CPU Scheduler

Created by: Anupriya Kumari
Enrollment Number: 22116015
Department: Electronics and Communication Engineering
ACM Open Project 2024

Introduction

CPU scheduling is the method used to decide which process will use the CPU for execution while another process is waiting. The primary goal of CPU scheduling is to ensure that the operating system always assigns a process from the ready queue for execution whenever the CPU is idle. This selection is done by the CPU scheduler, which chooses one of the processes in memory that are ready to run.

Objectives of Process Scheduling Algorithm

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Min turnaround time [Time taken by a process to finish execution]
- Min waiting time [Time a process waits in ready queue]
- Min response time [Time when a process produces first response]

Objective

The objective of this project is to understand

- Various scheduling algorithms that are used in operating systems
- Their implementation details
- When to use them
- Comparative analysis using relevant metrics
- Finally, using the knowledge in hand, we wish to create a CPU-Scheduler web application with a backend in C++
- The frontend displays the best algorithm out of all the algorithms that are being implemented.

There exist mainly six CPU scheduling algorithms however there are also some lesser known algorithms that we have explored in this project. In total, we are implementing nine scheduling algorithms. We are comparing them and showing the best one out of all.

Deliverables

We present a complete C++ based backend integrated using Crow and nlohmann/json to a frontend written in HTML, Vanilla JS and CSS along with Google charts and Charts.js for visualizations and comparative analysis.
The UI is simple and minimalist containing only necessary text and more effort has been put in the actual backend to frontend integration and the visualizations and metrics.
The functionalities of the project are detailed in later sections.

Implementation Details

Approach and Idea

The basic initial approach was to study, understand and implement all the cpu-scheduling algorithms. The six most important scheduling algorithms that we had planned to code first were: First Come First Serve (FCFS), Shortest-Job-First (SJF) Scheduling, Shortest Remaining Time, Priority Scheduling, Round Robin Scheduling and Multilevel Queue Scheduling. After heavy research into cpu scheduling algorithms, several different websites explained all the existing algorithms post which, we decided to add more algorithms to the list and make a more advanced CPU-Scheduler. Algorithms like longest job first, longest remaining job first, highest response ratio next are modified versions of the corresponding well-known algorithms (shortest job first etc).

The following algorithms are implemented in this project:

- First Come First Serve (FCFS) Non-Preemptive
  - Processes are executed in the order they arrive in the ready queue.

- Shortest Job First (SJF) Non-Preemptive
  - The shortest job in the ready queue is executed first.

- Shortest Remaining Job First (SRJF) Preemptive
  - The job with the shortest remaining time is executed first.

- Longest Job First (LJF) Non-Preemptive
  - The longest job in the ready queue is executed first.

- Longest Remaining Job First (LRJF) Preemptive
  - The job with the longest remaining time is executed first.

- Priority Non-Preemptive (PNP) Non-Preemptive
  - The highest priority job in the ready queue is executed first.

- Priority Preemptive (PP) Preemptive
  - The highest priority job preempts the running process.

- Round Robin (RR) Preemptive
  - Processes are given a fixed time quantum and cycled through.

- Highest Response Ratio Next (HRRN) Non-Preemptive
  - The job with the highest response ratio is executed first.

Choosing the best algorithm:

A ranking algorithm:

One of the most optimal techniques to decide which algorithm would be the best out of a given array of cpu-scheduling algorithms is by ranking them based on key performance metrics: average waiting time, turnaround time, response time, completion time, and context switches (lower the better). We first calculates the rank of each algorithm for these metrics by sorting and assigning ranks accordingly. Then we compute CPU utilization and throughput for each algorithm (rank by highest first). The overall rank of each algorithm is determined by averaging the ranks across all metrics. The algorithm with the lowest average rank is identified as the best. This way we pay equal amount of weightage to each metric as each is important in its own place and find the best suited algorithm for the given set of processes. If higher weightage needs to be given to some metrics, a weighted average approach can be adopted. This involves assigning weights to each metric based on their importance and then calculating a weighted average rank. However, to keep our implementation universal and straightforward, we can safely take an average.

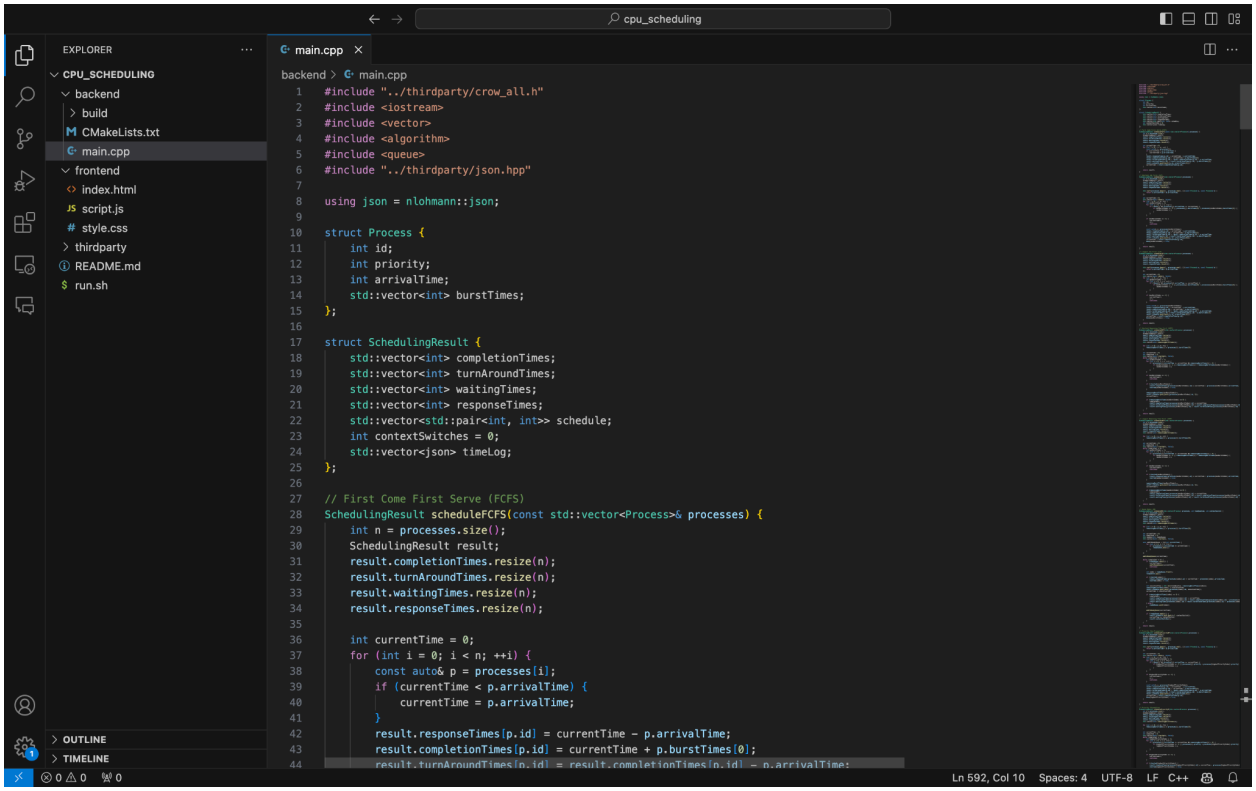
This also implies there is no way one to decide which is the best algorithm without analysing each one individually first.

C++ Backend

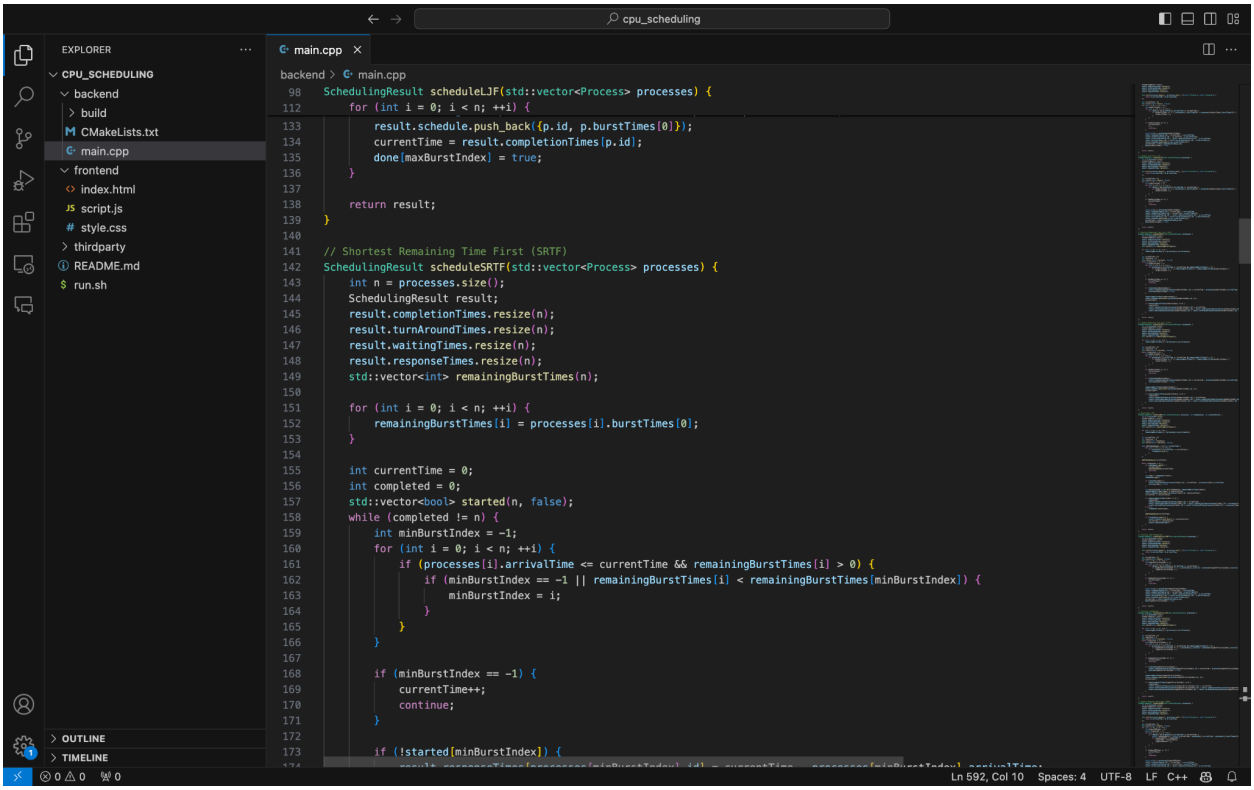
In this section, we explain how the C++ backend was written. This was not a simple task because we needed to create a web application. This is typically done by creating an API endpoint in a web application, where the endpoint responds with structured data in JSON format. Thus, apart from the logic behind each scheduling algorithm, we had to create a route in the Crow

application (a C++ microweb framework, that is used to handle HTTP requests and responses) that listens for HTTP POST requests. We then set up an endpoint in a Crow web application that, when triggered, constructs a JSON response containing various performance metrics and outputs related to our scheduling process and returns this JSON response to the client. This was the most difficult part of this project.

