LOVELY PROFESSIONAL UNIVERSITY

PROJECT REPORT

Operating Systems

Submitted by :- Ranjandeep Sahu

Section :_ K23RX

Course code :- CSE316

Under the Guidance of :- Gagandeep Kaur

Project Overview

The CPU Scheduling Simulator is designed to help understand and analyze different scheduling algorithms used in operating systems. This project provides a graphical user interface (GUI) to input process details and execute scheduling algorithms such as First Come First Serve (FCFS), Round Robin, and Priority Scheduling. The output includes a Gantt Chart representation, waiting times, and turnaround times for each process.

ABSTRACT

Developing CPU scheduling algorithms and understanding their impact in practice can be difficult and time consuming due to the need to modify and test operating system kernel code and measure the resulting performance on a consistent workload of real applications. As processor is the important resource, CPU scheduling becomes very important in accomplishing the operating system (OS) design goals. The intention should be allowed as many as possible running processes at all time in order to make best use of CPU.

SCHEDULING CRITERIA

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

1. Utilization/Efficiency: keep the CPU busy 100% of the time with useful work
2. Throughput: maximize the number of jobs processed per hour.
3. Turnaround time: from the time of submission to the time of completion, minimize the time batch users must wait for output
4. Waiting time: Sum of times spent in ready queue - Minimize this
5. Response Time: time from submission till the first response is produced, minimize response time for interactive users
6. Fairness: make sure each process gets a fair share of the CPU

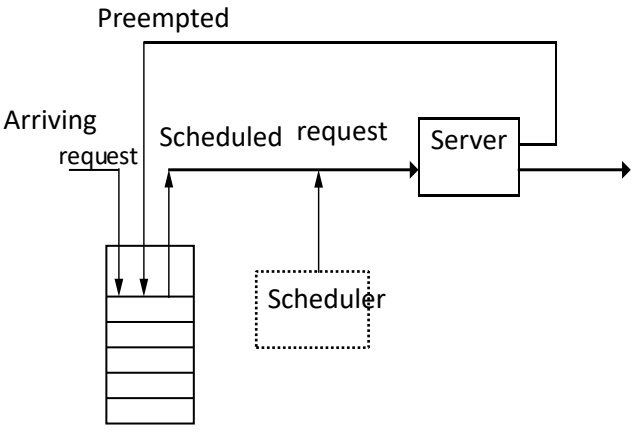


Figure : Schematic of Scheduling

Module-Wise Breakdown

1. Process Class

The Process Class acts as the fundamental data structure for the entire CPU Scheduling Simulator. It defines and stores all the necessary attributes of a process, which are later used by the scheduling algorithms and visualization modules. Here's a breakdown of the key attributes:

- Process ID (PID): A unique identifier assigned to each process. It distinguishes one process from another in both scheduling and visualization.
- Arrival Time: The time at which a process enters the ready queue, i.e., when it becomes available for execution.
- Burst Time: The total time the process requires on the CPU to complete its execution.
- Priority: A numerical value representing the importance of the process in Priority Scheduling. Processes with higher priority (usually denoted by a lower number) are scheduled before lower-priority ones.
- Waiting Time: The total time a process spends in the ready queue waiting for CPU execution. It is computed after scheduling.
- Turnaround Time: The total time taken from the arrival of the process to its completion (Turnaround Time = Completion Time - Arrival Time).

2. Scheduling Algorithms

This module includes the core scheduling logic. Each algorithm follows different rules and is implemented as a separate function or method. Here's a detailed explanation of the three implemented algorithms:

a. First Come First Serve (FCFS) Scheduling

- How it works: Processes are executed strictly in the order of their arrival times.
- Key points: It’s the simplest scheduling algorithm. Once a process starts executing, it runs to completion before the next process begins.
- Advantages: Easy to implement and understand.
- Disadvantages: Can lead to the "convoy effect," where shorter processes are delayed by longer ones.

