Project Report: CPU-Scheduler

Garvit Khurana
Department of Electrical Engineering
22115059

Project Overview

CPU-Scheduler is a desktop application which allow users to see on a ground level of how our CPU schedule the processes which are running in our real-time systems. A total of 4 CPU scheduling algorithms are implemented which are First Come First Serve (FCFS), Round Robin (RR), Preemptive Shortest Job First also known as Shortest Remaining Time (SRT) and Multi-Level Queue Scheduling (MLQ). The key feature is of Auto-Scheduling which decides on the basis of processes information that which algorithm will be more appropriate. A Gantt-Chart which shows the start and completion time of a process and an average statistics of chosen algorithm are displayed along with it.
The report clearly explain both textually and visually through images about how the front-end of the application looks as well as the explanation about the implementation of back-end which involves the algorithms. The Tech Stack used for back-end is same as mentioned that is C++ and the front-end uses JavaScript and it's framework Electron to build the window application. The challenges faced and possible updates or features which can be added are discussed at the end of the report.

Contents

1 Phases of building CPU-Scheduler

- **Learning Terminologies**: Since I was new to this I first learnt about why we actually needed a CPU scheduler and what are the terms associated with process information. Then I came to know about process ID, Arrival Time, Burst Time, Priority, Start Time,Completion Time and criteria of scheduling which involves Waiting Time, Response and Turn Around Time.
- **Learning Algorithms**: There are many algorithms and each one of them have their own use case. Learning all of them at once took a bit of time but with dry run it got more and more clear in my mind that how will I implement this particular algorithm. As stated earlier I used **First Come First Serve (FCFS), Round Robin (RR), Preemptive Shortest Job First also known as Shortest Remaining Time (SRT) and Multi-Level Queue Scheduling (MLQ)**.
- **Implementing Back-end**: I wanted to have clean and self explanatory directory structure so I separated each algorithm as a header file and included it in main **scheduler.cpp**. My most of the time went in this phase. Especially thinking of the logic that how I will link with the front-end part. Out of them the most difficult was implementing MLQ as it has some edge cases which are discussed in later part of report.
- **Implementing front-end**: On Web-surfing I came to know about a few ways and I tried each one of them but faced issues which are discussed in last portion of this report but finally came to know that making a desktop application using **electron.js** will be the most straight forward and convenient to build. Since this was new to me so I used the documentation code of Electron framework available on the internet. The HTML and CSS part wasn't a big deal but consumed a good amount of time. The main part of this phase was merging this interactive application with my C++ scheduler file which is already explained too in later part of this report.

  This was how I went through journey of making this project. To understand it visually of how things actually work diagrams and flowchart are added in next section.

