CPU Scheduler Application Report

Garv Sethi
22115057

June 16, 2024

1 Introduction

Operating systems play a crucial role in managing computer resources efficiently, with process scheduling being a fundamental component. The CPU Scheduler application presented in this report aims to simulate and evaluate various process scheduling algorithms through an intuitive user interface built using Electron and robust backend algorithms implemented in C++.

1.1 Purpose and Significance

The primary purpose of the CPU Scheduler application is to provide a platform for studying and comparing different process scheduling algorithms in a controlled environment. Process scheduling is critical for optimizing resource utilization and improving system performance by effectively allocating CPU time among competing processes. By implementing algorithms such as First-Come, First-Served (FCFS), Round Robin (RR), Shortest Job First (SJF), Priority Scheduling, and an automated scheduler, the application allows users to analyze their effectiveness under different scenarios.

Understanding the behavior and performance characteristics of these scheduling algorithms is essential for system administrators, software developers, and students studying operating systems. Through the CPU Scheduler application, users can visualize scheduling decisions using interactive Gantt charts, gaining insights into how different algorithms manage process execution and resource allocation.

1.2 Technologies Used

The CPU Scheduler application leverages modern technologies to deliver a seamless user experience and robust backend processing:

- **Electron**: A framework for building cross-platform desktop applications using web technologies (HTML, CSS, JavaScript). Electron enables the creation of a responsive and interactive user interface.
- **C++**: Chosen for its efficiency and low-level system access, C++ forms the backbone of the application's backend, implementing process scheduling algorithms and performance metrics calculations.
- **D3.js**: A JavaScript library used for creating dynamic and interactive data visualizations. D3.js facilitates the graphical representation of process scheduling and performance metrics through Gantt charts.
- **IPC (Inter-Process Communication)**: Facilitates seamless communication between the Electron frontend and C++ backend, enabling real-time data exchange and synchronization for interactive visualization.

2 System Architecture

The CPU Scheduler application is designed with a modular architecture that combines a frontend user interface developed using Electron with a backend implemented in C++. This section provides an overview of the system architecture, detailing the interaction between different components and their roles in ensuring efficient process scheduling and visualization.

2.1 Overview

The architecture of the CPU Scheduler application comprises two main components:

- **Frontend (Electron)**
- **Backend (C++)**

2.2 Frontend (Electron)

The frontend of the CPU Scheduler application is built using Electron, a framework that allows for the development of cross-platform desktop applications using web technologies such as HTML, CSS, and JavaScript.

- **User Interface**: The Electron frontend provides a graphical user interface (GUI) that allows users to interact with the application. It includes components such as dropdown menus for selecting scheduling algorithms, text areas for inputting process details, and buttons for initiating process scheduling.

- **Rendering Gantt Charts**: D3.js, a JavaScript library for visualizing data using web standards (SVG, CSS), is integrated into the Electron frontend. It dynamically renders Gantt charts based on the scheduling results received from the backend.

- **Inter-Process Communication (IPC)**: Electron facilitates IPC between the frontend and the backend, enabling real-time data exchange. This allows the frontend to send user inputs (selected algorithm and process details) to the backend for processing and receive scheduling results for visualization.

2.3 Backend (C++)

The backend of the CPU Scheduler application is implemented in C++, chosen for its efficiency, low-level system access, and suitability for implementing process scheduling algorithms.

- **Process Scheduling Algorithms**: The C++ backend implements various process scheduling algorithms, including First-Come, First-Served (FCFS), Round Robin (RR), Shortest Job First (SJF), Priority Scheduling, and an automated scheduler that selects the algorithm based on input characteristics.

- **Performance Metrics Calculation**: After executing the scheduling algorithms, the backend calculates performance metrics such as average turnaround time, average waiting time, and scheduling overhead. These metrics are then passed back to the Electron frontend for display.

- **Execution and Communication**: The backend executes the selected scheduling algorithm on the provided process data. It communicates with the Electron frontend via IPC, sending JSON-formatted scheduling results back to the frontend for visualization and display.

