Anthony Borza

Professor Tarquinio, Michael

Final Project: Scheduling Algorithms

Due Date: 10/12/17

Introduction:

As a student seeking a degree in computer science, it is crucial that i understand how an operating scheduler handles multiple processes or threads. This project was designed to simulate an operating system scheduler in java, using many of the scheduling algorithms that were addressed throughout this class. The paper will outline the required steps taken to implement a successful operating system scheduler. The first section of the paper will provide the projects objectives. This will address the overall goal of the project, and what is to be expected when running the program. The second section will provide multithreading's involvement when developing an operating system, and why it is important. The next couple sections will cover the process for implementing the First-Come-First Serve algorithm, the Shortest Job First, and the Round Robin scheduling algorithms. Each scheduling algorithm will display tables, figures, and screenshots that are used to measure the algorithms performance metrics. Lastly, there will be a lesson learned, and conclusion section that will cover an analysis of the project. These sections will include: the positives and negatives that occurred throughout this project, as well as, the differences and similarities with each scheduling algorithm.

Project Objective:

The purpose of this project was to develop a simulation of an operating system scheduler that can handle multiple threads by implementing the First-Come-First Serve (FCFS), Shortest Job First (SJF), and Round Robin (RR) scheduling algorithm in java. The FCFS algorithm that was implemented was based on the order in which the threads were either stored in an array. The SJF algorithm involved selecting the threads (worker threads), and ordering them in ascending order (least to greatest). The RR algorithm involved scheduling each thread (worker threads) based on a specified time quantum. This algorithm also implemented the suspend, and resume

methods offered by the synchronization library in java. You will find in the results that each algorithm records, the total time to complete each thread, the average wait time for all threads, and the overall runtime for each thread.

Multithreading and Operating System Scheduler:

Multithreading plays a huge role in the development of an operating system scheduler. The process of multithreading is where the same program contains multiple concurrent threads of execution. With multithreading a program, or application will first begin with one thread, and can then later create other threads to perform tasks. All operating systems that support multithreading have a scheduler that is responsible for preempting, and scheduling all threads of all processes (Krzyzanoeski, 2014).

First-Come-First Serve (FCFS) Implementation:

The FCFS scheduling algorithm processes the threads in the order in which they are queued in. This is a non-preemptive scheduling algorithm, where a running task is executed until it is finished, and; therefore, cannot be interrupted by another thread. This algorithm was implemented using Java, and assigned its own class called: FirstComeFirstServe.java. Within this class holds a method that is used to calculate a variety of performance metrics for each of the IOIntensive, and Computationally threads. The performance metrics that were calculated include: the wait time for each thread, average wait time for all threads, turnaround time for each thread, and average turnaround time for all threads. For calculating these times, I followed the Grants Chart, which is a bar chart that is used to illustrate how the scheduling algorithm works. I will provide an example of how it works. See figure 1 below:

See Figure 1:

Figure 1 Retrieved from: (Silberschatz et al. 2013).

For calculating the wait time for the first process p_1, the wait time is 0 milliseconds, for process p_2, the wait time is 24 milliseconds, and for process p_3, the wait time ins 27 milliseconds. To calculate the average wait time, you would add up all the processes times, and divide by the total processes. For example, $(0 + 24 + 27) / 3 = 17$ milliseconds (Silberschatz et al. 2013). The order the processes arrive does not matter, and the average wait time would not change.

First-Come-First Serve Results:

IOIntensive Before Computationally:

The following table, and screenshot below shows the results after running the IOIntensive threads before the Computationally threads. For the IOIntensive threads to show before the Computationally threads, I had to modify the code in the main method of the Controller class. There is a for loop within the main method that Is designated to creating the threads. Within that for loop is and if else statement. To show the IOIntensive threads first, I had to assign the variable 'i' so it was less than 5 ex: (if($i < 5$)). By doing this, I could display the first five threads as IOIntensive, and the next five as Computationally. The table below shows the total time and, total wait time for each thread, as well as, the average wait time, and overall runtime for all threads.

