NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY

MILITARY COLLEGE OF SIGNALS

Operating Systems

(CS-330)

Complex Engineering Problem

Submitted by:

1. Muhammad Ahmad Sultan
2. Muhammad Salaar
3. Haris Bin Amir

Course: BESE-28

Section: C

Submitted to: LE Hafiz M. Tayyab

Dated: 27-06-2024

CATALOG OF COMPLEX ENGINEERING PROBLEM

Table of Contents

DEPARTMENT OF COMPUTER SOFTWARE ENGINEERING
Military College of Signals
National University of Sciences and Technology

www.mcs.nust.edu.pk

Complex Engineering Problem

Semester Design Project

Process Scheduling Simulator

▪ Problem Statement

Process scheduling is a core concept of operating systems. Students are required to **Design and Implement a Process Scheduling Simulator with Multiple Scheduling Algorithms.**

By successfully completing this project, students will gain a deeper understanding of process scheduling algorithms and their impact on system performance, as well as practical experience in software development, simulation, and performance evaluation.

▪ Abstract

Process scheduling is a fundamental concept in operating systems, crucial for efficient CPU resource allocation and ensuring optimal system performance. This project aims to design and implement a Process Scheduling Simulator that incorporates multiple scheduling algorithms, including First Come First Serve (FCFS), Shortest Job First (SJF), Round Robin (RR), Priority Scheduling, and Multilevel Queue Scheduling. By simulating and evaluating these algorithms, this project provides insights into their performance metrics such as average turnaround time and average waiting time, as well as their impact on system efficiency. The simulator offers a user-friendly interface for loading processes, selecting algorithms, and visualizing execution through Gantt charts, facilitating a comprehensive understanding of how different scheduling policies affect process management in an operating system environment.

▪ Objectives

I. Scheduling Algorithms Implementation

- **First Come First Serve (FCFS):** Processes are scheduled in the order they arrive.
- **Shortest Job First (SJF):** Processes with the shortest burst time are scheduled first.
- **Round Robin (RR):** Processes are scheduled in a cyclic order with a fixed time quantum.
- **Priority Scheduling:** Processes with higher priority are scheduled first.
- **Multilevel Queue Scheduling:** Processes are divided into multiple queues based on priority levels, each with different scheduling policies.

II. Input Data Parsing

- **Process Input File:** Load processes from a file containing process details such as PID, arrival time, burst time, and priority.
- **Error Handling:** Ensure robustness by handling incorrect or malformed input data gracefully.
- **Data Storage:** Store parsed process data in a suitable data structure for efficient access during scheduling.

III. Gantt Chart Generation

- **Visualization:** Use Matplotlib to create Gantt charts that visualize the start and end times of each process.
- **Process Differentiation:** Utilize different colors to differentiate between processes.
- **Interactive Display:** Allow users to view the Gantt chart for the selected scheduling algorithm.

IV. Performance Evaluation

- **Turnaround Time:** Calculate the average turnaround time for each scheduling algorithm.
- **Waiting Time:** Calculate the average waiting time for each scheduling algorithm.
- **Comparison:** Compare the performance metrics of different algorithms to highlight their advantages and disadvantages.
- **Efficiency Metrics:** Include other relevant metrics, such as CPU utilization and throughput, if applicable.

V. User Interface

- **Tkinter GUI:** Develop a user-friendly graphical interface using Tkinter.
- **Process Table:** Display process details in a structured table format.
- **Control Buttons:** Provide buttons for loading processes, selecting algorithms, and generating results.
- **Metrics Display:** Show calculated performance metrics in a dedicated section of the interface.
- **Interactive Elements:** Ensure the interface is intuitive and responsive, allowing easy interaction.

VI. Documentation and Testing

- **Code Documentation:** Write clear and concise documentation for the codebase, including inline comments and function descriptions.
- **User Manual:** Create a user manual detailing how to use the simulator, load processes, select algorithms, and interpret results.
- **Test Cases:** Develop comprehensive test cases to validate the correctness of each scheduling algorithm.
- **Error Handling:** Test the simulator against various edge cases and handle errors gracefully.
- **Performance Testing:** Evaluate the simulator's performance with different input sizes to ensure scalability.