b. Round Robin Scheduling

- How it works: Processes are executed for a fixed time quantum (also known as a time slice). If a process does not finish within that time, it is preempted and placed at the end of the ready queue, and the next process is executed.
- Key points: This approach ensures fairness and is often used in time-sharing systems.
- Advantages: Reduces waiting time for shorter processes and provides better response times in interactive systems.
- Disadvantages: Performance heavily depends on the chosen time quantum. Too small leads to too many context switches; too large behaves like FCFS.

c. Priority Scheduling

- How it works: Processes are executed based on their priority levels. A process with higher priority is executed before one with a lower priority.
- Key points: Priorities can be static or dynamic. If two processes have the same priority, other criteria (e.g., arrival time) can resolve ties.
- Advantages: Allows important processes to be executed earlier.
- Disadvantages: Can lead to starvation (low-priority processes never get CPU time), unless aging or other techniques are used to prevent it.

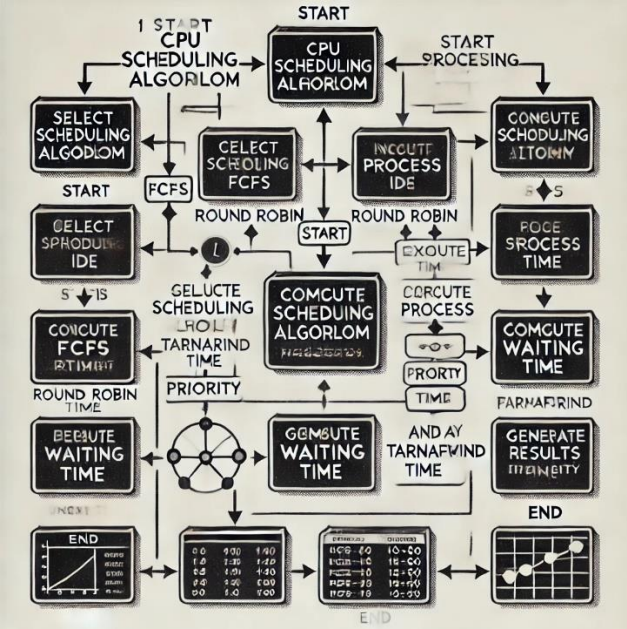
3. Graphical User Interface (GUI) Module

The GUI is developed using Python's Tkinter library. This module provides an interactive interface that makes the simulator user-friendly and intuitive, even for those unfamiliar with programming. Major features include:

- Input Forms: Users can enter process details such as PID, Arrival Time, Burst Time, and Priority through text fields or forms.
- Algorithm Selection: Radio buttons or drop-down menus allow users to select which scheduling algorithm to execute.
- Buttons and Event Handling: Buttons trigger events, such as executing algorithms, displaying results, or clearing input data.

The GUI abstracts the complexity behind simple clicks and inputs, making the simulator accessible and easy to use.

4. **Gantt Chart Visualization**



This module visualizes the sequence of process execution on the CPU over time, using Matplotlib, a powerful plotting library in Python.

- Gantt Chart: A bar chart that represents the execution timeline of each process, showing when it starts and finishes execution.
- Color Coding: Each process is typically represented by a different color for clarity.
- Purpose: It provides an intuitive and visual understanding of how processes are scheduled, how much time each process gets on the CPU, and how CPU time is allocated in various algorithms.

This visualization helps users better grasp the dynamic nature of CPU scheduling.

5. **Execution & Output Display**

After an algorithm executes, this module calculates and displays key performance metrics:

- Waiting Time: How long each process waited before getting CPU time.
- Turnaround Time: How long it took from a process's arrival to its completion.
- Tabular Display: The results are shown in a table format using Tkinter’s Treeview widget. The table typically includes columns for PID, Arrival Time, Burst Time, Priority, Waiting Time, and Turnaround Time.

This output allows users to compare algorithm efficiency and understand the impact of scheduling policies on system performance

Functionalities

Your CPU Scheduling Simulator is designed with a set of core functionalities aimed at providing an interactive, educational, and comprehensive experience for users who wish to understand and analyze CPU scheduling algorithms. Below is a detailed breakdown and explanation of each functionality implemented in the project:

_____

1. Input Process Details (PID, Arrival Time, Burst Time, and Priority)

What it does:

The simulator provides an interface for users to input the necessary details of each process that needs to be scheduled.

