Contents

1 Running the Program

The CPU Scheduler Program was written in Python 3.8. There is no graphical user interface implemented and everything is command line based. The program takes in command line arguments and reports it back in the command line interface.

1.1 Command Line Arguments

The program requires a file with process details in order to run, if it is not provided one it will close and inform the user. No flags are required and they only change the default functioning of the program.

--auto	makes the program run automatically, without this flag it will be in manual mode
--fcfs	sets the scheduling algorithm to first come first serve
--sjf	sets the scheduling algorithm to short job first
--ps	sets the scheduling algorithm to priority scheduling
--rr	sets the scheduling algorithm to round robin
--quantum	requires an integer value after the flag, sets the quantum time used in the round robin algorithm
--fps	requires an integer value after the flag, sets how many milliseconds to simulate per real world second
-- help, --h	prints out all the arguments that can be provided to the command line

Flags for choosing scheduling algorithms and providing simulation details
In automatic mode, to start and stop the program, press the enter key. If manual mode, press enter to advance the program one millisecond.

1.2 Test Files

The program comes with over a hundred example files to test on and a Python program to generate more test cases, the generator creates up to 10 processes, with a random priority between 0 and 9, a start time between 0 and 20 milliseconds, and up to 10 alternating CPU and IO bursts that last between 0 and 25 milliseconds. The largest test case we used had 100 processes and generated the most clear data. All the test files can be found in the testfiles directory. Here is an example of what they look like:

```
test17_0 9 1 11 20 4 13 9 23 8 23 16
test17_1 20 8 23 20 17 14 14 9 14 18 13
test17_2 20 6 17 11 13 19 20 2 9 1 6 8 1 1 18 25 8
test17_3 13 8 8 20 12 8 10
test17_4 18 9 12
test17_5 4 4 5 7 22 3 25 9 6 19 6
test17_6 16 3 3 1 25 25 8 8 2 13 1 10 1
test17_7 10 2 25 1 3 18 22 3 11 4 2 12 23 12 12 14 25 12 8 11 3
test17_8 16 1 8 16 13 21 7 5 8 18 24 22 6 21 2
test17_9 25 0 1
test17_10 16 6 1 13 16 17 20 14 5
test17_11 9 2 13 18 19 1 20 9 4 20 8 23 3
test17_12 7 8 20 3 14 9 20 2 1 23 15 16 24 10 21 17 4 18 7
test17_13 9 1 14 6 7 15 24 16 3 12 11 16 19 18 9 22 17 16 3 24 7 2 21
test17_14 6 0 16 7 13 13 14 8 24 8 5 9 9
test17_15 20 9 12 12 4 25 25 9 15 2 23 25 9 12 16 10 4 11 1 8 25 23 2
test17_16 17 0 12 11 3 18 21 10 8 2 22 6 8
test17_17 12 3 4 5 8 5 11 12 21 8 6 23 8 21 15 7 24 3 16
test17_18 3 3 19 1 6 8 22 1 22 18 8 16 17 1 25 20 21
```

```
prioritytest0 0 0 20
prioritytest1 0 1 20
prioritytest2 0 2 20
prioritytest3 0 4 30 10 1
prioritytest4 0 5 30 10 1
```

```
fcfstest 0 0 20
fcfstest 1 0 20
fcfstest 2 0 20
fcfstest 3 0 20
fcfstest 4 0 20
```

1.3 Main Program

To run the program, run the python file **cpu_scheduling_sim.py** like you would any other python file, followed by a file name string, and optional flags. The most basic example would be this:

```
python3 cpu_scheduling_sim.py "file_name"
```

With additional flags, it may look like this:

```
python3 cpu_scheduling_sim.py "file_name" --auto --fps 60 --sjf
```

