- **Introduction**

Name – Sahil Jain

Enrollment Number – 22118064

Branch – Metallurgical And Materials Engineering

**Project Overview** – CPU SCHEDULER project is designed to simulate various CPU scheduling algorithms. CPU Scheduling refers to the concept in operating systems, which determines the order in which processes are executed by the CPU. This project implements several popular scheduling algorithms, including First Come First Serve (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin (RR). The backend of the project is written in C++ and handles the core scheduling logic, while the frontend is developed using HTML, CSS, and JavaScript to provide a user-friendly interface for interacting with the scheduler.

**Objectives –** The primary objectives of the CPU Scheduler project are as follows:

1. **Implementation of Scheduling Algorithms:** Develop and implement the core logic for various CPU scheduling algorithms, including FCFS, SJF, Priority Scheduling, and RR, using C++

2. **Process Simulation**: Simulate the behaviour of processes in a CPU scheduling environment, allowing users to input and visualize different process scenarios.

3. **User Interface:** Implemented a frontend with html , css and javascript for user to input data on website and get the results there.

By achieving these objectives, the project aims to enhance the understanding of CPU scheduling algorithms and their practical implications in operating system design and performance.

# • Project Structure

## Directory Layout

The project is organized into several directories and files, each serving a specific purpose. The structure of the project is as follows:

### Backend

The backend directory contains the core implementation files for the CPU scheduling algorithms, written in C++. This is where the logic for scheduling processes is handled.

- **fcfs.h**: Header file for the First Come First Serve (FCFS) scheduling algorithm. It contains the class definition and function declarations for the FCFS scheduler.
- **priority.h**: Header file for the Priority Scheduling algorithm. It defines the classes and functions needed to prioritize processes based on their priority levels.
- **rr.h**: Header file for the Round Robin (RR) scheduling algorithm. It contains the class definition and function declarations for the RR scheduler.
- **sjf.h**: Header file for the Shortest Job First (SJF) scheduling algorithm. It contains the class definition and function declarations for the SJF scheduler.
- **process.h**: Header file that defines the Process class, which represents a process with attributes such as ID, burst time, arrival time, and priority.
- **scheduler.exe**: The executable file generated after compiling the project. This file can be run to execute the CPU scheduler.
- **main.cpp**: The main entry point of the program. This file initializes the program, processes user inputs, and invokes the scheduling algorithms.

### Frontend

The frontend directory contains the html, css, js files for the web-based interface which were made in an effort to make a user-friendly environment for the user to interact with the scheduler and get results displayed on the web.

- **index.html**: The main HTML file that structures the web interface. It includes elements for user input and displaying results.
- **styles.css**: The CSS file that defines the styling for the web interface, ensuring it is visually appealing and user-friendly.

- **Backend Implementation**

fcfs.h

**Variable Initialization**:

```
void fcfsScheduling(vector<Process> &processes){
    int n = processes.size();
    int totalWaitTime = 0;
    int totalTurnAroundTime = 0;
    int currTime = 0;
```

- Store the number of processes.
- Initialize the total wait time to 0.
- Initialize the total turnaround time to 0.
- Track the current time in the scheduler.

**Sorting Processes**:

```
sort(processes.begin(), processes.end(), [](const Process &a, const Process &b){
    return a.arrivalTime < b.arrivalTime;
});
```

- Sorts the processes based on their arrival times using a lambda function.

**Scheduling Loop**:

```
for(int i = 0; i < n; i++){

    //curr time > arrival time, process has to wait, if not, wait time = 0, so, took max with 0.
    int waitTime = max(0, currTime - processes[i].arrivalTime);
    totalWaitTime = totalWaitTime + waitTime;

    //turnaroundtime = sum of wait time and burst time
    int turnAroundTime = waitTime + processes[i].burstTime;
    totalTurnAroundTime = totalTurnAroundTime + turnAroundTime;

    //end of process execution
    currTime = max(currTime, processes[i].arrivalTime) + processes[i].burstTime;

    cout << "Process " << processes[i].id << ": Wait Time = " << waitTime << ", TurnAround Time = " <<
    turnAroundTime << endl;

}
```

- Iterate over each process.
- Calculate the wait time for the current process. If the current time is greater than the arrival time, the process has to wait. Otherwise, the wait time is 0.
- Add the wait time to the total wait time.
- Calculate the turnaround time, which is the sum of the wait time and burst time.
- Add the turnaround time to the total turnaround time.
- Update the current time to the end time of the process's execution.
- Print the wait time and turnaround time for the current process.

