than writeGo.Broadast()?

When a read finishes at most one writer can make progress and any of the writers can make progress. So, broadcast is wasteful and unnecessary.

3. (2 points) How can a semaphore be used as a mutual exclusion lock?

Create/Initialize the semaphore to 1.

4. (2 points) How do semaphores differ from locks & condition variables in synchronization?

A semaphore has memory and condition variables have no memory
A semaphore can have any non-zero positive integer value and locks are binary.

Continued on Back

Consider the following simple implementation of a hybrid user-level/kernel-level lock.

```
1:  class TooSimpleFutexLock {
2:      private :
3:          int val;
4:      public :
5:          TooSimpleMutex() : val (0) { } // Constructor

6:          void acquire () {
7:              int c;
8:              // atomic_inc returns ←old← value
9:              while ((c = atomic_inc (val)) != 0) {
10:                 futex_wait (&val , c + 1);
11:             }
12:         }

13:         void release () {
14:             val = 0;
15:             futex_wake (&val , 1);
16:         }
17: };
```

5. (2 points) The goal of this code is to avoid making expensive system calls in the uncontested case of an acquire on a FREE lock or a release of a lock with no other waiting threads. This code fails to meet this goal at LINE 15. Why?

Release always makes a system call, even if there are no waiting threads.

6. (2 points) A corner case (Lines 9-10) can cause the mutual exclusion correctness condition to be violated, allowing two threads to both believe they hold the lock. What is the problem?

val can be repeatedly incremented by each thread, so it is possible for it to wrap around to 0, allowing multiple threads to simultaneously believe they have acquired the lock.