CPU Scheduling Web Application

This project is a web application that allows users to input CPU scheduling parameters through a web form, submit them to a server, process them using a C++ program, and display the results back to the user. This document provides an in-depth explanation of various CPU scheduling algorithms implemented in a scheduler program using C++. Each algorithm manages the execution of processes (jobs) in different ways to optimize system performance based on specific criteria.

Technologies Used

- **Frontend**: HTML, JavaScript (with Fetch API), CSS
- **Backend**: Python (Flask)
- **Integration**: Subprocess module for executing a C++ program

CPU Scheduling Algorithms - README

Table of Contents

Introduction to CPU Scheduling

CPU scheduling is a fundamental aspect of operating systems where multiple processes compete for CPU time. The scheduler determines the order and duration of execution for these processes, aiming to optimize system performance metrics such as response time, throughput, and fairness.

First-Come, First-Served (FCFS)

Algorithm Overview:

FCFS is the simplest scheduling algorithm where jobs are executed in the order they arrive. It operates on the principle of a queue: the first job to arrive is the first to be executed.

Logic Behind the Algorithm:

- **Sorting by Arrival Time:** Jobs are sorted based on their arrival times to simulate the order they arrive in the system.
- **Execution:** The CPU executes each job sequentially, starting from the earliest arrival time to the latest.
- **Waiting Time Calculation:** The waiting time for each job is calculated as the difference between its start time and arrival time.

Merits:

- Easy to implement and understand.

- Ensures fairness as jobs are executed in the order of arrival.

Demerits:

- Can lead to high average waiting times, especially if longer jobs arrive first (convoy effect).
- Inefficient for scenarios with a mix of short and long jobs.

Best Use:

FCFS is suitable when all jobs have similar burst times and fairness is prioritized over minimizing waiting times. It's commonly used in batch processing systems.

Example Implementation and Explanation:

```
void fcfs(vector<Job> jobs, ofstream& outputFile) {
    vector<Execution> executions;
    int time_marker = 0;

    // Sort jobs by arrival time
    sort(jobs.begin(), jobs.end(), SortByArrival);

    for (auto& j : jobs) {
        // Ensure jobs start at their arrival time or later
        if (time_marker < j.arrival) {
            time_marker = j.arrival;
        }

        // Calculate waiting time
        j.wait = max(0, time_marker - j.arrival);

        // Record job execution details
        executions.push_back(Execution(j.id, time_marker, time_marker + j.burst));

        // Update current time marker
        time_marker += j.burst;
    }

    // Calculate turnaround time
    calculateWaitAndTurnaround(jobs);

    // Output results
    outputFile << "FCFS Scheduling:" << endl;
    printAverageTimes(jobs, outputFile);
    printGanttChart(executions, jobs, outputFile);
}
```

Explanation:

- **Sorting:** Jobs are sorted using `SortByArrival`, a custom comparator that sorts jobs based on their arrival times.

- **Execution Loop:** Each job's waiting time (`j.wait`) is calculated based on when it can start (either immediately if the CPU is idle or after the previous job completes).

- **Turnaround Time:** After calculating waiting times for all jobs, `calculateWaitAndTurnaround` computes the turnaround time (`j.turnaround = j.wait + j.burst`) for each job.

- **Output:** Finally, `printAverageTimes` calculates and prints the average waiting and turnaround times, and `printGanttChart` generates a visual representation of job execution over time.

Shortest Job First (SJF)

Algorithm Overview:

SJF schedules jobs based on their burst time - the time required for a job to complete execution. It executes the shortest job first to minimize average waiting time.

Logic Behind the Algorithm:

- **Sorting by Burst Time:** Upon arrival, jobs are sorted based on their burst times to prioritize shorter jobs.
- **Execution:** The CPU continually selects the job with the smallest burst time that has arrived and is ready to execute.
- **Preemption:** If a shorter job arrives while another job is executing, the current job may be preempted to execute the shorter one.

Merits:

- Minimizes average waiting time.
- Efficient for scenarios where burst times are known or predictable.

Demerits:

- Requires accurate prediction of burst times, which can be challenging in practice.
- Long jobs may suffer from starvation if shorter jobs continuously arrive.

Best Use:

SJF is ideal when burst times can be estimated or predicted with accuracy. It's suitable for batch processing systems with a mixture of job lengths.

