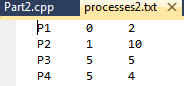
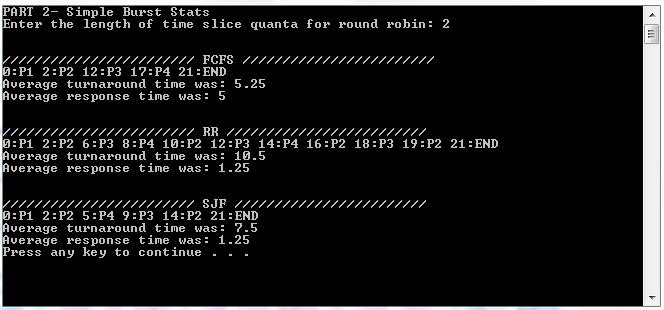
**Part 2**



Using this file containing process info:

I obtain these results.

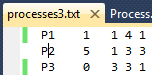


a) Looking at the image, FCFS turnaround time 5.25 is much better than RR 10.5. The conditions for FCFS to have better turnaround time are that the time slice for quantum is shorter than the run time of the processes. If processes run longer than the length of quanta, then they will start in RR and not be able to finish right away. FCFS processes always finish after they start though, so FCFS will have better turnaround when quanta lengths are shorter than burst lengths.

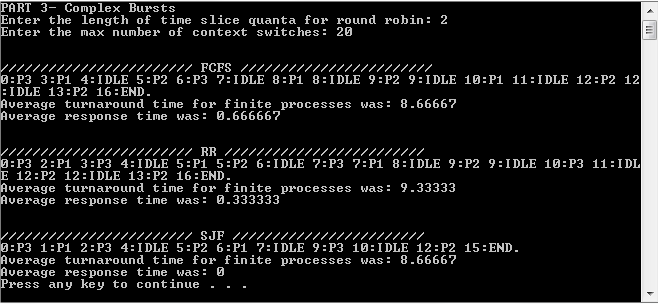
b) I don’t think this is possible. For the situation where only one CPU burst occurs at a time, FCFS will always have an advantage for turnaround time because a process will always start, run its burst duration, and finish in sequence after it started. For FCFS, processes will always get to run uninterrupted from start to finish, so their turnaround time will be at a minimum. For RR, best case scenario is that a process can start at the same time as a new quantum, giving turnaround time equal to FCFS. But outside of the best-case scenario, a process will start, run for a bit, pause while other processes run, and then finish, making its turnaround time worse. I.e. RR will either have a tied or worse turnaround time compared to FCFS.

c) Looking at the image again, SJF has a response time much equal to RR and better than FCFS because when a new, shorter process arrives, it starts immediately. Turnaround time for SJF vs RR was apparent, but SJF didn’t outperform FCFS in terms of turnaround. As discussed in part b, FCFS already has perfect turnaround time. Because the data set I used triggers SJF to pre-empt while running, SJF has a worse turnaround than FCFS. That being said, SJF could have tied the turnaround of FCFS had the data not included any processes arriving late and thereby triggering preemptive behavior.

**Part 3**



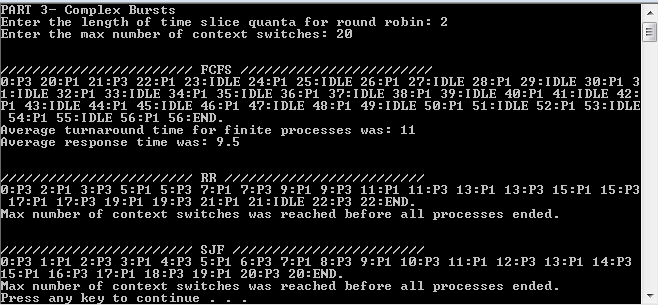
1. Using this file,

I obtained these results.

If you compare FCFS with SJF, you can see that at time 1 for SJF, there is an extra context switch not made for FCFS. This shows that SJF has been implemented to be pre-emptive; whenever a shorter CPU burst becomes available than the currently running one, the shorter burst will become active and will run. In my data set, P1 arrives at time 1, when the burst length of P3’s first burst has become 2. Since the newly arrived P1 has a shorted first burst than the currently running P3, P1 was preempted and began running.

