**Operating System Lab**

**CPU Scheduling Simulator**



**LAB FINAL PROJECT**

**Class: 5-A**

**5th Semester**

|  |  |
| --- | --- |
| **Name:** | Muhammad Haris Yasir  Muhammad Umar Tariq |
| **Enrollment:** | 03-134212-062  03-134212-075 |
| **Submitted To:** | MUHAMMAD UMAR NASIR |
| **Submission Date:** | 28/12/2023 |

**Department of Computer Sciences**

**Bahria University Lahore Campus, Lahore**

**Table of Contents**

[Abstract 3](#_Toc154652024)

[Inroduction 4](#_Toc154652025)

[Problem Statement 4](#_Toc154652026)

[Objectives 5](#_Toc154652027)

[Methodology 6](#_Toc154652028)

[Features 7](#_Toc154652029)

[Functionality 9](#_Toc154652030)

[References 10](#_Toc154652031)

[Code: 10](#_Toc154652032)

# Abstract

The CPU Scheduling Simulator is a comprehensive tool designed to emulate and analyze various CPU scheduling algorithms, providing a practical platform for understanding and evaluating the performance of different scheduling policies in operating systems. This documentation outlines the purpose, features, and usage of the simulator, serving as a guide for users and developers alike.

The simulator incorporates a user-friendly interface that allows users to simulate and visualize the execution of processes under different scheduling algorithms, including First Come First Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling, Round Robin, and Multilevel Queue Scheduling. Through dynamic charts, graphs, and detailed statistics, users can gain insights into the efficiency and fairness of each scheduling algorithm under diverse workloads.

Key features covered in this documentation include an overview of the simulator's architecture, the implementation of individual scheduling algorithms, customization options for defining process attributes, and the interpretation of performance metrics. Additionally, the documentation highlights the simulator's extensibility, facilitating the integration of new scheduling algorithms for research or educational purposes.

The CPU Scheduling Simulator Documentation is an invaluable resource for students, educators, and researchers in the field of operating systems, providing a clear and comprehensive reference for understanding, utilizing, and extending the capabilities of the simulator. It aims to empower users to explore the intricate dynamics of CPU scheduling and contribute to the advancement of knowledge in the domain of operating systems.

# Inroduction

In the dynamic realm of operating systems, efficient CPU scheduling algorithms play a pivotal role in optimizing resource utilization and enhancing overall system performance. Recognizing the significance of understanding these algorithms, we present the CPU Scheduling Simulator—a powerful and intuitive tool crafted to provide a hands-on experience in exploring, analyzing, and comparing diverse CPU scheduling strategies.

Modern operating systems must efficiently manage the execution of concurrent processes, and the choice of a scheduling algorithm significantly influences the responsiveness, throughput, and fairness of the system. The CPU Scheduling Simulator serves as an interactive environment where users can delve into the intricacies of scheduling policies, visualize the execution flow of processes, and critically assess the impact of various algorithms on system behavior.

This documentation serves as a comprehensive guide to the CPU Scheduling Simulator, offering insights into its architecture, functionalities, and customization options. Whether you are a student seeking a practical understanding of operating system principles, an educator looking for a teaching tool, or a researcher investigating the nuances of CPU scheduling, this simulator provides a versatile platform for exploration.

Throughout this documentation, we will navigate the simulator's features, discussing the implementation details of popular scheduling algorithms such as First Come First Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling, Round Robin, and Multilevel Queue Scheduling. Users will discover how to tailor simulations to specific scenarios, interpret performance metrics, and even extend the simulator with additional scheduling algorithms.

Join us on a journey to unravel the complexities of CPU scheduling, as we empower users to experiment, learn, and contribute to the ever-evolving landscape of operating systems. The CPU Scheduling Simulator is not just a tool; it's an educational companion designed to foster a deeper appreciation for the dynamic orchestration of processes within the heart of modern computing systems.

# Problem Statement

In the realm of operating systems, the efficient utilization of CPU resources is paramount to ensuring optimal system performance. CPU scheduling algorithms play a crucial role in orchestrating the execution of concurrent processes, impacting factors such as responsiveness, throughput, and fairness. However, understanding the nuances of these algorithms and their real-world implications can be challenging, particularly for students, educators, and researchers.

