CPU SCHEDULER

Project description: This project is a CPU SCHEDULER. It allows the users to stimulate different scheduling algorithms and analyze the performance of the particular selected algorithm. Also, it suggests the optimal algorithm for a particular set of processes.

How to run the project:

Step 1: Install Node.js

If you haven't already, download and install Node.js for your operating system.

Step 2: Install project dependencies

Navigate to your project directory in the terminal and run the following command to install the required dependencies:

- npm install express

Step 3: Compile the C++ code

Compile scheduler.cpp into an executable file. Use a C++ compiler like g++:

- g++ -o scheduler.exe scheduler.cpp

Step 4: Run the Express server

Start the Express server by running the following command in your project directory:

node server.js

Step 5: Access the simulator

Open a web browser and go to http://localhost:3000 to access the CPU scheduler simulator.

Step 6: Simulate scheduling

Fill in the details of the processes, select a scheduling algorithm, and click "Simulate" to run the simulation. The results will be displayed on the webpage.

Internal working of the project:

1. Client-Side (Frontend):
    - The user interacts with the frontend to input the number of processes, arrival time, burst time, priority, and scheduling algorithm details.
    - This information is sent to the server for simulation.
2. Server-Side (Backend):
    - The server receives the simulation request from the client and prepares an input file (input.txt) containing the process details.
    - It then compiles the scheduler.cpp file into an executable (scheduler.exe) using the g++ compiler.
    - The compiled executable is executed with input redirection from input.txt.
    - The output of the simulation is parsed and sent back to the client for display.
3. Important Code Snippets from scheduler.cpp:

```
11  struct process {
12      int pid;
13      int arrivaltime;
14      int bursttime;
15      int priority;
16      int remainingtime;
17      int waitingtime;
18      int turnaroundtime;
19      int completiontime;
20  };
```

- Defines a structure to hold process information.

```
22  bool comparearrival(process a, process b) {
23      return a.arrivaltime < b.arrivaltime;
24  }
25
26  bool comparepid(process a, process b) {
27      return a.pid < b.pid;
28  }
29
```

- This function is used to compare two process structures based on their arrival times and another one based on process ids.
- It returns true if the arrival time of process a is less than that of process b, otherwise it returns false. Another one returns true if the PID of process a is less than that of process b, otherwise it returns false.

```
void calculatefcfs(process processes[], int n) {
    int currenttime = 0;
    sort(processes, processes + n, comparearrival);
    for (int i = 0; i < n; i++) {
        if (currenttime < processes[i].arrivaltime) {
            currenttime = processes[i].arrivaltime;
        }
        processes[i].completiontime = currenttime + processes[i].bursttime;
        processes[i].turnaroundtime = processes[i].completiontime - processes[i].arrivaltime;
        processes[i].waitingtime = processes[i].turnaroundtime - processes[i].bursttime;
        currenttime += processes[i].bursttime;
    }
    sort(processes, processes + n, comparepid);
}
```

- Implements the FCFS scheduling algorithm. Similarly other functions implement their respective algorithms.

```
void suggestoptimalalgorithm(process processes[], int n, bool preemptive, int quantum) {
    int optimalalgorithm = 1;
    double optimalaveragetime = DBL_MAX;
    for (int algorithm = 1; algorithm <= 4; algorithm++) {
        process tempprocesses[n];
        copy(processes, processes + n, tempprocesses);
        calculatetimes(tempprocesses, n, algorithm, preemptive, quantum);
        double averagetime = calculateaveragetime(tempprocesses, n, false);
        if (averagetime < optimalaveragetime) {
            optimalaveragetime = averagetime;
            optimalalgorithm = algorithm;
        }
    }
    cout << "\nSuggested Optimal Algorithm: ";
    switch (optimalalgorithm) {
        case 1:
            cout << "First-Come, First-Served (FCFS)\n";
            break;
        case 2:
            cout << "Shortest Job First (SJF)\n";
            break;
        case 3:
            cout << "Priority Scheduling\n";
            break;
        case 4:
            cout << "Round Robin\n";
            break;
    }

    process optimalprocesses[n];
    copy(processes, processes + n, optimalprocesses);
    calculatetimes(optimalprocesses, n, optimalalgorithm, preemptive, quantum);
    printresults(optimalprocesses, n);
    double averagewaitingtime = calculateaveragetime(optimalprocesses, n, false);
    double averageturnaroundtime = calculateaveragetime(optimalprocesses, n, true);
    cout << "\nAverage Waiting Time for Optimal Algorithm: " << fixed << setprecision(2) << averagewaitingtime << "\n";
    cout << "Average Turnaround Time for Optimal Algorithm: " << fixed << setprecision(2) << averageturnaroundtime << "\n";
```

- Suggests the optimal algorithm which performs the best for chosen set of processes by the user. It outputs the optimal avg.waiting and avg.turnaround time for the chosen set of processes.

```cpp
void calculatetimes(process processes[], int n, int algorithm, bool preemptive, int quantum) {
    for (int i = 0; i < n; i++) {
        processes[i].remainingtime = processes[i].bursttime;
    }
    switch (algorithm) {
        case 1:
            calculatefcfs(processes, n);
            break;
        case 2:
            if (preemptive) {
                calculatepreemptivesjf(processes, n);
            } else {
                calculatenonpreemptivesjf(processes, n);
            }
            break;
        case 3:
            if (preemptive) {
                calculatepreemptivepriority(processes, n);
            } else {
                calculatenonpreemptivepriority(processes, n);
            }
            break;
        case 4:
            calculateroundrobin(processes, n, quantum);
            break;
        default:
            cout << "Invalid algorithm selection!" << endl;
            return;
    }
}
```

- The calculatetimes function initializes the remaining times for all processes and selects the scheduling algorithm based on the provided parameters. It calls the appropriate function for FCFS, SJF, priority scheduling, or round-robin, handling preemptive and non-preemptive cases as needed.

➢ Some other code snippets give the information about the output. Which are quite basic to understand…..

Learning Takeaways from the Project:

1. Integration of Frontend and Backend: I learned how to integrate C++ code with a web server using Node.js, allowing a web interface to interact with backend processes and execute C++ programs.

2. Understanding Scheduling Algorithms: Working on the scheduler.cpp file deepened my understanding of various CPU scheduling algorithms like FCFS, SJF, priority scheduling, and round-robin, and how to implement and test them.
3. Web Development: I enhanced my web development skills by creating a user-friendly frontend interface using HTML, CSS, and JavaScript, which communicates effectively with the backend server.

➢ In some situations I used Chatgpt:

- To make my cpp code look readable.
- I was having some confusions in integrating the front end and backend using node.js
- I encountered many errors in between such as path/directory related errors. For that error handling I used chatgpt to encounter it.

Resources and References used:

- I learned about CPU SCHEDULING and the scheduling algorithms from youtube. The playlist which was sent to us in the discord.
- I was aware of html and some basics of css earlier. But I have learnt javascript and node.js basics again from youtube. I enhanced the codes with the help of chatgpt.
- ChatGPT for assistance with error handling, integration of frontend and backend, and overall project guidance.