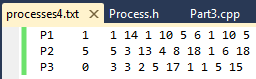


1. Using this input file,

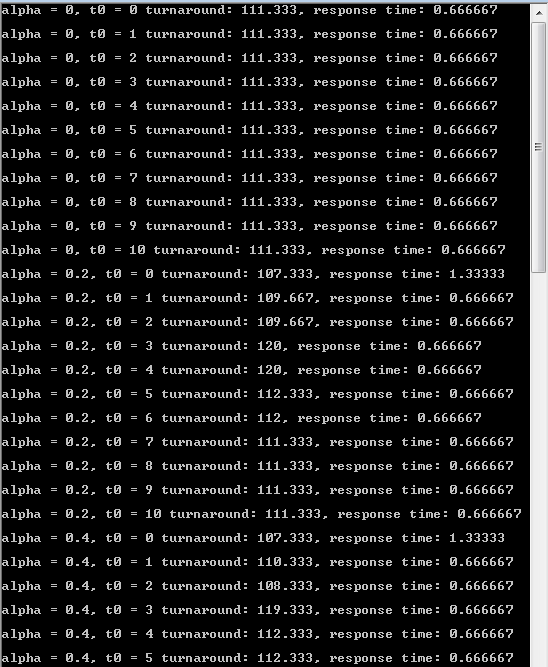
I obtained these results.

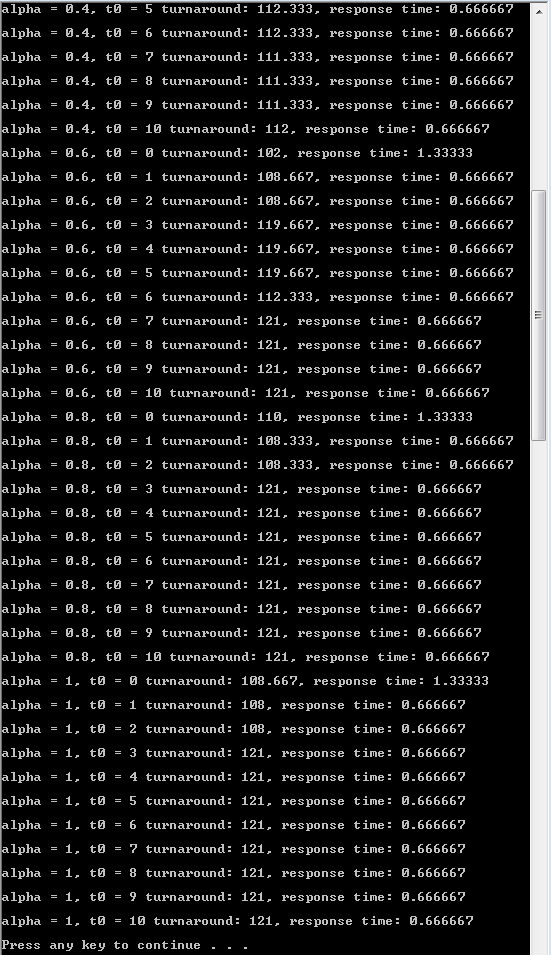
As you can see, The SJF scheduler said that “Max number of context switches was reached before all processes ended.” Because there are only two processes in this input, and one of the processes is infinite, that means that the other process (P3) was unable to finish according to the criteria for outputting turnaround and response time. P3 ended up starving because it had one main CPU burst that was long, and the competing P1 had very short CPU and IO bursts in infinite sequence. While P1 was busy running and restarting, P3 was only able to get through one second of its big burst at a time. And by setting the number of context switches to a low enough value, I was able to ensure that SJF would create starvation.

1. If you will look back to my images from part a, you will see that FCFS and RR were able to finish without starvation. FCFS is quite likely to be able to avoid starvation because of the way it lets processes run for as long as they need to at once. Only one maximum context switch happens for each burst in a process when running FCFS, so the number of context switches must be very low to cause any FCFS starvation. For RR, starvation is a little more likely. You can see this by comparing my images from part a (RR finishes without starvation) and part b (RR finished with starvation). For RR, avoiding starvation is less likely based on the value given not only for number of context switches but also for quanta. If the quantum time slice is really low with respect to the CPU bursts, then it is much more likely that starvation will occur in a dataset where a proportionally small number of context switches is allowed.