2 Performance and Comparison

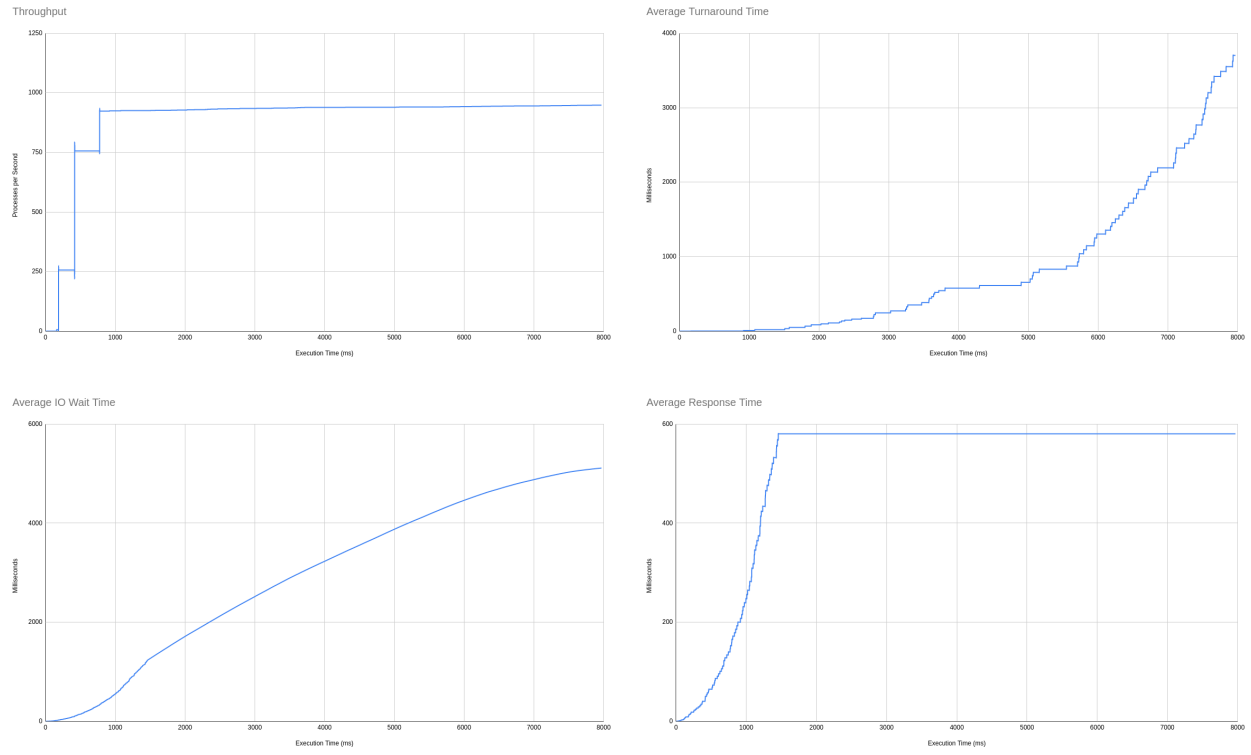
This section looks at the four algorithms that was implemented. Each algorithm was evaluated through the performance metrics. The performance metrics were:

- CPU Utilization: what percentage of CPU is not idle.
- Throughput: how many processes are completed per unit time.
- Turnaround Time: how long it takes a particular process to finish.
- Waiting Time: total time spent by all processes in the ready queue.
- Response Time: time between the user input and corresponding system action.

Response time is the sole criterion for real-time processes and the main criterion for interactive processes (and possibly system processes that they depend on). Batch processes are likely to care more about turnaround time. The other measures (which tend to be closely related) are most useful to check if the scheduler is being suboptimal or inconsistent, but waiting time can vary dramatically depending on scheduling policy, and is the factor that most closely affects response time.

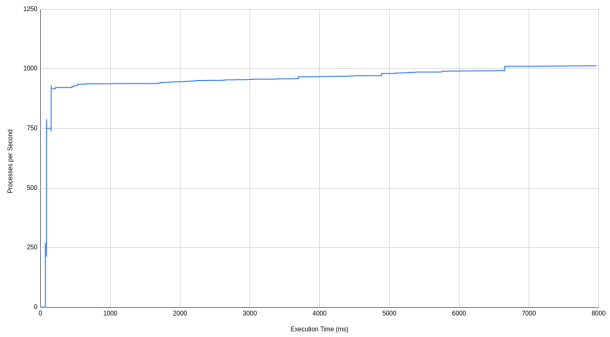
A summary of our performance metrics is visualized below.