2 Flowcharts & Diagrams

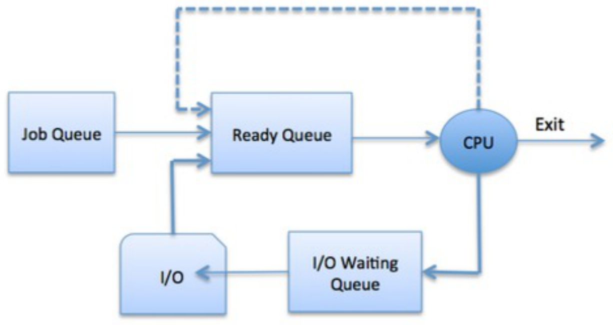


Figure 1: Queuing Diagram

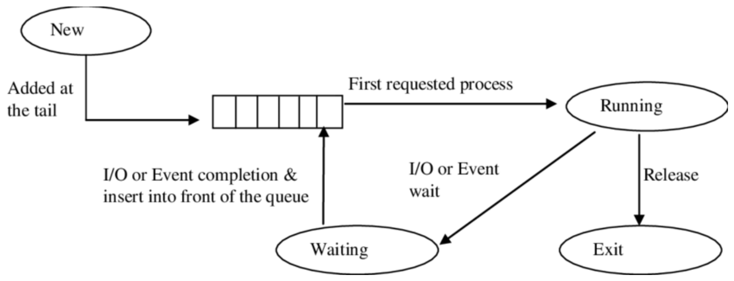


Figure 2: First Come First Serve Algorithm

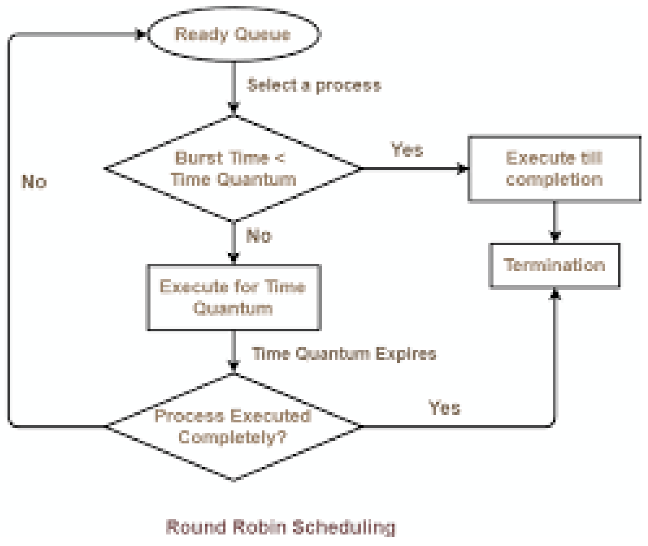


Figure 3: Round Robin Algorithm

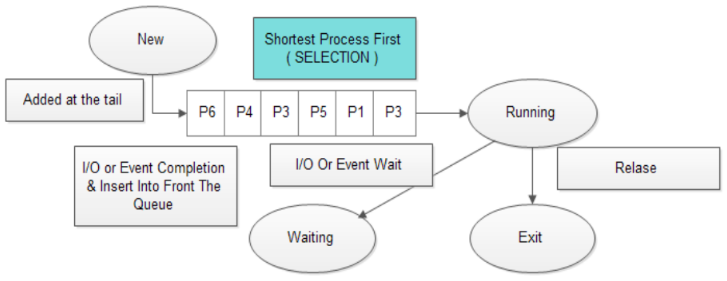

New
Shortest Process First (SELECTION)
Added at the tail
P6
P4
P3
P5
P1
P3
Running
I/O or Event Completion & Insert Into Front The Queue
I/O Or Event Wait
Relase
Waiting
Exit

Figure 4: Shortest Job First Algorithm

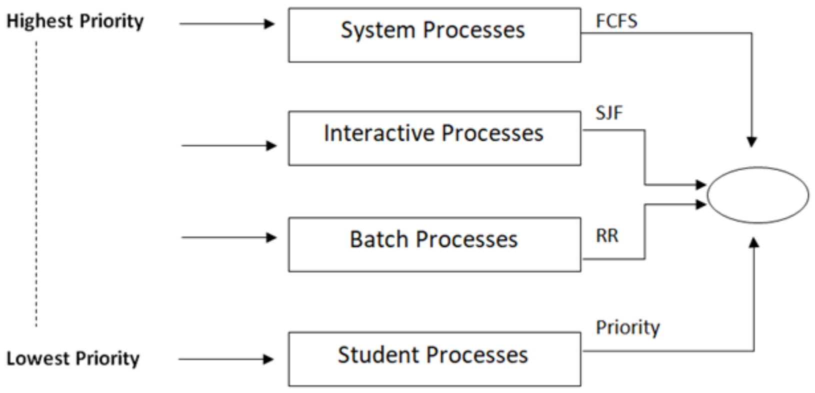


Figure 5: General Multilevel Queue Algorithm

I hope with these flowcharts and diagrams you are able to understand a little bit that what actually happens in the respective algorithm. There are more existing algorithm which are discussed in last portion of the report but first let's go into the implementation of the code base from next section.

3 Front-end Behaviour & Implementation

- CPU-Scheduler's front-end is written in HTML,CSS and Javascript. First let's see how will it look if you open the Application.