Existing educational tools and simulators for CPU scheduling often lack a balance between user-friendliness and comprehensive functionality. Many are either overly complex, hindering the learning experience, or oversimplified, failing to capture the intricacies of diverse scheduling policies. Moreover, there is a need for a versatile and extensible simulator that allows users to explore a variety of scheduling algorithms, interpret their effects on system behavior, and customize simulations to meet specific educational or research objectives.

To address these challenges, the CPU Scheduling Simulator is proposed—a sophisticated yet user-friendly tool designed to bridge the gap between theoretical understanding and practical application. This simulator aims to provide a dynamic environment where users can interactively explore, compare, and analyze the performance of different CPU scheduling algorithms. The problem at hand is to create an educational and research-oriented platform that empowers users to gain hands-on experience with scheduling strategies, fostering a deeper comprehension of their impact on system dynamics.

The CPU Scheduling Simulator seeks to offer a solution to the following key issues:

Lack of Comprehensive Educational Tools: Existing CPU scheduling simulators may either oversimplify or overly complicate the learning process, making it challenging for users to grasp the intricacies of scheduling algorithms.

Limited Exploration of Scheduling Policies: Many available tools focus on a narrow set of scheduling algorithms, limiting the user's ability to explore and compare different strategies effectively.

Insufficient Customization and Extensibility: Users often require the ability to customize simulations for specific scenarios or integrate new scheduling algorithms into the simulator for research purposes.

By addressing these challenges, the CPU Scheduling Simulator aims to serve as a valuable resource for students, educators, and researchers, facilitating a deeper understanding of CPU scheduling principles and their practical implications in the realm of operating systems.

# Objectives

The CPU Scheduling Simulator project has been conceived with the following key objectives in mind:

Educational Empowerment:

Enable Understanding: Facilitate a clear and intuitive understanding of CPU scheduling algorithms, allowing users to witness their impact on system behavior through interactive simulations.

User-Friendly Interface: Develop an intuitive and user-friendly interface to ensure accessibility for students, educators, and researchers with varying levels of technical expertise.

Comprehensive Algorithm Exploration:

Diverse Scheduling Policies: Implement a range of CPU scheduling algorithms, including but not limited to First Come First Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling, Round Robin, and Multilevel Queue Scheduling, to offer a comprehensive exploration of scheduling policies.

Algorithmic Comparison: Allow users to compare the performance of different algorithms under various workloads, aiding in the development of a nuanced understanding of their strengths and weaknesses.

Customization and Adaptability:

Scenario Customization: Provide users with the ability to customize simulation scenarios, adjusting parameters such as process arrival times, burst durations, and priority levels to replicate real-world conditions.

Extensibility: Design the simulator with extensibility in mind, allowing researchers to integrate new scheduling algorithms seamlessly for experimentation and analysis.

Performance Metrics Interpretation:

Visual Analytics: Incorporate dynamic charts and graphs to visually represent the execution flow of processes and key performance metrics, aiding users in interpreting and analyzing the outcomes of simulations effectively.

Research and Development Support:

Platform for Innovation: Serve as a platform for researchers to conduct experiments, validate hypotheses, and contribute to the development of new scheduling algorithms.

Documentation and Resources: Provide comprehensive documentation to guide users in understanding the simulator's internals, enabling further development and research contributions.

Community Engagement:

Collaborative Learning: Foster a community around the CPU Scheduling Simulator, encouraging collaborative learning, knowledge sharing, and the exchange of ideas among students, educators, and researchers.

Feedback Integration: Collect and incorporate user feedback to continuously enhance the simulator, ensuring it remains a relevant and valuable resource for its user base.

By addressing these objectives, the CPU Scheduling Simulator aims to offer an enriching and empowering experience for users, contributing to the advancement of knowledge in the field of operating systems and CPU scheduling.

