Operating System: Assignment 2

Mohammad Rehan Khan (2020MT10822)

Mohammad Areeb (2020MT10656)

Date: 05-09-2023

Discrete Time Event Scheduler

1. FCFS: First Come First Serve

1.1 Algorithm

```
• Process ← structure defining the process variables.
• Context ← structure containing the PID of running process, start time and time of
  next switch.
• Input: Array Processes of type 'struct Process', integer N (Number of Processes)

Steps:

• Initialize an array Contexts of type 'struct Context' of size N.
• Sort the Processes array by arrival time. If arrival time is same, sort by lexicographic
  ordering.
• Initialize current time ← Process[0].arrival_time
• For i=0; i< N; i++ : do
    o Create a context C.
        ▪ C.PID = PID of the current Process.
        ▪ C.start_time = current_time.
        ▪ C.end_time = C.start_time + burst time of the current Process
    o Process[i].response_time = current_time – Process[i].arrival_time
    o Process[i].turnaround_time = C.end_time – Process[i].arrival_time
    o Update current time to C.end_time.
• Output: Display the Blue print stored in the Contexts array average Response time
  and average Turnaround time.
```

1.2 Simulation

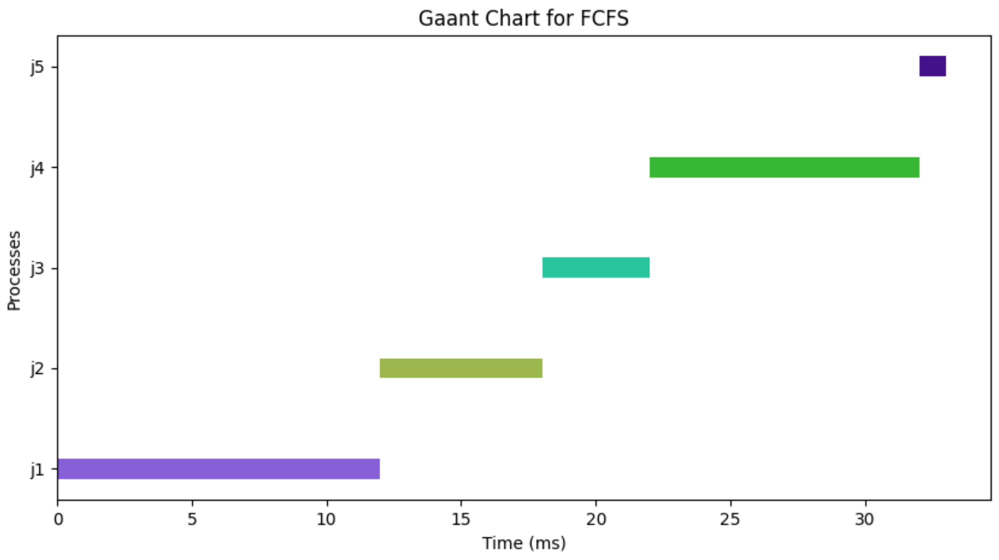
Simulation 1.2.1

Number of Process = 5

Interarrival time between process are sampled from exponential distribution with mean value 4. Burst time for each process is sampled randomly with an upper cap of 15 and lower cap of 1. Following table shows the details about the Process to be scheduled on the CPU by the OS.

PID	Arrival Time	Burst Time
J1	0	12
J2	3	6
J3	3	4
J4	3	10
J5	16	1

Gantt chart:



Blue Print: j1 0 12 j2 12 18 j3 18 22 j4 22 32 j5 32 33

Average Turnaround Time: 18.400

Average Response Time: 11.800

Analysis: FCFS is the one of the earliest developed algorithms, simplest and easiest to implement. The Job coming into the system are scheduled on the basis of their arrival times. It works on FIFO principle and is non-pre-emptive in nature.

However, being the simplest one, its not at all efficient. There are high chances of Convoy effect and starvation. A process, with significantly smaller burst time can have to wait for a long running job to complete its execution before being scheduled. This leads to bad waiting times. The Job j5 above arrived at time 16 and had a burst time of only 1 time units, but was scheduled only at time instant 32, leading to a waiting time of 16 time units, which is unacceptable for a job of burst time 1! Higher wait times further leads to bad performance in terms of average turnaround time.

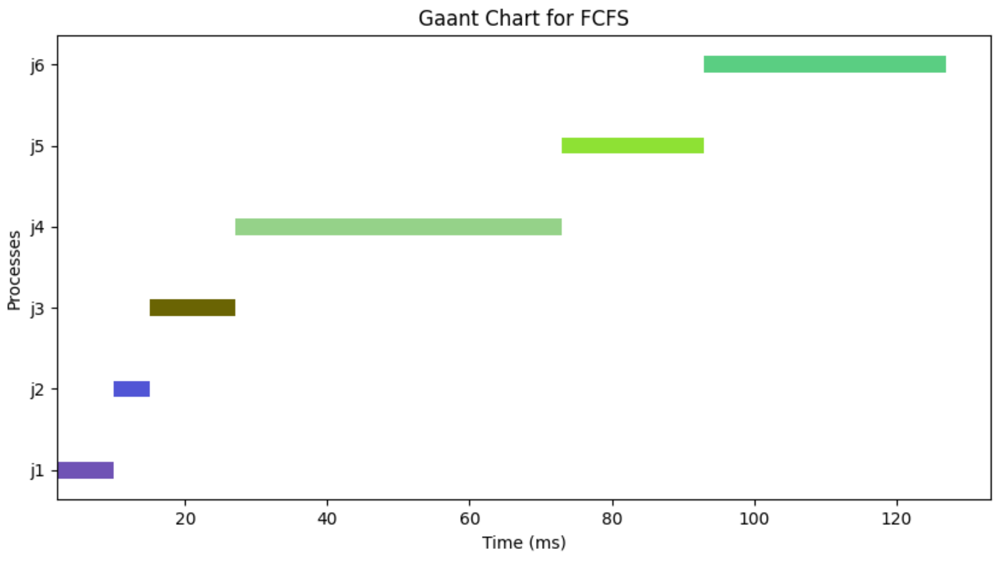
On the other hand, it has its advantages as well. Simplicity and no use of complex data structures in its implementation makes it easy to use. All the processes are scheduled in the order in which they arrive, thus no unbiasedness in scheduling.

