

Report : CPU Scheduler

1. Introduction

A CPU scheduler is a pivotal component of an operating system that orchestrates the execution of processes. It ensures that the CPU's resources are utilized efficiently by deciding which process should run at any given time and for how long. The goal is to optimize system performance by balancing various factors such as throughput, response time, and fairness. This report details the implementation of a sophisticated CPU scheduler using four different algorithms: First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin (RR), and Shortest Remaining Time First (SRTF). The project is implemented in C++ for the scheduler and uses React for the frontend and Flask for the backend.

2. Understanding Scheduling Algorithms

2.1 First Come First Serve (FCFS)

- **Description:** The FCFS algorithm schedules processes in the order they arrive in the ready queue. It is simple and straightforward but can lead to the "convoy effect" where shorter processes wait for longer ones to complete.
- **Implementation:** In C++, FCFS can be implemented using a simple queue data structure where processes are dequeued in the order they arrive.

```
std::vector<int> waitTime(count), turnTime(count), responseTime(count);
int elapsedTime = 0;
for (int i = 0; i < count; ++i) {
    if (elapsedTime < tasks[i].arrival) {
        elapsedTime = tasks[i].arrival;
    }
    waitTime[i] = elapsedTime - tasks[i].arrival;
    turnTime[i] = waitTime[i] + tasks[i].duration;
    responseTime[i] = waitTime[i];
    elapsedTime += tasks[i].duration;
}

double averageWait = 0, averageTurn = 0, averageResponse = 0;
for (int i = 0; i < count; ++i) {
    averageWait += waitTime[i];
    averageTurn += turnTime[i];
    averageResponse += responseTime[i];
}
averageWait /= count;
averageTurn /= count;
averageResponse /= count;
```

2.2 Shortest Job First (SJF)

- **Description:** SJF selects the process with the smallest execution time. This algorithm can be either preemptive or non-preemptive. It minimizes average waiting time but requires precise knowledge of how long each process will take.
- **Implementation:** SJF can be implemented using a priority queue where processes are ordered by their execution time.

```
std::sort(jobList.begin(), jobList.end(), burstTimeComparator);

std::vector<int> waitTime(jobCount), turnTime(jobCount),
respTime(jobCount);
int currentTime = 0;
for (int i = 0; i < jobCount; ++i) {
    if (currentTime < jobList[i].arrival) {
        currentTime = jobList[i].arrival;
    }
    waitTime[i] = currentTime - jobList[i].arrival;
    turnTime[i] = waitTime[i] + jobList[i].burst;
    respTime[i] = waitTime[i];
    currentTime += jobList[i].burst;
}

double avgWait = 0, avgTurn = 0, avgResp = 0;
for (int i = 0; i < jobCount; ++i) {
    avgWait += waitTime[i];
    avgTurn += turnTime[i];
    avgResp += respTime[i];
}
avgWait /= jobCount;
avgTurn /= jobCount;
avgResp /= jobCount;
```

2.3 Round Robin (RR)

- **Description:** RR assigns a fixed time quantum for each process and cycles through them in a circular queue. This ensures a fair distribution of CPU time but can lead to higher average waiting times compared to other algorithms.
- **Implementation:** RR can be implemented using a circular queue where each process gets executed for a fixed time slice.

```

vector<Job> jobs(numJobs);
for (int i = 0; i < numJobs; ++i) {
    inFile >> jobs[i].pid >> jobs[i].arrival >> jobs[i].burst;
}
inFile >> timeSlice;

sort(jobs.begin(), jobs.end(), compareArrival);
queue<Job> jobQueue;
map<int, int> completionMap;
map<int, int> firstRunMap;

int currentTime = 0;
int index = 0;
if (jobs[index].arrival > currentTime) {
    currentTime = jobs[index].arrival;
}

while (index < numJobs && jobs[index].arrival <= currentTime) {
    jobQueue.push(jobs[index]);
    ++index;
}

while (index < numJobs) {
    Job currentJob = jobQueue.front();
    if (firstRunMap.count(currentJob.pid) == 0) {
        firstRunMap[currentJob.pid] = currentTime;
    }
    jobQueue.pop();
    bool jobCompleted = false;
    if (currentJob.burst <= timeSlice) {
        currentTime += currentJob.burst;
        jobCompleted = true;
        completionMap[currentJob.pid] = currentTime;
    } else {
        currentTime += timeSlice;
        currentJob.burst -= timeSlice;
    }
    while (index < numJobs && jobs[index].arrival <= currentTime) {
        jobQueue.push(jobs[index]);
        ++index;
    }
    if (!jobCompleted) {

