CPSC 321: Project Assignment 1

An
Introductory Study of Scheduling Algorithms

By

Terry Regner & Craig Lacey
2300xxxxx 2300xxxxx

February 10, 2005

1 Introduction

The goal behind CPSC 321 is to introduce the concepts and fundamentals of the structure and functionality of Operating Systems. There are four main components to the course: Scheduling, Deadlock detection, Memory management, and File Systems. The purpose of this assignment was to analyze different scheduling algorithms in a simulated system.

The assignment entailed the implementation of 3 different Scheduling algorithms: Shortest Process First, Round Robin, and Priority Sequence. As an aid to developing these algorithms we were given code for a **MOSS** (*Modern Operating System Simulator*) implementation. However, we chose to design our own from scratch to gain a better understanding of the details and intricacies involved in programming a process Scheduler. All programming is done in Java, so it is entirely object orientated. The **Design** section of this report contains class diagrams for the program developed.

2 Problem Statement

We will be simulating three CPU scheduling algorithms in java: Shortest Process First (SPF), Round Robin (RR) and Priority Sequence(PS). The purpose of the project is to gain a deeper understanding of the algorithms and to gather performance data to enable us to analyze their individual strengths and weaknesses. Data will be collected and analyzed for each of the algorithms and comparisons will be made based on such performance metrics as: Average Turn around Time, Average Normalized Turn around Time, Throughput, and CPU Utilization.

The restrictions of this project being implemented as a simulation, and not as a real system environment, requires the generation of random values for processes that are being scheduled. The randomness of these values can have a significant impact on the data collected. As an attempt to minimize the effects of randomness in the simulation a range of data values for specific attributes were given.

The following are attributes given and their value ranges:

- **CPUR -** Range of CPU times from 1 – 500.
- **IONR -** Range of the Number of I/O's for a process from 1 – 10.
- **IOWR -** Range of time spent blocking for I/O's given as (**CPUR** * 100).

Other values that were used in gauging the performance of the algorithms are: Quantum time (q), the total number of processes generated (N), the intervals between IO blocks, the arrival time, and exit time.

All the data collected from the algorithms will be used to develop a performance metrics analysis of each. The results from the data gathered are included in the **Results** section of this report.

3 Solution Strategy

The first step in completing this assignment was a careful analysis of the problem description assigned. This involved looking for the objects, behaviors and properties that were detailed within the handout.

All possible objects that were required for the implementation of the algorithms were recorded and then we were able to assign the properties and behaviors of each object. The algorithms themselves were written last after all the generic programming was completed. This allowed us to split the work of writing the algorithms while ensuring that all the minor details of the program would be uniform regardless of the writer.

Flexibility and modularity were carefully considered in the implementation of our design. The Scheduler that we have developed can easily be adapted to support any number of different algorithms because of the modularity of its design. The added bonus of being portable comes from its development in the Java programming language.

There were several issues that we took into consideration for our implementation. The most important area that we had to deal with was the fact that we were dealing with a simulation and not a real system. A simulated logical clock was developed for the system which is incremented to resemble the behaviour of a real system clock. Processes used in the system are created in a process generation class which initializes a process queue implemented as a vector. Each of the processes created contains randomly generated values based on the data ranges given in the problem statement, additional values were added to the processes to aid in their manipulation and data collection. The details of the process class are available in the **Solution Design** section of this document.

A Common class was also developed to produce comprehensible output that displays the performance metric information in a clear and easily readable format, that can be used for any algorithm developed to run on the Scheduler.

4 Solution Design

4.1 Classes Developed:

The Scheduler class was designed as the driving class for our project. This class contains the essential member values such as presentTime, and processes which are used by all the algorithms studied. The Scheduler is used as the foundation for running the three different algorithms, and can easily be adapted to allow for any algorithm to be executed. The Scheduler has no class methods itself, but an example of how to run the different functions is shown below.

Scheduler

Memebers:

```
public static int presentTime = 0;
public static int processes = 0;
public static int queued = 500;
public static int quanta = 0;
public static int activeQuanta = 0;
```

Methods:

```
//Here the algorithms run functions are called as well as the process
//generation calls, and the call to display output.  An example is as follows


ProcessGen.processGen( processes );
SPF.run();
Common.output();


ProcessGen.processGen( processes );
RR.run(quanta);
Common.output();


ProcessGen.processGen( processes );
PS.run();
Common.output();

//The code above shows an example of how to run and display data for
//each of the three algorithms studied.
```

The name of this class is suffixed with "1" to differentiate between our process class the Java language library class "Process". This class is used by every algorithm used. It contains all of the necessary data values to keep track of the performance metric information recorded through the duration of the scheduling simulation. The only method of this class is the constructor which initializes the pid, cpuTime, numIO, nextIO, and waitIO.

Process1

Members:

```
public static int CPUR = 500;
private static int IONR = 10;
private static int IOWR = ( CPUR * 100 );
public int pid = -1;
public int priority = 0;
public int arrivalTime = 0;
public int exitTime = 0;
public int cpuTime = 0;              //Total time needed for process
public int numIO = 0;                //Total number of IO blocks needed
public int nextIO = 0;               //Time between IO blocks
public int waitIO = 0;               //Minimum time IO blocks last
public int timeWaited = 0;           //Time spent in IO queue
public int turnsCompleted = 0;       //Number of turns completed in RR
public int IOcounter = 0;            //Counts up to the next IO block
```

Methods:

public Process1(int pid) - The constructor takes one integer argument that is used to assign the Process ID. The rest of the necessary values are assigned values within the range of random values mentioned in the **Problem Statement**.

The ProcessGen class is used solely to generate the required number of processes that each algorithm requires. The number of processes to generate is specified in the Scheduler. This class does not need any members and contains only one method.

ProcessGen

Methods:

public static void processGen(int elements) - Generates the number of processes to be created, the processes that this function creates are stored in the processQueue, which is a member of the Queue class and used by all the algorithms

A container class, Queue, was developed as a means of storing each of the processes in the appropriate states, and to ease in the management of holding processes that were in an IO block state. This class contains no methods.