**Average Times**:

```
cout << "Average Wait Time = " << (double)totalWaitTime/n << endl;
cout << "Average TurnAround Time = " << (double)totalTurnAroundTime/n << endl;
```

- Calculate and prints the average wait time.
- Calculate and prints the average turnaround time.

## priority.h

**Variable Initialization**:

```
void priorityScheduling(vector<Process>& processes){
    int n = processes.size();
    priority_queue<Process, vector<Process>, ComparePriority> pq;
    int totalWaitTime = 0;
    int totalTurnAroundTime = 0;
    int currentTime = 0;
    int processIndex = 0;
```

- Store the number of processes.
- Define a priority queue to store processes, ordered by priority.
- Initialize the total wait time to 0.
- Initialize the total turnaround time to 0.
- Track the current time in the scheduler.

- Track the index of the next process to be added to the queue.

**Sorting** Processes:

```
sort(processes.begin(), processes.end(), [](const Process &a, const Process &b){
    return a.arrivalTime < b.arrivalTime;
});
```

- Sorts the processes based on their arrival times using a lambda function.

**Scheduling Loop**:

```
while(!pq.empty() || processIndex < n){
    //push all elements that have arrived by the current time in priority queue
    while(processIndex < n && processes[processIndex].arrivalTime <= currentTime){
        pq.push(processes[processIndex]);
        processIndex++;
    }
    if(!pq.empty()){
        Process currentProcess = pq.top();
        pq.pop();

        int waitTime = currentTime - currentProcess.arrivalTime;
        totalWaitTime += waitTime;

        currentTime += currentProcess.burstTime;

        int turnAroundTime = currentTime - currentProcess.arrivalTime;
        totalTurnAroundTime += turnAroundTime;

        cout << "Process " << currentProcess.id << ": Wait Time = " << waitTime << ", Turnaround Time = "
        << turnAroundTime << endl;
    }
    else{
        // if queue is empty move the current time to the next process's arrival time
        if(processIndex < n){
            currentTime = processes[processIndex].arrivalTime;
        }
    }
```

- Continue until all processes are scheduled.
    - Push all processes that have arrived by the current time into the priority queue.
    - If the priority queue is not empty, it processes the process with the highest priority (lowest priority number).
        - Calculate the wait time for the current process.
        - Add the wait time to the total wait time.
        - Update the current time to the end time of the process's execution.
        - Calculate the turnaround time.
        - Add the turnaround time to the total turnaround time.
        - Print the wait time and turnaround time for the current process.

- If the priority queue is empty, it moves the current time to the next process's arrival time.

**Average Times**:

```
cout << "Average Wait Time = " << (double)totalWaitTime / n << endl;
cout << "Average Turnaround Time = " << (double)totalTurnAroundTime / n << endl;
```

- Calculate and print the average wait time.
- Calculate and print the average turnaround time.

This Priority Scheduling algorithm schedules processes based on their priority levels, with the highest priority process (lowest priority number) being scheduled first. It calculates the wait time and turnaround time for each process and prints these values along with the average wait and turnaround times for all processes. This approach ensures that higher priority processes are given preference, which can be critical in real-time systems where certain tasks must be completed promptly.

Coded the other two algorithms on almost similar pattern of initializing variables, sorting, looping and taking average to generate the desired output.

scheduler.h

```
void InputProcesses(vector<Process> &processes){
    int numProcesses;
    cout << "Enter the number of Processes: ";
    cin >> numProcesses;

    for(int i = 0; i < numProcesses; i++){
        Process p;
        p.id = i+1;

        cout << "Enter burst time for Process " << p.id << ": ";
        cin >> p.burstTime;

        cout << "Enter arrival time for Process " << p.id << ": ";
        cin >> p.arrivalTime;

        cout << "Enter priority for Process " << p.id << ": ";
        cin >> p.priority;

        p.remainingTime = p.burstTime;
        processes.push_back(p);
    }
}
```

**Purpose:** This function is responsible for gathering process information from the user and populating a vector of Process objects.