**Part 4**

1. Using this file for input,

I was able to get output like this

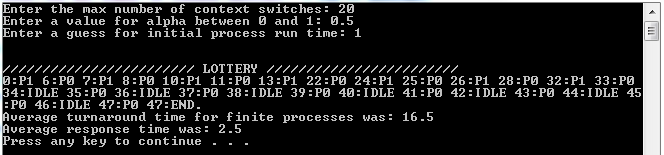


Some interesting things showed up! First, the very best turnaround time was found by using an initial t0 guess of 0 and an alpha value of 0.5 (102) and the best response time was a result of many of the runs (0.666667). Some trends I noticed with varying alpha and t0 include that for both the lower the t0 meant the better the turnaround and response time. On the contrary, having both high alpha and high t0 led to the highest turnaround in this survey (121). The general trend was that the best results were at the lowest alpha and t0 values. As t0 increased for a given alpha, the results tended to get worse. And as alpha increased, both alpha and t0 both always tended to get worse.

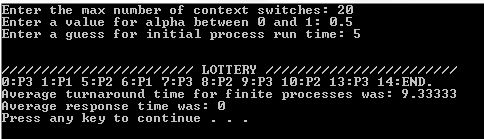
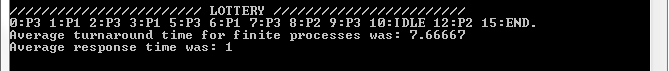
1. I think that my approximation is good. In my input data set, I never used values that seem totally unreasonable, so TAU(n+1) was never too much bigger than TAU(n). Because of this, the results were generally smooth. Granted, if you could know the actual run time of each burst, you could essentially run a perfect SJF. I went ahead and tried a SJF with this input file and found a turnaround time of 90 and response time of 0. This is noticeable already… If I calculate the average of my SRTF results turnaround of 115 and response of 0.75 approximately, (115/90) is a 27% variation for run time and obviously a response time of 0 for SJF is the absolute best, so SRTF can’t even compare in that department. With 0 response time, any other response time greater than 0 will be an infinite number of times larger. In terms of actual values, 0.75 is not a huge margin different than 0 in most respects, but it might be just enough to frustrate a user if their program is not snappy to start when it should. For turnaround, 27% difference in runtime is not insignificant… For the question of “How good is this approximation”, I’m initially inclined to say that it’s not that good at 27% variation. However, thinking about this in the real world, SJF is an idealized version that isn’t necessarily truly possible. For our crude implementation with rough calculation algorithms, I think that our approximation does a fair job at coming closer to the ideal and optimal situation of SJF.

\*Note: For the above runs, 400 context switches were allowed.

**Part 5**

1. Using the same input file for part 3 b, I am comparing this lottery result to the result of the SJF run that ended in starvation. For the lottery, I got this output

If you look at time 32, you will see the last instance of process P1 running. Using this lottery, P1 was actually able to finish and not starve! If you recall from part 3, the schedule ended after reaching the max number of context switches before P1 could finish. In this system, P1 must have gotten lucky enough to get chosen to run during the context switches enough times within the same number of context switches to complete. P1 was able to finish in the lottery because there was no guarantee that P0 would be able to jump back in just because it had a shorter CPU burst as was the case in part 3. This can easily be seen if you look at time 13 where P1 is able to run uninterrupted by context switches for 9 seconds before P0 jumps back in. I did help P1 out by setting t0 to 1 knowing that P0’s average would approach 1. (Me setting P1’s t0 to 1 helped P1 be able to compete at what ends up being a 50% chance of running initially.) You would not need to fix all situations to avoid starvation, but for the processes I did in part 3, this made it easy to show without running lots of times that starvation can be avoided.