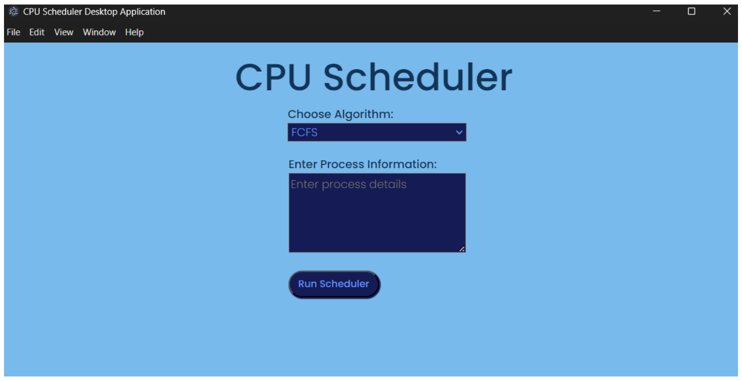
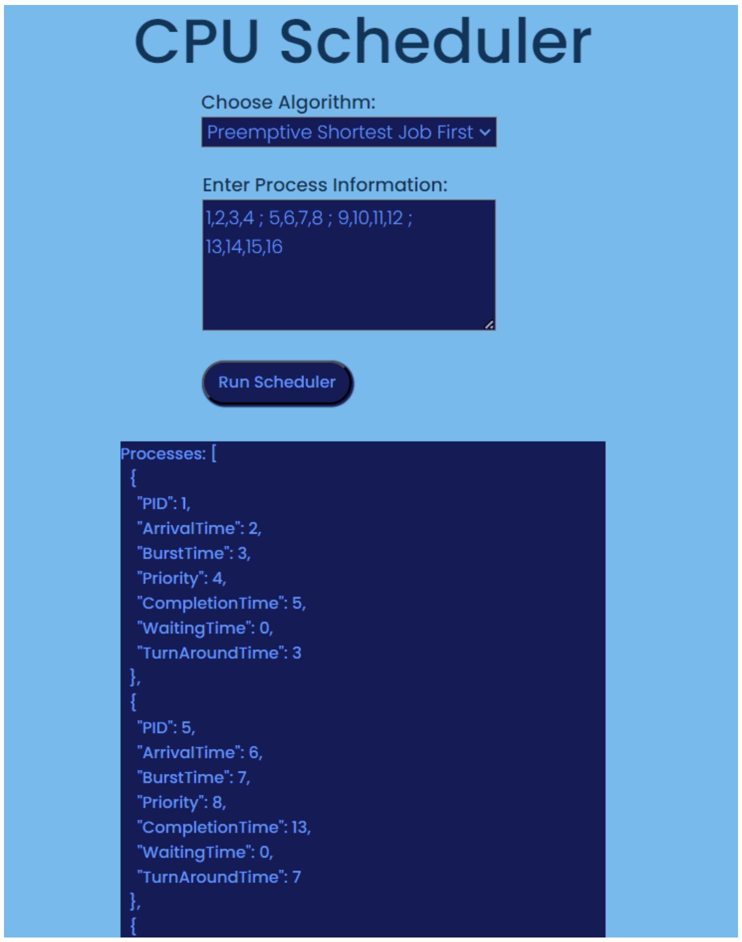

CPU Scheduler Desktop Application
File Edit View Window Help
CPU Scheduler
Choose Algorithm:
FCFS
Enter Process Information:
Enter process details
Run Scheduler

