ICT1007 Operating Systems

Assignment 1 Report

Project Group: Class P3 (Group 03)

Group Members:

Soo Jing Xuan	2000917
Lee Xian Fu	2001995
Chong Fu Min	2002039
Gabriel Kok Heng Hong	2001701
Goh Jun Jie	2001191

Date of submission: 19 March 2021

Table of Contents

1. Contribution Statement

Name	**Description of Task**	**Percentage**	**Signature**	**Date**
Soo Jing Xuan	- Paper 2 - Performance Analysis & Discussion	20	[illegible]	19 March 2021
Lee Xian Fu	- Paper 1 - Performance Analysis & Discussion	20	[illegible]	19 March 2021
Gabriel Kok Heng Hong	- Paper 3 - Performance Analysis & Discussion	20	[illegible]	19 March 2021
Goh Jun Jie	- Paper 4 - Performance Analysis & Discussion	20	[illegible]	19 March 2021
Chong Fu Min	- Paper 5 - Performance Analysis & Discussion	20	[illegible]	19 March 2021
	Total	**100%**		

2. Algorithm Implementation

2.1 Efficient Shortest Remaining Time Round Robin (ESRR) [Paper 1]

The purpose of this paper is to show how ESRR (Efficient Shortest Time Round Robin) algorithm utilises RR (Round Robin) scheduling, SJF (Shortest Job First) algorithm and preemptive version of SRTF (Shortest Remaining Time First) to eradicate the downsides of RR and SRTF scheduling algorithms. CPU Scheduling algorithm would depend on the Turnaround time, number of context switches, waiting time and response time. When the last process has not reached the waiting queue,

Before the last process reaches the waiting queue, it will switch to the SRTF algorithm where the process with shortest burst will be processed in the CPU depending on the time quantum. New processes will be moved to the ready queue from the waiting queue and sorted from smallest to largest burst time.

Once the last process arrives, it will switch to the RR algorithm but it will be queued from smallest to biggest burst time. The shorter processes will be processed first hence it makes the algorithm efficient. The likelihood of starvation will be deterred when longer processes get moved to the CPU, in comparison with solely a SRTF algorithm.

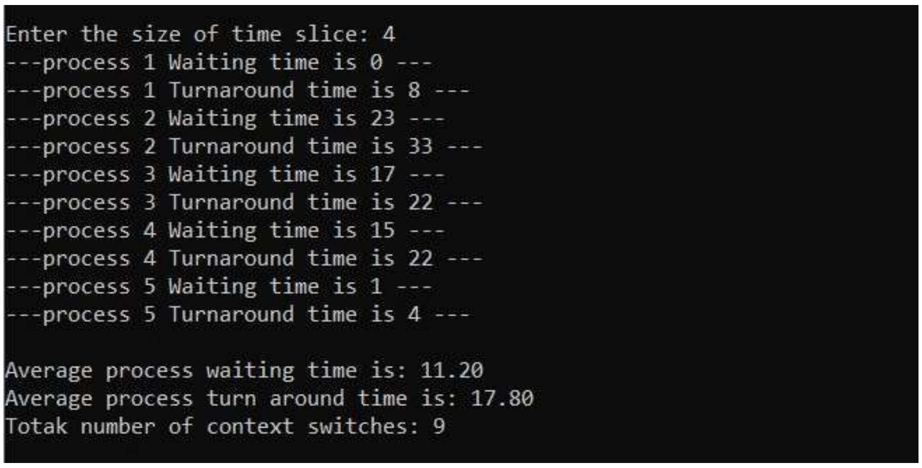
Paper's Test Cases

Process Name	P1	P2	P3	P4	P5
Burst Time	8	10	5	7	3
Arrival Time	0	0	2	5	7

Gantt Chart

P3	**P4**	**P2**	**P3**	**P4**	**P2**

11 15 19 23 24 27 33

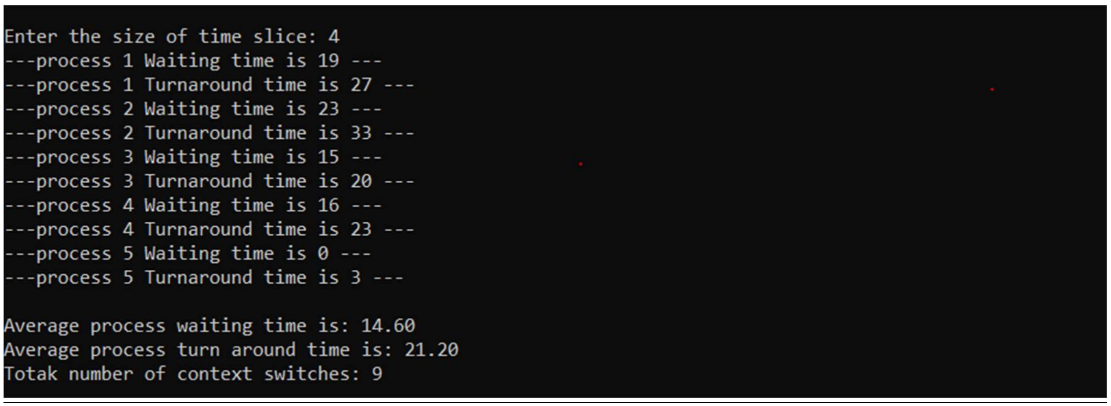


```
Enter the size of time slice: 4
---process 1 Waiting time is 0 ---
---process 1 Turnaround time is 8 ---
---process 2 Waiting time is 23 ---
---process 2 Turnaround time is 33 ---
---process 3 Waiting time is 17 ---
---process 3 Turnaround time is 22 ---
---process 4 Waiting time is 15 ---
---process 4 Turnaround time is 22 ---
---process 5 Waiting time is 1 ---
---process 5 Turnaround time is 4 ---

Average process waiting time is: 11.20
Average process turn around time is: 17.80
Totak number of context switches: 9
```

Illustration 2.1.1

Paper 1 Test Case with no arrival time

Process Name	P1	P2	P3	P4	P5
Burst Time	8	10	5	7	3
Arrival Time	0	0	0	0	0



```
Enter the size of time slice: 4
---process 1 Waiting time is 19 ---
---process 1 Turnaround time is 27 ---
---process 2 Waiting time is 23 ---
---process 2 Turnaround time is 33 ---
---process 3 Waiting time is 15 ---
---process 3 Turnaround time is 20 ---
---process 4 Waiting time is 16 ---
---process 4 Turnaround time is 23 ---
---process 5 Waiting time is 0 ---
---process 5 Turnaround time is 3 ---

Average process waiting time is: 14.60
Average process turn around time is: 21.20
Totak number of context switches: 9
```

Illustration 2.1.2

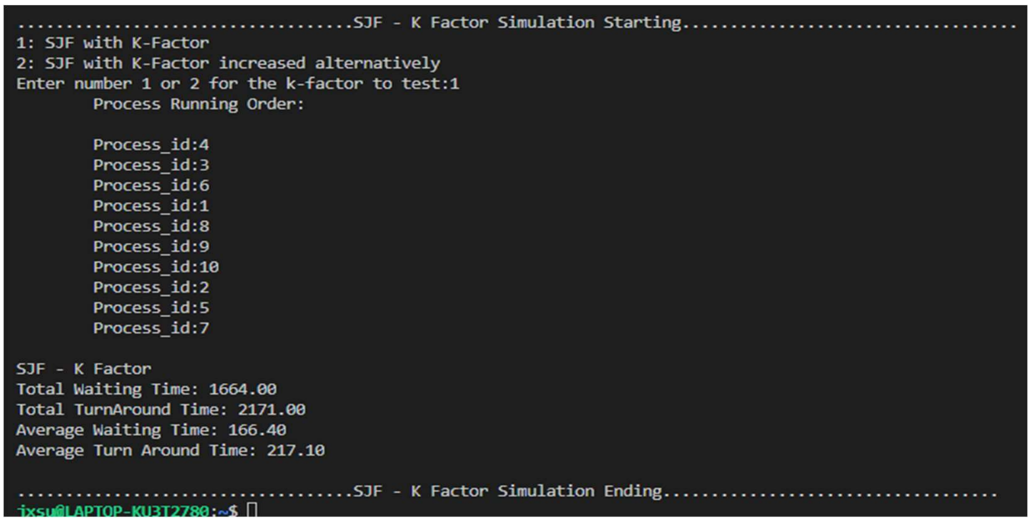
2.2 Shortest Job First with K Factor [Paper 2]

The purpose of paper 2 is to come up with a proposed SJF algorithm with k factor to remove or minimize the limitations posed by the SJF algorithm. Starvation is a problem in SJF where processes are denied necessary resources. Not all processes have equal chance to run and infinite postponement occurs when short processes are added continuously. The proposed k factor algorithm is not to replace the normal SJF, but to improve it and minimize starvation of longer processes.

Paper's test cases

The number of processes and the burst time for each process is in the table below.

Process	1	2	3	4	5	6	7	8	9	10
Burst	42	68	35	1	70	25	79	59	63	65



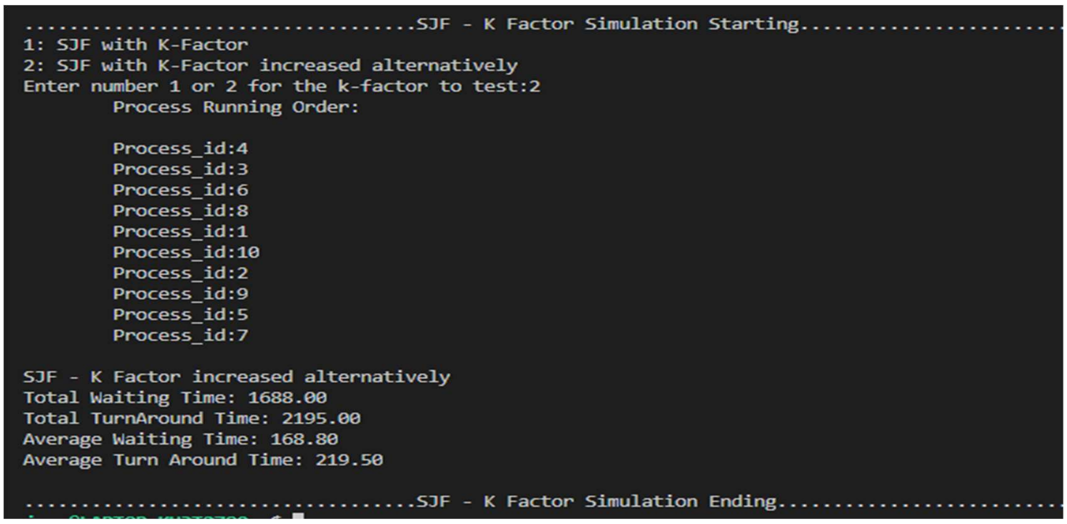
```
..................................SJF - K Factor Simulation Starting..................................
1: SJF with K-Factor
2: SJF with K-Factor increased alternatively
Enter number 1 or 2 for the k-factor to test:1
        Process Running Order:

        Process_id:4
        Process_id:3
        Process_id:6
        Process_id:1
        Process_id:8
        Process_id:9
        Process_id:10
        Process_id:2
        Process_id:5
        Process_id:7

SJF - K Factor
Total Waiting Time: 1664.00
Total TurnAround Time: 2171.00
Average Waiting Time: 166.40
Average Turn Around Time: 217.10

..................................SJF - K Factor Simulation Ending..................................
jxsu@LAPTOP-KU3T2780:~$ 
```

SJF with K-Factor 2.2.1

Gantt Chart

P4	P3	P6	P1	P8	P9	P10	P2	P5	P7

0 1 36 61 103 162 225 290 358 428 507



```
..................................SJF - K Factor Simulation Starting.........................
1: SJF with K-Factor
2: SJF with K-Factor increased alternatively
Enter number 1 or 2 for the k-factor to test:2
        Process Running Order:

        Process_id:4
        Process_id:3
        Process_id:6
        Process_id:8
        Process_id:1
        Process_id:10
        Process_id:2
        Process_id:9
        Process_id:5
        Process_id:7

SJF - K Factor increased alternatively
Total Waiting Time: 1688.00
Total TurnAround Time: 2195.00
Average Waiting Time: 168.80
Average Turn Around Time: 219.50

..................................SJF - K Factor Simulation Ending.........................
```

SJF K-Factor increased alternatively 2.2.2

Gantt Chart

P4	P3	P6	P8	P1	P10	P2	P9	P5	P7

0 1 36 61 120 162 227 295 358 428 507

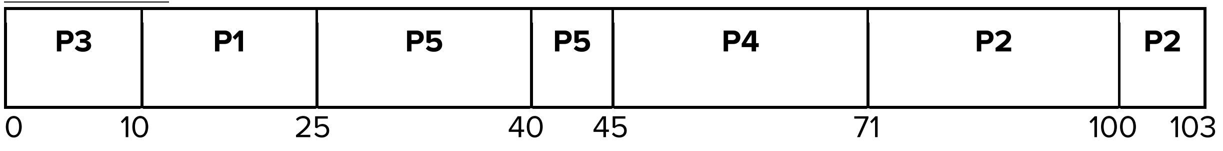
2.3 Improved Round Robin with Varying Time Quantum (IRRVQ) [Paper 3]

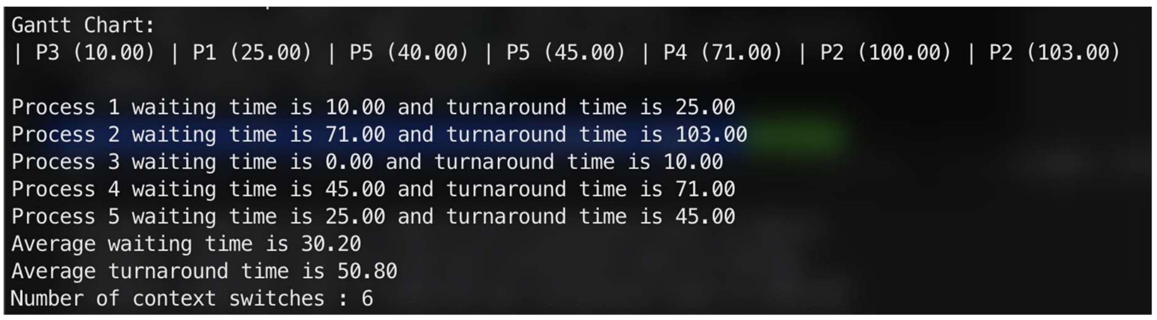
The proposed algorithm for this paper is an improved version of the Round Robin (RR) Algorithm with varying time quantum compared to a traditional Round Robin algorithm. The time quantum for this algorithm is determined by the average burst time of the tasks and thus can help with improving the average waiting time and turnaround time.
Below are test cases adapted from the paper that depict the performance of the algorithm.

