Project Name: **Dynamic CPU Scheduling Simulator**
Name: **MD. ABSAR**
BSc. Computer Science and Engineering
ID: **21070104**

Abstract

This report presents the development of a Dynamic CPU Scheduling Simulator to evaluate and compare three scheduling algorithms: Shortest Job First (SJF), First-Come, First-Serve (FCFS), and Priority Scheduling. This simulator allows users to input processes with defined attributes and observe their scheduling in real-time, visualizing metrics like waiting and turnaround time to illustrate the impact of each algorithm on CPU performance.

Keywords

CPU Scheduling, FCFS, SJF, Priority Scheduling, Simulation, Gantt Chart.

I. INTRODUCTION

In computer science, efficient CPU scheduling is crucial to optimize system performance. Scheduling algorithms allocate CPU resources to processes, impacting response times, resource utilization, and system throughput. This project implements a simulator to visually compare the performance of three scheduling algorithms—FCFS, SJF, and Priority Scheduling—by allowing the user to observe how different algorithms handle process scheduling based on varying arrival times, burst times, and priorities.

II. BACKGROUND

A. Project

This project aims to provide a practical simulation environment where users can input multiple processes and select among the three scheduling algorithms. By examining the impact of scheduling choices on process order and CPU performance metrics, the simulator serves as a learning tool for understanding the behavior of common scheduling algorithms.

B. Simulator

The simulator is implemented in Python and employs a modular approach to separately handle each scheduling algorithm. It uses Matplotlib to generate Gantt charts that display process execution order, making it easy for users to visually analyze each algorithm’s scheduling effectiveness.

III. PROJECT EVALUATION

The project was divided into several tasks to simplify implementation. Each scheduling algorithm is executed as a standalone function, allowing independent testing and visualization of results. The following functionalities were incorporated:

- **FCFS Scheduling**: Processes are scheduled in the order they arrive.
- **SJF Scheduling**: The process with the shortest burst time is given priority.
- **Priority Scheduling**: Higher-priority processes are scheduled first.

C. Setup Environment

The simulator requires Python with **`matplotlib`** installed for visualizations. Users are prompted to enter details for each process, including arrival time, burst time, and priority, before selecting the scheduling algorithm for simulation.

D. Code in Controller

The scheduling algorithms are implemented as follows:

- **FCFS**: Sorts processes by arrival time.
- **SJF**: Sorts by arrival time, then burst time to prioritize shorter processes.
- **Priority Scheduling**: Sorts by arrival time and priority, with higher-priority processes executed first.

E. Pseudo Code

An example of the SJF pseudo code:

```python
if available_processes:
    shortest = min(available, key=lambda x: x.burst_time)
    execute shortest process
else:
    increment current_time
```

Fig: 01

An example of the FCFS pseudo code:

```plaintext
Function FCFS(processes):
    Sort processes by arrival_time
    current_time = 0

    For each process in processes:
        Set completion_time = max(current_time, arrival_time) + burst_time
        Calculate turnaround_time and waiting_time
        Update current_time
```

Fig: 02

An example of the FCFS pseudo code:

```plaintext
Function Priority_Scheduling(processes):
    Sort processes by arrival_time, then by priority
    current_time = 0

    While processes are not empty:
        Get processes with arrival_time <= current_time
        If available:
            Pick process with highest priority
            Set completion_time, turnaround_time, and waiting_time
            Update current_time
        Else:
            current_time += 1
```

Fig: 03

F. Simulation and Visualization

Each algorithm’s output is visualized using Gantt charts to display the order and duration of process execution. The Gantt charts and average waiting and turnaround times provide insights into each algorithm’s effectiveness.