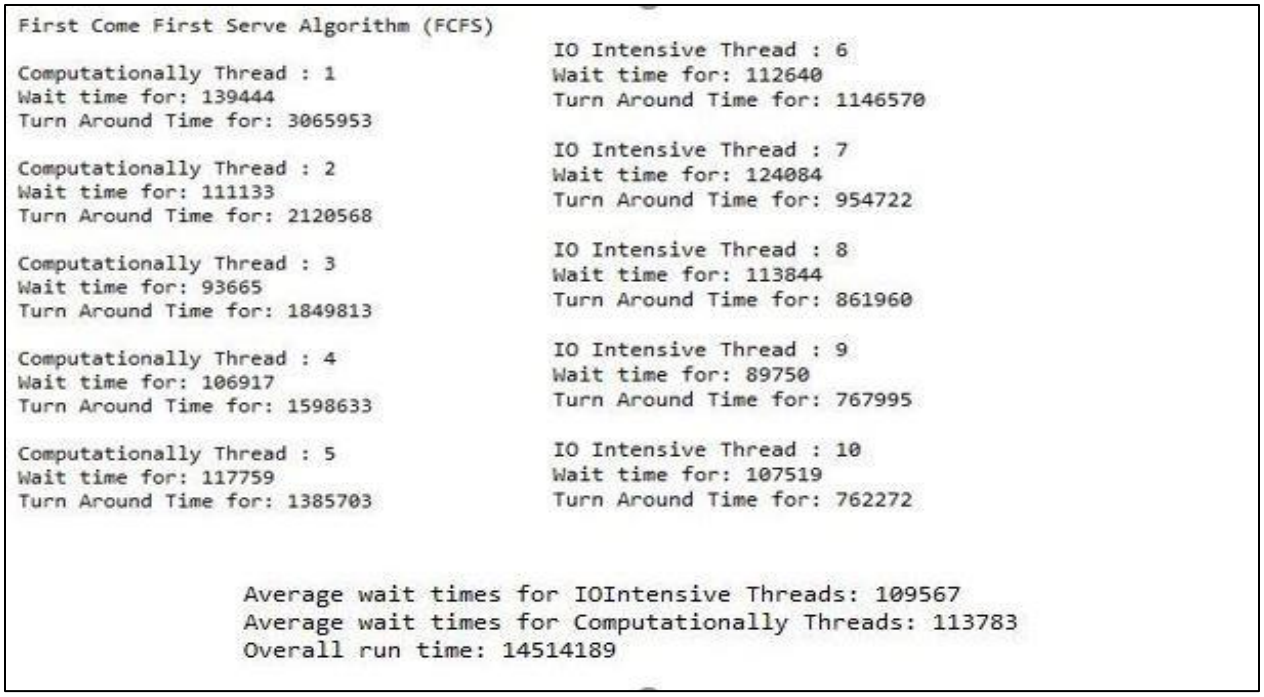
IOIntensive Before Computationally Threads					
IOIntensive Threads	**Wait Time**	**Turn Around Time**	**Computationally Threads**	**Wait Time**	**Turn Around Time**
Thread 1:	143058(Nano)	2959338(Nano)	**Thread 6:**	102700(Nano)	1554963(Nano)
Thread 2:	100592(Nano)	2282599(Nano)	**Thread 7:**	112639(Nano)	1476055(Nano)
Thread 3:	139143(Nano)	2654248(Nano)	**Thread 8:**	115349(Nano)	901716(Nano)
Thread 4:	106315(Nano)	1693804(Nano)	**Thread 9:**	99388(Nano)	1308602(Nano)
Thread 5:	116554(Nano)	1478766(Nano)	**Thread 10:**	114146(Nano)	1025799(Nano)
Average Wait Time for IOIntensive Threads: 121132(Nano)			**Average Wait Time for Computationally Threads:** 108844(Nano)		
Overall Run Time for both IOIntensive and Computationally Threads: 17335890(Nano)					

```
First Come First Serve Algorithm (FCFS)

IO Intensive Thread : 1                    Computationally Thread : 6
Wait time for: 143058                      Wait time for: 102700
Turn Around Time for: 2959338              Turn Around Time for: 1554963

IO Intensive Thread : 2                    Computationally Thread : 7
Wait time for: 100592                      Wait time for: 112639
Turn Around Time for: 2282599              Turn Around Time for: 1476055

IO Intensive Thread : 3                    Computationally Thread : 8
Wait time for: 139143                      Wait time for: 115349
Turn Around Time for: 2654248              Turn Around Time for: 901716

IO Intensive Thread : 4                    Computationally Thread : 9
Wait time for: 106315                      Wait time for: 99388
Turn Around Time for: 1693804              Turn Around Time for: 1308602

IO Intensive Thread : 5                    Computationally Thread : 10
Wait time for: 116554                      Wait time for: 114146
Turn Around Time for: 1478766              Turn Around Time for: 1025799


        Average wait times for IOIntensive Threads: 121132
        Average wait times for Computationally Threads: 108844
        Overall run time: 17335890
```