Queue

Members:

```
//Queues for the processes implemented with Vectors.
public static Vector ioQueue = new Vector();
public static Vector processQueue = new Vector( Scheduler.processes );
public static Vector activeQueue = new Vector( );
public static Vector mainQueue = new Vector();
public static Vector auxiliaryQueue = new Vector();
```

A Simulated logical clock was incorporated in order to keep track of the system time, this is used to record arrival and exit times for each of the processes and to keep track of IO wait times and other necessary values. This class has no members; it simply updates the presentTime member of the Scheduler class.

Clock

Methods:

public static void tick( ) - The tick function increments the presentTime value which is a data member of the Scheduler class. This function is designed to update the time by 1 increment per clock tick.

The Common class that we developed includes functionality to display the gathered performance data from each of the algorithms to the screen, as well as do some house keeping duties, cleaning up the Queue's and resetting counters. This class also does the majority of the calculations for the output, and aids in the generation of random values.

Common

Members:

```
//The process declared here is used in collecting the data from the Queue
private static Process1 newProcess;
```

Methods:

public static int rand(int range) - This function returns a random value within the range specified by the passed in argument.

public static void output( ) - The output function calculates and displays the necessary performance metrics information gathered from the algorithm. It also resets the clock and clears the Queues to prepare for the next algorithm.

The Shortest Process First algorithm schedules the next process based on the shortest remaining cpuTime value found in the processes that have not completed. The SPF class was designed to implement this algorithm. The algorithm is executed by a call to the run() method in the class. This method performs all the necessary actions to carry out the Scheduling of the shortest process.

SPF

Members:

```
//Members declared to keep track of the processes completed, and
//to manage the different processes in use.
public static Process1 newProcess;
public static Process1 currentProcess;
public static int completed = 0;
```

Methods:

public static void waitforProcess( ) - The waitforProcess function is called when the algorithm is looking for another process to execute, but none are presently available. It check if it can generate a new process, and if not it decrements the ioWait time of all the processes in the ioQueue. It repeats this process until a new process can be executed, incrementing clock each time.

public static void run( ) - The run function drives the algorithm, getting and setting which process is active, it also manages the processes in the ioQueue, and ensures that the necessary data is kept during its run.

The Round Robin algorithm seeks to maintain fairness in the distribution of CPU time to each process by allotting a quanta value that determines how long each of the processes can utilize the CPU. Each process is scheduled based on the number of turns it's had at the CPU giving preference to the process that has completed the fewest number of turns using the CPU. The RR class was developed to carry out the Round Robin algorithm, using a man run() function call to start the scheduling.

RR

Members:

```
private static int quanta = 0;                    //Quanta for each process
private static int consumed = 0;                  //CPU time consumed
private static int completed = 0;                 //Completed processes
private static Process1 currentProcess;           //Active process
private static Process1 ioProcess;                //IO process
private static int turnNumber = 0;                //The current RR turn
private static boolean contextSwitch = false; //Context switch flag
private static int[] values = new int[1000];      //Stores cpuTime
```

Methods:

public static void run(int quantaValue) – The run method carries out the Round Robin scheduling algorithm, getting the next process from the active or process Queues depending on the arrival values, and the number of turns each process has completed. The following functions are used to aid the run function in ensuring the processes are treated fairly and that IO processes are handled properly.

public static Process1 getNextProcess( ) – This function is called from the run method and is used to obtain the first process that qualifies to be scheduled, taking processes that have not become active yet until all processes are active. After all processes are active it checks the activeQueue for the process that has completed the fewest number of turns at the CPU.

public static void checkIOQueue( ) – The purpose of this method is to manage the ioQueue processes, updating the amount of time each process has spent in its IO state, and removing those that have completed their IO wait times. Restoring those that have completed to the activeQueue.

The Priority Sequence algorithm uses a priority value to determine which process is scheduled next. Contention values for each process are determined when the process becomes active. Contention is assigned only once by the scheduler for each process. The total number of processes and contention are different. Contention value is incremented whenever new process is taken by scheduler and is decremented whenever a process completes. The priority of the process is updated as the processes use the CPU according to the formula given in the assignment guidelines.

PS

Members:

```
//The members of this class are used to manage the processes and to keep track
//of the number of processes that have completed their use of the CPU and exited
//the system
public static Process1 newProcess;
public static Process1 currentProcess;
public static int completed = 0;
```

Methods:

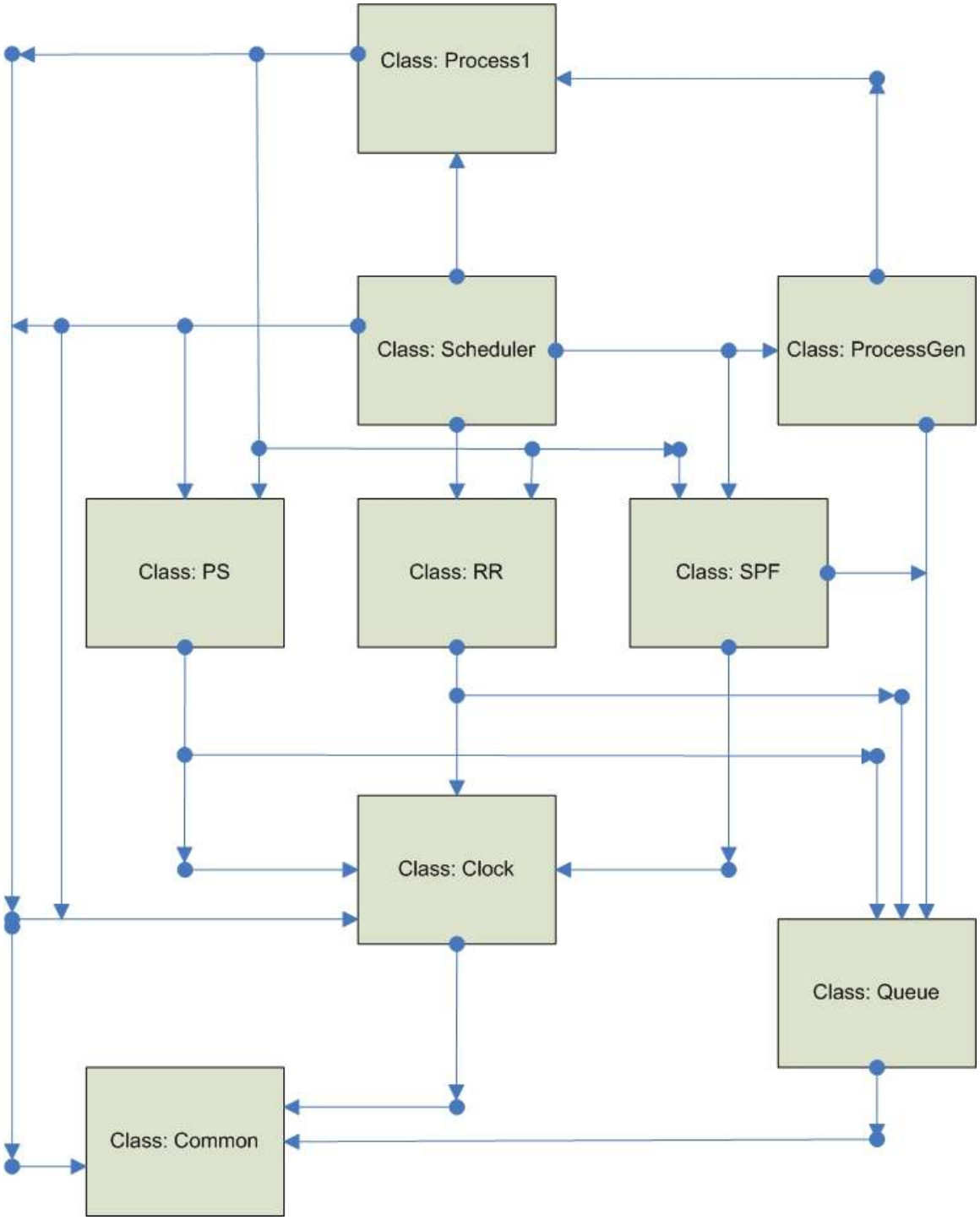
public static void waitforProcess( ) – This method is used to wait for the next available process to be scheduled for CPU use.

private static Process1 calculatePriority(int cc) – This function is used to calculate the priority values for each of the processes that remain in the system.

public static void nextProcess( ) – This method gets the next available process for scheduling.

public static void run( ) – This is the main method of this function, and executes the Priority Sequence algorithm for the processes in that are active in the system.

4.2 Conceptual Architecture Diagram of the Scheduler:



5 User Guide

The Scheduler software developed was implemented using the Java programming language and therefore is highly portable. The files necessary to execute the program are as follows:

Scheduler.java	Process1.java
ProcessGen.java	Common.java
Clock.java	Queue.java
SPF.java	RR.java
PS.java	

All of these files are necessary for the proper execution of the Scheduler. To compile this software on a Windows XP platform use the following instructions:
NOTE: You must have a Java Development Kit installed. Such as jdk1.5.0

1. Click on the Start button located on the lower right of the screen
2. Select the Run command from the right hand pane of the Start window
3. In the Open: dialog box type: cmd and press Enter or click the Okay button to open a Windows Command Prompt window.
4. Use the cd <directory> command to navigate to the folder where the files are located i.e. cd C:\Java\Scheduler
5. Compile the files using the javac command. i.e. C:\Java\Scheduler> javac *.java
6. The compiler creates an executable file name "Scheduler" to execute the program use the java command and this file name. i.e. C:\Java\Scheduler> java Scheduler

To run the scheduler on a Solaris platform use your bash console to navigate to the directory containing the necessary files. Then perform the following commands.

1. At the bash prompt use the javac command to compile the files. i.e $javac *.java
2. Run the executable file created using the java command. i.e. $java Scheduler

The parameters of the program are easily changed, by setting the values for quanta and processes in the Scheduler.java file. All the algorithms are called using their run functions. Examples are given below.

EX 1. Round Robin algorithm with 500 processes and a quanta of 150 would be executed as follows.

```
processes = 500                          //set the number of processes to create
ProcessGen.processGen(processes);  //creates 500 processes
quanta = 150;                            //sets the quanta value to 150
RR.run(quanta);                          //calls the Round Robin algorithm
```

```
Common.output( );                    //displays the data for RR after run completes
```

EX 2. Shortest Process First with 1000 processes, this algorithm doesn't use the quanta

```
processes = 1000;                    //set the number of processes to create
ProcessGen.processGen(processes);    //creates 1000 processes
SPF.run( );                          //run the SPF algorithm
Common.output( );                    //display the output
```

EX 3. Priority Sequence with 50 processes and a quanta value of 250

```
processes = 50;                      //set the number of processes to create
ProcessGen.processGen(processes);    //creates 50 processes
PS.run( );                           //run the PS algorithm
Common.output( );                    //display the output
```

NOTE: To ensure that the program is properly compiled all the files need to be in the same directory, and no other java files should be present.

The output function displays the information assembled to the screen in a format that allows for easy inspection of the results. The **Results** section of this document contains the output format that is displayed, with the values collected from running each algorithm with the data input given in the problem assignment.

6 Results

The following pages contain the information for each of the algorithms used in the Scheduler. Graphical data is provided for each of the algorithms. Each of the three algorithms was run 3 times to gather solid data for comparison between each. The values that were assembled to compare each of these algorithms were the Turn Around Time, the Normalized Turn Around Time, the Throughput, and the CPU Utilization. All of the resulting data

The graphs and output values are given starting with Shortest Process First on the next page.