2.4 Interaction Flow

The interaction flow between the frontend and backend of the CPU Scheduler application is as follows:

1. The user selects a scheduling algorithm and enters process details (process ID, arrival time, burst time, priority) through the Electron frontend.

2. Upon clicking the "Run Scheduler" button, the frontend sends an IPC message containing the selected algorithm and process details to the backend.

3. The backend receives the IPC message, processes the scheduling request using the specified algorithm, and computes the necessary performance metrics.

4. After completing the scheduling process, the backend sends a response back to the Electron frontend, including JSON-formatted data containing scheduling results (processes details, performance metrics).

5. The Electron frontend receives the scheduling results, updates the GUI to display the process details and Gantt chart, and shows performance metrics such as average turnaround time, average waiting time, and scheduling overhead.

2.5 Diagram

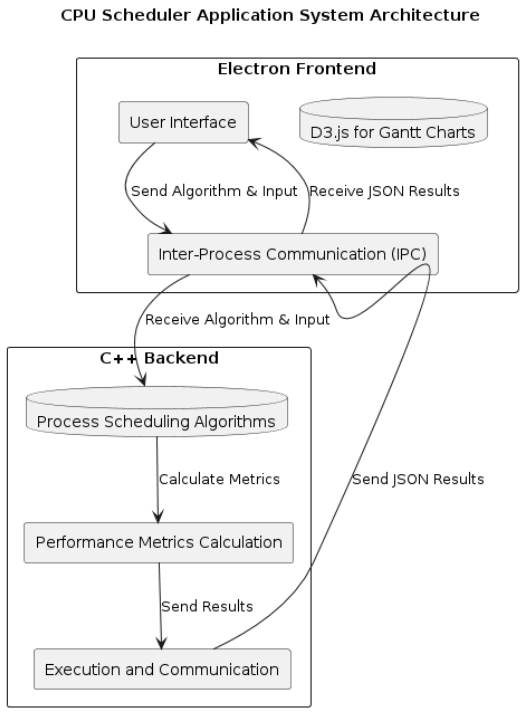


Figure 1: High-level System Architecture of the CPU Scheduler Application

Figure 1 illustrates the high-level system architecture of the CPU Scheduler application, highlighting the interaction flow between the Electron frontend and C++ backend components.

3 Electron Frontend

The Electron frontend of the CPU Scheduler application provides a user-friendly interface for interacting with the scheduling algorithms implemented in the backend. This section outlines the architecture, features, and interaction flow of the Electron frontend.

3.1 Architecture

The Electron frontend architecture consists of several key components:

3.1.1 User Interface (UI)

The UI component provides an interactive platform where users can select scheduling algorithms, input process details, and view scheduling results.

3.1.2 D3.js for Gantt Charts

D3.js is utilized to dynamically generate Gantt charts that visually represent the scheduling of processes over time.

3.1.3 Inter-Process Communication (IPC)

IPC facilitates communication between the Electron frontend and the C++ backend, enabling data exchange and result retrieval.

3.2 Features

The Electron frontend offers several key features to enhance user interaction:

3.2.1 Algorithm Selection

Users can choose from various scheduling algorithms including FCFS, Round Robin, SJF, Priority Scheduling, or Automatic Scheduling (Auto).

3.2.2 Process Input

Provides a text area where users can input details such as Process ID, Arrival Time, Burst Time, and Priority for scheduling.

3.2.3 Run Scheduler Button

Initiates the execution of the selected scheduling algorithm based on the provided input.

3.2.4 Result Display

Displays detailed results including processes, average turnaround time, average waiting time, and scheduling overhead after executing the scheduler.

3.2.5 Gantt Chart Visualization

Utilizes D3.js to create interactive Gantt charts that visually illustrate the scheduling timeline of processes.

3.3 Interaction Flow

The interaction flow of the Electron frontend is as follows:

3.3.1 User Interaction

Users interact with the UI by selecting an algorithm, entering process details, and triggering the scheduler.