Computationally Before IOIntensive:

The following table, and screenshot below shows the results after running the Computationally threads before the IOIntensive threads. For the Computationally threads to show before the IOIntensive threads, I had to modify the code in the main method of the Controller class. There is a for loop within the main method that Is designated to creating the threads. Within that for loop is and if else statement. To show the Computationally threads first, I had to assign the variable 'i' so it was greater than or equal to 5 ex: (if(i >= 5)) By doing this, I could display the first five threads as Computationally, and the next five as IOIntensive. The table below shows the total time and, total wait time for each thread, as well as, the average wait time, and overall runtime for all threads.

Computationally Before IOIntensive Threads					
Computationally Threads	**Wait Time**	**Turn Around Time**	**IOIntensive Threads**	**Wait Time**	**Turn Around Time**
Thread 1:	139444(Nano)	3065953(Nano)	**Thread 6:**	112640(Nano)	1146570(Nano)
Thread 2:	111133(Nano)	2120568(Nano)	**Thread 7:**	124084(Nano)	954722(Nano)
Thread 3:	93665(Nano)	1849813(Nano)	**Thread 8:**	113844(Nano)	861960(Nano)
Thread 4:	106917(Nano)	1598633(Nano)	**Thread 9:**	89750(Nano)	767995(Nano)
Thread 5:	117759(Nano)	1385703(Nano)	**Thread 10:**	107519(Nano)	762272(Nano)
Average Wait Time for Computationally Threads: 113783(Nano)			**Average Wait Time for IOIntensive Threads:** 109567(Nano)		
Overall Run Time for both Computationally and IOIntensive Threads: 14514189(Nano)					



Results for Mixing IOIntensive and Computationally:

The following table, and screenshot below shows a mixture of both IOIntensive and Computationally threads. A Java class was created to mix the threads. Within the class contains a method that is used for displaying the performance metrics shown in the table. To mix the threads, I put all of them in an ArrayList, and used the Collections library to display them in random order.

Mixture of IOIntensive and Computationally Threads		
IOIntensive & Computationally Threads	**Wait Time**	**Turn Around Time**
IOIntensive Thread 4:	96376(Nano)	763477(Nano)
IOIntensive Thread 1:	103002(Nano)	866478(Nano)
Computationally Thread 6:	120771(Nano)	685172(Nano)
IOIntensive Thread 5:	127999(Nano)	710772(Nano)
Computationally Thread 10:	116555(Nano)	670414(Nano)
IOIntensive Thread 2:	104507(Nano)	739383(Nano)
Computationally Thread 8:	100592(Nano)	704748(Nano)
Computationally Thread 7:	89750(Nano)	729745(Nano)