6.1 Performance Metric Results for Shortest Process First (SPF)

Run # 1

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
500	57184.13	3.89	0.004154	0.13

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	20599.02	0.63	0.008081	10.37

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1500	18493.77	0.42	0.010417	16.79

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
2000	13687.38	0.20	0.013210	23.10

Run # 2

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
500	56015.21	4.32	0.004022	0.48

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	22394.71	0.75	0.007528	9.49

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1500	16306.46	0.30	0.010734	17.38

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
2000	12697.66	0.18	0.013358	23.85

Run # 3

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
500	61504.60	4.92	0.003948	0.12

Shortest Process First Performance Metrics

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	21113.71	0.63	0.007758	9.36

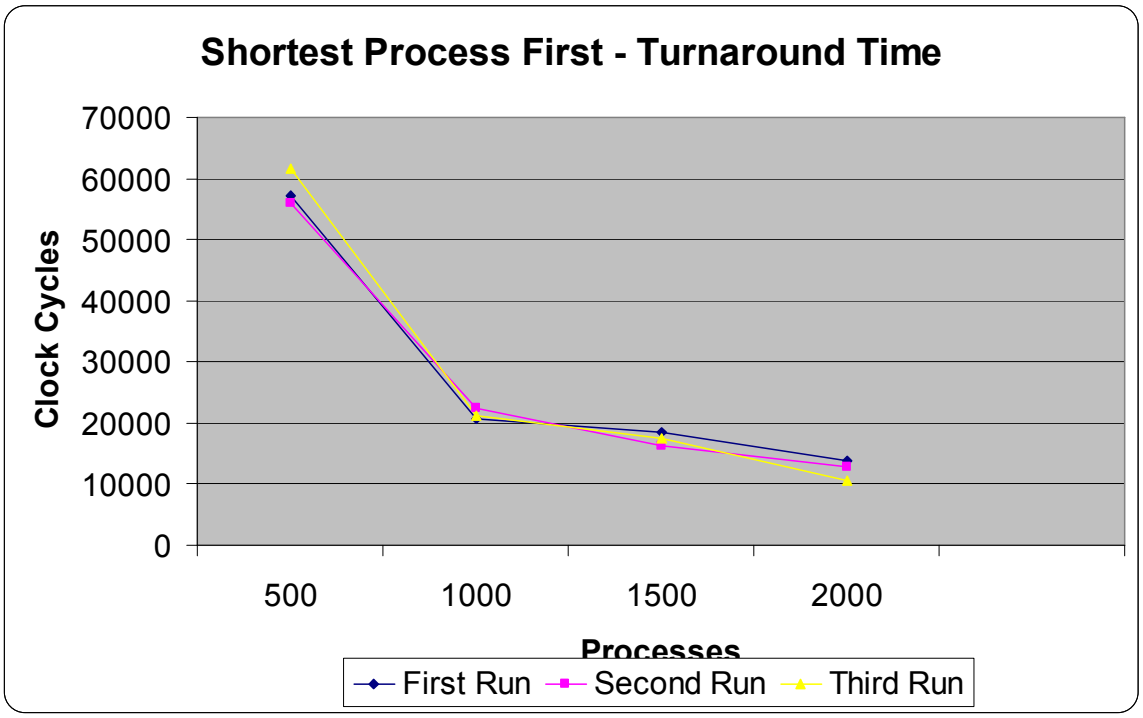
Shortest Process First Performance Metrics

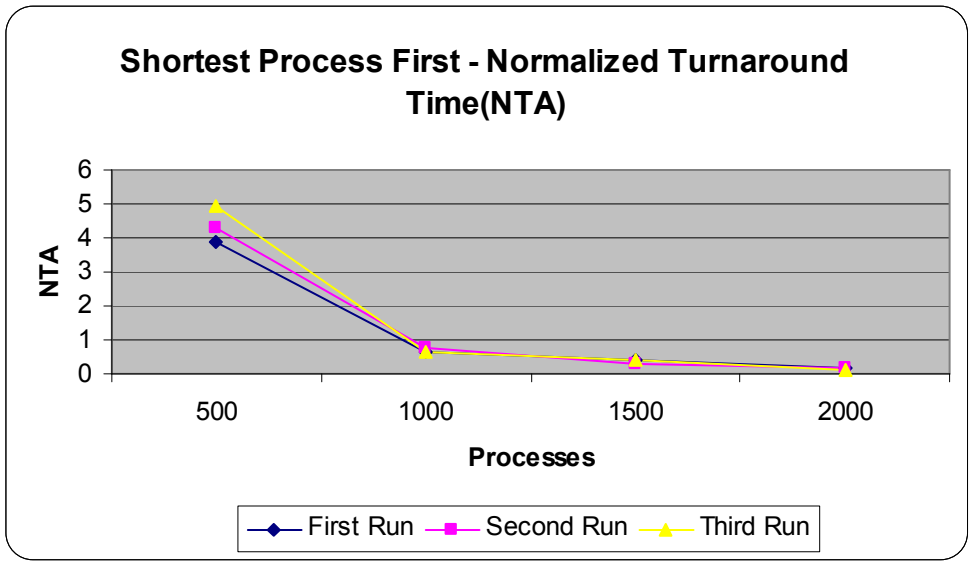
Processes	TA Time	NTA Time	Throughput	CPU Utilization
1500	17619.80	0.42	0.010342	16.08

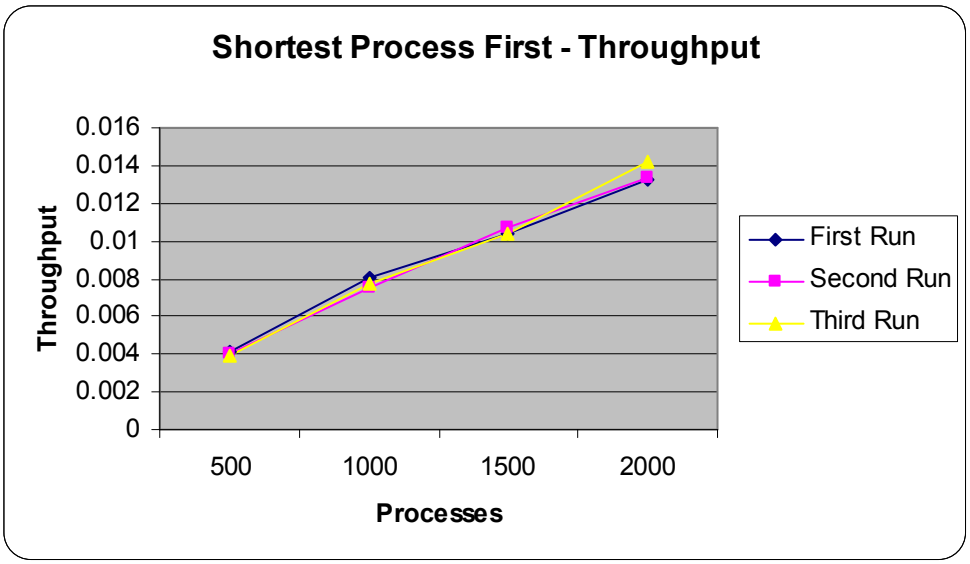
Shortest Process First Performance Metrics

