Problem Set #3

ECE 3140/CS 3420

Due Friday April 20th 2018, at 5pm

1. Time sharing

Draw a state transition diagram for process scheduling. Recall a process can have three states,

- Ready
- Waiting/ Suspended/ Blocked
- Running

and mark state transitions with arrows. Also use arrows to mark process entry and the exit.

2. Mutual exclusion

The following code is called *Peterson's synchronization algorithm* for two processes *{P0, P1}*. (Only P0's code is shown; P1's is symmetric.)

```
P0: while(1) {
      intent[0] = false;
      // Non-critical section
      // ...
      // End of non-critical section
      intent[0] = true; // P0 wants
access to CS
      turn = 1; // Make it P1's turn
      while (intent[1] && turn == 1) {
        // Busy-wait
      }
      // Critical section
      // ...
      // End of critical section
    }
```

```
P1: while(1) {
      intent[1] = false;
      // Non-critical section
      // ...
      // End of non-critical section
      intent[1] = true; // P1 wants
access to CS
      turn = 0; // Make it P0's turn
      while (intent[0] && turn == 0) {
        // Busy-wait
      }
      // Critical section
      // ...
      // End of critical section
    }
```

Explain whether this algorithm provides *(a)* safety, *(b)* progress, and *(c)* fairness. Assume code sections are of nontrivial length. You may comment the code above as needed.

Outline a method to extend this algorithm to more than 2 processes.

3. Locks

Construct a synchronization primitive called *ticket lock* out of simple locks. Ticket locks work like those counters with an electronic board to show which ticket is served at present: You grab a ticket, then you wait until your number is called, then you get served. Comment on the fairness of the implementation.

```
// globally shared variables
int simple_lock;              // use as needed for correctness
```

```
int next_available;           // next available ticket
int currently_serving;        // ticket currently being served

// lock/unlock function prototypes
void lock(int *lock_ptr);     // simple atomic t&s-based lock
void unlock(int *lock_ptr);   // simple atomic unlock

void ticket_lock() {
  int my_ticket;
                              // grab a ticket
                              // wait for my ticket to be called

}

void ticket_unlock() {

                              // yield to next ticket's holder
}
```

4. Semaphores and Condition Variables

A barrier is a commonly used group synchronization mechanism. A program is divided into multiple phases and no thread may proceed into the next phase until all threads are ready to proceed to the next phase. In other words, all threads need to reach the barrier before any thread continues on. The behavior may be achieved by letting each thread execute barrier() at the end of each phase. Assume that there are N threads in the program, a call to barrier() should block until all of the N threads have called barrier(). Then all threads proceed.

(a) Show an implementation of barrier() using semaphores. Here is the basic idea.

- Use a shared variable 'counter' to represent the number of thread that reached the barrier and are waiting in the barrier() function.
- Use one semaphore to protect the 'counter' variable. This works as a lock to ensure that only one thread can check and increment the counter at a time.
- Use another semaphore to block threads in the barrier until all threads reach the barrier.

(b) Show an implementation of barrier() using condition variables. This implementation is similar to the above one in a sense that it uses a 'counter' variable to keep track the number of threads in the barrier. But instead of semaphore, use a lock and a condition variable to protect the counter and block/unblock threads.