IOIntensive Thread 3:	128601(Nano)	693605(Nano)
Computationally Thread 9:	88545(Nano)	732757(Nano)
Average Wait Time for IOIntensive Threads: 112097(Nano) **Average Wait Time for Computationally Threads:** 103242(Nano)		
Overall Runtime for both Threads: 7296551(Nano)		

```
Mixing both IOIntensive and Computationally Threads

IO Intensive Thread : 4                          IO Intensive Thread : 2
Wait time for: 96376                             Wait time for: 104507
Turn Around Time for: 763477                     Turn Around Time for: 739383

IO Intensive Thread : 1                          Computationally Thread : 8
Wait time for: 103002                            Wait time for: 100592
Turn Around Time for: 866478                     Turn Around Time for: 704748

Computationally Thread : 6                       Computationally Thread : 7
Wait time for: 120771                            Wait time for: 89750
Turn Around Time for: 685172                     Turn Around Time for: 729745

IO Intensive Thread : 5                          IO Intensive Thread : 3
Wait time for: 127999                            Wait time for: 128601
Turn Around Time for: 710772                     Turn Around Time for: 693605

Computationally Thread : 10                      Computationally Thread : 9
Wait time for: 116555                            Wait time for: 88545
Turn Around Time for: 670414                     Turn Around Time for: 732757


          Average wait times for IOIntensive Threads: 112097
          Average wait times for Computationally Threads: 103242
          Overall run time: 7296551
```

Shortest Job First (SJF) Implementation:

The SJF scheduling algorithm is a non-preemptive scheduling algorithm that processes threads in ascending order from least to greatest. The sorting in done based on their burst time. This algorithm was implemented using Java, and assigned its own class called: ScheduleFirstJob.java. Within this class holds a method that is used to calculate a variety of performance metrics for each of the IOIntensive, and Computationally threads. To display the order of the threads in ascending order, I used a sorting algorithm known as, the bubble sort

method. The bubble sort method allowed me to compare each threads burst time, and then use that information to determine, which thread should start first. For calculating these times, I followed the Grants Chart, which is a bar chart that is used to illustrate how the scheduling algorithm works. I will provide an example of how it works. See figure 2 bellow:

See Figure 2:

Process	Burst Time
p_1	6
p_2	8
p_3	7
p_4	3

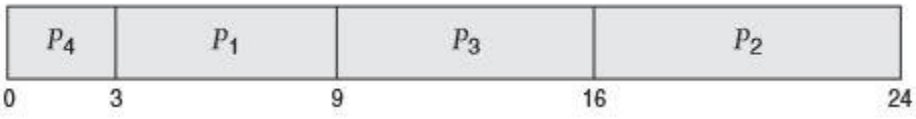


Figure 2 Retrieved from: (Silberschatz et al. 2013).

Since this scheduling algorithm puts the processes in ascending order from shortest job to longest job, the results from implementing this algorithm would look like the following:

Process	Burst Time
p_4	3
p_1	6
p_3	7
p_2	8

For calculating the wait time for the first process p_1, the wait time is 3 milliseconds, for process p_2, the wait time is 16 milliseconds, and for process p_3, the wait time is 9 milliseconds, and for process p_4, the wait time is 0 milliseconds. To calculate the average wait time, you would add up all the processes times, and divide by the total processes. For example, $(3 + 16 + 9 + 0) / 4 = 7$ milliseconds (Silberschatz et al. 2013). This algorithm shows that order

does matter in regards, to ordering each thread by its burst time in ascending order. This algorithm is considered optimal because it gives the minimum average waiting time for a given set of processes (Silberschatz et al. 2013). To calculate the turnaround time, it is the

$$\text{burst time} + \text{the wait time} = \text{turnaround time}.$$

Shortest Job First Results:

The following table, and screenshot below show that regardless of the order that the threads are in, they will be sorted in accordance to their burst time.