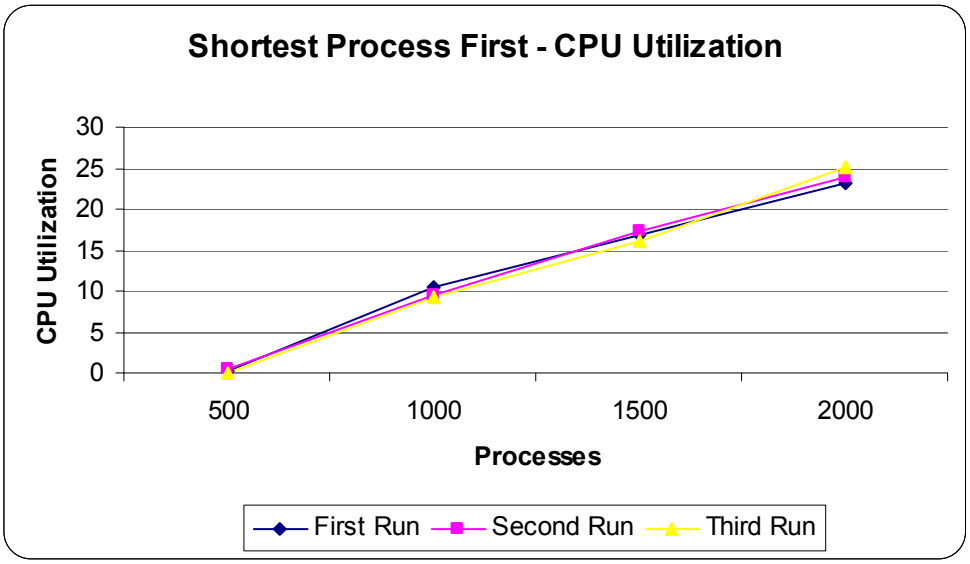
Processes	TA Time	NTA Time	Throughput	CPU Utilization
2000	10543.94	0.14	0.014170	25.16

Graphical Representation of the resulting data






Shortest Process First - Throughput
Throughput
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
500
1000
1500
2000
Processes
First Run
Second Run
Third Run


Shortest Process First - CPU Utilization
CPU Utilization
30
25
20
15
10
5
0
500
1000
1500
2000
Processes
First Run
Second Run
Third Run

6.2 Performance Metric Results for Round Robin (RR)

Run # 1

Round Robin Performance Metrics – Quanta = 50

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	8467.76	0.08	0.000302	8.18

Round Robin Performance Metrics – Quanta = 100

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	10077.44	0.10	0.000303	7.75

Round Robin Performance Metrics – Quanta = 150

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	6797.66	0.06	0.000344	7.84

Round Robin Performance Metrics – Quanta = 200

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	4839.70	0.04	0.000483	11.20

Round Robin Performance Metrics – Quanta = 250

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	5185.42	0.07	0.000532	13.02

Press any key to continue...

Run # 2

Round Robin Performance Metrics – Quanta = 50

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	7800.72	0.09	0.000290	6.71

Round Robin Performance Metrics – Quanta = 100

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	10035.18	0.12	0.000307	8.40

Round Robin Performance Metrics – Quanta = 150

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	6538.32	0.08	0.000326	7.94

Round Robin Performance Metrics – Quanta = 200

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	7574.84	0.09	0.000466	11.61

Round Robin Performance Metrics – Quanta = 250

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	6285.18	0.11	0.000502	11.64

Press any key to continue...

Run # 3

Round Robin Performance Metrics – Quanta = 50

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	10317.58	0.13	0.000303	7.18

Round Robin Performance Metrics – Quanta = 100

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	8511.02	0.11	0.000314	9.32

Round Robin Performance Metrics – Quanta = 150

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	5938.94	0.06	0.000368	8.14

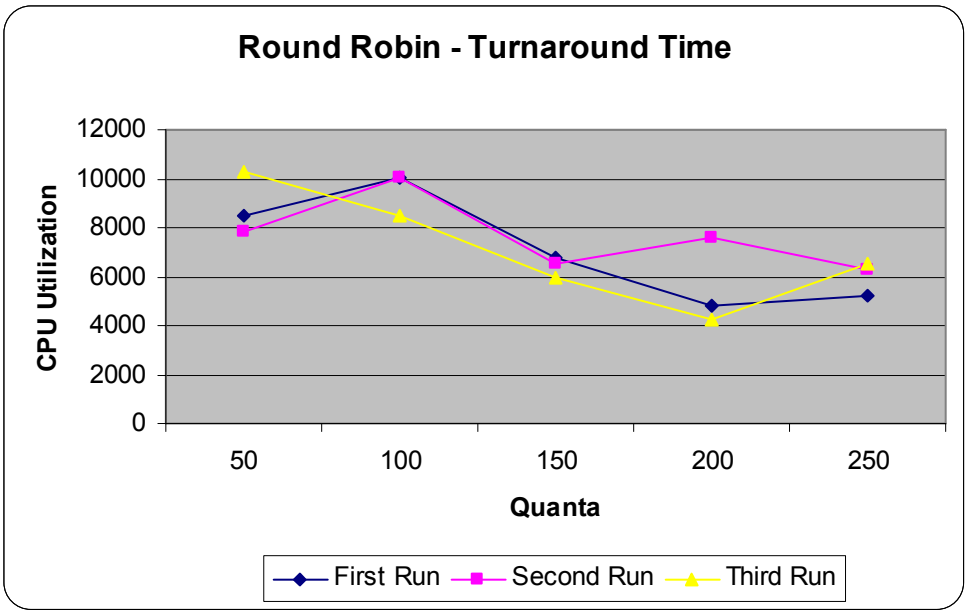
Round Robin Performance Metrics – Quanta = 200

