1. (25 pts) **Short answer questions.**

(a) (5 pts) You have a threaded application that would benefit from a different scheduling algorithm than the one your operating system implements. Would you prefer to use kernel or user threads to implement your scheduling algorithm and why?

**Solution:** User threads, since then I can write my scheduler without modifying the OS, making it much easier to test and use. Also correct is user threads because they are quicker.

(b) (5 pts) Which of the scheduling algorithms we covered in class is the fairest? What effect does fairness have on response time?

**Solution:** Round robin (RR). Fairness always decreases average response time.

(c) (5 pts) Compare using the Test&Set instruction versus disabling and enabling interrupts to implement locks (or any other high-level synchronization construct).

**Solution:** Test&Set requires busy waiting and it must be supported in hardware. However, it is easier to get right and programming errors will not lead to system deadlock. With disabling and enabling interrupts, we don’t get busy waiting, but if make a programming error the system may deadlock. Also, we must be concerned with the length of time we postpone other system activities.

(d) (5 pts) Will a monitor that executes correctly using Mesa-style semantics execute correctly if Hoare-style semantics are used instead. Why or why not?

**Solution:** Yes. Mesa-style semantics is more general since after the signal the signaler keeps the lock, and then any of the participating processes can grab the lock. With Hoare-style semantics, the signaler directly gives the lock to the waiter. (Typically Hoare-style code can use “if” tests on the condition and Mesa-style uses a “while”.)

(e) (5 pts) If we want to use semaphores within a monitor instead of condition variables, what problems might this cause and how could we solve them?

**Solution:** Semaphores don’t release the lock on a wait and thus could result in deadlock if used inside a monitor. To solve this problem, we would have to explicitly release the lock before doing a semaphore.wait and then reacquire it after the wait. We need not change the semaphore.signals as long as we are satisfied with Mesa-style semantics.
2. (25 pts) **Scheduling.**

(a) (5 pts) Name the five scheduling algorithms we considered in class.

**Solution:** Round Robin (RR), First-In-First-Out (FIFO), Shortest Response Time First (SRTF), Multilevel Feedback Queues (MLFQ), and lottery.

(b) (10 pts) Pick 2 scheduling algorithms and a job mix with 2 or more jobs that have the same average response time.

**Solution:**

<table>
<thead>
<tr>
<th>Job</th>
<th>length</th>
<th>Completion Time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SRTF</td>
<td>FIFO</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Avg. RT</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

(c) (10 pts) Pick another 2 scheduling algorithms and a job mix with 2 or more jobs that yield the same average response time. (Note, you can only overlap at most one scheduling algorithm with part b.)

**Solution:**

<table>
<thead>
<tr>
<th>Job</th>
<th>length</th>
<th>Completion Time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RR</td>
<td>MLFQ</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Avg. RT</td>
<td>8.5</td>
<td>8.5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue</th>
<th>Time Slice</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$0_1^1$, $1_2^1$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$0_4^3$, $1_6^3$, $0_7^3$, $1_10^6$</td>
</tr>
</tbody>
</table>
3. (25 pts) Processes & Threads.

(a) (5 pts) Why is a context switch faster for user threads than for kernel threads?

Solution: Kernel threads must enter the kernel first before storing the control variables (PC, SP, registers) for the current thread, and loading the new thread’s control variables, and then leave the kernel. In user threads, the control variables are stored and the new ones loaded - there is no switching to and from kernel mode.

(b) (10 pts) What happens on a context switch between independent processes? (In your Solution:, discuss all the affected OS data structures and include a drawing of the process state diagram and discuss the possible transitions.)

Solution: The OS stores the hardware context (i.e., PC, SP, HP, registers, memory management information, etc.) of the current process from the CPU in the process’s PCB. The OS must change process state from running to either terminated, waiting, or ready depending on the reason for the context switch.

FIGURE MISSING (SORRY)

The OS then selects a process \( p_i \) from the ready queue, changes its state from ready to running, and loads its hardware context on to the CPU.

(c) (10 pts) Is there any reason to use threads if the scheduler is non-preemptive (i.e., context switches may occur only inside blocking synchronization primitives, in response to an explicit request to yield the processor, or in response to an I/O request)?

Solution: Yes, of course. We don’t need a time slice to get multiprogramming benefits for cooperating threads. Synchronization primatives like wait and I/O events give us these benefits as well.
4. (25 pts) **Semaphores & Monitors.** You are given a game with many players and 3 colors (red, blue, green). You must ensure that the players only move pieces in the order: red, blue, green, red, blue, green, etc. Write three routines for the players to call: MoveRed, MoveBlue, and MoveGreen. Remember to write the initialization routines, and you may assume red always starts.

(a) (10 pts) Write a semaphore Solution:

**Solution:**

```cpp
class Game {
    public: RedCS(), BlueCS(), GreenCS();
    private: Semaphore Red, Blue, Black;
}
Game::Game(){
    Red.value = 1; // Red Semaphore is available
    Blue.value = 0; // Blue Semaphore is not available
    Green.value = 0; // Green Semaphore is not available
}
Game::RedCS() {
    Red.Wait();
    ... red move ... 
    Blue.Signal();
}
Game::BlueCS() {
    Blue.Wait();
    ... blue move ...
    Green.Signal();
}
Game::GreenCS() {
    Green.Wait();
    ... green move ...
    Red.Signal();
}
```
(b) (15 pts) Write a monitor Solution:

**Solution:**

```cpp
class GameMonitor {
    public: RedCS(); BlueCS(), GreenCS();
    private:
        ConditionVar Red, Blue, Green;
        int turn = {red, blue, green};
        Lock lock;
}
GameMonitor::GameMonitor() {
    lock = FREE;
    turn = red;
}
GameMonitor::RedCS() {
    lock.Wait();
    while (turn != red) {
        Red.Wait();
    }
    ... red move ... 
    turn = blue;
    Blue.Signal();
    lock.Signal();
}
GameMonitor::BlueCS() {
    lock.Wait();
    while (turn != blue) {
        Blue.Wait();
    }
    ... blue move ... 
    turn = green;
    Blue.Signal();
    lock.Signal();
}
GameMonitor::GreenCS() {
    lock.Wait();
    while (turn != green) {
        Green.Wait();
    }
    ... green move ... 
    turn = red;
    lock.Signal();
}
```