CPU Scheduling Algorithms

First-Come-First-Served vs. Shortest-Job-First vs. Round Robin

Ethan Wolff

10/05/2024

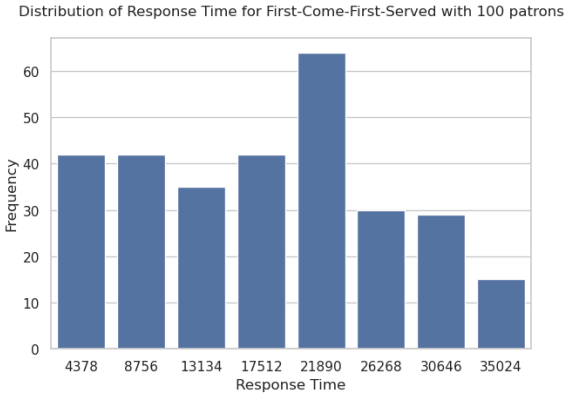
CSC3002F OS2

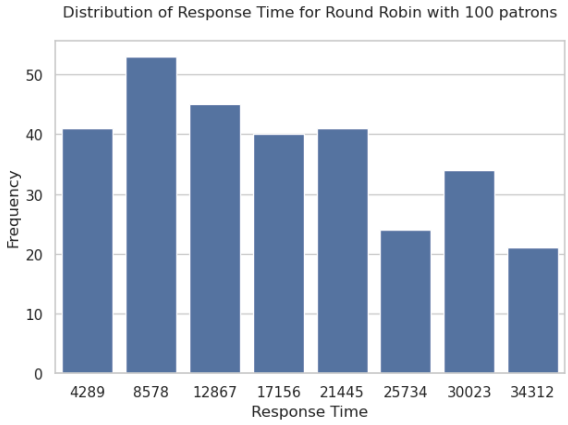
INTRODUCTION

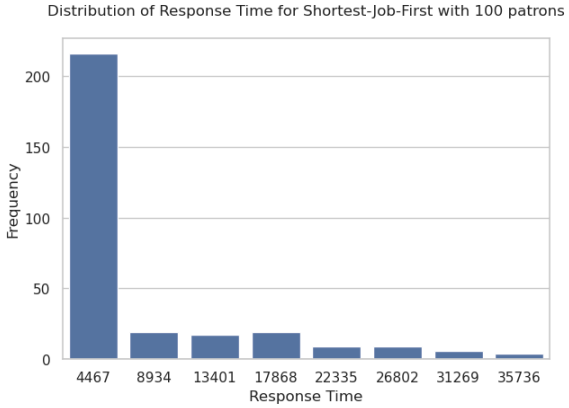
This project simulates CPU scheduling of processes in Java. The barman (CPU) implements First-Come-First-Served (FCFS), Shortest-Job-First (SJF) and Round Robin (RR) scheduling. The simulation records throughput, waiting time, turnaround time and response time, which are all plotted and compared for the different algorithms.

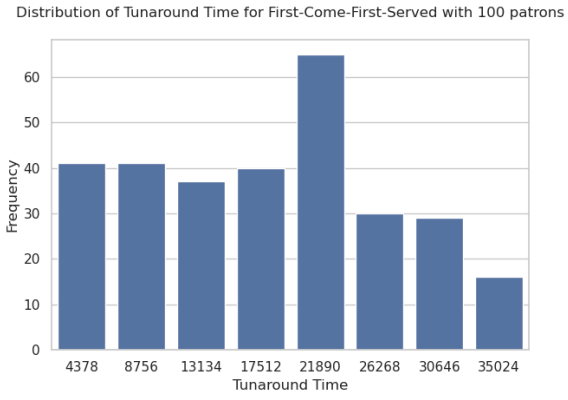
GRAPHS

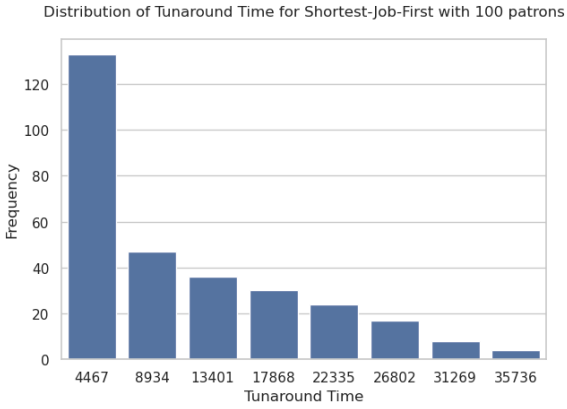
All the distributions were run with 100 patrons and the x-axis labels show all the values that landed between the previous label (or 0) and that label. i.e. for the first graph, the values that counted towards the first column were between 0 and 4378. Simulations were repeated over three iterations to dilute outliers. For the average and standard deviation of throughput, the y-axis label “Seconds” refers to patrons per second.