Schedule Job First (SJF)			
IOIntensive & Computationally Threads	**Burst Times**	**Total Wait Times**	**Turn Around Time**
Computationally Thread 9:	234496678(Nano)	0(Nano)	234496678(Nano)
Computationally Thread 7:	234496679(Nano)	234496678(Nano)	468993357(Nano)
IOIntensive Thread 3:	234496680(Nano)	468993357(Nano)	703490038(Nano)
Computationally Thread 10:	234496681(Nano)	703490038(Nano)	937986719(Nano)
IOIntensive Thread 1:	234496681(Nano)	937986719(Nano)	1172483401(Nano)
IOIntensive Thread 2:	234496682(Nano)	1172483401(Nano)	1406980083(Nano)
IOIntensive Thread 4:	234496683(Nano)	1406980083(Nano)	1641476767(Nano)
Computationally Thread 6:	234496684(Nano)	1641476767(Nano)	1875973451(Nano)
IOIntensive Thread 5:	234496685(Nano)	1875973451(Nano)	2110470136(Nano)
Computationally Thread 8:	234496685(Nano)	2110470136(Nano)	2344966822(Nano)
Average Wait Time: 1055235063(Nano) **Average Turn Around Time:** 12897317457(Nano) **Overall run time:** 6876107(Nano)			

As you can see below, it does not matter the order in which the threads go in, it matters which thread has the shortest burst time, wait time, and turnaround time.

```
Schedule Job First Algorithm (SJF)

Computationally Thread : 9:
Burst Time: 234496678
Wait Time: 0
Turn Around Time: 234496678

Computationally Thread : 7:
Burst Time: 234496679
Wait Time: 234496678
Turn Around Time: 468993357

IO Intensive Thread : 3:
Burst Time: 234496680
Wait Time: 468993357
Turn Around Time: 703490038

Computationally Thread : 10:
Burst Time: 234496681
Wait Time: 703490038
Turn Around Time: 937986719

IO Intensive Thread : 1:
Burst Time: 234496681
Wait Time: 937986719
Turn Around Time: 1172483401

IO Intensive Thread : 2:
Burst Time: 234496682
Wait Time: 1172483401
Turn Around Time: 1406980083

IO Intensive Thread : 4:
Burst Time: 234496683
Wait Time: 1406980083
Turn Around Time: 1641476767

Computationally Thread : 6:
Burst Time: 234496684
Wait Time: 1641476767
Turn Around Time: 1875973451

IO Intensive Thread : 5:
Burst Time: 234496685
Wait Time: 1875973451
Turn Around Time: 2110470136

Computationally Thread : 8:
Burst Time: 234496685
Wait Time: 2110470136
Turn Around Time: 2344966822


Average Wait Time: 1055235063
Average Turn Around Time: 12897317457
Overall run time: 6876107
```

Round Robin (RR) Implementation:

The Round Robin algorithm is a preemption scheduling algorithm that is designed for time sharing systems. This follows the First-Come-First Serve (FCFS) algorithm, where threads are dispatched in a first-in-first-out sequence. While, it is like the FCFS algorithm, what makes it different is that it can switch between processes (threads). The way this is done is by restricting the CPU time by using what is known as a quantum, or time slice. This works by the CPU scheduler picking the first process from the ready queue, or list, and then sets a timer to interrupt after the 1-time quantum, and then it dispatches the process (Silberschatz et al. 2013). In this case, there are two things that could happen. The first one is the process may have a CPU burst time of less than 1 quantum. If this is the case, then the process will release the CPU freely, and

move onto the next process in the queue or list. On the other hand, if the CPU burst time of the currently running process it greater than 1-time quantum, the timer will be alerted and caused an interruption to the operating system. This will cause a context switch to take place, and the currently running process will move to the end of the queue or list, and; therefore, then select the next process that is in the queue or list (Silberschatz et al. 2013).