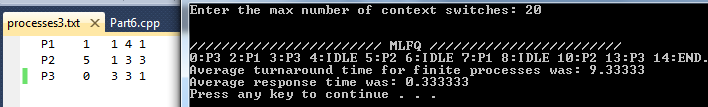
1. Recall from part 3 a that FCFS had turnaround time of 8.66667 and response time of 0.66667 and RR had turnaround time of 9.33333 and response time of 0.33333. Let’s compare that to some of the best results I found using this lottery after running it hundreds of times. The first image shows one of many runs that had a response time of 0. The turnaround time was nothing to write home about, but it outperformed both FCFS and RR in terms of response time. The second image shows the run that yielded the best turnaround time that I saw. 7.66667. That’s pretty impressive to me. Compared to 8.66667, that is a 13% better turnaround, and compared to 9.33333, that is a 22% better turnaround. I also think it is interesting that the schedule with a better turnaround time ended in real time 1 second before the schedule that had the improved turnaround time (14 vs 15).
2. The biggest drawback I could see to this lottery-style approach is that there is always going to be some chance that a process will run, regardless of its priority. If processes have even one ticket, they have some chance to be run at the next context switch. This is quite favorable to avoid starvation, but I could easily think of a scenario when that would not be ideal. For one example, if you think about trying to stream a video while leaving your Digits-Of-Pi-Counter program running, letting your side program always jump in would interfere with bandwidth and video quality. Instead, you would want your pi counter to “pseudo-starve” until you are not actively trying to do something else. Another example where you would not want programs unnecessarily taking CPU time would be if you are trying to do displaced real time communication with someone. If you are (for example) trying to synchronize your watch to the second with your friend in Australia that you are chatting with on Hangouts, having side programs jump in could easily slow down the process just enough to throw time-critical information out of sync.

Another problem I could foresee would be deriving an optimal algorithm for splitting up tickets at context switches. How do you know which processes to give how many tickets to? This could probably be settled in an arbitrary manner but could still be a problem.

This fair share lottery style seems like it would be good when you are trying to run multiple long processes that don’t have time critical information. For this situation, it would be good to give every process some breathing time so it doesn’t starve as long as nothing would be thrown off by the unimportant processes getting some run time.

**Part 6**

1. Using the same input file as part 3a,

I got this output.

From part 3a, we saw SJF get a response time of 0 and a turnaround of 8.66667. The results of this test seem to resemble the results of the lottery. An interesting aspect of the multi-level feedback queue is that it prioritizes IO-bound jobs. This is a very different approach than the other algorithms we implemented. Where SJF is designed to optimize response and turnaround times, the MLFQ algorithm is set up to optimize the amount of time a program is blocked while it is IO-bound. This means that even though MLFQ won’t get the perfect 0 response time that we saw in part 3a with SJF, a

user would have to wait (during the whole course of their program running) less time while the OS rewards his or her program for being IO intensive.

1. In my sample situation here, response time was certainly better for SJF. For my data set, SJF activated a process every time it arrived. This would not always be the case if processes that were actively running were shorter than the newly arriving processes. In SJF, the limiting factor for response time was the length of arriving bursts coming into play. For MLFQ, the limiting factor is the RR style scheduling within each queue. Instead of prioritizing response time and setting the incremental time to be 1 unit of time at a time, we let our program run RR which slightly slowed down my results. Also, like all of the other parts to this assignment, testing more with varying data sets would have certainly changed the results. Overall, the MLFQ seems like the most sensible algorithm to implement given appropriate interval times and RR quanta. In defense of this claim is that “The Mac OS X and Microsoft Windows schedulers can both be regarded as examples of the broader class of multilevel feedback queue schedulers.”[[1]](#footnote-1)

ASSUMPTIONS

Part 2)

1.The process has only one CPU intensive burst

Part 3)

1. Processes arrive at different times

2. Processes can have alternating CPU and IO bursts

3. There is a max number of context switches.

4. There will never be a CPU sequence of only -1.

5. Quanta do not pertain to FCFS

6. A CPU burst will come first and last. An IO burst will never start or finish the sequence

7. If a -1 is in the burst sequence, it will always follow a CPU burst

Part 4)

1. Processes can have alternating CPU and IO bursts

2. There is a max number of context switches.

3. A CPU burst will come first and last. An IO burst will never start or finish the sequence

4. There will never be a CPU sequence of only -1.

Part 5)

1. Processes can have alternating CPU and IO bursts

2. There is a max number of context switches.

3. A CPU burst will come first and last. An IO burst will never start or finish the sequence

4. There will never be a CPU sequence of only -1.

Part 6)

1. Processes arrive at different times

2. Processes can have alternating CPU and IO bursts

3. There is a max number of context switches.

4. There will never be a CPU sequence of only -1.

5. I use a fixed duration of 10 units of time before aging

6. Processes blocking at higher IO priorities will always block first. Not FCFS at different priorities

1. 1. Taken from Wikipedia. https://en.wikipedia.org/wiki/Multilevel\_feedback\_queue

   [↑](#footnote-ref-1)