Simulation 1.2.2

Lets us reduce the mean value of exponential distribution to 2 to reduce the arrival times and increase the burst times to up-to 50 time units. Following 6 processes are generated using the mentioned parameters:

PID	Arrival Time	Burst Time
J1	2	8
J2	3	5
J3	4	12
J4	4	46
J5	4	20
J6	8	34

Gantt chart:



Blue Print: j1 2 10 j2 10 15 j3 15 27 j4 27 73 j5 73 93 j6 93 127

Average Turnaround Time: 53.333

Average Response Time: 32.500

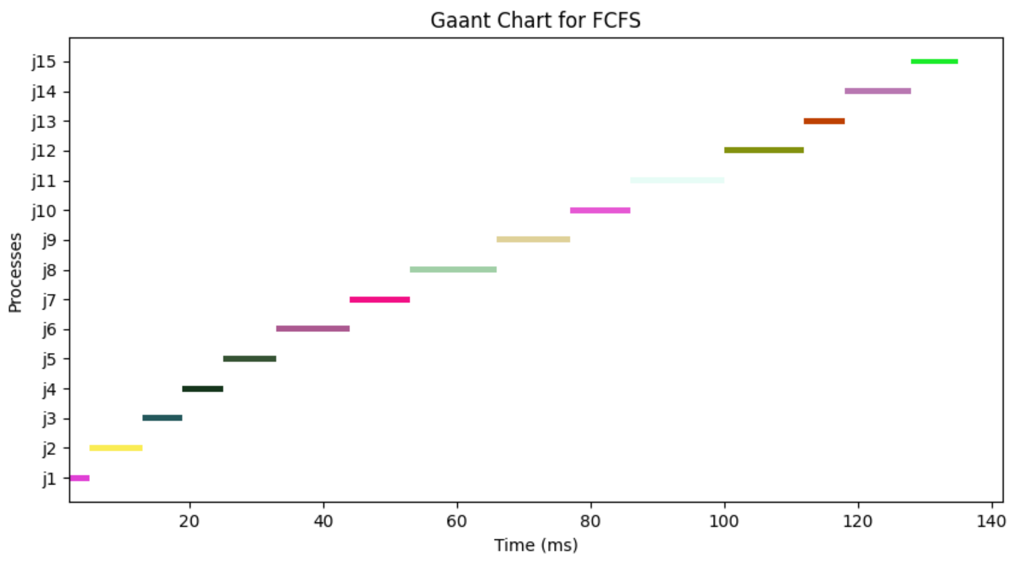
Although frequently, FCFS leads to unacceptable turnaround and response times for some processes, sometimes, when the bursting time of the processes are in roughly increasing order, it can be a good algorithm as it is unbiased w.r.t all the processes. Jobs are scheduled only in the order they arrive. In practise, its still not feasible to use because of no priority considerations of different jobs and significant Convoy effects.

Simulation 1.2.3

We have generated 15 processes here with exponential time distribution having mean value of 4 and burst time generated randomly with an upper cap of 15. The following table describes the generated processes:

PID	Arrival Time	Burst Time
J1	2	3
J2	3	8
J3	4	6
J4	14	6
J5	15	8
J6	17	11
J7	18	9
J8	23	13
J9	32	11
J10	36	9
J11	37	14
J12	40	12
J13	44	6
J14	45	12
J15	48	7

Gantt chart:



Blue Print: j1 2 5 j2 5 13 j3 13 19 j4 19 25 j5 25 33 j6 33 44 j7 44 53 j8 53 66 j9 66 77 j10 77 86 j11 86 100 j12 100 112 j13 112 118 j14 118 128 j15 128 135

Average Turnaround Time: 42.400

Average Response Time: 33.533

2. SJB: Shortest Job First

2.1 Algorithm

```
•  Process ← structure defining the process variables.
•  Context ← structure containing the PID of running process, start time and time of
   next switch.
•  Input: Array Processes of type 'struct Process', integer N (Number of Processes)

Steps:

•  Initialize an array Contexts of type 'struct Context' of size N.
•  Initialize completed = 0, current_time = 0.
•  Define a function 'findShortestJob' :
   o  Input: Processes array, size N, and current_time
   o  Loop over the Processes array to find all processes with arrival_time <=
      current_time and not yet completed.
   o  Output: The index of the process with smallest burst time among these.
•  While(completed < N)
   o  Index ← findShortestJob(Processes, N, current_time)
   o  Run Process[Index] on the CPU.
   o  Create a context C.
         ▪  C.PID ← Processes[Index].PID
         ▪  C.start_time ← current_time.
         ▪  C.end_time ← current_time + Processes[Index].burst_time
   ▪  Increment completed by 1.
   ▪  Update the current_time to C.end_time.
   ▪  Calculate the Response time and Turnaround time of Processes[Index]
•  Output: Display the Blue print stored in the Contexts array, average Response time
   and average Turnaround time.
```

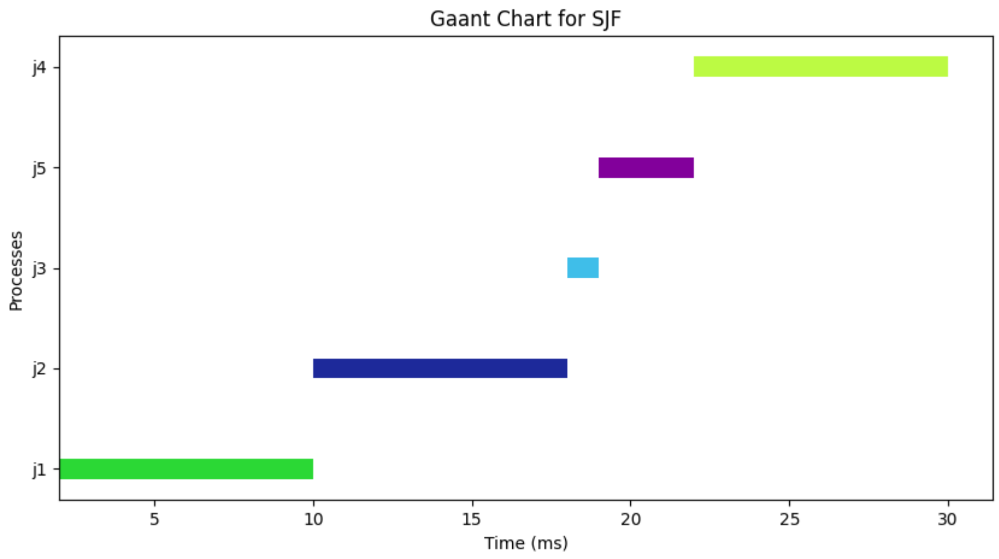
2.2 Simulation

Simulation 2.2.1:

We have generated 5 processes with exponential distribution parameter 5 and burst time randomly with an upper cap of 15. Following are the details for the processes generated:

PID	Arrival Time	Burst Time
J1	2	8
J2	3	8
J3	11	1
J4	11	8
J5	12	3

Gantt chart:



Blue Print: j1 2 10   j2 10 18   j3 18 19   j5 19 22   j4 22 30

Average Turnaround Time: 12.000

Average Response Time: 6.400

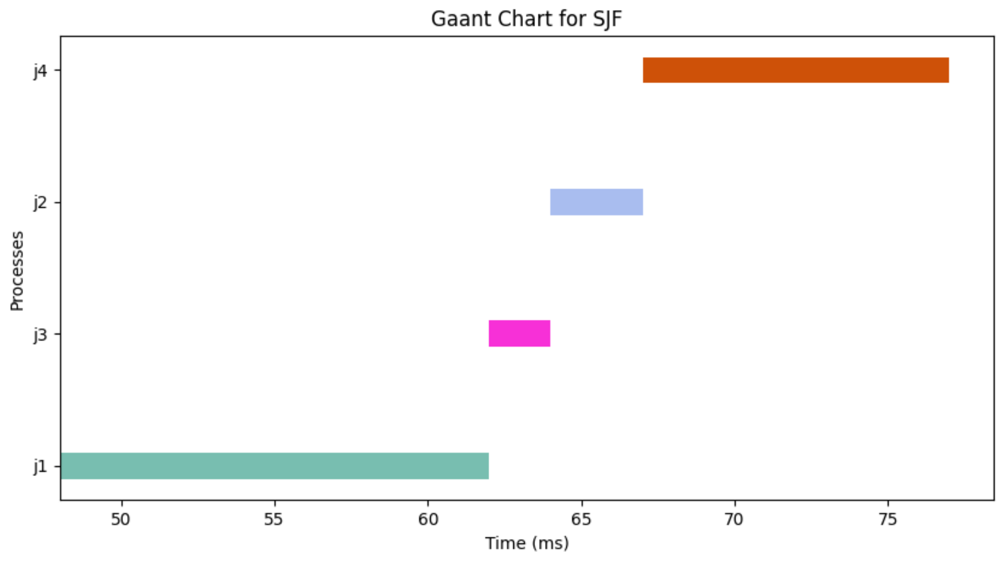
Analysis: SJF provides optimal average waiting time among all the scheduling algorithms known. Since shortest jobs are executed first, they don't have to starve behind longer jobs arriving early. This leads to reduction in Convoy effect. SJF also leads to better CPU utilization, since its occupied in always processing several shorter jobs, instead of being monopolized over a single long running Job. At 11 time instant, both j3 and j4 arrived in the system, but since j3 has shorter burst time than j4, it got scheduled before j4. J3 finished execution at time instant 19, by then j5 had also arrived in the system with burst time even less than j4. So j4 was pushed to be executed at the last stage.

Simulation 2.2.2

Now we have generated 4 processes using exponential distribution with mean value of 10 and burst time generated randomly, but with its upper cap slightly increased to 20. Following table summarized the processes generated:

PID	Arrival Time	Burst Time
J1	48	14
J2	51	3
J3	56	2
J4	58	10

Gantt chart:



Blue Print: j1 48 62   j3 62 64   j2 64 67   j4 67 77

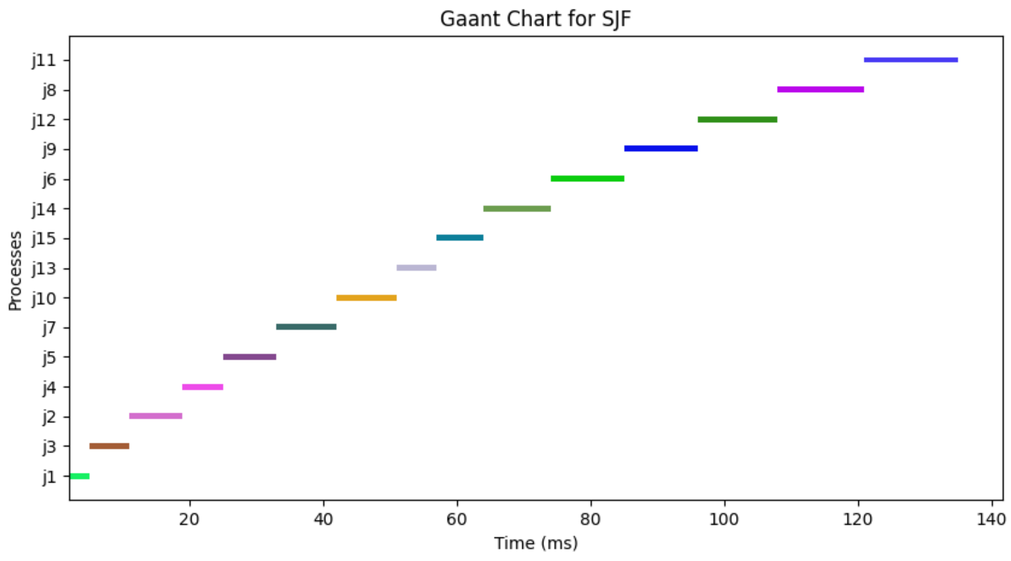
Average Turnaround Time: 14.250

Average Response Time: 7.000

SJF, also being a non-pre-emptive scheduling algorithm, has the biggest drawback that in real world scenarios, the completion/burst time of job cannot be known beforehand. This algorithm also leads to starvation for longer processes, when shorter ones keeps arriving. This will be particularly painful when new tasks are continuously incoming. Like FCFS, SJF also does not take the priority of the process into account at all. In actual implementation in a real OS, it would require estimation of the completion time of a job, based on historical data, which always comes with its margin of error, thus limiting its usage. Although the relatively shorter Jobs j2 and j3 arrived in the system just after j1, they had to starve till the execution of the longer process j1 to finish its execution before being scheduled, hampering their response and wait time.

Simulation 2.2.3:

Refer to the 15 processes generated in **simulation 1.2.3.** Following is the gantt chart with SJF algorithm applied to the same set of 15 processes.