<u>Paper 3 test case (with arrival time of all processes at t = 0)</u>

Processes	P1	P2	P3	P4	P5
Burst time	15	32	10	26	20

<u>Gantt Chart</u>



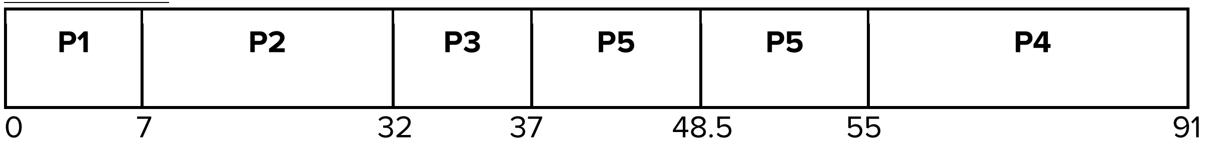


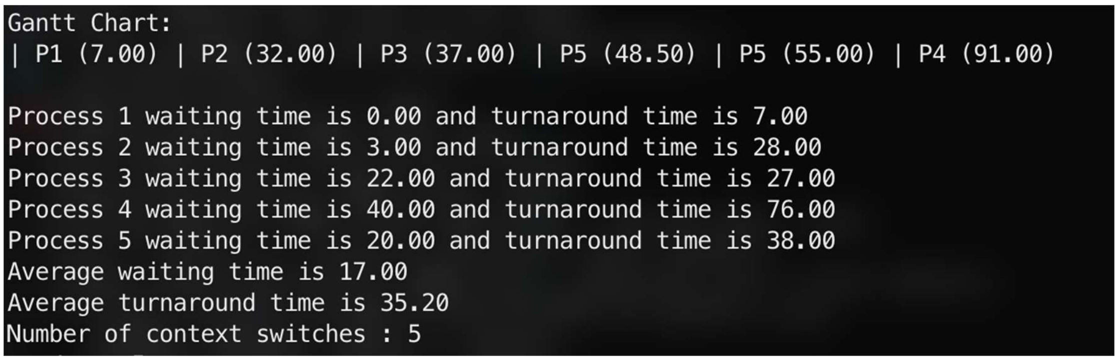
```
Gantt Chart:
| P3 (10.00) | P1 (25.00) | P5 (40.00) | P5 (45.00) | P4 (71.00) | P2 (100.00) | P2 (103.00)

Process 1 waiting time is 10.00 and turnaround time is 25.00
Process 2 waiting time is 71.00 and turnaround time is 103.00
Process 3 waiting time is 0.00 and turnaround time is 10.00
Process 4 waiting time is 45.00 and turnaround time is 71.00
Process 5 waiting time is 25.00 and turnaround time is 45.00
Average waiting time is 30.20
Average turnaround time is 50.80
Number of context switches : 6
```

<u>Paper 3 test case (with varying arrival time)</u>

Processes	P1	P2	P3	P4	P5
Burst Time	7	25	5	36	18
Arrival Time	0	4	10	15	17

<u>Gantt Chart</u>




```
Gantt Chart:
| P1 (7.00) | P2 (32.00) | P3 (37.00) | P5 (48.50) | P5 (55.00) | P4 (91.00)

Process 1 waiting time is 0.00 and turnaround time is 7.00
Process 2 waiting time is 3.00 and turnaround time is 28.00
Process 3 waiting time is 22.00 and turnaround time is 27.00
Process 4 waiting time is 40.00 and turnaround time is 76.00
Process 5 waiting time is 20.00 and turnaround time is 38.00
Average waiting time is 17.00
Average turnaround time is 35.20
Number of context switches : 5
```

2.4 Efficient Dynamic Round Robin (EDRR) [Paper 4]

The algorithm that the paper 4 has proposed is to create an efficient Round Robin (RR) with dynamic time quantum instead of fixed time quantum. The purpose of implementing this algorithm is to reduce the execution time of the algorithm without sorting any of the processes and also reduce the context switch, average waiting time and turnaround time of the processes. Hence, by implementing the proposed algorithm, the efficiency of an operating system will be improved and also resulted in a better embedded system. Below is the proof of concept using test cases found in the research paper.

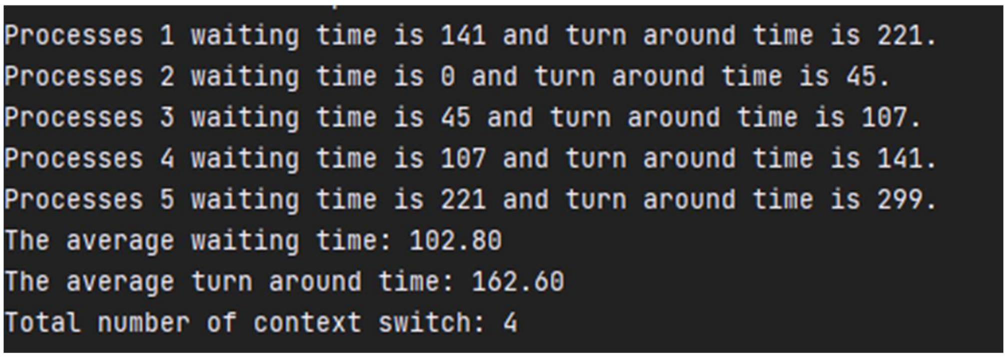
Paper's test cases (without arrival time)

Processes	P1	P2	P3	P4	P5
Burst time	80	45	62	34	78

Gantt Chart

P2	**P3**	**P4**	**P1**	**P5**

0 45 107 141 221 299

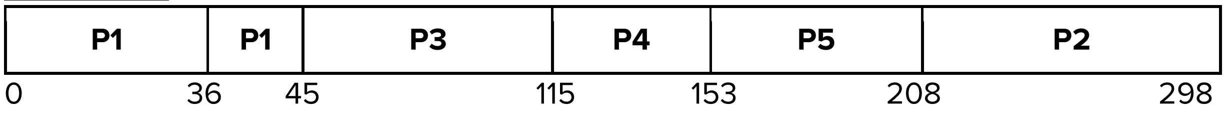

```
Processes 1 waiting time is 141 and turn around time is 221.
Processes 2 waiting time is 0 and turn around time is 45.
Processes 3 waiting time is 45 and turn around time is 107.
Processes 4 waiting time is 107 and turn around time is 141.
Processes 5 waiting time is 221 and turn around time is 299.
The average waiting time: 102.80
The average turn around time: 162.60
Total number of context switch: 4
```
Illustration 2.4.1

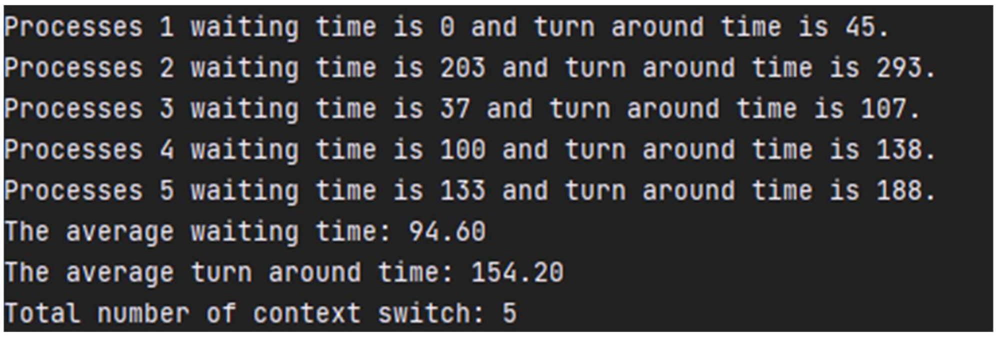
Paper's test cases (with arrival time)

Processes	P1	P2	P3	P4	P5

Burst Time	45	90	70	38	55
Arrival Time	0	5	8	15	20

Gantt Chart





```
Processes 1 waiting time is 0 and turn around time is 45.
Processes 2 waiting time is 203 and turn around time is 293.
Processes 3 waiting time is 37 and turn around time is 107.
Processes 4 waiting time is 100 and turn around time is 138.
Processes 5 waiting time is 133 and turn around time is 188.
The average waiting time: 94.60
The average turn around time: 154.20
Total number of context switch: 5
```

Illustration 2.4.2

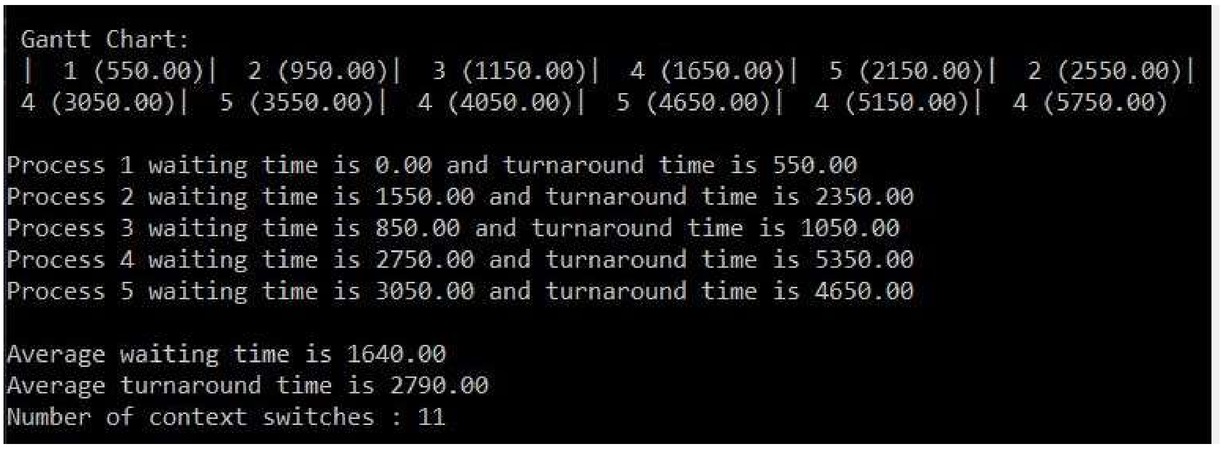
2.5 Improved Round Robin with Priority based Quantum (IRRPQ) [Paper 5]

The algorithm is a combination of three existing algorithms First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin (RR) into one. The purpose of this improved algorithm is to reduce the number of context switches. The priority of each process will determine the dynamic time quantum for execution; high priority processes will have a longer time quantum than low priority processes. This improved algorithm also executes the remaining burst time of a process in its time slice when its burst time hits a 'too low' condition, thus reducing context switch.

Below are the output of the given Table 1 and 2 (both have time quantum of 500)

Table 1

Processes	P1	P2	P3	P4	P5
Burst Time	550	800	200	2600	1600
Arrival Time	0	200	100	400	0
Priority	3	1	3	2	2



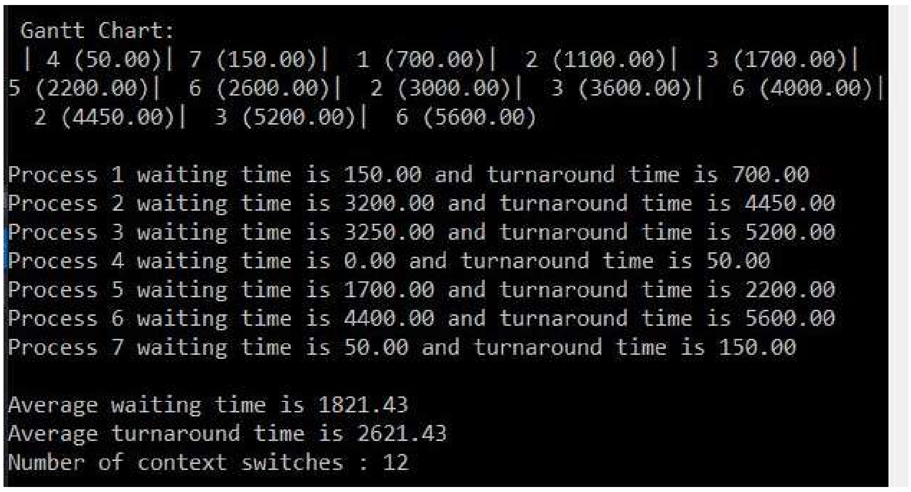
```
Gantt Chart:
|  1 (550.00)|  2 (950.00)|  3 (1150.00)|  4 (1650.00)|  5 (2150.00)|  2 (2550.00)|
4 (3050.00)|  5 (3550.00)|  4 (4050.00)|  5 (4650.00)|  4 (5150.00)|  4 (5750.00)

Process 1 waiting time is 0.00 and turnaround time is 550.00
Process 2 waiting time is 1550.00 and turnaround time is 2350.00
Process 3 waiting time is 850.00 and turnaround time is 1050.00
Process 4 waiting time is 2750.00 and turnaround time is 5350.00
Process 5 waiting time is 3050.00 and turnaround time is 4650.00

Average waiting time is 1640.00
Average turnaround time is 2790.00
Number of context switches : 11
```

Illustration 2.5.1

P1	P2	P3	P4	P5	P2	P4	P5	P4	P5	P4	P4
550	950	1150	1650	2150	2550	3050	3550	4050	4650	5150	5750

Table 2

Processes	P1	P2	P3	P4	P5	P6	P7
Burst Time	550	1250	1950	50	500	1200	100
Arrival Time	0	0	0	0	0	0	0
Priority	3	1	3	3	2	1	3



```
Gantt Chart:
| 4 (50.00)| 7 (150.00)|  1 (700.00)|  2 (1100.00)|  3 (1700.00)|
5 (2200.00)|  6 (2600.00)|  2 (3000.00)|  3 (3600.00)|  6 (4000.00)|
  2 (4450.00)|  3 (5200.00)|  6 (5600.00)

Process 1 waiting time is 150.00 and turnaround time is 700.00
Process 2 waiting time is 3200.00 and turnaround time is 4450.00
Process 3 waiting time is 3250.00 and turnaround time is 5200.00
Process 4 waiting time is 0.00 and turnaround time is 50.00
Process 5 waiting time is 1700.00 and turnaround time is 2200.00
Process 6 waiting time is 4400.00 and turnaround time is 5600.00
Process 7 waiting time is 50.00 and turnaround time is 150.00

Average waiting time is 1821.43
Average turnaround time is 2621.43
Number of context switches : 12
```

Illustration 2.5.2

P4	P7	P1	P2	P3	P5	P6	P2	P3	P6	P2	P3	P6
50	150	700	1100	1700	2200	2600	3000	3600	4000	4450	5200	5600

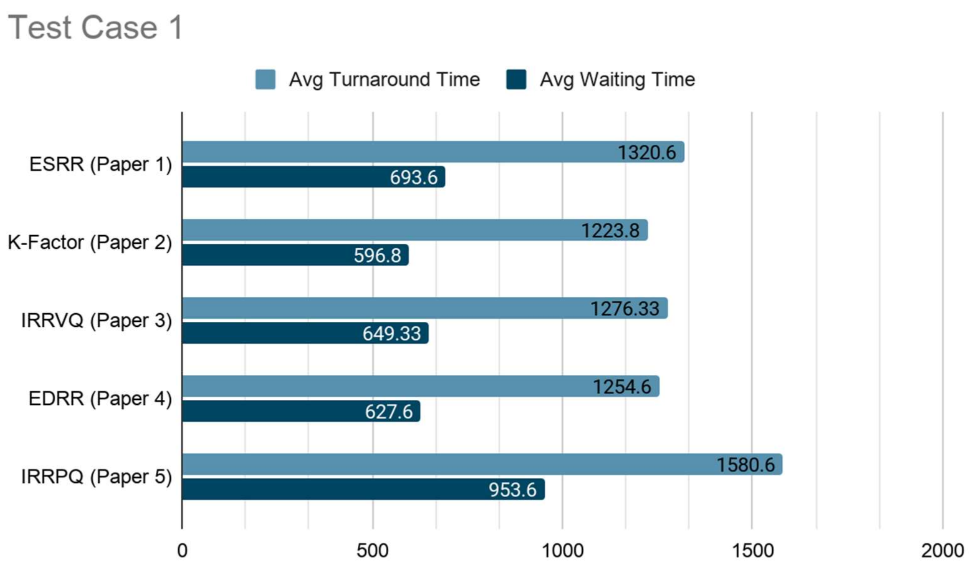
3. Performance Analysis and Discussion

With the implemented algorithms of all papers as seen in Section 2, we will be analysing and comparing the algorithms with two different test cases. Performance of the algorithms will be inferred from the average waiting time, average turnaround time and the number of context switches. The inclusion of calculating the number of context switches allows us to determine which algorithm has the maximum CPU utilization.

The first test case includes 5 processes having the same arrival time of 0. A process is given a higher burst time to simulate a heavy workload applied to the CPU.

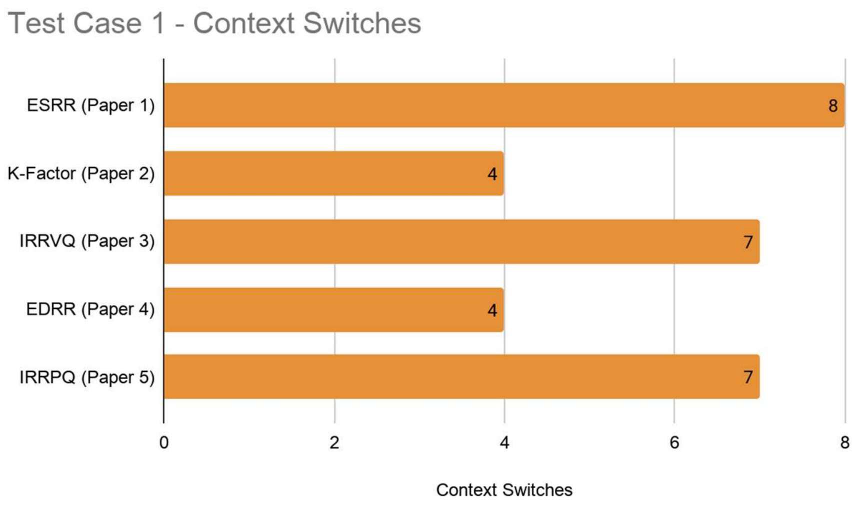
Test Case 1 (with arrival time of all processes at t = 0)

Process #	1	2	3	4	5
Burst Time	330	160	1653	346	646
Priority #	2	3	3	1	1



From the results gathered after conducting Test Case 1 on the algorithms, we have observed that the K-Factor algorithm (Paper 2) provides the lowest average turnaround time of 1223.8 time units and waiting time of 596.8 time units out of all the algorithms. This is most likely due to the fact that the K-Factor algorithm mostly uses SJF scheduling to execute processes, resulting in an overall shorter average waiting time and turnaround time across the processes. This is true given this test case as the k condition was only triggered once between process 4 and 1 with close proximity burst time making the algorithm behaving like a normal SJF.

