Midterm 2 solutions

```
/* Question 1 - mutexes */

typedef struct _mutex {
    int lock; //initialized to 0
    queue_t queue; //initialized to a valid empty queue
    thread_t owner; //initialized to NULL
} *mutex_t;

void mutex_lock(mutex_t mutex) {
    assert(mutex);
    while(mutex->lock) {
//the exact details here are unimportant, the idea is
//to put current thread onto mutex->queue and yield
//here is one possible solution
        thread_t temp = current_thread;
        current_thread = thread_dequeue(ready_queue);
        thread_enqueue(mutex->queue, temp);
        thread_yield();
    }
    mutex->lock = 1;
    mutex->owner = current_thread;
}

void mutex_unlock(mutex_t mutex) {
    assert(mutex);
    if(mutex->owner != current_thread) {
        fprintf(stderr, "Error: unlocking thread doesn't own mutex");
        assert(0);
    }
    if(! thread_queue_empty(mutex->queue)) {
//here, pull a waiting thread off the queue and schedule it to run
        thread_t th = thread_dequeue(mutex->queue);
        assert(th);
        thread_enqueue(ready_queue, th);
    }
    mutex->owner = NULL;
    mutex->lock = 0;
}
```

```
/* Question 2 - spinlocks */

typedef struct _spinlock {
    int lock; //initialized to 0
    thread_t owner; //initialized to NULL
} *spinlock_t;

void spinlock_acquire(spinlock_t s) {
//check for errors
    assert(s);
    while(test_and_set(&s->lock)); //spin until the lock is atomically taken
once free
        assert(s->owner == NULL);
        s->owner = current_thread;
    }

void spinlock_release(spinlock_t s) {
//check for errors ...
    assert(s);
    if(s->owner != current_thread) {
        fprintf(stderr, "Error: unlocking thread doesn't own spinlock");
        assert(0);
    }
    s->owner = NULL;
    s->lock = 0; // ... and release the lock
}
```

Question 3.

1. Give two compelling reasons why a program might use threads.

Two primary reasons are multiprocessors and I/O concurrency. A secondary reason is using threads as a program structuring tool.

2. Consider the following three program fragments found within the same multithreaded program.

   fragment 1:
   ```
   P(s1);
   P(s2);
   do_stuff();
   V(s2);
   V(s1);
   ```

   fragment 2:
   ```
   P(s2);
   P(s1);
   do_stuff();
   V(s1);
   V(s2);
   ```
   SWAP THE ORDER OF P(S1) AND P(S2)

   fragment 3:
   ```
   P(s1);
   P(s2);
   do_stuff();
   V(s2);
   V(s1);
   ```

   When this program runs, it deadlocks. While still providing for mutual exclusion using the two semaphores s1 and s2, eliminate the deadlock condition by making the fewest possible changes to the code above.

3. In a priority-based scheduling system, explain why we assign I/O bound tasks higher priority than CPU bound tasks?

To increase CPU and device utilization and avoid the CONVOY EFFECT. The convoy effect is when all processes wait for one big process to get off the CPU.

4. Give two important reasons why programs can't simply disable interrupts to achieve mutual exclusion.

Multiprocessors
Can't disable interrupts and run a user program

5. Show the proper way to wait on a condition variable having Hoare semantics?

if (condition) wait

6. From an implementation perspective, what is the principle advantage of Mesa monitor semantics?

They are less strict, programmer can be lazier, sloppier.
INCORRECT ANSWER: EASIER TO IMPLEMENT

7. Monitors are often preferable to semaphores for two reasons. What are the reasons?

Separation of scheduling and mutual exclusion
Language level support