Project README

Overview

This project demonstrates the implementation of various CPU scheduling algorithms using C++ and provides a web interface using Flask for interactive scheduling and visualization of Gantt charts. The following scheduling algorithms are implemented:

1. First-Come, First-Served (FCFS)
2. Shortest Job First (SJF)
3. Shortest Remaining Time First (SRTF)
4. Round Robin (with dynamic quantum estimation)

Files

The project comprises four main files:

1. `main.cpp` - Implements the scheduling algorithms and predicts the best algorithm based on task characteristics.
2. `server.py` - Flask application that handles scheduling requests, executes the C++ program, and generates Gantt charts.
3. `index.html` - Frontend interface for inputting processes and displaying the scheduling results.
4. `script.js` - JavaScript file that handles frontend interactions and sends requests to the Flask backend.

Detailed Description

1. main.cpp

This C++ file includes the implementation of the scheduling algorithms, prediction logic for the best scheduling algorithm, and functions to calculate average waiting and turnaround times.

Key Components

- **Task Class**: Represents a task with attributes such as task ID, burst time, arrival time, remaining time, waiting time, and turnaround time.
- **Execution Class**: Represents the execution of a task with attributes such as task ID, start time, and end time.
- **compareByArrival Function**: Comparator function for sorting tasks by their arrival time.
- **predictBestAlgorithm Function**: Predicts the best scheduling algorithm based on task characteristics.
- **printAverageTimes Function**: Calculates and prints the average waiting and turnaround times.
- **printGanttChart Function**: Generates and prints the Gantt chart for task executions.
- **Scheduling Functions**: Implements FCFS, SJF, SRTF, and Round Robin algorithms.

Usage

The `main.cpp` file reads task data from `input.txt`, predicts the best scheduling algorithm, and writes the results to `output.txt`. It also runs all scheduling algorithms and prints their respective average times and Gantt charts.

2. server.py

This Python file sets up a Flask web server to handle HTTP requests for scheduling and visualization.

Key Components

- **generate_gantt_chart Function**: Parses the output data to generate a Gantt chart using Matplotlib.
- **Flask Routes**: Defines routes for serving the frontend and handling scheduling requests.
  - `/`: Serves the `index.html` file.
  - `/schedule`: Handles scheduling requests by compiling and running the C++ program, and returning the results along with Gantt chart URLs.

3. index.html

This HTML file provides the frontend interface for the user to input processes and view the scheduling results.

Key Components

- **Form for Inputting Processes**: Allows users to add multiple processes with fields for PID, burst time, and arrival time.
- **Output Section**: Displays the scheduling results and Gantt charts.

4. script.js

This JavaScript file handles user interactions on the frontend and sends the scheduling requests to the Flask backend.

Key Components

- **Event Listener for Adding Processes**: Dynamically adds input fields for new processes.
- **Form Submission Handling**: Collects process data, sends it to the backend, and displays the results.

Setup and Execution

Prerequisites

- Python setup in VS CODE
- Flask should be installed
- g++ compiler
- Matplotlib (for generating Gantt charts)

Steps to Run the Project

1. **Compile the C++ Program**:

   ```
   g++ main.cpp -o scheduling
   ```

2. **Setup Python development environment in your code editor(say VS code)**

3. **Run the Flask Application**:

- Run these commands in terminal:

```
pip install flask
python server.py
```

- Follow the link given in the terminal (Ctrl + link).

5. **Access the Frontend Interface**: Open a web browser and navigate to `http://127.0.0.1:5000/`.

6. **Input Processes**:

   - Enter the PID, burst time, and arrival time for each process(note that burst time must be positive).
   - Click "Add Process" to add more processes.
   - Click "Schedule" to perform scheduling and view the results.

7. **If all these is not working in your setup you can create an `input.txt` file and run the `main.cpp` and see the `output.txt`.**

Scheduling Algorithms

Let's delve into each scheduling algorithm implemented in the `main.cpp` file, along with their theoretical background and code implementation details.

1. First-Come, First-Served (FCFS) Scheduling

Theory

FCFS is the simplest scheduling algorithm where processes are executed in the order they arrive in the ready queue. It is non-preemptive, meaning once a process starts executing, it runs to completion. It suffers from the "convoy effect," where shorter processes may get delayed behind longer ones that arrive earlier.

Implementation Details

- **Function**: `void fcfs(std::vector<Task> tasks, std::ofstream& outputFile)`
- **Description**: This function implements FCFS by sorting tasks based on their arrival time and then executing them sequentially.
- **Key Steps**:
  - Sort tasks by arrival time.
  - Execute each task in the order they appear, calculating waiting and turnaround times.
  - Print average waiting and turnaround times.
  - Generate and print the Gantt chart of task executions.

