

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:

```

EXPLORER          ...   cpu_scheduling
CPU_SCHEDULING
  backend > main.cpp
    #include "../thirdparty/crow_all.h"
    #include <iostream>
    #include <vector>
    #include <algorithm>
    #include <queue>
    #include "../thirdparty/json.hpp"
    using json = nlohmann::json;
    struct Process {
        int id;
        int priority;
        int arrivalTime;
        std::vector<int> burstTimes;
    };
    struct SchedulingResult {
        std::vector<int> completionTimes;
        std::vector<int> turnAroundTimes;
        std::vector<int> waitingTimes;
        std::vector<int> responseTimes;
        std::vector<std::pair<int, int>> schedule;
        int contextSwitches = 0;
        std::vector<json> timeLog;
    };
    // First Come First Serve (FCFS)
    SchedulingResult scheduleFCFS(const 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);

        int currentTime = 0;
        for (int i = 0; i < n; ++i) {
            const auto& p = processes[i];
            if (currentTime < p.arrivalTime) {
                currentTime = p.arrivalTime;
            }
            result.responseTimes[i.id] = currentTime - p.arrivalTime;
            result.completionTimes[i.id] = currentTime + p.burstTimes[0];
            result.turnAroundTimes[i.id] = result.completionTimes[i.id] - p.arrivalTime;
        }
    }

```

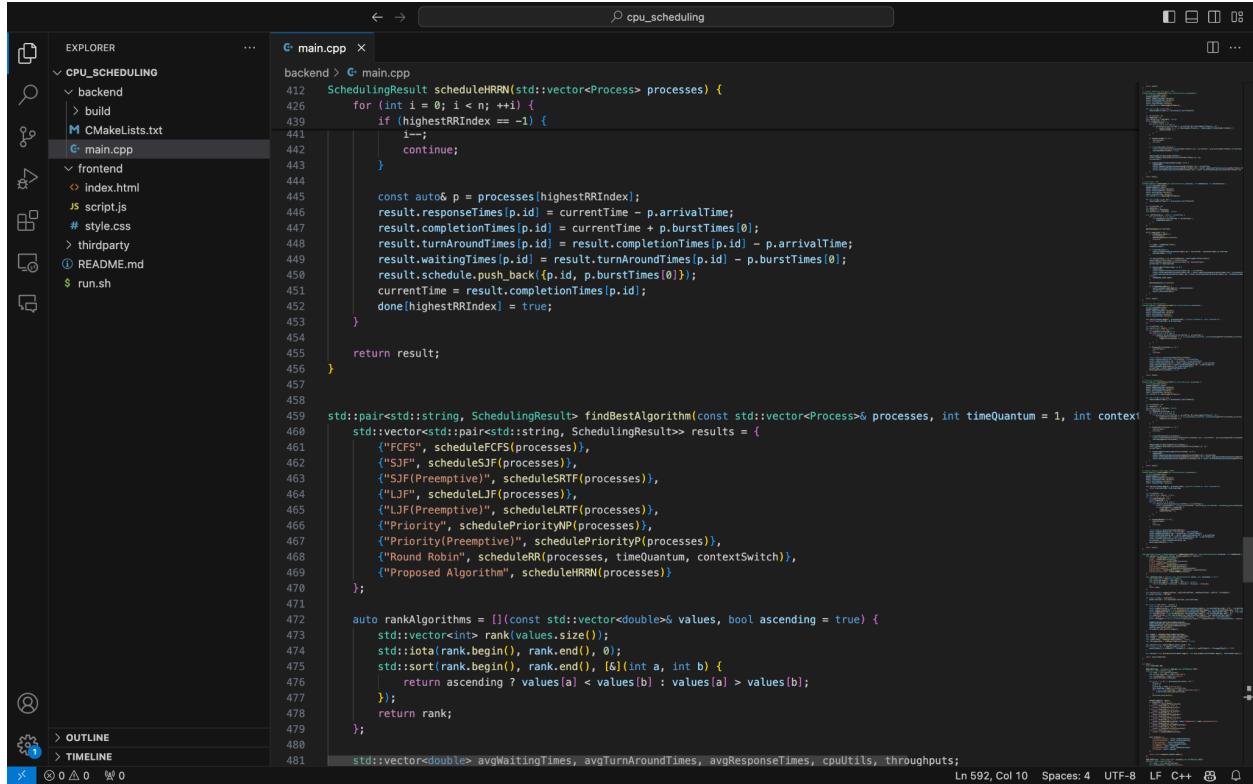
Above, we observe the basic structure of our code. We are using the Crow web framework, we must include all dependencies. We define each of our algorithms in the backend. The logic and implementation for those are studied from the references listed at the end of the documentation. The execution of each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

The screenshot shows a code editor interface with the following details:

- EXPLORER** panel on the left showing project files:
 - CPU_SCHEDULING
 - backend
 - build
 - CMakeLists.txt
 - main.cpp
 - frontend
 - index.html
 - script.js
 - style.css
 - thirdparty
 - README.md
 - run.sh
- main.cpp** file open in the center editor area, showing C++ code for CPU scheduling. The code implements Shortest Remaining Time First (SRTF) scheduling algorithm.
- STATUS BAR** at the bottom right showing: Ln 592, Col 10 | Spaces: 4 | LF | C++ | Q

```
main.cpp
backend > main.cpp
98     SchedulingResult scheduleLJF(std::vector<Process> processes) {
112     for (int i = 0; i < n; ++i) {
133         result.schedule.push_back({p.id, p.burstTimes[0]});
134         currentTime = result.completionTimes[p.id];
135         done[maxBurstIndex] = true;
136     }
137
138     return result;
139 }
140
141 // Shortest Remaining Time First (SRTF)
142 SchedulingResult scheduleSRTF(std::vector<Process> processes) {
143     int n = processes.size();
144     SchedulingResult result;
145     result.completionTimes.resize(n);
146     result.turnAroundTimes.resize(n);
147     result.waitingTimes.resize(n);
148     result.responseTimes.resize(n);
149     std::vector<int> remainingBurstTimes(n);
150
151     for (int i = 0; i < n; ++i) {
152         remainingBurstTimes[i] = processes[i].burstTimes[0];
153     }
154
155     int currentTime = 0;
156     int completed = 0;
157     std::vector<bool> started(n, false);
158     while (completed != n) {
159         int minBurstIndex = -1;
160         for (int i = 0; i < n; ++i) {
161             if (processes[i].arrivalTime <= currentTime && remainingBurstTimes[i] > 0) {
162                 if (minBurstIndex == -1 || remainingBurstTimes[i] < remainingBurstTimes[minBurstIndex]) {
163                     minBurstIndex = i;
164                 }
165             }
166         }
167
168         if (minBurstIndex == -1) {
169             currentTime++;
170             continue;
171         }
172         if (!started[minBurstIndex]) {
173             if (minBurstIndex == -1 || remainingBurstTimes[minBurstIndex] < remainingBurstTimes[minBurstIndex + 1]) {
174                 minBurstIndex = minBurstIndex + 1;
175             }
176             started[minBurstIndex] = true;
177             currentTime += processes[minBurstIndex].burstTimes[0];
178             completed++;
179         }
180     }
181
182     return result;
183 }
```