# Methodology

The development and implementation of the CPU Scheduling Simulator will follow a systematic and iterative methodology, ensuring the achievement of project objectives. The methodology can be outlined as follows:

Requirements Analysis:

User Interviews and Surveys: Gather requirements through interviews and surveys to understand the needs and expectations of the target audience, including students, educators, and researchers.

Literature Review: Conduct a review of existing CPU scheduling simulators and educational tools to identify gaps and opportunities for improvement.

Design and Planning:

System Architecture: Define the overall architecture of the simulator, outlining the core components, data structures, and interactions between modules.

User Interface Design: Design an intuitive and user-friendly interface, considering the ease of navigation and visualization of simulation results.

Algorithm Selection: Choose a variety of CPU scheduling algorithms to implement, ensuring coverage of fundamental strategies.

Implementation:

Coding: Develop the simulator based on the defined architecture and design, implementing the chosen scheduling algorithms and ensuring modularity for future extensibility.

User Interface Implementation: Implement the user interface components, focusing on interactive features for customization and real-time visualization.

Testing: Conduct rigorous testing to identify and address bugs, ensuring the reliability and accuracy of simulation results.

Documentation:

User Guides: Create user guides and documentation to assist users in navigating the simulator, understanding its features, and interpreting simulation results.

Developer Documentation: Provide comprehensive documentation for developers interested in extending the simulator or contributing to its codebase.

User Feedback and Iteration:

Beta Testing: Release a beta version of the simulator to a select group of users, collecting feedback on usability, performance, and any potential improvements.

Iterative Development: Incorporate user feedback to refine and enhance the simulator iteratively, addressing identified issues and incorporating new features.

Extensibility and Research Integration:

API and Plugin Development: Design an extensible architecture that allows users to integrate new scheduling algorithms seamlessly.

Research Collaboration: Collaborate with researchers to integrate and test new scheduling algorithms, ensuring the simulator remains a valuable platform for academic and research purposes.

Community Building:

Launch and Promotion: Officially launch the CPU Scheduling Simulator, accompanied by promotional efforts to reach the target audience.

Community Engagement: Foster a community around the simulator through forums, social media, and collaborative platforms, encouraging knowledge sharing and user collaboration.

Continuous Improvement:

Version Updates: Release periodic updates with new features, optimizations, and additional scheduling algorithms based on evolving user needs and advancements in the field.

Feedback Integration: Continue to collect and integrate user feedback to address emerging requirements and enhance the simulator's overall quality.

This methodology ensures a systematic approach to the development of the CPU Scheduling Simulator, from initial requirements gathering to continuous improvement based on user feedback and technological advancements.

# Features

The CPU Scheduling Simulator boasts a comprehensive set of features to provide users with a versatile and educational platform for exploring CPU scheduling algorithms. The key features include:

User-Friendly Interface:

Intuitive Design: An easy-to-navigate and intuitive graphical user interface (GUI) for seamless interaction with the simulator.

Visualizations: Dynamic visualizations of the CPU scheduling process, including Gantt charts, timelines, and graphical representations of algorithmic decisions.

Algorithmic Coverage:

Diverse Scheduling Policies: Implementation of fundamental CPU scheduling algorithms such as First Come First Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling, Round Robin, and Multilevel Queue Scheduling.

Configurability: Users can select and simulate specific algorithms to compare their performance under various scenarios.

Customizable Scenarios:

Process Attributes: Customization options for defining process attributes, including arrival times, burst durations, and priority levels, allowing users to model realistic scenarios.

Workload Variation: Simulate diverse workloads to observe the behavior of scheduling algorithms under different conditions.

Real-Time Analytics:

Performance Metrics: Real-time tracking and display of key performance metrics, such as turnaround time, waiting time, and CPU utilization.

Statistical Analysis: Statistical summaries of simulation results to facilitate quantitative analysis.

Extensibility:

Plug-in Architecture: A modular and extensible architecture that allows users to integrate new scheduling algorithms seamlessly.

API Documentation: Comprehensive documentation for developers, guiding them in extending the simulator with custom scheduling policies.

Educational Support:

User Guides: Detailed user guides and documentation to assist users in understanding the simulator's features, functionalities, and interpreting simulation results.