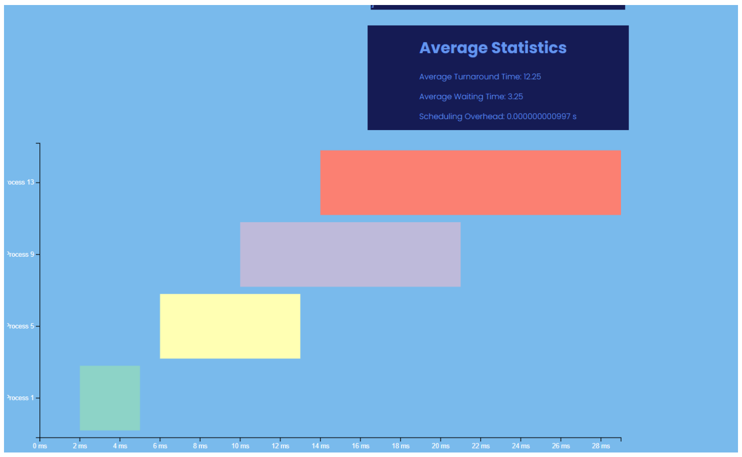
Figure 6: Application Front-end Look

- There are total of 3 feature which includes Algorithm Selector input, Text Area to give process details and a Run Schedule Button. The moment you click on button after providing valid information 3 more things will appear which include each process details including the calculated details associated with each one of them that is completion time, start time etc. Secondly Average Stats of the algorithm and the Gantt-Chart in the end.
  Let me show it first. **Please go to the next page.**



```
1,2,3,4 ; 5,6,7,8 ; 9,10,11,12 ;
13,14,15,16
```

```
Processes: [
 {
  "PID": 1,
  "ArrivalTime": 2,
  "BurstTime": 3,
  "Priority": 4,
  "CompletionTime": 5,
  "WaitingTime": 0,
  "TurnAroundTime": 3
 },
 {
  "PID": 5,
  "ArrivalTime": 6,
  "BurstTime": 7,
  "Priority": 8,
  "CompletionTime": 13,
  "WaitingTime": 0,
  "TurnAroundTime": 7
 },
 {
```



Average Statistics

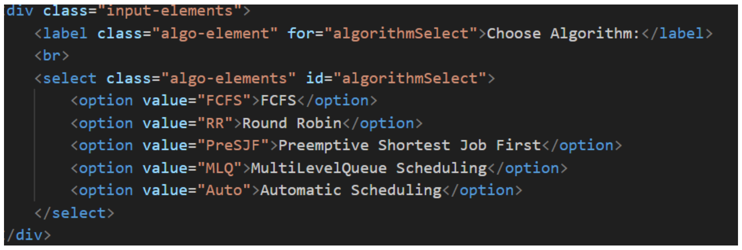
Average Turnaround Time: 12.25

Average Waiting Time: 3.25

Scheduling Overhead: 0.000000000997 s

Figure 7: After Running Algorithm

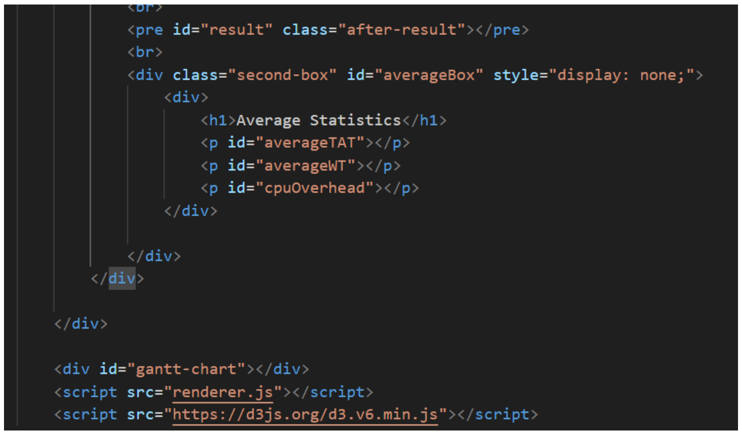
- Let's go through some code snippets and we will be going in order how these files are connected to each other and explaining their functionalities.
- **HTML Structure**: This file consist of the basic structure of how our window will look like the moment you run the application.



```
div class="input-elements">
   <label class="algo-element" for="algorithmSelect">Choose Algorithm:</label>
   <br>
   <select class="algo-elements" id="algorithmSelect">
       <option value="FCFS">FCFS</option>
       <option value="RR">Round Robin</option>
       <option value="PreSJF">Preemptive Shortest Job First</option>
       <option value="MLQ">MultiLevelQueue Scheduling</option>
       <option value="Auto">Automatic Scheduling</option>
   </select>
/div>
```

Figure 8: HTML Code Snippet

- Here I have created a div element which includes a select input type HTML element. The Algorithms chosen by me are just listed down here.