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.



```

CPU_SCHEDULING
└── main.cpp
    └── main.cpp
        ├── backend
        ├── frontend
        └── ...

```

```

412     SchedulingResult scheduleHRRN(std::vector<Process> processes) {
413         for (int i = 0; i < n; ++i) {
414             if (highestRRIndex == -1) {
415                 highestRRIndex = i;
416                 continue;
417             }
418
419             const auto& p = processes[highestRRIndex];
420             result.responseTimes[p.id] = currentTime - p.arrivalTime;
421             result.completionTimes[p.id] = currentTime + p.burstTimes[0];
422             result.turnAroundTimes[p.id] = result.completionTimes[p.id] - p.arrivalTime;
423             result.waitingTimes[p.id] = result.turnAroundTimes[p.id] - p.burstTimes[0];
424             result.schedule.push_back({p.id, p.burstTimes[0]});
425             currentTime = result.completionTimes[p.id];
426             done[highestRRIndex] = true;
427         }
428
429         return result;
430     }
431
432     std::pair<std::string, SchedulingResult> findBestAlgorithm(const std::vector<Process>& processes, int timeQuantum = 1, int contextSwitch = 1) {
433         std::vector<std::pair<std::string, SchedulingResult>> results = {
434             {"FCFS", scheduleFCFS(processes)},
435             {"SJF", scheduleSJF(processes)},
436             {"SJF(Preemptive)", scheduleSRTF(processes)},
437             {"LJF", scheduleLJF(processes)},
438             {"LJF(Preemptive)", scheduleLRTF(processes)},
439             {"Priority", schedulePriorityNP(processes)},
440             {"Priority(Preemptive)", schedulePriorityP(processes)},
441             {"Round Robin", scheduleRR(processes, timeQuantum, contextSwitch)},
442             {"Proposed Algorithm", scheduleHRRN(processes)}
443         };
444
445         auto rankAlgorithms = [](const std::vector<double>& values, bool ascending = true) {
446             std::vector<int> rank(values.size());
447             std::iota(rank.begin(), rank.end(), 0);
448             std::sort(rank.begin(), rank.end(), [&](int a, int b) {
449                 return ascending ? values[a] < values[b] : values[a] > values[b];
450             });
451             return rank;
452         };
453
454         std::vector<double> avgWaitingTimes, avgTurnAroundTimes, avgResponseTimes, cpuUtils, throughputs;

```

The screenshot shows a code editor interface with the file `main.cpp` open. The code implements various scheduling algorithms and a function to find the best one based on performance metrics. The interface includes a sidebar with project files like `CMakeLists.txt` and `index.html`, and a bottom status bar showing line and column counts, spaces used, and file type.

```

CPU_SCHEDULING
└── backend
    ├── build
    └── CMakeLists.txt
    └── main.cpp
        └── index.html
        └── script.js
        └── style.css
        └── thirdparty
        └── README.md
        └── run.sh

main.cpp
459     std::pair<std::string, SchedulingResult> findBestAlgorithm(const std::vector<Process>& processes, int timeQuantum = 1, int contextSwitches = 0) {
460         auto rankAlgorithms = [](const std::vector<double> &values, bool ascending = true) {
461             std::iota(rank.begin(), rank.end(), 0);
462             std::sort(rank.begin(), rank.end(), [&](int a, int b) {
463                 return ascending ? values[a] < values[b] : values[a] > values[b];
464             });
465             return rank;
466         };
467
468         std::vector<double> avgWaitingTimes, avgTurnAroundTimes, avgResponseTimes, cpuUtils, throughputs;
469         int minArrivalTime = INT_MAX;
470
471         for (const auto& p : processes) {
472             minArrivalTime = std::min(minArrivalTime, p.arrivalTime);
473         }
474
475         for (const auto& result : results) {
476             const auto& res = result.second;
477             double avgWaitingTime = std::accumulate(res.waitingTimes.begin(), res.waitingTimes.end(), 0.0) / res.waitingTimes.size();
478             double avgTurnAroundTime = std::accumulate(res.turnAroundTimes.begin(), res.turnAroundTimes.end(), 0.0) / res.turnAroundTimes.size();
479             double avgResponseTime = std::accumulate(res.responseTimes.begin(), res.responseTimes.end(), 0.0) / res.responseTimes.size();
480             int totalBurstTime = std::accumulate(res.turnAroundTimes.begin(), res.turnAroundTimes.end(), 0);
481             int completionTime = *std::max_element(res.completionTimes.begin(), res.completionTimes.end());
482             double cpUtil = static_cast<double>(totalBurstTime) / (completionTime + res.contextSwitches);
483             double throughput = static_cast<double>(processes.size()) / (completionTime + res.contextSwitches - minArrivalTime);
484
485             avgWaitingTimes.push_back(avgWaitingTime);
486             avgTurnAroundTimes.push_back(avgTurnAroundTime);
487             avgResponseTimes.push_back(avgResponseTime);
488             cpuUtils.push_back(cpUtil);
489             throughputs.push_back(throughput);
490         }
491
492         auto wtRank = rankAlgorithms(avgWaitingTimes);
493         auto tatRank = rankAlgorithms(avgTurnAroundTimes);
494         auto rtRank = rankAlgorithms(avgResponseTimes);
495         auto cpUtilRank = rankAlgorithms(cpuUtils, false);
496         auto throughputRank = rankAlgorithms(throughputs, false);
497
498         std::vector<double> overallRanks(results.size(), 0);
499         for (size_t i = 0; i < results.size(); ++i) {
500             overallRanks[i] = (wtRank[i] + tatRank[i] + rtRank[i] + cpUtilRank[i] + throughputRank[i]) / 5.0;
501         }
502     }
503
504     auto wtRank = rankAlgorithms(avgWaitingTimes);
505     auto tatRank = rankAlgorithms(avgTurnAroundTimes);
506     auto rtRank = rankAlgorithms(avgResponseTimes);
507     auto cpUtilRank = rankAlgorithms(cpuUtils, false);
508     auto throughputRank = rankAlgorithms(throughputs, false);
509
510     std::vector<double> overallRanks(results.size(), 0);
511     for (size_t i = 0; i < results.size(); ++i) {
512         overallRanks[i] = (wtRank[i] + tatRank[i] + rtRank[i] + cpUtilRank[i] + throughputRank[i]) / 5.0;
513     }
514
515     auto overallRank = rankAlgorithms(overallRanks);
516
517     std::vector<double> finalRanks(results.size(), 0);
518     for (size_t i = 0; i < results.size(); ++i) {
519         finalRanks[i] = overallRank[overallRanks[i]];
520     }
521
522     return { "RoundRobin", finalRanks };
523 }

