Abraham Yepremian

CS 4310.01
Operating Systems
Assignment 1 Report

2/24/2019

Introduction

In this assignment our goal is to simulate an operating system scheduler with the First-Come-First-Serve (FCFS), Shortest-Job-First (SJF), Round-Robin (RR), and Lottery scheduling algorithms. The Round-Robin and lottery scheduling algorithms will take a parameter of time quantum. The time quantum is used for interactive schedulers where a process will only be run for a period of time. RR will be tested with time quantums of 20 and 40 while lottery will be tested with a time quantum of 40. The switch cost for the CPU is set as a constant to 3.

My program is written in Java. I have two files that comprise this project. The first is SchedulingAlgorithms.java. This file is the main file that runs the implementations of the various schedule algorithms and outputs the results of the algorithms into properly named csv files. The second file is Process.java. This file is a helper class that models the Process object. The program will read in a test data text file that reads in data for processes. It will read through and grab the pid, burst time and priority for each process. The process information is used to create a Process object. This object is stored in an ArrayList that is used throughout the main class implementation that helps to run the algorithms. For the FCFS algorithm, the processes are taken in the order in which they are received from the text file. The text file is in order of ascending pid. This is essentially how the FCFS algorithm works. The SJF algorithm first sorts the arraylist into the order of ascending burst time. Then takes in the processes in the order of shortest burst time first. These two were relatively simple to implement. More difficult to implement was the Round-Robin (RR) algorithm and then the lottery algorithm. The RR algorithm takes a time quantum and first come first serve order for the processes. The time quantum means that the process is only run for that period of time. This means I need to manage the burst times and

completion times carefully. I followed this logic in my code. The lottery scheduling algorithm is the one that I found most difficult to implement. This is because it wasn't discussed in depth in class and I could not find information online on how to implement it. I ended up creating my own implementation of the lottery system that I believe fits requirements. I will discuss it in the RR40 vs Lottery40 section of this paper. Let's jump right into data and analysis!

Data

Data is collected from a total of 4 different test files: testdata1.txt, testdata2.txt, testdata3.txt, and testdata4.txt. Both testdata1.txt and testdata2.txt have a total of 9 processes. Testdata3.txt has 19 processes and testdata4.txt has 27 processes.

Figure 1 - Average Turnaround Table

Algorithms	Average turnaround from testdata1.txt (9 processes)	Average turnaround from testdata2.txt (9 processes)	Average turnaround from testdata3.txt (19 processes)	Average turnaround from testdata4.txt (27 processes)
FCFS	1963.11	1913.89	7013.0	8770.52
SJF	1518.67	1370.44	4935.58	5685.41
RR20	2989.88	2704.33	10634.11	12381.26
RR40	2785.0	2532.11	9908.47	11583.37
Lottery40 (averages vary but trend is stable)	2476.0	2791.0	8423.63	9627.30

