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. False. Preemptive multithreading uses interrupts to schedule context switches.
2. Despite the overhead of context switching, multithreading can provide speed-up even on a single-core CPU. True. Context switch to avoid blocking on I/O.
3. 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.
4. Synchronization is achieved with strictly software-level support. False. Synchronization is aided by hardware-level primitives.
5. Every interrupt results in a transition from user to kernel mode. Hint: think Inception. False. Another interrupt can occur while servicing the current interrupt.

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.

   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 a spin lock will waste many fewer cycles than putting threads to sleep/waking them up. The important issue is that the expected wait time must be less than the time to put a thread to sleep and wake it up.

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

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

1) It prevents hardware events from occurring during the critical section
2) User programs cannot use this lock
3) It doesn't work for data shared between different processors.

3. What are some of the 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.

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 thread IDs.

1.

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

void acquire(lock) {
        while (lock->held);
        //can context switch right here, and multiple threads end up holding the lock.
        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
}
```

Given a pre-emptive scheduler and two threads with the same priority, this locking system would work. However, if there was a priority scheduler and the two threads had different priorities (as you'll see in Project 1), you would need a priority donation to take place to ensure that starvation does not occur.