**CS 4310 Operating Systems**

**Project #1 Simulating Job Scheduler and Performance Analysis**

**Due: 10/24**

(Total: 100 points)

**Student Name: \_\_\_**Branden Hitt**\_\_\_\_\_\_\_\_**

**Date: \_\_\_\_\_\_\_**10/24/19**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***Important:   
-*** *Please read this document completely before you start coding.   
- Also, please read the submission instructions (provided at the end of this document) carefully before submitting the project.*

***Project #1 Description:***

Simulating Job Scheduler of the Operating Systems by programming the following four scheduling algorithms that we covered in the class:

1. First-Come-First-Serve (FCFS)
2. Shortest-Job-First (SJF)
3. Round-Robin with Time Slice = 2 (RR-2)
4. Round-Robin with Time Slice = 5 (PR-5)

You can use either Java or your choice of programming language for the implementation. The objective of this project is to help student understand how above four job scheduling algorithms operates by implementing the algorithms, and conducting a performance analysis of them based on the performance measure of their average turnaround times (of all jobs) for each scheduling algorithm using multiple inputs. Output the details of each algorithm’s execution. You need to show which jobs are selected at what times as well as their starting and stopping burst values. You can choose your display format, for examples, you can display the results of each in *Schedule Table* or *Gantt Chart* format (as shown in the class notes). The project will be divided into three parts (phases) to help you to accomplish above tasks in in a systematic and scientific fashion: Design and Testing, Implementation, and Performance Analysis.

The program will read process burst times from a file (job.txt) – this file will be generated by you. Note that you need to generate multiple testing cases (with inputs of 5 jobs, 10 jobs, 15 jobs, 20 jobs, 25 jobs and 30 jobs). A sample input file of five jobs is given as follows (burst time in ms):

[Begin of job.txt]

Job1

7

Job2

18

Job3

10

Job4

4

Job5

12

[End of job.txt]

Note: you can assume that

1. There are no more than 30 jobs in the input file (job.txt).
2. Processes arrive in the order they are read from the file for FCFS, RR-2 and RR-5.
3. All jobs arrive at time 0 for SJF.
4. FCFS use the order of the jobs, Job1, Job2, Job3, …

You can implement the algorithms in your choice of data structures based on the program language of your choice. Note that you always try your best to give the most efficient program for each problem. The size of the input will be limited to be within 30 jobs.

***Submission Instructions:***

* ***turn in the following @blackboard.cpp.edu after the completion of all three parts, part 1, part 2 and part 3***
  + - * 1. ***(1) four program (your choice of programming language with proper documentation) files***

***(2) this document (complete all the answers)***

**Part1**

**Design & Testing (30 points)**

* 1. Design the program by providing pseudocode or flowchart for each sorting algorithm.

<insert answers here>

First Come First Serve

Create a Queue

Push the jobs into the queue as they are read from the file

While there are still jobs to process

Record the start time of the job

Process the job (pop)

Record the end time of the job

Add the end time to the ‘avgTurnAround’ time

Divide the ‘avgTurnAround’ time by the job count

Return the ‘avgTurnAround’ time

Shortest Job First

Create a priority queue

Insert the jobs into the priority queue with the shorter burst time being a higher priority

While there are still jobs to process

Record the start time of the job

Process the job (pop)

Record the end time of the job

Add the end time to the ‘avgTurnAround’ time

Divide the ‘avgTurnAround’ time by the job count

Return the ‘avgTurnAround’ time

Round Robin (time slice = 2 or 5)

Create a queue

Push the jobs into the queue as they are read from the file

While there are still jobs to process

Give a time slice to the front job

Pop the job off the queue

If the job still has burst time remaining

Push the job to the end of the queue

Otherwise

Record the end time of the job

Add the end time to the ‘avgTurnAround’ time

Divide the ‘avgTurnAround’ time by the job count

Return the ‘avgTurnAround’ time

* 1. Design the program correctness testing cases. Give at least 3 testing cases to test your program, and give the expected correct output of the program for each case in order to test the correctness of each algorithm.