Example Implementation and Explanation:

```
void sjf(vector<Job> jobs, ofstream& outputFile) {
    sort(jobs.begin(), jobs.end(), SortByArrival);

    vector<Execution> executions;
    int currentTime = 0;
    int completedJobs = 0;
    int n = jobs.size();
    vector<bool> isJobCompleted(n, false);

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

        // Find the shortest job that has arrived and is not completed
        for (int i = 0; i < n; i++) {
            if (jobs[i].arrival <= currentTime && !isJobCompleted[i] && jobs[i].burst < shortestBurstTime) {
                shortestBurstTime = jobs[i].burst;
                shortestJobIndex = i;
            }
        }

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

        // Execute the shortest job
        Job& j = jobs[shortestJobIndex];
        j.wait = max(0, currentTime - j.arrival);
        executions.push_back(Execution(j.id, currentTime, currentTime + j.burst));
        currentTime += j.burst;
        j.turnaround = j.wait + j.burst;
        isJobCompleted[shortestJobIndex] = true;
        completedJobs++;
    }

    // Calculate turnaround time for each job
    calculateWaitAndTurnaround(jobs);

    // Output results
    outputFile << "SJF Scheduling:" << endl;
    printAverageTimes(jobs, outputFile);
    printGanttChart(executions, jobs, outputFile);
}
```

Explanation:

- **Sorting:** Jobs are initially sorted by arrival time (`SortByArrival`) to manage job arrival sequence.
- **Execution Loop:** The algorithm continuously selects the job with the shortest burst time that has arrived and is ready to execute.
- **Completion Check:** Jobs are marked as completed (`isJobCompleted`) once they finish execution, ensuring each job is executed exactly once.

- **Output:** Average waiting and turnaround times are computed using `printAverageTimes`, and a Gantt chart of job execution is generated with `printGanttChart`.

Shortest Remaining Time First (SRTF)

Algorithm Overview:

SRTF is a preemptive version of SJF where the CPU may switch to a shorter job if it arrives and requires less time to complete than the currently executing job.

Logic Behind the Algorithm:

- **Dynamic Selection:** Jobs are dynamically selected based on their remaining burst time at each scheduling decision point.
- **Preemption:** If a shorter job arrives that can complete sooner than the currently executing job, the CPU switches to the new job.
- **Execution and Completion:** Jobs are executed in increments until completion, adjusting remaining burst times after each execution.

Merits:

- Further reduces average waiting time compared to SJF.
- Provides better responsiveness for interactive systems.

Demerits:

- Increased overhead due to frequent context switching.
- May result in starvation of longer jobs if shorter jobs continuously arrive.

Best Use:

SRTF is suitable for environments where burst times vary significantly and jobs arrive frequently. It ensures responsiveness and reduces waiting times effectively.

Example Implementation and Explanation:

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

    while (completed != n) {
        // Find the job with the shortest remaining time
        for (int i = 0; i < n; ++i) {
            if (jobs[i].arrival <= currentTime && jobs[i].remaining > 0) {
                if (jobs[i].remaining < shortestRemainingTime) {
                    shortestRemainingTime = jobs[i].remaining;
                    shortestIndex = i;
                    check = true;
                }
            }
        }

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

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

        // Update remaining time
        shortestRemainingTime = jobs[shortestIndex].remaining;
        if

 (shortestRemainingTime == 0) {
            shortestRemainingTime = INT_MAX;
            jobs[shortestIndex].turnaround = currentTime + 1 - jobs[shortestIndex].arrival;
            jobs[shortestIndex].waiting = jobs[shortestIndex].turnaround - jobs[shortestIndex].burst;
            ++completed;
            check = false;
        }
        ++currentTime;
    }

    // Calculate the average waiting time
    calculateWaitAndTurnaround(jobs);