Meanwhile, the Improved Round Robin with Priority based Time Quantum (IRRPQ) algorithm (Paper 5) has the longest average turnaround time and waiting time of 1580.6 time units and 953.6 time units respectively. This is due to the fact that the IRRPQ algorithm sets the time quantum for the round robin processing based on the process priority number and because the process with the largest burst time also has a high priority, the time quantum allocated is higher and therefore any process executed after this heavy process will have a longer waiting time which also contributes to a longer turnaround time.



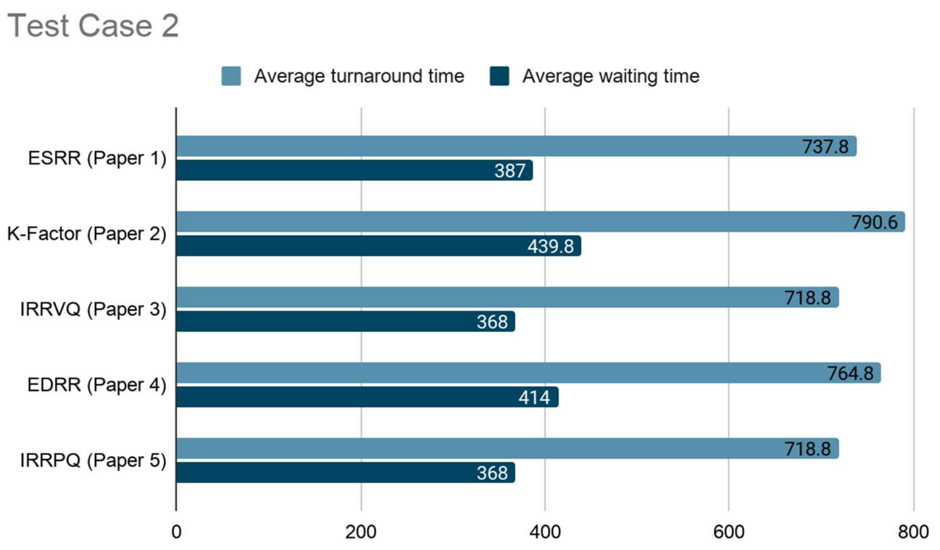
The performance analysis of context switches in test case 1 shows that paper 2 and 4 have the least number of context switches. This observation shows that processes are executed in their entirety for the K-Factor (Paper 2) and EDRR (Paper 4) algorithm. When there are more context switches in an algorithm for the same number of processes, it shows that the algorithm has less useful CPU utilization while a process with lesser context switch has more useful CPU utilization.

To improve the accuracy of the performance analysis, we included a second test case as different algorithms have different conditions for process selection and execution in the ready queue. For our Second Test Case, we have included varying arrival time for the different processes to see how a difference in the arrival time will affect the performance of the various algorithms. The deviation among the burst time for the processes are also smaller.

Test Case 2 (with varying arrival time)

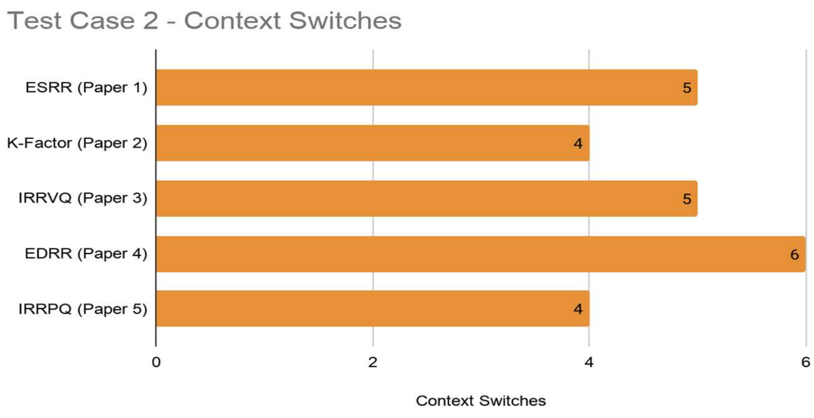
Process #	1	2	3	4	5
Burst Time	103	450	71	476	654

Arrival Time	19	59	0	48	3
Priority #	2	1	2	3	3



From the results we have gathered after running Test Case 2 on the algorithms, we can observe that both IRRVQ (Paper 3) and IRRPQ (Paper 5) tie to be the best algorithms for Test Case 2's dataset. This is because IRRVQ and IRRPQ behave quite similarly whereby processes with a lower burst time will be completed first and that time quantum can vary throughout the execution of the processes. Compared to Test Case 1 where IRRPQ (Paper 5) performed poorly, it performed better here as the differences in the burst time of the processes are lower, making the execution of processes more consistent and not having any one process significantly bringing up the waiting time and turnaround time.

On the other hand, the worst performing algorithm for Test Case 2 is the K-Factor (Paper 2) algorithm. In this test case, the k factor condition was triggered more than once and to prevent starvation of process. Processes with bigger burst time are forced to run before processes with smaller burst time to achieve fairness within processes. If the k factor condition is not triggered at all, the whole scheduling process does not differ much from a normal SJF.



The performance analysis of context switches in test case 2 shows that K-Factor (Paper 2) and IRRPQ (Paper 5) have the least number of context switches. This test case has taken the arrival time into consideration. But for K-Factor (Paper 2), the arrival time does not affect the computation as processes will execute to completion in a non-preemptive scheduler.

As can be observed in the chart above, IRRPQ (Paper 5) performed rather well here because processes have a smaller deviation in the burst time compared to Test Case 1 and that the process with the largest burst time (Process 5) also had the highest priority number of 3. This results in a higher time quantum allocated for process 5 which results in the process being able to complete with a lower number of context switches and in this Test Case even complete within a single cycle of Round Robin.

4. Conclusion

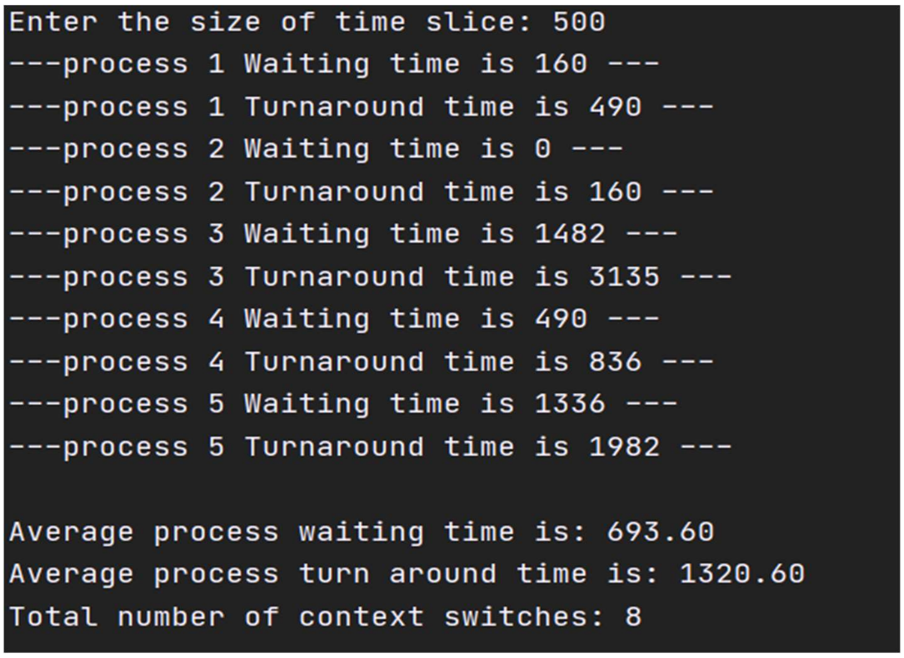
Across all of the algorithms, we have observed that the average turnaround time and waiting time are directly proportional to each other whereby algorithms with a lower waiting time will also have a lower turnaround time. This is because the turnaround time of processes are directly impacted by the waiting time and the burst time. As such, for any comparisons whereby the process has the same burst time, a lower waiting time will result in a lower turnaround time.

In conclusion, there is no single algorithm that is optimal and has the best performance. Performance of any algorithm will vary when different factors are taken into account as can be seen in both test cases that we have conducted. It will always be an unfair comparison as different schedulers take different data types into consideration. The performance for each algorithm is also dependent on the test case performed. There are many factors that can make an algorithm perform well and the algorithm that has the test case with the most beneficial factor for it will always outperform other algorithms.

5. Appendix

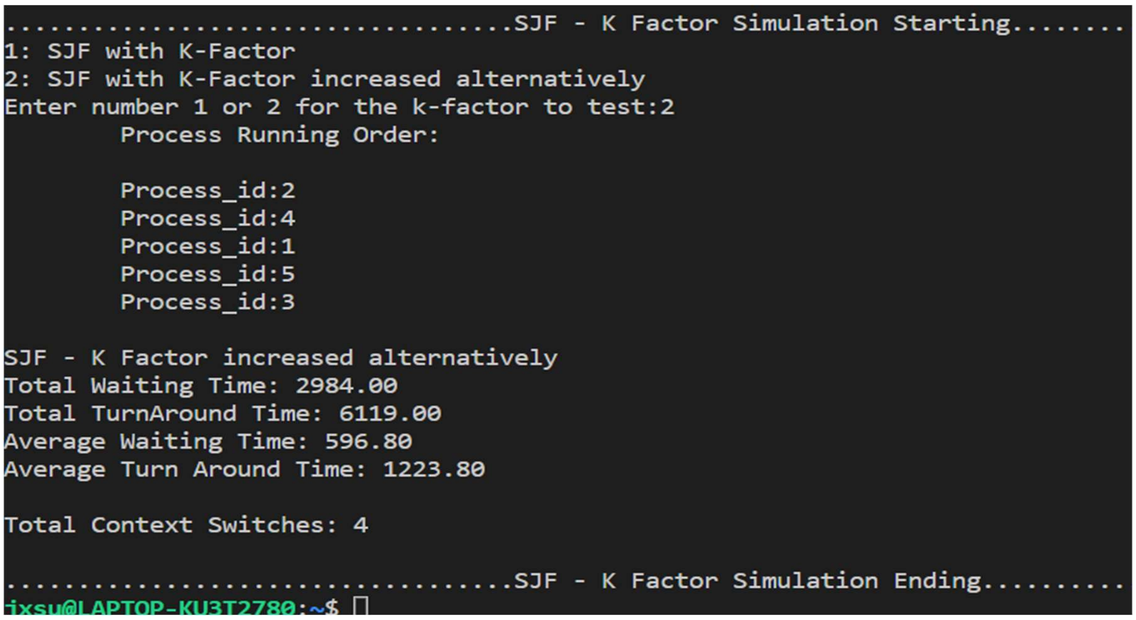
5. 1 Appendix A (Performance Analysis and Discussion)

Test case 1



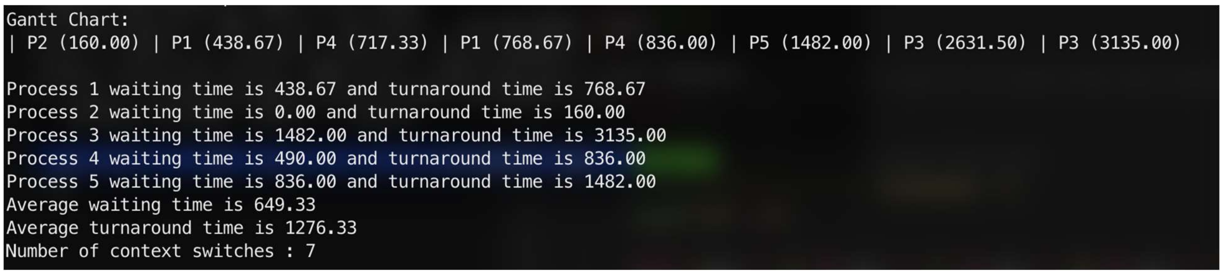
```
Enter the size of time slice: 500
---process 1 Waiting time is 160 ---
---process 1 Turnaround time is 490 ---
---process 2 Waiting time is 0 ---
---process 2 Turnaround time is 160 ---
---process 3 Waiting time is 1482 ---
---process 3 Turnaround time is 3135 ---
---process 4 Waiting time is 490 ---
---process 4 Turnaround time is 836 ---
---process 5 Waiting time is 1336 ---
---process 5 Turnaround time is 1982 ---

Average process waiting time is: 693.60
Average process turn around time is: 1320.60
Total number of context switches: 8
```

Efficient Shortest Time Round Robin Algorithm (Paper 1) on Test Case 1



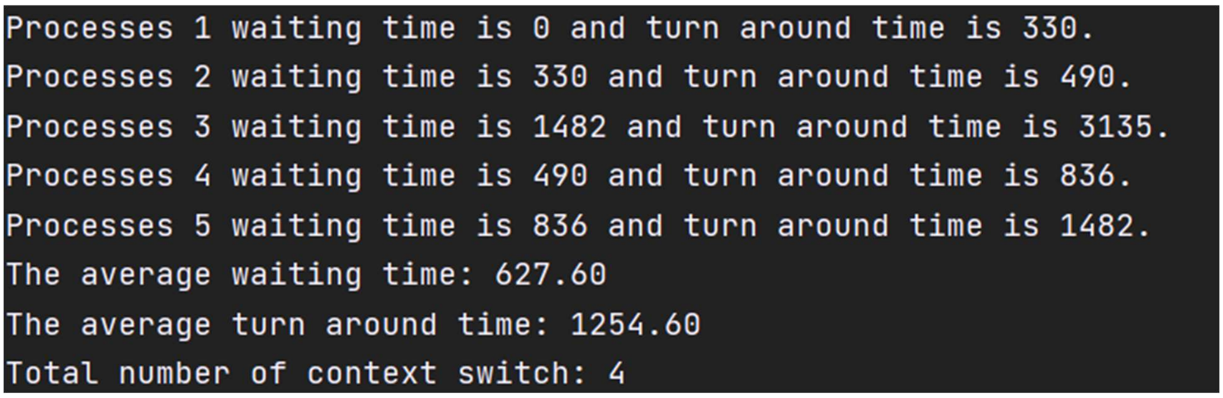
```
...................................SJF - K Factor Simulation Starting........
1: SJF with K-Factor
2: SJF with K-Factor increased alternatively
Enter number 1 or 2 for the k-factor to test:2
        Process Running Order:

        Process_id:2
        Process_id:4
        Process_id:1
        Process_id:5
        Process_id:3

SJF - K Factor increased alternatively
Total Waiting Time: 2984.00
Total TurnAround Time: 6119.00
Average Waiting Time: 596.80
Average Turn Around Time: 1223.80

Total Context Switches: 4

...................................SJF - K Factor Simulation Ending..........
jxsu@LAPTOP-KU3T2780:~$
```

SJF with K factor increased alternatively (Paper 2) on Test Case 1



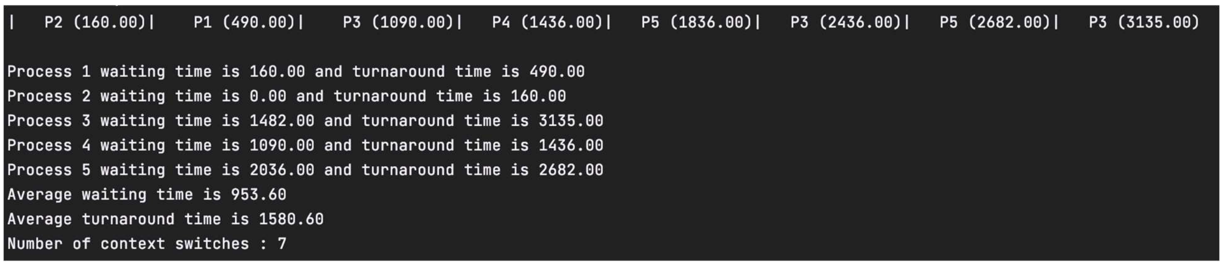
```
Gantt Chart:
| P2 (160.00) | P1 (438.67) | P4 (717.33) | P1 (768.67) | P4 (836.00) | P5 (1482.00) | P3 (2631.50) | P3 (3135.00)

Process 1 waiting time is 438.67 and turnaround time is 768.67
Process 2 waiting time is 0.00 and turnaround time is 160.00
Process 3 waiting time is 1482.00 and turnaround time is 3135.00
Process 4 waiting time is 490.00 and turnaround time is 836.00
Process 5 waiting time is 836.00 and turnaround time is 1482.00
Average waiting time is 649.33
Average turnaround time is 1276.33
Number of context switches : 7
```