```

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.

The screenshot shows a code editor interface with the following details:

- EXPLORER** sidebar: Shows project structure with **CPU_SCHEDULING** as the root. Inside are **backend**, **frontend**, **build**, **CMakeLists.txt**, **main.cpp**, **index.html**, **script.js**, **style.css**, **thirdparty**, **README.md**, and **run.sh**.
- main.cpp** file content (partial):

```
int main() {
    CROW_ROUTE(app, "/schedule").methods(crow::HTTPMethod::POST)
        ([](const crow::request& req) {
            auto body = json::parse(req.body);
            std::string algorithm = body["algorithm"];
            auto processesJson = body["processes"];
            std::vector<Process> processes;
            for (size_t i = 0; i < processesJson.size(); ++i) {
                Process p;
                p.id = i;
                p.priority = body["priority"][i];
                p.arrivalTime = body["arrivalTime"][i];
                for (const auto& burstTime : body["processTime"][i]) {
                    p.burstTimes.push_back(burstTime);
                }
                processes.push_back(p);
            }

            SchedulingResult result;
            if (algorithm == "fcfs") {
                result = scheduleFCFS(processes);
            } else if (algorithm == "sjf") {
                result = scheduleSJF(processes);
            } else if (algorithm == "ljf") {
                result = scheduleLJF(processes);
            } else if (algorithm == "srft") {
                result = scheduleSRFT(processes);
            } else if (algorithm == "lrtf") {
                result = scheduleLRTF(processes);
            } else if (algorithm == "rr") {
                result = scheduleRR(processes, body["timeQuantum"], body["contextSwitch"]);
            } else if (algorithm == "np") {
                result = schedulePriorityNP(processes);
            } else if (algorithm == "pp") {
                result = schedulePriorityP(processes);
            } else if (algorithm == "hrnn") {
                result = scheduleHRNN(processes);
            }

            json response = {
                {"completionTimes", result.completionTimes},
            };
        });
}
```
- Bottom status bar: Ln 592, Col 10 | Spaces: 4 | UTF-8 | LF | C++ | Icons for close, save, etc.

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 displays two code editor tabs side-by-side.

Top Tab: main.cpp

```

backend > G main.cpp
522 int main() {
526     ([](const crow::request& req) {
527         ;
528         return crow::response(response.dump());
529     });
530 }
531 CROW_ROUTE(app, "/best-algorithm").methods(crow::HTTPMethod::POST)
532 ([](const crow::request& req) {
533     auto body = json::parse(req.body);
534     auto processesJson = body["processId"];
535     std::vector<Process> processes;
536
537     for (size_t i = 0; i < processesJson.size(); ++i) {
538         Process p;
539         p.id = i;
540         p.priority = body["priority"][i];
541         p.arrivalTime = body["arrivalTime"][i];
542         for (const auto& burstTime : body["processTime"][i]) {
543             p.burstTimes.push_back(burstTime);
544         }
545         processes.push_back(p);
546     }
547
548     int timeQuantum = body.contains("timeQuantum") ? body["timeQuantum"] : 1;
549     int contextSwitch = body.contains("contextSwitch") ? body["contextSwitch"] : 1;
550
551     auto bestAlgResult = findBestAlgorithm(processes, timeQuantum, contextSwitch);
552
553     json response = {
554         {"bestAlgorithm", bestAlgResult.first},
555         {"cpuUtilization", bestAlgResult.second.cpuUtilization},
556         {"throughput", bestAlgResult.second.throughput},
557         {"turnAroundTime", bestAlgResult.second.turnAroundTimes},
558         {"waitingTime", bestAlgResult.second.waitingTimes},
559         {"responseTime", bestAlgResult.second.responseTimes}
560     };
561
562     return crow::response(response.dump());
563 };
564
565 app.port(18080).multithreaded().run();
566
567 
```

Bottom Tab: script.js

```

1 let priorityPreference = 1;
2 document.getElementById("priority-toggle-btn").onclick = () => {
3     let currentPriorityPreference = document.getElementById("priority-preference").innerText;
4     if (currentPriorityPreference == "high") {
5         document.getElementById("priority-preference").innerText = "low";
6     } else {
7         document.getElementById("priority-preference").innerText = "high";
8     }
9     priorityPreference *= -1;
10 };
11
12 let selectedAlgorithm = document.getElementById('algo');
13
14 function checkTimeQuantumInput() {
15     let timequantum = document.querySelector("#time-quantum").classList;
16     if (selectedAlgorithm.value == 'rr') {
17         timequantum.remove("hide");
18     } else {
19         timequantum.add("hide");
20     }
21 }
22
23 function checkPriorityCell() {
24     let prioritycell = document.querySelectorAll(".priority");
25     if (selectedAlgorithm.value == "pnp" || selectedAlgorithm.value == "pp") {
26         prioritycell.forEach(element => {
27             element.classList.remove("hide");
28         });
29     } else {
30         prioritycell.forEach(element => {
31             element.classList.add("hide");
32         });
33     }
34 }
35
36 selectedAlgorithm.onchange = () => {
37     checkTimeQuantumInput();
38     checkPriorityCell();
39 };
40
41 function inputOnChange() {
42     let inputs = document.querySelectorAll('input');
43     inputs.forEach((input) => {
44         if (input.type == 'number') { 
```