3.3.2 IPC Handling

The selected algorithm and process input are sent via IPC to the C++ backend for execution.

3.3.3 Backend Processing

The C++ backend executes the chosen algorithm, computes metrics, and sends back the results via IPC to the Electron frontend.

3.3.4 Result Presentation

The UI updates to display the processed results, including textual details and visualizations (Gantt chart).

4 C++ Backend

The C++ backend of the CPU Scheduler application serves as the core processing engine responsible for executing scheduling algorithms and computing metrics based on provided process data.

4.1 Architecture

The backend architecture is structured to efficiently manage process scheduling and computation tasks:

4.1.1 Process Struct

Defines the `Process` struct containing attributes such as ID, Arrival Time, Burst Time, Priority, Completion Time, Waiting Time, Turnaround Time, and Start Time.

4.1.2 Algorithm Functions

Includes implementations for various scheduling algorithms:

- **First-Come, First-Served (FCFS)**: Executes processes based on their arrival times in a non-preemptive manner.
- **Round Robin (RR)**: Implements a preemptive scheduling algorithm where each process is assigned a fixed time slice (quantum) in a circular queue.
- **Shortest Job First (SJF)**: Schedules processes based on their burst times, either preemptively or non-preemptively.
- **Priority Scheduling**: Assigns priorities to processes and schedules them based on priority levels.
- **Automatic Scheduling (Auto)**: Selects an appropriate algorithm dynamically based on input process attributes to avoid starvation and optimize scheduling.

4.2 Implementation Details

4.2.1 Input Parsing

Utilizes input parsing functions to convert user-provided process details from string format into `Process` structs.

4.2.2 Algorithm Execution

Each scheduling algorithm function processes the list of `Process` structs according to its scheduling logic:

- Calculates completion times, waiting times, and turnaround times for each process.
- Ensures proper handling of process arrival times and quantum slices (for RR).
- Manages scheduling overhead and algorithmic complexities.

4.2.3 Output Generation

Produces JSON-formatted output containing detailed scheduling results, including processes list, average turnaround time, average waiting time, and scheduling overhead.

4.3 Performance Considerations

The backend is optimized for performance and scalability:

4.3.1 Algorithm Selection Logic

The `auto_schedule` function dynamically selects the most suitable scheduling algorithm based on input process characteristics to ensure efficient task scheduling.

4.3.2 Execution Time Metrics

Utilizes `std::chrono` library to measure and report scheduling overhead, providing insights into algorithm efficiency and performance.

5 Integration via IPC

In the CPU Scheduler application, the frontend (Electron) and backend (C++) communicate using IPC mechanisms provided by Electron's `ipcRenderer` and `ipcMain` modules. IPC allows asynchronous communication between the Electron main process and the C++ backend process, facilitating data exchange and coordination of scheduling tasks.

5.1 Electron Frontend

The Electron frontend serves as the user interface layer, implemented using HTML, CSS, and JavaScript. It utilizes the Electron framework to create a windowed desktop application that interacts with the user and communicates with the backend for scheduling operations.

5.1.1 Components

- **Renderer Process**: This is where the frontend code executes within the Electron framework. It includes:
  - **HTML/CSS**: Defines the UI components and styles using HTML for structure and CSS for presentation.
  - **JavaScript**: Handles user interactions, triggers scheduling requests, and updates UI elements dynamically.
  - **IPC Renderer**: Communicates asynchronously with the main process to send scheduling requests and receive results.

5.1.2 Implementation Details

The IPC Renderer uses the `ipcRenderer` module to send messages to the Electron main process (`main.js`) and handle scheduling results asynchronously:

```
const { ipcRenderer } = require('electron');

// Sending scheduling request to main process
ipcRenderer.send('runScheduler', { algorithm, input });

// Handling scheduler result from main process
ipcRenderer.on('schedulerResult', (event, { error, result }) => {
    if (error) {
        // Handle error
    } else {
        // Process and display scheduling result
    }
});
```

5.2 C++ Backend

The C++ backend handles the core scheduling algorithms and computation tasks. It runs as a separate process from the Electron main process and communicates with Electron via IPC using `ipcMain`.