<complete the following table>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Testing case # | Input  (table of jobs with its job# and length | Expected output for FCFS  (√ if Correct after testing in Part ) | Expected output for  SJF (√ if Correct after testing in Part 3) | Expected output for  RR-2 (√ if Correct after testing in Part 3) | Expected output for  RR-5 (√ if Correct after testing in Part 3) |
| 1  (5 jobs) | Job 1  9  Job 2  12  Job 3  3  Job 4  6  Job 5  5 | 9+21+24+30+35  =119  119/5 = 23.8  ✓ | 3+8+14+23+35  =83  83/5= 16.6  ✓ | 15+25+26+31+35  =132  132/5= 26.4  ✓ | 13+23+27+33+35  =131  131/5=26.2  ✓ |
| 2 (10 jobs) | Job 1  9  Job 2  12  Job 3  3  Job 4  6  Job 5  5  Job 6  8  Job 7  13  Job 8  14  Job 9  4  Job 10  3 | 9+21+24+30+35+43  +56+70+74+77  =439  539/10= 43.9  ✓ | 3+6+10+15+21+29  +38+50+63+77  =312  312/10=31.2  ✓ | 25+37+38+44+45+  57+62+70+75+77  =530  530/10=53.0  ✓ | 13+23+42+45+49+ 55+58+70+73+77  =505  505/10=50.5  ✓ |
| 3  (15 jobs) | Job 1  9  Job 2  12  Job 3  3  Job 4  6  Job 5  5  Job 6  8  Job 7  13  Job 8  14  Job 9  4  Job 10  3  Job 11  13  Job 12  7  Job 13  15  Job 14  2  Job 15  15 | 9+21+24+30+35+  43+56+70+74+  77+90+97+112  +114+129  =981  981/15=65.4  ✓ | 2+5+8+12+17+23  +30+38+47+59  +72+85+99  +114+129  =740  740/15=49.33  ✓ | 28+35+47+48+  62+63+83+90+95  +109+120+122  +123+128+129  =1282  1282/15=85.47  ✓ | 13+23+42+45+  62+71+77+80+97  +109+112+116  +119+124+129  =1219  1219/15=81.27  ✓ |

* 1. Design testing strategy for the programs. Discuss about how to generate and structure the randomly generated inputs for experimental study later in Part 3.

*Hint 1: To study the performance evaluation of the four job scheduling algorithms, this project will use three different input sizes, 5 jobs, 10 jobs and 15 jobs. It is the easiest to use a random number generator for generating the inputs. Note that you need to decide the maximum value of job length (use at least 20). However, student should store each data set in various sizes and use the same data set for each job scheduling algorithm.*

*Hint 2: The average time of each input data size can be calculated after an experiment is conducted in m (use at least 10) trails. We can denote the run time results (performance of the average turnaround time) as the set X which contains the m number of trails, where X = {x1, x2, x3 … xm} and each xi is one trial of the documented run time.  
  
The Average time =*

*The student should think about and decide how many trials (the value of m) are needed for this experiment, and the maximum of the job length values.*

My testing strategy was very straightforward. I first created 10 separate files that each included 30 jobs. The jobs themselves were named Job1 to Job30 and they included burst times that were randomized from 1 to 20.

[beginning of Job1.txt]

Job 1

9

Job 2

12

Job 3

3

…

Job30

17

[end of Job1.txt]

By creating these files once, I could use them for multiple trials while also keeping the same data sets to compare between the algorithms. I could also test at different job sizes by only reading the amount of jobs needed from a specific file. For instance, using a job size of 5, I would read the first file Job1.txt and take the first 5 jobs and input them into each of the four algorithms. Since every algorithm had the same data set, I could grab the 4 average turnaround times from the result and then move on to the next job file, Job2.txt. I would repeat the process until each algorithm contained 10 average turnaround times for the job size of 5. These 10 were then averaged to give 4 final average times (one for each algorithm). By doing this process, I found a more accurate depiction of which algorithm was better than the others. Still, this was not enough since I had a small job size of 5.