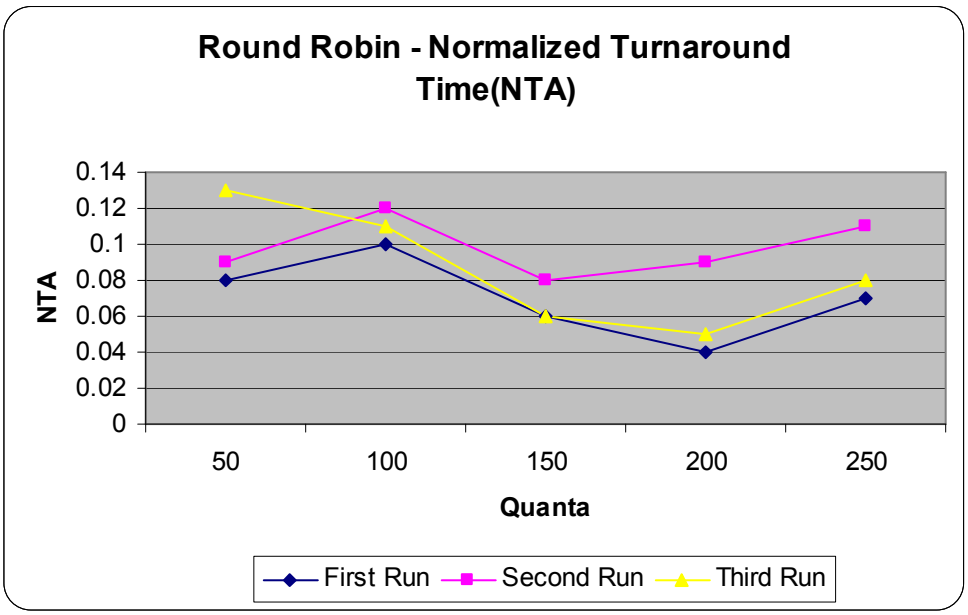
Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	4236.62	0.05	0.000404	11.05

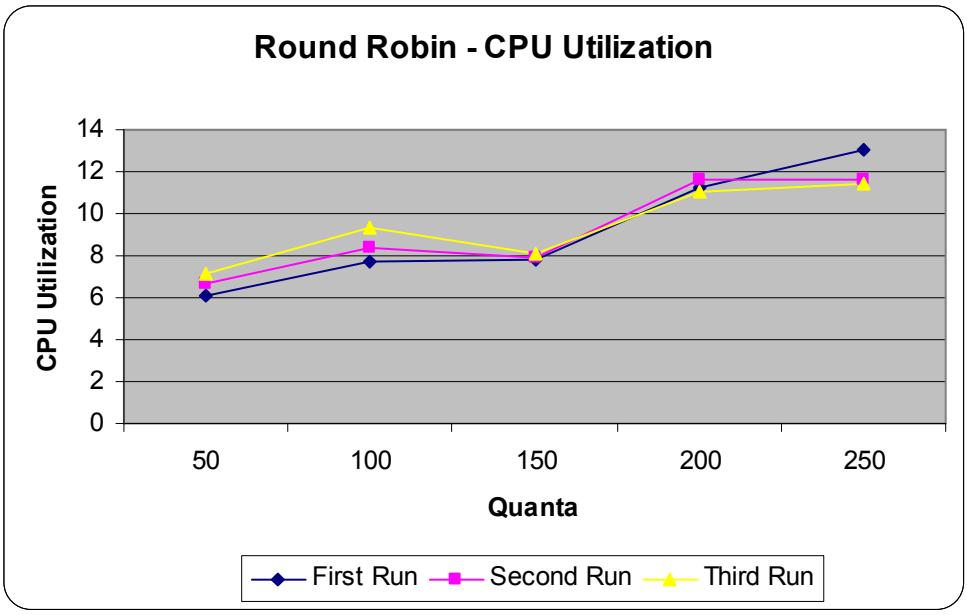
Round Robin Performance Metrics – Quanta = 250

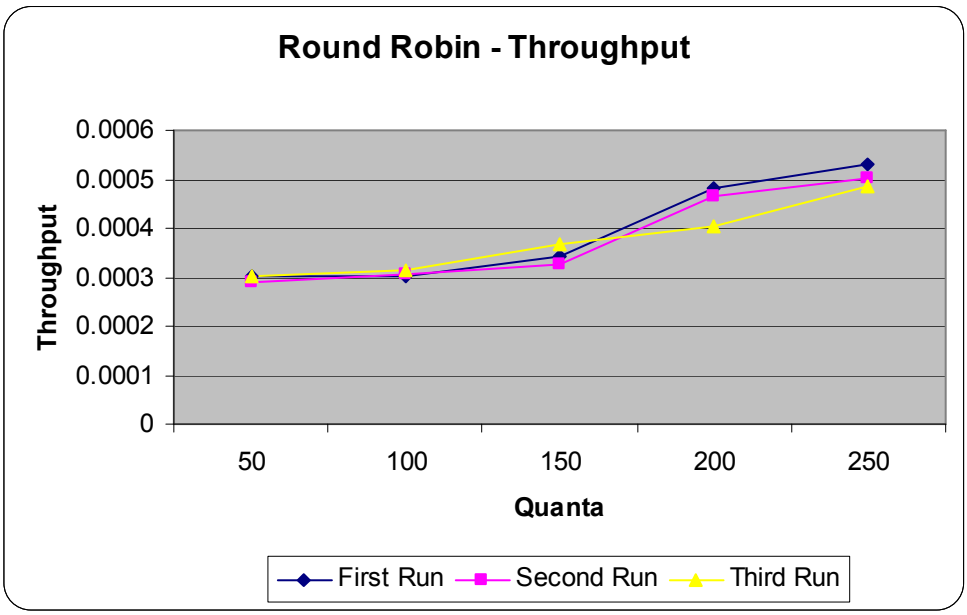
Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	6558.60	0.08	0.000487	11.46

Press any key to continue...


