Jonathan Giraud

CMSC 312 Project 2

1) Implement the counting semaphores using binary semaphores. Pseudo-code for the

implementation is given in the attached notes on BB: 10xcountingSemUsingBinarySem.pdf

Use the bounded-buffer producer-consumer problem as uploaded on BB (first code at

<https://jlmedina123.wordpress.com/2014/04/08/255/> Show the output of this code for

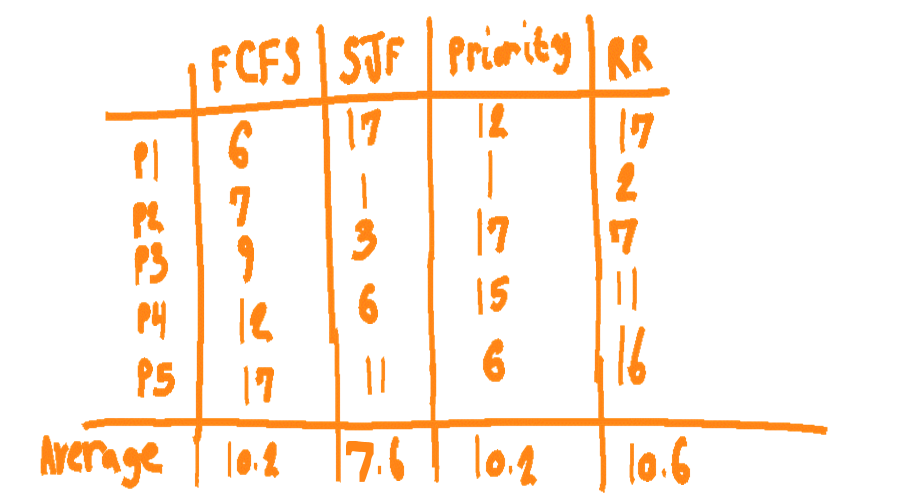
BOTH the incorrect and correct implementations. Also, attach your codes (two versions:

for incorrect and correct solutions) on BB. (10+10 points)

2) Prove/disprove that SJF is optimal in terms of average turnaround times of the processes.

(2 points)

**The SJF IS the optimal choice in scheduling to achieve the shortest average turnaround time. The scheduler specifically puts the shorter jobs first and orders them to run in ascending order of their length. For example, if we take the results from question 10:**

****

**We can see that the SJF yields the shortest turnaround time average. This is because the shortest processes run first, meaning they will run their entirety and exit first and faster over longer processes. This means there will be shorter turn around times more often than not, as shown in the chart. SJF has more consistently smaller numbers and turnaround times, resulting in an optimal turn around time. Average Turn around time is from: Sum Of ((Time Process Completed – Time Process Arrived))/ (Number of Processes) And since they are ordered from smallest process to largest, the smaller processes will complete faster relative to their arrival time and will pull down the average time and impact it more than larger processes.**

3) For what types of workloads does SJF deliver the same turnaround times as FIFO? (2

points)

**Shortest Job First has the same turnaround time as First In First Out if the jobs arrive in the order of smallest job to largest job. So if the first job that arrives has length 1, second has length 2, third has length 3, SJF will have the same turnaround time as FIFO. This is true if all of the jobs have the same length too.**

4) For what types of workloads and quantum lengths does SJF deliver the same response

times as RR? (2 points)

**Shortest Job First must have a workload where the jobs arrive in the order of smallest job to largest job OR if all of the jobs have the same length. The quantum length for Round Robin must be equal to or longer than the longest job scheduled. This would make Round Robin non-preemptive so it matches SJF in response time.**

5) What happens to response time with SJF as job lengths increase? (2 points)

**When job length increases, so will the average response time. Assuming SJF is ordered from smallest job to largest job, then the response time effectively stacks upon itself. If the time of the first job is increased, then the response time for the job scheduled after that will be higher. And as a result, the response time for the NEXT job scheduled after that will also be higher.**

6) What happens to response time with RR as quantum lengths increase? Explain in words

and write an equation that gives the worst-case response time, given N jobs. (2 points)

**Well the quantum time designates an amount of cycles for a process to be completed. If it can’t be completed within the designated time, then the process is interrupted and not performed on until there is a next chance for it to be worked on. So naturally, the larger the quantum length is, the longer the response time is since the quantum time must be all used up, even if the process finishes before the allotted time is up.**

**Since response time goes up alongside quantum time, and the next process must wait for the previous process to burn through its runtime, then the worst case with quantum time being T is response time = T(N-1).**

7) "Preemptive schedulers are always less efficient than non-preemptive schedulers."