G. Output

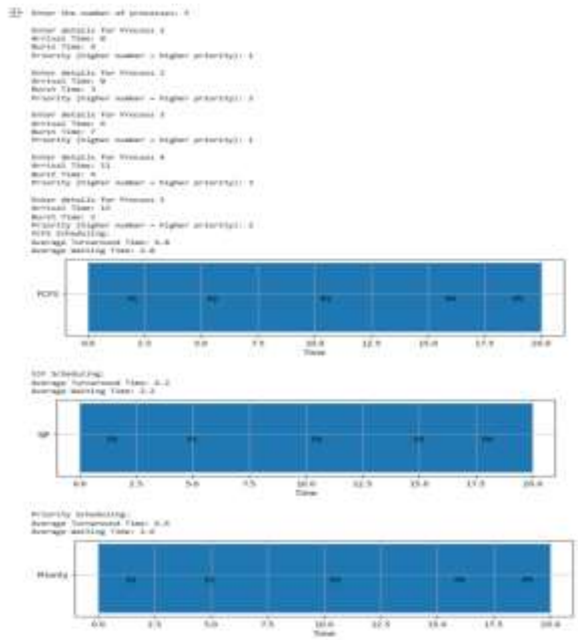


Fig: 04

IV. CRITICAL EVALUATION

The simulator effectively demonstrates differences in CPU performance metrics across algorithms. Key findings include:

- FCFS: Simple and effective for jobs with similar arrival times but may result in long waiting times for later processes.
- SJF: Optimal for minimizing turnaround time, but starvation can occur if short processes continue arriving.
- Priority Scheduling: Allows flexibility in process handling based on importance but may suffer from starvation of low-priority processes.

V. CONCLUSION

The Dynamic CPU Scheduling Simulator successfully visualizes the effects of FCFS, SJF, and Priority Scheduling on CPU performance. The Gantt chart visualizations and calculated metrics provide clear, comparative insights into each algorithm's behavior, making the simulator an effective educational tool.

VI. ACKNOWLEDGMENT

The development of this simulator was made possible by resources and support from the Computer Science and Engineering department, especially in the area of operating systems and process management.

VII. REFERENCES

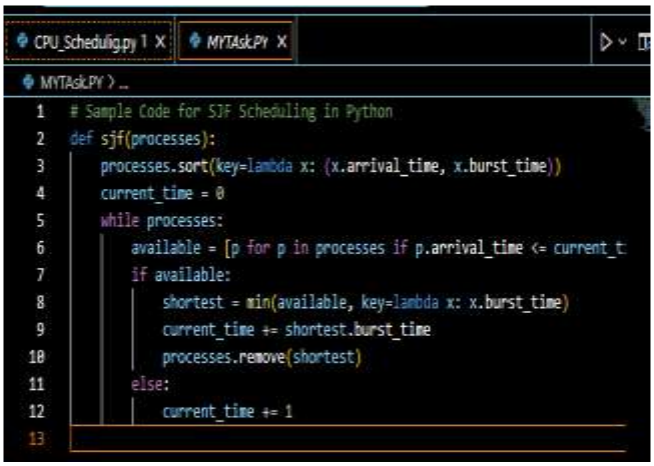
[1] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating System Concepts, 10th ed. Wiley, 2018. [Online]. Available: https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119456339

[2] W. Stallings, Operating Systems: Internals and Design Principles, 9th ed. Pearson, 2017. [Online]. Available: https://www.pearson.com/store/p/operating-systems-internals-and-design-principles/P100000236940

[3] GitHub: https://github.com/absar-mahmud13/Dynamic-CPU-Scheduling-Simulator-Sim/tree/main

VIII. APPENDIX

H. CODE FOR MY TASKS:




```python
# Sample Code for SJF Scheduling in Python
def sjf(processes):
    processes.sort(key=lambda x: (x.arrival_time, x.burst_time))
    current_time = 0
    while processes:
        available = [p for p in processes if p.arrival_time <= current_t
        if available:
            shortest = min(available, key=lambda x: x.burst_time)
            current_time += shortest.burst_time
            processes.remove(shortest)
        else:
            current_time += 1
```