Blue Print: j1 2 5   j3 5 11   j2 11 19   j4 19 25   j5 25 33   j7 33 42   j10 42 51   j13 51 57   j15 57 64   j14 64 74   j6 74 85   j9 85 96   j12 96 108   j8 108 121   j11 121 135

Average Turnaround Time: 36.533

Average Response Time: 27.667

3. SRTF: Shortest Remaining Time First

31. Algorithm

```
• Process 🡐 structure defining the process variables.
• Context 🡐 structure containing the PID of running process, start time and time of
  next switch.
• Input: Array Processes of type 'struct Process', integer N (Number of Processes)

Steps:

• Initialize an array Contexts of type 'struct Context' of dynamic size.
• Initialize completed = 0, current_time = 0, index = 0, currentJobIndex = -1
• Define a function 'findShortestRemainingTimeJob' :
    o Input: Processes array, size N, and current_time
    o Loop over the Processes array to find all processes with arrival_time <=
      current_time and not yet completed.
    o Output: The index of the process with smallest Remaining Burst time among
      these.
• While(completed < N)
    o shortestJobIndex 🡐 findShortestRemainingTimeJob(Processes, N, current_time)
    o If shortestJobIndex is not Equal to currentJobIndex:
        o …Perform Context Switch…
        o Create a new Context C with:
            ▪ C.PID 🡐 Processes[shortestJobIndex].PID
            ▪ C.start_time 🡐 current_time
        o Append the Context C to the Contexts array.
        o Terminate the last Process: Contexts[index-1].end_time 🡐 current_time
          and increment index by 1.
        o Update currentJobIndex 🡐 shortestJobIndex
    o Else shortestJobIndex is same as the curentJobIndex:
        o Run Process[currentJobIndex] on the CPU.
        o Calculate the response time of the current executing Process.
        o Update the current time and decrement the remaining time of the current
          Process.
        o If the remaining time is 0, increment completed by 1 and calculate the
          current Processes' turnaround time.
• Output: Display the Blue print stored in the Contexts array, average Response time
  and average Turnaround time.
```

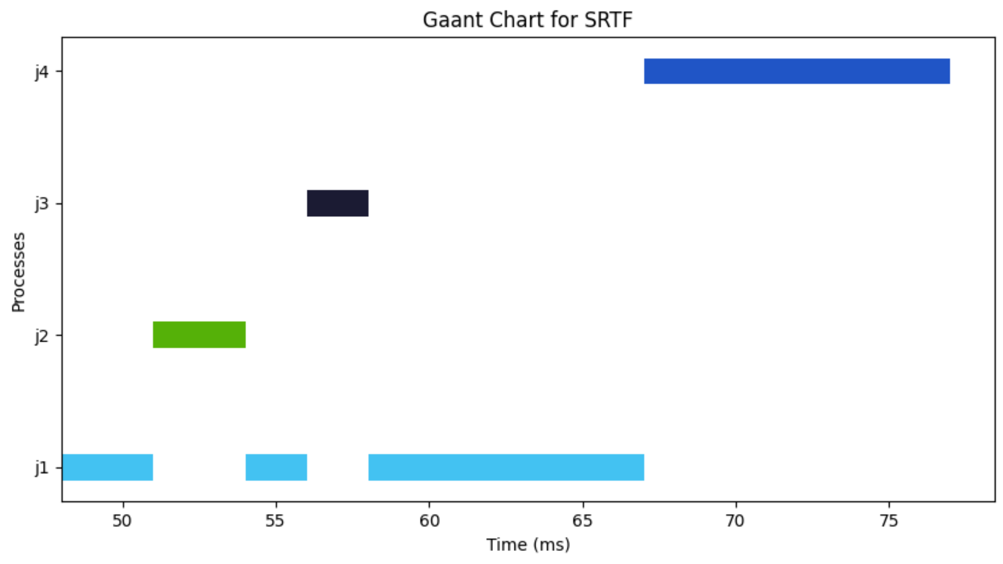
3.1 Simulation

Simulation 3.1.1:

Since, SRTF is just a pre-emptive version of SJF, lets observe the results of SRTF on the same data set of processes generated and used in SJF in simulation 2.2.2.

PID	Arrival Time	Burst Time
J1	48	14
J2	51	3
J3	56	2
J4	58	10

Gantt chart:



Blue Print: j1 48 51 j2 51 54 j1 54 56 j3 56 58 j1 58 67 j4 67 77

Average Turnaround Time: 10.750

Average Response Time: 2.250

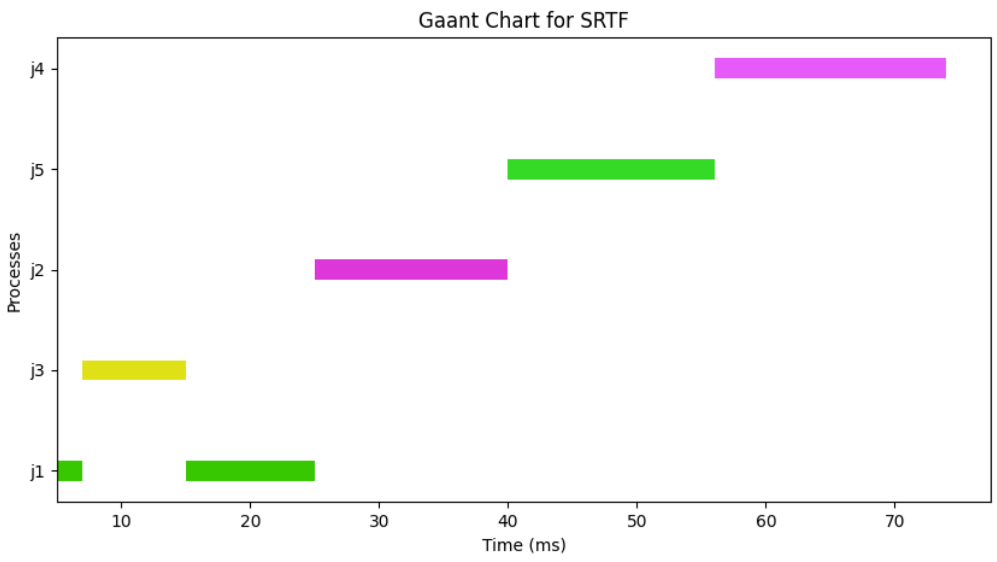
As compared to SJF, SRTF beats SJF both in terms of average response time as well as average response time. The improvement in average response time is significant. SRTF is the first pre-emptive algorithm we are discussing, justifying its significant improvement in performance as compared to non-pre-emptive algorithms. Like SJF, SRTF also provides optimal average waiting time among all the known algorithms. Further, unlike SJF, it is also adaptive to new incoming jobs, since it can interrupt a running process and schedule another. While j1 was running and had a higher burst time of 14 units, it was interrupted shortly after when j2 came in after 3 time units and requiring CPU for a short amount of time. J1 was stopped and j2 was executed first, then the CPU was again allotted to j1. Resuming the execution of j1, was followed by another incoming of a shorter job in the system, namely j3 and thus j1 was again interrupted. This way, SRTF keeps adapting to new jobs and provides improvement in efficiency.