Explain how this statement can be true if you define efficiency one way and how it is

false if you define efficiency another way. (2 points)

**A scheduler might be efficient in regards to how much time is spent switching between processes. In this case, preemptive is much less efficient. Since processes can be interrupted and revisited later, even multiple times, there is a lot of time wasted on switching between processes. Non-preemptive is more ‘efficient’ in terms of time spent on switching, since there are no interruptions and the processes are switched to only once.**

**On the other hand, if we’re talking about response time, the preemptive schedulers are more efficient. The respond time can be much shorter since the scheduler can interrupt and switch to another process at any given time. In non-preemptive, the entire process must complete before the next can respond. This means longer processes cause the next process to take longer to respond and it doesn’t have a chance to be interrupted and switched off to decrease response time like in preemptive.**

8) What is the priority inversion problem? Does this problem occur if round-robin

scheduling is used instead of priority scheduling? Discuss. (2 points)

**Priority inversion arises when a lower priority process preempts a higher priority one.**

**This means even though a process has a lower priority, it’s running first. This can happen if the lower priority process has a resource that the higher priority process needs, meaning the higher process cannot run until that resource is freed. This defies the point of priorities, as the low priority process is running before the high priority one.**

**Round Robin’s set quantum time will ensure that every process will occur. Priority is irrelevant and processes being locked out from a resource is not possible, because when a process eats through its set quantum time, it’s forced to give up that resource and let the next process that needs it get a chance to use it.**

9) Does Peterson's solution to the mutual exclusion problem work when process scheduling

is preemptive? How about when it is non-preemptive? (2 points)

**Yes, Peterson’s solution works when the processing is preemptive. Peterson’s solution uses shared memory and flags to allow multiple processes to share a single resource without conflicting. One process is allowed access to the resource if its flag is requesting to use the data, and the flag for the next process is set to not needing that resource. It also uses a value to keep track of which process’s turn it is to enter the critical section of code.**

**However, this does NOT work for non-preemptive, because if a process starts out of order, then the requesting flag will never be able to change. The process that ran out of order would loop forever because it is not running during its proper turn, so the other processes can’t check for the correct flag from the process that was SUPPOSED to run first. This means the process that’s looping with be stuck with the resource and have no way of letting go.**

10)Consider the following set of processes, with the length of the CPU burst time given in

milliseconds:

Process Burst-Time Priority

P1 6 3

P2 1 1

P3 2 5

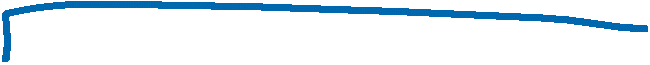
P4 3 4

P5 5 2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

(a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a

nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.



(b) What is the turnaround time of each process for each of the scheduling algorithms in part (a)?



(c) What is the waiting time of each process for each of the scheduling algorithm s in part (a)?



(d) Which of the schedules in part a results in the minimal average waiting time (over all

processes)? (5 points)

**The Shortest Job First Algorithm at 4.2 avg.**

11) The aging algorithm with a = 1/2 is being used to predict run times. The previous four

runs, from oldest to most recent, are 40, 20, 40, and 15 msec. What is the prediction of

the next time? (2 points)

**Runs 1 and 2**

**(40 + 20) / 2 = 30**

**Result 1 and Run 3**

**(30 + 40) / 2 = 35**

**Result 2 and Run 4**

**(35 + 15 ) / 2 = 25**

12) Explain what a multi-level feedback scheduler is and why it approximates SRTF.

**True or False** (also give an explanation for your choice): If a user knows the length of a CPU time-slice and can determine precisely how long his process has been running, then he can cheat a multi-level feedback scheduler. (2 points)

**A multi-level feedback scheduler (MLFS) is a scheduler in which there are queues that have priorities. A queue with high priority will have all of its processes execute first before any of the processes in the queues with lower priorities get a chance. If a process within a higher priority queue does not complete within an allotted time, it gets moved down to a lower priority queue, allowing the other higher priority processes to get a chance to run, but potentially delaying and starving lower priority processes. Processes that are more important, like INPUT/OUTPUT can be moved up into higher priority queues, thus interrupting lower priority processes. Less important processes like computational processes can be moved into lower priority queues.**

**This makes the statement true, because IF the user knows the length of a CPU time-slice, he can keep adding I/O requests to raise the priority level queue of a process. Anytime a process is about to get kicked down to a lower tier for using too much time, the downgrade can be avoided by simply forcing a more important process like I/O to maintain the process’s higher priority queue.**

13)Consider a system consisting of processes P1, P2, ..., Pn, each of which has a unique

priority number. Write the pseudo-code of a monitor that allocates three identical line

printers to these processes, using the priority numbers for deciding the order of

allocation. (5 points)

**Start with the following and populate the two functions request\_printer() and**

**release\_printer():**

monitor printers {

int num\_avail = 3;

int waiting\_processes[MAX PROCS];

int num\_waiting;

condition c;

void request\_printer(int proc\_number) {

**if (num\_avail > 0)**

**{**

**Decrement num\_avail;**

**}**

**Waiting\_processes[num\_waiting] = proc\_number;**

**Increment num\_waiting;**

**Sort the waiting processes;**

**While(num\_avail is 0 and waiting\_processes[0] does not equal proc number)**

**c.wait();**

**waiting\_processes[0] = waiting\_processes[num\_waiting-1];**

**decrement num\_waiting;**

**sort through the waiting processes again;**

**decrement the num\_avail;**

}

void release\_printer() {

**increment num\_avail;**

**c.broadcast(); to wake up all waiting processes**

}

}