Figure 2 - Average Turnaround Time per Algorithm by Data Set

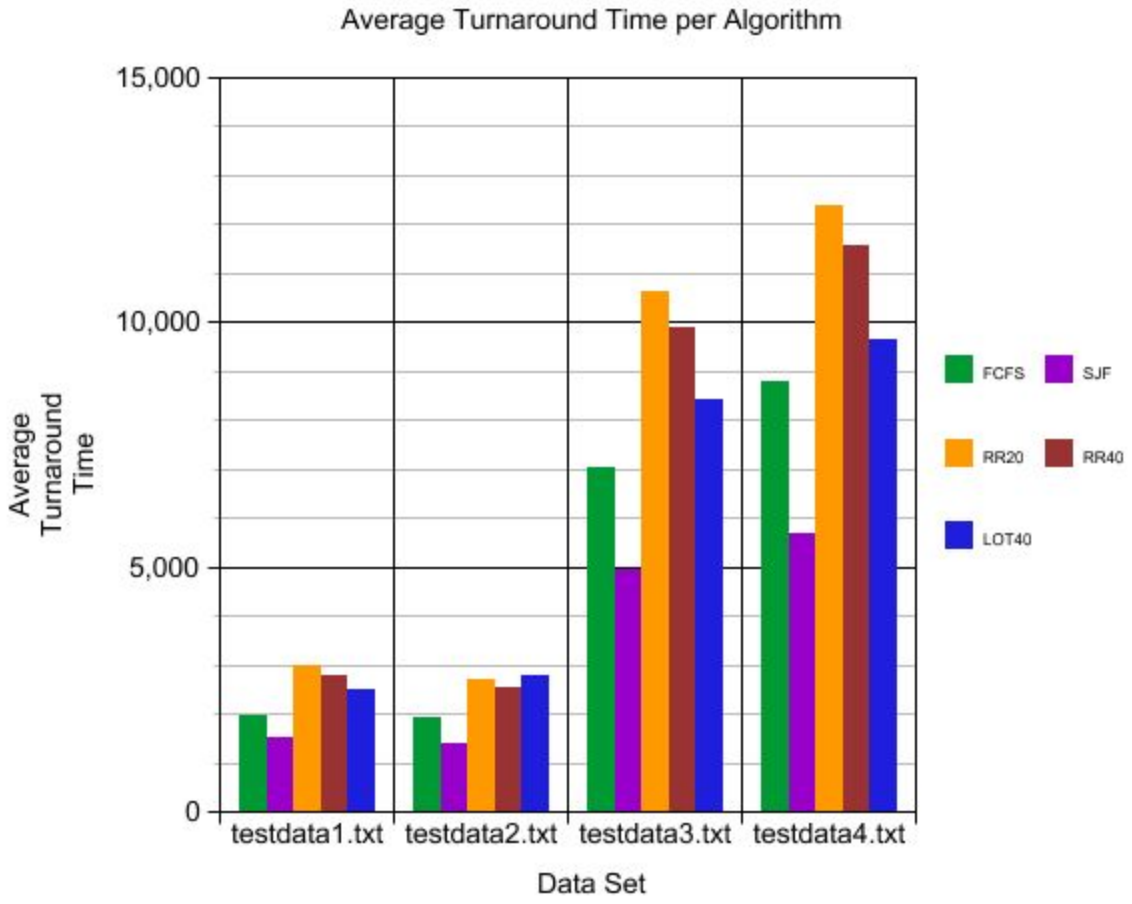


Figure 3 - FCFS vs. SJF

Figure 4 - RR20 vs. RR40

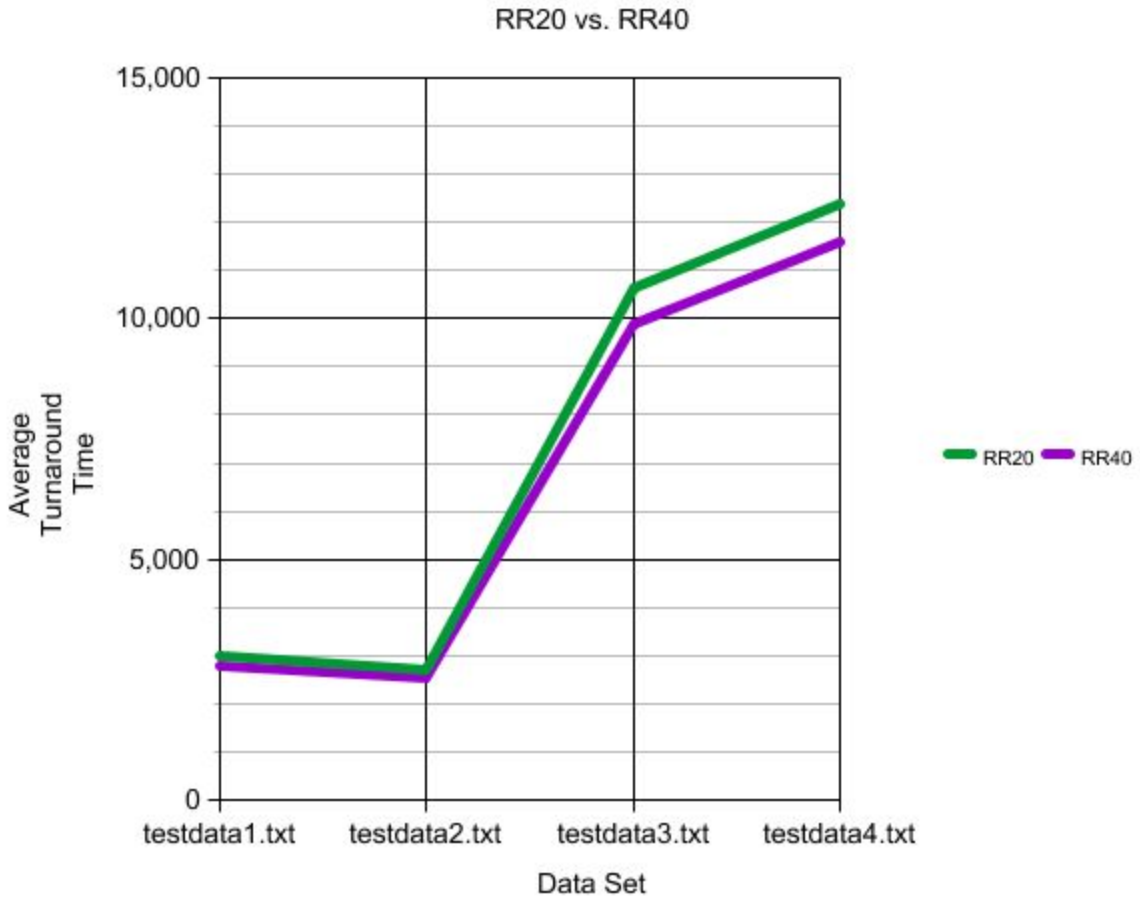
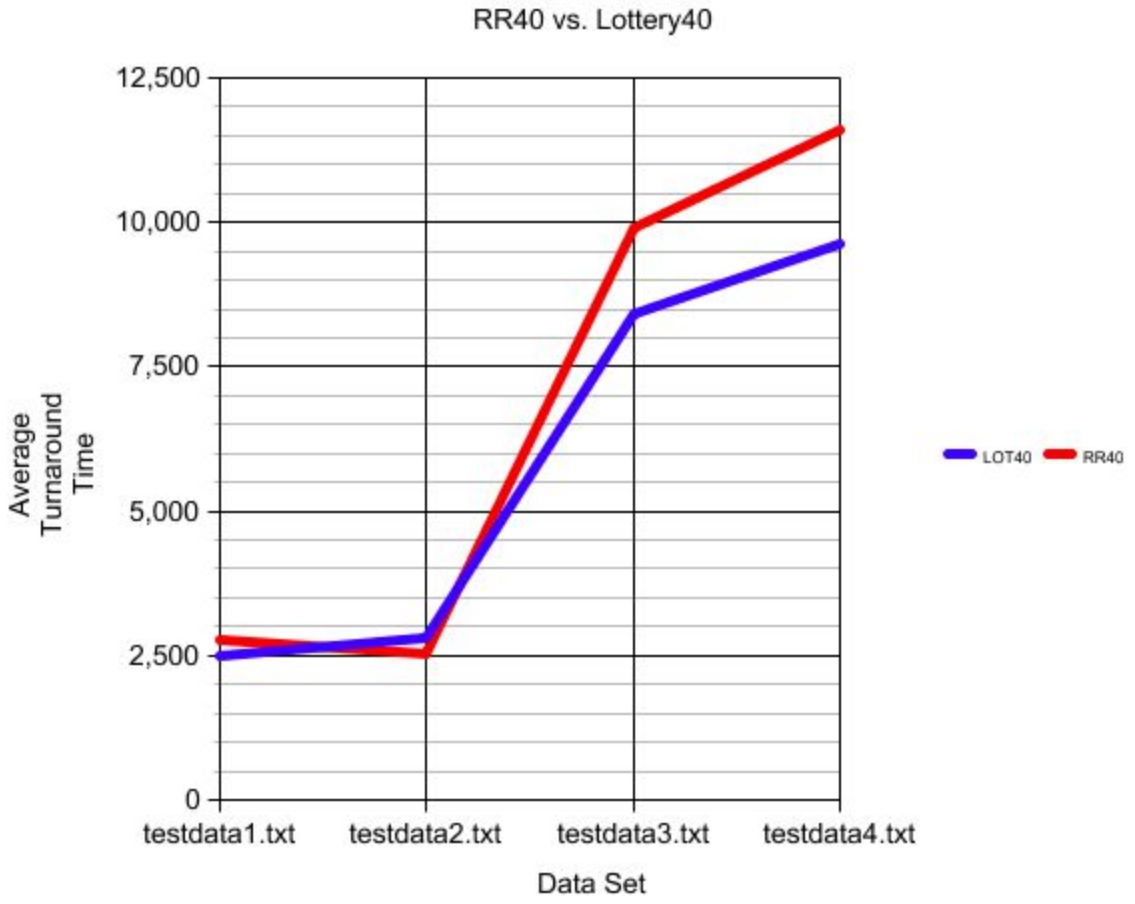


Figure 5 - RR40 vs. Lottery40



Analysis

We will compare FCFS vs. SJF, RR20 vs. RR40, and RR40 vs Lottery40. The data collected from a total of 4 different test data text files will be used as evidence for the comparisons.

FCFS vs. SJF

Looking at the data, we can clearly see that SJF performs better in terms of average turnaround time than FCFS. The average turnaround time is shorter and this performance makes sense because the average waiting time for each process is reduced. SJF will yield better performance than FCFS because it minimizes total wait time for the processes by completing the jobs with the smallest burst times first. FCFS and SJF are both non-preemptive algorithms which means that they will both wait until each process is finished before starting the next one. This is okay when the number of processes are not many, but it can lead to lag when a very large amount of processes need to be executed. Please refer to Figure 3 and you can clearly see that SJF always outperforms FCFS in terms of average turnaround time. Since testdata3.txt and testdata4.txt contain more processes than the first 2 data sets, they have a longer average turnaround time. I am impressed with the performance of SJF in all the test data because it has showed itself to be superior to all the other algorithms, according to Figure 2. However, there can be cases where SJF is not optimal, such as the case where all processes have equal burst time.

