Chris Kasper
CS 370-001
Homework Set #2

Chapter 5

1. If a compute-bound process (it never does I/O) takes T seconds to run, will the time taken for n such processes be less than, equal to, or greater than nT on a realistic system that does round-robin scheduling? Why?

Theoretically, if the realistic system has round-robin scheduling, and only has the compute-bound processes in the process queue, it should take nT (time) to finish all the processes. While each process will take turns computing, each will have an equal fraction of the CPU time.

4. Among the mutual exclusion methods, disabling interrupts, test-and-set instructions, and semaphores, which can be used by user processes, and which should be reserved for use by the operating system? Why?

Disabling interrupts should be reserved for use by the operating system because if any process could disable interrupts then erroneous or malicious processes could disable them and never reenable them. Test-and-set instructions are not ideal for user processes as they protect critical sections of code by just making the process wait until it retrieves the lock, also known as busy waiting. Semaphores on the other hand, are more ideal for user processes because they are simpler (integer value object) higher level technique that does not have busy waiting.

5. Describe how disabling interrupts prevents two processes from entering the critical section at the same time.

For a process to be preempted, an interrupt must occur and be serviced. With disabling interrupts, a process can lock a critical section and make sure it won't be preempted while in it.

8. In Example 5.6, suppose that process a only uses 75% of each of its slots, process b uses 80% of each of its slots, process c uses 20% of its slot, and process d uses 15% of its slot. What is the efficiency of the overall scheduling cycle? Ignore any scheduling and context switching overhead.

Process a usage (in mS) => (0.75 * 10 mS) * 5 = 37.5 mS
Process b usage (in mS) => (0.80 * 10 mS) * 3 = 24 mS
Process c usage (in mS) => (0.20 * 10 mS) * 1 = 2 mS
Process d usage (in mS) => (0.15 * 10 mS) * 1 = 1.5 mS

Total process usage = 65 mS (0.65% efficiency of the 100 mS long cycle)

9. Suppose we have a multilevel feedback queue such as the one illustrated in Figure 5-5 with three processes, a, b, and c, where each has a base priority of 1 and where each process moves down one level on completing a time slice. Let p(a) be boosted to 4, p(b) to 3, and p(c) to 2. How many time slices will expire before all three processes are back to their base priority? How many will each process get in the process?

Using tables to illustrate the process

Time Slice 1

4	**A**
3	B
2	C
1	

Time Slice 2

4	
3	**B** - A
2	C
1	

Time Slice 3

4	
3	**A**
2	C -> B
1	

Time Slice 4

4	
3	
2	**C** -> B -> A
1	

Time Slice 5

4	
3	
2	**B** -> A
1	C

Time Slice 6

4	
3	
2	**A**
1	C -> B

Final State (would be time slice 7)

4	
3	
2	
1	**C** -> B -> A

Process A time slices: 3
Process B time slices: 2
Process C time slices: 1

6 time slices total till each has a base priority of 1

13. Show that an atomic exchange (between a register and a memory location) instruction can be used to implement a lock equivalent to the one implemented with the *tas* instruction.

When acquiring a lock via an atomic exchange instruction, a register gets set to a value, and then is exchanged for a value (the lock value) in memory. Depending on the new register value (0 or 1), the process will find out whether or not it retrieved the lock.

15. Suppose we adjust a process's priority by $p' = \alpha p$ for each time slice where it is running and by $p' = 1 - \alpha(1 - p)$ when it is blocked. (There is no change for processes that are ready but not running.) If $0<p<1$ and $0<\alpha<1$, is it possible for a ready process to ever starve? Why or why not?

With these priority adjusting equations, it is not possible for a ready process to starve. This is because of the p and α possible ranges (importantly, α can't be 0). There is at least a guarantee that each blocked process will get an increased priority each time it gets adjusted.

Chapter 7

2. In the per-instruction loop of *xec*(), why can't we call the instruction execution function with the following line?

optab [R.PC→op]( );

The reason why is because before we run the *optab* function pointers, the current instruction gets decoded, the op value for the instruction is saved to a variable, and the program counter is incremented. If you run *optab [R.PC→op]( )* instead of *optab [op]( )*, you will be trying to run the next instruction (with no decoding), rather than the current one (which has been decoded).

4. In the infinite scheduling loop of *vmachine*(), we check only the *vmq* list to see if there are any processes on it. Why do we not also look at the *idlevmq* list?

Idlevmq is a list of idle (have no associated user processes) kprocs.

5. Why do you suppose that the *isched* structure has a pointer to the tail of the *vmq* list and not to the tail of the *idlevmq* list?

There is no need to have a pointer to tail of *idlevmq*. In the situation where the *vmq* list has only one ready kproc, it can possibly assign an idle process from *idlevmq* (the head which should be moved first) to the tail of the *vmq* list.