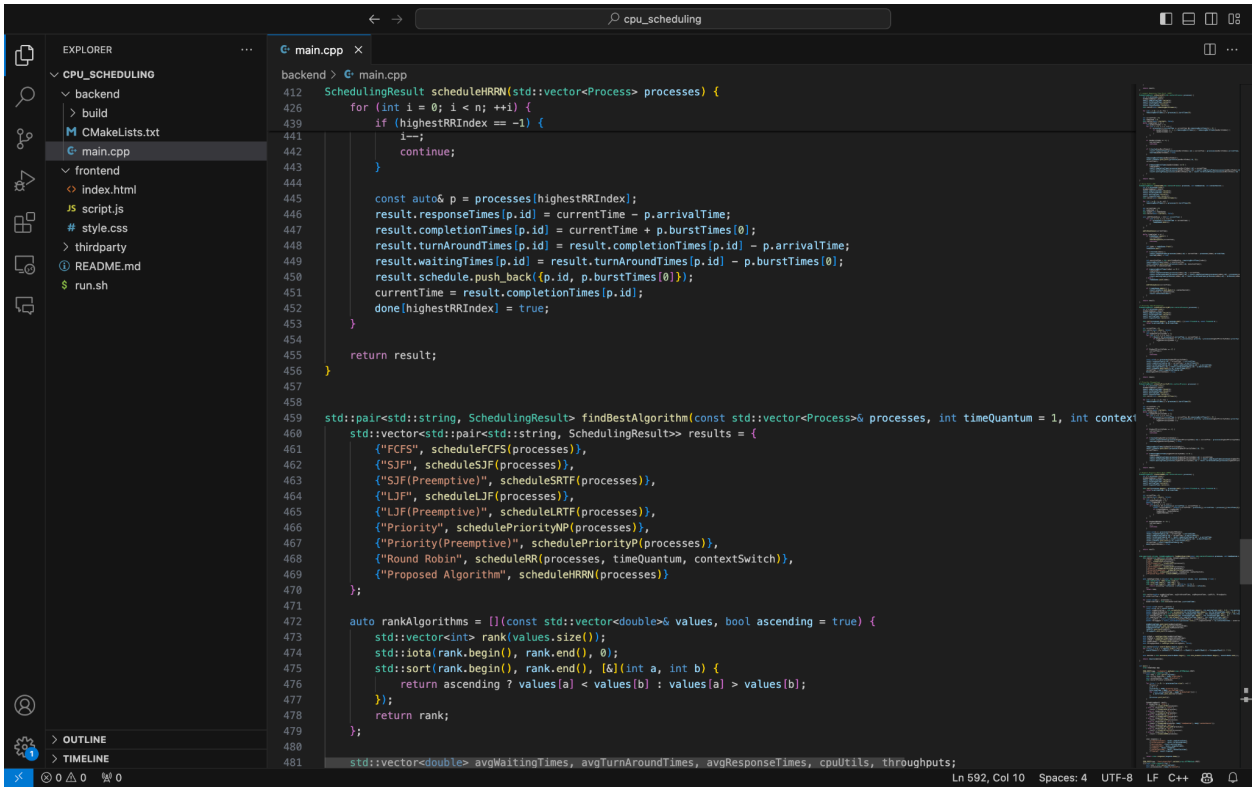
The screenshots below show the code snippets with the scheduling algorithm logic with the explanation to what they are doing and why:

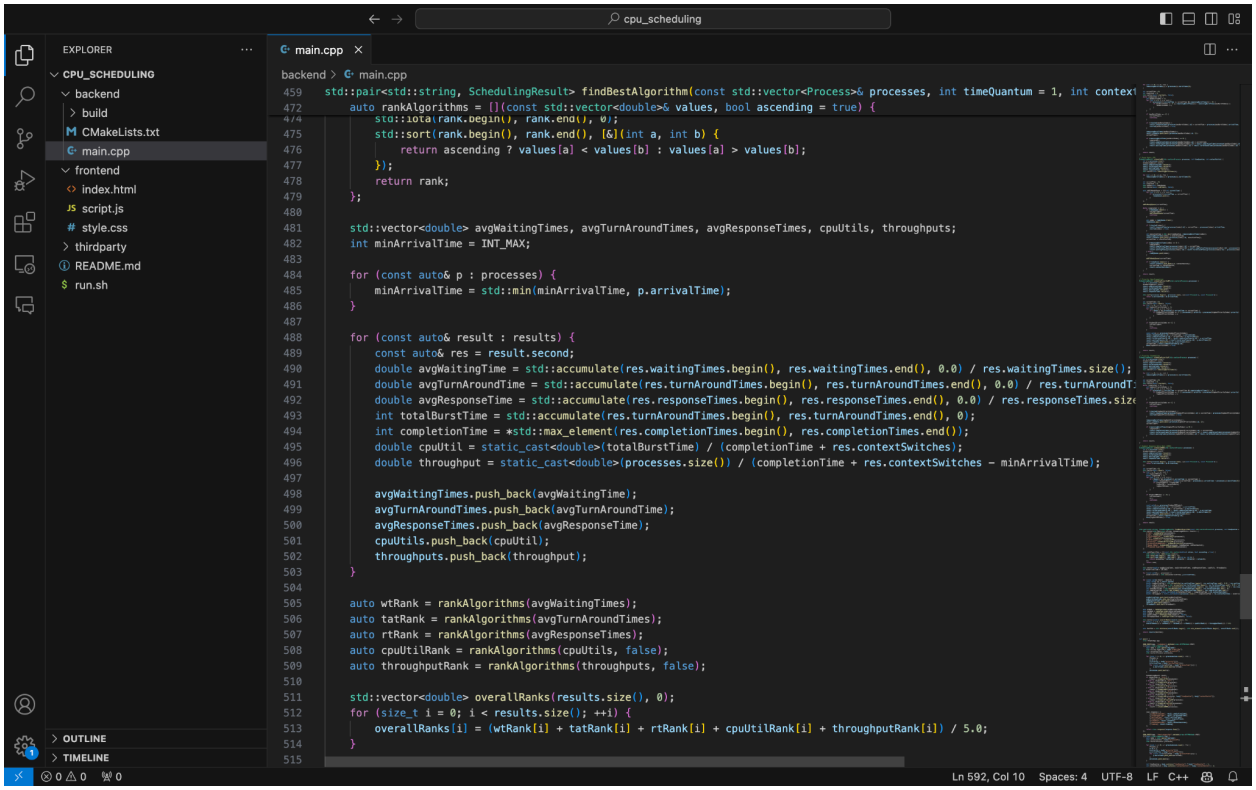


Above, we observe the basic structure of our code. We are using the Crow web framework, we must include all dependencies. We define each of our algorithms in the backend. The logic and implementation for those are studied from the references listed at the end of the documentation. The execution of each scheduling algorithm is important.