I then repeated this process of 10 trials but at different job sizes. I found the average times at job size 5, 10, 15, 20, 25, and 30 for each of the four algorithms. Following that, I plotted these averages on a graph vs the job size. This gave me a better idea of the growth of time as the job size increased. These four graphs were then combined into one graph to give me a visual comparison of the performance between the four algorithms.

**Part 2**

**Implementation (30 points)**

1. Code each program based on the design (pseudocode or flow chart) in Part 1(a).

<generate four programs and stored them in four files, needed to be submitted>

1. Document the program appropriately.

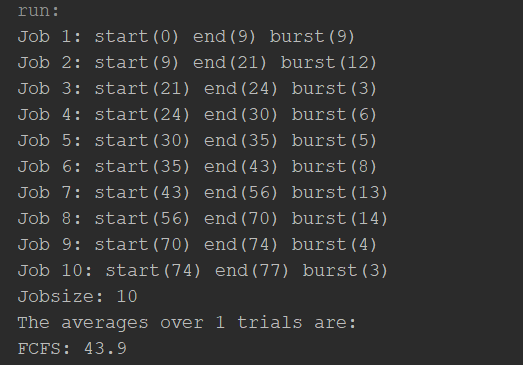
<generate documentation inside the four program files>

1. Test you program using the designed testing input data given in the table in Part 1(b), Make sure each program generates the correct answer by marking a “√” if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

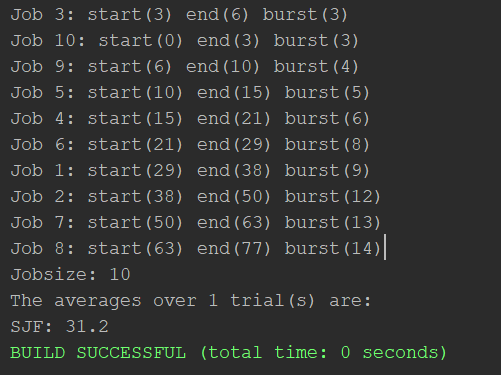
<Complete the four columns of the four algorithms in the table @Part 1(b)>

1. For each program, capture a screen shot of the execution (Compile&Run) using one testing case to show how this program works properly

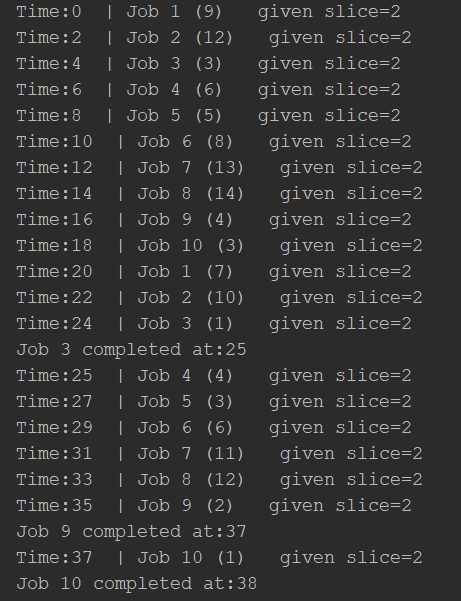
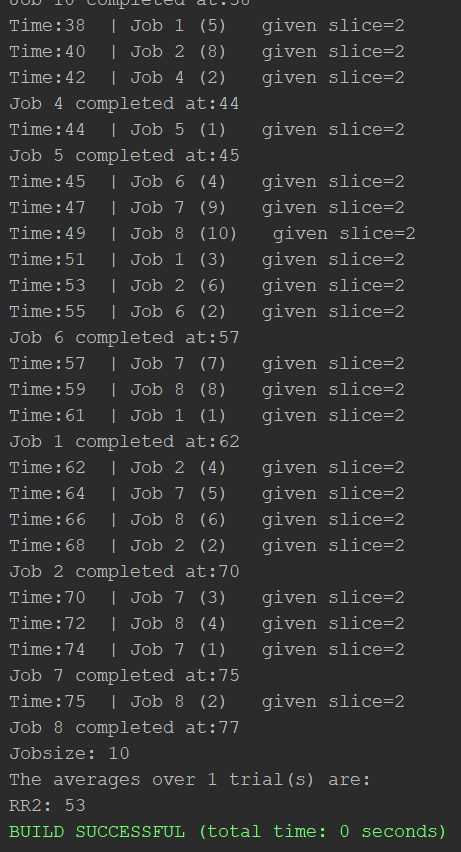
First Come First Serve