Code Explanation

```
void fcfs(std::vector<Task> tasks, std::ofstream& outputFile) {
    std::sort(tasks.begin(), tasks.end(), compareByArrival);

    std::vector<Execution> executions;
    int currentTime = 0;

    for (int i = 0; i < tasks.size(); ++i) {
        if (currentTime < tasks[i].arrival) {
            currentTime = tasks[i].arrival;
        }
        tasks[i].wait = std::max(0, currentTime - tasks[i].arrival);
        executions.push_back(Execution{tasks[i].id, currentTime, currentTime + tasks[i].burst});
        currentTime += tasks[i].burst;
    }

    // Print average times and Gantt chart
    outputFile << "FCFS Scheduling:" << std::endl;
    printAverageTimes(tasks, outputFile);
    printGanttChart(executions, outputFile);
}
```

- **Explanation**:
  - Tasks are sorted based on arrival time using the `compareByArrival` function.
  - `currentTime` is updated to ensure tasks start at their arrival time if the CPU is idle.
  - Waiting time is calculated as the difference between `currentTime` and arrival time.
  - `executions` vector records each task's execution details for generating the Gantt chart.
  - Finally, average times and the Gantt chart are printed to the output file.

2. Shortest Job First (SJF) Scheduling

Theory

SJF scheduling selects the process with the smallest burst time next, aiming to minimize average waiting time. It can be preemptive (Shortest Remaining Time First, SRTF) or non-preemptive (when all burst times are known in advance).

Implementation Details

- **Function**: `void sjf(std::vector<Task> tasks, std::ofstream& outputFile)`
- **Description**: This function implements non-preemptive SJF by selecting the task with the shortest burst time that has arrived and is ready to execute.
- **Key Steps**:
  - Sort tasks by arrival time.
  - Execute each task with the shortest burst time first.
  - Calculate waiting and turnaround times.
  - Print average waiting and turnaround times.
  - Generate and print the Gantt chart of task executions.

Code Explanation

```
void sjf(std::vector<Task> tasks, std::ofstream& outputFile) {
    std::sort(tasks.begin(), tasks.end(), compareByArrival);

    std::vector<Execution> executions;
    int currentTime = 0;
    int completedTasks = 0;
    int n = tasks.size();
    std::vector<bool> isTaskCompleted(n, false);

    while (completedTasks < n) {
        int shortestJobIndex = -1;
        int shortestBurstTime = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (tasks[i].arrival <= currentTime && !isTaskCompleted[i] && tasks[i].burst < shortestBurstTime) {
                shortestBurstTime = tasks[i].burst;
                shortestJobIndex = i;
            }
        }

        if (shortestJobIndex == -1) {
            currentTime++;
            continue;
        }

        Task& t = tasks[shortestJobIndex];
        t.wait = std::max(0, currentTime - t.arrival);
        executions.push_back(Execution(t.id, currentTime, currentTime + t.burst));
        currentTime += t.burst;
        t.turnaround = t.wait + t.burst;
        isTaskCompleted[shortestJobIndex] = true;
        completedTasks++;
    }

    outputFile << "SJF Scheduling:" << std::endl;
    printAverageTimes(tasks, outputFile);
    printGanttChart(executions, outputFile);
}
```

- **Explanation**:
    - Tasks are sorted based on arrival time using `compareByArrival`.
    - A while loop executes tasks based on their burst times, selecting the shortest available task.
    - Waiting time is computed as the difference between `currentTime` and arrival time.
    - `executions` records task execution details for the Gantt chart.
    - Average times and the Gantt chart are printed to the output file.

3. Shortest Remaining Time First (SRTF) Scheduling

Theory

SRTF is a preemptive version of SJF where the scheduler selects the process with the smallest remaining burst time whenever a new process arrives or the current process completes.

Implementation Details

- **Function**: `void srtf(std::vector<Task> tasks, std::ofstream& outputFile)`
- **Description**: This function implements preemptive SRTF by continuously selecting the task with the shortest remaining burst time until all tasks are completed.
- **Key Steps**:
    - Sort tasks by arrival time.
    - Continuously select the task with the shortest remaining time.
    - Calculate waiting and turnaround times dynamically.
    - Print average waiting and turnaround times.
    - Generate and print the Gantt chart of task executions.

Code Explanation

```
void srtf(std::vector<Task> tasks, std::ofstream& outputFile) {
    int n = tasks.size();
    std::vector<Execution> executions;
    int currentTime = 0;
    int completed = 0;
    int shortestIndex = 0;
    int shortestRemainingTime = INT_MAX;
    bool check = false;

    while (completed != n) {
        for (int i = 0; i < n; ++i) {
            if (tasks[i].arrival <= currentTime && tasks[i].remaining > 0) {
                if (tasks[i].remaining < shortestRemainingTime) {
                    shortestRemainingTime = tasks[i].remaining;
                    shortestIndex = i;
                    check = true;
                }
            }
        }

        if (!check) {
            currentTime++;
            continue;
        }

        // Execute the task for 1 unit of time
        executions.push_back(Execution(tasks[shortestIndex].id, currentTime, currentTime + 1));
        tasks[shortestIndex].remaining--;

        shortestRemainingTime = tasks[shortestIndex].remaining;
        if (shortestRemainingTime == 0) {
            shortestRemainingTime = INT_MAX;
        }

        // Check if the task has completed
        if (tasks[shortestIndex].remaining == 0) {
            completed++;
            check = false;

            tasks[shortestIndex].wait = std::max(0, currentTime + 1 - tasks[shortestIndex].arrival - tasks[shortestIndex].burst);
            tasks[shortestIndex].turnaround = tasks[shortestIndex].wait + tasks[shortestIndex].burst;
        }

        currentTime++;
    }

    outputFile << "SRTF Scheduling:" << std::endl;
    printAverageTimes(tasks, outputFile);
    printGanttChart(executions, outputFile);
}
```

- **Explanation**:
    - Tasks are sorted based on arrival time.
    - A while loop continuously selects the task with the shortest remaining burst time.
    - Tasks are preempted and executed in units of time until completion.
    - Waiting and turnaround times are dynamically updated.
    - Average times and the Gantt chart are printed to the output file.