2.1 First Come First Serve

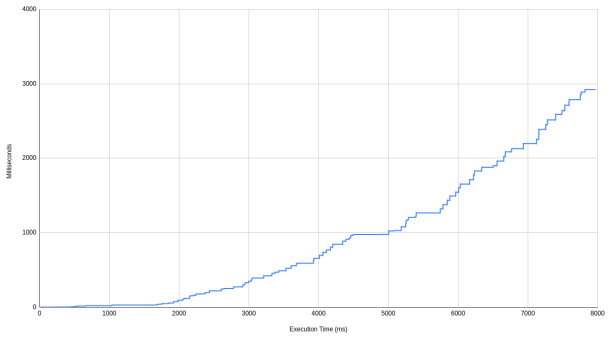


2.2 Shortest Job First

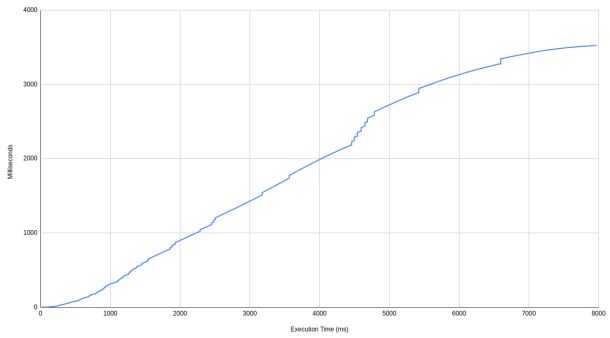
Throughput



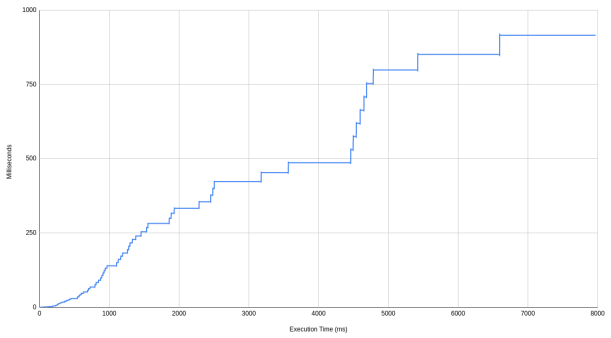
Average Turnaround Time



Average IO Wait Time

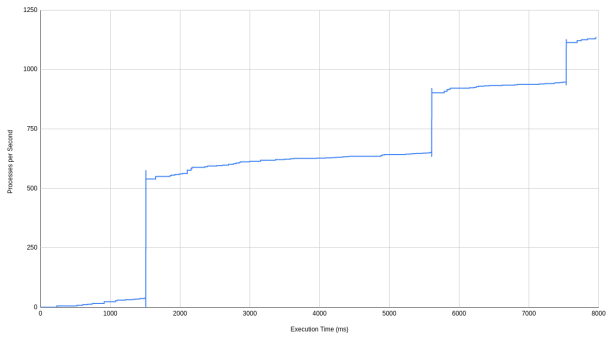


Average Response Time

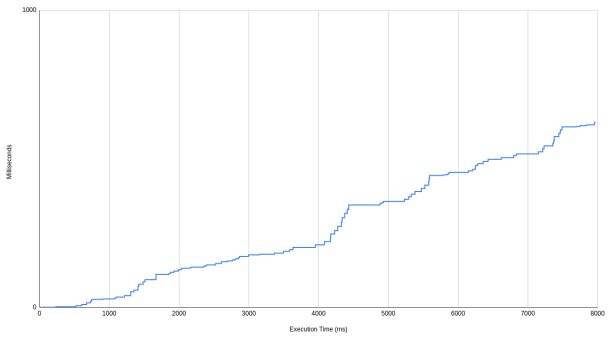


2.3 Priority Scheduling

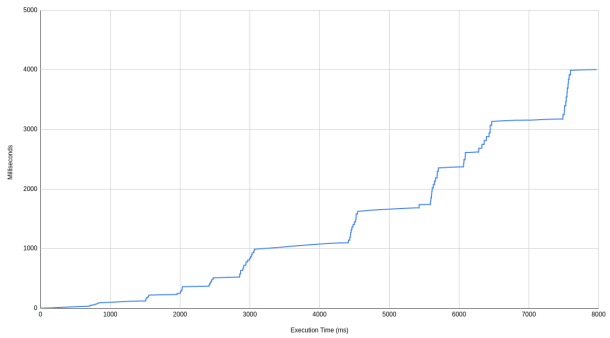
Throughput



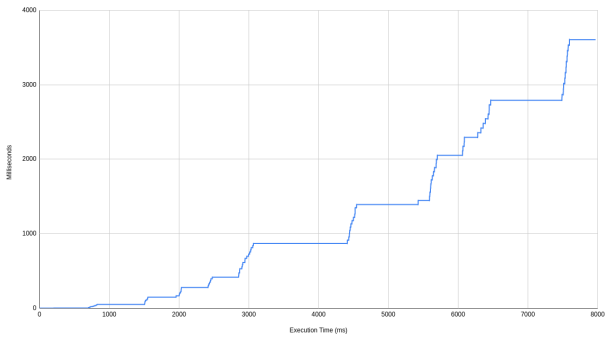
Average Turnaround Time



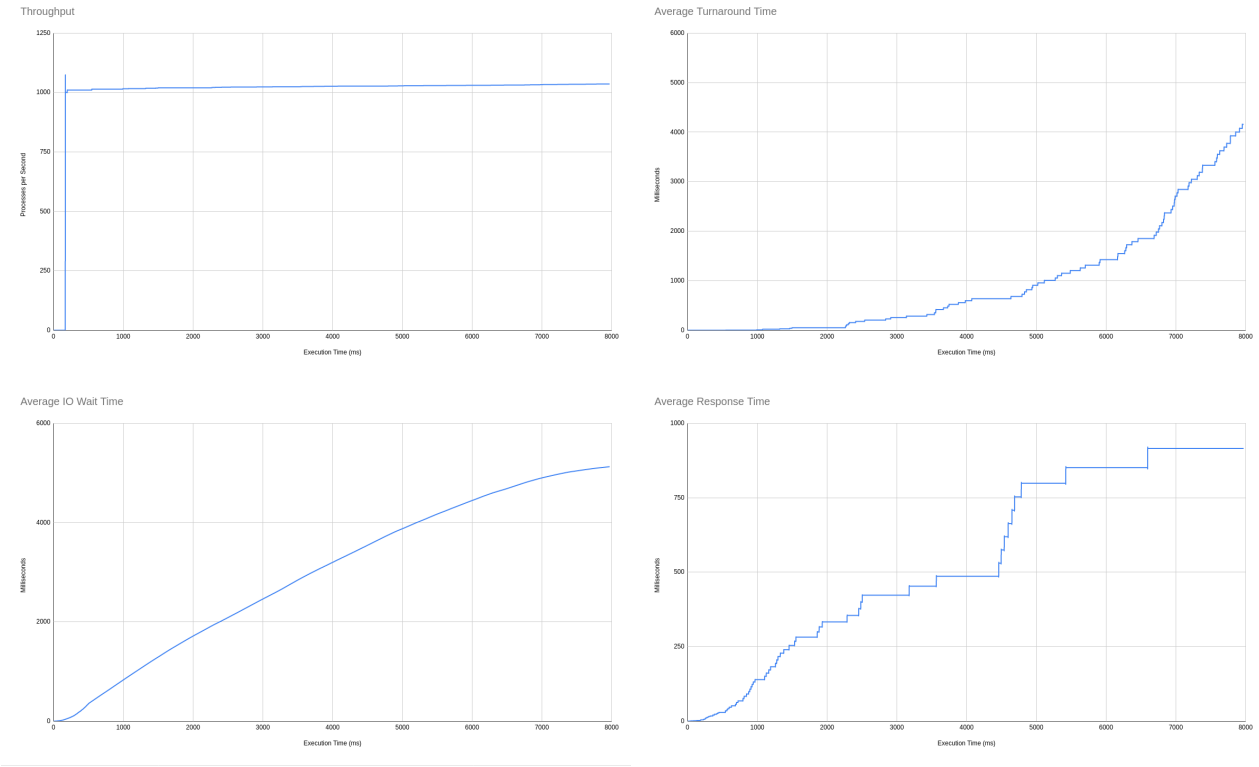
Average IO Wait Time



Average Response Time

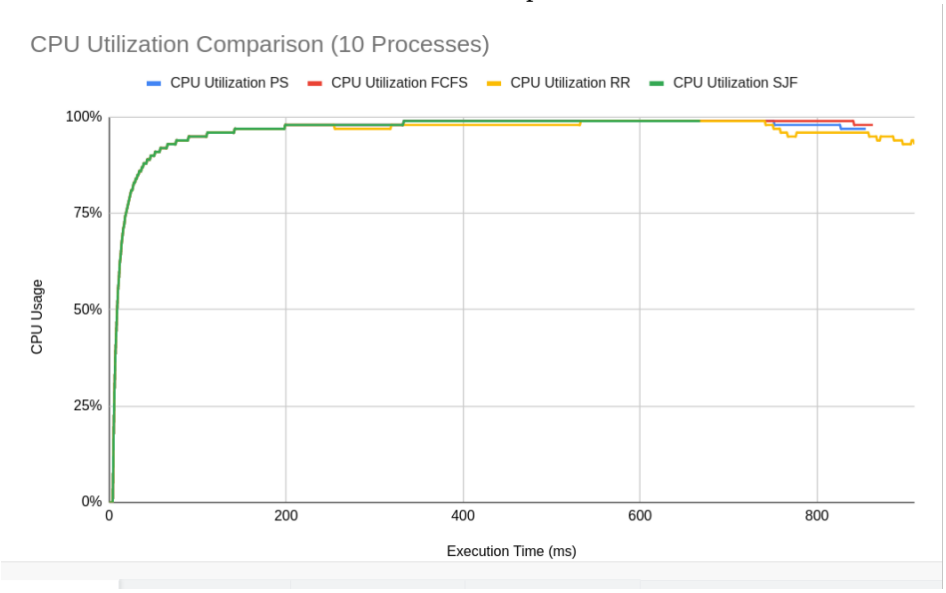


2.4 Round Robin



2.5 Comparing Algorithms

The test file ”performancetest” was used to benchmark the algorithms for many processes and the file ”test0” was used as a benchmark for a smaller set of processes.



Round Robin has the least average CPU utilization among the four algorithms. SJF has the highest average CPU utilization. Round Robin has the longest execution time. SJF has the least execution time.
The throughput graphs for FCFS, SJF, and RR are similar. The throughput graph for Priority Scheduling is different. This is because different processes have different priorities.

3 Discussion

This project included the four algorithms, FCFS, SJF, RR, and PR, which were implemented and tested. Multi-threading was implemented and two threads were used to simulate the scheduling algorithms and to handle the command line interface input. The outputs were used to evaluate the performance metrics through visualizations and CSVs were constructed for numerical analysis.