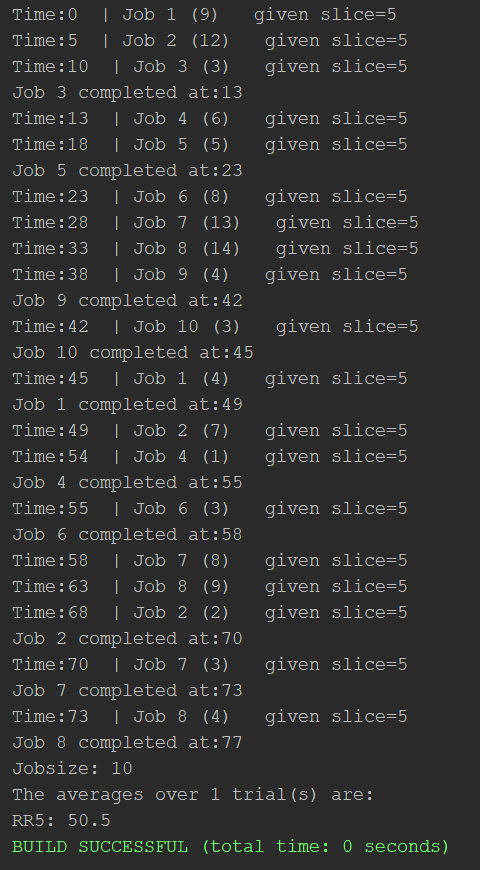
Shortest Job First



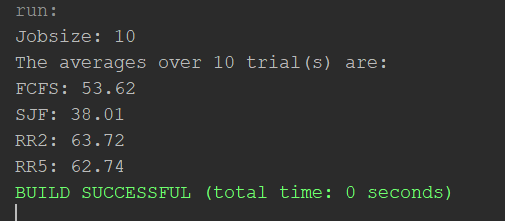
Round Robin (time slice 2)



Round Robin (time slice 5)



All four scheduling algorithms over 10 trials



By now, four working programs are created and ready for experimental study in the next part, Part 3.