```

```

        jobQueue.push(currentJob);
    }
}

while (!jobQueue.empty()) {
    Job currentJob = jobQueue.front();
    if (firstRunMap.count(currentJob.pid) == 0) {
        firstRunMap[currentJob.pid] = currentTime;
    }
    jobQueue.pop();
    if (currentJob.burst <= timeSlice) {
        currentTime += currentJob.burst;
        completionMap[currentJob.pid] = currentTime;
    } else {
        currentTime += timeSlice;
        currentJob.burst -= timeSlice;
        jobQueue.push(currentJob);
    }
}

for (auto &job : jobs) {
    job.completion = completionMap[job.pid];
    job.turnaround = job.completion - job.arrival;
    job.waiting = job.turnaround - job.burst;
    job.response = firstRunMap[job.pid] - job.arrival;
}

ldouble avgWaiting = 0, avgResponse = 0, avgTurnaround = 0;
for (const auto &job : jobs) {
    avgResponse += job.response;
    avgWaiting += job.waiting;
    avgTurnaround += job.turnaround;
}

avgWaiting /= numJobs;
avgResponse /= numJobs;
avgTurnaround /= numJobs;

```

2.4 Shortest Remaining Time First (SRTF)

- **Description:** SRTF is a preemptive version of SJF. It selects the process with the shortest remaining execution time. This approach is optimal for minimizing waiting time but can be challenging to implement due to the need for continuous updating of remaining times.
- **Implementation:** SRTF can be implemented using a priority queue that is continuously updated with the remaining times of processes.

```
std::vector<int> waitTimes(numJobs), turnaroundTimes(numJobs),
responseTimes(numJobs, -1);

int currentTime = 0, jobsCompleted = 0;

while (jobsCompleted < numJobs) {
    int shortestJobIndex = -1;
    int minimumRemaining = INT_MAX;

    for (int i = 0; i < numJobs; ++i) {
        if (jobList[i].arrival <= currentTime && jobList[i].remaining > 0
&& jobList[i].remaining < minimumRemaining) {
            minimumRemaining = jobList[i].remaining;
            shortestJobIndex = i;
        }
    }

    if (shortestJobIndex != -1) {
        if (responseTimes[shortestJobIndex] == -1) {
            responseTimes[shortestJobIndex] = currentTime -
jobList[shortestJobIndex].arrival;
        }
        currentTime++;
        jobList[shortestJobIndex].remaining--;
        if (jobList[shortestJobIndex].remaining == 0) {
            jobsCompleted++;
            waitTimes[shortestJobIndex] = currentTime -
jobList[shortestJobIndex].arrival - jobList[shortestJobIndex].burst;
            turnaroundTimes[shortestJobIndex] = currentTime -
jobList[shortestJobIndex].arrival;
        }
    } else {
        currentTime++;
    }
}

double totalWaitTime = 0, totalTurnaroundTime = 0, totalResponseTime = 0;
for (int i = 0; i < numJobs; ++i) {
    totalWaitTime += waitTimes[i];
    totalTurnaroundTime += turnaroundTimes[i];
}
```

```
        totalResponseTime += responseTimes[i];
    }
    double avgWaitTime = totalWaitTime / numJobs;
    double avgTurnaroundTime = totalTurnaroundTime / numJobs;
    double avgResponseTime = totalResponseTime / numJobs;
```