Round Robin - Turnaround Time
CPU Utilization
12000
10000
8000
6000
4000
2000
0
50
100
150
200
250
Quanta
First Run
Second Run
Third Run


Round Robin - Normalized Turnaround Time(NTA)
NTA
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
50
100
150
200
250
Quanta
First Run
Second Run
Third Run


Round Robin - CPU Utilization
CPU Utilization
14
12
10
8
6
4
2
0
50
100
150
200
250
Quanta
First Run
Second Run
Third Run


Round Robin - Throughput
Throughput
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
50
100
150
200
250
Quanta
First Run
Second Run
Third Run

6.3 Performance Metric Results for Priority Sequence (PS)

Run #1

Priority Sequence Performance Metrics – Quanta = 50

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	112532.28	6.56	0.004005	4.62

Priority Sequence Performance Metrics – Quanta = 100

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	152056.06	10.67	0.004011	0.35

Priority Sequence Performance Metrics – Quanta = 150

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	162871.25	15.63	0.003830	0.22

Priority Sequence Performance Metrics – Quanta = 200

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	149450.69	9.02	0.003980	1.29

Priority Sequence Performance Metrics – Quanta = 250

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	144614.46	11.62	0.003983	2.05

Run # 2

Priority Sequence Performance Metrics – Quanta = 50

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	102758.84	4.82	0.004154	8.53

Priority Sequence Performance Metrics – Quanta = 100

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	160541.29	14.05	0.003846	0.16

Priority Sequence Performance Metrics – Quanta = 150

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	157676.66	10.84	0.003899	0.56

Priority Sequence Performance Metrics – Quanta = 200

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	151724.60	14.48	0.003972	1.47

Priority Sequence Performance Metrics – Quanta - 250

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	147005.06	12.90	0.003970	1.74

Run # 3

Priority Sequence Performance Metrics – Quanta = 50

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	107306.65	6.83	0.003986	6.70

Priority Sequence Performance Metrics – Quanta = 100

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	171452.99	13.88	0.003706	0.04

Priority Sequence Performance Metrics – Quanta = 150

Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	163962.66	13.31	0.003811	0.36

Priority Sequence Performance Metrics – Quanta = 200

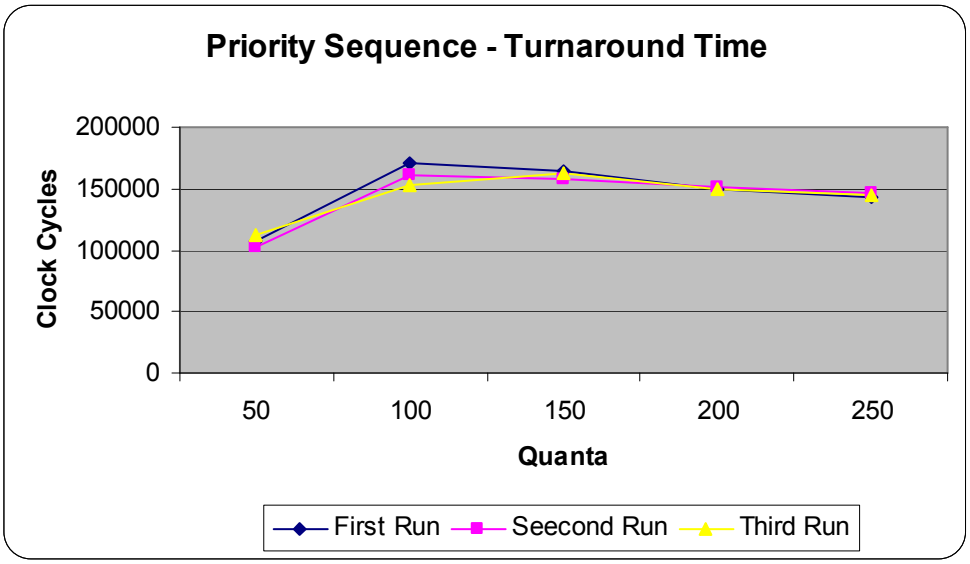
Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	149942.34	13.52	0.003923	0.57

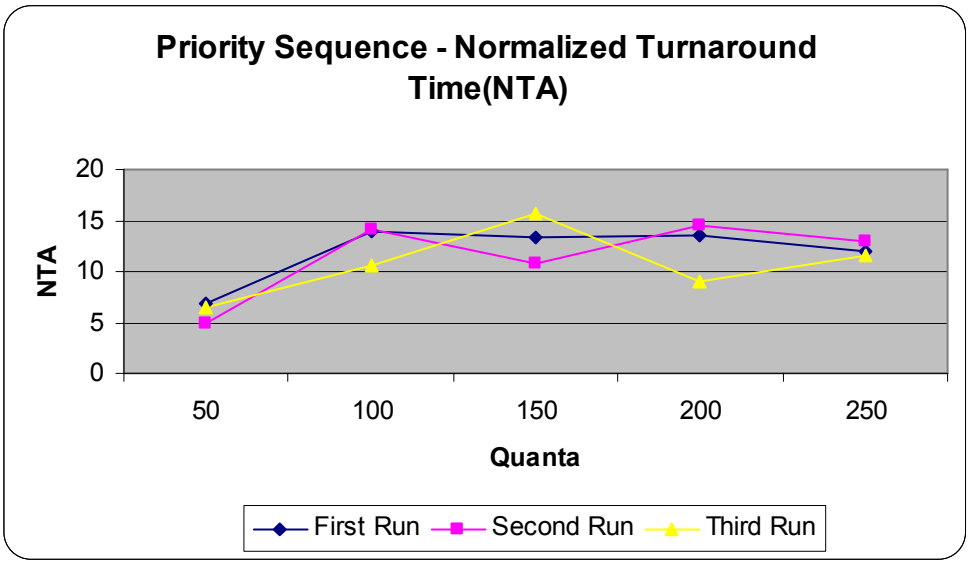
Priority Sequence Performance Metrics – Quanta = 250