The performance metrics were Throughput, Average Response Time, Average IO Wait Time, Average Turnaround Time. The performance metric of CPU Utilization was not included for the large scale processing visualization and analysis since the CPU utilization was almost always 100% for all the algorithms due to the amount of processes in the ready queue. On the small scale analysis, CPU utilization was the sole metric compared, as the visualizations for the small scale performance metrics is practically equivalent to that of the large scale.

Through the analysis it was found that the best performing algorithm for the program that was written was SJF because it had the most efficient CPU utilization and also had the fastest processing time.

4 Contributions

We started the project in Java, but we kept running into issues with it having miscellaneous bugs and discontinuities between our system environments. This prompted us to change our implementation to Python. Due to this, some of our contributions may not be visible in the current state of the program, but nevertheless they were a vital part of figuring out how to approach this once we switched platforms.

David: Did bug fixing on the JAVA project. After switching to Python, devised the process import system and developed the process scheduling system. Heavily commented and revised all the code in order for it to be as modular and reliable as possible. Implemented the delta timing used in the simulation and added polish in various places as the program was developed. Helped develop the documentation.

Fahad: Initially developed the back-end of the JAVA program. The program failed and could not fix a lot of problems that were reviewed so switched the program to Python, which the three of us were more familiar with. Created the process.py file and coded the scheduling algorithms for the Python project. Wrote the code to generate log after running the program. Worked on the documentation and performance comparison.

Himanshu: Initially developed the front-end of the JAVA program. The program failed, the outputs were not showing correctly, and GUI was finicky. Switched to Python and worked on making the command line arguments work. Wrote the code for the exgen.py, which creates example files for the program and stores it in the examples directory. Also, worked on the documentation and performance comparison.

5 Conclusion

This project was designed to simulate the short-term scheduler in a multiprocessor operating system. A program was developed to implement a simulator of CPU scheduling with different scheduling algorithms such as FCFS, SJF, PS, and RR. We initially wrote the program in Java but the output was not correct and debugging the program was more of a problem than creating a new program. The program was rewritten in Python.

The results show that the Shortest Job First algorithm is the most efficient algorithm out of the four algorithms when tested on the file ”performancetest”. The reason being that SFJ had the most efficient CPU utilization and also had the fastest processing time.

Overall, this project was a great learning experience to learn how to code in Python, how to code a simulation program, and it allowed us to build a deeper intuition of the strengths and weaknesses of each scheduling algorithm.