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 through 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.

```

CPU_SCHEDULING
  - backend
    - build
    - CMakeLists.txt
    - main.cpp
  - frontend
    - index.html
    - script.js
    - style.css
  - thirdparty
  - README.md
  - run.sh

script.js

function showBestAlgorithm() {
  let input = new Input();
  for (let i = 1; i <= process; i++) {
    input.processId.push(i - 1);
    let rowCells1 = document.querySelector(".main-table").rows[2 * i - 1].cells;
    let rowCells2 = document.querySelector(".main-table").rows[2 * i].cells;
    input.priority.push(Number(rowCells1[1].firstElementChild.value));
    input.arrivalTime.push(Number(rowCells1[2].firstElementChild.value));
    let ptn = Number(rowCells2.length);
    let pta = [];
    for (let j = 0; j < ptn; j++) {
      pta.push(Number(rowCells2[j].firstElementChild.value));
    }
    input.processTime.push(pta);
    input.processTimeLength.push(ptn);
  }
  input.totalBurstTime = new Array(process).fill(0);
  input.processTime.forEach((e1, i) => {
    e1.forEach((e2, j) => {
      if (j % 2 == 0) {
        input.totalBurstTime[i] += e2;
      }
    });
  });
  let bestAlgorithmHeading = document.createElement("h3");
  bestAlgorithmHeading.innerHTML = "Best Algorithm: " + output.bestAlgorithm;
  outputDiv.appendChild(bestAlgorithmHeading);

  let cpuUtilization = 0;
  let totalBurstTime = output.turnAroundTime.reduce((sum, time) => sum + time, 0);
  let lastCompletionTime = Math.max(...output.completionTimes);

  let cpuUtilizationElem = document.createElement("p");
  cpuUtilizationElem.innerHTML = "CPU Utilization : " + (totalBurstTime / lastCompletionTime * 100).toFixed(2) + "%";
  outputDiv.appendChild(cpuUtilizationElem);

  let throughput = document.createElement("p");
  throughput.innerHTML = "Throughput : " + (process / lastCompletionTime).toFixed(2);
  outputDiv.appendChild(throughput);

  let tat = document.createElement("p");
  tat.innerHTML = "Turn Around Time: " + output.turnAroundTime.map(time => time.toFixed(2)).join(", ");
  outputDiv.appendChild(tat);
}

let tat = document.createElement("p");
tat.innerHTML = "Turn Around Time: " + output.turnAroundTime.map(time => time.toFixed(2)).join(", ");

```

The screenshot shows the VS Code interface with the file `index.html` open in the editor. The code is an HTML document for a CPU Scheduler application. It includes a header with meta tags, a title, and a link to a CSS file. The main content features a heading, a note about tiebreakers, and a list of instructions. Below this is a section for toggling priority, which includes a button with a refresh icon. A form for selecting scheduling algorithms follows, containing options for FCFS, SJF, LJF, SRTF, LRFT, Round Robin, Priority (Non Preemptive), Priority (Preemptive), and HRRN. The code ends with a table element.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CPU Scheduler | Anupriya</title>
    <link rel="stylesheet" href="style.css">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">
  </head>
  <body>
    <h1>Advanced CPU Scheduler</h1>
    <h4>Created by Anupriya, IIT Roorkee:</h4>
    <h3>Note :</h3>
    <ol class="instructions">
      <li>The process ID is the tiebreaker.</li>
      <li>Total burst time is considered when burst time is criteria for comparison.</li>
      <li>Context Switch not considered at start and end of processes.</li>
    </ol>
    <h3>Toggle priority :</h3>
    <ol class="preferences">
      <li>
        indicates
        <button id="priority-toggle-btn">
          <span id="priority-preference">high</span>
          <i class="fa fa-refresh"></i>
        </button>
      </li>
    </ol>
    <form id="algorithms-form">
      <label for="algo"><h3>Algorithms :</h3></label>
      <select name="algo" id="algo">
        <option value="fcfs">First Come First Serve (FCFS)</option>
        <option value="sjf">Shortest Job First (SJF)</option>
        <option value="ljf">Longest Job First (LJF)</option>
        <option value="srtf">Shortest Remaining Job First (SRTF)</option>
        <option value="lrft">Longest Remaining Job First (LRFT)</option>
        <option value="rr">Round Robin</option>
        <option value="ppn">Priority (Non Preemptive)</option>
        <option value="ppr">Priority (Preemptive)</option>
        <option value="hrrn">Highest Response Ratio Next (HRRN)</option>
      </select>
    </form>
    <br>
    <table class="main-table">
      <thead>

```

The screenshot shows the VS Code interface with the file `CMakeLists.txt` open in the editor. The code defines a C++ executable named `cpu_scheduling` using the `add_executable` command. It specifies the project name as `cpu_scheduling` and includes the `main.cpp` file. The target is set to include directories for header files from the `asio` library and to link the `pthread` library.

```

cmake_minimum_required(VERSION 3.10)
project(cpu_scheduling)

set(CMAKE_CXX_STANDARD 14)

add_executable(cpu_scheduling main.cpp)