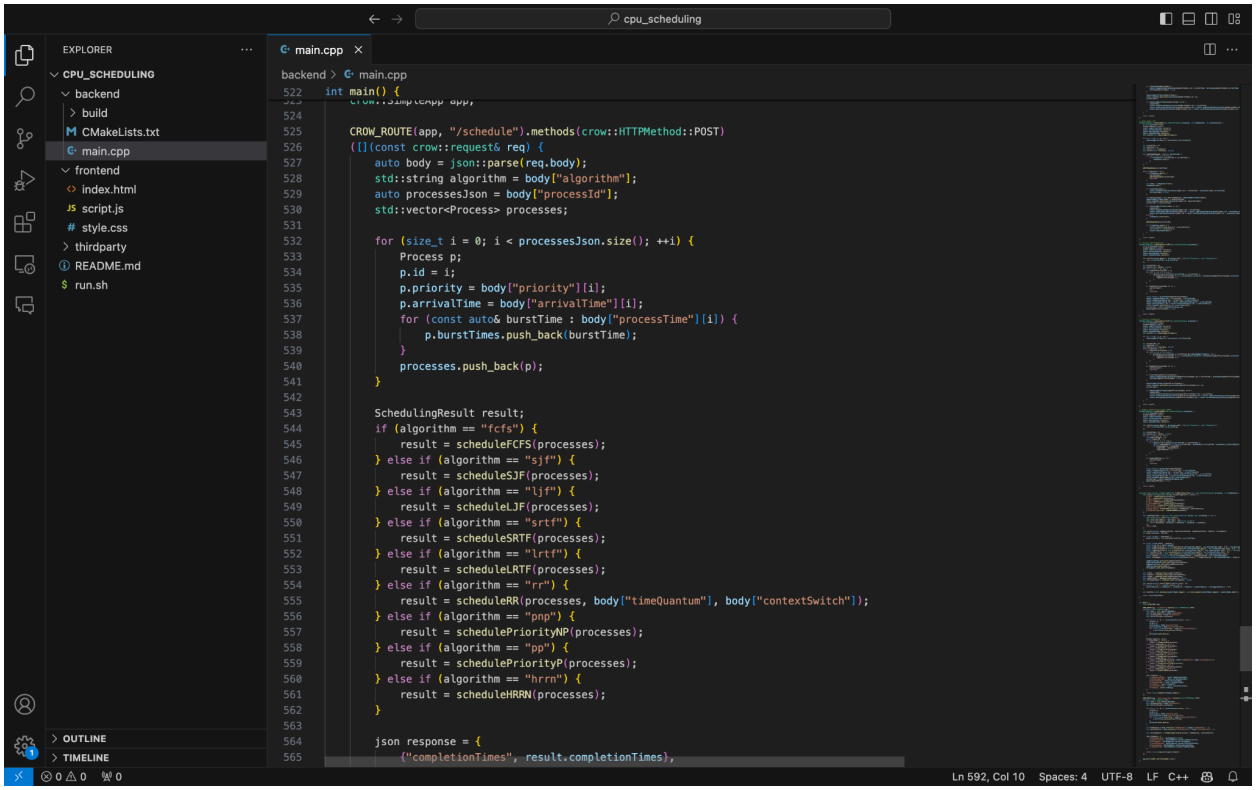

cpu_scheduling
EXPLORER
CPU_SCHEDULING
backend
build
CMakeLists.txt
main.cpp
frontend
index.html
script.js
style.css
thirdparty
README.md
run.sh
OUTLINE
TIMELINE
main.cpp
backend > main.cpp
SchedulingResult scheduleLJF(std::vector<Process> processes) {
for (int i = 0; i < n; ++i) {
result.schedule.push_back({p.id, p.burstTimes[0]});
currentTime = result.completionTimes[p.id];
done[maxBurstIndex] = true;
}
return result;
}
// Shortest Remaining Time First (SRTF)
SchedulingResult scheduleSRTF(std::vector<Process> processes) {
int n = processes.size();
SchedulingResult result;
result.completionTimes.resize(n);
result.turnAroundTimes.resize(n);
result.waitingTimes.resize(n);
result.responseTimes.resize(n);
std::vector<int> remainingBurstTimes(n);
for (int i = 0; i < n; ++i) {
remainingBurstTimes[i] = processes[i].burstTimes[0];
}
int currentTime = 0;
int completed = 0;
std::vector<bool> started(n, false);
while (completed != n) {
int minBurstIndex = -1;
for (int i = 0; i < n; ++i) {
if (processes[i].arrivalTime <= currentTime && remainingBurstTimes[i] > 0) {
if (minBurstIndex == -1 || remainingBurstTimes[i] < remainingBurstTimes[minBurstIndex]) {
minBurstIndex = i;
}
}
}
if (minBurstIndex == -1) {
currentTime++;
continue;
}
if (!started[minBurstIndex]) {
Ln 592, Col 10
Spaces: 4
UTF-8
LF
C++

In the findBestAlgorithm, we are storing pairs of algorithm names and their corresponding scheduling results obtained by running different scheduling functions. We then use a lambda function to rank algorithms based on their performance metrics. It sorts the indices of the values in ascending or descending order.

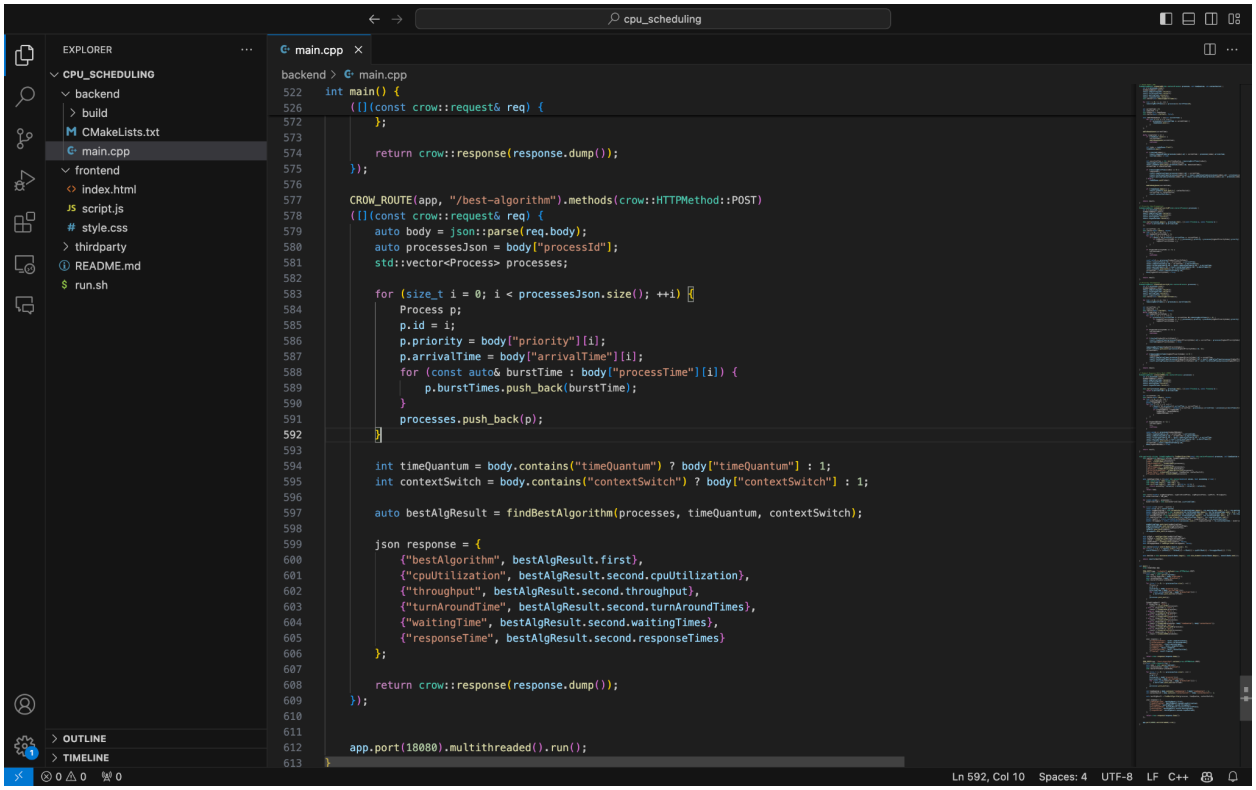


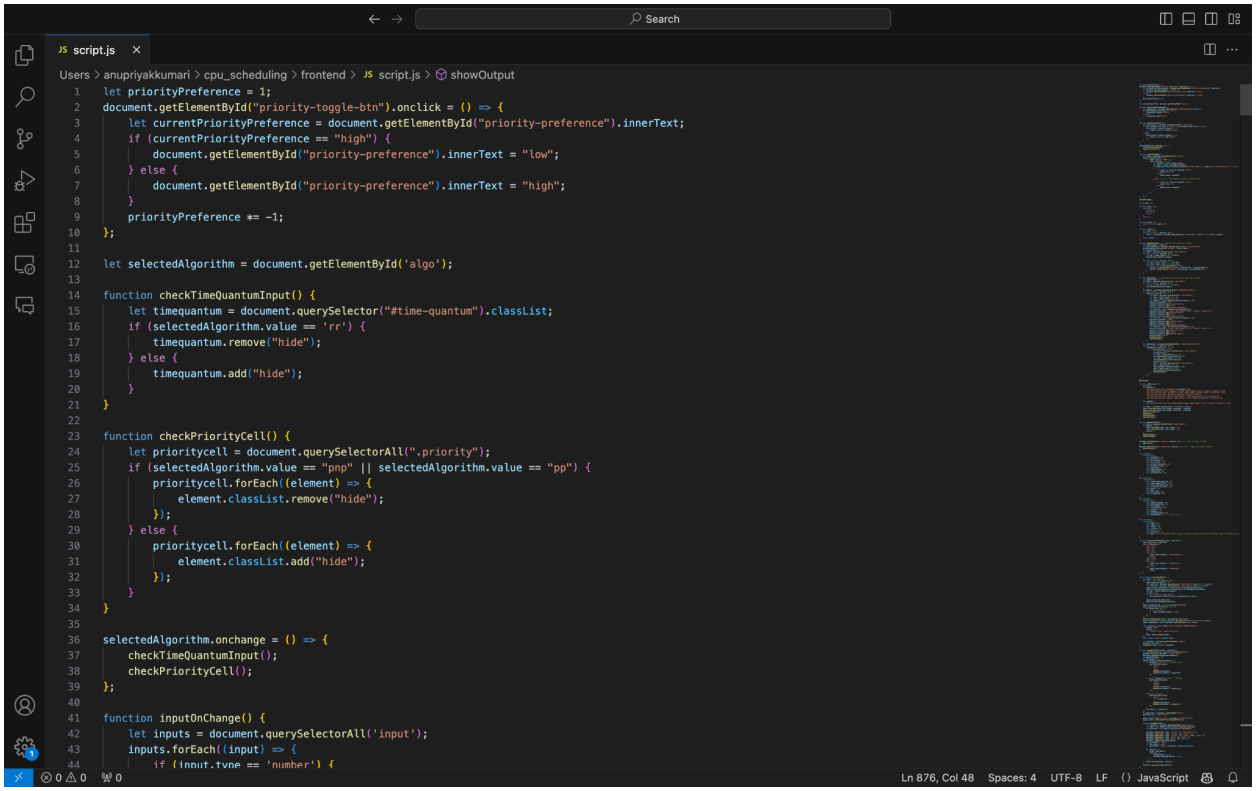


Then we use vectors store the average waiting times, turnaround times, response times, CPU utilization, and throughput for each algorithm and initialize a minArrivalTime to the maximum integer value to find the minimum arrival time of all processes.
We use a loop calculates the average waiting time, turnaround time, response time, CPU utilization, and throughput for each algorithm and stores them in their respective vectors.
Algorithms are ranked for each metric. CPU utilization and throughput are ranked in descending order (better performance has higher values).
The overall rank for each algorithm is calculated by averaging the ranks of all metrics.