▪ Scope

The project covers the implementation and comparison of process scheduling algorithms in a simulated environment. The simulator will:

Parse Input Data from a File

- Read process details such as Process ID, Arrival Time, Burst Time, and Priority from a text file.
- Validate and handle potential errors or inconsistencies in the input file format.

Implement and Simulate Different Scheduling Algorithms

- Develop functions for multiple scheduling algorithms including FCFS, SJF, RR, Priority Scheduling, and Multilevel Queue Scheduling.
- Ensure each algorithm adheres to its respective scheduling principles and policies.

Generate Gantt Charts for Visualization

- Create visual representations of process execution timelines using Gantt charts.
- Display start and end times, context switches, and interruptions clearly.

Calculate and Display Performance Metrics

- Compute average turnaround time and average waiting time for each scheduling algorithm.
- Compare the performance metrics to evaluate the effectiveness of different scheduling policies.

User Interface

- Develop a user-friendly GUI using Tkinter to facilitate interaction with the simulator.
- Allow users to load processes, select scheduling algorithms, customize parameters, and view results easily.

Documentation and Testing

- Provide comprehensive documentation detailing the design, implementation, and usage of the scheduling simulator.
- Conduct thorough testing to ensure the correctness and robustness of the program, including handling edge cases and boundary conditions.

▪ Beyond Current Focus

Future iterations of the project could extend to:

Distributed Scheduling

- Simulate scheduling algorithms in a distributed system environment.
- Handle process scheduling across multiple CPUs or nodes, taking communication delays and resource sharing into account.

Integration with Real Systems

- Extend the simulator to interface with actual operating systems or hardware to observe real-world performance.
- Analyze the impact of scheduling algorithms on real-time processes and system responsiveness.

User-defined Algorithms

- Provide a framework for users to define and test their custom scheduling algorithms.
- Allow users to input custom logic and parameters to evaluate new or experimental scheduling policies.

Advanced Visualization Techniques

- Enhance the visualization module to support interactive Gantt charts and other advanced graphical representations.
- Allow users to zoom in/out, filter, and manipulate the visualization for deeper analysis and understanding.

▪ Tools and Technologies Used

- **Programming Language:** Python
- **GUI Library:** Tkinter
- **Data Visualization:** Matplotlib

- **Development Environment:** Visual Studio Code

▪ Design and Implementation

Design

The Process Scheduling Simulator is designed to facilitate the understanding and comparison of various process scheduling algorithms through a simulated environment. The simulator comprises a frontend for user interaction and a backend for algorithm implementation.

Frontend

Tkinter GUI

The frontend is developed using Tkinter, a standard Python library for creating graphical user interfaces. The GUI provides a user-friendly interface for loading process details, selecting scheduling algorithms, and displaying the results. Key components of the frontend include:

- **Process Table (Treeview):** Displays the list of processes along with their attributes such as PID, Arrival Time, Burst Time, and Priority.
- **Load Button:** Allows users to load processes from a text file.
- **Algorithm Buttons:** Provides buttons for running different scheduling algorithms such as FCFS, SJF, RR, Priority Scheduling, and Multilevel Queue Scheduling.
- **Gantt Chart Display:** Visualizes the execution timeline of processes using Matplotlib.
- **Metrics Display:** Shows performance metrics such as average turnaround time and average waiting time.
- **Time Quantum Entry:** Allows users to input the time quantum for the Round Robin scheduling algorithm.
- **Status Label:** Displays the status of the processes (loaded or not).
- **Display Stack Button:** Displays the loaded processes in a separate message box.

Backend

Process Class

The backend includes a Process class that represents each process with the following attributes:

- **pid:** Process ID.
- **arrival_time:** Arrival time of the process.
- **burst_time:** Burst time required for execution.
- **priority:** Priority of the process.
- **remaining_time:** Remaining burst time (initially set to burst time).
- **start_time:** Start time of execution.
- **completion_time:** Completion time of execution.

Scheduling Algorithms

The backend implements several scheduling algorithms, each as a separate function:

- **FCFS (First Come First Serve):** Processes are scheduled in the order they arrive.
- **SJF (Shortest Job First):** Processes with the shortest burst time are scheduled first.
- **RR (Round Robin):** Processes are scheduled in a cyclic order with a fixed time quantum.
- **Priority Scheduling:** Processes with higher priority are scheduled first.
- **Multilevel Queue Scheduling:** Processes are divided into multiple queues based on priority levels, with each queue having different scheduling policies.