target_include_directories(cpu_scheduling PRIVATE /usr/local/Cellar/asio/1.30.2/include)
target_link_libraries(cpu_scheduling PRIVATE pthread)

```

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:

The screenshot shows a web-based CPU scheduler interface. At the top, it says "Advanced CPU Scheduler" and "Created by Anupriya, IIT Roorkee". Below that is a note: "Note : 1. The process ID is the tiebreaker. 2. Total burst time is considered when burst time is criteria for comparison. 3. Context Switch not considered at start and end of processes." A "Toggle priority :" button is present, with "high" selected. Under "Algorithms:", "First Come First Serve (FCFS)" is chosen. A table shows one process: P1 with arrival time 0 and burst time 1. Buttons for "Add Process" and "Delete Process" are available. A dropdown for "Context Switch Time" is set to 0. Buttons for "Calculate" and "Reset" are at the bottom.

Select any algorithm of your choice:

The screenshot shows the same web-based CPU scheduler interface. The "Algorithms:" dropdown is open, displaying several scheduling algorithms: First Come First Serve (FCFS), Shortest Job First (SJF), Longest Job First (LJF), Shortest Remaining Job First (SRTF), Longest Remaining Job First (LRTF), Round Robin, Priority (Non Preemptive), Priority (Preemptive), and Highest Response Ratio Next (HRRN). The "First Come First Serve (FCFS)" option is checked. The rest of the interface remains the same as the previous screenshot, including the process table and control buttons.

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)

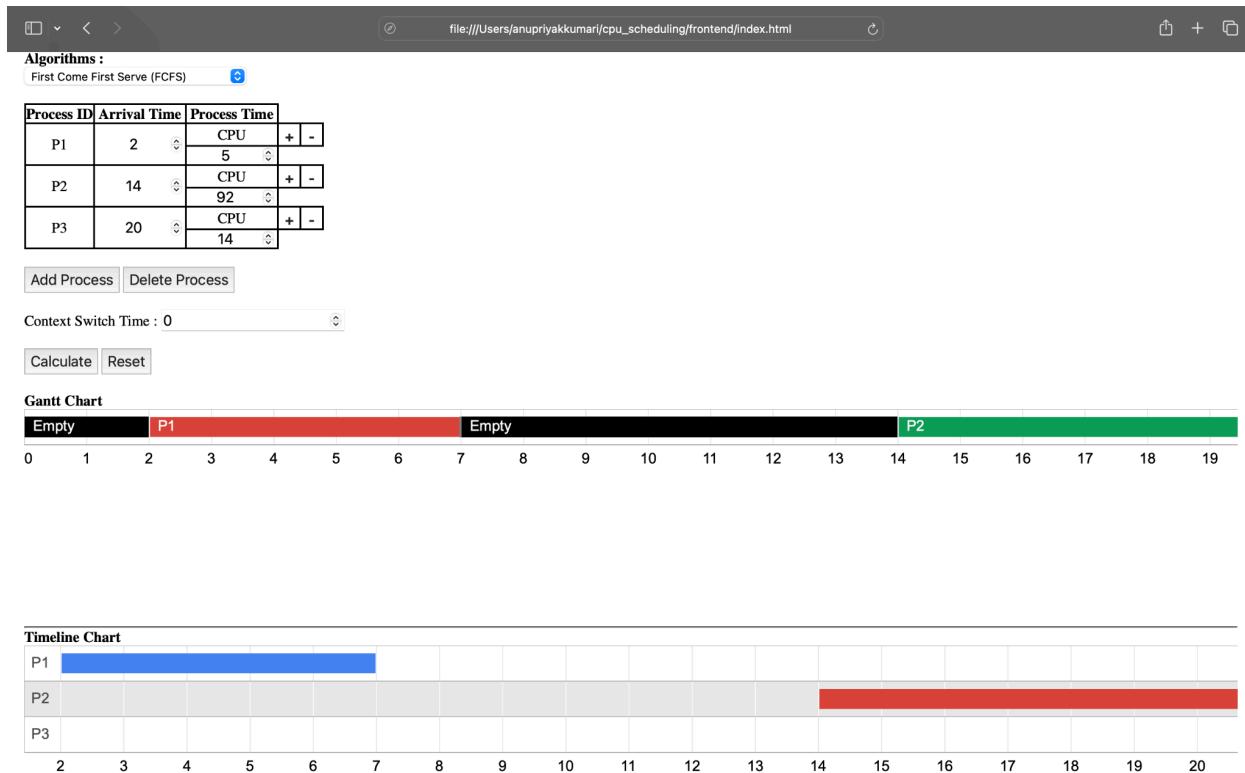
Set a context switch time and click on Calculate:

The screenshot shows a web-based CPU scheduler application. At the top, there's a header bar with icons for file operations and a title 'file:///Users/anupriyakkumar/cpu_scheduling/frontend/index.html'. Below the header, the page title is 'CPU Schedular | Anupriya'. The main content area has a heading 'Advanced CPU Scheduler' and a note 'Created by Anupriya, IIT Roorkee'. It includes a 'Note:' section with three points: 1. The process ID is the tiebreaker. 2. Total burst time is considered when burst time is criteria for comparison. 