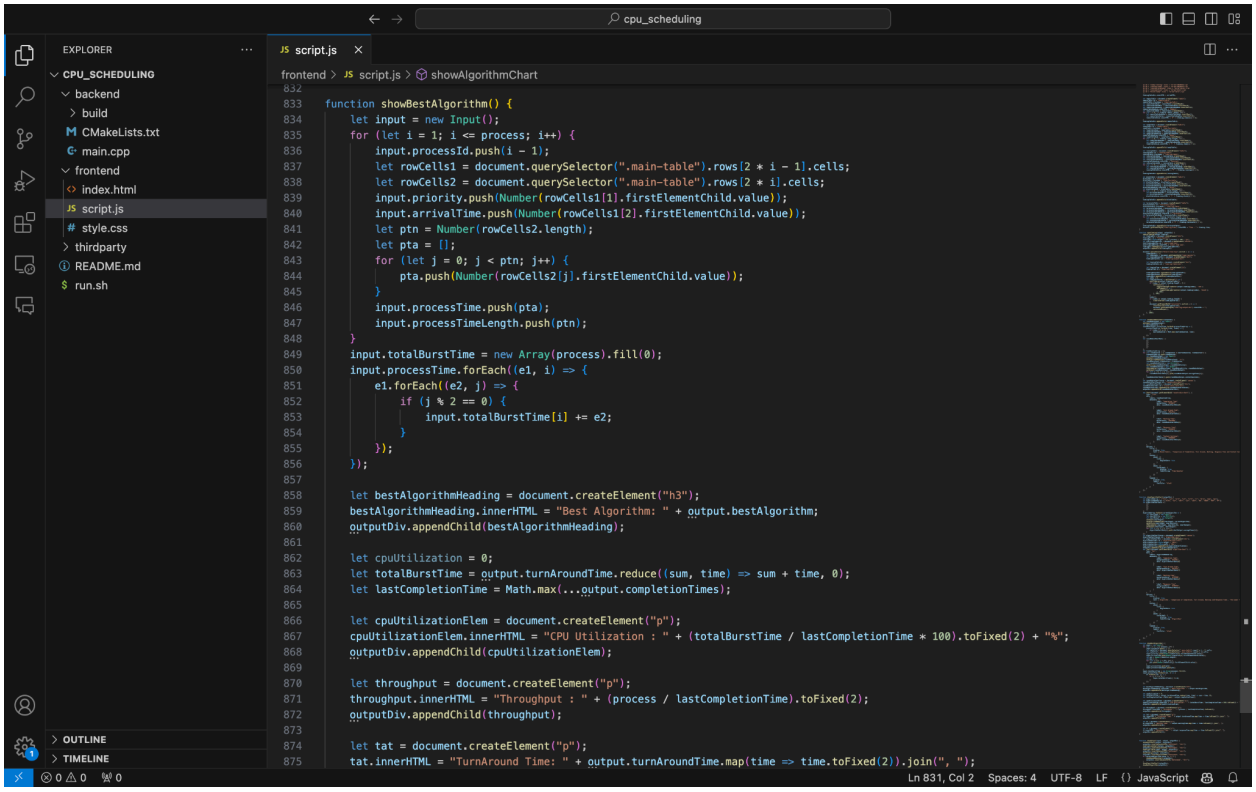
Finally, we create a route that handles POST requests for scheduling processes using the specified algorithm. We set up a Crow-based RESTful endpoint to determine the best CPU scheduling algorithm based on the input processes. It processes the request data, runs various scheduling algorithms, evaluates their performance, and responds with the results in JSON format.

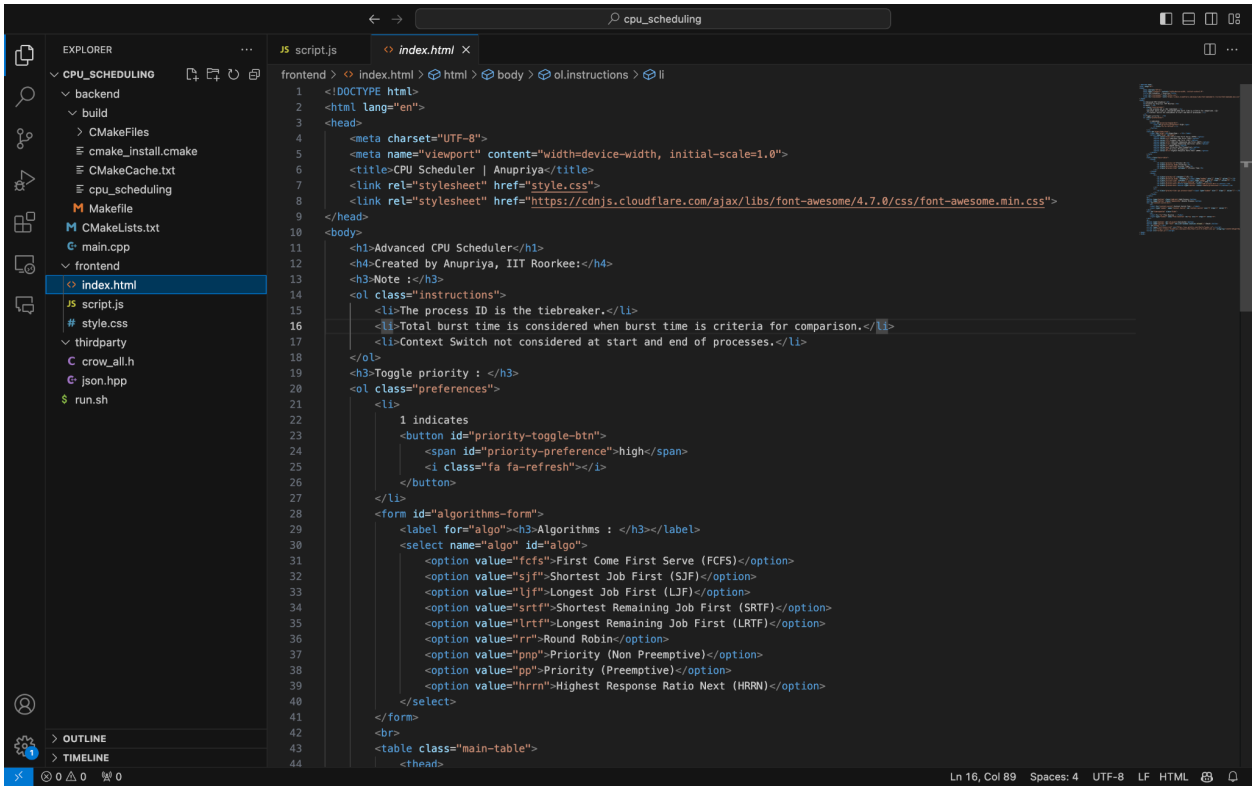


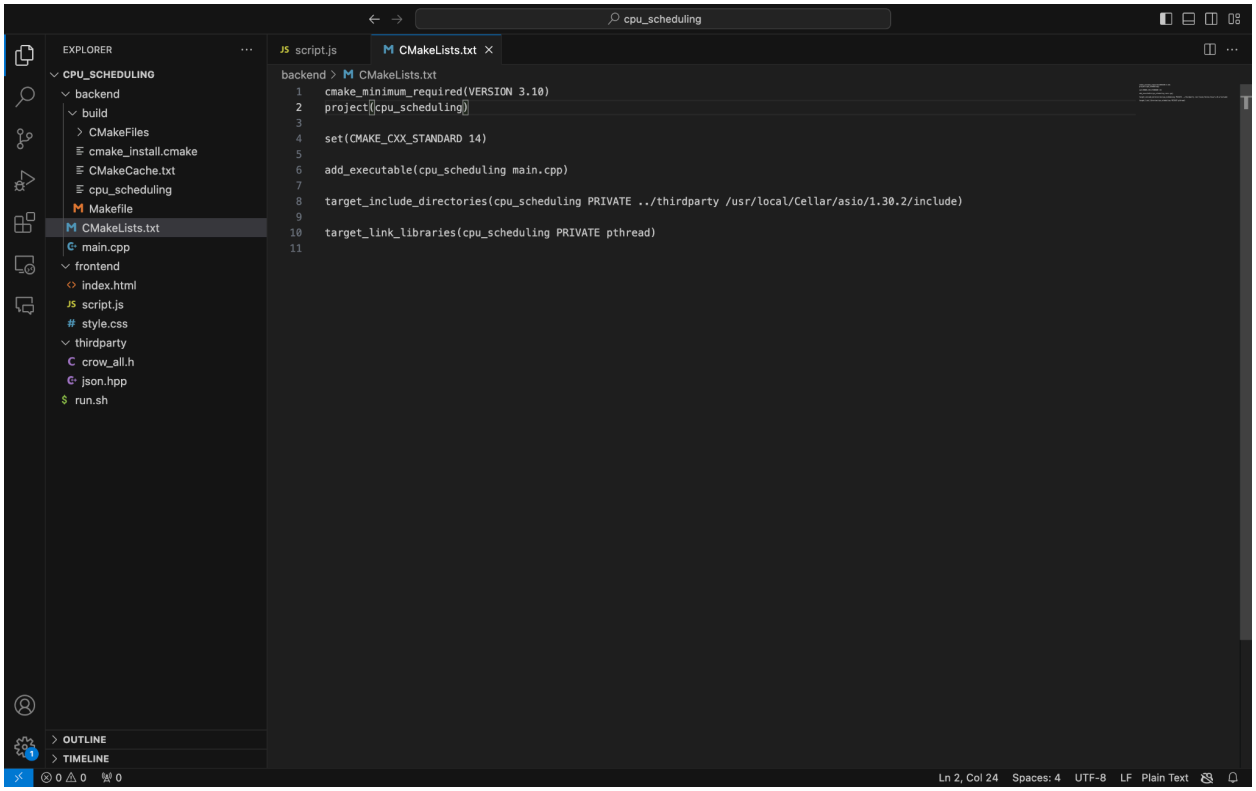


The screenshot above shows the script.js that receives the response from our C++ backend via Crow in a json format, which is parses and shows throught various functions. It integrates with an HTML interface to allow users to input process details, select scheduling algorithms, and

visualize the results. Overall, the script.js provides an interactive interface for users to configure CPU scheduling simulations. It dynamically manages the process table, validates inputs, sends scheduling requests to a server, and visualizes the results through various charts and tables. The script leverages the Crow web framework for backend communication and Google Charts for visualization.