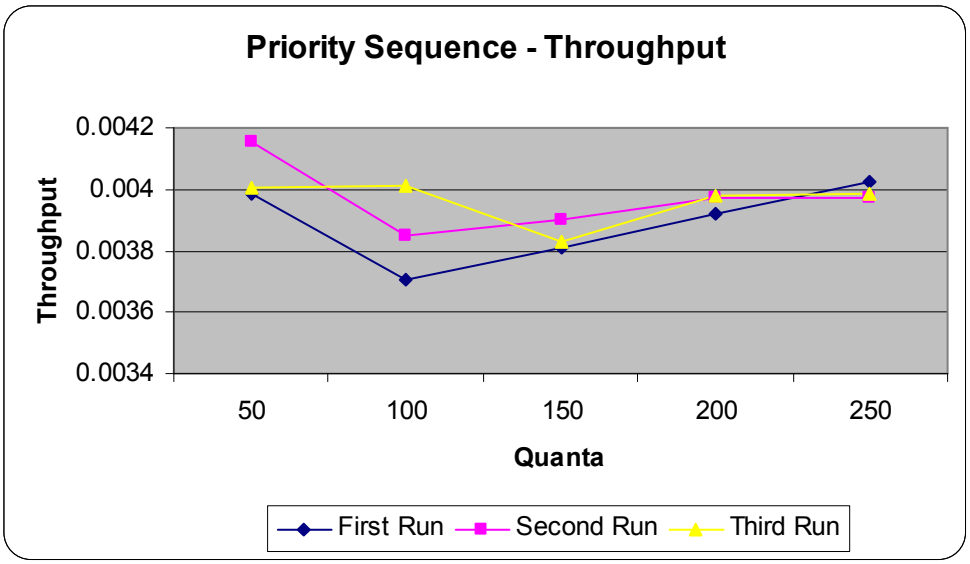
Processes	TA Time	NTA Time	Throughput	CPU Utilization
1000	143443.25	11.90	0.004027	1.92

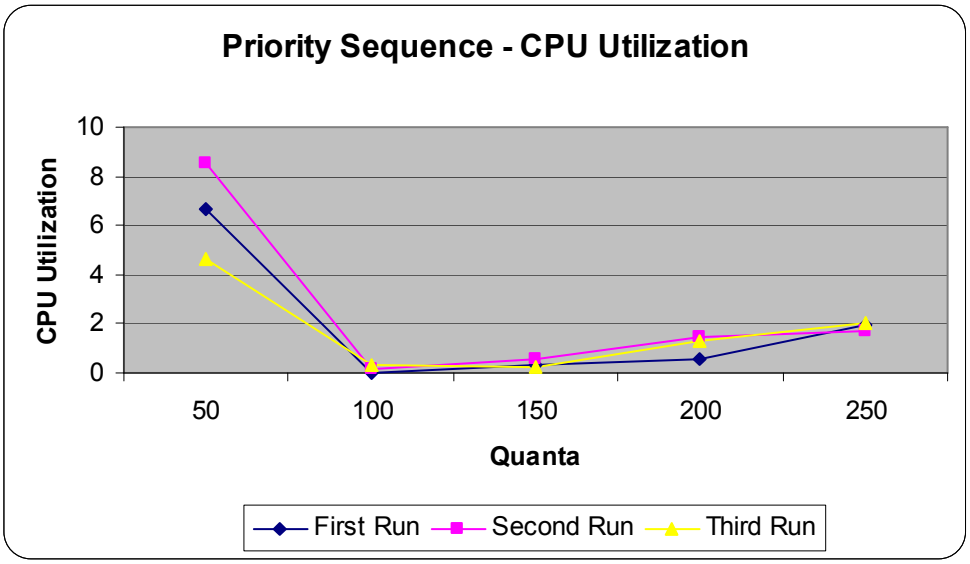
Press any key to continue...

Graphical Representation of the resulting data






Priority Sequence - Throughput
Throughput
0.0042
0.004
0.0038
0.0036
0.0034
50
100
150
200
250
Quanta
First Run
Second Run
Third Run


Priority Sequence - CPU Utilization
CPU Utilization
10
8
6
4
2
0
50
100
150
200
250
Quanta
First Run
Second Run
Third Run

7 Analysis and Conclusion

There were some difficulties encountered in programming the Scheduler, namely the learning curve of programming this project in java. Several hours were spent in debugging code that contained the kind of mistakes made from programming in an unfamiliar language. The algorithms are in working order, and produce quality output values given the completely random data values created. The Round Robin algorithm is in working order, but has a horrific time complexity, it is not recommended running this algorithm with more than a few hundred processes. The required run of 1000 processes for the 5 different quanta values took nearly 8 hours. The other two algorithms, Shortest Process First, and Priority Sequence, run in a much more user friendly timeframe, completing in just a few minutes in the worst case.

Comparing the three algorithms we can see that the CPU utilization values indicate that the Shortest Process First has the highest throughput values with CPU utilization times comparable to those of the Round Robin algorithm for 1000 processes and a quantum of 200. The Priority Sequence algorithm has the worst CPU utilization numbers; as well its throughput is significantly lower than the other two algorithms tested. Based on this information the Priority Sequence algorithm does not produce results that would motivate its use as a primary scheduling algorithm.

The Shortest Process First algorithm seems to produce the most favourable results regarding its throughput and CPU utilization. The turn around values for the SPF and the RR are comparable at 1000 processes with the RR algorithm having a slightly lower turn around time, and the SPF having a higher throughput.

Depending on the type of system that processes need to be scheduled in, either the Round Robin algorithm or the Shortest Process first are the two best choices out of these three algorithms. However the Shortest Process First seems to have the advantage based on the resulting data.