3. Context Switch not considered at start and end of processes. There are buttons for 'Toggle priority' (set to 'high'), 'Algorithms' (set to 'First Come First Serve (FCFS)'), and a dropdown menu. A table lists three processes (P1, P2, P3) with their arrival times (2, 14, 20), burst times (5, 92, 14), and context switch times (0). Buttons for 'Add Process' and 'Delete Process' are below the table. A 'Context Switch Time' input field is set to 0. At the bottom are 'Calculate' and 'Reset' buttons.

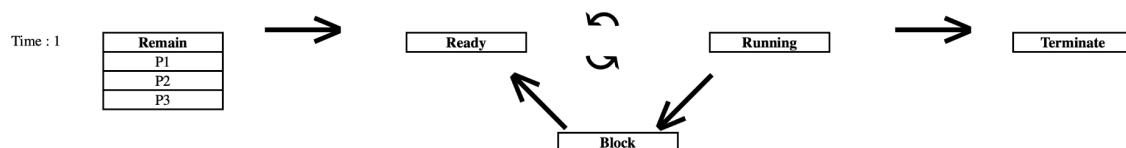
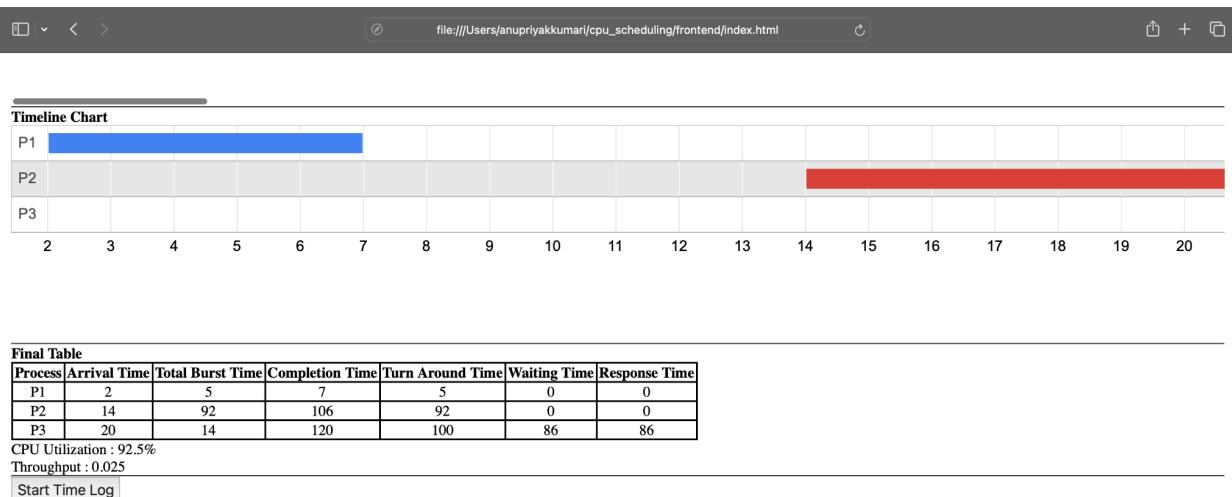
Process ID	Arrival Time	Process Time
P1	2	CPU 5
P2	14	CPU 92
P3	20	CPU 14

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:



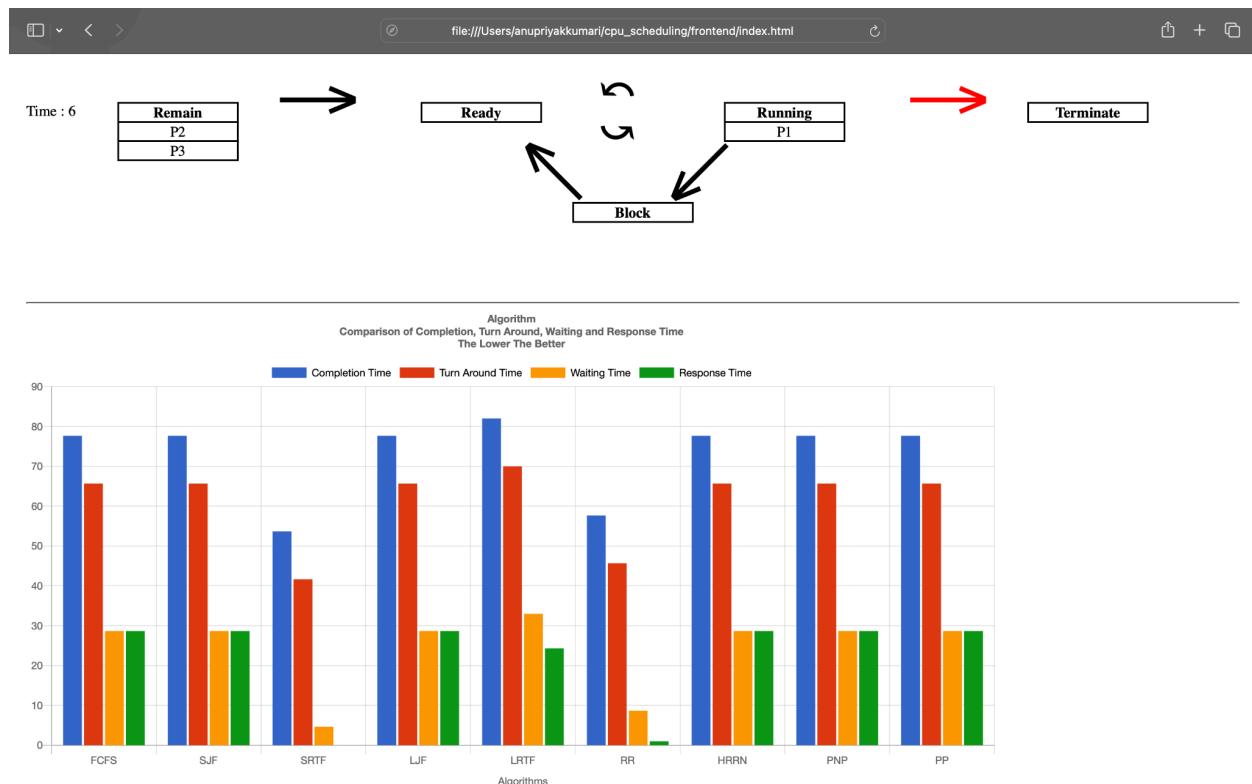
As we scroll down, we see the final table that completely describes the processes - completion, turn around, waiting and response time. The lower these are, the better is the algorithm.

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:

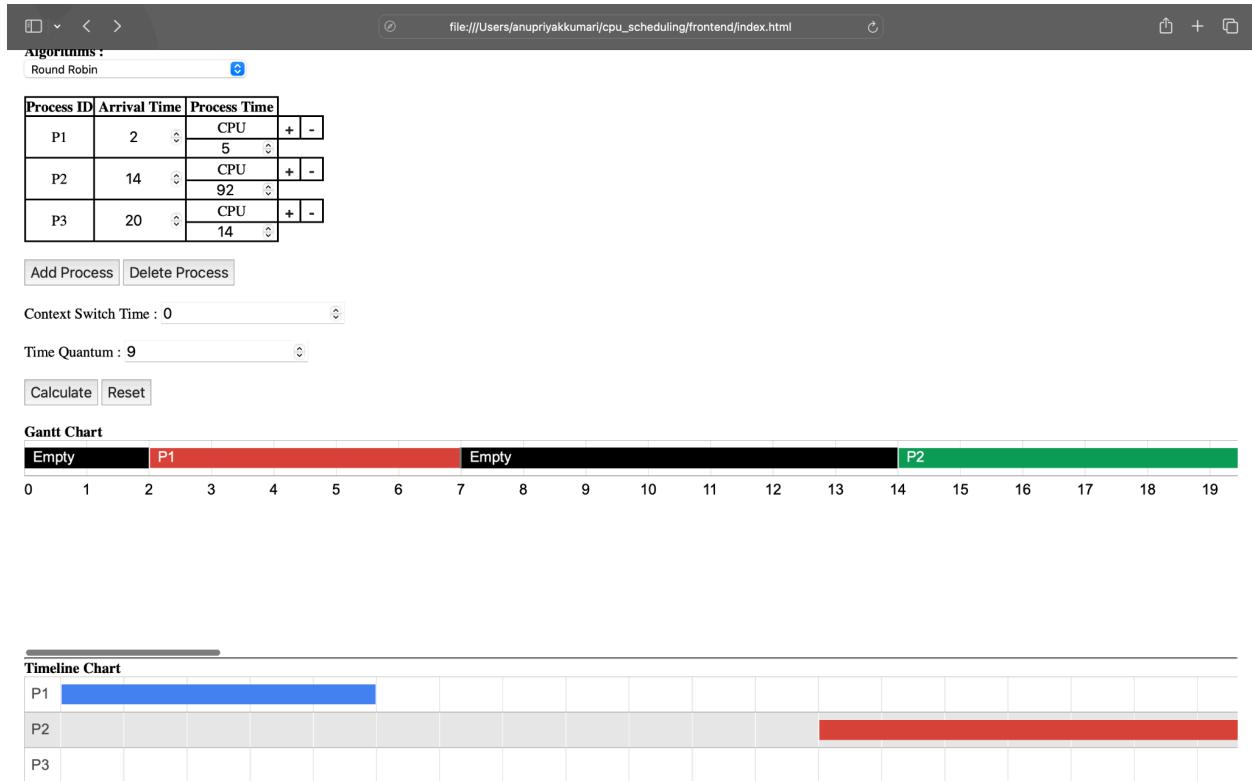
To choose the best algorithm, we perform a comprehensive comparison analysis and look at the composite bar chart showing the completion, turn around, waiting and response time for each algorithm:



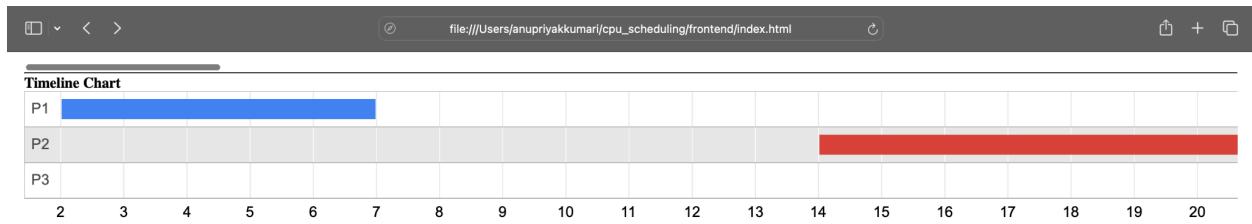