Fig: 05

I. CODE FOR THE OVERALL PROJECT

```python
import matplotlib.pyplot as plt

# Define a class for Process
class Process:
    def __init__(self, process_id, arrival_time, burst_time, priority):
        self.process_id = process_id
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.priority = priority
        self.completion_time = 0
        self.turnaround_time = 0
        self.waiting_time = 0

# First-Come, First-Serve (FCFS) Scheduling
def fcfs(processes):
    processes.sort(key=lambda x: x.arrival_time)  # Sort by arrival time
    current_time = 0
    for process in processes:
        current_time = max(current_time, process.arrival_time) + process.burst_time
        process.completion_time = current_time
        process.turnaround_time = process.completion_time - process.arrival_time
        process.waiting_time = process.turnaround_time - process.burst_time
    return processes

# Shortest Job First (SJF) Scheduling
def sjf(processes):
    processes.sort(key=lambda x: (x.arrival_time, x.burst_time))  # Sort by arrival time, then by burst time
    current_time = 0
    completed = []
    while processes:
        available = [p for p in processes if p.arrival_time <= current_time]
        if available:
            shortest = min(available, key=lambda x: x.burst_time)
            processes.remove(shortest)
            current_time += shortest.burst_time
            shortest.completion_time = current_time
            shortest.turnaround_time = shortest.completion_time - shortest.arrival_time
            shortest.waiting_time = shortest.turnaround_time - shortest.burst_time
            completed.append(shortest)
        else:
            current_time += 1  # Increment time if no process is ready
    return completed

# Priority Scheduling
def priority_scheduling(processes):
    processes.sort(key=lambda x: (x.arrival_time, -x.priority))  # Sort by arrival time, then by priority
    current_time = 0
    completed = []
    while processes:
        available = [p for p in processes if p.arrival_time <= current_time]
        if available:
            highest_priority = max(available, key=lambda x: x.priority)
            processes.remove(highest_priority)
            current_time += highest_priority.burst_time
            highest_priority.completion_time = current_time
            highest_priority.turnaround_time = highest_priority.completion_time - highest_priority.arrival_time
            highest_priority.waiting_time = highest_priority.turnaround_time - highest_priority.burst_time
            completed.append(highest_priority)
        else:
            current_time += 1
    return completed

# Function to calculate and print the metrics
def calculate_metrics(processes):
    total_turnaround_time = sum(p.turnaround_time for p in processes)
    total_waiting_time = sum(p.waiting_time for p in processes)
    avg_turnaround_time = total_turnaround_time / len(processes)
    avg_waiting_time = total_waiting_time / len(processes)
    return avg_turnaround_time, avg_waiting_time

# Function to visualize the Gantt Chart
def visualize_gantt_chart(processes, algorithm_name):
    fig, ax = plt.subplots(figsize=(10, 3))
    start_time = 0
    for process in processes:
        ax.broken_barh([(start_time, process.burst_time)], (10, 9), facecolors=('tab:blue'))
        ax.text(start_time + process.burst_time / 2, 14, f"P{process.process_id}", ha='center')
        start_time += process.burst_time
    ax.set_xlabel('Time')
    ax.set_yticks([15])
    ax.set_yticklabels([algorithm_name])
    ax.grid(True)
    plt.show()

# Function to simulate scheduling and report
def simulate_scheduling(process_list):
    # FCFS Simulation
    fcfs_processes = fcfs(process_list.copy())
    fcfs_avg_tat, fcfs_avg_wt = calculate_metrics(fcfs_processes)
    print("FCFS Scheduling:")
    print("Average Turnaround Time:", fcfs_avg_tat)
    print("Average Waiting Time:", fcfs_avg_wt)
    visualize_gantt_chart(fcfs_processes, "FCFS")

    # SJF Simulation
    sjf_processes = sjf(process_list.copy())
    sjf_avg_tat, sjf_avg_wt = calculate_metrics(sjf_processes)
    print("\nSJF Scheduling:")
    print("Average Turnaround Time:", sjf_avg_tat)
    print("Average Waiting Time:", sjf_avg_wt)
    visualize_gantt_chart(sjf_processes, "SJF")

    # Priority Scheduling Simulation
    priority_processes = priority_scheduling(process_list.copy())
    priority_avg_tat, priority_avg_wt = calculate_metrics(priority_processes)
    print("\nPriority Scheduling:")
    print("Average Turnaround Time:", priority_avg_tat)
    print("Average Waiting Time:", priority_avg_wt)
    visualize_gantt_chart(priority_processes, "Priority")

# Function to get user input for processes
def get_user_input():
    process_list = []
    num_processes = int(input("Enter the number of processes: "))
    for i in range(num_processes):
        print(f"\nEnter details for Process {i + 1}")
        process_id = i + 1
        arrival_time = int(input("Arrival Time: "))
        burst_time = int(input("Burst Time: "))
        priority = int(input("Priority (higher number = higher priority): "))
        process = Process(process_id, arrival_time, burst_time, priority)
        process_list.append(process)
    return process_list

# Main execution
if __name__ == "__main__":
    process_list = get_user_input()
    simulate_scheduling(process_list)
```