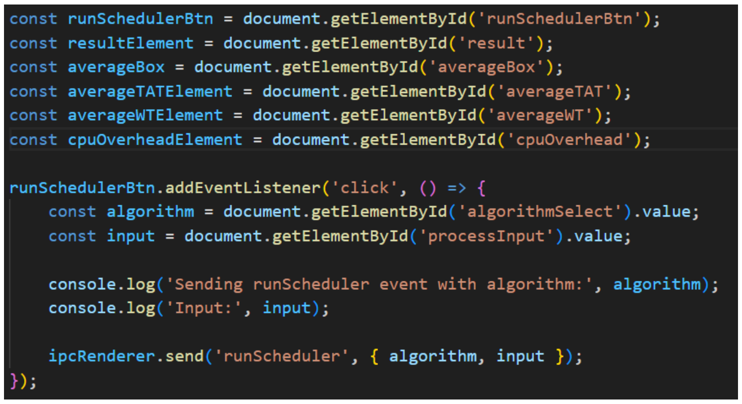


```
            <br>
            <pre id="result" class="after-result"></pre>
            <br>
            <div class="second-box" id="averageBox" style="display: none;">
                <div>
                    <h1>Average Statistics</h1>
                    <p id="averageTAT"></p>
                    <p id="averageWT"></p>
                    <p id="cpuOverhead"></p>
                </div>

            </div>
        </div>

    </div>

    <div id="gantt-chart"></div>
    <script src="renderer.js"></script>
    <script src="https://d3js.org/d3.v6.min.js"></script>
```

Figure 9: HTML Code Snippet

- A **pre** element is created which initially is empty and average statistics div has a display of none. The reason of doing is to make them appear only when a user run the algorithm. You can see a script is being loaded. Yes there are two scripts needed one which is **renderer.js** which consist of logic for the button and changing the layout of html structure whenever it receives result. Secondly a **D3 Script File** which is required for making the Gantt-Chart.
- I don't think there is a need of explaining **CSS Style File** as it just contain the information of font, color used and margin. In my Application I used different **shades of blue color** and font is **poppins, sans-serif** took from **Googlefonts site**, So let's go to JavaScript Files and see their logic.

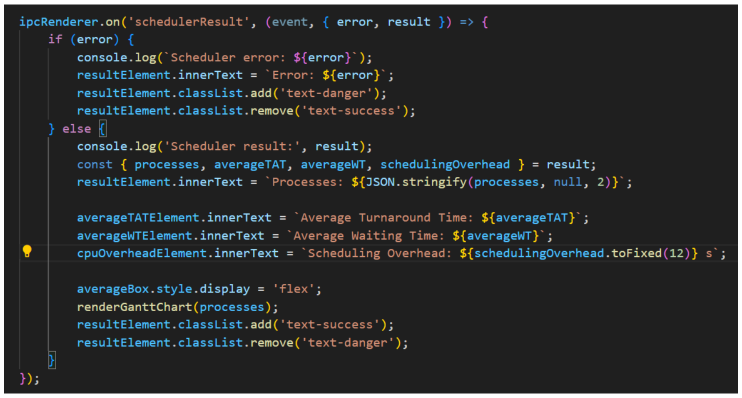
- **Renderer.js & Main.js**: Renderer file is in direct link with html files and associated with sending messages of user.



```
const runSchedulerBtn = document.getElementById('runSchedulerBtn');
const resultElement = document.getElementById('result');
const averageBox = document.getElementById('averageBox');
const averageTATElement = document.getElementById('averageTAT');
const averageWTElement = document.getElementById('averageWT');
const cpuOverheadElement = document.getElementById('cpuOverhead');

runSchedulerBtn.addEventListener('click', () => {
    const algorithm = document.getElementById('algorithmSelect').value;
    const input = document.getElementById('processInput').value;

    console.log('Sending runScheduler event with algorithm:', algorithm);
    console.log('Input:', input);

    ipcRenderer.send('runScheduler', { algorithm, input });
});
```