Improved Round Robin with varying time quantum algorithm (Paper 3) on Test Case 1



```
Processes 1 waiting time is 0 and turn around time is 330.
Processes 2 waiting time is 330 and turn around time is 490.
Processes 3 waiting time is 1482 and turn around time is 3135.
Processes 4 waiting time is 490 and turn around time is 836.
Processes 5 waiting time is 836 and turn around time is 1482.
The average waiting time: 627.60
The average turn around time: 1254.60
Total number of context switch: 4
```

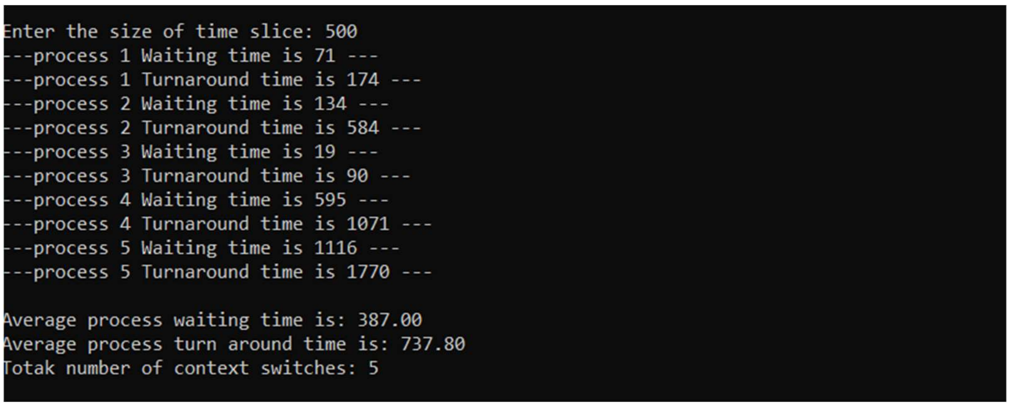
Efficient Dynamic Round Robin Algorithm (Paper 4) on Test Case 1



```
|   P2 (160.00)|    P1 (490.00)|    P3 (1090.00)|   P4 (1436.00)|   P5 (1836.00)|   P3 (2436.00)|   P5 (2682.00)|   P3 (3135.00)

Process 1 waiting time is 160.00 and turnaround time is 490.00
Process 2 waiting time is 0.00 and turnaround time is 160.00
Process 3 waiting time is 1482.00 and turnaround time is 3135.00
Process 4 waiting time is 1090.00 and turnaround time is 1436.00
Process 5 waiting time is 2036.00 and turnaround time is 2682.00
Average waiting time is 953.60
Average turnaround time is 1580.60
Number of context switches : 7
```

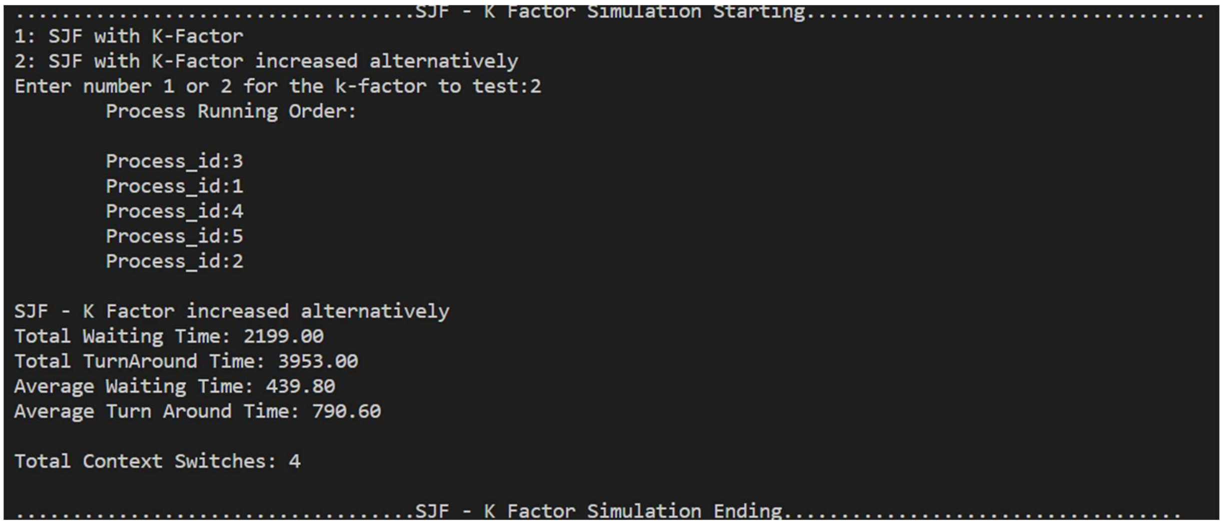
Improved Round Robin CPU Scheduling Algorithm (Paper 5) on Test Case 1

Test case 2



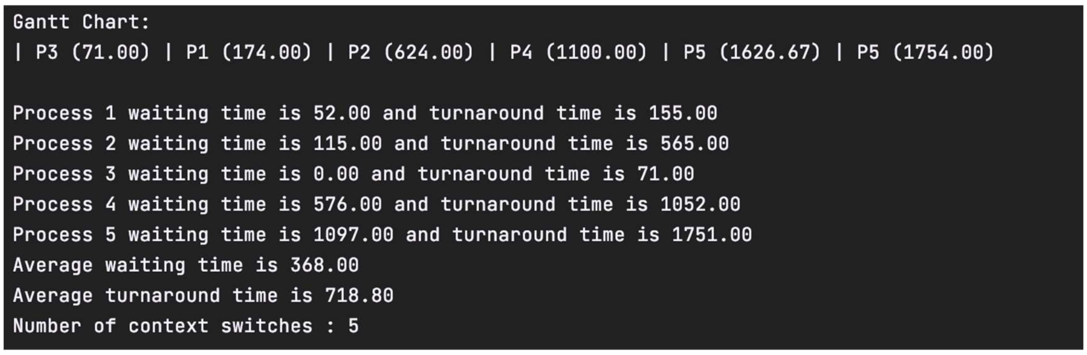
```
Enter the size of time slice: 500
---process 1 Waiting time is 71 ---
---process 1 Turnaround time is 174 ---
---process 2 Waiting time is 134 ---
---process 2 Turnaround time is 584 ---
---process 3 Waiting time is 19 ---
---process 3 Turnaround time is 90 ---
---process 4 Waiting time is 595 ---
---process 4 Turnaround time is 1071 ---
---process 5 Waiting time is 1116 ---
---process 5 Turnaround time is 1770 ---

Average process waiting time is: 387.00
Average process turn around time is: 737.80
Totak number of context switches: 5
```

Efficient Shortest Time Round Robin (Paper 1) on Test Case 2



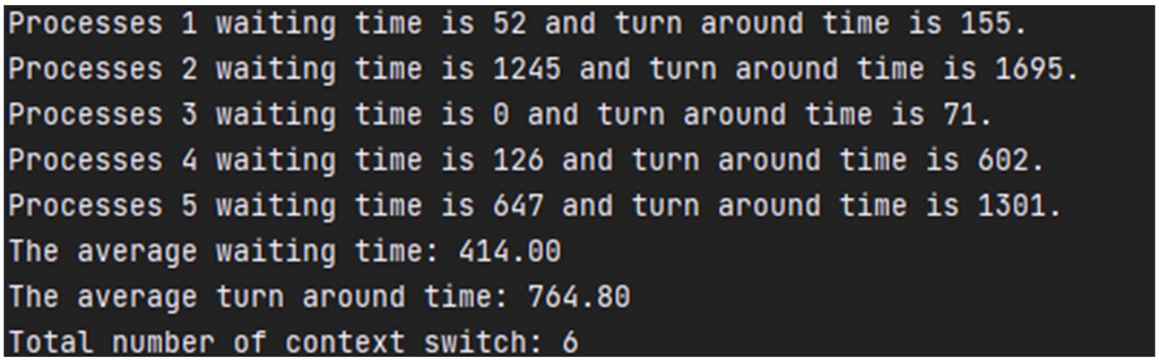
```
..................................SJF - K Factor Simulation Starting..................................
1: SJF with K-Factor
2: SJF with K-Factor increased alternatively
Enter number 1 or 2 for the k-factor to test:2
        Process Running Order:

        Process_id:3
        Process_id:1
        Process_id:4
        Process_id:5
        Process_id:2

SJF - K Factor increased alternatively
Total Waiting Time: 2199.00
Total TurnAround Time: 3953.00
Average Waiting Time: 439.80
Average Turn Around Time: 790.60

Total Context Switches: 4

..................................SJF - K Factor Simulation Ending..................................
```

SJF with K factor increased alternatively (Paper 2) on Test Case 2



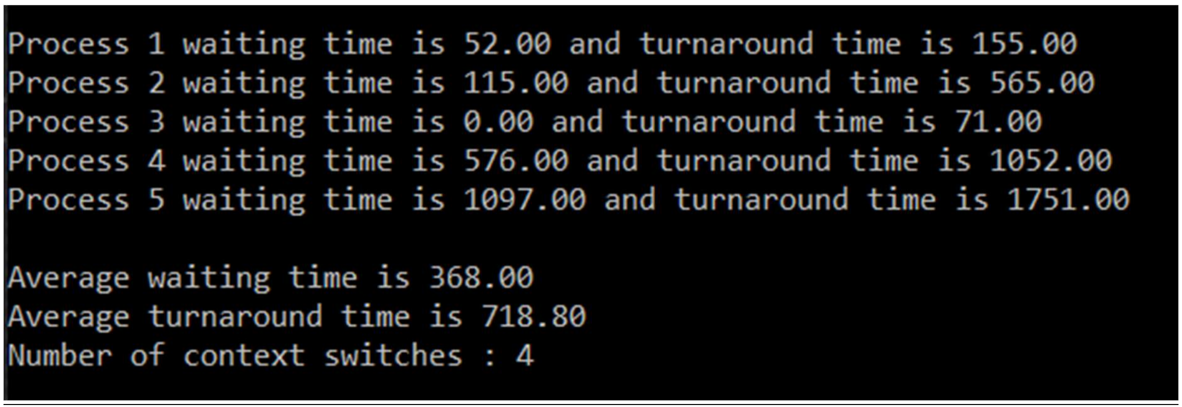
```
Gantt Chart:
| P3 (71.00) | P1 (174.00) | P2 (624.00) | P4 (1100.00) | P5 (1626.67) | P5 (1754.00)

Process 1 waiting time is 52.00 and turnaround time is 155.00
Process 2 waiting time is 115.00 and turnaround time is 565.00
Process 3 waiting time is 0.00 and turnaround time is 71.00
Process 4 waiting time is 576.00 and turnaround time is 1052.00
Process 5 waiting time is 1097.00 and turnaround time is 1751.00
Average waiting time is 368.00
Average turnaround time is 718.80
Number of context switches : 5
```

Improved Round Robin with varying time quantum algorithm (Paper 3) on Test Case 2



```
Processes 1 waiting time is 52 and turn around time is 155.
Processes 2 waiting time is 1245 and turn around time is 1695.
Processes 3 waiting time is 0 and turn around time is 71.
Processes 4 waiting time is 126 and turn around time is 602.
Processes 5 waiting time is 647 and turn around time is 1301.
The average waiting time: 414.00
The average turn around time: 764.80
Total number of context switch: 6
```

Efficient Dynamic Round Robin Algorithm (Paper 4) on Test Case 2



```
Process 1 waiting time is 52.00 and turnaround time is 155.00
Process 2 waiting time is 115.00 and turnaround time is 565.00
Process 3 waiting time is 0.00 and turnaround time is 71.00
Process 4 waiting time is 576.00 and turnaround time is 1052.00
Process 5 waiting time is 1097.00 and turnaround time is 1751.00

Average waiting time is 368.00
Average turnaround time is 718.80
Number of context switches : 4
```

Improved Round Robin CPU Scheduling Algorithm (Paper 5) on Test Case 2

5.2 Appendix B - ESRR (Paper 1)

The ESRR algorithm is made of RR and SRTF and it aims to reduce RR waiting and turnaround time and SRTF's starvation effect. The algorithm first operates in SRTF mode by taking the processes that have arrived in the queue and assigning them to the ready queue. The ready queue is sorted in ascending order according to the burst time of the process. The process with the smallest burst time that is at the head/top of the ready queue is assigned to the CPU. The dwell time on the CPU is determined by the time quantum. Each time a new process arrives, it is compared to the processes in the standby queue and added to the correct position in the standby queue. The standby queue is sorted in ascending order. The algorithm is non-preemptive in the sense that while a process still has time in its time quantum and a process with a shorter burst time arrives, the original process is still assigned to the CPU until it's time quantum is used up. Only then can the new, shorter process be assigned to CPU, as the ready queue is reordered according to the ascending burst time order. The algorithm runs in SRTF mode until the last process in the queue has arrived, and then switches to RR mode. It then switches to RR mode. It is then a normal RR, with the only exception that the queue is still sorted in ascending order, which increases efficiency.

The algorithm was implemented using a for loops to store the inputs. The largest burst time will be stored in int max, arrival time in int at, process burst time in int bu, number of processes is stored in int n. If the earliest arrival is not 0, int currenttime will be loaded to dismiss the error. the algorithm will run in SRTF configuration until the last process is in the waiting queue, then it switches to RR. ESRR will now utilise the RR aspect of the algorithm unless it is already sorted from smallest to biggest , an array named posarr is created, it is used to keep track of array position after sorting is performed. Int temp has the same usage as currenttime which is used to keep tabs on the time elapsed.

A macro for loop is made for int max, the macro for loop has a mini for loop which loops the processes and assigns it to the CPU in an equal time -share. The activity will carry on till all processes are completed.