Simulation 3.2.2:

Here we generate 5 process with exponential distribution mean as 2.5 and reduce the burst time to an upper cap of 12, generated randomly. The following table summarizes the processes:

PID	Arrival Time	Burst Time
J1	5	12
J2	5	15
J3	7	8
J4	7	18
J5	7	16

Gantt chart:



Blue Print: j1 5 7   j3 7 15   j1 15 25   j2 25 40   j5 40 56   j4 56 74

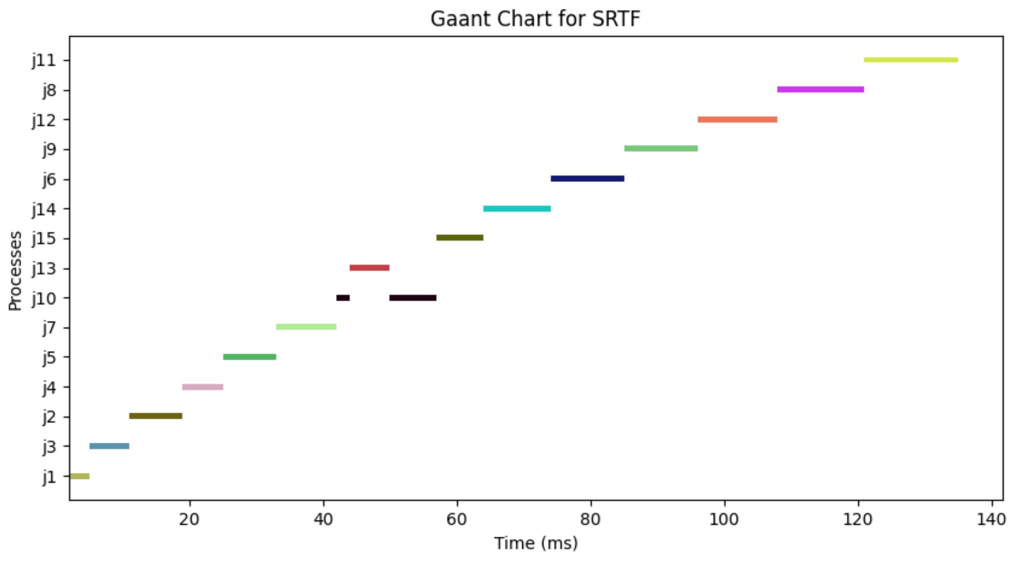
Average Turnaround Time: 35.800

Average Response Time: 20.400

Due to its pre-emptive nature, SRTF leads to increased context switching, more overheads and reduced system throughput. SJF would have achieved the scheduling of the above process in just 5 context switches. SRTF adds 1 more to it, and often adds more than 1. In simulation 3.2.1, as compared to SJF, SRTF adds to 2 more context switches. This algorithm also does take into account the priorities of the incoming jobs and the burst time of the jobs will not be know in real world, again restricting its applications in real world OS.

Simulation 3.2.3:

Refer to the 15 processes generated in **simulation 1.2.3.** Following is the gantt chart with SRTF algorithm applied to the same set of 15 processes.



Blue Print: j1 2 5 j3 5 11 j2 11 19 j4 19 25 j5 25 33 j7 33 42 j10 42 44 j13 44 50 j10 50 57 j15 57 64 j14 64 74 j6 74 85 j9 85 96 j12 96 108 j8 108 121 j11 121 135

Average Turnaround Time: 36.467

Average Response Time: 27.200

4. RR: Round Robin

4.1 Algorithm

```
• Process 🡐 structure defining the process variables.
• Context 🡐 structure containing the PID of running process, start time and time of
  next switch.
• Input: Array Processes of type 'struct Process', integer N (Number of Processes),
  integer time_slice

Steps:

• Initialize an array Contexts of type 'struct Context' of dynamic size
• Initialize completed = 0, current_time 🡐 Processes[0].arrival_time
• Sort the Processes array by arrival time. If the arrival time is same, sort by
  lexicographic ordering.
• Initialize a Queue Q_process and Enqueue Processes[0] to Q_process.
• While(Q_process is Not Empty)
    o P🡐 dequeue(Q_process).
    o Run Process P on the CPU for maximum of time_slice much amount of time.
    o Create a context C.
        ▪ C.PID 🡐 Processes[Index].PID
        ▪ C.start_time 🡐 current_time.
    o If P.remaining_burst_time <= time_slice,
        ▪ C.end_time = current_time + P.remaining_burst_time.
        ▪ Current_time = C.end_time.
        ▪ Calculate response and turnaround time for P and mark P as completed
    o Else:
        ▪ P.remaining_burst_time --= Q.
        ▪ C.end_time = current_time + Q.
        ▪ Current_time = C.end_time.
    o For each process P in Processes, if process has not been completed and has not
      been added to Q_processes and process.arrival_time <= current_time:
        ▪ Enqueue(Q.processes, P).
    o If P.remaining_burst_time > 0
        ▪ Eneuque P back to Q_processes.
• Output: Display the Blue print stored in the Contexts array, average Response time
  and average Turnaround time.
```

4.2 Simulation

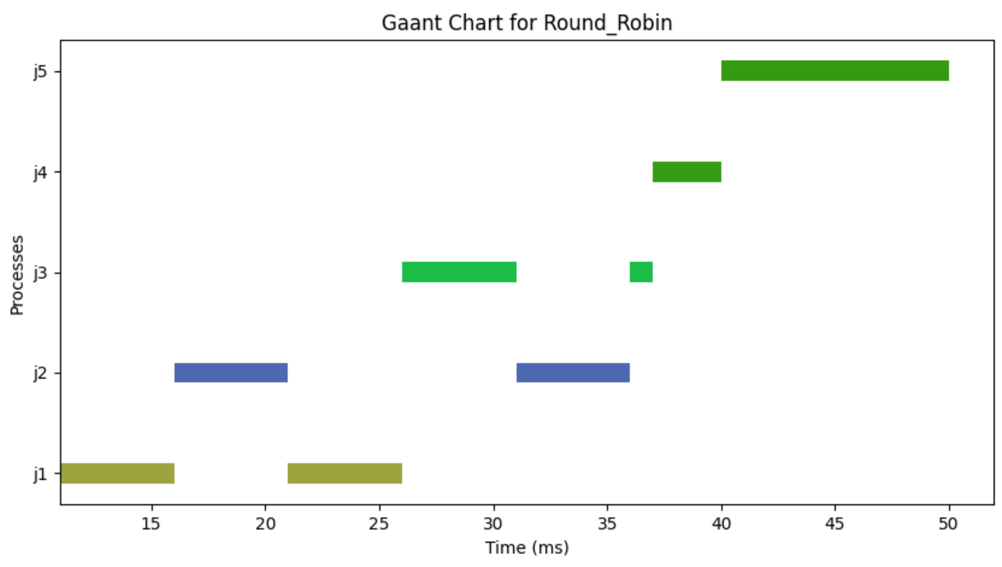
Simulation 4.2.1:

Lets observe the results of Round Robin algorithm on the 5 processes generated using exponential distribution with mean value 10 and burst time generated randomly with an upper cap of 15.