5.2.1 Components

- **Main Process**: Runs the Electron application and manages the backend process.
- **IPC Main**: Listens for messages from the frontend and processes scheduling requests using C++ functions.

5.2.2 Implementation Details

The IPC Main uses the `ipcMain` module to listen for messages from the frontend and execute corresponding backend functionalities:

```
const { ipcMain } = require('electron');
const { spawn } = require('child_process');
const path = require('path');

ipcMain.on('runScheduler', (event, { algorithm, input }) => {
    // Spawn a new instance of the C++ backend process
    const exe = spawn('./scheduler.exe', [algorithm, input]);

    // Handle stdout data from C++ backend
    exe.stdout.on('data', (data) => {
        try {
            let result = JSON.parse(data);
            event.sender.send('schedulerResult', { result });
        } catch (error) {
            event.sender.send('schedulerResult', { error: 'Error parsing JSON' });
        }
    });

    // Handle stderr data from C++ backend
    exe.stderr.on('data', (data) => {
        event.sender.send('schedulerResult', { error: data.toString() });
    });

    // Handle process exit or error
    exe.on('close', (code) => {
        if (code !== 0) {
            event.sender.send('schedulerResult', { error:`${code}$` });
        }
    });
```

```
    exe.on('error', (err) => {
        event.sender.send('schedulerResult', { error: err.message });
    });
});
```

This structured approach ensures efficient communication and coordination between the frontend and backend components of the CPU Scheduler application using IPC mechanisms provided by Electron. Adjust the details as necessary based on your specific implementation and application requirements.

6 Gantt Chart Visualization

6.1 Purpose and Functionality

The Gantt chart serves several purposes in your project:

- **Visualization of Process Execution**: It visually represents when each process starts, how long it runs, and when it completes. This helps in understanding the execution sequence and duration of processes within a scheduling algorithm.
- **Resource Allocation**: It shows how CPU time is allocated among different processes, highlighting periods of CPU utilization and idle times.
- **Performance Analysis**: Users can analyze the efficiency of different scheduling algorithms by observing the distribution of CPU time among processes and identifying potential areas for optimization.

6.2 Implementation Details

In your implementation, the Gantt chart is dynamically generated and updated in the Electron frontend based on the scheduling results received:

- **D3.js Integration**: You use the D3.js library to create the Gantt chart dynamically based on the scheduling data received from the C++ backend.
- **SVG Rendering**: Using D3.js, SVG elements are manipulated to create bars representing each process on the Gantt chart. Each bar's length corresponds to the process's execution time, and its position indicates its start time relative to others.

