# CS 162 Section 2 Spring 2014

## True/False

1. Preemptive multithreading requires threads to give up the CPU using the `yield()` system call
   **False –** Preemptive multithreading using interrupts to schedule context switches
2. In some cases, two threads can both be executing a specific critical section at the same time
   **False –** A critical section can only be executed by one thread at a time.
3. Every interrupt results in a transition from user to kernel mode. Hint: think Inception
   **False –** Consider the scenario where an interrupt occurs while the machine is servicing the current interrupt (in kernel mode, transitioning to kernel mode)
4. A thread needs to own a semaphore, meaning the thread has called `semaphore.P()`, before it can call `semaphore.V()`
   **False –** Semaphores are commutative, so V() can be called before P() by any thread with a reference to the semaphore.
5. A thread needs to own the monitor lock before it can `signal()` a condition variable.
   **True –** A thread must acquire the monitor lock before it can access any of the monitor's state or call any of its methods.

## Short Answer

1. With spinlocks, threads spin in a loop (busy waiting) until the lock is freed – in general this is a bad idea and wastes many CPU cycles. However, there are certain cases where a spinlock is more efficient than a blocking lock (puts waiting thread to sleep). When?

   If the expected wait time of the lock is very short (such as because the lock is rarely contested or the critical sections are very short), then it is possible that the spin lock will waste fewer cycles than putting the threads to sleep/waking them up. The important tissue is that the expected wait time must be less than the time to put a thread to sleep and wake it up.

2. What are some similarities and differences between interrupts and system calls?

   Interrupt: electronic signal to the processor from an external device indicating that an external event needs attention.

   System calls: also referred to as software interrupts or synchronous interrupts (as opposed to asynchronous hardware interrupts). Special instruction that causes a transition from user to kernel mode when executed. Examples: fork(), open(), etc. (more about these in project 2).

## Locks

Consider the two spinlock implementations below. Are they correct? If not, give an example scenario that would cause the lock to break.
Assume a system with two threads, and in #2 this_thread and other_thread correspond to integer thread IDs.

1.

```
struct lock {        void acquire(lock) {         void release(lock) {
     int held = 0;         while(lock->held);          lock->held = 0;
}                          // context switch here!
                           lock->held = 1;       }
                      }
```

Context switching at the commented-in line will cause multiple threads to "hold" the lock.

2.

```
struct lock {          void acquire(lock) {
     int turn = 0;          while(lock->turn != this thread);
}                      }

void release(lock) {
     lock->turn = other_thread;
}
```

This implementation of locks would function correctly, given a pre-emptive scheduler and two threads with the same priority. If there was a priority schedule and the two threads had different priorities (more on this in Project 1), we would need priority donation to ensure no starvation.

## Semaphores

Implement the P() and V() methods of a Semaphore class backed by monitors (i.e. the Lock and CondVar classes).
Neither of the methods should require more than five lines.
Assume **Mesa**-scheduled monitors.

```
public class Semaphore {
      Lock lock; // every monitor has a Lock and CondVar
      CondVar c;
      Int value;  // semaphores have a positive integer value

public Semaphore(int initialValue) {
      value = initialValue;
      lock = new Lock();
      c = new CondVar(lock);
}

public void P() {
/* YOUR CODE HERE */
      lock.Acquire();
      while (value == 0)
            c.Wait();
      value--;
      lock.Release();
}
```

```
public void V() {
/* YOUR CODE HERE */
      lock.Acquire();
      value++;
      c.Signal();
      lock.Release();

}
```

How would the implementation change for **Hoare**-scheduled monitors?

```
public void P() {
/* YOUR CODE HERE */
      lock.Acquire();
      if (value == 0)
            c.Wait();
      value--;
      lock.Release();
}
```

The "while" would change to an "if" because the waiting thread would **immediately** be given access to the resource and CPU time.