Gantt Chart Generation

Gantt charts are generated using Matplotlib to visualize the start and end times of each process. Different colors are used to differentiate processes, providing a clear visual representation of process execution timelines.

Performance Metrics Calculation

The simulator calculates and displays performance metrics for each scheduling algorithm, including:

- **Average Turnaround Time:** The average time taken from the arrival to the completion of the processes.
- **Average Waiting Time:** The average time the processes spend waiting in the queue.

Implementation

Frontend (Tkinter)

The frontend of the Process Scheduling Simulator is developed using Tkinter, a Python library for creating graphical user interfaces. The GUI is designed to provide an intuitive interface for users to load process details, select scheduling algorithms, and view results.

Main Components

Process Table (Treeview)

- Displays the details of each process, including Process ID (PID), Arrival Time, Burst Time, and Priority.
- Allows users to see the list of loaded processes in a tabular format.

Load Button

- Enables users to load process details from a text file.
- The loaded processes are displayed in the Process Table.
- Provides feedback on the loading status through a status label.

Algorithm Buttons

Provides buttons for each scheduling algorithm:

- FCFS (First Come First Serve)
- SJF (Shortest Job First)
- RR (Round Robin)
- Priority Scheduling
- Multilevel Queue Scheduling

Users can click on these buttons to run the selected scheduling algorithm on the loaded processes.

Gantt Chart Display

- Visualizes the execution timeline of processes using Matplotlib.
- Displays the start and end times of each process in a graphical format.
- Uses different colors to differentiate between processes.

Metrics Display

- Shows performance metrics such as average turnaround time and average waiting time.
- Provides a text area below the buttons where these metrics are displayed after running a scheduling algorithm.

Additional Components

Time Quantum Entry

- Allows users to input the time quantum for the Round Robin scheduling algorithm.
- The default value is set to 2 but can be adjusted by the user.

Status Label

- Displays the current status of the processes (e.g., whether they have been loaded successfully or not).

Display Stack Button

- Displays the stack of loaded processes in a separate message box.
- Provides a way for users to verify the loaded processes before running any scheduling algorithm.

Backend (Scheduling Algorithms)

The backend of the Process Scheduling Simulator is responsible for implementing the scheduling algorithms. Each algorithm is implemented as a separate function, allowing for modularity and ease of testing.

Scheduling Algorithms

1. FCFS (First Come First Serve)

- Processes are scheduled in the order they arrive.
- The algorithm iterates through the processes based on their arrival time and schedules them one after another.

2. SJF (Shortest Job First)

- Processes with the shortest burst time are scheduled first.
- The algorithm selects the process with the shortest burst time among the available processes and schedules it next.

3. RR (Round Robin)

- Processes are scheduled in a cyclic order with a fixed time quantum.
- The algorithm assigns a fixed time slice to each process in a cyclic manner, ensuring that no process starves.
- Utilizes a time quantum input by the user to determine the time slice for each process.

4. Priority Scheduling

- Processes with higher priority are scheduled first.
- The algorithm selects the process with the highest priority among the available processes and schedules it next.

5. **Multilevel Queue Scheduling**

- Processes are divided into multiple queues based on priority levels.
- Each queue has different scheduling policies.
- The algorithm schedules processes from the high-priority queue first, followed by lower-priority queues.

Helper Functions

1. **Process Class**

- Represents each process with attributes such as PID, Arrival Time, Burst Time, Priority, Remaining Time, Start Time, and Completion Time.
- Facilitates the management of process details throughout the scheduling simulation.

2. **Load Processes**

- Reads process details from a text file.
- Initializes a list of Process objects based on the file content.
- Handles errors and provides feedback if the process loading fails.

3. **Calculate Metrics**

- Calculates performance metrics such as average turnaround time and average waiting time.
- Uses the completion time and arrival time of processes to compute these metrics.