PID	**Arrival Time**	**Burst Time**
J1	11	10
J2	13	10
J3	17	6
J4	37	3
J5	37	10

With a time slice of 5 units, RR shows the following performance:

Gantt chart:



Blue Print: j1 11 16 j2 16 21 j1 21 26 j3 26 31 j2 31 36 j3 36 37 j4 37 40 j5 40 50

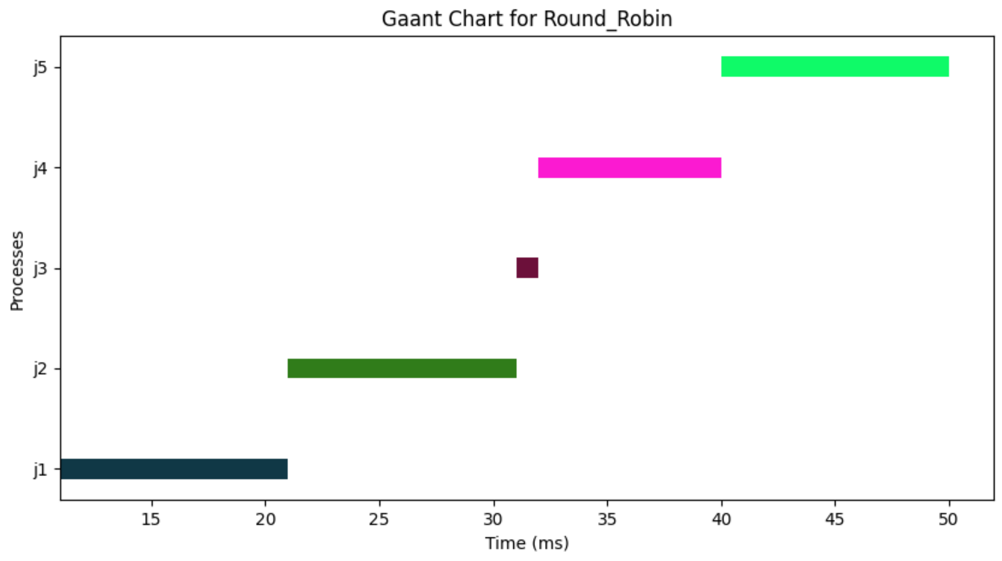
Average Turnaround Time: 14.800

Average Response Time: 3.000

Simulation 4.2.2:

On the same data set, if we change the time slice to 10 units, the following performance is noted:

Gantt chart:



Blue Print: j1 11 21   j2 21 31   j3 31 32   j4 32 40   j5 40 50

Average Turnaround Time: 28

Average Response Time: 12.8

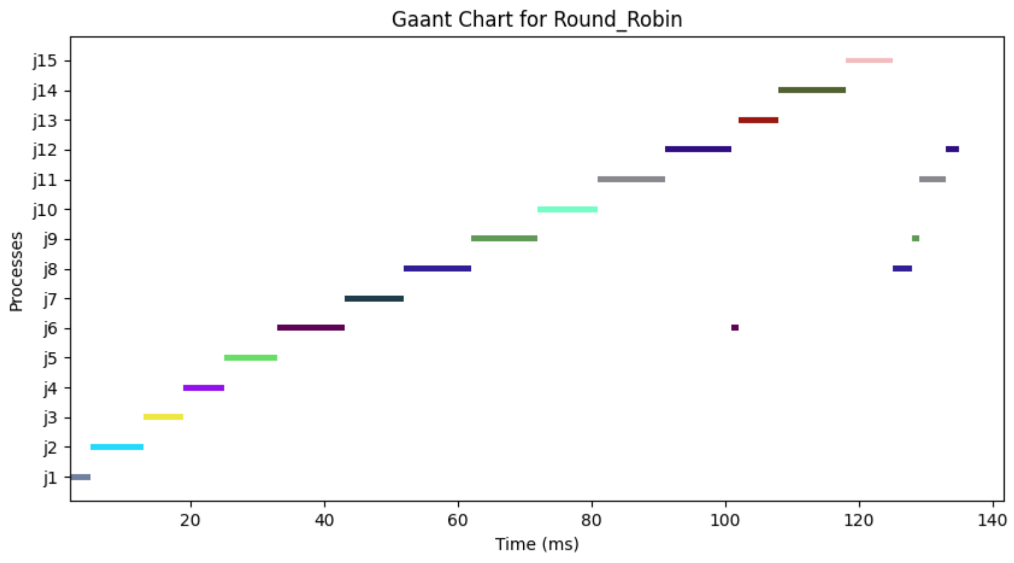
There is significantly notable difference between the two gantt charts (time slice of 5 and 10 respectively). Time slice of 5 ensures better response time to new incoming processes, however at the expense of a large number of context switches, leading to decrease in performance of the system and increased overheads. Increasing time slice reduces the context switches to just a few, however jeopardizing the wait time of shorter jobs arriving in the system. Establishing a balance by optimally choosing the time slice is of crucial importance in RR algorithm.

The biggest advantage of RR comes with fairness. The algorithm is never biased towards any process. All jobs get equal share of the CPU. Shorter jobs can be executed faster with RR, if the time slice is of sufficient length. The outcome of this algorithm is very straightforward, one can easily predict which job will have the CPU at any given time.

The efficiency of the RR highly depends on the choice of the time slice. A smaller time slice will increase the number of context switches, while a larger time slice will drive the algorithm towards FCFS and will be less responsive. RR does not always provide average waiting time as good as SJF or SRTF. This algorithm also lacks the consideration of priorities to the jobs arriving in the system. Further, if processes frequently block I/O or other waits, the CPU can be left idle, waiting for processes to become ready again, especially if the time quantum isn't fully used.