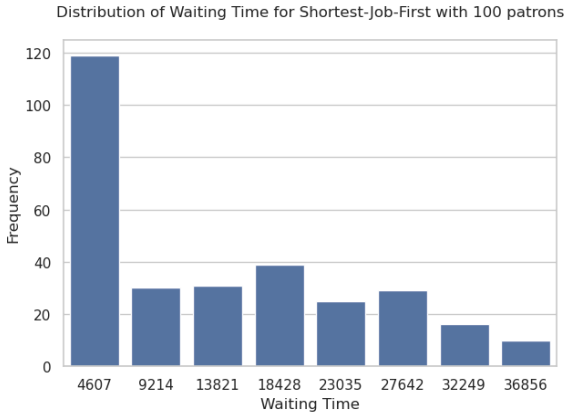


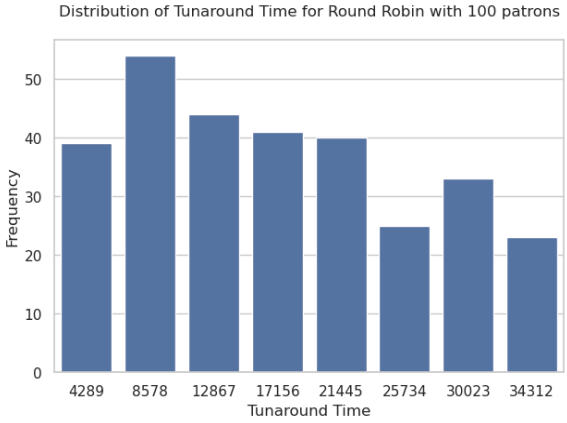


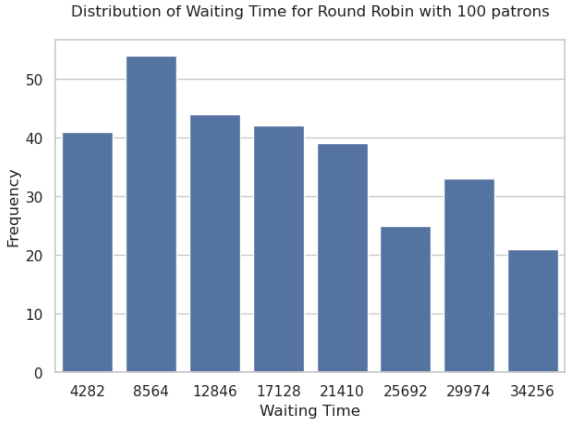


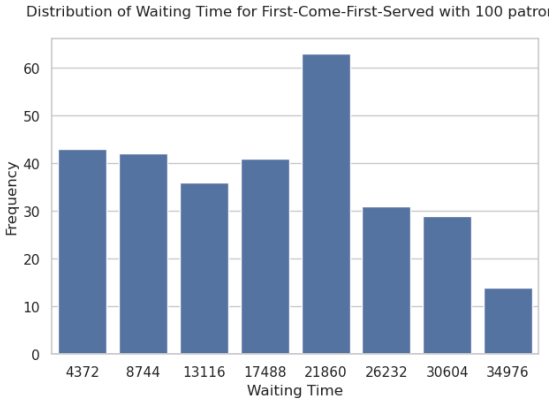


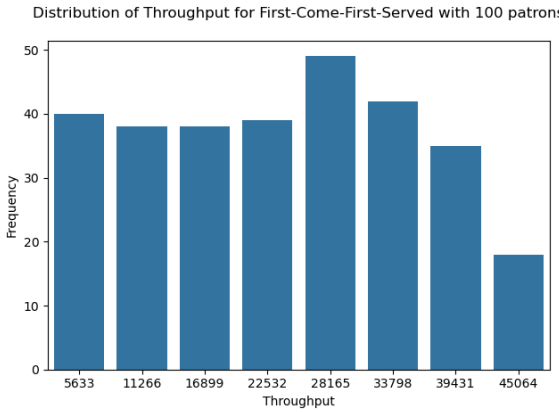

Distribution of Tunaround Time for Shortest-Job-First with 100 patrons
Frequency
0
20
40
60
80
100
120
4467
8934
13401
17868
22335
26802
31269
35736
Tunaround Time