- Code

Frontend (Tkinter)

# frontend.py

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox, Toplevel
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from Backend import *

class SchedulingSimulatorApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Process Scheduling Simulator")
        self.processes = []  # To store loaded processes
        self.processes_loaded = False  # Flag to track whether processes are
loaded
        self.frame = None
        self.status_label = None
        self.processes_tree = None  # Treeview to display processes as a table
        self.metrics_text = None
        self.time_quantum_entry = None
        self.canvas = None  # To store the canvas widget

        self.create_widgets()

    def create_widgets(self):
        self.frame = ttk.Frame(self.root, padding="10")
        self.frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

        self.processes_label = ttk.Label(self.frame, text="Processes:")
        self.processes_label.grid(row=0, column=0, sticky=tk.W)

        # Create a Treeview widget to display processes as a table
        self.processes_tree = ttk.Treeview(self.frame, columns=("PID",
"Arrival Time", "Burst Time", "Priority"), show="headings", height=10)
        self.processes_tree.heading("PID", text="PID")
        self.processes_tree.heading("Arrival Time", text="Arrival Time")
        self.processes_tree.heading("Burst Time", text="Burst Time")
        self.processes_tree.heading("Priority", text="Priority")
        self.processes_tree.grid(row=1, column=0, columnspan=2, sticky=(tk.W,
tk.E))
```

```
        self.load_button = ttk.Button(self.frame, text="Load Processes",
command=self.load_processes)
        self.load_button.grid(row=2, column=0, pady=5)

        # Buttons for scheduling algorithms (FCFS, SJF, RR, Priority)
        self.run_fcfs_button = ttk.Button(self.frame, text="Run FCFS
Scheduling", command=lambda: self.run_algorithm(fcfs, "FCFS Scheduling"))
        self.run_fcfs_button.grid(row=3, column=0, pady=5)

        self.run_sjf_button = ttk.Button(self.frame, text="Run SJF
Scheduling", command=lambda: self.run_algorithm(sjf, "SJF Scheduling"))
        self.run_sjf_button.grid(row=3, column=1, pady=5)

        self.run_rr_button = ttk.Button(self.frame, text="Run RR Scheduling",
command=lambda: self.run_algorithm(round_robin, "Round Robin Scheduling"))
        self.run_rr_button.grid(row=4, column=0, pady=5)

        self.run_priority_button = ttk.Button(self.frame, text="Run Priority
Scheduling", command=lambda: self.run_algorithm(priority_scheduling, "Priority
Scheduling"))
        self.run_priority_button.grid(row=4, column=1, pady=5)

        self.run_multilevel_button = ttk.Button(self.frame, text="Run
Multilevel Queue Scheduling", command=lambda: self.run_algorithm(lambda
processes: multilevel_queue(processes, int(self.time_quantum_entry.get())),
"Multilevel Queue Scheduling"))
        self.run_multilevel_button.grid(row=5, column=0, pady=5, columnspan=2)

        self.time_quantum_label = ttk.Label(self.frame, text="Time Quantum:")
        self.time_quantum_label.grid(row=6, column=0, sticky=tk.E)

        self.time_quantum_entry = ttk.Entry(self.frame)
        self.time_quantum_entry.grid(row=6, column=1, sticky=tk.W)
        self.time_quantum_entry.insert(0, "2")

        self.metrics_label = ttk.Label(self.frame, text="Metrics:")
        self.metrics_label.grid(row=7, column=0, sticky=tk.W)

        self.metrics_text = tk.Text(self.frame, height=5, width=40)
        self.metrics_text.grid(row=8, column=0, columnspan=2, sticky=(tk.W,
tk.E))

        self.status_label = ttk.Label(self.frame, text="")
        self.status_label.grid(row=2, column=1, pady=5)

        self.stack_button = ttk.Button(self.frame, text="Display Stack",
command=self.display_stack)
```

```
        self.stack_button.grid(row=9, column=0, pady=5, columnspan=2)

    def reset_processes(self, processes):
        for process in processes:
            process.remaining_time = process.burst_time
            process.start_time = None
            process.completion_time = None

    def load_processes(self):
        if self.processes_loaded:
            messagebox.showinfo("Processes Already Loaded", "Processes are
already loaded.")
            return

        file_path = filedialog.askopenfilename(filetypes=[("Text Files",
"*.txt"), ("All Files", "*.*")])
        if file_path:
            try:
                self.processes = load_processes(file_path)
                self.update_processes_treeview()
                self.status_label.config(text="Processes loaded
successfully.")
                self.processes_loaded = True  # Update flag
            except RuntimeError as e:
                messagebox.showerror("Error", str(e))
                self.status_label.config(text="Error loading processes.")