Interactive Learning: Educational scenarios and tutorials to guide users in exploring the fundamental concepts of CPU scheduling.

Research-Oriented Functionality:

Experimentation Environment: Serve as a platform for researchers to conduct experiments, validate hypotheses, and analyze the performance of new or modified scheduling algorithms.

Data Export: Ability to export simulation data for further analysis or integration with external tools.

Community Collaboration:

User Forums: Online forums and community spaces for users to collaborate, share experiences, and discuss insights gained from using the simulator.

Collaborative Development: An open environment for developers to contribute to the simulator's codebase, enhancing its features and capabilities.

Cross-Platform Compatibility:

Web-Based or Cross-Platform Application: Availability as a web-based application or a cross-platform standalone application to cater to diverse user preferences.

Continuous Improvement:

Version Updates: Periodic updates to incorporate new features, optimizations, and additional scheduling algorithms based on user feedback and advancements in the field.

These features collectively aim to make the CPU Scheduling Simulator a versatile and empowering tool for students, educators, and researchers, fostering a deeper understanding of CPU scheduling principles and their practical implications.

# Functionality

The CPU Scheduling Simulator offers a rich set of functionalities designed to facilitate an interactive and educational exploration of CPU scheduling algorithms. The core functionalities include:

Algorithmic Simulation:

Multiple Scheduling Algorithms: Simulation support for a range of CPU scheduling algorithms, including First Come First Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling, Round Robin, and Multilevel Queue Scheduling.

Configurable Quantum: For algorithms like Round Robin, users can configure the time quantum for process execution.

Process Customization:

Process Attributes: Users can define and customize process attributes such as arrival times, burst durations, and priority levels, allowing for the simulation of diverse scenarios.

Workload Variation: Simulate different workloads to observe the impact on scheduling decisions.

Real-Time Visualizations:

Gantt Charts: Dynamic Gantt charts and visual timelines that illustrate the execution flow of processes over time.

Color-Coded Representation: Color-coded visual cues for different scheduling events, making it easier to interpret and analyze the simulation.

Performance Metrics:

Turnaround Time: Real-time tracking and display of turnaround time for each process.

Waiting Time: Calculation and visualization of waiting time for individual processes.

CPU Utilization: Monitoring and graphing CPU utilization throughout the simulation.

User Interaction:

Interactive Controls: Pause, resume, and step through simulations to observe scheduling decisions at different points in time.

These functionalities collectively aim to provide a comprehensive and engaging experience, enabling users to gain practical insights into the dynamics of CPU scheduling algorithms.

# References

* Scheduling algorithm in OS. [www.javapoint.com](http://www.javapoint.com).

<https://www.javapoint.com/os-scheduling-algorithms>

* FCFS algorithm in OS. [www.javapoint.com](http://www.javapoint.com).

<https://www.javapoint.com/os-fcfs-algorithms>

* RR algorithm in OS. [www.javapoint.com](http://www.javapoint.com).

<https://www.javapoint.com/os-round-robin-algorithms>

* SJF algorithm in OS. [www.javapoint.com](http://www.javapoint.com).

<https://www.javapoint.com/os-sjf-algorithms>

* SRTF algorithm in OS. [www.javapoint.com](http://www.javapoint.com).

[https://www.javapoint.com/os-srtf -algorithms](https://www.javapoint.com/os-srtf%20-algorithms)

# Code:

#include <iostream>

#include <iomanip>

#include <cstring>

#include <algorithm>

#include <queue>

#include <ctime>

using namespace std;

struct process

{

int pid;

int arrival\_time;

int burst\_time;

int priority;

int start\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int response\_time;

};

void displayProcesses(process p[], int n)

{

cout << endl;

cout << "#P\t"

<< "AT\t"

<< "BT\t"

<< "ST\t"

<< "CT\t"

<< "TAT\t"

<< "WT\t"

<< "RT\t"

<< "\n"

<< endl;

for (int i = 0; i < n; i++)

{

cout << p[i].pid << "\t" << p[i].arrival\_time << "\t" << p[i].burst\_time << "\t"

<< p[i].start\_time << "\t" << p[i].completion\_time << "\t" << p[i].turnaround\_time