Simulation 4.2.3:

Refer to the 15 processes generated in **simulation 1.2.3.** Following is the gantt chart with RR algorithm (with time slice of 10 units) applied to the same set of 15 processes



Blue Print: j1 2 5 j2 5 13 j3 13 19 j4 19 25 j5 25 33 j6 33 43 j7 43 52 j8 52 62 j9 62 72 j10 72 81 j11 81 91 j12 91 101 j6 101 102 j13 102 108 j14 108 118 j15 118 125 j8 125 128 j9 128 129 j11 129 133 j12 133 135

Average Turnaround Time: 55.200

Average Response Time: 29.867

5. MLFQ: Multilevel Feedback Queue

5.1 Algorithm

```
• Process ← structure defining the process variables.
• Context ← structure containing the PID of running process, start time and time of
  next switch.
• Input: Array Processes of type 'struct Process', integer N (Number of Processes),
  TsRR1, TsRR2, TsRR3, TsRR4, boost_time

Steps:

• Initialize an array Contexts of type 'struct Context' of dynamic size.
• Initialize 3 Queues Q1, Q2 and Q3 of type 'struct Process'
• Initialize completed = 0, current_time = 0.
• Sort all the processes by arrival time. If arrival time is same, then prefer lexicographic
  ordering.
• While(completed < N)
    o Load all the new arrivals upto the current_time into Q1.
    o Create a Context C for the process to be executed.
    o If (Q1 is Not Empty)
        ▪ P ← Q1.front() and initialize time ← 0
        ▪ While(time < TsRR1)
            • Execute P on the CPU. Increment time and current time.
              Decrement the remaining time of P.
            • If P.remaining_time == 0, dequeue P from Q1 and store the
              context. Break.
        ▪ If p.remaining_time > 0 , then dequeue P from Q1 and append P into
          Q2.
    o Else if (Q2 is Not Empty)
        ▪ P ← Q2.front() and initialize time ← 0
        ▪ While(time < TsRR2)
            • Execute P on the CPU. Increment time and current time.
              Decrement the remaining time of P.
            • If P.remaining_time == 0, dequeue P from Q2 and store the
              context. Break.
        ▪ If p.remaining_time > 0 , then dequeue P from Q2 and append P into
          Q3.
    o
    o Else if (Q3 is Not Empty)
        ▪ P ← Q3.front() and initialize time ← 0
        ▪ While(time < TsRR3)
            • Execute P on the CPU. Increment time and current time.
              Decrement the remaining time of P.
            • If P.remaining_time == 0, dequeue P from Q3 and store the
              context. Break.
        ▪ If p.remaining_time > 0 , then dequeue P from Q3 and append P at
          the back of Q3.
    o
    o Else
        ▪ No current arrivals are there to be executed. Increment current time.
```

- If (boosting period occurs)
    - Move all the new arrivals, and process from Q2 and Q3 into Q1.

- **Output**: Display the Blue print stored in the Contexts array, average Response time and average Turnaround time.

  REMARKS: We have followed the first implementation discussed on Piazza, i.e., when boosting happens when a process is being executed on the CPU, first we complete the current execution upto the time slice of Queue, then push all the processes into Q1.

5.2 Simulation

We have simulated for N = 5 processes. The processes are generated with exponential distribution having a mean value of 5. The burst time are randomly generated with an upper cap of 20. The following is the summary of the processes generated:

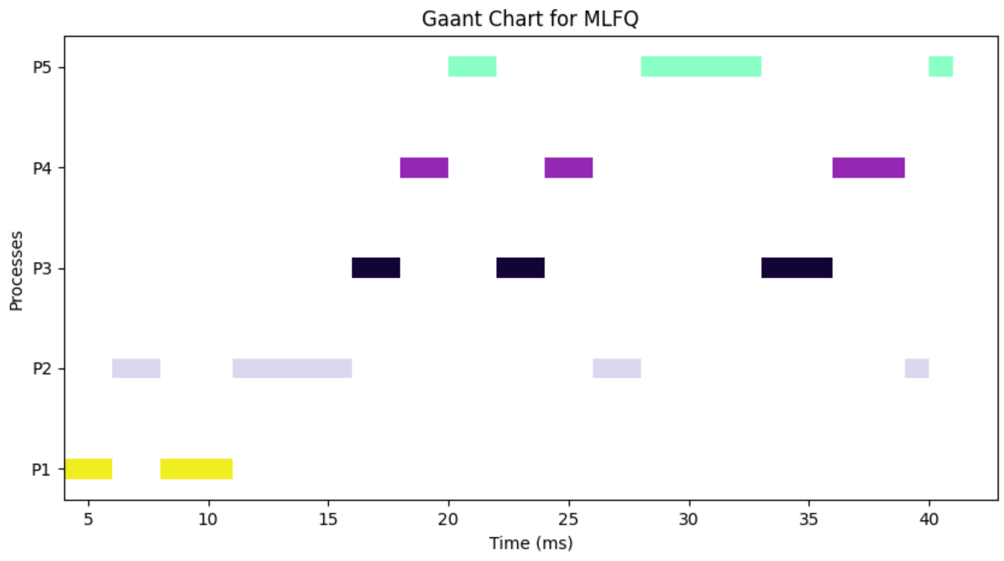
PID	Arrival Time	Burst Time
J1	4	5
J2	4	10
J3	14	7
J4	15	7
J5	20	8

Simulation 5.2.1:

We have tried to vary the time slices for the 3 different queues and observe and report the significant changes visible through the gantt charts and evaluation metric. Let Q1, Q2 and Q3 be the 3 Queues with Q1 of the highest priority and Q3 being the lowest priority. TsRR1, TsRR2, and TsRR3 are the corresponding time slices for the 3 Queues.

TsRR1 = 1, TsRR2 = 2, TsRR3 = 4

Gantt chart:



Blue Print: P1 4 5 P2 5 6 P1 6 8 P2 8 10 P1 10 12 P2 12 16 P3 16 17 P4 17 18 P3 18 20 P5 20 21 P4 21 23 P5 23 25 P2 25 28 P3 28 32 P4 32 36 P5 36 41

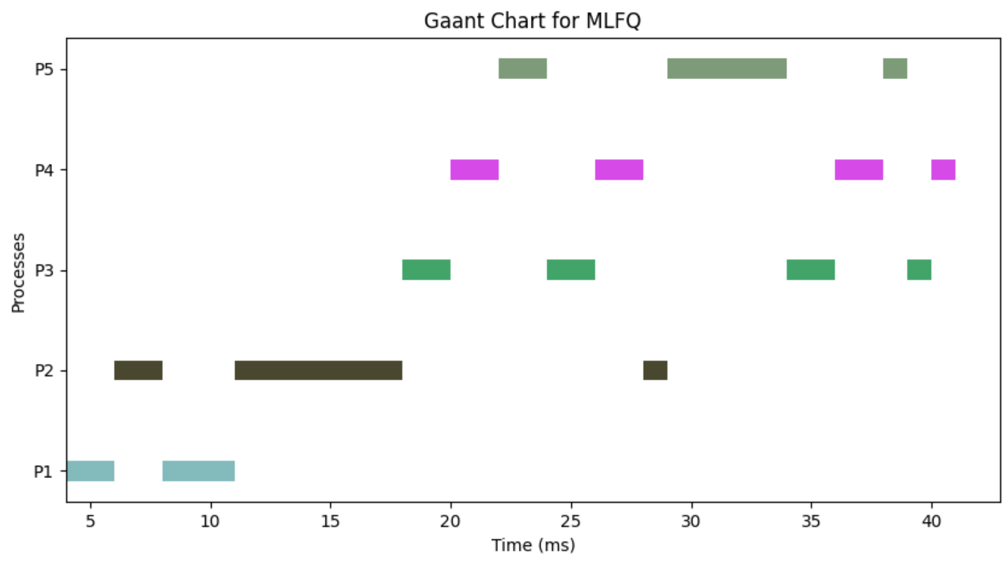
Avergae Turnaround Time: 18.400

Simulation 5.2.2:

Let us now increase all the 3 parameters by some extent, keeping the boosting time fixed at 5.