    def update_processes_treeview(self):
        # Clear existing items in the treeview
        for item in self.processes_tree.get_children():
            self.processes_tree.delete(item)

        # Insert loaded processes into the treeview
        for process in self.processes:
            self.processes_tree.insert("", tk.END, values=(process.pid,
process.arrival_time, process.burst_time, process.priority))

    def display_stack(self):
        stack_text = ""
        for process in self.processes:
            stack_text += f"PID: {process.pid}, Arrival:
{process.arrival_time}, Burst: {process.burst_time}, Priority:
{process.priority}\n"
        messagebox.showinfo("Loaded Processes Stack", stack_text)

    def run_algorithm(self, algorithm_func, title):
        if not self.processes_loaded:
            messagebox.showwarning("No Processes", "Please load processes
first.")
            return

        try:
            # Reset processes to their initial state before running the
algorithm
            self.reset_processes(self.processes)

            if algorithm_func == round_robin:
                time_quantum = int(self.time_quantum_entry.get())
                scheduled_processes, intervals =
algorithm_func(self.processes, time_quantum)
            else:
                scheduled_processes = algorithm_func(self.processes.copy())  #
Ensure processes list is copied
                intervals = None

            if not scheduled_processes:
                messagebox.showwarning("No Processes Scheduled", "No processes
were scheduled.")
                return

            fig = self.generate_gantt_chart(scheduled_processes, intervals,
title)
            self.open_gantt_chart_window(fig, title)

            # Display metrics only after chart is displayed
            self.display_metrics(scheduled_processes)

        except ValueError as ve:
            messagebox.showerror("Value Error", f"Invalid value: {ve}")
        except RuntimeError as re:
            messagebox.showerror("Runtime Error", f"Error: {re}")
        except Exception as e:
            messagebox.showerror("Error", f"Failed to run {title}: {e}")

    def generate_gantt_chart(self, processes, intervals, title):
        fig, gnt = plt.subplots()
        gnt.set_xlabel('Time')
        gnt.set_ylabel('Processes')
        gnt.set_title(title)

        # Define colors for processes or intervals
        colors = ['tab:blue', 'tab:orange', 'tab:green', 'tab:red',
'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan']
        color_index = 0
        color_map = {}
        try:
            if intervals:
                for pid, interval_list in intervals.items():
                    color = colors[color_index % len(colors)]
                    color_map[pid] = color
                    for start, duration in interval_list:
                        gnt.broken_barh([(start, duration)], (pid * 10, 9), 
facecolors=(color), label=f'PID {pid}')
                    color_index += 1
            else:
                for process in processes:
                    if process.start_time is not None and 
process.completion_time is not None:
                        color = colors[color_index % len(colors)]
                        color_map[process.pid] = color
                        gnt.broken_barh([(process.start_time, 
process.burst_time)], (process.pid * 10, 9), facecolors=(color), label=f'PID 
{process.pid}')
                        color_index += 1

            # Add legend
            handles = [plt.Rectangle((0,0),1,1, color=color_map[pid]) for pid 
in sorted(color_map.keys())]
            labels = [f'PID {pid}' for pid in sorted(color_map.keys())]
            gnt.legend(handles, labels, loc='upper right', 
bbox_to_anchor=(1.15, 1))

        except Exception as e:
            print(f"Error in generate_gantt_chart: {e}")
            raise  # Reraise the exception to see the full traceback

        return fig


    def open_gantt_chart_window(self, fig, title):
        chart_window = Toplevel(self.root)
        chart_window.title(title)
        chart_canvas = FigureCanvasTkAgg(fig, master=chart_window)
        chart_canvas.draw()
        chart_canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, 
expand=True)

    def display_metrics(self, processes):
        avg_turnaround_time, avg_waiting_time = calculate_metrics(processes)
        self.metrics_text.delete(1.0, tk.END)
        self.metrics_text.insert(tk.END, f"Average Turnaround Time: 
{avg_turnaround_time:.2f}\n")
```

```
        self.metrics_text.insert(tk.END, f"Average Waiting Time:
{avg_waiting_time:.2f}\n")

