CS 162 Section 2 9/17-9/18

True/False

1. Preemptive multithreading requires threads to give up the CPU using the yield() system call.
2. Despite the overhead of context switching, multithreading can provide speed-up even on a single-core CPU.
3. In some cases, two threads can both be executing a specific critical section at the same time.
4. Synchronization is achieved with strictly software-level support.
5. Every interrupt results in a transition from user to kernel mode. Hint: think Inception.

Short Answer

1. With spinlocks, threads spin in a loop (busy waiting) until the lock is freed. In general, spinlocks are a bad idea because they can waste a lot of processor cycles. The alternative is to put a waiting process to sleep while it is waiting for the lock (using a blocking lock). However, there are certain cases where a spinlock is more efficient than a blocking lock. Give a circumstance in which this is true and explain why a spinlock is more efficient.

2. Give two reasons why this is a bad implementation for a lock:

   lock.acquire() { disable interrupts; }
   lock.release() { enable interrupts; }

3. What are some of the similarities and differences between interrupts and system calls?

Locks

Are these two implementations of spinlocks correct? If they aren't, give a scenario in which they wouldn't work. Assume that the system only has two threads. For #2, assume that this_thread and other_thread are ints corresponding to the thread IDs (of the current thread and the thread that isn't running).

1.

```
struct lock {
        int held = 0;
}

void acquire(lock) {
        while (lock->held);
        lock->held = 1;
}

void release(lock) {
        lock->held = 0;
}
```

2.

```
struct lock {
        int turn = 0;
}

void acquire(lock) {
        while (lock->turn != this_thread);
}

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