When all processes are completed, the algorithm will switch out of the RR algorithm and print the tail-end of the algorithm code. It will print out the waiting time which is calculated by taking the total time subtract burst and arrival time. Turnaround time is calculated by using Total wait time divided by number of processes. Average Turnaround Time is calculated by taking the total turnaround time divided by the number of processes. Context Switch is calculated by taking all the cs++ at the end of every function and subtracting 1.

```c
#include<stdio.h>
void main()
{
    // i is loop counter
    int i, j, n, at[10], bu[10], wa[10], tat[10], t, ct[10], max, temp = 0,cs=0;
    float awt = 0.0, att = 0.0;

    // n is number of process
    printf("Enter the no of processes: ");
    scanf_s("%d", &n);
    for (i = 0; i < n; i++)
    {
        // at is arrival time
        printf("\nEnter Arrival Time for process %d: ", i + 1);
        scanf_s("%d", &at[i]);
        // bu is burst time
        printf("\nEnter Burst Time for process %d: ", i + 1);
        scanf_s("%d", &bu[i]);
        ct[i] = bu[i]; // ct is clone/ replica of process burst time

    }
    // t is time spent on CPU/ time quantum
    printf("\nEnter the size of time slice: ");
    scanf_s("%d", &t);
    max = bu[0];

    //find the process that has the longest burst time, to be used for RR for the
macro loop
    for (i = 1; i < n; i++) {
        if (max < bu[i])
            max = bu[i]; // max contains biggest burst time of all the processes
    }

    int posarr[10]; // process's postion, to help keep track during sorting base on
```

```
SRTF/SJF
    int SJF = 1; // SJF mode
    int index, burst, biggestarrival, tempos;
    int currenttime = 0; // current time and temp are same

    // to ensure both wait time and turn around time arrays are 0 and empty
    for (int i = 0; i < n; i++) {
        tat[i] = 0;
        wa[i] = 0;
    }

    for (int)

    currenttime = at[0]; // ensure that even when the first process arrival time is
not 0 program will still run

    while (SJF == 1) {
        index = 0;
        burst = 999;
        biggestarrival = 0;

        for (int i = 0; i < n; i++) {
            // find the process with the smallest burst time at arrival 0 or at the
earliest arrival time
            if (currenttime >= at[i] && burst >= bu[i] && bu[i] != 0) {
                index = i;
                burst = bu[i];
            }
            // find the process with the latest arrival time
            if (biggestarrival <= at[i]) {
                biggestarrival = at[i];
            }
        }
        //if theres no process running or arrive/ current time keep adding till a
process has come
        if (burst == 999) {
            currenttime += 1;
        }
        // if process burst time is more than quant do this (process allocated to
CPU)
        else if (burst > t) {
            currenttime += t; // add the quant to the current time, as time is
passing.
            tat[index] += t; // add the TUT to the process that is being allocated.
            bu[index] -= t; // decrement the burst time of the process that has
been allocated to CPU.
            cs++;
        }
        //if process burst time less than quant do this (process allocated to CPU)
        else if (burst <= t) {
            currenttime += burst; // add the remaining burst time of the process to
the current time, as time is passing.
            tat[index] = currenttime;  // add the TUT to the process that is being
allocated.
            //tat[index] += burst + wa[i];  //*additonal option not used*
            bu[index] = 0; // burst time of process become 0 as it is finished.
            cs++;
        }

        // once current time is over/ higher than the latest process arrival time
SJF mode is tuned off.
        if (currenttime > biggestarrival) {
            break;
        }
    }

    // used to keep track of the process position
    for (i = 0; i < n; i++) {
        posarr[i] = i;
    }
    int newlis[10];
    int tempa;
    // buffer/ replica of the remaining process's burst time to be used below to
help sort the process's position.
    for (i = 0; i < n; i++) {
        newlis[i] = bu[i];
    }

    // sorting of the process's remaining burst time from smallest to biggest.
    for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (newlis[i] > newlis[j])
            {
                tempa = newlis[i];
                newlis[i] = newlis[j];
                newlis[j] = tempa;
                tempos = posarr[i];
                posarr[i] = posarr[j];
                posarr[j] = tempos;
            }
        }
    }

    int flag = 0; // flag to change if RR mode is not run/not needed

    temp = currenttime; // temp is the current time after the SJF mode is over
```

```
        // perform RR
    for (j = 0; j < (max / t) + 1; j++) { // macro loop
        for (i = 0; i < n; i++) // micro loop for per process
        {
            //if process still has remainng burst time continue loop
            if (bu[posarr[i]] != 0)
            {
                // if process burst time is lesser or equal to CPU time slice
                if (bu[posarr[i]] <= t) {
                    // tat is turn around time for each process. (temp is wait
time, bu is burst time)
                    //tat[posarr[i]] += temp + bu[posarr[i]];
                    temp = temp + bu[posarr[i]];
                    tat[posarr[i]] = temp;
                    // process is completed
                    bu[posarr[i]] = 0;
                    flag = 1; // if set to 1 RR mode is indeed run/ needed.
                    cs++;
                }
                else
                    // if process burst time is more than CPU time slice
                {
                    // process is allocated to CPU. burst time is subtracted
                    bu[posarr[i]] = bu[posarr[i]] - t;
                    // temp is total time past
                    temp = temp + t;
                    flag = 1; // if set to 1 RR mode is indeed run/ needed.
                    cs++;
                }
            }
        }
    }



    /* print all the process waiting and turn around time and
    Calculate and print the 'average waiting time' and the
    'average turn-around-time'. */
    for (int i = 0; i < n; i++) {

        printf("---process %d Waiting time is %d ---\n", i + 1, tat[i] - ct[i] -
at[i]);
        printf("---process %d Turnaround time is %d ---\n", i + 1, tat[i] - at[i]);
        awt += tat[i] - ct[i] - at[i];
        att += tat[i] - at[i];

    }
    printf("\nAverage process waiting time is: %.2f\n", awt / n);
    printf("Average process turn around time is: %.2f\n", att / n);
    printf("Total number of context switches: %d\n", cs-1);


}
```

5.3 Appendix C - K-Factor (Paper 2)

This algorithm was implemented using a struct to store the details of each process like the burst time, process id, waiting time, arrival time and turnaround time. The processes will first be arranged according to their burst time in ascending order. A simple menu is also created to select the different burst time for processes.

```
#include <stdio.h>
#include <stdlib.h>


#define SIZE 100000 //Define the size for an array of structures
typedef struct pcb{ //Create a Process_Control_Block Structure to store each
process details
    int burst_time, id, waiting_time, turnaround_time, arrival_time;
}PCB;

//Common_Functions
int sort_burst(const void * a, const void * b); //To sort processes by burst time
void cal_average(float avgWT, float avgTT,int num_process,char *scheduler_name);//
To compute average waiting time and turn around time for processes

//Main_function
void main(){
  /*
   * VARIABLES declaration:
   * PCB is an array of structure with defined size 100000
   * fixed_sample is an array to store fixed burst time for testing
   * tWT - totalWaitingTime
   * tTT - totalTurnAroundTime
   * num_process - number of processes
   * input - to get user input for fixed burst time for processes or random burst
time
   * i - int to iterate through arrays
   * k - k factor for algorithm
   * time_spend - time spend for CPU working
   */
  PCB process[SIZE];
  int fixed_sample[10] = {42,68,35,1,70,25,79,59,63,65};
  int compare_sample[5] = {19,29,12,20,2};
```

```c
  int num_process,input,i,k=2,time_spend=0;
  float tWT, tTT;

  printf("...................................SJF - Simulation 
Starting...................................\n");
  printf("Enter the number of processes to compare:");
  scanf("%d", &num_process);
  printf("1: Assign Burst Time(Fixed number of 10 processes) \n2: Random Burst Time 
\n3: Compare Burst Time(Fixed number of 5 processes) \n");
  printf("Enter number 1 or 2 or 3 for the burst time to test:");
  scanf("%d", &input);

  //Store number of processes into the structure
  for(i =0;i<num_process; i++){
      /*
       * Assign the processes the id & their respective arrival time
       * The first process will have an id of 0 + 1 = 1
       */
      process[i].id = i+1;
      process[i].arrival_time = 0;
  }

  //To select between random generated burst time or fixed/assign burst time
  switch(input){
        //Case 1 : Assign fixed burst time for number of processes for testing
        case 1:
            // Fixed sample contains burst time|42 68 35 1 70 25 79 59 63 65
            for(i = 0; i < num_process; ++i){
                process[i].burst_time = fixed_sample[i];
            }
            break;
        //Case 2: Assign random burst time for n number of processes
        case 2:
            for(i = 0;i<num_process; i++){
                /*
                *   Setting burst time of each process randomly from 1 to 100
                *   rand() % 100 returns 0 to 100, + 1 makes it 1 to 100
                */
                process[i].burst_time = (rand() % 100) + 1;
            }
            break;
        case 3:
            // Compare sample contains burst time|19,29,12,20,2
            for(i = 0; i < num_process; ++i){
                process[i].burst_time = compare_sample[i];
            }
            break;
        default:
            break;
```

```
     }
   /*
    *qsort: C library function to sort the processes in ascending order of their
burst_time in the ready queue
    *Parameters:
    *    process = pointer to the first element of the array to be sorted
    *    num_proces = number of elements in the array
    *    sizeof(struct pcb) = sizes in bytes of each element in the structure
    *    sort_burst = the function that compares the two elements
   */
   qsort(process,num_process,sizeof(struct pcb), sort_burst);

   printf("\tProcess Running Order:\n");
   //Run Normal SJF
   for (i = 0; i < num_process; i++) {
       /*
        * This computation is executed for every process
        * 1) Process waiting time starts at 0
        * 2) Process turn around time = waiting_time + burst_time
        * 3) Process total turn around time is incremented by each process's turn
around time
        * 4) Process total waiting time is incremented by each process's waiting
time
        * 5) The waiting time for the next process is equivalent to the
incrementation of the burst time of the previous process
       */
       process[i].waiting_time = time_spend;
       process[i].turnaround_time = process[i].waiting_time  +
process[i].burst_time;
       tTT+=process[i].turnaround_time;
       tWT+=process[i].waiting_time;
       time_spend+=process[i].burst_time;
       printf("\n\tProcess_id:%d", process[i].id);
    }

    //Function to compute average and output results
    cal_average(tWT,tTT,num_process,"\nSJF - Normal Version");
    printf("..................................SJF - Simulation
Ending..................................\n");

    //Run SJF with K-Factor based on user input, if 1 run SJF with factor else run
SJF with k-factor increased alternatively
    printf("\n..................................SJF - K Factor Simulation
Starting..................................\n");
    printf("1: SJF with K-Factor\n2: SJF with K-Factor increased alternatively\n");
    printf("Enter number 1 or 2 for the k-factor to test:");
    scanf("%d", &input);
    printf("\tProcess Running Order:\n");
    tWT=0,tTT=0,time_spend = 0;//Reset variables to start a new computation
    for (i = 0; i < num_process; i++) {
        //Allocate the CPU to the first process and compute and ensure last process
is computed as well
        if(i == 0 || i+1 >= num_process){
             /*
              * When i = 0, compute the first process
              * When i + 1 >= num_process, compute the last process
              * For example: in array of size 10[0,1,2,3,4,5,6,7,8,9]
              * First process is executed which is at index 0, the last process is
index 9
              * In the k-factor algorithm, the comparison is made between the next
two process in the ready queue
              * If 0 index process run first, 1 & 2 wil be compare follow by 2 & 3
and so on.
              * The 9th index doesn't have a next process to compare with.
              * To ensure the last process is executed, we have to check if i+1 >=
number of processes and compute the last process accordingly.
             */
             process[i].waiting_time= time_spend;
             process[i].turnaround_time = time_spend + process[i].burst_time;
             tWT+=process[i].waiting_time;
             tTT+=process[i].turnaround_time;
             time_spend+=process[i].burst_time;

        }
        else{
              // Compare the first two processes in the ready queue
             if ( (k * process[i].burst_time) > (time_spend +
process[i+1].burst_time) ){
                 /*
                 * If current process burst time is greater than next process burst
time + the time_spend, next process will get the CPU
                 * For example: in array of size 10[0,1,2,3,4,5,6,7,8,9]
                 * Where index i = 1(Process 6) has a burst time of (25*k=2)=50 and
index i + 1(Process 3) has a burst time of 35)
                 * The CPU has only run the first process at index 0 with a burst
time of 1, so the time_spend = 1
                 * if(50) > (1 + 35) == True then Process 3 will get the CPU which
is at index i + 1
                 * if(50) > (1 + 35) == False then process 6 will get the CPU which
is at index i
                 */
                 //Process at index i + 1 gets the CPU and compute accordingly
                 process[i+1].waiting_time = time_spend;
                 process[i+1].turnaround_time =time_spend+ process[i+1].burst_time;
                 tWT+=process[i+1].waiting_time;
                 tTT+=process[i+1].turnaround_time;
                 time_spend+=process[i+1].burst_time;
```

```
                /*
                 * Store the index i + 1 burst time and id to temp variable
                 * Replace i + 1 burst time and id to index i burst time and id
                 * Store index i burst time and id as temp
                 * This is to tell the program the current process burst time and
id is i + 1 instead of i
                 * For example: process 6 which is at index i with burst time of
25,process 3 is at index i+1 with burst time of 35
                 * The process that was not selected to run is actually at index i,
by storing the burst time of index i+1 to temp = 35
                 * The program knows i'm currently running proccess at index i with
a burst time of 35, and my next i+1 process has a burst time of 25
                */
                int temp = process[i+1].burst_time, id = process[i+1].id;
                process[i+1].burst_time= process[i].burst_time;
                process[i+1].id = process[i].id;
                process[i].burst_time = temp;
                process[i].id = id;
                }
            else{
                //Process at index i gets the CPU and compute accordingly
                process[i].waiting_time = time_spend;
                process[i].turnaround_time = time_spend + process[i].burst_time;
                tWT+=process[i].waiting_time;
                tTT+=process[i].turnaround_time;
                time_spend+=process[i].burst_time;
            }

        }

        //If user select 2 then increase k-factor alternatively else compute k-
factor only
        if(input ==2){
            //Increase K value alternatively.
            if((i+2) % 2 == 1)
            {
                /* When does K increment:
                When I=1 | K-Number: 3
                When I=3 | K-Number: 4
                When I=5 | K-Number: 5
                When I=7 | K-Number: 6
                When I=9 | K-Number: 7
                */
                k+=1;
            }
        }
        //Debug mode
        /*
```

```
          printf("\tI-Number: %d \tK-Number: %d\n",i,k);
          printf("\tProcess Number    \tBurst Time  \tWaiting Time \tTurnaround
Time \tTime Spend \n");
          printf("\tP%d   \t\t\t%d \t\t%d \t\t%d \t\t\t%d \n",
process[i].id ,process[i].burst_time,process[i].waiting_time,process[i].turnaround_
time,time_spend);
        */
        printf("\n\tProcess_id:%d", process[i].id);
   }
   //If user select 2 then compute k-factor increase alternatively else compute k-
factor only
   if(input == 2){
       cal_average(tWT,tTT,num_process,"\nSJF - K Factor increased alternatively");
   }
   else{
       cal_average(tWT,tTT,num_process,"\nSJF - K Factor");
   }
   printf("\n..................................SJF - K Factor Simulation
Ending...................................\n");
}


// Sort processes in ascending order of their burst time
int sort_burst(const void * a, const void * b){
    /*
     * This function is called in qsort() and it takes in const void * because
qsort() doesn't know the actual types being sorted
     * By finding the difference between the values a and b we can sort the
structure PCB in ascending order by burst time
     * If return positive, then the value of a is greater than value b
     * If return negative, then value of b is greater than value a
     * If return 0, then a is equal to b
    */
    return (((PCB*)a)->burst_time - ((PCB*)b)->burst_time);
}

// Calculate average waiting time and turn around time
void cal_average(float tWT, float tTT,int num_process,char *scheduler_name){
   /*
   * avgWT - averageWaitingTime/number of processes in array
   * avgTT - averageTurnAroundTime/number of proccesses in array
   */
   float avgWT,avgTT;
   avgWT = tWT/num_process;
   avgTT = tTT/num_process;
   printf("\n%s",scheduler_name);
   printf("\nTotal Waiting Time: %.2f",tWT);
   printf("\nTotal TurnAround Time: %.2f",tTT);
   printf("\nAverage Waiting Time: %.2f",avgWT);
```

```
    printf("\nAverage Turn Around Time: %.2f\n",avgTT);
}
```

5.4 Appendix D - IRRVQ (Paper 3)

This algorithm is an Improved Round Robin with Varying Time Quantum (IRRVQ) and how tasks are completed will go in the following order. The algorithm will first rearrange tasks in the ready queue according to their burst time in ascending order. It will then calculate the average burst time and divide the tasks into 2 sub queues called the Light Task Queue (LTQ) and Heavy Task Queue (HTQ). Tasks that have a smaller burst time compared to the average burst time will get pushed to the LTQ and tasks with burst time equal or more than the average burst time will get pushed into the HTQ. The tasks will then be executed in the order of the LTQ first then HTQ.

When executing tasks in the LTQ and HTQ, the average burst time for each of the respective queues is calculated and is set as the time quantum for that queue respectively.