Figure 10: Renderer.JS Code Snippet

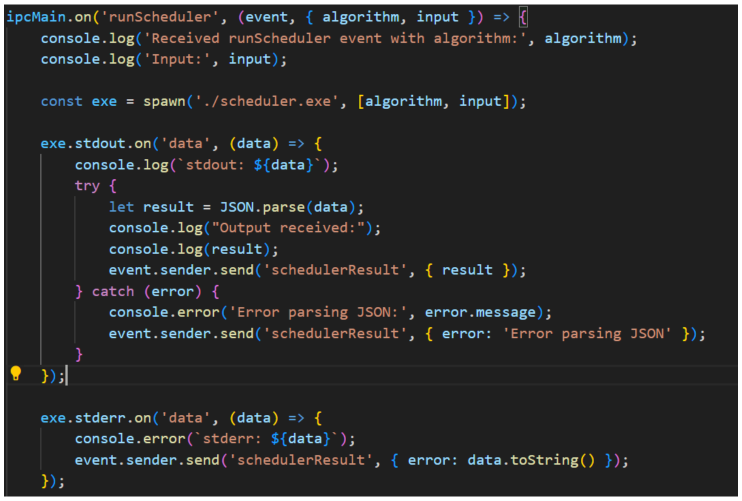
- This is how event listeners are added that is by fetching the HTML element by their ID. There are multiple ways to do so such as doing it by class or querySelector. Here you can see I added a listener of click on the button because the moment user perform this action we must take the algorithm value in form of string and the process details and send them **Main.js** file which will listen for such command that is **runScheduler**.



```
ipcRenderer.on('schedulerResult', (event, { error, result }) => {
    if (error) {
        console.log(`Scheduler error: ${error}`);
        resultElement.innerText = `Error: ${error}`;
        resultElement.classList.add('text-danger');
        resultElement.classList.remove('text-success');
    } else {
        console.log('Scheduler result:', result);
        const { processes, averageTAT, averageWT, schedulingOverhead } = result;
        resultElement.innerText = `Processes: ${JSON.stringify(processes, null, 2)}`;

        averageTATElement.innerText = `Average Turnaround Time: ${averageTAT}`;
        averageWTElement.innerText = `Average Waiting Time: ${averageWT}`;
        cpuOverheadElement.innerText = `Scheduling Overhead: ${schedulingOverhead.toFixed(12)} s`;

        averageBox.style.display = 'flex';
        renderGanttChart(processes);
        resultElement.classList.add('text-success');
        resultElement.classList.remove('text-danger');
    }
});
```

Figure 11: Inter-Process Communication

- ipcRenderer is a feature provided by javascript **Electron Framework** which helps to recieve and send message to main and vice versa. Here on receiving the results from main with no errors then we will change the text of that HTML document. At last a Gantt-Chart render function will be called to where we used **D3.js** to make bars whose starting point and endpoints are already calculated in processes array. That function includes basic styling and adjustments and available in D3 framework's documentation on internet.



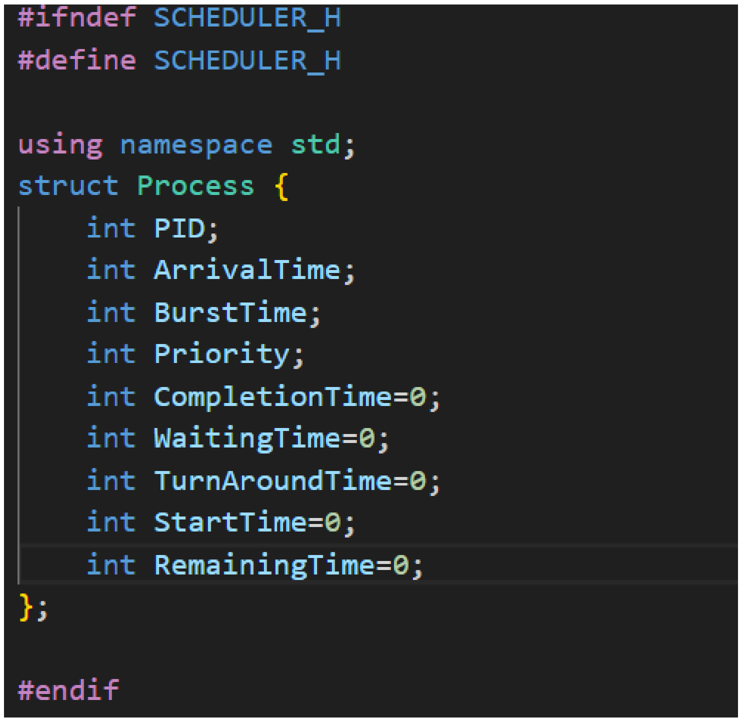
```
ipcMain.on('runScheduler', (event, { algorithm, input }) => {
    console.log('Received runScheduler event with algorithm:', algorithm);
    console.log('Input:', input);

    const exe = spawn('./scheduler.exe', [algorithm, input]);

    exe.stdout.on('data', (data) => {
        console.log(`stdout: ${data}`);
        try {
            let result = JSON.parse(data);
            console.log("Output received:");
            console.log(result);
            event.sender.send('schedulerResult', { result });
        } catch (error) {
            console.error('Error parsing JSON:', error.message);
            event.sender.send('schedulerResult', { error: 'Error parsing JSON' });
        }
    });

    exe.stderr.on('data', (data) => {
        console.error(`stderr: ${data}`);
        event.sender.send('schedulerResult', { error: data.toString() });
    });
```