TsRR1 = 2, TsRR2 = 5, TsRR3 = 7

Gantt chart:



The average turnaround time has gone up a bit, but the number of context switches have gone down a little bit, leading to enhanced performance, by increasing the time slices of all 3 simultaneously.

Blue Print: P1 4 6   P2 6 8   P1 8 11   P2 11 16   P3 16 18   P4 18 20   P5 20 22   P3 22 27   P4 27 32   P5 32 37   P2 37 40   P5 40 41

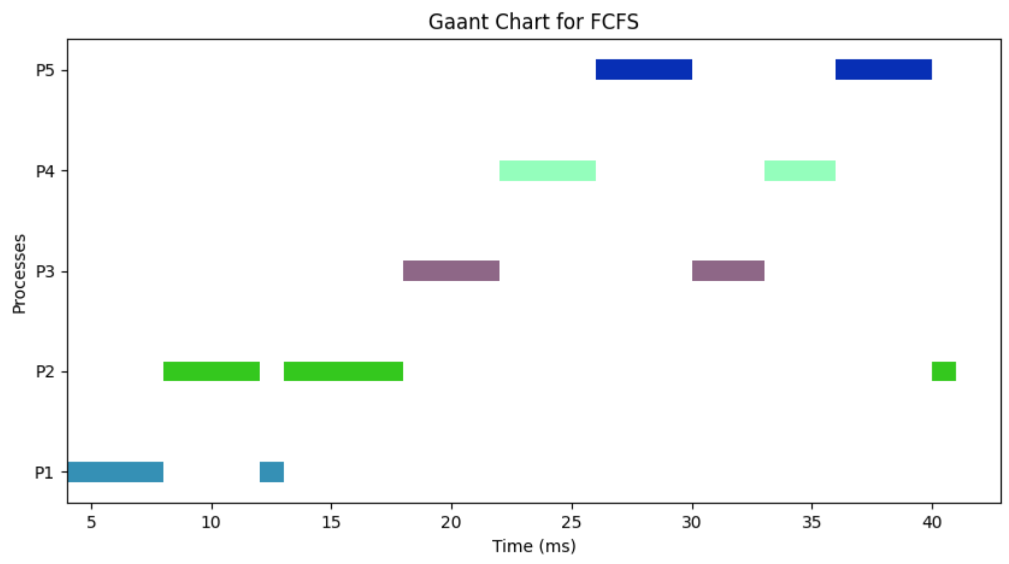
Avergae Turnaround Time: 18.800

Simulation 5.2.3:

Now let us keep everything fixed but only increase TsRR1 up to 4, but maintaining it less than TsRR2.

TsRR1 = 4, TsRR2 = 5, TsRR3 = 7

Gantt chart:



Blue Print: P1 4 8   P2 8 12   P1 12 13   P2 13 18   P3 18 22   P4 22 26   P5 26 30   P3 30 33   P4 33 36   P5 36 40   P2 40 41

Avergae Turnaround Time: 21.200

Increasing TsRR1 has lead to poor performance in terms of average response time.

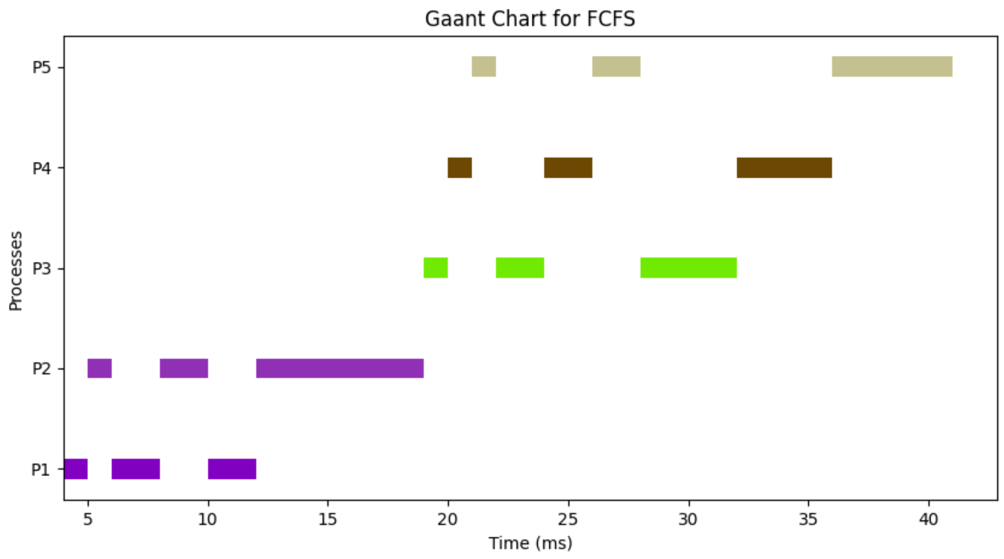
Conclusion: The higher priority Queue should have low values of time slice.

<u>Simulation 5.2.4:</u>

We shall now revert back to our low value of TsRR1 to 1. Keep TsRR2 fixed and now increase TsRR3.

TsRR1 = 1, TsRR2 = 2, TsRR3 = 9

<u>Gantt chart:</u>



Blue Print: P1 4 5 P2 5 6 P1 6 8 P2 8 10 P1 10 12 P2 12 19 P3 19 20 P4 20 21 P5 21 22 P3 22 24 P4 24 26 P5 26 28 P3 28 32 P4 32 36 P5 36 41

Avergae Turnaround Time: 16.600

Conclusion: Average Turnaround Time has significantly improved. Increase the time slices of lower priority Queues leads to enhanced performance.

NOTE: We have mostly dealt with integer time inputs of arrival and burst time in the experimenting part as the precision of time was not clarified/mentioned in the assignment.