**Steps:**

- It prompts the user to enter the number of processes (numProcesses).
- It iterates numProcesses times to gather details for each process:
    - Creates a new Process object p.
    - Assigns an ID to p based on the current iteration.
    - Asks for burst time, arrival time, and priority for each process, storing these values in p.
    - Initializes p.remainingTime to p.burstTime.
    - Adds p to the processes vector using processes.push_back(p).

```
void runScheduler(){
    vector<Process> processes;
    InputProcesses(processes);

    int choice;
    cout << "Select Scheduling Algorithm" << endl;
    cout << "1.FCFS" << endl;
    cout << "2.SJF" << endl;
    cout << "3.Round Robin" << endl;
    cout << "4.Priority Scheduling" << endl;
    cout << "ENTER YOUR CHOICE: " ;
    cin >> choice;

    switch(choice){

        case 1:
            fcfsScheduling(processes);
            break;

        case 2:
            sjfScheduling(processes);
            break;

        case 3:
            int timeQuantum;
            cout << "ENTER TIME QUANTUM FOR ROUND ROBIN: ";
            cin >> timeQuantum;
            rrScheduling(processes, timeQuantum);
```

```
            break;

        case 4:
            priorityScheduling(processes);
            break;

        default:
        cout << "INVALID CHOICE!" << endl;
    }
}
```

- **Purpose:** runScheduler orchestrates the scheduling simulation based on user input.
- **Steps:**

- Initializes an empty vector processes.
- Calls InputProcesses(processes) to populate processes with user-entered data.
- Prompts the user to select a scheduling algorithm (choice).
- Uses a switch statement to execute the chosen algorithm:
    - **Case 1 (FCFS)**: Calls fcfsScheduling(processes) to perform First-Come, First-Served scheduling.
    - **Case 2 (SJF)**: Calls sjfScheduling(processes) for Shortest Job First scheduling.
    - **Case 3 (Round Robin)**: Prompts for a time quantum, then calls rrScheduling(processes, timeQuantum) for Round Robin scheduling.
    - **Case 4 (Priority Scheduling)**: Calls priorityScheduling(processes) for Priority Scheduling.
    - **Default:** Prints an error message for an invalid choice.

Together, these functions provide a framework for simulating different CPU scheduling algorithms based on user-defined processes and choices.

## main.cpp

```
1  #include "scheduler.h"
2
3  int main(){
4      runScheduler();
5      return 0;
6  }
```

Calls the runScheduler() function for implementing the function and algorithms according to the input provided by the user as defined in the runScheduler() function.

## • Frontend Implementation

### index.html

```
<html>
<body>
    <form id = "schedulerForm">
        <select id = "algorithm" name = "algorithm">
            <option value = "RR"> Round Robin </option>
            <option value = "Priority"> Priority </option>
        </select><br><br>

        <label for = "timeQuantum"> Time Quantum (for RR): </label>
        <input type = "number" id = "timeQuantum" name = "timeQuantum" min = "1"><br><br>


        <div id = "processes">
            <div class="process">
                <label for="processId1">ID:</label><input type="text" id="processId1" name="processId1"
                class="id" required><br>
                <label for="burstTime1">Burst Time:</label><input type="number" id="burstTime1"
                name="burstTime1" class="burstTime" min="1" required><br>
                <label for="arrivalTime1">Arrival Time:</label><input type="number" id="arrivalTime1"
                name="arrivalTime1" class="arrivalTime" min="0" required><br>
                <label for="priority1">Priority:</label><input type="number" id="priority1" name="priority1"
                class="priority" min="1" required><br>
                <label for="remainingTime1">Remaining Time:</label><input type="number" id="remainingTime1"
                name="remainingTime1" class="remainingTime" min="0"><br>
            </div>
        </div>
        <button type = "button" id = "addProcessButton">Add Process</button><br><br>

        <button type = "submit">Schedule</button>
    </form>
```

- **Head Section**: Contains a <title> tag for the webpage title and a <link> tag to link an external stylesheet (styles.css), which presumably contains additional styling rules not shown in this snippet.
- **Body Section**:
    - **<h1> Heading**: Displays "CPU SCHEDULER".
    - **Form (<form id="schedulerForm">)**:
        - **Scheduling Algorithm Selection**: Uses a <select> dropdown (<select id="algorithm">) where users can choose between FCFS, SJF, Round Robin (RR), and Priority scheduling algorithms.

        - **Time Quantum Input**: An <input> field (<input type="number" id="timeQuantum">) allows users to specify the time quantum for Round Robin scheduling. It has a minimum value set (min="1").
        - **Processes Section ( )**:
            - Initially contains fields for one process ( ), including ID (processId1), Burst Time (burstTime1), Arrival Time (arrivalTime1), Priority (priority1), and Remaining Time (remainingTime1). These are <input> fields of various types (text, number) with minimum values set for numeric inputs.
            - **Add Process Button**: A <button type="button" id="addProcessButton">Add Process</button> allows users to dynamically add more process input fields ( ) as needed.
        - **Submit Button**: The final <button type="submit">Schedule</button> submits the form, triggering the scheduling process.
- **Results Section ( )**: Placeholder where the results of the scheduling simulation will be displayed dynamically.
- **Script**: Includes a <script> tag (<script src="script.js"></script>) to link an external JavaScript file (script.js), indicating that client-side scripting is used to handle form submission and possibly other dynamic interactions.