The tasks are then executed with the Time Quantum set and with every cycle of Round Robin executed, the average burst time will be recalculated with the remaining burst time values and the time quantum being updated respectively. This process gets repeated until all processes in the queue is completely executed.

```
#include <stdio.h>
#include <stdbool.h>

/* int n is the number of processes. int i and j are used as indices in for loops.
ProcessData[x][3] is a 2D array that stores info like this
[[BurstTime, ProcessNo#, Arrival Time, isDone(bool)],[BurstTime, ProcessNo#, Arrival Time,
isDone(bool)],...]
staticBurst[] stores the Burst time for each process as is with no change to the data.
ArrTime[] Stores the arrival time for each process. */
int i, j, n, totalBurst, ProcessData[10][4], staticBurst[10], ArrTime[10];
// Counter variables.
int countTrack = 0;
int LTQTrack = 0;
int HTQTrack = 0;
int contextSwitch = 0;
/* Light Task Queue(LTQ), Heavy Task Queue(HTQ), Medium Burst Time(MBT)
tempMove[] helps to store data when rearranging data around.
```

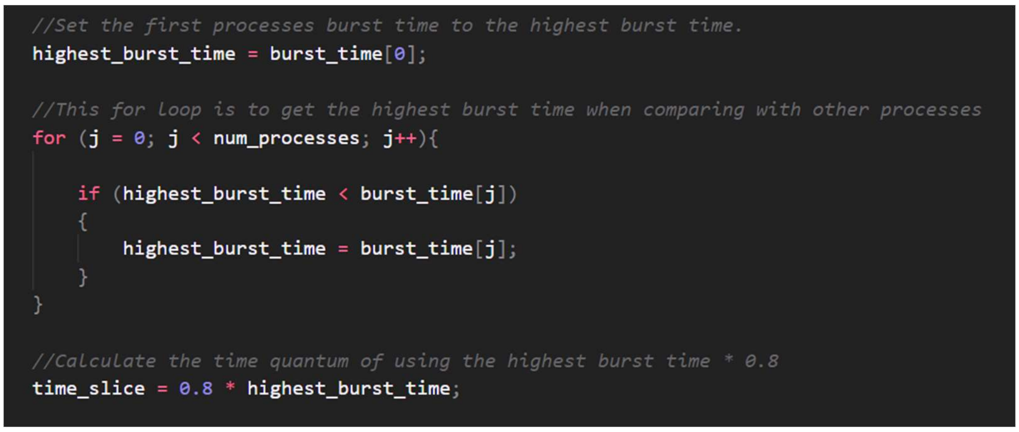
```c
waitingTime[] stores the waiting time of each process. */
float awt = 0, att = 0, MBT = 0, t = 0, max = 0, temp = 0, turnaroundTime[10][2],
LTQ[10][3], HTQ[10][3], tempMove[2], waitingTime[10];
// Variables to store the total turnaround time and the total waiting time.
float ttat = 0;
float twt = 0;
/* MainReadyQueue[][] will contain any process that is in the ready queue at a point in
time.
The format of MainReadyQueue[][] is as follows [[BurstTime, ProcessNo#, ArrTime,
isDone(bool)], [BurstTime, ProcessNo#, ArrTime, isDone(bool)]] */
float MainReadyQueue[10][4];

// Boolean variable to know if all processes arrive at the same time or have varying arrival
time.
bool AllArrivalSame = true;

// Function Prototypes
void processReadyQueue();
void sortTurnAroundTime();
void proposedRoundRobin(int m, int startOffset);

void main()
{
   // Character to store the yes or no input from user later on.
   char arr = '?';

   //Get the number of processes from user.
   printf("Enter the no of processes: ");
   scanf("%d", &n);

   while (arr != 'Y' && arr != 'y' && arr != 'N' && arr != 'n') {
       // Get input from user to know if all processes have the same arrival time or not.
       printf("\nDo all processes have the same arrival time of 0? Reply with y/n: ");
       scanf(" %c", &arr);
   }

   if (arr == 'Y' || arr == 'y') {
       //User indicated that all processes will arrive at t = 0
       AllArrivalSame = true;
   } else if (arr == 'N' || arr == 'n') {
       //User did not indicate that all processes will arrive at t = 0
       AllArrivalSame = false;
   }

   if (AllArrivalSame) {
       /* For the scenario where all processes arrive at time = 0 */
       for (i = 0; i < n; i++)
       {
           //Get the burst time for each process
           printf("\nEnter Burst Time for process %d: ", i + 1);
           scanf("%d", &ProcessData[i][0]);
           ProcessData[i][1] = i + 1;
           ProcessData[i][2] = 0;
           ArrTime[i] = 0;
           staticBurst[i] = ProcessData[i][0];
```

```
        }

        //Populate the ready queue
        for (int g = 0; g < n; g++) {
            MainReadyQueue[g][0] = ProcessData[g][0];
            MainReadyQueue[g][1] = ProcessData[g][1];
            MainReadyQueue[g][2] = ProcessData[g][2];
        }

        printf("Gantt Chart:\n");

        /* Run the proposed RR algorithm, pass in n as number of processes in ready queue, 0
for the startOffset
        as there was no process that was computed before this cycle. */
        proposedRoundRobin(n, 0);

        //Sorting the turnaroundTime array so that we can print the results in running order
based on Process No#.
        sortTurnAroundTime();

    } else {
        //The processes have varying arrival time.
        for (i = 0; i < n; i++)
        {
            //Get the burst time for each process
            printf("\nEnter Burst Time for process %d: ", i + 1);
            scanf("%d", &ProcessData[i][0]);
            ProcessData[i][1] = i + 1;
            printf("\nEnter Arrival Time for process %d: ", i + 1);
            scanf("%d", &ProcessData[i][2]);
            ArrTime[i] = ProcessData[i][2];
            ProcessData[i][3] = 0;
            staticBurst[i] = ProcessData[i][0];
        }

        printf("Gantt Chart:\n");

        //Start checking the Ready Queue and processing the tasks
        processReadyQueue();

        //Sorting the turnaroundTime array so that we can print the results in running order
based on Process No#.
        sortTurnAroundTime();
    }

    // Print new lines as gantt chart is done printing.
    printf("\n\n");

    // Calculate and print the waiting time and the turnaround time for each process.
    temp = 0;
    for (i = 0; i < n; i++)
    {
        //temp will store the cumulative waiting time which is = turnaround time - burst
time.
        temp = turnaroundTime[i][0] - (float)staticBurst[i];
        waitingTime[i] = temp;
        printf("Process %d waiting time is %.2f and turnaround time is %.2f\n", (i + 1),
temp, turnaroundTime[i][0]);
    }

    // Calculate and print the average waiting time and the average turnaround time.
    for (i = 0; i < n; i++)
    {
        twt += waitingTime[i];
        ttat += turnaroundTime[i][0];
    }

    //Calculating the average waiting time and the average turnaround time.
    awt = twt / (float)n;
    att = ttat / (float)n;

    printf("Average waiting time is %.2f\n", awt);
    printf("Average turnaround time is %.2f\n", att);

    printf("Number of context switches : %d\n", contextSwitch - 1);
}

void processReadyQueue() {
    //m to keep track of the number of tasks in the ready queue at this point of time.
    int m = 0;
    //startOffset is the number of processes that have been processed earlier before this
cycle started.
    int startOffset = countTrack;

    //Populate the ready queue
    for (int g = 0; g < n; g++) {
        if ((ProcessData[g][2] <= temp) && (ProcessData[g][3] == 0)) {
            MainReadyQueue[m][0] = ProcessData[g][0];
            MainReadyQueue[m][1] = ProcessData[g][1];
            MainReadyQueue[m][2] = ProcessData[g][2];
            MainReadyQueue[m][3] = ProcessData[g][3];
            m++;
            countTrack++;
        }
    }

    if (m > 1) {
        //Multiple processes in ready queue

        //Run the proposed RR Algorithm
        proposedRoundRobin(m, startOffset);

        if (countTrack != n) {
            //Not all processes have been done. Using recursion to check what tasks should be
in the ready queue.
            processReadyQueue();
        }
    } else {
        //Only one process in ready queue
        t = MainReadyQueue[0][0];
        //Temp will track the cumulative time so that the waiting time for subsequent
processes is properly compounded.
        temp = temp + MainReadyQueue[0][0];
        //turnaroundTime[x][0] is the turnaround time (waiting time + burst time) for each
process. turnaroundTime[x][1] is the process no. #
        turnaroundTime[countTrack - 1][0] = temp - MainReadyQueue[0][2];
        turnaroundTime[countTrack - 1][1] = MainReadyQueue[0][1];
        // Set the isDone flag for the process to true to mark it as done.
        ProcessData[(int)MainReadyQueue[0][1] - 1][3] = 1;
        //For printing out the gantt chart
        printf("| P%d (%.2f) ", (int)MainReadyQueue[0][1], temp);
        //Increment Context Switch count by 1
        contextSwitch++;

        if (countTrack != n) {
            //Not all processes have been done. Using recursion to check what tasks should be
in the ready queue.
            processReadyQueue();
        }
    }
}

void sortTurnAroundTime() {
    /* This function will sort the global array storing the turn around time for each
process, so that printing out results
     * will be in running order of Process No#. */
    for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (turnaroundTime[i][1] > turnaroundTime[j][1])
            {
                tempMove[0] = turnaroundTime[i][0];
                tempMove[1] = turnaroundTime[i][1];

                turnaroundTime[i][0] = turnaroundTime[j][0];
                turnaroundTime[i][1] = turnaroundTime[j][1];

                turnaroundTime[j][0] = tempMove[0];
                turnaroundTime[j][1] = tempMove[1];
            }
        }
    }
}

void proposedRoundRobin(int m, int startOffset) {
    /* This function contains the logic for the proposed Round Robin algorithm where
processes are sorted in ascending order
     * based on their burst time and then divided into LTQ (Light task queue) and HTQ (Heavy
task queue) based on the
     * MBT (Medium burst time) of all processes and then cleared in the following order LTQ
then HTQ. Burst time for each
     * queue is also based on the MBT of the queue. */

    //Sort the ready queue in ascending order based on burst time of processes.
    for (i = 0; i < m; i++)
    {
        for (j = i + 1; j < m; j++)
        {
            if (MainReadyQueue[i][0] > MainReadyQueue[j][0])
            {
                tempMove[0] = MainReadyQueue[i][0];
                tempMove[1] = MainReadyQueue[i][1];

                MainReadyQueue[i][0] = MainReadyQueue[j][0];
                MainReadyQueue[i][1] = MainReadyQueue[j][1];

                MainReadyQueue[j][0] = tempMove[0];
                MainReadyQueue[j][1] = tempMove[1];
            }
        }
    }

    //Find the MBT
    totalBurst = 0;
    for (i = 0; i < m; i++) {
        totalBurst += MainReadyQueue[i][0];
    }
    MBT = (float)totalBurst / (float)m;

    //Divide the tasks into LTQ and HTQ based on MBT
    LTQTrack = 0;
    HTQTrack = 0;
    for (i = 0; i < m; i++) {
        // If Burst Time is less than Medium Burst Time
        if ((float)MainReadyQueue[i][0] < MBT) {
            //Put in LTQ
            LTQ[LTQTrack][0] = (float)MainReadyQueue[i][0];
            LTQ[LTQTrack][1] = (float)MainReadyQueue[i][1];
            LTQTrack++;
        } else {
            //Put in HTQ
            HTQ[HTQTrack][0] = (float)MainReadyQueue[i][0];
            HTQ[HTQTrack][1] = (float)MainReadyQueue[i][1];
            HTQTrack++;
        }
    }

    //Time to clear the LTQ
    //Get the Time quantum for LTQ
    totalBurst = 0;
    for (int k = 0; k < LTQTrack; k++) {
        totalBurst += LTQ[k][0];
    }
    //The time quantum for LTQ (Light Task Queue) which is equals to the MBT (Medium Burst
 Time) of LTQ.
    t = (float)totalBurst / (float)LTQTrack;

    //Know the max burst time in this queue (LTQ Light Task Queue).
    max = LTQ[LTQTrack - 1][0];
```

```c
    //This is the juicy part, The heart and soul of RR. We will clear LTQ first according to
the Research Paper.
    //for loop where it loops for the maximum no. of times a process will ever need to queue
again.
    for (j = 0; j < (max / t) + 1; j++)
    {
        //for loop for each process in LTQ
        for (i = 0; i < LTQTrack; i++)
        {
            //if the process burst time is not 0,
            if (LTQ[i][0] != 0)
            {
                //if the burst time is less than or equals to time quantum
                if (LTQ[i][0] <= t)
                {
                    //Temp will track the cumulative time so that the waiting time for
subsequent processes is properly compounded.
                    temp = temp + LTQ[i][0];
                    //turnaroundTime[][] is the turnaround time (time now - arrival time) for
each process.
                    turnaroundTime[startOffset + i][0] = temp - ArrTime[(int)LTQ[i][1] - 1];
                    turnaroundTime[startOffset + i][1] = LTQ[i][1];
                    //Because the remaining burst time is =< 0, after running the process,
the remaining burst time must be 0, indicating
                    //that the process has completed.
                    LTQ[i][0] = 0;
                    // Set the isDone flag for the process to true to mark it as done.
                    ProcessData[(int)LTQ[i][1] - 1][3] = 1;
                    //For printing out the gantt chart
                    printf("| P%d (%.2f) ", (int)LTQ[i][1], temp);
                    //Increment Context Switch count by 1
                    contextSwitch++;
                }
                else
                {
                    //Comes here when the remaining burst time is more than time quantum, so
the remaining burst time is still going
                    //To be more than 0 after this cycle.
                    LTQ[i][0] = LTQ[i][0] - t;
                    temp = temp + t;
                    //For printing out the gantt chart
                    printf("| P%d (%.2f) ", (int)LTQ[i][1], temp);
                    //Increment Context Switch count by 1
                    contextSwitch++;
                }
            }
        }
        //Sort LTQ again based on ascending order of their remaining burst time.
        for (i = 0; i < LTQTrack; i++)
        {
            for (int a = i + 1; a < LTQTrack; a++)
            {
                if (LTQ[i][0] > LTQ[a][0])
                {
                    tempMove[0] = LTQ[i][0];
                    tempMove[1] = LTQ[i][1];

                    LTQ[i][0] = LTQ[a][0];
                    LTQ[i][1] = LTQ[a][1];

                    LTQ[a][0] = tempMove[0];
                    LTQ[a][1] = tempMove[1];
                }
            }
        }
    }

    //Now that LTQ is done, Time to process HTQ
    //Get the Time quantum for HTQ
    totalBurst = 0;
    for (int k = 0; k < HTQTrack; k++) {
        totalBurst += HTQ[k][0];
    }
    //The time quantum for HTQ (Heavy Task Queue) which is equals to the MBT (Medium Burst
Time) of HTQ.
    t = (float)totalBurst / (float)HTQTrack;

    //Know the max burst time in this queue (HTQ Heavy Task Queue).
    max = HTQ[HTQTrack - 1][0];

    //RR logic lai liao. Now clear HTQ with RR.
    for (j = 0; j < (max / t) + 1; j++)
    {
        //for loop for each process
        for (i = 0; i < HTQTrack; i++)
        {
            //if the process burst time is not 0,
            if (HTQ[i][0] != 0)
            {
                //if the burst time is less than or equals to time quantum
                if (HTQ[i][0] <= t)
                {
                    //Temp will track the cumulative time so that the waiting time for
subsequent processes is properly compounded.
                    temp = temp + HTQ[i][0];
                    //turnaroundTime[][] is the turnaround time (time now - arrival time) for
each process.
                    turnaroundTime[(startOffset + LTQTrack) + i][0] = temp -
ArrTime[(int)HTQ[i][1] - 1];
                    turnaroundTime[(startOffset + LTQTrack) + i][1] = HTQ[i][1];
                    //Because the remaining burst time is =< 0, after running the process,
the remaining burst time must be 0, indicating
                    //that the process has completed.
                    HTQ[i][0] = 0;
                    // Set the isDone flag for the process to true to mark it as done.
                    ProcessData[(int)HTQ[i][1] - 1][3] = 1;
                    //For printing out the gantt chart
                    printf("| P%d (%.2f) ", (int)HTQ[i][1], temp);
                    //Increment Context Switch count by 1
                    contextSwitch++;
                }
                else
                {
                    //Comes here when the remaining burst time is more than time quantum, so
the remaining burst time is still going
                    //To be more than 0 after this cycle.
                    HTQ[i][0] = HTQ[i][0] - t;
                    temp = temp + t;
                    //For printing out the gantt chart
                    printf("| P%d (%.2f) ", (int)HTQ[i][1], temp);
                    //Increment Context Switch count by 1
                    contextSwitch++;
                }
            }
        }
        //Sort HTQ again based on ascending order of their remaining burst time.
        for (i = 0; i < HTQTrack; i++)
        {
            for (int b = i + 1; b < HTQTrack; b++)
            {
                if (HTQ[i][0] > HTQ[b][0])
                {
                    tempMove[0] = HTQ[i][0];
                    tempMove[1] = HTQ[i][1];

                    HTQ[i][0] = HTQ[b][0];
                    HTQ[i][1] = HTQ[b][1];

                    HTQ[b][0] = tempMove[0];
                    HTQ[b][1] = tempMove[1];
                }
            }
        }
    }
}
```