# Main entry point of the application
if __name__ == "__main__":
    root = tk.Tk()
    app = SchedulingSimulatorApp(root)
    root.mainloop()
```

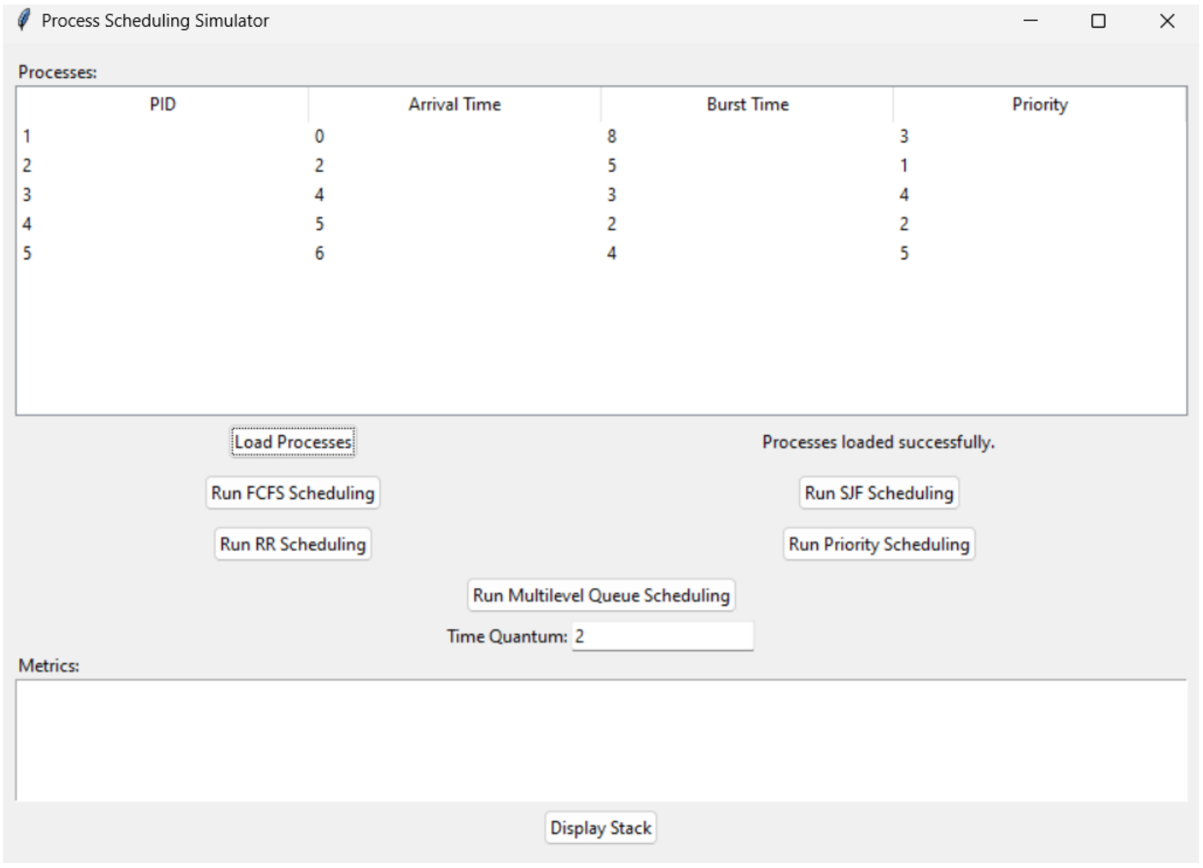
Backend (Algorithms and Helper Functions)

# backend.py

```
# backend.py

class Process:
    def __init__(self, pid, arrival_time, burst_time, priority):
        self.pid = pid                          # Process ID
        self.arrival_time = arrival_time        # Arrival time of the process
        self.burst_time = burst_time            # Burst time required for
execution
        self.priority = priority                # Priority of the process
        self.remaining_time = burst_time        # Remaining burst time initially
set to burst time
        self.start_time = None                  # Start time of execution
(initialized later)
        self.completion_time = None             # Completion time of execution
(initialized later)