This HTML structure provides a user-friendly interface for configuring and simulating different CPU scheduling algorithms, allowing users to input process details, select algorithms, and view simulation results within the same webpage.

### styles.css

```css
body{
    font-family: Arial, sans-serif;
}

h1, h2, h3{
    color: #333;
}

form{
    margin-bottom: 20px;
}

label{
    display: inline-block;
    width: 150px;
}

input, select, button{
    margin-bottom:10px;
}

pre{
    background-color: #f8f8f8;
    padding: 10px;
    border: 1px solid #ddd;
}
```

- **body**: Sets the overall font family for the entire document to Arial or a generic sans-serif font.
- **h1, h2, h3**: Defines the text color of headings (<h1>, <h2>, <h3>) to a dark gray (#333).
- **form**: Adds a margin at the bottom of forms to separate them from other content (margin-bottom: 20px).
- **label**: Styles labels associated with form elements. Makes them inline-block elements with a fixed width of 150px, which helps in aligning them neatly next to their corresponding input fields.
- **input, select, button**: Applies consistent spacing below these form elements (margin-bottom: 10px), ensuring they are uniformly spaced vertically.
- **pre**: Styles <pre> elements typically used for displaying preformatted text or code snippets. Sets a light gray background (#f8f8f8), adds padding (10px), and a subtle border (1px solid #ddd) around the element for better readability.

These styles collectively enhance the visual coherence and usability of the HTML form and content. They ensure consistent spacing, readable text, and a clean presentation throughout the CPU scheduler simulation interface.

## script.js

This JavaScript code enhances the functionality of the HTML CPU scheduler interface by adding dynamic behavior and handling form submissions.

```
1   document.addEventListener('DOMContentLoaded', () => {
6       if (schedulerForm) {
7           schedulerForm.addEventListener('submit', async function (e) {
8               e.preventDefault();
9
10              console.log('Form submitted');
11
12              const algorithm = document.getElementById('algorithm').value;
13              const timeQuantum = document.getElementById('timeQuantum').value || 0;
14              const processes = Array.from(document.querySelectorAll('.process')).map(process => {
15                  return {
16                      id: process.querySelector('.id').value,
17                      burstTime: process.querySelector('.burstTime').value,
18                      arrivalTime: process.querySelector('.arrivalTime').value,
19                      priority: process.querySelector('.priority').value,
20                      remainingTime: process.querySelector('.remainingTime').value || 0
21                  };
22              });
```

### DOMContentLoadedEvent Listener:

```
document.addEventListener('DOMContentLoaded', () => {
```

- This function wraps all the code inside it to ensure it runs when the DOM is fully loaded and parsed.

**Variables Initialization**:

```
const schedulerForm = document.getElementById('schedulerForm');
const addProcessButton = document.getElementById('addProcessButton');
const results = document.getElementById('results');
```

- These variables store references to important elements in the HTML document (<form>, "Add Process" button, ).

**Form Submission Handling**:

```
if (schedulerForm) {
    schedulerForm.addEventListener('submit', async function (e) {
        e.preventDefault();

        console.log('Form submitted');
```

- This event listener listens for form submissions (submit event) on the schedulerForm.
- e.preventDefault() prevents the default form submission behavior, allowing JavaScript to handle the form submission asynchronously.

**Data Collection**:

```
    const algorithm = document.getElementById('algorithm').value;
    const timeQuantum = document.getElementById('timeQuantum').value || 0;
    const processes = Array.from(document.querySelectorAll('.process')).map(process => {
        return {
            id: process.querySelector('.id').value,
            burstTime: process.querySelector('.burstTime').value,
            arrivalTime: process.querySelector('.arrivalTime').value,
            priority: process.querySelector('.priority').value,
            remainingTime: process.querySelector('.remainingTime').value || 0
        };
    });
```

- Inside the form submission event handler:
    - Collects the selected algorithm (algorithm) and time quantum (timeQuantum) from the form.
    - Collects details of each process dynamically added by the user (processes). It iterates over each .process element in the form and retrieves values from input fields (id, burstTime, arrivalTime, priority, remainingTime).

```
try {
    const response = await fetch('/schedule', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ algorithm, timeQuantum, processes })
    });

    if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
    }

    const result = await response.json();
    console.log('Received result:', result);

    if (results) {
        results.textContent = result.output;
    }
} catch (error) {
    console.error('Error:', error);
    if (results) {
        results.textContent = `Error: ${error.message}`;
    }
}
```

**Response Handling**:

- const result = await response.json();
- Parses the JSON response from the server.
- Updates the results element with the output from the server (result.output).