Explanation:

- Process ID (PID): Acts as a unique identifier for each process. It helps differentiate processes in the input, output tables, and Gantt Chart visualization.
- Arrival Time: Indicates when a process becomes ready to enter the CPU queue. It's an essential parameter for determining the order of execution in scheduling algorithms like FCFS and Priority Scheduling.
- Burst Time: The total amount of time the CPU needs to execute the process. It directly affects the turnaround and waiting time calculations.
- Priority: Used specifically in Priority Scheduling algorithms to determine which process should be executed first based on its priority level (lower numbers typically represent higher priority).

How it's implemented:

- In the GUI, text fields or form entries are available for the user to input these details for multiple processes.
- Validation ensures the user provides complete and valid data before proceeding.

Technology Used

- **Programming Languages**: Python
- **Libraries and Tools**: Tkinter (GUI), Matplotlib (Chart Visualization)
- **Other Tools**: Treeview for table representation

Flow Diagram The following flowchart illustrates the execution of the CPU Scheduling Simulator:

This flowchart represents the execution of a CPU Scheduling Simulator, illustrating the sequential steps involved in selecting a scheduling algorithm, inputting process details, executing the algorithm, and displaying results.

Step-by-Step Breakdown:

1. Start the Application

o The user launches the simulator, initializing the program.

2. Select the Scheduling Algorithm

o The user chooses one of the available scheduling algorithms:

 First-Come, First-Served (FCFS)

 Round Robin (RR)

 Priority Scheduling

 (Other possible scheduling algorithms can be included)

3. Input Process Details

o The user enters the following process details:

 Process ID (PID) – Unique identifier for each process

 Arrival Time – When the process arrives in the queue

 Burst Time – Execution time required by the process

 Priority – (Only applicable in Priority Scheduling)

4. Execute the Selected Algorithm

o The simulator runs the chosen scheduling algorithm to determine execution order.

5. Compute Waiting Time and Turnaround Time

o Waiting Time (WT): The time a process spends waiting in the queue before execution.

o Turnaround Time (TAT): The total time from arrival to completion of a process.

6. Display Results in a Table

o The computed results (WT and TAT) for each process are displayed in a structured tabular format.

7. Generate and Display the Gantt Chart

o A Gantt Chart is generated to visually represent process execution order over time.

8. End

o The simulation ends, and the user may restart or exit the program.

Conclusion and Future Scope

The CPU Scheduling Simulator successfully fulfills its intended purpose of simulating and analyzing various CPU scheduling algorithms in a graphical and interactive environment. By offering support for First Come First Serve (FCFS), Round Robin, and Priority Scheduling algorithms, the simulator provides users—especially students and those new to operating systems—with a practical and hands-on tool to understand complex theoretical concepts.

References

- Operating System Concepts by Silberschatz, Galvin, Gagne
- Python Documentation (Tkinter, Matplotlib)
- Online Resources on Scheduling Algorithms

B. **Problem Statement** :- **Intelligent CPU Scheduler Simulator**

Description: Develop a simulator for CPU scheduling algorithms (FCFS, SJF, Round Robin, Priority Scheduling) with real-time visualizations. The simulator should allow users to input processes with arrival times, burst times, and priorities and visualize Gantt charts and performance metrics like average waiting time and turnaround time.