6.3 User Interaction

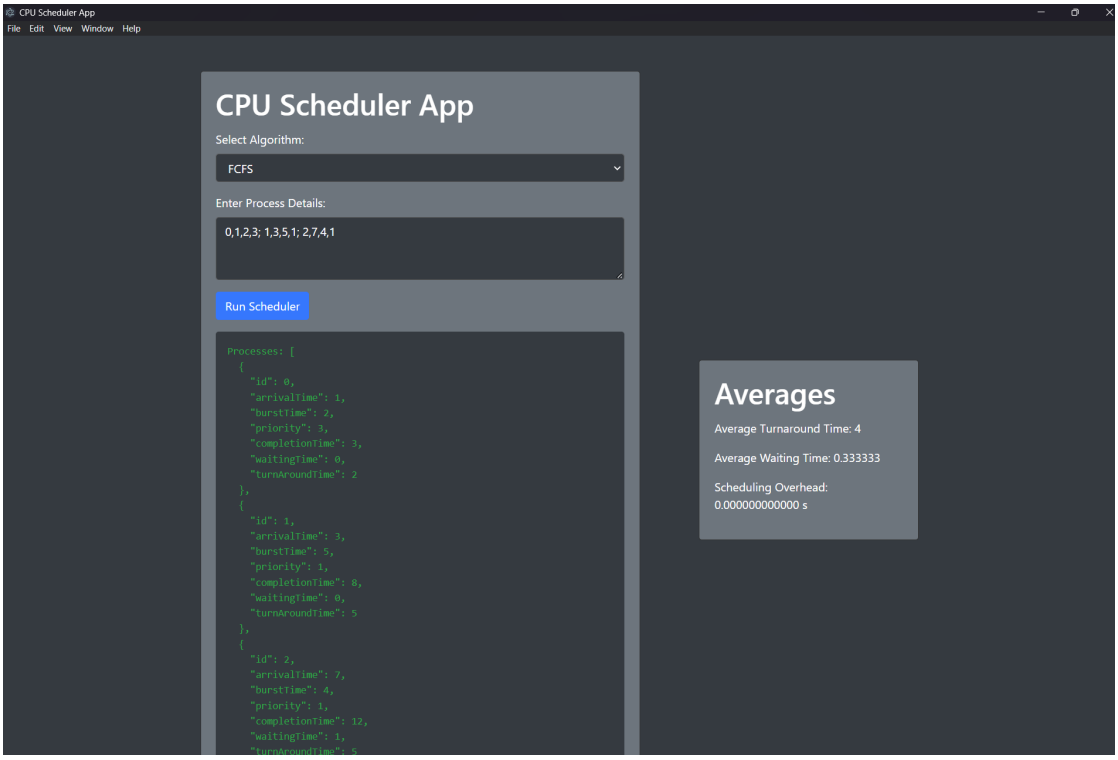
- **Dynamic Updates**: When the user selects a scheduling algorithm and clicks "Run Scheduler," the frontend sends a request to the backend. Upon receiving the results, including the scheduling data for each process, the frontend updates the Gantt chart to reflect the newly scheduled processes.