**Part 3   
Performance Analysis (100 points)**

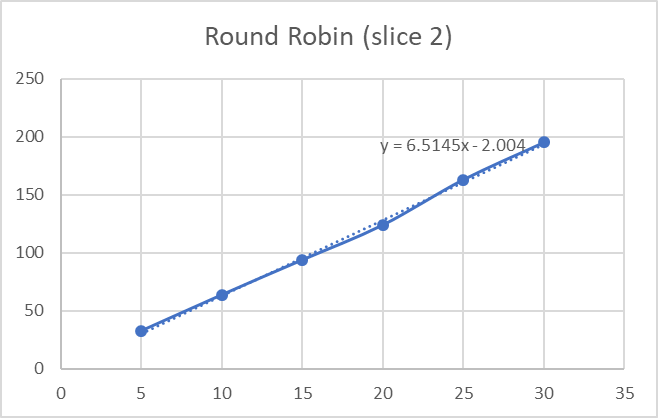
1. Run each program with the designed randomly generated input data given in Part 1(c). Generate a table for all the experimental results for performance analysis as follows.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size  n jobs | Average of average turnaround times  (FCFS Program) | Average of average turnaround times  (SFJ Program) | Average of average turnaround times  (RR-2) | Average of average turnaround times  (RR-5) |
| 5 jobs | *Trials(M)=10*  *AVGT1 = 23.8*  *AVGT2 = 44.8*  *AVGT3 = 20.6*  *AVGT4 = 36.0*  *AVGT5 = 29.6*  *AVGT6 = 37.4*  *AVGT7 = 37.0*  *AVGT8 = 27.4*  *AVGT9 = 17.2*  *AVGT10 = 24.0*  *=AVG*  *AVG=29.78* | *M=10*  *AVGT1 = 16.6*  *AVGT2 = 37.2*  *AVGT3 = 14.6*  *AVGT4 = 28.0*  *AVGT5 = 19.4*  *AVGT6 = 32.2*  *AVGT7 = 22.0*  *AVGT8 = 10.6*  *AVGT9 = 14.0*  *AVGT10 = 20.6*  *AVG=21.52* | *M=10*  *AVGT1 = 26.4*  *AVGT2 = 59.2*  *AVGT3 = 20.4*  *AVGT4 = 42.0*  *AVGT5 = 28.2*  *AVGT6 = 52.4*  *AVGT7 = 33.8*  *AVGT8 = 14.6*  *AVGT9 = 18.8*  *AVGT10 = 30.2*  *AVG=32.6* | *M=10*  *AVGT1 = 26.2*  *AVGT2 = 59.2*  *AVGT3 = 19.8*  *AVGT4 = 43.4*  *AVGT5 = 31.0*  *AVGT6 = 50.8*  *AVGT7 = 34.0*  *AVGT8 = 18.6*  *AVGT9 = 17.2*  *AVGT10 = 30.0*  *AVG=33.02* |
| 10 jobs | *M=10*  *AVGT1 = 43.9*  *AVGT2 = 72.6*  *AVGT3 = 39.7*  *AVGT4 = 60.1*  *AVGT5 = 54.4*  *AVGT6 = 64.7*  *AVGT7 = 62.6*  *AVGT8 = 36.3*  *AVGT9 = 51.4*  *AVGT10 = 50.5*  *AVG=53.62* | *M=10*  *AVGT1 = 31.2*  *AVGT2 = 51.4*  *AVGT3 = 27.7*  *AVGT4 = 40.6*  *AVGT5 = 41.7*  *AVGT6 = 44.4*  *AVGT7 = 44.0*  *AVGT8 = 19.0*  *AVGT9 = 42.4*  *AVGT10 = 37.7*    *AVG=38.01* | *M=10*  *AVGT1 = 53.0*  *AVGT2 = 90.2*  *AVGT3 = 44.4*  *AVGT4 = 68.2*  *AVGT5 = 67.6*  *AVGT6 = 79.0*  *AVGT7 = 75.0*  *AVGT8 = 28.6*  *AVGT9 = 68.9*  *AVGT10 = 62.3*  *AVG=63.72* | *M=10*  *AVGT1 = 26.4*  *AVGT2 = 59.2*  *AVGT3 = 20.4*  *AVGT4 = 42.0*  *AVGT5 = 28.2*  *AVGT6 = 52.4*  *AVGT7 = 33.8*  *AVGT8 = 14.6*  *AVGT9 = 18.8*  *AVGT10 = 30.2*  *AVG=62.74* |
| 15 jobs | *M=10*  *AVGT1 = 65.4*  *AVGT2 = 91.3*  *AVGT3 = 61.9*  *AVGT4 = 80.4*  *AVGT5 = 79.5*  *AVGT6 = 90.7*  *AVGT7 = 87.7*  *AVGT8 = 55.5*  *AVGT9 = 85.9*  *AVGT10 = 74.9*  *AVG=77.31* | *M=10*  *AVGT1 = 49.3*  *AVGT2 = 46.3*  *AVGT3 = 45.6*  *AVGT4 = 45.1*  *AVGT5 = 48.6*  *AVGT6 = 71.5*  *AVGT7 = 64.7*  *AVGT8 = 38.3*  *AVGT9 = 67.1*  *AVGT10 = 58.7*  *AVG=53.52* | *M=10*  *AVGT1 = 85.5*  *AVGT2 = 84.0*  *AVGT3 = 77.3*  *AVGT4 = 78.5*  *AVGT5 = 84.5*  *AVGT6 = 128.7*  *AVGT7 = 114.5*  *AVGT8 = 61.9*  *AVGT9 = 116.5*  *AVGT10 = 101.6*  *AVG=93.8* | *M=10*  *AVGT1 = 81.3*  *AVGT2 = 85.4*  *AVGT3 = 72.3*  *AVGT4 = 80.5*  *AVGT5 = 89.9*  *AVGT6 = 123.9*  *AVGT7 = 112.9*  *AVGT8 = 60.3*  *AVGT9 = 112.3*  *AVGT10 = 99.7*    *AVG=91.85* |