C. **Solution/Code** :-

```
import tkinter as tk

from tkinter import ttk, messagebox

import matplotlib.pyplot as plt

import copy


class Process:

    def __init__(self, pid, arrival_time, burst_time, priority=0):

        self.pid = pid

        self.arrival_time = arrival_time

        self.burst_time = burst_time

        self.remaining_time = burst_time

        self.priority = priority

        self.waiting_time = 0

        self.turnaround_time = 0


# FCFS Scheduling
```

```
def fcfs_scheduling(processes):
    processes.sort(key=lambda x: x.arrival_time)
    current_time = 0
    gantt_chart = []
    for process in processes:
        start_time = max(current_time, process.arrival_time)
        gantt_chart.append((process.pid, start_time, start_time + process.burst_time))
        process.waiting_time = start_time - process.arrival_time
        process.turnaround_time = process.waiting_time + process.burst_time
        current_time = start_time + process.burst_time
    return processes, gantt_chart

# Round Robin Scheduling
def round_robin_scheduling(processes, quantum):
    queue = sorted(copy.deepcopy(processes), key=lambda x: x.arrival_time)
    current_time = 0
    gantt_chart = []
    ready_queue = []
    process_map = {p.pid: p for p in processes}
    while queue or ready_queue:
        if not ready_queue and queue:
            current_time = max(current_time, queue[0].arrival_time)
        while queue and queue[0].arrival_time <= current_time:
            ready_queue.append(queue.pop(0))
        if ready_queue:
            process = ready_queue.pop(0)
            execute_time = min(process.remaining_time, quantum)
            gantt_chart.append((process.pid, current_time, current_time + execute_time))
            process.remaining_time -= execute_time
            current_time += execute_time
            if process.remaining_time > 0:
                while queue and queue[0].arrival_time <= current_time:
```

```
                ready_queue.append(queue.pop(0))
            ready_queue.append(process)
        else:
            original = process_map[process.pid]
            original.turnaround_time = current_time - original.arrival_time
            original.waiting_time = original.turnaround_time - original.burst_time
    return list(process_map.values()), gantt_chart

# SJF Scheduling (Non-preemptive)
def sjf_scheduling(processes):
    processes = copy.deepcopy(processes)
    completed = []
    gantt_chart = []
    current_time = 0
    while processes:
        available = [p for p in processes if p.arrival_time <= current_time]
        if available:
            process = min(available, key=lambda p: p.burst_time)
        else:
            current_time = min(processes, key=lambda p: p.arrival_time).arrival_time
            continue
        start_time = current_time
        end_time = start_time + process.burst_time
        gantt_chart.append((process.pid, start_time, end_time))
        process.waiting_time = start_time - process.arrival_time
        process.turnaround_time = process.waiting_time + process.burst_time
        current_time = end_time
        completed.append(process)
        processes.remove(process)
    return completed, gantt_chart

# Priority Scheduling
```

```
def priority_scheduling(processes):
    processes.sort(key=lambda x: (x.arrival_time, x.priority))
    current_time = 0
    gantt_chart = []
    for process in processes:
        start_time = max(current_time, process.arrival_time)
        gantt_chart.append((process.pid, start_time, start_time + process.burst_time))
        process.waiting_time = start_time - process.arrival_time
        process.turnaround_time = process.waiting_time + process.burst_time
        current_time = start_time + process.burst_time
    return processes, gantt_chart

# Draw Gantt Chart
def draw_gantt_chart(gantt_chart):
    fig, ax = plt.subplots(figsize=(8, 4))
    colors = plt.cm.get_cmap('tab20', len(gantt_chart))
    for idx, (process_id, start, end) in enumerate(gantt_chart):
        ax.barh(y=0, width=end - start, left=start, height=0.5, align='center', edgecolor='black', color=colors(idx))
        ax.text((start + end) / 2, 0, f"P{process_id}", ha='center', va='center', color='white')
    ax.set_xlabel("Time")
    ax.set_yticks([])
    ax.set_title("Gantt Chart")
    plt.tight_layout()
    plt.show()

# Execute Scheduling
def execute_scheduling():
    try:
        processes = []
        for row in process_rows:
            pid = int(row[0].get())
            arrival_time = int(row[1].get())
```