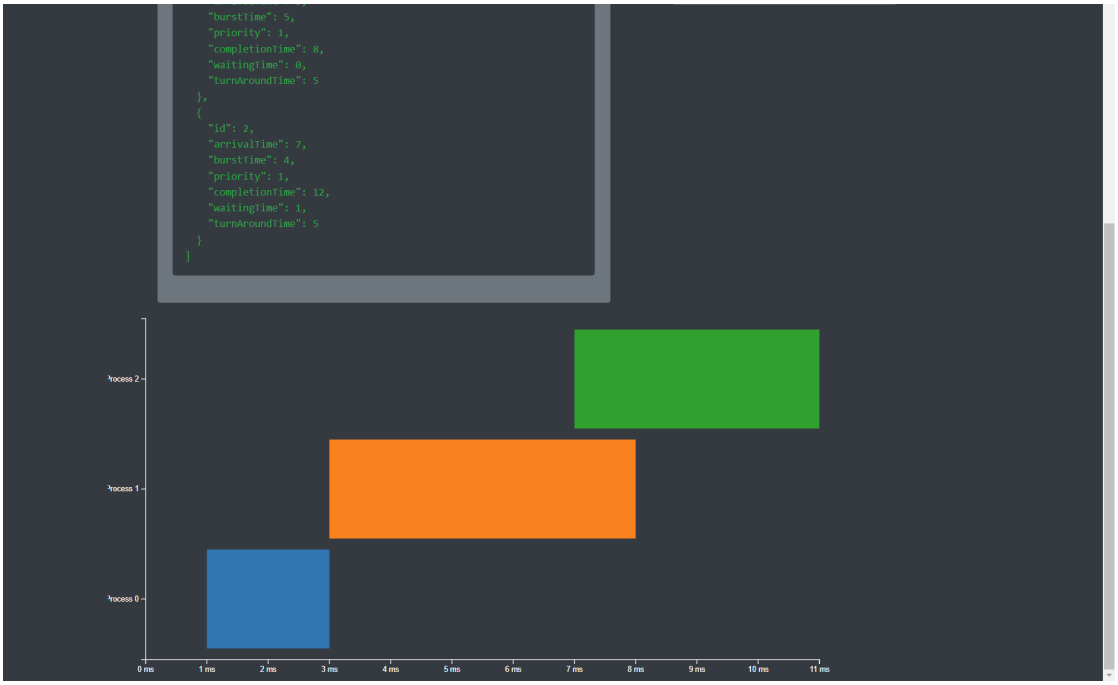
6.4 Benefits

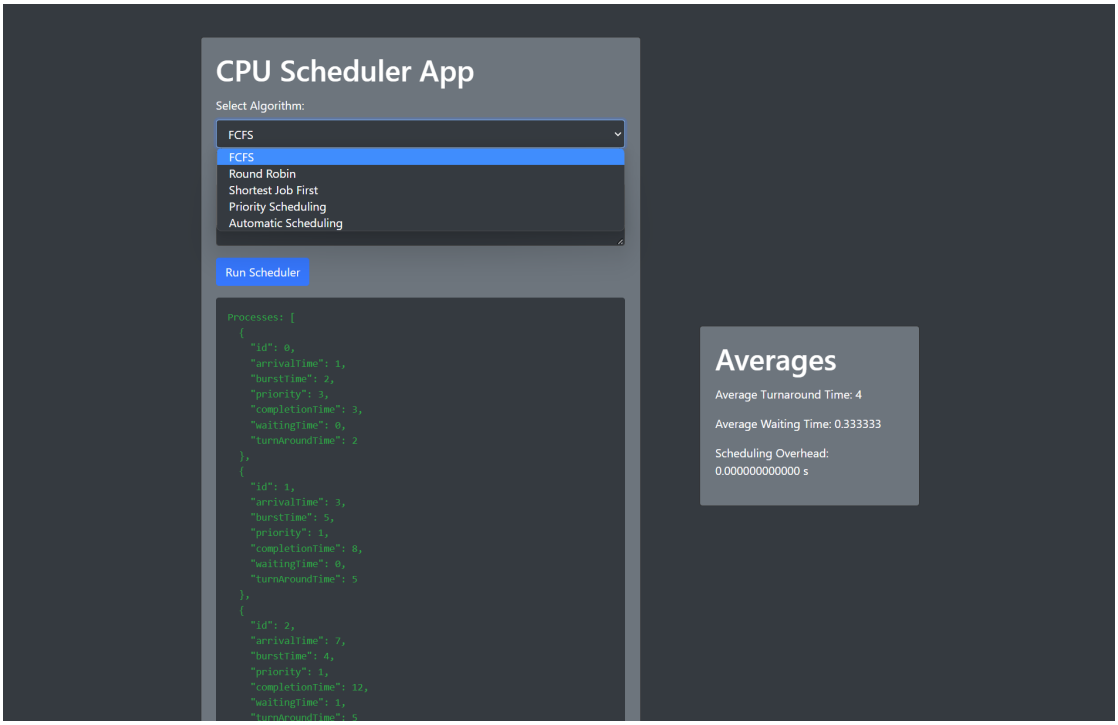
- **Visualization of Algorithms**: The Gantt chart provides a clear visual representation of how different scheduling algorithms (FCFS, Round Robin, SJF, Priority) handle process execution and CPU allocation.

- **Educational Tool**: It serves as an educational tool for users to understand the concepts of CPU scheduling algorithms visually, helping them grasp complex scheduling behaviors more intuitively.

7 Screenshots




"burstTime": 5,
"priority": 1,
"completionTime": 8,
"waitingTime": 0,
"turnAroundTime": 5
},
{
"id": 2,
"arrivalTime": 7,
"burstTime": 4,
"priority": 1,
"completionTime": 12,
"waitingTime": 1,
"turnAroundTime": 5
}
]
rocess 2
rocess 1
rocess 0
0 ms
1 ms
2 ms
3 ms
4 ms
5 ms
6 ms
7 ms
8 ms
9 ms
10 ms
11 ms