This CMake file is used to configure the build system for our C++ project. It requires CMake version 3.10 or higher, sets the project name, specifies the use of the C++14 standard, defines an executable target, includes additional directories for header files, and links the pthread library.

Integration of backend with frontend using Crow and JSON

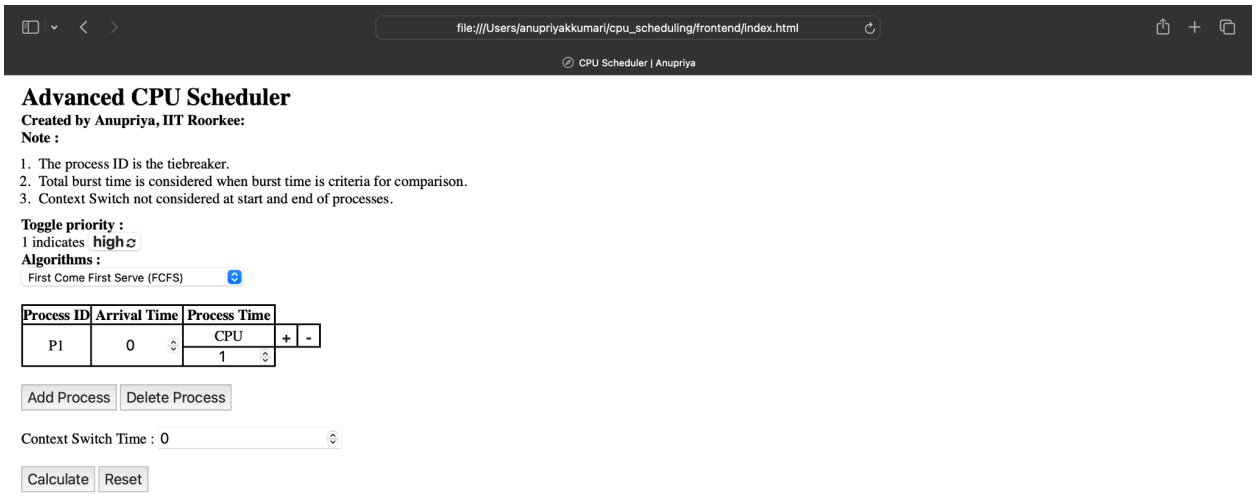
After creating the above files, we build the project using cmake.
We need the crow_all.h file and the json.hpp file. To build the crow_all.h file, we followed the official Crow documentation We also need to install Asio. We used Homebrew for installation. Ensuring that the installed dependencies are in the same directory as our project is important.

```
anupriyakkumari@anupriyakkumari-C02GG1JJML7H ~ % cd cpu_scheduling
anupriyakkumari@anupriyakkumari-C02GG1JJML7H cpu_scheduling % cd backend
anupriyakkumari@anupriyakkumari-C02GG1JJML7H backend % mkdir build
anupriyakkumari@anupriyakkumari-C02GG1JJML7H backend % cd build
anupriyakkumari@anupriyakkumari-C02GG1JJML7H build % cmake ..
-- The C compiler identification is AppleClang 15.0.0.15000309
-- The CXX compiler identification is AppleClang 15.0.0.15000309
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (1.4s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/anupriyakkumari/cpu_scheduling/backend/build
anupriyakkumari@anupriyakkumari-C02GG1JJML7H build % make
[ 50%] Building CXX object CMakeFiles/cpu_scheduling.dir/main.cpp.o
[100%] Linking CXX executable cpu_scheduling
[100%] Built target cpu_scheduling
anupriyakkumari@anupriyakkumari-C02GG1JJML7H build % ./cpu_scheduling
(2024-06-19 17:01:10) [INFO    ] Crow/master server is running at http://0.0.0.0:18080 using 8 threads
(2024-06-19 17:01:10) [INFO    ] Call `app.loglevel(crow::LogLevel::Warning)` to hide Info level logs.
```

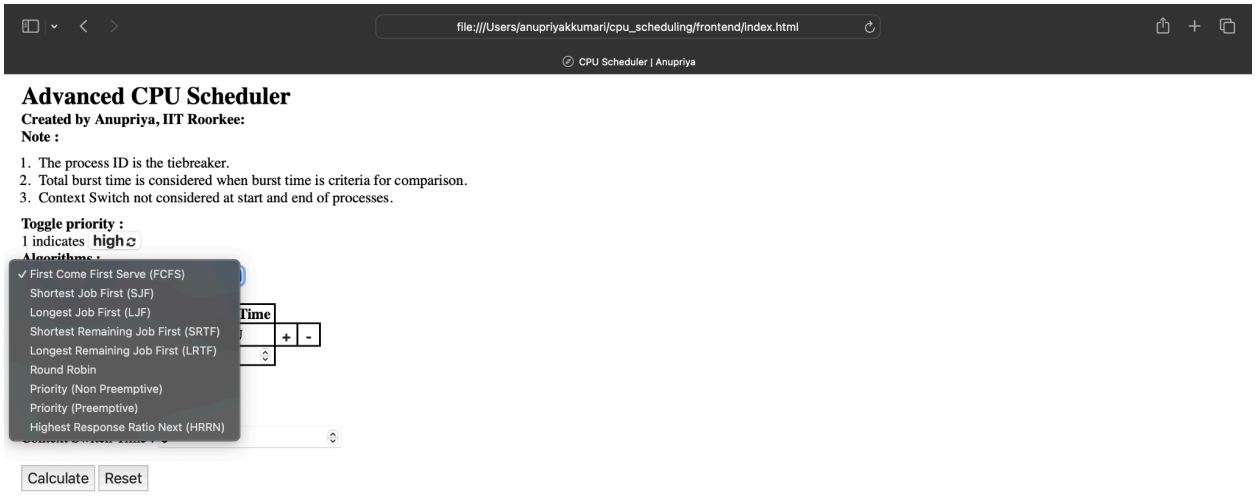
Working demo

Please note that while the user is free to choose any scheduling algorithm from the given list, we still present a comprehensive analysis and comparison between every algorithm, looking at which will tell us which is the most optimal algorithm to use based on the comparison criteria. We show the completion, turnaround, waiting, and response time for all the algorithms. The lower these are the better the algorithm. So, based on the metric of choice, one can unselect other metrics and focus only on that algorithm with the minimum value of the chosen metric.

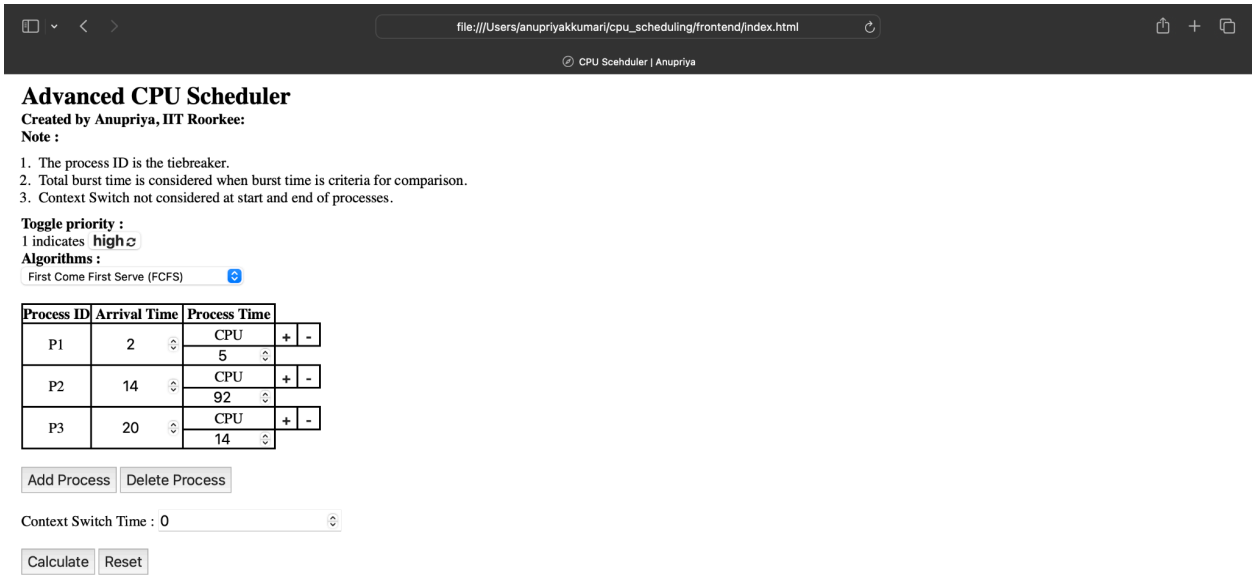
Basic first page:



Select any algorithm of your choice:

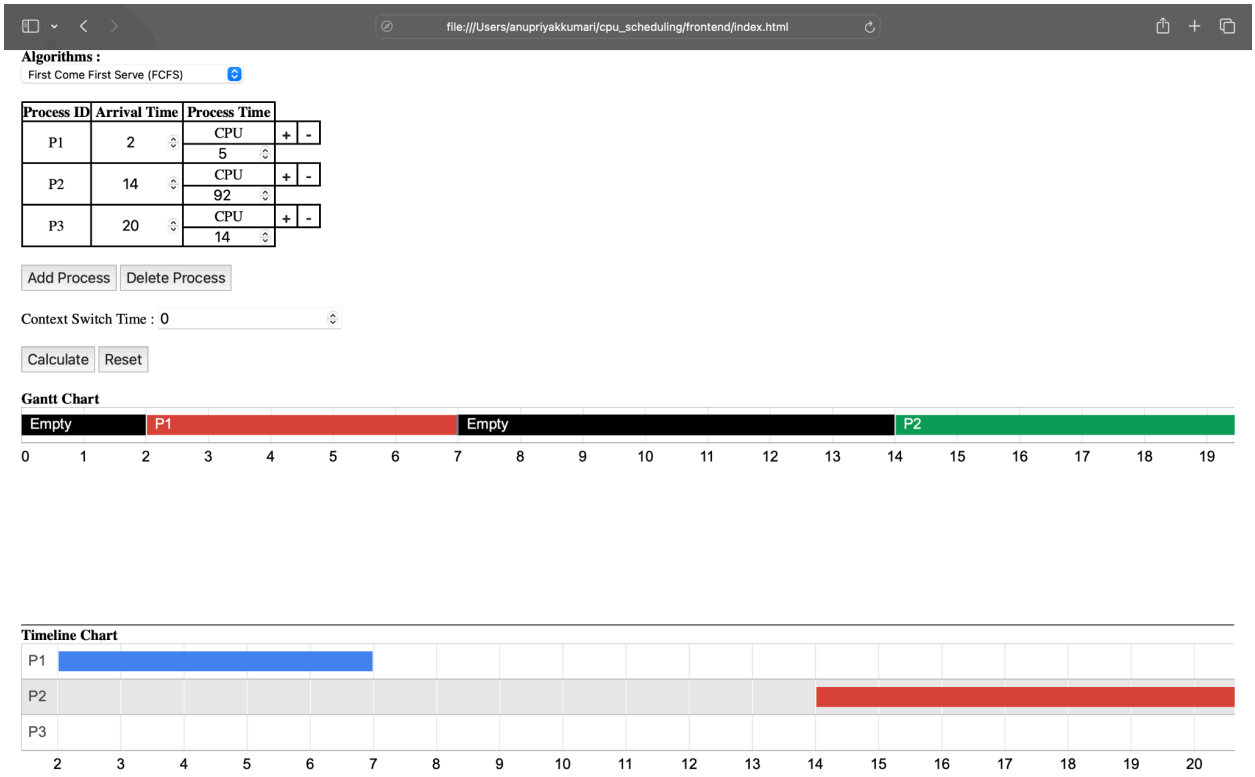


Set the priority preference (1 indicated high/low, sets a priority for process ID)
Add/Delete processes
Set the Process Time in CPU (use the +/- button to add I/O)
Set a context switch time and click on Calculate:

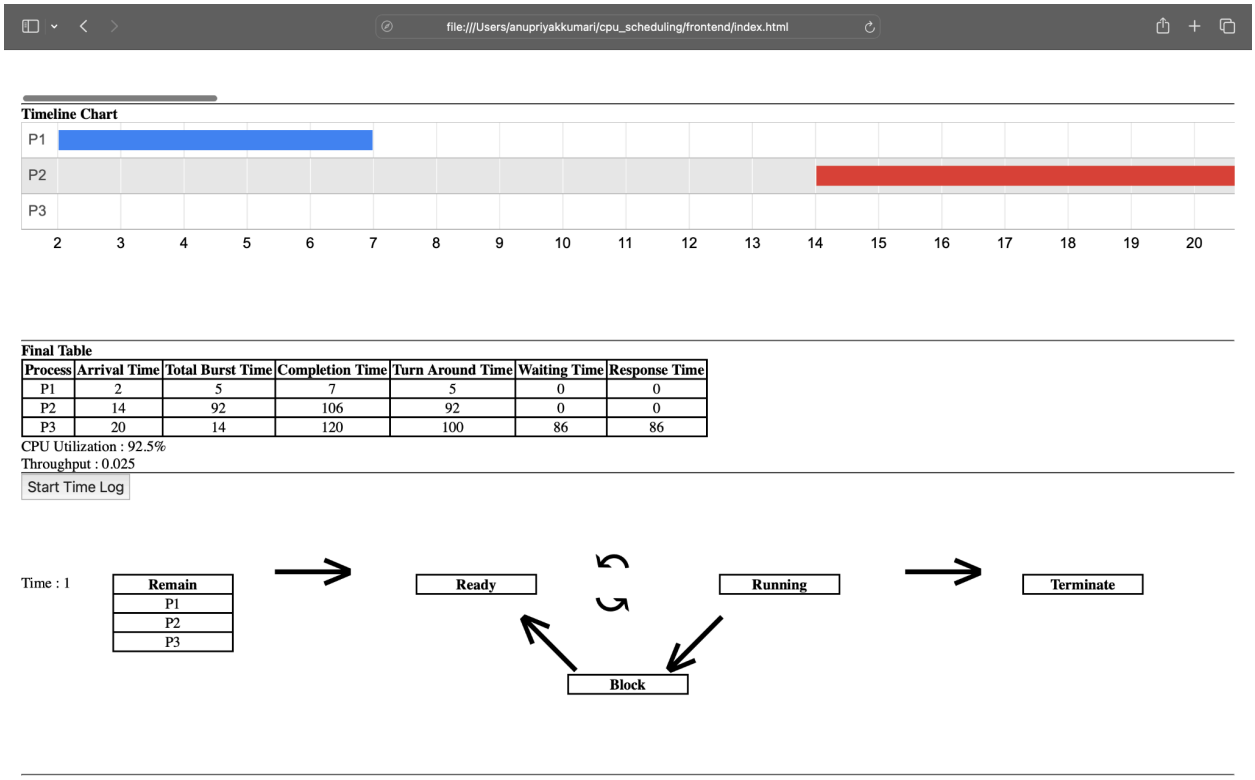


Visualizations and metrics for the selected algorithm:

The following screenshots show the visualization and metrics that are displayed as we scroll down. These are helpful for studying the algorithm currently in use as well as a comprehensive comparison between all the algorithms for a given task:



The above screenshot shows the Gantt chart and a timeline chart:

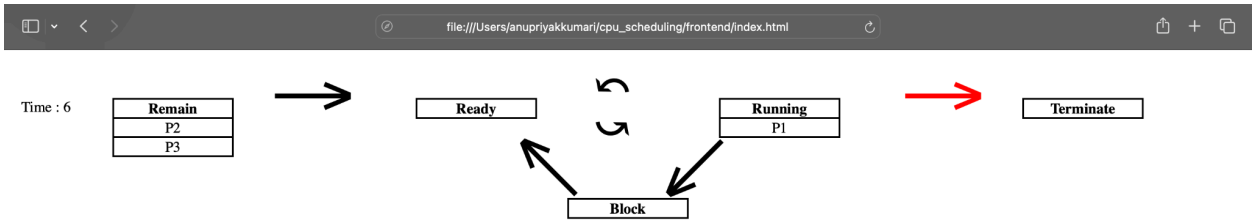


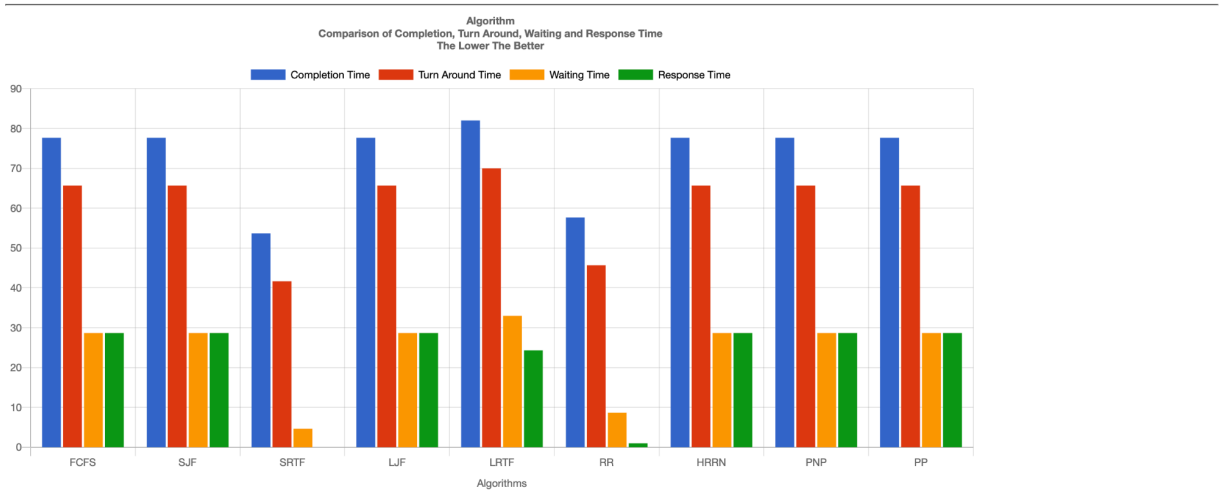
As we scroll down, we see the final table that completely describes the processes - completion, turn around, waiting and response time. The lower these are, the better is the algorithm.
Just below the table we obtain the CPU Utilization and the throughput. The higher these are the better is the algorithm.

We also see a time log here which shows a running time tab and how the processes are being executed, which ones are remaining, which are ready, running, blocked and terminated. This gives a close depiction of what is actually happening in the OS.