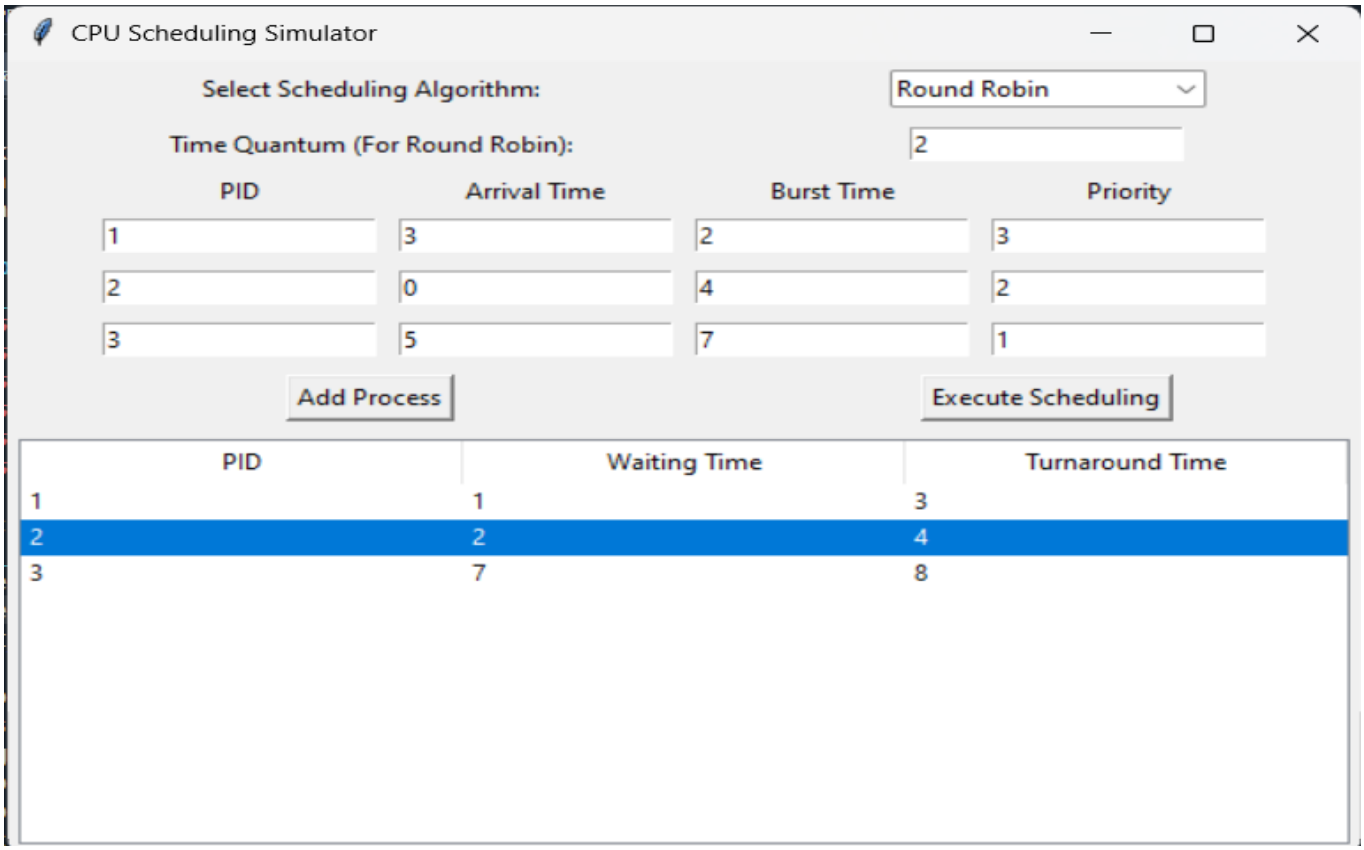
```
    burst_time = int(row[2].get())
    priority = int(row[3].get()) if priority_shown else 0
    processes.append(Process(pid, arrival_time, burst_time, priority))

selected_algorithm = algorithm_var.get()
if selected_algorithm == "FCFS":
    result, gantt_chart = fcfs_scheduling(processes)
elif selected_algorithm == "Round Robin":
    if not quantum_entry.get().isdigit():
        messagebox.showerror("Error", "Please enter a valid time quantum.")
        return
    quantum = int(quantum_entry.get())
    result, gantt_chart = round_robin_scheduling(processes, quantum)
elif selected_algorithm == "Priority Scheduling":
    result, gantt_chart = priority_scheduling(processes)
elif selected_algorithm == "SJF":
    result, gantt_chart = sjf_scheduling(processes)
else:
    messagebox.showerror("Error", "Please select a scheduling algorithm")
    return

for item in table.get_children():
    table.delete(item)

total_waiting = 0
total_turnaround = 0

for process in result:
    table.insert("", "end", values=(process.pid, process.waiting_time, process.turnaround_time))
    total_waiting += process.waiting_time
    total_turnaround += process.turnaround_time
```

```
        avg_waiting = total_waiting / len(result)
        avg_turnaround = total_turnaround / len(result)

        avg_label.config(text=f"Average Waiting Time: {avg_waiting:.2f}, Turnaround Time: {avg_turnaround:.2f}")
        draw_gantt_chart(gantt_chart)
    except ValueError:
        messagebox.showerror("Error", "Please enter valid numeric values for all fields.")

# Update UI elements visibility based on selected algorithm
def update_priority_visibility(event=None):
    global priority_shown
    algorithm = algorithm_var.get()

    # Show/hide priority column
    priority_shown = algorithm == "Priority Scheduling"
    for row in process_rows:
        if priority_shown:
            row[3].grid()
        else:
            row[3].grid_remove()

    # Show/hide quantum entry
    if algorithm == "Round Robin":
        quantum_label.grid()
        quantum_entry.grid()
    else:
        quantum_label.grid_remove()