CPU Scheduler App
Select Algorithm:
FCFS
FCFS
Round Robin
Shortest Job First
Priority Scheduling
Automatic Scheduling
Run Scheduler
Processes: [
{
"id": 0,
"arrivalTime": 1,
"burstTime": 2,
"priority": 3,
"completionTime": 3,
"waitingTime": 0,
"turnAroundTime": 2
},
{
"id": 1,
"arrivalTime": 3,
"burstTime": 5,
"priority": 1,
"completionTime": 8,
"waitingTime": 0,
"turnAroundTime": 5
},
{
"id": 2,
"arrivalTime": 7,
"burstTime": 4,
"priority": 1,
"completionTime": 12,
"waitingTime": 1,
Averages
Average Turnaround Time: 4
Average Waiting Time: 0.333333
Scheduling Overhead:
0.000000000000 s

8 Code Snippets

```
int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: scheduler.exe <algorithm> <input>" << std::endl;
        return 1;
    }

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

    std::vector<Process> processes = parseInput(input);

    if (algorithm == "FCFS") {
        FCFS(processes);
    } else if (algorithm == "RR") {
        roundRobin(processes, QUANTUM);
    } else if (algorithm == "SJF") {
        shortestJobFirst(processes);
    } else if (algorithm == "Priority") {
        priorityScheduling(processes);
    }
    else if (algorithm == "Auto") {
        auto_schedule(processes);
    } else {
        std::cerr << "Invalid algorithm choice\n";
        return 1;
    }

    double totalTAT = 0;
    double totalWT = 0;
    int n = processes.size();

    for (const auto& process : processes) {
        totalTAT += process.turnAroundTime;
        totalWT += process.waitingTime;
    }

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

```
    std::cout << "{ \"processes\": [";
    for (size_t i = 0; i < processes.size(); ++i) {
        std::cout << "{"
                  << "\"id\":" << processes[i].id << ","
                  << "\"arrivalTime\":" << processes[i].arrivalTime << ","
                  << "\"burstTime\":" << processes[i].burstTime << ","
                  << "\"priority\":" << processes[i].priority << ","
                  << "\"completionTime\":" << processes[i].completionTime << ","
                  << "\"waitingTime\":" << processes[i].waitingTime << ","
                  << "\"turnAroundTime\":" << processes[i].turnAroundTime
                  << "}";
        if (i < processes.size() - 1) {
            std::cout << ",";
        }
    }
    auto chrono_end = std::chrono::steady_clock::now();
    long double schedulingOverhead = 1e-12 *
    std::chrono::duration_cast<std::chrono::microseconds>(chrono_end - chrono_begi
    std::cout << "],"
              << "\"averageTAT\": " << averageTAT << ","
              << "\"averageWT\": " << averageWT << ","
              << "\"schedulingOverhead\": " << schedulingOverhead << "}";
    std::cout.flush();
    return 0;
}
```

I write all the data about the process to the standard output in a json token form where the IPC listeners extracts JSON token and can parse to gather the task's scheduling information and display it to the frontend. We also used the chrono library in std::c++ to find out scheduling overhead or in other words how much time it takes in for noticibly small inputs the overhead is almost always 0 but for larger number of processes (say 100 it took about 10^{-5} seconds which is accurate as a normal c++ program on average can execute 10^8 instructions.

```
const { app, BrowserWindow, ipcMain } = require('electron');
const { spawn } = require('child_process');
const path = require('path');
```

```
let mainWindow;

function createWindow() {
    mainWindow = new BrowserWindow({
        width: 1200,
        height: 900,
        webPreferences: {
            nodeIntegration: true,
            contextIsolation: false
        }
    });

    mainWindow.loadFile('index.html');

}

app.on('ready', createWindow);

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]);
```

Here i used Node to make a system call and create a child process (scheduler.out or scheduler.exe depending on your system) and then extract what it writes to stdout and stderr. This ensures the backend is in C++ and the front end communicates with it via system calls and the frontend is electron js.

```
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 = 'block';
        renderGanttChart(processes);
        resultElement.classList.add('text-success');
        resultElement.classList.remove('text-danger');
    }
});
```

Here I use the concept of inter process communication to achieve fast and efficient data transfer this also decreases coupling and increases modularity of my solution.