Figure 12: Main.JS Code Snippet

- So this basic callback function is what interacting with our **C++ EXE** file. Whatever the standard output is coming in exe file we are reading this and we specifically made the output of scheduler file in form of array of string so that we can read it in context with javascript.
- Now Let's look at our back-end files too.**Please go to next page**.

4 Back-end Algorithms

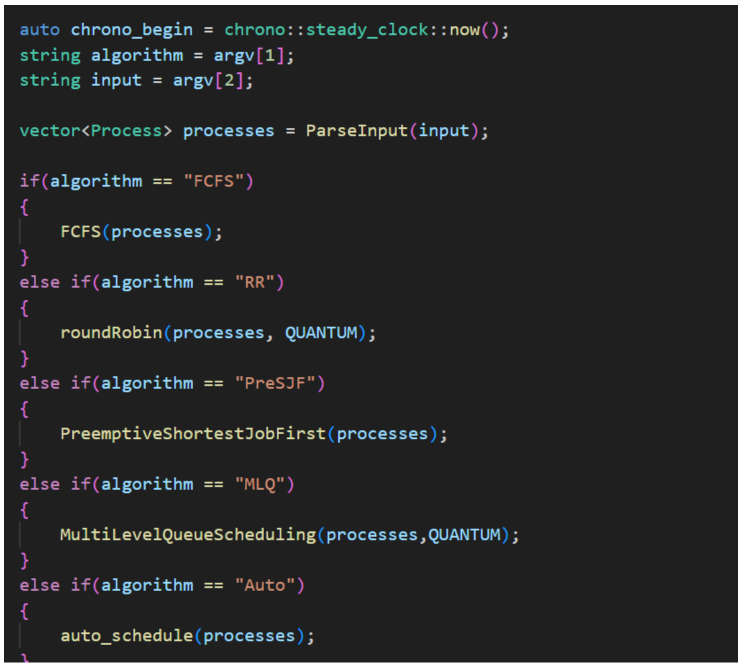
- Instead of explaining algorithm here, I will be explaining how things are organised because that algorithm part could be easily understood from Neso Academy YouTube Channel (resource given by ACM club). So here it goes:-



```
#ifndef SCHEDULER_H
#define SCHEDULER_H

using namespace std;
struct Process {
    int PID;
    int ArrivalTime;
    int BurstTime;
    int Priority;
    int CompletionTime=0;
    int WaitingTime=0;
    int TurnAroundTime=0;
    int StartTime=0;
    int RemainingTime=0;
};

#endif
```

Figure 13: Process Structure Block

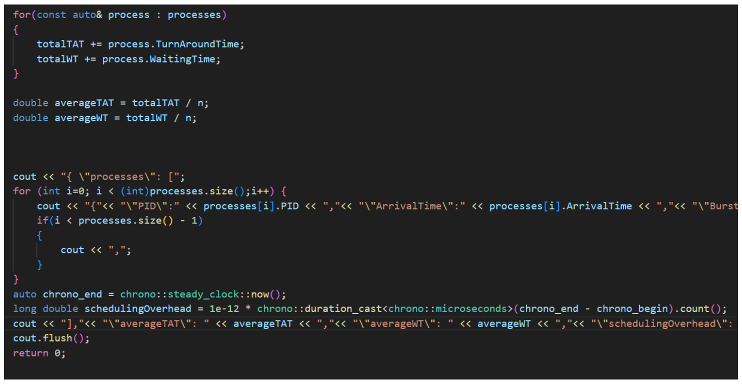
- This is a header file which will be included in almost all files and defines a particular process object. Some of the information will be given by user while others will be calculated after execution of the algorithm.
- Just like this all algorithms are stored as in the form of header files and will be included in our **scheduler.cpp** file which handles main execution of back-end. Let's check that file as it is the core of our back-end.



```cpp
auto chrono_begin = chrono::steady_clock::now();
string algorithm = argv[1];
string input = argv[2];

vector<Process> processes = ParseInput(input);

if(algorithm == "FCFS")
{
    FCFS(processes);
}
else if(algorithm == "RR")
{
    roundRobin(processes, QUANTUM);
}
else if(algorithm == "PreSJF")
{
    PreemptiveShortestJobFirst(processes);
}
else if(algorithm == "MLQ")
{
    MultiLevelQueueScheduling(processes,QUANTUM);
}
else if(algorithm == "Auto")
{
    auto_schedule(processes);
```

