CPU SCHEDULER

1. Introduction

The project aims to develop a CPU scheduling simulator that incorporates various scheduling algorithms using a combination of C++ for the backend logic and Python with Tkinter for the frontend graphical user interface (GUI). This report provides a detailed overview of the project, covering the design choices, implementation details, features, and potential improvements.

2. Details about the Project

2.1 C++ Backend

The C++ backend forms the core of the CPU scheduling simulator, responsible for implementing the scheduling algorithms and computing performance metrics such as average turnaround time and average waiting time. Let's delve into the key components of the C++ code:

2.1.1 Task Struct Definition

cpp

```
struct Task
{
    double id;
    double burstTime;
    double arrivalTime;
    double priority;

    Task(double id, double burstTime, double arrivalTime, double priority = 0)
        : id(id), burstTime(burstTime), arrivalTime(arrivalTime), priority(priority) {}
};
```

Explanation:

Purpose: The Task struct encapsulates information about each task to be scheduled, including its ID, burst time, arrival time, and priority (optional).

Attributes:

id: Unique identifier for the task.
burstTime: Time required by the task to complete its execution.
arrivalTime: Time at which the task arrives in the system.
priority: Priority assigned to the task, defaulting to 0 if not specified explicitly.

Constructor: Initializes a Task object with provided values for id, burstTime, arrivalTime, and priority.

2.1.2 Comparator Functions

cpp

```
bool compareBurstTime(const Task &a, const Task &b)
{
    return a.burstTime < b.burstTime;
}

bool compareArrivalTime(const Task &a, const Task &b)
{
    return a.arrivalTime < b.arrivalTime;
}

bool comparePriority(const Task &a, const Task &b)
```

```
{
    return a.priority > b.priority;
}
```

Explanation:

Purpose: Comparator functions used for sorting tasks based on different criteria.
compareBurstTime: Compares tasks based on their burst time in ascending order.
compareArrivalTime: Compares tasks based on their arrival time in ascending order.
comparePriority: Compares tasks based on their priority in descending order (higher priority first).

These comparator functions are essential for sorting tasks before applying scheduling algorithms such as Shortest Job First (SJF), First Come First Served (FCFS), and Priority Scheduling.

2.1.3 Scheduling Algorithms

Several scheduling algorithms are implemented in the backend to simulate different scheduling policies:

FCFS (First Come, First Served): Tasks are executed in the order they arrive.
SJF (Shortest Job First): The task with the smallest burst time is executed first.
Round Robin (RR): Tasks are executed for a fixed time quantum in a cyclic manner.
Priority Scheduling (Non-preemptive and Preemptive): Tasks with higher priority are executed first, with preemptive allowing tasks to be interrupted.

Each algorithm calculates the average turnaround time and average waiting time for the set of tasks provided.

3. Python Frontend (Tkinter)

The Python frontend provides a user-friendly interface for interacting with the CPU scheduling simulator. It utilizes Tkinter to create a GUI that allows users to input task details and parameters, run the simulation, and visualize the results.

3.1 GUI Design and Functionality

The frontend includes the following key components:

Input Fields: Dynamically generated entry fields for users to input task details (burst time, arrival time, priority).
Validation: Ensures that all inputs are valid (positive values for burst time and arrival time, non-negative values for priority).
Run Simulation Button: Executes the C++ backend program with user inputs and displays the results.
Result Display: Shows the average turnaround time and average waiting time for each scheduling algorithm in a structured table format.

3.2 Integration with C++ Backend

The Python script interacts with the C++ backend by passing input data (number of tasks, task details, scheduling parameters) through subprocess calls. It captures the output from the C++ program and parses it to display results on the GUI.

4. Features Implemented

4.1 Core Features

Multiple Scheduling Algorithms: FCFS, SJF, RR, and Priority Scheduling (both preemptive and non-preemptive) are implemented to compare their performance.
User-friendly Interface: Tkinter provides an intuitive interface with validation and dynamic updates based on user inputs.
Comprehensive Simulation: Calculates and displays average turnaround time and average waiting time

for each scheduling algorithm based on user-provided task details.

4.2 Additional Creativity

Error Handling: Ensures robust error handling for invalid inputs and edge cases.
Dynamic UI Updates: Adjusts the UI dynamically to accommodate varying numbers of tasks.
Integration of Languages: Demonstrates interoperability between C++ and Python for backend processing and frontend presentation.

5. Summary and Further Improvements
5.1 Summary

The project successfully implements a CPU scheduling simulator that allows users to explore different scheduling algorithms and visualize their effects. It combines efficient backend logic in C++ with an interactive frontend in Python, providing a comprehensive learning and analysis tool for CPU scheduling strategies.
5.2 Points of Improvement

Enhanced Visualization: Incorporate graphical representations (e.g., Gantt charts) to visually depict task scheduling and execution.
Advanced Algorithms: Implement more complex scheduling algorithms like Multilevel Feedback Queue (MLFQ) or Shortest Remaining Time (SRT).
Real-time Simulation: Develop capabilities for real-time simulation to observe dynamic task arrivals and scheduling changes.
Performance Metrics: Include additional performance metrics such as response time and CPU utilization for deeper analysis.
User Experience: Improve the GUI with additional features like task grouping, scenario saving, and comparative analysis tools.

6. Conclusion

The CPU scheduling simulator project demonstrates effective integration of backend algorithms and frontend GUI, providing a practical tool for studying and comparing CPU scheduling strategies. By leveraging C++ for computational efficiency and Python with Tkinter for user interaction, the project achieves a balanced approach to simulating and analyzing scheduling algorithms in operating systems.
7. Future Directions

Future iterations of the project could focus on expanding functionality to include interactive tutorials, integration with cloud-based simulations, or extending support for distributed computing environments. Moreover, enhancing the GUI with more customization options and accessibility features would further enhance its usability and educational value.