3. Project Implementation

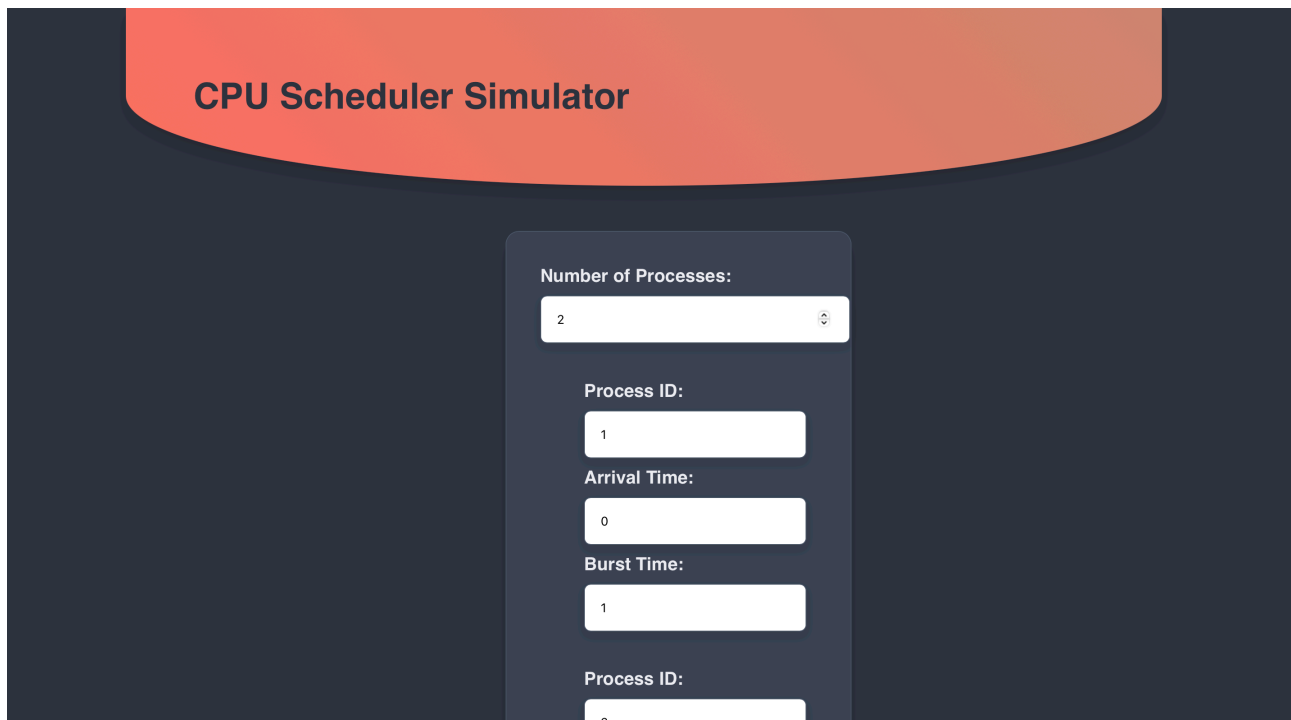
3.1 Backend Implementation

The backend, implemented in Flask, handles the core scheduling logic. It processes task information and schedules them according to the selected algorithm. The backend exposes RESTful APIs for the frontend to interact with.

3.2 Scheduler Implementation in C++

The scheduler is implemented in C++, incorporating the four algorithms mentioned above. The scheduler takes in a list of tasks with their arrival times and burst times and outputs the order of execution along with various performance metrics.

3.3 Frontend Implementation



The screenshot shows a web application titled "CPU Scheduler Simulator". The interface is dark-themed with a central form for inputting process details. The form includes the following fields:

- Number of Processes:** A text input field containing the value "2".
- Process ID:** A text input field containing the value "1".
- Arrival Time:** A text input field containing the value "0".
- Burst Time:** A text input field containing the value "1".
- Process ID:** A text input field containing the value "2".

Arrival Time:

1

Burst Time:

3

Add Process

Algorithm:

First-Come, First-Served

Run Algorithm

Results

Average Waiting Time: 0

Average Turnaround Time: 2

Average Response Time: 0

The frontend, developed in React, provides an interactive interface for users to input task information and select a scheduling algorithm. It visually demonstrates the scheduling decisions made by the backend, showing Gantt charts and other statistics to illustrate the efficiency and fairness of the scheduling.

4. Demonstrating Optimal Decisions

The frontend showcases various statistics to demonstrate the optimality of scheduling decisions:

- Average Response time : time taken by the process to respond.
- **Average Waiting Time:** Time processes spend waiting in the ready queue.
- **Average Turnaround Time:** Total time taken from arrival to completion of each process.

5. Conclusion

This project successfully implements a sophisticated CPU scheduler using multiple algorithms and demonstrates their effectiveness through an interactive frontend. By balancing throughput, response time, and fairness, the scheduler ensures efficient resource utilization. The combination of C++ for core logic and React with Flask for the user interface provides a robust and flexible solution for visualizing and understanding CPU scheduling.

6. Future Work

Future improvements could include:

- **Support for Additional Algorithms:** Implementing more complex algorithms like Multilevel Queue Scheduling.
- **Real-time Simulation:** Adding support for real-time task simulation to observe scheduling decisions in a live environment.
- **Enhanced Visualization:** More detailed and interactive visualizations for deeper analysis of scheduling behavior.

By continuing to refine and expand this project, it can serve as an educational tool for understanding the intricacies of CPU scheduling and its impact on system performance.