Figure 14: Calling Of Algorithm

- On receiving the input to our main function from **Main.js** we call the associated function stored in the header file.



```cpp
for(const auto& process : processes)
{
    totalTAT += process.TurnAroundTime;
    totalWT += process.WaitingTime;
}

double averageTAT = totalTAT / n;
double averageWT = totalWT / n;



cout << "{ \"processes\": [";
for (int i=0; i < (int)processes.size();i++) {
    cout << "{"<< "\"PID\":" << processes[i].PID << ","<< "\"ArrivalTime\":" << processes[i].ArrivalTime << ","<< "\"Burst
    if(i < processes.size() - 1)
    {
        cout << ",";
    }
}
auto chrono_end = chrono::steady_clock::now();
long double schedulingOverhead = 1e-12 * chrono::duration_cast<chrono::microseconds>(chrono_end - chrono_begin).count();
cout << "],"<< "\"averageTAT\": " << averageTAT << ","<< "\"averageWT\": " << averageWT << ","<< "\"schedulingOverhead\":
cout.flush();
return 0;
```

Figure 14: Calculation & STD Output

- After running of algorithm the processes vector which is passed by reference will be changed as per the correct output. Lastly we will be calculating **Average Turn Around Time and Waiting Time**. We will print all this in terminal in such a way that our javascript syntax way so that when it will be converted to string we receive our result in front-end as well.

5 Challenges faced

- I personally faced a lot of difficulty in doing the front-end part. First I tried using react framework but I personally found it difficult to comprehend. When I came to know about electron framework and we can make desktop application then also an issue was coming that is of **ffi-napi**. If you don't know this just consider it that it's used to interact with dynamic library created by us. I was actually trying to make a library of those algorithm but it didn't work out well as file allocation bugs were coming so finally I came up with approach to make use of .exe file.
- In back-end I faced difficulties in MultiLevel Queue Scheduling which basically switches between FCFS and Round Robin algorithm. The issue was with the edge case I thought. The edge case is explained in the code base. The problem is that when I fixed it the runtime was quite high. It was giving an error of time limit exceeded or we can say it wasn't fast enough. So I gave good time in optimising it to reduce the CPU overhead time.

6 Possible upcoming updates & features

Here are some updates or features which can be added to this project :-

- I personally feel that we can easily add more algorithms to it rather than just sticking to a few, However this will take a good amount of time.
- Optimising each algorith is what we can update in future.
- We can refine the logic used in Auto Scheduling to make it more accurate.

7 Dependencies & Conditions

- To run this First you need Node Package Manager(npm) and Node.js.
- Then in terminal run command **npm install electron –save-dev**.
- In terminal run go to your CPU-Scheduler directory and run command **npm start**.
- Your Desktop Application will open.
- **Condition**: whenever your are giving input of your process details make sure you give in this bold text format **1,2,3,4 ; 5,6,7,8**. What does this mean?

  - 1 - Process[1] identity
  - 2 - Process[1] Arrival Time
  - 3 - Process[1] Burst Time
  - 4 - Process[1] Priority
  - 5 - Process[2] identity
  - 6 - Process[2] Arrival Time
  - 7 - Process[2] Burst Time
  - 8 - Process[2] Priority

- **Make sure your add ';' between any two process.**

With this I would like to end my project's report. The extensive research and testing during the project helped to achieve it's objectives. I would like to share my sincere gratitude to ACM for giving me a chance to build this amazing project.