RR20 vs. RR40

RR20 showed to be less effective than RR40 in my experiment. RR20 took a time quantum of 20 while RR40 took a time quantum of 40. This difference in time quantum made RR40 the more effective algorithm through the testing on the 4 datasets, as evident by Figure 4. The Round-Robin algorithm is preemptive, meaning that it does not wait until a process is done before it moves onto the next process. It runs it for the duration of the time quantum before

moving on. Since it is preemptive, the cost of the context switch makes a much greater impact on the total bottomline. Since it is preemptive, there are more cases where the switch cost comes into play and drives the average turnaround time up. RR20 causes the costs of the context switches to add up to more than the context switches of RR40. The time quantum of 20 is too small for this algorithm to shine, therefore reducing performance. The time quantum of 40 is more suitable for this algorithm. If a time quantum that is too high is chosen, then the algorithm will become to represent the behavior of FCFS and behave like a non-preemptive algorithm. RR is like a FCFS with a time quantum, where it handles each process equally in the order that they come. RR40 performs better than RR20 for the reasons stated. It is possible to get RR to perform better if you choose a time quantum that allows most processes to be completed in their first run. With an optimal time quantum chosen for RR, you can balance the cpu efficiency and the average turnaround time. Overall, given this test data, RR seems to be a worse performer than the other algorithms. I see potential for this algorithm if we learn how to use it optimally.

RR40 vs. Lottery40

As evident by Figure 5, Lottery40 showed to be more efficient than RR40 in average turnaround time. This is especially interesting because I didn't expect that adding randomness would actually improve performance and decrease average turnaround time. Unbeknownst to me, the data clearly shows that as the number of processes increase (going from testdata1.txt to testdata4.txt is going from 9 processes to 27 processes), the effectiveness of Lottery40 vs RR40 is amplified. This is shown by Figure 2 and Figure 5.

I implemented the Lottery algorithm with my own method for randomness. First, I sort my ArrayList of Process object that were read in from the test data text file so that they are sorted by priority, highest first. Then I will generate a random lottery number that is in the sum of all the priorities added up from the ArrayList. For each process, I generate random numbers for each priority until a process matches my lottery number. Then the winner is chosen by which process matched my lottery number. I find the winning process and then run the winning process. The time quantum is used, similarly to RR. The processes with higher priority have a higher chance of matching my lottery number because they have more lottery tickets, so to speak.

Each process in the Lottery40 algorithm has a chance to be selected, but the chances of high priority processes to be run first are higher because they have more lottery tickets. To prevent against process starvation, the lottery will allow processes that have been waiting chances for execution while it is still prioritizing important processes. The priorities assigned to processes largely impacts the performance of the lottery algorithm. When priorities are assigned more heavily to long-waiting processes, starvation time can be reduced remarkably. Time quantum chosen may also affect the performance of the lottery algorithm, but it is not something we tested for in this experiment. The main focus from this section is that the data clearly shows that adding randomness via the lottery algorithm can provide significant improvements in efficiency over the RR40 algorithm, especially as the number of processes increase.

Conclusion

In this project, we explored the FCFS, SJF, RR20, RR40, and Lottery40 algorithms. We compared FCFS vs. SJF and found that SJF performed much better than FCFS in terms of average turnaround time, according to Figure 3. Its effect is extremely amplified as the number of processes increase. SJF seemed to perform the best given our test data; however, there is no guarantee of its performance as it depends on the burst times. Moreover, it does not solve the issue of process starvation as it is non-preemptive. We compared RR20 vs. RR40 and found that RR20 is more efficient than RR20 in terms of average turnaround time, according to Figure 4. This is because of the time quantums chosen. If the time quantum was too low and causes processes to have to wait longer for completion, then the cost of context switches adds up to make a big overall negative difference in performance. If the time quantum is too great, then the performance will begin to behave like non-preemptive algorithms, such as FCFS. The time quantum of 20 was too low and made for excess context switch costs. RR40 was more suitable than RR20, given our test data. Finally, we analyzed RR40 vs. Lottery40 and found that Lottery40 is more efficient than RR40 by adding randomness, according to Figure 5. Moreover, the effectiveness of Lottery40 is more evident as the number of processes increase. This assignment provided interesting insight into the FCFS, SJF, RR and Lottery scheduling algorithms. I honed my Java skills to develop these algorithms and test them against each other and output the results in csv files. Next, I analyzed the data from the experiment and learned of the trade offs between the algorithms. For future experimentation, we can try to use different time quantums and priorities. This may drastically affect how the Lottery algorithm works, and may be worth looking into. Overall, this was an interesting, thorough project that taught me all about the various scheduling algorithms.