Additionally, the RR scheduling algorithm that was implemented in my program was done in Java, and has its own class called: RoundRobin.java. Within the class contains two methods, one method that implements the suspend and resume methods that are offered in the concurrency library, and one that keeps an active count of the threads. The method that implements the suspend and resume methods controls when a thread will resume, and when it will be suspended based on the time slice that it is given. The method basically checks to see if the total number of threads is not equal to 0, and if that is true, it will resume the threads.

Furthermore, the method will then check to see if the current time of that thread is not equal to the time slice, and whether the thread is alive. If these conditions are true, the thread will be suspended, and will wait for one quantum. If it passes all those conditions, it will then give the next time slice to the next process. This is all surrounded in a try and catch method, so if there are any interruptions they will be detected. Finally, the method calculates the performance metrics to include: the wait time for each thread, average wait time for all threads, turnaround time for each thread, and average turnaround time for all threads. For calculating these times, I followed the Grants Chart, which is a bar chart that is used to illustrate how the scheduling algorithm works. I will provide an example of how it works. See figure 3 below:

Figure 3:

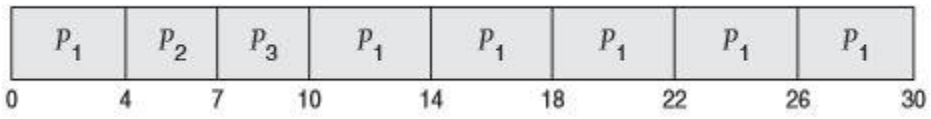


Figure 3 Retrieved from: (Silberschatz et al. 2013).

For calculating the wait time for the first process p_1, the wait time is 6 milliseconds (10 - 4), for process p_2, the wait time is 4 milliseconds, and for process p_3, the wait time is 7 milliseconds, to calculate the average wait time, you would add up all the processes times, and divide by the total processes. For example, $(6 + 4 + 7) / 3 = 5.66$ milliseconds (Silberschatz et al. 2013). This algorithm, like the FCFS algorithm shows that processes arrive does not matter.

Round Robin Results:

The table, and screenshot below shows the results after running the program using the Round Robin scheduling algorithm.

Round Robin (RR)					
IOIntensive Threads	**Wait Time**	**Turn Around Time**	**Computationally Threads**	**Wait Time**	**Turn Around Time**
Thread 1:	1286917(Nano)	841180(Nano)	**Thread 6:**	6405976(Nano)	1089047(Nano)
Thread 2:	2036540(Nano)	1247163(Nano)	**Thread 7:**	7471230(Nano)	1151088(Nano)
Thread 3:	3236117(Nano)	1105912(Nano)	**Thread 8:**	8545518(Nano)	1033329(Nano)
Thread 4:	4263724(Nano)	1136331(Nano)	**Thread 9:**	9531864(Nano)	1016463(Nano)
Thread 5:	5340120(Nano)	1140848(Nano)	**Thread 10:**	10466708(Nano)	1120670(Nano)
Average Wait Time: 5858471(Nano) **Average Wait Time:** 1088203(Nano) **Average Turn Around Time:** 10882031(Nano)					

```
Round Robin Algorithm (RR)

IO Intensive Thread : 1
Wait Time : 1286917
Turn Around Time: 841180

IO Intensive Thread : 2
Wait Time : 2036540
Turn Around Time: 1247163

IO Intensive Thread : 3
Wait Time : 3236117
Turn Around Time: 1105912

IO Intensive Thread : 4
Wait Time : 4263724
Turn Around Time: 1136331

IO Intensive Thread : 5
Wait Time : 5340120
Turn Around Time: 1140848

Computationally Thread : 6
Wait Time : 6405976
Turn Around Time: 1089047

Computationally Thread : 7
Wait Time : 7471230
Turn Around Time: 1151088

Computationally Thread : 8
Wait Time : 8545518
Turn Around Time: 1033329

Computationally Thread : 9
Wait Time : 9531864
Turn Around Time: 1016463

Computationally Thread : 10
Wait Time : 10466708
Turn Around Time: 1120670

Average waiting time is : 5858471
Average turn around time is : 1088203
Overall Runtime is: 10882031
```