        quantum_entry.grid_remove()

# Add Process Row
def add_process_row():
    row_index = len(process_rows) + 1
```

```
    row_entries = []
    for col in range(4):
        entry = tk.Entry(frame)
        entry.grid(row=row_index, column=col, padx=5, pady=5)
        if col == 3 and not priority_shown:
            entry.grid_remove()
        row_entries.append(entry)
    process_rows.append(row_entries)

# GUI Setup
root = tk.Tk()
root.title("CPU Scheduling Simulator")

priority_shown = False

# Algorithm Selection
tk.Label(root, text="Select Scheduling Algorithm:").grid(row=0, column=0, padx=5, pady=5)
algorithm_var = tk.StringVar()
algorithm_dropdown = ttk.Combobox(root, textvariable=algorithm_var, values=["FCFS", "Round Robin", "Priority Scheduling", "SJF"])
algorithm_dropdown.grid(row=0, column=1, padx=5, pady=5)
algorithm_dropdown.bind("<<ComboboxSelected>>", update_priority_visibility)

# Quantum Input
quantum_label = tk.Label(root, text="Time Quantum (For Round Robin):")
quantum_entry = tk.Entry(root)
quantum_label.grid(row=1, column=0, padx=5, pady=5)
quantum_entry.grid(row=1, column=1, padx=5, pady=5)
quantum_label.grid_remove()
quantum_entry.grid_remove()

# Process Table
frame = tk.Frame(root)
```

```
frame.grid(row=2, column=0, columnspan=2)
for i, label in enumerate(["PID", "Arrival Time", "Burst Time", "Priority"]):
    tk.Label(frame, text=label).grid(row=0, column=i)

process_rows = []

# Add Process Button
add_process_button = tk.Button(root, text="Add Process", command=add_process_row)
add_process_button.grid(row=3, column=0, padx=5, pady=5)

# Execute Button
execute_button = tk.Button(root, text="Execute Scheduling", command=execute_scheduling)
execute_button.grid(row=3, column=1, padx=5, pady=5)

# Result Table
table = ttk.Treeview(root, columns=("PID", "Waiting Time", "Turnaround Time"), show="headings")
table.grid(row=4, column=0, columnspan=2, padx=5, pady=5)
for col in ["PID", "Waiting Time", "Turnaround Time"]:
    table.heading(col, text=col)

# Average Time Label
avg_label = tk.Label(root, text="")
avg_label.grid(row=5, column=0, columnspan=2, pady=10)

root.mainloop()
```

Outcome:-



Gantt Chart:-