def load_processes(file_path):
    """Load processes from a text file."""
    try:
        processes = []
        with open(file_path, 'r') as file:
            for line in file:
                parts = line.strip().split(',')
                if len(parts) == 4:
                    pid, arrival_time, burst_time, priority = map(int, parts)
                    processes.append(Process(pid, arrival_time, burst_time,
priority))
                else:
                    raise ValueError(f"Format error in line:
'{line.strip()}'")
        return processes
    except FileNotFoundError:
        raise RuntimeError("File not found.")
    except ValueError as ve:
        raise RuntimeError(f"Error parsing file: {ve}")
    except Exception as e:
        raise RuntimeError(f"Failed to load file: {e}")

def fcfs(processes):
    """First Come, First Served (FCFS) Scheduling Algorithm"""
    processes.sort(key=lambda x: x.arrival_time)
    time = 0
    for process in processes:
        if time < process.arrival_time:
            time = process.arrival_time
        process.start_time = time
        time += process.burst_time
        process.completion_time = time
    return processes

def sjf(processes):
    """Shortest Job First (SJF) Scheduling Algorithm"""
    time = 0
    completed_processes = []
    remaining_processes = processes[:]

    while remaining_processes:
        ready_queue = [p for p in remaining_processes if p.arrival_time <=
time]
        if ready_queue:
            shortest_job = min(ready_queue, key=lambda x: x.burst_time)
            shortest_job_index = remaining_processes.index(shortest_job)
            process = remaining_processes.pop(shortest_job_index)
            process.start_time = time
            time += process.burst_time
            process.completion_time = time
            completed_processes.append(process)
        else:
            time += 1

    return completed_processes
```

```python
def round_robin(processes, time_quantum):
    """Round Robin Scheduling Algorithm"""
    queue = processes[:]
    time = 0
    intervals = {process.pid: [] for process in processes}

    while queue:
        current_process = queue.pop(0)
        if current_process.arrival_time <= time:
            if current_process.remaining_time == current_process.burst_time:
                current_process.start_time = time if
current_process.start_time is None else current_process.start_time

            if current_process.remaining_time <= time_quantum:
                intervals[current_process.pid].append((time,
current_process.remaining_time))
                time += current_process.remaining_time
                current_process.remaining_time = 0
                current_process.completion_time = time
            else:
                intervals[current_process.pid].append((time, time_quantum))
                time += time_quantum
                current_process.remaining_time -= time_quantum
                queue.append(current_process)
        else:
            queue.append(current_process)
            time += 1

    return processes, intervals

def priority_scheduling(processes):
    """Priority Scheduling Algorithm"""
    time = 0
    completed_processes = []
    remaining_processes = processes[:]

    while remaining_processes:
        ready_queue = [p for p in remaining_processes if p.arrival_time <=
time]
        if ready_queue:
            highest_priority = min(ready_queue, key=lambda x: x.priority)
            highest_priority_index =
remaining_processes.index(highest_priority)
            process = remaining_processes.pop(highest_priority_index)
            if process.start_time is None:
                process.start_time = time
            time += process.burst_time
            process.completion_time = time
            completed_processes.append(process)
        else:
            time += 1

    return completed_processes

def multilevel_queue(processes, time_quantum_high=2, time_quantum_low=4):
    """Multilevel Queue Scheduling Algorithm with time quantum for high-
priority queue"""
    high_priority_queue = [p for p in processes if p.priority < 5]
    low_priority_queue = [p for p in processes if p.priority >= 5]

    scheduled_processes = []

    # High-priority queue runs Round Robin with specified time quantum
    queue = high_priority_queue[:]
    time = 0
    while queue:
        current_process = queue.pop(0)
        if current_process.arrival_time <= time:
            if current_process.remaining_time == current_process.burst_time:
                current_process.start_time = time if
current_process.start_time is None else current_process.start_time

            if current_process.remaining_time <= time_quantum_high:
                time += current_process.remaining_time
                current_process.remaining_time = 0
                current_process.completion_time = time
                scheduled_processes.append(current_process)
            else:
                time += time_quantum_high
                current_process.remaining_time -= time_quantum_high
                low_priority_queue.append(current_process)  # Move to low-
priority queue if not finished
        else:
            queue.append(current_process)
            time += 1