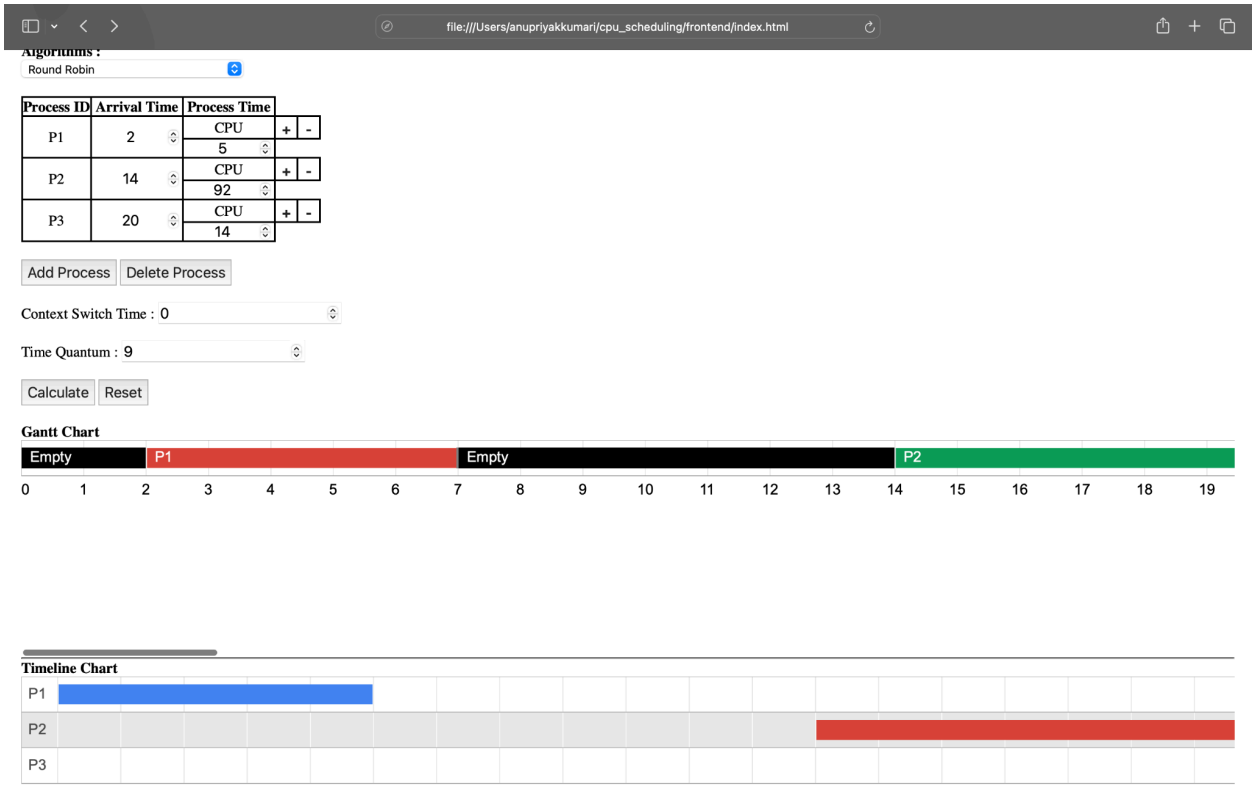
Comparison between all the algorithms:

To choose the best algorithm, we perform a comprehensive comparison analysis and look at the composite bar chart showing the completion, turn around, waiting and response time for each algorithm:

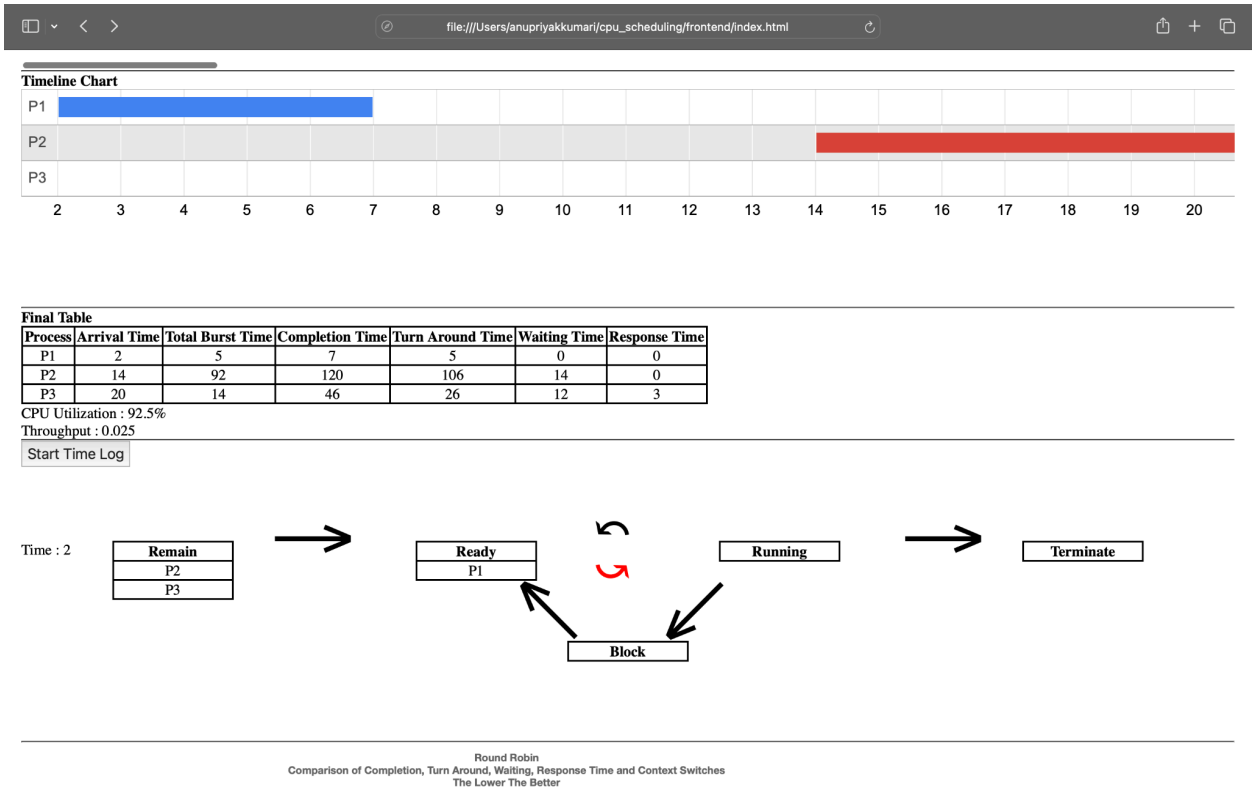




We can unselect a certain metric used for comparison and focus only on one metric. The lower the time value, the better the algorithm. We would also observe that for the most optimal algorithm, the CPU Utilization and throughput would be maximum.
This whole demo was for FCFS (First come first serve) algorithm.
Let us see one more example for Round Robin, where we also input a time quantum value.

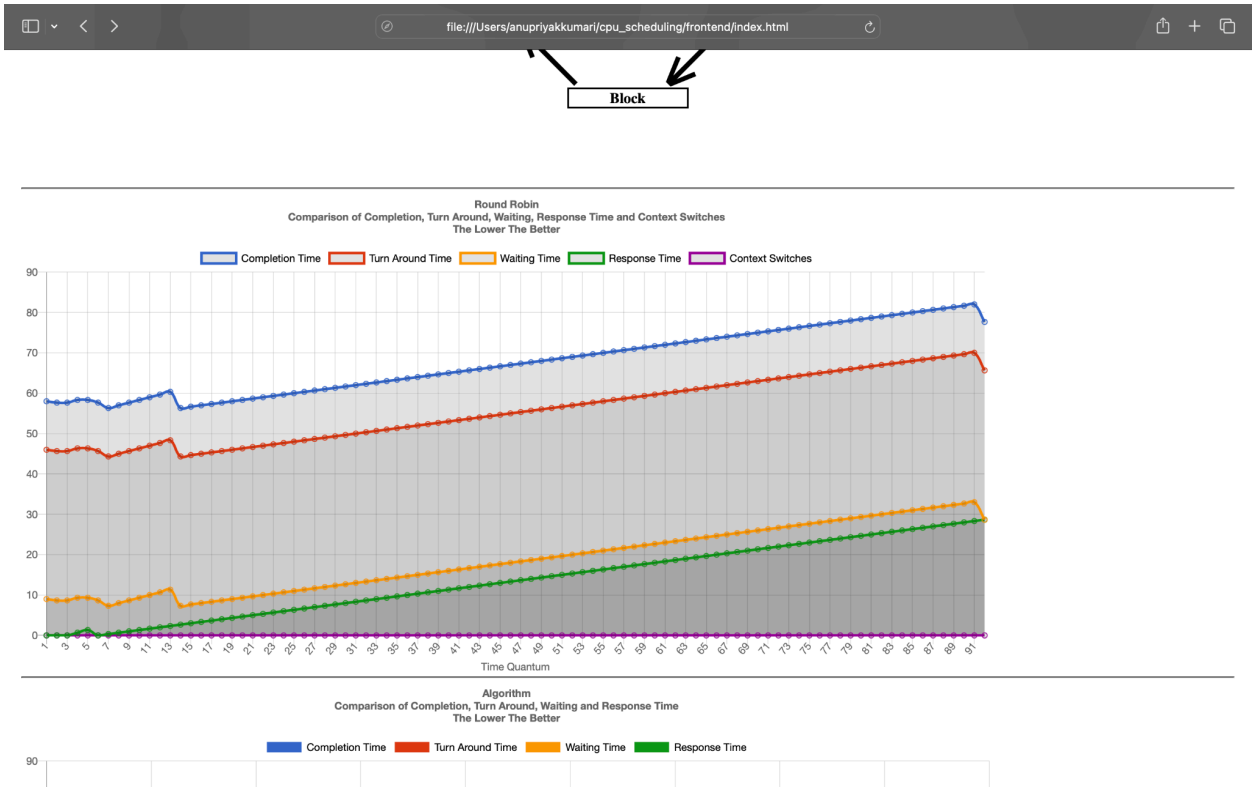


Here we observe an extended Gantt chart and timeline chart that we would have to scroll through to fully see.