5.5 Appendix E - EDRR (Paper 4)

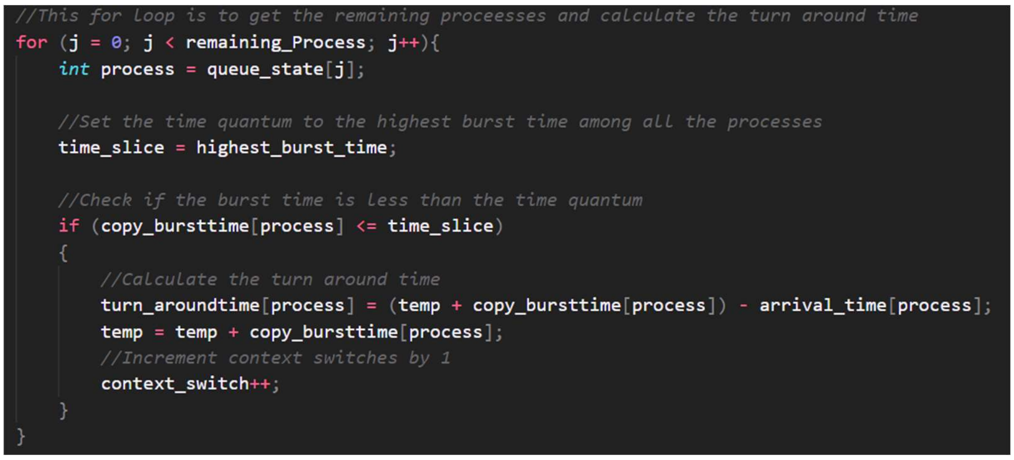
The time quantum for the proposed algorithm is calculated based on the 0.8th fraction of the maximum burst time from the processes in the ready queue (Refer to illustration 5.4.1).



```c
//Set the first processes burst time to the highest burst time.
highest_burst_time = burst_time[0];

//This for loop is to get the highest burst time when comparing with other processes
for (j = 0; j < num_processes; j++){

    if (highest_burst_time < burst_time[j])
    {
        highest_burst_time = burst_time[j];
    }
}

//Calculate the time quantum of using the highest burst time * 0.8
time_slice = 0.8 * highest_burst_time;
```

illustration 5.4.1

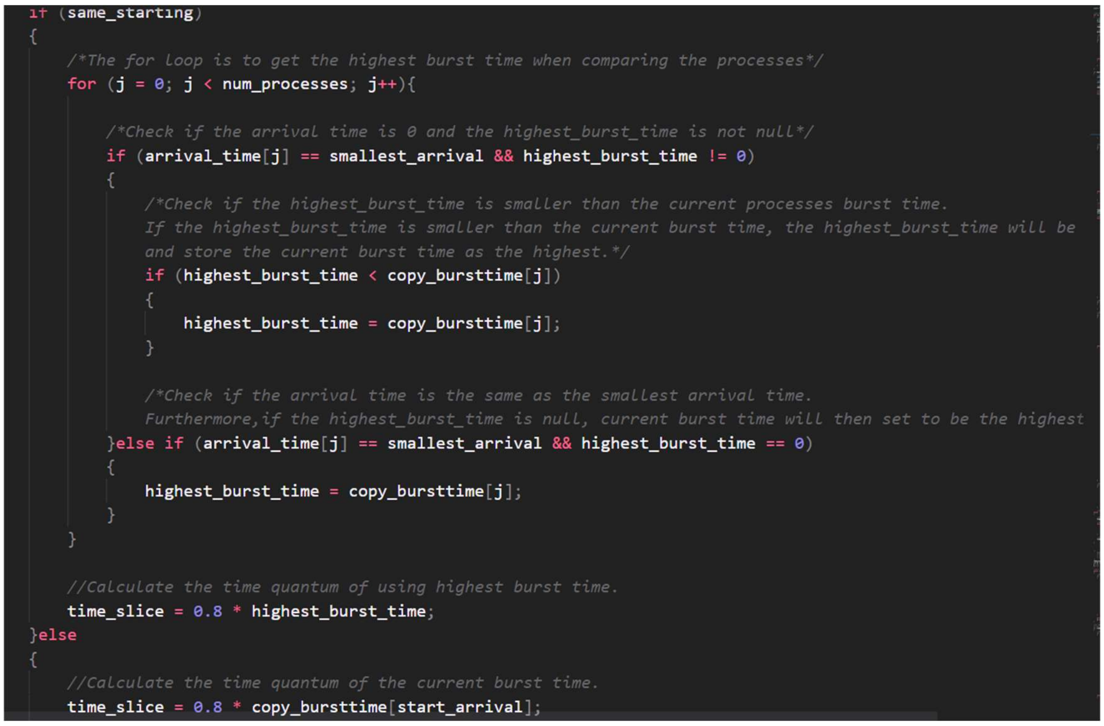
Those processes whose burst time is lower than the time quantum will be assigned to the CPU to execute. As soon as all the processes that were assigned to the CPU have finished executing, the time quantum is set equal to the maximum burst time and the remaining processes will then continue to be assigned to the CPU to execute (Refer to illustration 5.4.2).



```c
//This for loop is to get the remaining proceesses and calculate the turn around time
for (j = 0; j < remaining_Process; j++){
    int process = queue_state[j];

    //Set the time quantum to the highest burst time among all the processes
    time_slice = highest_burst_time;

    //Check if the burst time is less than the time quantum
    if (copy_bursttime[process] <= time_slice)
    {
        //Calculate the turn around time
        turn_aroundtime[process] = (temp + copy_bursttime[process]) - arrival_time[process];
        temp = temp + copy_bursttime[process];
        //Increment context switches by 1
        context_switch++;
    }
}
```

illustration 5.4.2

In the event that the processes have different arrival time and there is only one processes in the ready queue, the time quantum will be calculated based on the 0.8th fraction of the processes burst time. On the other hand, if there are two more processes which are in the ready queue, the time quantum will be calculated based on the 0.8 fraction of the maximum burst time (Refer to illustration 5.4.3).



```
if (same_starting)
{
    /*The for loop is to get the highest burst time when comparing the processes*/
    for (j = 0; j < num_processes; j++){

        /*Check if the arrival time is 0 and the highest_burst_time is not null*/
        if (arrival_time[j] == smallest_arrival && highest_burst_time != 0)
        {
            /*Check if the highest_burst_time is smaller than the current processes burst time.
            If the highest_burst_time is smaller than the current burst time, the highest_burst_time will be
            and store the current burst time as the highest.*/
            if (highest_burst_time < copy_bursttime[j])
            {
                highest_burst_time = copy_bursttime[j];
            }

            /*Check if the arrival time is the same as the smallest arrival time.
            Furthermore,if the highest_burst_time is null, current burst time will then set to be the highest
        }else if (arrival_time[j] == smallest_arrival && highest_burst_time == 0)
        {
            highest_burst_time = copy_bursttime[j];
        }
    }

    //Calculate the time quantum of using highest burst time.
    time_slice = 0.8 * highest_burst_time;
}else
{
    //Calculate the time quantum of the current burst time.
    time_slice = 0.8 * copy_bursttime[start_arrival];
```

illustration 5.4.3

The calculated time quantum will be used to compare the processes burst time in the queue and check if other processes have arrived in the ready queue. Once the first execution has been completed, the time quantum will be changed to 0.8th fraction of the maximum burst time. The processes which have already executed at the beginning and have remaining burst time will then continue to be assigned to the CPU followed by other processes whose burst time is lower than the time quantum. After the CPU execution, if there is only one processes remaining, the time quantum will be set equal to the burst time. On the condition that there are more than 2 processes remaining, the time quantum will be calculated based on 0.8th fraction of the maximum burst time. All these calculations and comparison of time quantum will keep repeating until all the processes have been executed by the CPU.

```
#include<stdio.h>
#include<string.h>

void main()
{
    //Declare and initialize variable
    /*
     i, j ,a = int for iterative loop
     ready = counter for ready queue
     notready = counter for non ready queue
     num_processes = number of processes
     arrival_time[10] = store arrival time of the processes
```

```
      burst_time[10] = store the burst time of the processes
      waiting_time[10] = store the waiting time of the processes
      turn-aroundtime[10] = store the turn around time of the processes
      time-slice = time quantum
      copy_bursttime[10] = a copy of the burst time for each processes
      highest_burst_time = store maximum burst time
      process_num[10] = store the processes number
      queue_state[10] =
      temp = counter for total time that the CPU has executed
      have_arrival = true or false for checking if there is arrival time
      smallest_arrival = store lowest arrival time
      remaining_Process = counter for remaining process queue
      all_ready = true or false for checking of all processes in the ready queue
      context_switch = counter for count context switch
      yes_no = store input from user to check for if there is arrival time
      */
    int i, j, a, ready, notready, num_processes, arrival_time[10],
burst_time[10], waiting_time[10], turn_aroundtime[10],time_slice,
copy_bursttime[10], highest_burst_time, process_num[10], queue_state[10], temp
= 0;
    int have_arrival = 0, smallest_arrival,remaining_Process, all_ready = 1;
    int context_switch = 0;
    char yes_no[20];

    //Get the input of the number of processes that the CPU needs to process
    printf("Enter the no of processes: ");
    scanf("%d", &num_processes);

    //Get a yes or no input if the arrival time is the same for each processes
    printf("Does all the processes have same arrival time (yes/no): ");
    scanf("%s", yes_no);


    //Get the input of burst time for each processes and the arrival time if
have for each processes
    for (j =0; j < num_processes; j++)
    {
        //Get the burst time for each processes
        printf("\nEnter Burst Time for process %d: ", j+1);
        scanf("%d", &burst_time[j]);

        //Make a copy of the burst time so that the original burst time will
not be amended during the process of calculation
        copy_bursttime[j] = burst_time[j];

        //Store the process number accordingly
        process_num[j] = j+1;
        /*Check the input is a yes or no
        if the input is a yes, the arrival time will be set default to 0
        if the input is a no, will get the arrival time each processes from the
user*/
        if (strcmp(yes_no, "no") == 0)
        {

            //Get the arrival time for each processes
            printf("\nEnter Arrival Time for process %d: ", j+1);
            scanf("%d", &arrival_time[j]);

            /*Set have_arrival variable to 1
            if 1 means there is an arrival time for each processes
            if 0 means there is no arrival time for each processes*/
            have_arrival = 1;
        }else
        {
            /*Set the arrival time for each processes 0*/
            arrival_time[j] = 0;
        }

    }

    if (have_arrival)
    {
        //This variable act as a counter for counting how many processes have a
same arrival time
        int same_starting = 0;

        //This variable act as
        int start_arrival = 0;

        //Set the first arrival time as the smallest arrival time
        smallest_arrival = arrival_time[0];

        /*The for loop is to get the smallest arrival time and also check which
processes arrive first and to be assigned to CPU first.
        Furthermore, the for loop also check how many processes have a same
arrival time.*/
        for (j = 0; j < num_processes; j++){

            //Get the smallest arrival time
            if (arrival_time[j] < smallest_arrival)
            {
                smallest_arrival = arrival_time[j];
            }
            //Compare the current processes arrival time with next processes
arrival time
            for (a = j+1; a < num_processes; a++){


                if (arrival_time[j] != arrival_time[a])
                {
                    //Initialize start_arrival value to the processes
                    if (smallest_arrival > arrival_time[a])
                    {
                        smallest_arrival = arrival_time[a];
                        start_arrival = a;
                    }
                }/*If the arrival time is the same, same_starting to 1.*/
                else if (arrival_time[j] == arrival_time[a])
                {
                    same_starting = 1;
                }
            }
        }



        /*If the same_starting is 1, it means there are processes have same
arrival time
        If the same_starting is 0, it means there are no processes that have
same arrival time*/
        if (same_starting)
        {

            /*The for loop is to get the highest burst time when comparing the
processes*/
            for (j = 0; j < num_processes; j++){

                /*Check if the arrival time is 0 and the highest_burst_time is
not null*/
                if (arrival_time[j] == smallest_arrival &&
highest_burst_time != 0)
                {
                    /*Check if the highest_burst_time is smaller than the
current processes burst time.
                    If the highest_burst_time is smaller than the current burst
time, the highest_burst_time will be override
                    and store the current burst time as the highest.*/
                    if (highest_burst_time < copy_bursttime[j])
                    {
                        highest_burst_time = copy_bursttime[j];
                    }

                    /*Check if the arrival time is the same as the smallest
arrival time.
                    Furthermore,if the highest_burst_time is null, current
burst time will then set to be the highest burst time.*/
                }else if (arrival_time[j] == smallest_arrival &&
highest_burst_time == 0)
                {
                    highest_burst_time = copy_bursttime[j];
                }
            }

            //Calculate the time quantum of using highest burst time.
            time_slice = 0.8 * highest_burst_time;
        }else
        {
            //Calculate the time quantum of the current burst time.
            time_slice = 0.8 * copy_bursttime[start_arrival];

        }

        /*ready = Number of processes is in the ready state after the first
processes has started running.
         * notready = Number of processes is not in the ready state after the
first processes has started running.
         * ready_queue = It is an array that store the processes that is in the
ready state.
         * notready_queue = It is an array that store the processes that is not
in the ready state.*/
        ready = 0;
        notready = 0;
        int ready_queue[10], notready_queue[10];

        /*This for loop is to check if all the processes have been arrived in
the ready state when the first processes has been assigned to CPU.*/
        for (j = 0; j < num_processes; j ++){


            /*If the arrival time is more than the time quantum,
             * Store the processes to the notready_queue array
             * Increment notready by 1.*/
            if (arrival_time[j] > time_slice)
            {
                notready_queue[notready] = j;
                notready++;
            }else
            {
                /*Store the processes to the ready_queue array.
                 * Increment ready by 1.*/
                ready_queue[ready] = j;
                ready++;
            }


            /*Check if the arrival time is the smallest arrival time and the
burst time is more than the time quantum.*/
            if (arrival_time[j] == smallest_arrival && copy_bursttime[j] >
time_slice)
            {
                /*Calculate the turn around time of the processes*/
                turn_aroundtime[j] = (temp + time_slice) - arrival_time[j];
                temp = temp + time_slice;

                //Minus the time quantum from the current burst time to get the
remaining burst time for the processes
                copy_bursttime[j] = copy_bursttime[j] - time_slice;
                //Increment context switches by 1
                context_switch++;



                //Check if the arrival time is the smallest arrival time and
the burst time is less than the time quantum
            }else if (arrival_time[j] == smallest_arrival)
            {
                //Calculate the turn around time of the processes
                turn_aroundtime[j] = (temp + copy_bursttime[j]) -
arrival_time[j];
                temp = temp + copy_bursttime[j];

                /*Set the current burst time to 0.
                It means that the CPU has finish running the processes
                set the start_arrival to 0, which the first processes has
finish running*/
                copy_bursttime[j] = 0;
                start_arrival = 0;
                //Increment context switches by 1
                context_switch++;
            }
        }
        /*If noready is more than 0, it means there are processes that is still
not in the ready state after the first processes has started running.*/
        if (notready > 0)
        {
            /*Set the all_ready to 0*/
            all_ready = 0;
        }

        /*If the all_ready is 1, it means that all the processes is in the
ready state after the first processes runs.
        If the all_ready is 0, it means that not all the processes is in the
ready state.*/
        if (all_ready)
        {
            //Set the first processes burst time to the highest burst time.
            highest_burst_time = copy_bursttime[0];

            //This for loop is to get the highest burst time when comparing
with other processes
            for (j = 0; j < num_processes; j++){

                if (highest_burst_time < copy_bursttime[j])
                {
                    highest_burst_time = copy_bursttime[j];
                }
            }

            //Calculate the time quantum of using the highest burst time * 0.8
            time_slice = 0.8 * highest_burst_time;

        }else if (ready > 0)
        {
            //Set the first processes burst time to the highest burst time.
            highest_burst_time = copy_bursttime[ready_queue[0]];

            //This for loop is to get the highest burst time when comparing
with other processes
            for (j = 0; j < ready; j++){
                int process = ready_queue[j];

                if (highest_burst_time < copy_bursttime[process])
                {
                    highest_burst_time = copy_bursttime[process];
                }
            }

            //Calculate the time quantum of using the highest burst time * 0.8
            time_slice = 0.8 * highest_burst_time;


            /*If the start_arrival is more than 0, it means the first processes 
that assigned to the CPU, have not finish running*/
            if (start_arrival > 0)
            {
                //Calculate the turn around time of the processes
                turn_aroundtime[start_arrival] = (temp + 
copy_bursttime[start_arrival]) - arrival_time[start_arrival];
                temp = temp + copy_bursttime[start_arrival];

                /*Set the current burst time to 0.
                It means that the CPU has finish running the processes*/
                copy_bursttime[start_arrival] = 0;
                start_arrival = 0;
                //Increment context switches by 1
                context_switch++;


            }

            for (j = 0; j < ready; j++){
                int index = ready_queue[j];
                /*Check if the arrival time is the smallest arrival time and 
the burst time is more than the time quantum.*/
                if (copy_bursttime[index] > time_slice)
                {
                    /*Calculate the turn around time of the processes*/
                    turn_aroundtime[index] = (temp + time_slice) - 
arrival_time[index];
                    temp = temp + time_slice;

                    //Minus the time quantum from the current burst time to get 
the remaining burst time for the processes
                    copy_bursttime[index] = copy_bursttime[index] - time_slice;
                    //Increment context switches by 1
                    context_switch++;
                    /* Store the processes to the notready_queue array
                    * Increment notready by 1.*/
                    notready_queue[notready] = index;
                    notready++;

                }else if (copy_bursttime[index] > 0)
                {
                    //Calculate the turn around time of the processes
                    turn_aroundtime[index] = (temp + copy_bursttime[index]) - 
```

```
arrival_time[index];
                    temp = temp + copy_bursttime[index];

                    /*Set the current burst time to 0.
                    It means that the CPU has finish running the processes*/
                    copy_bursttime[index] = 0;
                    context_switch++;
                }
            }



            /*Check if the processes in the notready_queue array is in the
ready state*/
            for(j = 0; j < notready; j++){
                int index = notready_queue[j];

                /*If the arrival time is less than the time taken that the
processes has already ran, it means that the processes is in the ready state
and set all_ready to 1.
                 * If the arrival time is more than the time taken that the
processes has already ran, it means that the processes is not in the ready
state and set all_ready to 0. */
                if (arrival_time[index] < temp)
                {
                    all_ready = 1;
                } else
                {
                    all_ready = 0;
                }
            }

            /*If the all_ready is 1, it means that all the processes is in the
ready state after the first processes runs.
            If the all_ready is 0, it means that not all the processes is in
the ready state.*/
            if (all_ready)
            {
                highest_burst_time = copy_bursttime[notready_queue[0]];

                //This for loop is to get the highest burst time when comparing
with other processes
                for (j = 0; j < notready; j++){
                    int index = notready_queue[j];
                    if (highest_burst_time < copy_bursttime[index])
                    {
                        highest_burst_time = copy_bursttime[index];
                    }
                }

                //Calculate the time quantum of using the highest burst time *
0.8
                time_slice = 0.8 * highest_burst_time;
            }
        }

        /*If the start_arrival is more than 0, it means the first processes
that assigned to the CPU, have not finish running*/
        if (start_arrival > 0)
        {
            //Calculate the turn around time of the processes
            turn_aroundtime[start_arrival] = (temp +
copy_bursttime[start_arrival]) - arrival_time[start_arrival];
            temp = temp + copy_bursttime[start_arrival];

            /*Set the current burst time to 0.
            It means that the CPU has finish running the processes*/
            copy_bursttime[start_arrival] = 0;
            start_arrival = 0;
            //Increment context switches by 1
            context_switch++;

        }

        //If there is no arrival time or same arrival time for all the
processes
    }else
    {

        //Set the first processes burst time to the highest burst time.
        highest_burst_time = burst_time[0];

        //This for loop is to get the highest burst time when comparing with
other processes
        for (j = 0; j < num_processes; j++){

            if (highest_burst_time < burst_time[j])
            {
                highest_burst_time = burst_time[j];
            }
        }

        //Calculate the time quantum of using the highest burst time * 0.8
        time_slice = 0.8 * highest_burst_time;
    }




    /*
        Remaining_Process = remainig processes that haven't been assign to CPU
to run
        N = number of the processes
        j = just the integer that is used for looping
        i = the processes number
    */
    remaining_Process = 0;
    int N = num_processes;
    j = 0;
    i = 1;

    /*This is the algorithm that calculate the turn around time for each
processes*/
    while (i <= N)
    {
        //Check if i is less than the number of processes and the burst time is
not 0
        if (i <= N && copy_bursttime[j] != 0)
        {
            //Check if the burst time is equal or less than the time quantum
            if (copy_bursttime[j] <= time_slice && copy_bursttime[j] > 0)
            {

                //Calculate the turn around time for the processes
                turn_aroundtime[j] = (temp + copy_bursttime[j]) -
arrival_time[j];
                temp = temp + copy_bursttime[j];
                //Increment context switches by 1
                context_switch++;


                //Check if the burst time is more than the time quantum
            }else if (copy_bursttime[j] > time_slice)
            {
                //Set the processes to in the queue state
                queue_state[remaining_Process] = j;


                //Increment the remaining processes by 1
                remaining_Process++;
            }
        }

        i++;
        j++;
    }

    /*If the remaining process is more than 0, it means that there are
processes that have assign to CPU to run.
    If the remaining process is less than 0, it means that all the processes
has finish running.*/
    if (remaining_Process > 0)
    {
        //Check there is arrival time for each processes
        if (have_arrival)
        {
            /*Check if the remaining process is 1.
            If the remaining process is 1, it means that there is only 1
processes that haven't been assign to CPU to run.
            If the remaining process is more than 1, it means that there is
more than 1 processes that haven't been assign to CPU to run.*/
            if (remaining_Process == 1)
            {
                //This for loop is to get the remaining processes and calculate
the turn around time
                for (j = 0; j < remaining_Process; j++){
                    int process = queue_state[j];

                    //Set the time quantum to the burst time of the processes
                    time_slice = copy_bursttime[process];

                    //Calculate the turn around time for the processes
                    turn_aroundtime[process] = (temp + time_slice) -
arrival_time[process];
                    temp = temp + copy_bursttime[process];
                    //Increment context switches by 1
                    context_switch++;
                }
            }else
            {
                /*This for loop is to get the remaining processes and calculate
the turn around time*/
                for (j = 0; j < remaining_Process; j++){
                    int process = queue_state[j];

                    /*Set the time quantum to the burst time of the processes*/
                    time_slice = copy_bursttime[process];
                    /*Calculate the turn around time for the processes*/
                    turn_aroundtime[process] = (temp + time_slice) - 
arrival_time[process];
                    temp = temp + copy_bursttime[process];
                    //Increment context switches by 1
                    context_switch++;
                }
            }

        }else
        {
            //This for loop is to get the remaining proceesses and calculate 
the turn around time
            for (j = 0; j < remaining_Process; j++){
                int process = queue_state[j];

                //Set the time quantum to the highest burst time among all the 
processes
                time_slice = highest_burst_time;

                //Check if the burst time is less than the time quantum
                if (copy_bursttime[process] <= time_slice)
                {
                    //Calculate the turn around time
                    turn_aroundtime[process] = (temp + copy_bursttime[process]) 
- arrival_time[process];
                    temp = temp + copy_bursttime[process];
                    //Increment context switches by 1
                    context_switch++;
                }
            }
        }

    }

    //Calculate the waiting time for each processes
    for (j = 0; j < num_processes; j++)
    {

        waiting_time[j] = turn_aroundtime[j] - burst_time[j];

        //Print the waiting time and turn around time for each processes
        printf("Processes %d waiting time is %d and turn around time is %d.\n", 
process_num[j], waiting_time[j], turn_aroundtime[j]);
    }
    float total_WT, total_tat, average_WT, average_TAT;

    /*Calculate total waiting time and turn-around-time*/
    for (j =0; j < num_processes; j++)
    {
        //total waiting time
        total_WT = total_WT + waiting_time[j];

        //total turn around time
        total_tat = total_tat + turn_aroundtime[j];

    }

    //Calculate average waiting time and turn-around-time
    average_WT = total_WT/num_processes;
    average_TAT = total_tat/num_processes;

    //Print the average waiting time and turn-around-time
    printf("The average waiting time: %0.2f\n", average_WT);
    printf("The average turn around time: %0.2f\n", average_TAT);
    printf("Total number of context switch: %d\n", context_switch-1);

}
```