Lessons Learned:

Throughout the course of the semester, there are many lessons to be learned when dealing with the three scheduling algorithms: First-Come-First Serve (FCFS), Schedule Job First (SJF), and Round Robin (RR). The FCFS algorithm was an effective algorithm to implement when all threads that needed to be scheduled had different performance times. The reason this is good is because if a thread comes in at a later time, the priority in which that thread is executed will not matter. With that being said, the SJF algorithm was an effective algorithm to implement when determining the length of the CPU's burst time, as well as, when to apply the FCFS algorithm if there is a case when the burst times of the threads are the same. The RR algorithm was great for timesharing, and context switching among threads. Each thread was assigned a time quantum, which was then used to determine the length in which that thread would run for.

Additionally, if we look at the performance measurements when using all the scheduling algorithms, it is evident that when calculating the wait times for each thread, and the average wait times for all threads, the FCFS, and RR algorithm took a lot longer than it did compared to the SJF algorithm. Another thing to mention is the changes in the variances every time the program ran. The times did change for all algorithms. If we look at the FCFS and RR algorithm, the reason the results changed each time we ran the program is because these are not algorithms that are used to check for ordering. With these two algorithms, the threads are dispatched accordingly to their arrival time; and therefore, the times will change almost every time the program is run. With the SJF algorithm, the times also changed every time the program ran; however, since the order in which the processes get executed is important, each time the program ran, the burst times for each thread would always display in ascending order.

Furthermore, it is important that I address the differences in wait time between IO Intensive threads, and Computationally threads. The main differences I found when running the IO Intensive threads before the Computationally threads, was that the wait times for IO Intensive threads were longer than the wait times for the Computationally threads. When running the Computationally threads before the IO Intensive threads, the average wait time appeared to be longer for the Computationally threads. Lastly, when running the IO Intensive and Computationally threads at random, the average wait time for completing the Computationally threads appeared to still be longer than the average wait time for completing the IO Intensive threads. Ultimately, I think that the average wait time varies between IO Intensive and Computationally threads because it depends on whether there are available resources.

Difficulties:

During this project, I ran into several difficulties when implementing the three scheduling algorithms that were talked about throughout the paper. I did not run into too much trouble when implementing the FCFS and SJF algorithm; however, I did struggle a lot when implementing the RR algorithm. The main problem I had was figuring out how the suspend, and resume methods should work. I followed the example that was provided by the instructor, and could implement both the suspend and resume methods; however, I am still struggling to understand if what I did is right. The way I addressed this issue was by throwing in some print statements to print out the thread id, and thread state. In the end, I am confident in my work; however, I am still concerned that my implementation of the RR algorithm is incorrect.

Conclusion:

This project outlined the many scheduling algorithms that are used when developing an operating system scheduler that handles multiple threads or processes. The goal behind this project was to develop a simulation of an operating system using the three scheduling algorithms mentioned throughout this paper. It was vital that I reported on the performance measurements of these scheduling algorithms. As a final point, the reason multithreading was implemented when developing a simulation of an operating system is because it provided fast execution of the program.

References:

IBM. (2017). Multithreading. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SS6SGM_3.1.0/com.ibm.aix.cbl.doc/PGandLR/concepts/cpthr01.htm

Krzyzanoeski, Paul. (2014). Threads: Multiple Flows of Execution Within a Process. *Rutgers College.* Retrieved from https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

Kishor, Lalit, Goyal, Dinesh. (2013). Comparative Analysis of Various Scheduling Algorithms. *International Journal of Advanced Research in Computer Engineering & Technology.* Retrieved from http://ijarcet.org/wp-content/uploads/IJARCET-VOL-2-ISSUE-4-1488-1491.pdf

Silberschatz, Abraham, Galvin Peter, Gagne, Greg. (2013). *Operating System Concepts: Chapter 6: CPU Scheduling.* Ninth Edition.