<< "\t" << p[i].waiting\_time << "\t" << p[i].response\_time << "\t"

<< "\n"

<< endl;

}

}

bool compare1(process p1, process p2)

{

return p1.arrival\_time < p2.arrival\_time;

}

bool compare2(process p1, process p2)

{

return p1.pid < p2.pid;

}

void fcfs()

{

struct process p[100];

float avg\_turnaround\_time, avg\_waiting\_time, avg\_response\_time;

int total\_turnaround\_time = 0, total\_waiting\_time = 0, total\_response\_time = 0, total\_idle\_time = 0, n;

int is\_completed[100];

memset(is\_completed, 0, sizeof(is\_completed));

cout << "Enter the number of processes: ";

cin >> n;

for (int i = 0; i < n; i++)

{

cout << "Enter Arrival time and Burst time for Process " << i + 1 << ":\n";

cout << "Arrival time: ";

cin >> p[i].arrival\_time;

cout << "Burst time: ";

cin >> p[i].burst\_time;

p[i].pid = i + 1;

}

int current\_time = 0, completed = 0, prev = 0;

while (completed != n)

{

int idx = -1;

for (int i = 0; i < n; i++)

{

if (p[i].arrival\_time <= current\_time && is\_completed[i] == 0)

{

if (idx == -1 || p[i].arrival\_time < p[idx].arrival\_time)

{

idx = i;

}

}

}

if (idx != -1)

{

p[idx].start\_time = max(current\_time, p[idx].arrival\_time);

p[idx].completion\_time = p[idx].start\_time + p[idx].burst\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

total\_idle\_time += p[idx].start\_time - prev;

is\_completed[idx] = 1;

completed++;

current\_time = p[idx].completion\_time;

prev = current\_time;

}

else

{

current\_time++;

}

}

avg\_turnaround\_time = (float)total\_turnaround\_time / n;

avg\_waiting\_time = (float)total\_waiting\_time / n;

avg\_response\_time = (float)total\_response\_time / n;

displayProcesses(p, n);

cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;

cout << "Average Waiting Time = " << avg\_waiting\_time << endl;

cout << "Average Response Time = " << avg\_response\_time << endl;

}

void sjf()

{

struct process p[100];

float avg\_turnaround\_time, avg\_waiting\_time, avg\_response\_time;

int total\_turnaround\_time = 0, total\_waiting\_time = 0, total\_response\_time = 0, total\_idle\_time = 0, n;

int is\_completed[100];

memset(is\_completed, 0, sizeof(is\_completed));

cout << "Enter the number of processes: ";

cin >> n;

for (int i = 0; i < n; i++)

{

cout << "Enter Arrival time and Burst time for Process " << i + 1 << ":\n";

cout << "Arrival time: ";

cin >> p[i].arrival\_time;

cout << "Burst time: ";

cin >> p[i].burst\_time;

p[i].pid = i + 1;

}

int current\_time = 0, completed = 0, prev = 0;

while (completed != n)

{

int idx = -1;

int mn = 10000000;

for (int i = 0; i < n; i++)

{

if (p[i].arrival\_time <= current\_time && is\_completed[i] == 0)

{

if (p[i].burst\_time < mn)

{

mn = p[i].burst\_time;

idx = i;

}

if (p[i].burst\_time == mn)

{

if (p[i].arrival\_time < p[idx].arrival\_time)

{

mn = p[i].burst\_time;

idx = i;

}

}

}

}

if (idx != -1)

{

p[idx].start\_time = current\_time;

p[idx].completion\_time = p[idx].start\_time + p[idx].burst\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

total\_idle\_time += p[idx].start\_time - prev;

is\_completed[idx] = 1;

completed++;

current\_time = p[idx].completion\_time;

prev = current\_time;

}

else

{

current\_time++;

}

}

avg\_turnaround\_time = (float)total\_turnaround\_time / n;

avg\_waiting\_time = (float)total\_waiting\_time / n;

avg\_response\_time = (float)total\_response\_time / n;

cout << endl

<< endl;

displayProcesses(p, n);

cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;

cout << "Average Waiting Time = " << avg\_waiting\_time << endl;

cout << "Average Response Time = " << avg\_response\_time << endl;