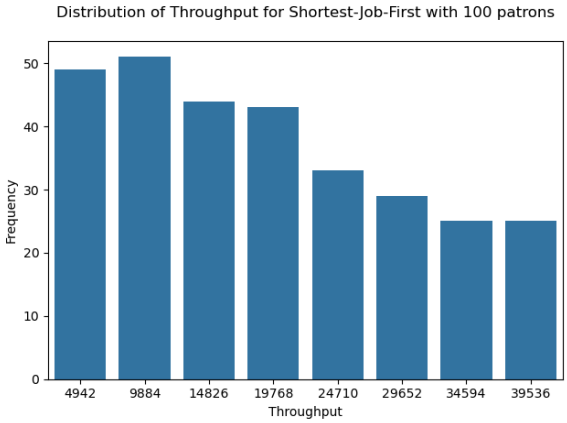

Distribution of Waiting Time for Shortest-Job-First with 100 patrons
Frequency
0
20
40
60
80
100
120
4607
9214
13821
18428
23035
27642
32249
36856
Waiting Time

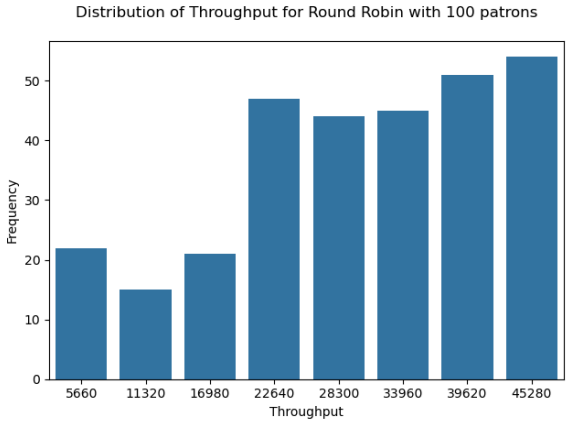

Distribution of Tunaround Time for Round Robin with 100 patrons
Frequency
0
10
20
30
40
50
4289
8578
12867
17156
21445
25734
30023
34312
Tunaround Time

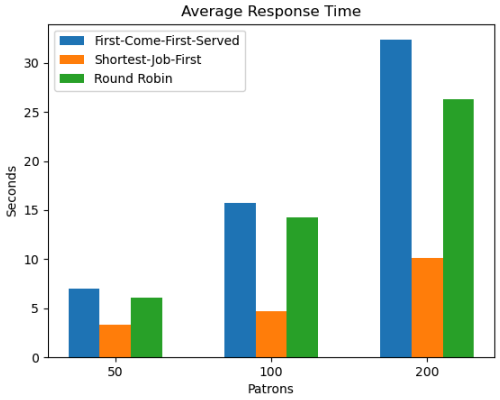

Distribution of Waiting Time for Round Robin with 100 patrons
Frequency
0
10
20
30
40
50
4282
8564
12846
17128
21410
25692
29974
34256
Waiting Time

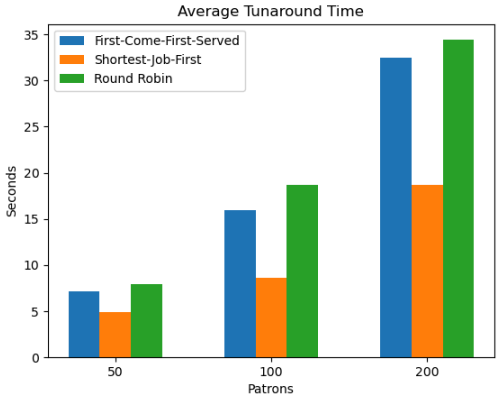

Distribution of Waiting Time for First-Come-First-Served with 100 patrons
Frequency
0
10
20
30
40
50
60
4372
8744
13116
17488
21860
26232
30604
34976
Waiting Time