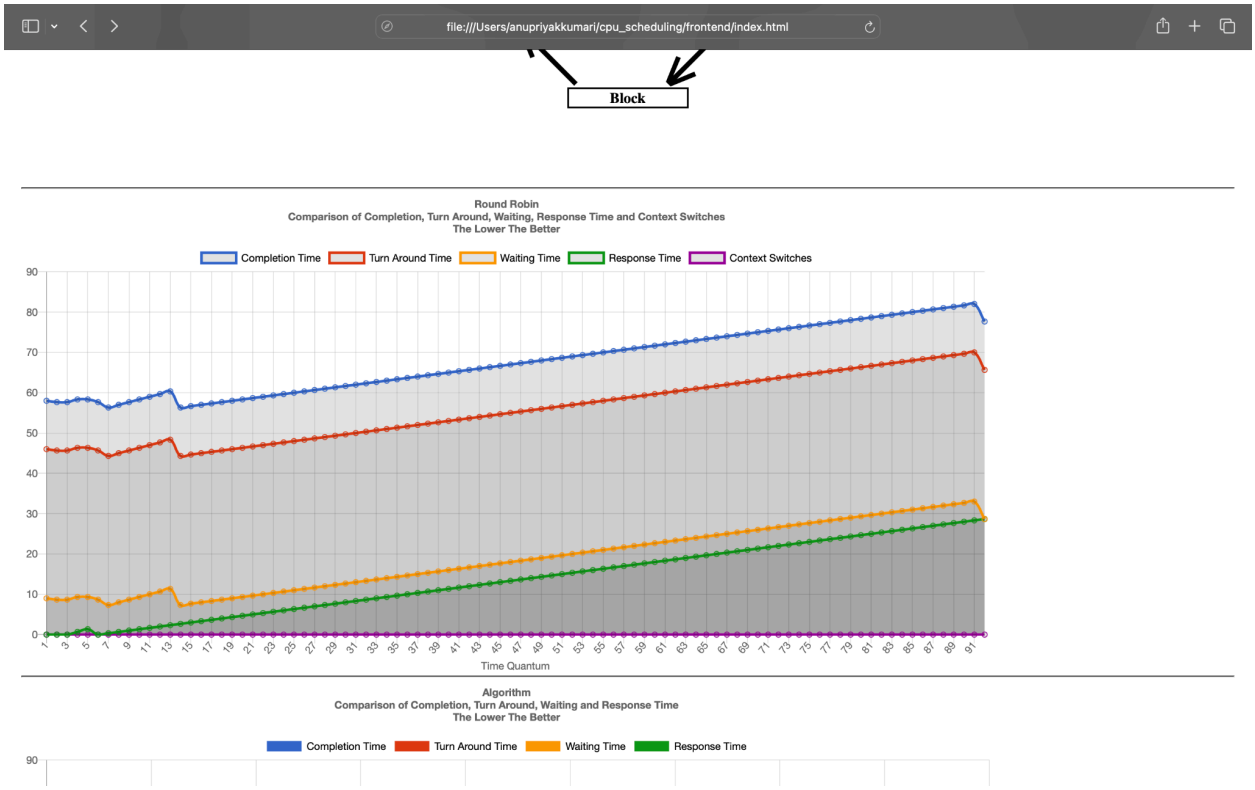


As before, we show a final table, CPU utilization and throughput metrics, a time log explaining the process.

Below we show a very helpful continuous line graph that shows all the time metrics and context switches and how RR performance varies with time quantum.

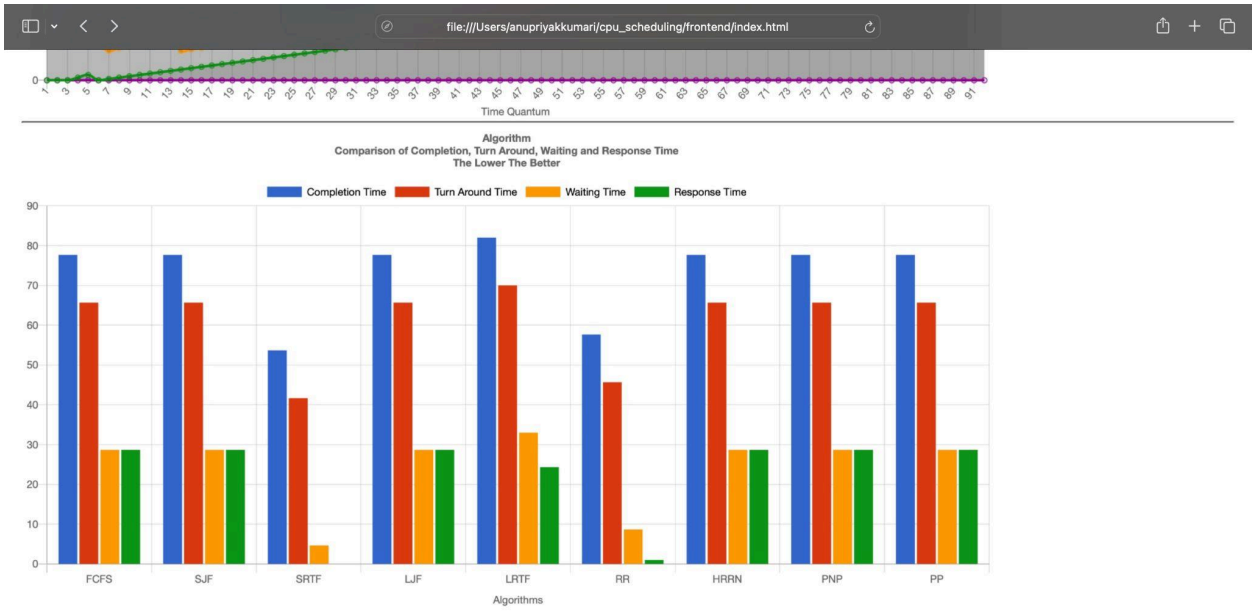


Finally, we show the composite bar chart which visually compares all the algorithms for the defined processes:



The best algorithm:

Finally as we scroll down further, we see the best algorithm out of all nine with the stats. For both our examples, we only changed our choice of the algorithm. Our scheduler suggests the following best algorithm for our process.



Best Algorithm out of all: Shortest Job First (SJF)

1. CPU Utilization : 96.00 %
2. Throughput : 0.024
3. TurnAround Time: 41.67
4. Waiting Time: 4.67
5. Response Time: 0.00

We can see the best algorithm out of all based on our ranking algorithm is SJF for this case. For a more complicated case, we may observe a different best algorithm.

Features and additional visualizations

Original target

Originally, we had tried implementing six main algorithms with the option to select the process, add arrival time, add cpu and i/o process time, context switch time and time quantum (for round robin). This was implemented on the same frontend as we can see right now and was showing only the gantt chart, timeline chart, the final table and the cpu utilization, throughput and number of context switches.

Current visualizations and features

In the later half of the project, we decided to include other algorithms as well and add more metrics such as the time log, the algorithm comparison chart and the round robin time versus time quantum graph. As part of additional features, we have accomplished the following:

1. Nine scheduling algorithms

2. Users choice to specify every detail of the process that they want - arrival time, process time, priority of process, algorithm choice, context switch time, time quantum.
3. A complete gantt chart, timeline chart.
4. The final table showing the total burst time, completion, waiting, turnaround and response time for each process.
5. The CPU Utilization, throughput and number of context switches.
6. Animated time log showing the working of the algorithm step by step for each second.
7. An interactive time versus time quantum chart for round robin
8. An interactive composite bar chart displaying the completion, waiting, turnaround and response time for each algorithm for a comprehensive comparative analysis.

Summary and scope of improvement

An additional improvement in this project includes making the overall code more configurable. Apart from this the front end is currently simple and only shows the complete metrics for the selected and the best algorithm. In future iterations of the project we wish to improve the metric visualization.

References

The following books and websites played an important role in building a thorough understanding of the CPU scheduling algorithms used in this project. They were also necessary for learning about Crow and nlohmann/json and how to build a web application using these:

- ➔ https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/
- ➔ https://www.scaler.com/topics/operating-system/cpu-scheduling/
- ➔ https://ravipatel1309.github.io/CPUScheduler/docs.html
- ➔ https://www.youtube.com/playlist?list=PLBlnK6fEyqRitWSE_AyyySWfhRgyA-rHk
- ➔ https://www.doc-developpement-durable.org/file/Projets-informatiques/cours-&-manuels-informatiques/Linux/Linux%20Kernel%20Development,%203rd%20Edition.pdf
- ➔ https://www.researchgate.net/publication/49619229_An_Improved_Round_Robin_Scheduling_Algorithm_for_CPU_Scheduling
- ➔ https://crowcpp.org/master/getting_started/setup/linux/
- ➔ https://crowcpp.org/master/getting_started/your_first_application/
- ➔ https://json.nlohmann.me/integration/
- ➔ https://formulae.brew.sh/formula/nlohmann-json

- ➔ https://medium.com/@mdzaki2611/title-building-a-simple-website-with-c-a-quick-guide-using-the-crow-framework-ea788e13dd2c

The simplistic design and implementation of the front end used in this project was inspired by an existing project. However, please note that the backend was written from scratch in C++ as this was a tricky integration.
The following github repositories proved useful in building a thorough understanding of the code structure and the implementation of the scheduling algorithms along with the integration :

- ➔ https://github.com/Ravipatel1309/CPUScheduler
- ➔ https://github.com/CrowCpp/Crow
- ➔ https://github.com/nlohmann/json
- ➔ https://github.com/yousefkotp/CPU-Scheduling-Algorithms
- ➔ https://github.com/pyxploiter/CPU-Scheduling-Algorithms
- ➔ https://github.com/anmol-tripathi/CPU-Scheduling-Algorithms
- ➔ https://github.com/M-aboelsafa/Multiprocessor-scheduling-algorithms

Finally, StackExchange, ChatGPT, Copilot helped debug the code and provide useful insights and suggestions to improve the code structure and make it as readable as possible.