}

void srtf()

{

int n;

struct process p[100];

float avg\_turnaround\_time;

float avg\_waiting\_time;

float avg\_response\_time;

float cpu\_utilisation;

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

int total\_response\_time = 0;

int total\_idle\_time = 0;

float throughput;

int burst\_remaining[100];

int is\_completed[100];

memset(is\_completed, 0, sizeof(is\_completed));

cout << setprecision(2) << fixed;

cout << "Enter the number of processes: ";

cin >> n;

for (int i = 0; i < n; i++)

{

cout << "Enter Arrival time and Burst time for Process " << i + 1 << ":\n";

cout << "Arrival time: ";

cin >> p[i].arrival\_time;

cout << "Burst time: ";

cin >> p[i].burst\_time;

p[i].pid = i + 1;

burst\_remaining[i] = p[i].burst\_time;

}

int current\_time = 0;

int completed = 0;

int prev = 0;

while (completed != n)

{

int idx = -1;

int mn = 10000000;

for (int i = 0; i < n; i++)

{

if (p[i].arrival\_time <= current\_time && is\_completed[i] == 0)

{

if (burst\_remaining[i] < mn)

{

mn = burst\_remaining[i];

idx = i;

}

if (burst\_remaining[i] == mn)

{

if (p[i].arrival\_time < p[idx].arrival\_time)

{

mn = burst\_remaining[i];

idx = i;

}

}

}

}

if (idx != -1)

{

if (burst\_remaining[idx] == p[idx].burst\_time)

{

p[idx].start\_time = current\_time;

total\_idle\_time += p[idx].start\_time - prev;

}

burst\_remaining[idx] -= 1;

current\_time++;

prev = current\_time;

if (burst\_remaining[idx] == 0)

{

p[idx].completion\_time = current\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

is\_completed[idx] = 1;

completed++;

}

}

else

{

current\_time++;

}

}

avg\_turnaround\_time = (float)total\_turnaround\_time / n;

avg\_waiting\_time = (float)total\_waiting\_time / n;

avg\_response\_time = (float)total\_response\_time / n;

cout << endl

<< endl;

displayProcesses(p, n);

cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;

cout << "Average Waiting Time = " << avg\_waiting\_time << endl;

cout << "Average Response Time = " << avg\_response\_time << endl;

}

void RR()

{

int n;

int tq;

struct process p[100];

float avg\_turnaround\_time;

float avg\_waiting\_time;

float avg\_response\_time;

float cpu\_utilisation;

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

int total\_response\_time = 0;

int total\_idle\_time = 0;

float throughput;

int burst\_remaining[100];

int idx;

cout << setprecision(2) << fixed;

cout << "Enter the number of processes: ";

cin >> n;

cout << "Enter time quantum: ";

cin >> tq;

for (int i = 0; i < n; i++)

{

cout << "Enter arrival time of process " << i + 1 << ": ";

cin >> p[i].arrival\_time;

cout << "Enter burst time of process " << i + 1 << ": ";

cin >> p[i].burst\_time;

burst\_remaining[i] = p[i].burst\_time;

p[i].pid = i + 1;

cout << endl;

}

sort(p, p + n, compare1);

queue<int> q;

int current\_time = 0;

q.push(0);

int completed = 0;

int mark[100];

memset(mark, 0, sizeof(mark));

mark[0] = 1;

while (completed != n)

{

idx = q.front();

q.pop();

if (burst\_remaining[idx] == p[idx].burst\_time)

{

p[idx].start\_time = max(current\_time, p[idx].arrival\_time);

total\_idle\_time += p[idx].start\_time - current\_time;

current\_time = p[idx].start\_time;

}

if (burst\_remaining[idx] - tq > 0)

{

burst\_remaining[idx] -= tq;

current\_time += tq;

}

else

{

current\_time += burst\_remaining[idx];

burst\_remaining[idx] = 0;

completed++;

p[idx].completion\_time = current\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

}

for (int i = 1; i < n; i++)