5.6 Appendix F - IRRPQ (Paper 5)

First I set the low burst time to 200 and saved it as variable. Then I set a 2D array which translates the table into 2D array. The code below will show how the array works. But a brief example based on Table 1 is this, MainTable[which process][which column], MainTable[1][1] will return Process 1’s Burst Time. MainTable[1][2] will return Process 1’s Arrival Time, etc.

The calculation of low, high priority is also saved in a variable so I won’t have to keep typing the formula out. All the needed variables are initialised on top such as the total turnaround time, arrival time, average time. The basis of all integers are changed to float, this is to have a more precise data than whole numbers.

Now, a ‘doneProcesses’ variable is to count when all processes are done running and then the while loop will end. First thing, I will try to find out any crucial processes that have burst time less than 200, those that can execute, I will execute (line 59).

I will then loop through the processes and see if there are processes currently at the moment have an arrival time less than or equal to the current time. Example, the very first iteration, process 1 and process 5 will be checked first since both of the processes arrival time are 0 which is equal or less than current time which is 0. Then FCFS will come in, since process 1 comes first before process 5, process 1 will run first.

I will repeat this for every iteration, and it will also check if any processes match those conditions given. If so, then execute them completely. Because of these conditions, context switches can be reduced and processes have a higher chance to execute at once rather than get affected by time quantum and have to wait for their turn again just to run a tiny bit of remaining burst time.

```
#include <stdio.h>

#define LOW_BT 200

// i used for loops, n is the number of processes. JQ[i] is the Job Queue Flag.
// MainTable is a 2D array, eg: MainTable[no.of process allow][which column, so 1
will call the burst time, 2 will call arrival time]
int i, n, doneProcesses, JQ[10], MainTable[10][4], contextSwitches;

/* Format is something like MainTable[[ProcessNo#, BurstTime, ArrTime, Priority],
[ProcessNo#, BurstTime, ArrTime, Priority]]
lowTQ, medTQ, highTQ is the TQ for the various priorities and the s suffix infront
is for the if statements */
float TQ = 0, lowTQ = 0, highTQ = 0, sLowTQ = 0, sMedTQ = 0, sHighTQ = 0,
effectiveTQ = 0, sEffectiveTQ = 0, remainingBurstTime[10][2], tat[10], wt[10];
// ttat = total turnaround time, twt = total waiting time, awt = avg Waiting time,
atat = avg turnaround time.
float ttat = 0;
float twt = 0;
float awt = 0;
float atat = 0;

//Time now is 0
float t = 0;

void main()
{
    doneProcesses = 0;

    printf("Enter the no of processes: ");
    scanf("%d", &n);
```

```c
    for (i = 0; i < n; i++)
    {
        // Fill in the Process No #.
        MainTable[i][0] = i + 1;
        // Fill in the Burst Time
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &MainTable[i][1]);
        remainingBurstTime[i][0] = i + 1;
        remainingBurstTime[i][1] = MainTable[i][1];
        // Fill in the Arr Time
        printf("Enter arrival time for process %d: ", i + 1);
        scanf("%d", &MainTable[i][2]);
        // Fill in the priority
        printf("Enter priority time for process %d: ", i + 1);
        scanf("%d", &MainTable[i][3]);
        printf("-----------------------------------------\n");
        JQ[i] = 0;
    }
    printf("Enter Time Quantum: ");
    scanf("%f", &TQ);
    printf("\n ");
    printf("Gantt Chart: \n ");

    lowTQ = (float)0.8 * TQ;
    highTQ = (float)1.2 * TQ;
    sLowTQ = (float)(lowTQ + (0.2 * lowTQ));
    sMedTQ = (float)(TQ + (0.2 * TQ));
    sHighTQ = (float)(highTQ + (0.3 * highTQ));

    //Check for any processes that have lowBT (<200) and finish it first.
    for (i = 0; i < n; i++)
    {
        if ((remainingBurstTime[i][1] < (float)LOW_BT) && (JQ[i] == 0) &&
((float)MainTable[i][2] <= t)) {
            /*Burst time is low (less than 200) and that the flag is 0 (Not done
processing yet)
              According to algo, we just finish it and set flag to true.*/
            t += remainingBurstTime[i][1];

            // since we end this process le, then we will set its remaining burst
time to 0
            remainingBurstTime[i][1] = 0;
            //Set the turnaround time (time now - arr time)
            tat[i] = t - (float)MainTable[i][2];
            //Set the waiting time (turnaround time - burst time)
            wt[i] = tat[i] - (float)MainTable[i][1];
            JQ[i] = 1;
            doneProcesses++;
            contextSwitches++;
            //For gantt chart
            printf("| %d (%.2f)", i + 1, t);
        }
    }

    /* Need to check the against TQ and burst time
    While not all processes are done... */
    while (doneProcesses != n)
    {
        for (i = 0; i < n; i++)
        {
            /*Check if process has arrived at the ready queue.
            if there is any process that have arrival time that equal or less than
the CURRENT time, then I will check those processes first */

            if ((float)MainTable[i][2] <= t) {
                /* We need to know what is its TQ first via the priority number
                   start measuring its priority and set it for the process
                   sEffective is just the 2nd given condition (see qns 8,9,10 of
Paper5) */
                if (MainTable[i][3] == 1) {
                    effectiveTQ = lowTQ;
                    sEffectiveTQ = sLowTQ;
                }
                else if (MainTable[i][3] == 2) {
                    effectiveTQ = TQ;
                    sEffectiveTQ = sMedTQ;
                }
                else if (MainTable[i][3] == 3) {
                    effectiveTQ = highTQ;
                    sEffectiveTQ = sHighTQ;
                }

                //Now is about checking how long we are gonna run the process
                if ((remainingBurstTime[i][1] <= effectiveTQ) && (JQ[i] == 0)) {
                    //According to algo, we just finish it and set flag to true.
                    t += remainingBurstTime[i][1];
                    remainingBurstTime[i][1] = 0;
                    //Set the turnaround time (time now - arr time)
                    tat[i] = t - (float)MainTable[i][2];
                    //Set the waiting time (turnaround time - burst time)
                    wt[i] = tat[i] - (float)MainTable[i][1];
                    JQ[i] = 1;
                    doneProcesses++;
                    contextSwitches++;
                    //For gantt chart
                    printf("|  %d (%.2f)", i + 1, t);
                }
                //Priority is medium or Low
                else if ((MainTable[i][3] == 1 || MainTable[i][3] == 2) && (JQ[i]
== 0)) {
                    if ((remainingBurstTime[i][1] > effectiveTQ) &&
(remainingBurstTime[i][1] <= sEffectiveTQ)) {
                        //According to algo, we just finish it and set flag to
true.
                        t += remainingBurstTime[i][1]; // take the burst time and
add to the time, [i] is which process u want, then [1] takes the burst time
                        remainingBurstTime[i][1] = 0;
                        //Set the turnaround time (time now - arr time)
                        tat[i] = t - (float)MainTable[i][2];
                        //Set the waiting time (turnaround time - burst time)
                        wt[i] = tat[i] - (float)MainTable[i][1];
                        JQ[i] = 1;
                        doneProcesses++;
                        contextSwitches++;
                        //For gantt chart
                        printf("|  %d (%.2f)", i + 1, t);
                    }
                    else {
                        //Just run for its assigned TQ..
                        t += effectiveTQ;
                        remainingBurstTime[i][1] = remainingBurstTime[i][1] -
effectiveTQ;
                        contextSwitches++;
                        //For gantt chart
                        printf("|  %d (%.2f)", i + 1, t);
                    }
                }
                //Priority is high
                else if ((MainTable[i][3] == 3) && (JQ[i] == 0)) {
                    if ((remainingBurstTime[i][1] > effectiveTQ) &&
(remainingBurstTime[i][1] <= sEffectiveTQ)) {
                        //According to algo, we just finish it and set flag to
true.
                        t += remainingBurstTime[i][1];
                        remainingBurstTime[i][1] = 0;
                        //Set the turnaround time (time now - arr time)
                        tat[i] = t - (float)MainTable[i][2];
                        //Set the waiting time (turnaround time - burst time)
                        wt[i] = tat[i] - (float)MainTable[i][1];
                        JQ[i] = 1;
                        doneProcesses++;
                        contextSwitches++;
                        printf("|  %d (%.2f)", i + 1, t);
                    }
                    else {
                        //Just run for its assigned TQ..
                        t += effectiveTQ;
                        remainingBurstTime[i][1] = remainingBurstTime[i][1] -
effectiveTQ;
                        contextSwitches++;
                        //For gantt chart
                        printf("|  %d (%.2f)", i + 1, t);
                    }
                }
            }
        }
    }

    //End of gantt chart line.
    printf("\n\n");

    // Loop through and print out information about each process.
    for (i = 0; i < n; i++)
    {
        printf("Process %d waiting time is %.2f and turnaround time is %.2f\n", (i
+ 1), wt[i], tat[i]);
    }

    printf("\n");

    // Calculate the total waiting time and total turnaround time.
    for (i = 0; i < n; i++)
    {
        twt += wt[i];
        ttat += tat[i];
    }

    // Calculate average waiting time
    awt = twt / (float)n;
    // Calculate average turnaround time
    atat = ttat / (float)n;

    printf("Average waiting time is %.2f\n", awt);
    printf("Average turnaround time is %.2f\n", atat);
    printf("Number of context switches : %d\n", contextSwitches - 1);
}
```