1. **Plot a graph of each algorithm, average turnaround time vs input size (# of jobs), and summarize the performance of each algorithm based on its own graph.**

The first come first serve scheduling algorithm performed fairly consistent. The more jobs that were added meant that it would have a longer average turnaround time but the added jobs did not drastically change the trend of the graph. At 30 jobs, the average time was 153ms. This algorithm was only beat out by the shortest job first algorithm.

When compared to the other algorithms, the shortest job first algorithm was always the fastest. By prioritizing the shortest job, this method was able to cut down on unnecessary turnaround time between processes. At 30 jobs, this algorithm only took 106ms.



Both round-robin algorithms did not fair so well when compared to the first two algorithms. Although the algorithm’s approach was to spread out processing time, the average turnaround time was greatly hindered. At 30 jobs, this algorithm’s average was 195ms. This was almost double the shortest job first’s result.

The second round-robin algorithm was very similar to the first round-robin algorithm. When ignoring process switching overhead, the similar approach of sharing time between processes led to almost the same exact outcome regardless of the time slice value. I’m sure that if we were to account for process switching overhead, there would be a more substantial difference between the two. At 30 jobs, the second round-robin algorithm average was 191ms.

**Plot all four graphs on the same graph and compare the performance of all four algorithms. Rank four scheduling algorithms. Try giving the reasons for the findings.**    

As you can see from the graph above, the algorithms all began at similar results but then diverged as the job lengths increased. Consistently, the shortest job first algorithm performed the best while both round robin algorithms performed the worst. This makes sense as we did not need to worry about the insertion of the jobs into the priority queue for the shortest job first algorithm. That step alone would add overhead much like process switching could add time to the round robin algorithm.

1. **Conclude your report with the strength and constraints of your work. At least 100 words.**

**(Note: It is reflection of this project. If you have a chance to re-do this project again, what would you like to keep and what would you like to do differently in order get a better quality of results.)**

Overall, this project was very enlightening. It was interesting to see how each algorithm would play out and then try to guess what was influencing the results. The biggest strength of my implementation was that I used queues for the algorithms. Queues made it quick work to add the jobs into a line that could process them quickly and efficiently. In the case of the round robin, queues came in handy quite a bit since I could just pop a job off the front and send it to the back if it still needed additional processing after feeding it a time slice. The priority queue I used with the shortest job first algorithm was also helpful as the initial insertion into the queue allowed me to avoid constant searching for the shortest job.

I think going forward, my implementation could be useful to extend the variables considered in the scheduling program. For instance, adding a process switching time to the job class would not affect the structure of my code (when considering my generic queues) but could allow me to better analyze the difference between time slices.

The one regret I have of my implementation was reading the jobs from a file. It may have been more beneficial to have separate trials of data created on execution and stored in a 2d array rather than read from files every time. This would allow me to analyze the algorithms at higher trial numbers and different maximum sizes of jobs.

The other thing to consider is the language chosen to implement these simulations. I programmed this in Java which proved to be easy to write and read, however there may be more efficient languages. The creation and processing of queues would probably be faster and less memory intensive in C++.