We can unselect a certain metric used for comparison and focus only on one metric. The lower the time value, the better the algorithm. We would also observe that for the most optimal algorithm, the CPU Utilization and throughput would be maximum.

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.

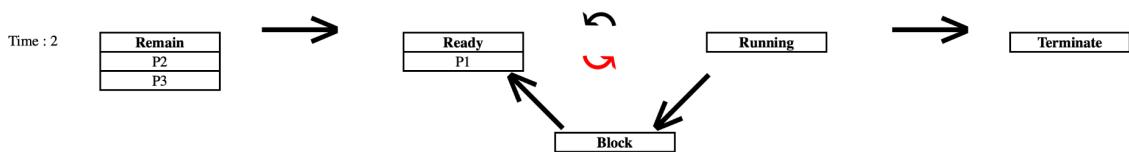


Final Table

Process	Arrival Time	Total Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
P1	2	5	7	5	0	0
P2	14	92	120	106	14	0
P3	20	14	46	26	12	3

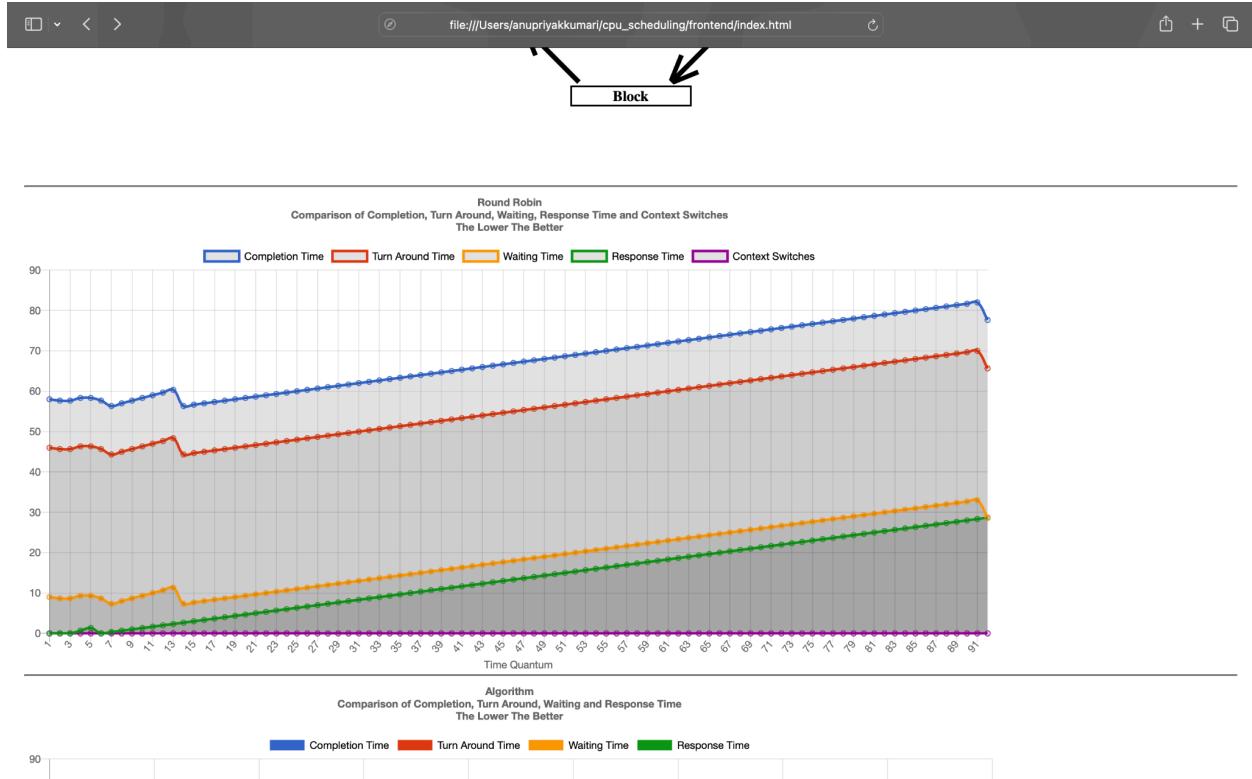
CPU Utilization : 92.5%
Throughput : 0.025

Start Time Log

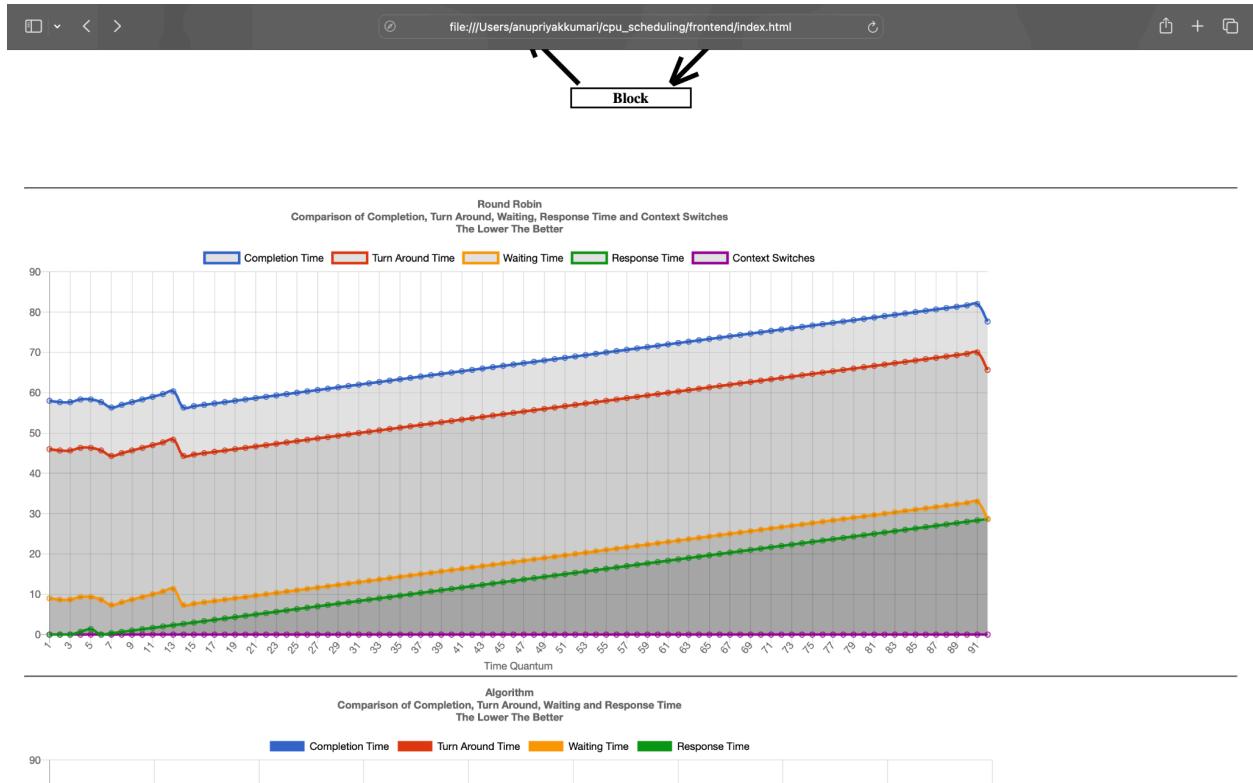


As before, we show a final table, CPU utilization and throughput metrics, a time log explaining the process.

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.

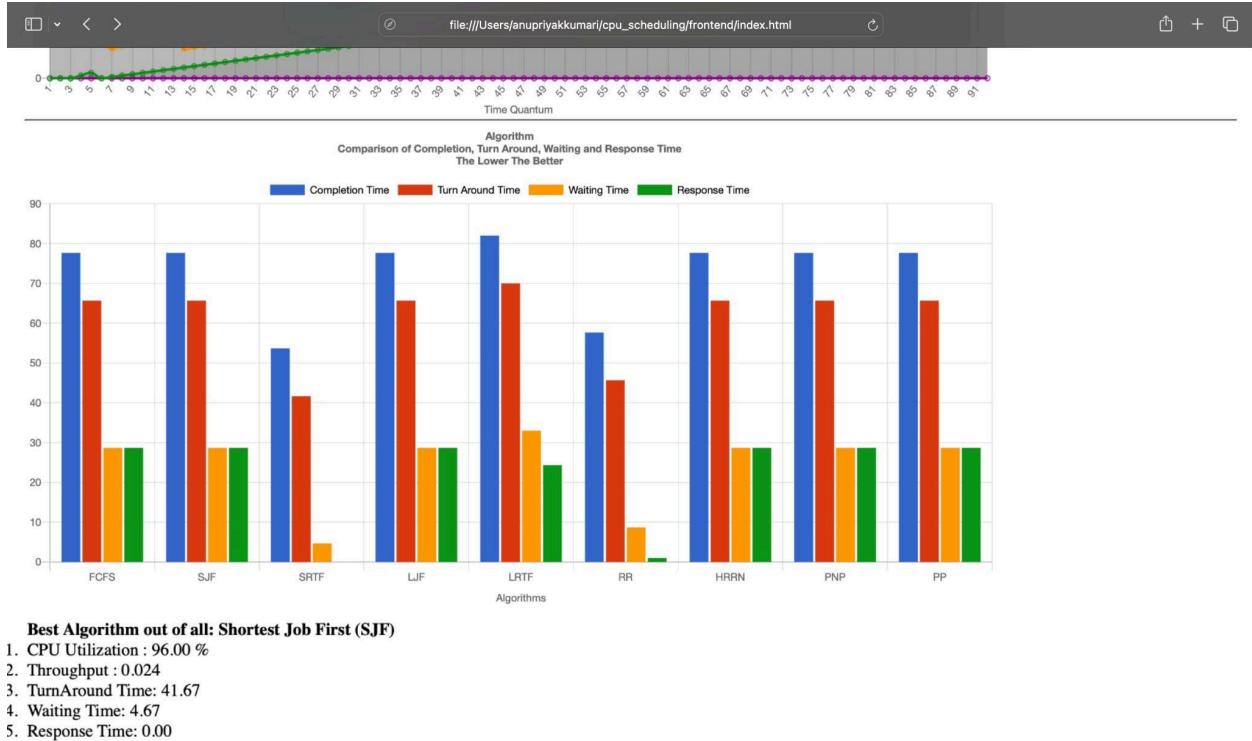


Finally, we show the composite bar chart which visually compares all the algorithms for the defined processes:



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.



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:

- 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_AyyvSWfhRgyA-iHk
- <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.