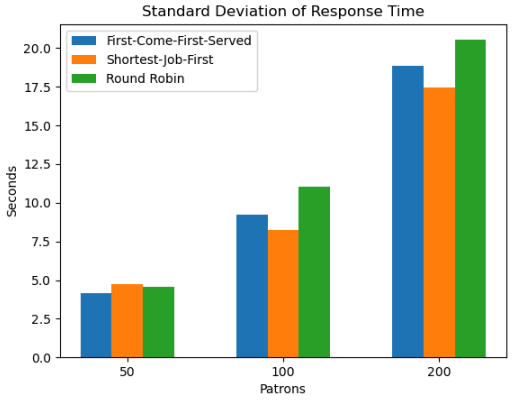

Distribution of Throughput for First-Come-First-Served with 100 patrons
Frequency
0
10
20
30
40
50
5633
11266
16899
22532
28165
33798
39431
45064
Throughput

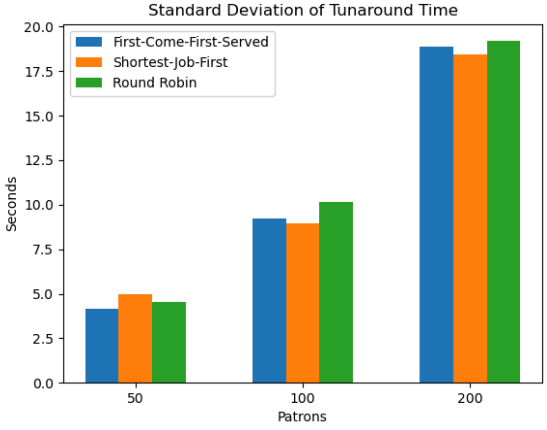

Distribution of Throughput for Shortest-Job-First with 100 patrons
Frequency
Throughput
4942
9884
14826
19768
24710
29652
34594
39536

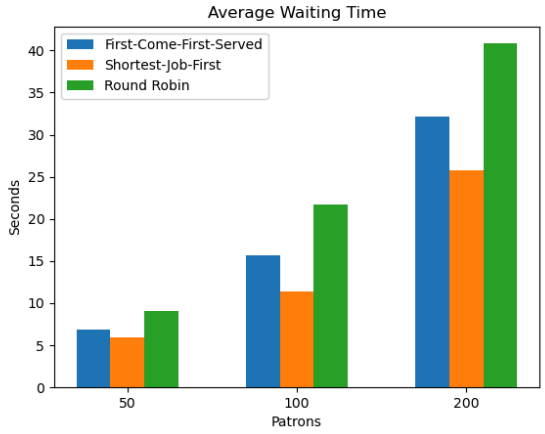

Distribution of Throughput for Round Robin with 100 patrons
Frequency
Throughput
5660
11320
16980
22640
28300
33960
39620
45280

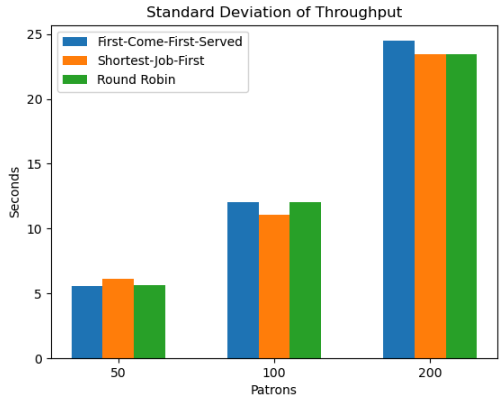

Average Response Time
First-Come-First-Served
Shortest-Job-First
Round Robin
Seconds
Patrons
50
100
200

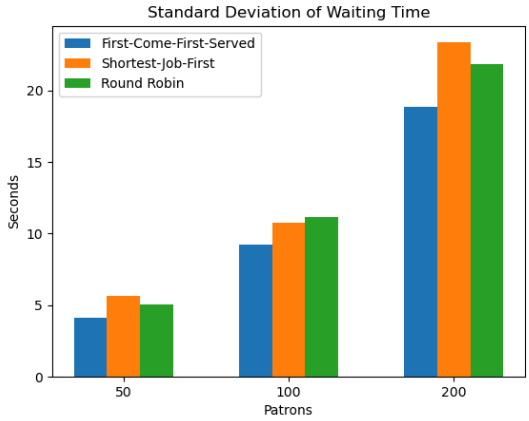

Average Tunaround Time
First-Come-First-Served
Shortest-Job-First
Round Robin
Seconds
Patrons
50
100
200