    // Results output
    outputFile << "SRTF Scheduling:" << endl;
    printAverageTimes(jobs, outputFile);
    printGanttChart(executions, jobs, outputFile);
}
```

Explanation:

- **Finding Shortest Job:** The algorithm iterates through jobs to find the one with the shortest remaining time (`jobs[i].remaining`).
- **Execution:** Each job is executed for one time unit (`currentTime++`), and its remaining time (`jobs[shortestIndex].remaining`) is decremented.
- **Completion Check:** If a job completes (`jobs[shortestIndex].remaining == 0`), it's marked as completed, and waiting and turnaround times are calculated.

Round Robin (RR)

Algorithm Overview:

Round Robin (RR) is a preemptive scheduling algorithm where each process is assigned a fixed time unit or quantum. If a process does not finish within its quantum, it's preempted and added to the end of the ready queue.

Logic Behind the Algorithm:

- **Initialization:** Each job is given an initial quantum for execution.
- **Execution:** Jobs are executed in a cyclic manner with each job getting a turn for a time slice (quantum).
- **Preemption:** If a job's quantum expires, it's moved to the end of the ready queue to wait for its next turn.
- **Completion:** Jobs are continuously executed in a loop until all are completed.

Merits:

- Fairly simple to implement.
- Suitable for time-sharing systems where responsiveness is crucial.
- Prevents starvation as every job gets a turn.

Demerits:

- Higher average waiting times compared to SJF or SRTF, especially for longer quantum times.
- Performance heavily depends on the quantum size: too small leads to high overhead, too large reduces responsiveness.

Best Use:

- RR is effective in time-sharing systems and environments where fairness and responsiveness are prioritized over minimizing waiting times.

Example Implementation and Explanation:

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

    // Initialize remaining burst times for each job
    for (auto& j : jobs) {
        j.remaining = j.burst;
    }

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

    while (completed != jobs.size()) {
        // Add jobs to the ready queue that have arrived
        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();

        // Execute the job for a time slice (quantum)
        int timeSlice = min(quantum, jobs[current].remaining);
        executions.push_back(Execution(jobs[current].id, currentTime, currentTime + timeSlice));
        jobs[current].remaining -= timeSlice;
        currentTime += timeSlice;

        // Add jobs back to the ready queue if they're ready to run again
        for (int i = 0; i < n; i++) {
            if (arrival[i] <= currentTime && arrival[i] != -1) {
                readyQueue.push(i);
                arrival[i] = -1;
            }
        }

        // Check if the job has completed
        if (jobs[current].remaining > 0) {
            readyQueue.push(current);
        } else {
            completed++;
            // Calculate waiting time for the completed job
            jobs[current].wait = currentTime - jobs[current].arrival - jobs[current].burst;
        }
    }

    // Calculate turnaround time for each job
    calculateWaitAndTurnaround(jobs);

    // Output results
    outputFile << "Round Robin Scheduling (Dynamic Quantum):" << endl;
    printAverageTimes(jobs, outputFile);
    printGanttChart(executions, jobs, outputFile);
}
```

Explanation:

- **Initialization:** The quantum size (`quantum`) is provided as an input parameter.
- **Ready Queue:** `queue<int> readyQueue` manages the sequence of jobs ready to execute.
- **Execution Loop:** Jobs are continuously selected from the ready queue and executed for the specified quantum time (`timeSlice`).
- **Completion Check:** After executing for the time slice, if a job still has remaining burst time (`jobs[current].remaining > 0`), it's added back to the ready queue. If not, it's marked as completed (`completed++`).
- **Turnaround Time:** `calculateWaitAndTurnaround(jobs)` computes the turnaround time (`j.turnaround = j.wait + j.burst`) for each job.
- **Output:** Average waiting and turnaround times are printed using `printAverageTimes`, and a Gantt chart of job execution is generated with `printGanttChart`.

Choosing the Best Algorithm

Algorithm Selection Criteria:

When choosing a CPU scheduling algorithm, consider the following factors:

- **Type of Workload:** Whether the workload consists of long or short jobs, or a mix of both.
- **System Requirements:** Prioritize factors such as throughput, response time, fairness, and CPU utilization.
- **Implementation Complexity:** Some algorithms are easier to implement and maintain than others.

Example of Algorithm Selection:

Based on the characteristics of the jobs (job burst times, arrival times, etc.), the program predicts the best scheduling algorithm using statistical analysis (`predictBestAlgorithm(jobs)`). The predicted algorithm (`predictedAlgorithm`) is then used to determine which scheduling algorithm (FCFS, SJF, SRTF, or RR) to apply.

Front End Description

The frontend consists of three main components:

1. **HTML (index.html)**:
    - Provides the user interface with a form to input job details (Job ID, Burst Time, Arrival Time).

- Dynamically adds job input fields using JavaScript when users click the "Add Another Job" button.
- Submits job data to the server upon form submission.

2. **JavaScript (script.js)**:
   - Adds functionality to dynamically add job input fields (`addJob()` function).
   - Handles form submission using the Fetch API to send job data to the server (`submit` event listener).

3. **Python (app.py)**:
   - Uses Flask to handle web requests and serve the HTML template.
   - Receives job data via a POST request (`/schedule` endpoint).
   - Writes job data to a file (`input.txt`), executes a C++ program (`./main`) using subprocess to process the job scheduling, and reads the output from a file (`output.txt`).
   - Returns the output as a JSON response to the frontend.

Setup and Usage

1. *Compile the C++ Program*: bash 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 app.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/ (http://127.0.0.1:5000/).

6. *Input Processes*:
   - Enter the PID, burst time, and arrival time for each process.
   - Click "Add Process" to add more processes.
   - Click "Schedule" to perform scheduling and view the results. **Usage:**

- Modify the `input.txt` file with your own set of jobs if needed.
- Execute the program to analyze and compare different scheduling algorithms based on the provided job details.
- Review the `output.txt` file for detailed scheduling results including average waiting times, turnaround times, and Gantt charts.

This README provides comprehensive insights into different CPU scheduling algorithms implemented in C++, including their logic, implementation details, merits, and demerits. Each algorithm's example code illustrates how it operates, computes metrics, and generates visual representations of job scheduling.