**Error Handling**:

- Catches any errors that occur during the fetch request or processing of the response.
- Logs errors to the console and displays an error message in the results element.

```
    if (addProcessButton) {
        addProcessButton.addEventListener('click', addProcess);
    }
});
```

**Add Process Button Event Listener**:

- addProcessButton.addEventListener('click', addProcess);

- Listens for clicks on the "Add Process" button (addProcessButton).
- Calls the addProcess function to dynamically add a new process input section ( ) to the form.

**addProcess Function**:

- Dynamically adds HTML for a new process input section ( ) each time the button is clicked.
- Uses processCount to generate unique IDs

This JavaScript code enhances the HTML CPU scheduler interface by making it interactive and dynamic, allowing users to add processes on-the-fly and submit scheduling requests to a server-side endpoint for computation and results display.

## server.js

The server.js file provided sets up a backend server using Node.js and Express to handle requests from the frontend for simulating CPU scheduling using a C++ program (scheduler.exe).

### Dependencies

```
const express = require('express');
const path = require('path');
const bodyParser = require('body-parser');
const { execFile } = require('child_process');
```

### Express App Setup

```
const app = express();
const schedulerPath = path.join(__dirname, 'scheduler.exe');
```

Creates an instance of Express to handle HTTP requests.

## Middleware Setup

```
app.use(bodyParser.json());
app.use(express.static(path.join(__dirname, 'frontend')));
```

Middleware to parse JSON-encoded bodies of incoming requests, making req.body accessible as a JSON object.

## POST Endpoint /schedule

```
app.post('/schedule', (req, res) => {
    const { algorithm, timeQuantum, processes } = req.body;
    console.log('Received data:', { algorithm, timeQuantum, processes });

    // Convert processes array to arguments for C++ program
    let args = [algorithm, timeQuantum];
    processes.forEach(process => {
        args.push(process.id, process.burstTime, process.arrivalTime, process.priority, process.remainingTime);
    });

    console.log('Input arguments for scheduler:', args);

    execFile(schedulerPath, args, (error, stdout, stderr) => {
        if (error) {
            console.error('Exec error:', error);
            console.error('Stderr:', stderr);
            res.status(500).send(`Error: ${stderr}`);
            return;
        }
        console.log('Stdout:', stdout);
        res.send(stdout);
    });
});
```

- Defines a POST route /schedule that expects JSON data containing algorithm, timeQuantum, and processes from the frontend.
- Logs received data (algorithm, timeQuantum, processes) to the console.
- Constructs an array args to pass arguments to scheduler.exe based on the received data.
- Uses execFile to execute scheduler.exe with the constructed arguments.
- Handles the result of the execution:
- If there's an error (error), logs it along with stderr output and sends a 500 status response with the error message.
- If successful, logs stdout output and sends it as the response (res.send(stdout)).

## Server Listening

```
app.listen(3000, () => {
    console.log('Server running at http://localhost:3000');
});
```

Starts the server on port 3000 (http://localhost:3000) and Logs a message to indicate the server is running and accessible.

This server.js file sets up a backend server using Express to handle CPU scheduling requests from a frontend interface. It parses incoming JSON data, prepares arguments for a C++ program (scheduler.exe), executes it, and sends back the results to the client. This setup allows for dynamic scheduling simulations based on user input through a web interface.

## • Conclusion

So, this project integrates a CPU scheduler where the core logic and algorithms reside in C++ files with separate header files defining the algorithms. The frontend interface, crafted using HTML, CSS, and JavaScript, facilitates user interaction by allowing dynamic input of scheduling parameters and process details. The backend, orchestrated by server.js in Node.js, acts as a bridge between the frontend and the C++ backend, orchestrating the execution of scheduling simulations using the C++ logic. This architectural design not only enhances user engagement and interface responsiveness but also underscores the seamless integration of diverse technologies to manage and execute intricate computational tasks like CPU scheduling with efficiency and clarity.

Thanks!