{

if (burst\_remaining[i] > 0 && p[i].arrival\_time <= current\_time && mark[i] == 0)

{

q.push(i);

mark[i] = 1;

}

}

if (burst\_remaining[idx] > 0)

{

q.push(idx);

}

if (q.empty())

{

for (int i = 1; i < n; i++)

{

if (burst\_remaining[i] > 0)

{

q.push(i);

mark[i] = 1;

break;

}

}

}

}

avg\_turnaround\_time = (float)total\_turnaround\_time / n;

avg\_waiting\_time = (float)total\_waiting\_time / n;

avg\_response\_time = (float)total\_response\_time / n;

sort(p, p + n, compare2);

displayProcesses(p, n);

cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;

cout << "Average Waiting Time = " << avg\_waiting\_time << endl;

cout << "Average Response Time = " << avg\_response\_time << endl;

}

void PR()

{

int n;

struct process p[100];

float avg\_turnaround\_time;

float avg\_waiting\_time;

float avg\_response\_time;

int total\_turnaround\_time = 0;

int total\_waiting\_time = 0;

int total\_response\_time = 0;

int total\_idle\_time = 0;

int burst\_remaining[100];

int is\_completed[100];

memset(is\_completed, 0, sizeof(is\_completed));

cout << setprecision(2) << fixed;

cout << "Enter the number of processes: ";

cin >> n;

for (int i = 0; i < n; i++)

{

cout << "Enter arrival time of process " << i + 1 << ": ";

cin >> p[i].arrival\_time;

cout << "Enter burst time of process " << i + 1 << ": ";

cin >> p[i].burst\_time;

cout << "Enter priority of the process " << i + 1 << ": ";

cin >> p[i].priority;

p[i].pid = i + 1;

burst\_remaining[i] = p[i].burst\_time;

cout << endl;

}

int current\_time = 0;

int completed = 0;

int prev = 0;

while (completed != n)

{

int idx = -1;

int mx = -1;

for (int i = 0; i < n; i++)

{

if (p[i].arrival\_time <= current\_time && is\_completed[i] == 0)

{

if (p[i].priority > mx)

{

mx = p[i].priority;

idx = i;

}

if (p[i].priority == mx)

{

if (p[i].arrival\_time < p[idx].arrival\_time)

{

mx = p[i].priority;

idx = i;

}

}

}

}

if (idx != -1)

{

if (burst\_remaining[idx] == p[idx].burst\_time)

{

p[idx].start\_time = current\_time;

total\_idle\_time += p[idx].start\_time - prev;

}

burst\_remaining[idx] -= 1;

current\_time++;

prev = current\_time;

if (burst\_remaining[idx] == 0)

{

p[idx].completion\_time = current\_time;

p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

total\_turnaround\_time += p[idx].turnaround\_time;

total\_waiting\_time += p[idx].waiting\_time;

total\_response\_time += p[idx].response\_time;

is\_completed[idx] = 1;

completed++;

}

}

else

{

current\_time++;

}

}

avg\_turnaround\_time = (float)total\_turnaround\_time / n;

avg\_waiting\_time = (float)total\_waiting\_time / n;

avg\_response\_time = (float)total\_response\_time / n;

cout << endl

<< endl;

displayProcesses(p, n);

cout << "Average Turnaround Time = " << avg\_turnaround\_time << endl;

cout << "Average Waiting Time = " << avg\_waiting\_time << endl;

cout << "Average Response Time = " << avg\_response\_time << endl;

}

int main()

{

int choice;

cout << "Choose scheduling algorithm:" << endl;

cout << "1. FCFS (First-Come-First-Serve)" << endl;

cout << "2. SJF (Shortest Job First)" << endl;

cout << "3. SRTF (Shortest Remaining Time First)" << endl;

cout << "4. RR (Round Robin)" << endl;

cout << "5. Priority" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

fcfs();

break;

case 2:

sjf();

break;

case 3:

srtf();

case 4:

RR();

break;

case 5:

PR();

break;

default:

cout << "Invalid choice. Exiting..." << endl;

break;

}

return 0;

}