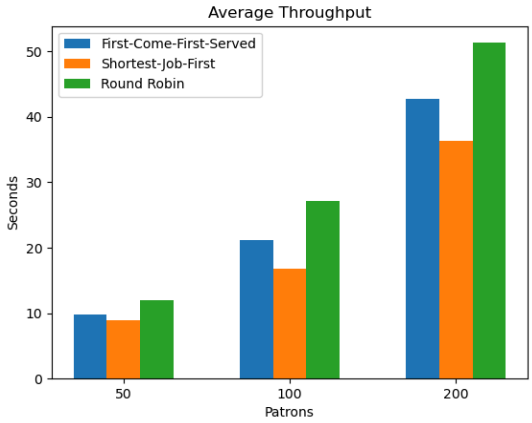

Standard Deviation of Response Time
First-Come-First-Served
Shortest-Job-First
Round Robin
Seconds
Patrons
50
100
200


Standard Deviation of Tunaround Time
First-Come-First-Served
Shortest-Job-First
Round Robin
Seconds
Patrons
50
100
200









DISCUSSION

As shown by the results being repeated for 50, 100 and 200 patrons, the number of patrons does not affect the results drastically enough to warrant any manipulation of the data.

As can be seen from the distribution of response time, SJF minimises response time massively compared to FCFS or RR (surprisingly). This would make it a far better choice for interactive applications that require low response time for users. For turnaround time, again, SJF minimises it far better than both FCFS and RR which have similar distributions. This could be because for all of the shorter tasks of which there are more than longer tasks, the waiting time is kept very, very short. The same occurs again for waiting time, as SJF is known to minimise waiting time. Throughput becomes more interesting, with SJF completing far more jobs in the beginning of execution as would be expected as shorter jobs are finished quicker, earlier. FCFS finishes jobs similarly, but

less sharply. Interestingly, RR shows the reverse, with more jobs finishing towards the end of execution. This makes sense however as in the beginning only the shortest tasks are completed before the CPU moves onto the next tasks, meaning that the number of tasks completed early before execution times are whittled down will be small.

From the averages, we see that FCFS and RR show similar results, with FCFS sometimes being better and RR sometimes being better, but SJF is always significantly better. It is possible that this simulation, where the demands are relatively similar and end after a certain period of time, makes SJF advantageous as starvation never becomes a possibility as new processes stop being added.

From the standard deviations, we see that the variability of these algorithms is extremely similar in this case. Again, this is perhaps biassed by the simulation, but that is out of scope for this project. None of the algorithms produce many more or fewer outliers and thus have roughly equivalent predictability. This provides no advantage to any of the algorithms over the others. Generally, both FCFS and SJF have cases in which the variability of their metrics will be affected, short processes for SJF as longer processes are avoided and long processes for FCFS as short processes must wait unnecessarily. Both of these cases are unlikely to occur in our simulation due to the randomness at play however, leading to steady predictability.

We can see that SJF seems to be superior to both FCFS and RR. SJF has a lower average response time to FCFS as shorter jobs will be prioritised, leading to processes that arrive later being run earlier than with FCFS. RR has the same effect as processes with long execution times are moved on from to allow other processes a chance. Clearly then, SJF and RR are far fairer than FCFS which falls prey to the “convoy effect”, which is when a complex process arrives first and blocks the execution of faster processes, making it seem like the system is not performant.

Interestingly, RR seems not nearly as performant as could be expected. This is possibly because the quantum of time each process was given before being placed back onto the queue was too short and the overhead of moving processes off and onto the queue weighted the final results more than it should have. A longer quantum would have reduced the percentage of processing time that this overhead consumes and therefore made the results more favourable to RR.

SJF seems like the optimal choice, but falls prey itself to starvation problems. What can happen is that one process has a long task that can never execute because smaller processes are constantly added to the queue quickly enough that the CPU can never

reach the larger process. This leads to starvation as the larger process may never execute or may simply take a very long time to execute. RR avoids this by ensuring that all processes are executed in an order similar to that in which they arrived (looping, of course) which means that starvation of this kind cannot occur, but also ensures that no process is prioritised as each gets the same amount of processing time so that the convoy effect cannot occur. In this sense, RR is the fairest of the three algorithms that we have compared. For cases where this form of starvation is known to not be possible, i.e. a limited number of processes are executed so large numbers of new small processes cannot be created to prevent large processes from running, SJF is the best choice as it will minimise waiting time and other metrics. In cases where this cannot be known, RR is the best choice as it will avoid starvation and the convoy effect of FCFS, as well as ensure that processing time is spread equally among processes, keeping metrics low, if not optimal as shown.