4. Round Robin (with dynamic quantum estimation)

Theory

Round Robin (RR) scheduling assigns a fixed time unit (quantum) per process. If a process doesn't complete within one quantum, it's preempted and added to the end of the ready queue. Dynamic quantum estimation adjusts the quantum dynamically based on burst times.

Implementation Details

- **Function**: `void roundRobin(std::vector<Task> tasks, int quantum, std::ofstream& outputFile)`
- **Description**: This function implements RR with dynamic quantum estimation by allocating a fixed time slice to each task and adjusting the quantum based on burst times.
- **Key Steps**:
    - Initialize tasks and arrival times.
    - Allocate time slices based on the quantum or remaining burst time.
    - Calculate waiting and turnaround times.
    - Print average waiting and turnaround times.
    - Generate and print the Gantt chart of task executions.

Code Explanation

```
void roundRobin(vector<Task> tasks, int quantum, ofstream& outputFile) {
    vector<Execution> executions;
    int currentTime = 0;
    queue<int> readyQueue;
    int completed = 0;

    for (auto& t : tasks) {
        t.remaining = t.burst;
    }

    int n = tasks.size();
    int arrival[n];
    for (int i = 0; i < n; i++) {
        arrival[i] = tasks[i].arrival;
    }

    while (completed != tasks.size()) {
        for (int i = 0; i < n; i++) {
            if (arrival[i] <= currentTime && arrival[i] != -1) {
                readyQueue.push(i);
                arrival[i] = -1;
            }
        }

        if (readyQueue.empty()) {
            currentTime++;
            continue;
        }

        int current = readyQueue.front();
        readyQueue.pop();

        int timeSlice = min(quantum, tasks[current].remaining);
        executions.push_back(Execution(tasks[current].id, currentTime, currentTime + timeSlice));
        tasks[current].remaining -= timeSlice;
        currentTime += timeSlice;

        for (int i = 0; i < n; i++) {
            if (arrival[i] <= currentTime && arrival[i] != -1) {
                readyQueue.push(i);
                arrival[i] = -1;
            }
        }

        if (tasks[current].remaining > 0) {
            readyQueue.push(current);
        } else {
            completed++;
            tasks[current].wait = currentTime - tasks[current].arrival - tasks[current].burst;
        }
    }

    outputFile << "Round Robin Scheduling (Dynamic Quantum):" << endl;
    outputFile << "Estimated Quantum Time: " << quantum << endl;
    printAverageTimes(tasks, outputFile);
    printGanttChart(executions, outputFile);
}
```

Explanation (continued):

- **Execution Logic**:
    - The `readyQueue` stores indices of tasks ready to be executed.
    - Processes are executed based on the specified quantum or the remaining burst time, whichever is smaller.

    - Each execution is recorded in the `executions` vector for generating the Gantt chart.
    - After execution, if the process is completed (i.e., its remaining time is zero), its waiting and turnaround times are calculated.
    - If a process isn't completed within the quantum, it's preempted and added back to the end of the ready queue for future execution.

- **Completion Condition**:
    - The loop continues until all tasks are completed (`completed == tasks.size()`).

- **Output**:
    - Finally, the function prints the average waiting and turnaround times, along with the Gantt chart, to the output file.

Summary of Scheduling Algorithms

- **FCFS (First-Come, First-Served)**:
    - **Theory**: Executes tasks in the order they arrive.
    - **Code**: Sorts tasks by arrival time and processes them sequentially.

- **SJF (Shortest Job First)**:
    - **Theory**: Executes the shortest burst time task first to minimize waiting time.
    - **Code**: Dynamically selects the task with the shortest remaining burst time until all tasks are completed.

- **SRTF (Shortest Remaining Time First)**:
    - **Theory**: Preemptively selects the task with the shortest remaining burst time.
    - **Code**: Continuously selects and executes the task with the shortest remaining time until all tasks are completed.

- **Round Robin (RR)**:
    - **Theory**: Allocates a fixed time quantum per task, preempting and resuming tasks in a circular queue fashion.
    - **Code**: Executes tasks in round-robin fashion, adjusting the quantum dynamically based on burst times.

Each scheduling algorithm has its advantages and disadvantages depending on the workload characteristics (e.g., CPU-bound vs. I/O-bound tasks). The implementation details provided give a comprehensive understanding of how each algorithm operates and how they can be compared in terms of performance metrics such as average waiting and turnaround times.