    # Low-priority queue runs Round Robin with specified time quantum
    queue = low_priority_queue[:]
    while queue:
        current_process = queue.pop(0)
        if current_process.arrival_time <= time:
            if current_process.remaining_time == current_process.burst_time:
                current_process.start_time = time if
current_process.start_time is None else current_process.start_time

            if current_process.remaining_time <= time_quantum_low:
                time += current_process.remaining_time
                current_process.remaining_time = 0
                current_process.completion_time = time
                scheduled_processes.append(current_process)
            else:
                time += time_quantum_low
                current_process.remaining_time -= time_quantum_low
                queue.append(current_process)
        else:
            queue.append(current_process)
            time += 1

    return scheduled_processes

def calculate_metrics(processes):
    """Calculate and return average turnaround time and average waiting
time"""
    if not processes:
        return 0, 0  # Avoid division by zero
    total_turnaround_time = 0
    total_waiting_time = 0
    for process in processes:
        turnaround_time = process.completion_time - process.arrival_time
        waiting_time = turnaround_time - process.burst_time
        total_turnaround_time += turnaround_time
        total_waiting_time += waiting_time
    avg_turnaround_time = total_turnaround_time / len(processes)
    avg_waiting_time = total_waiting_time / len(processes)
    return avg_turnaround_time, avg_waiting_time
```

- ## Output:


Process Scheduling Simulator

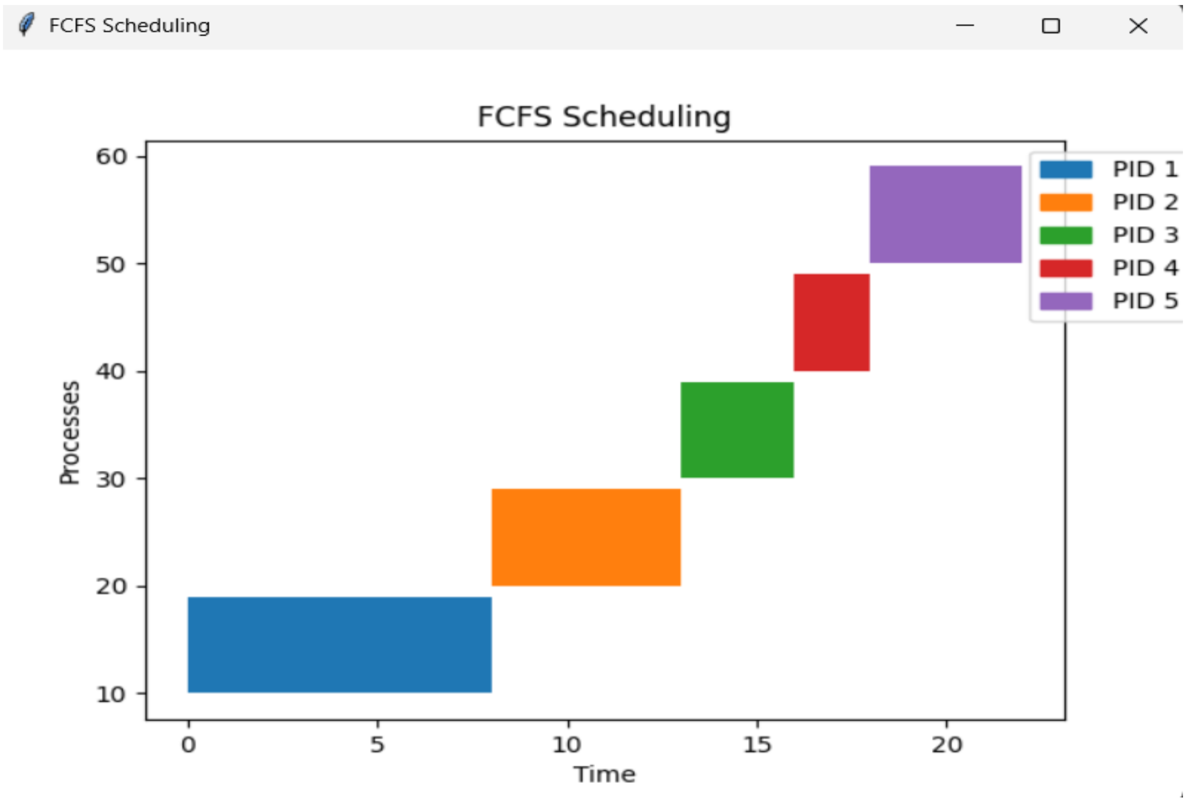
Processes:

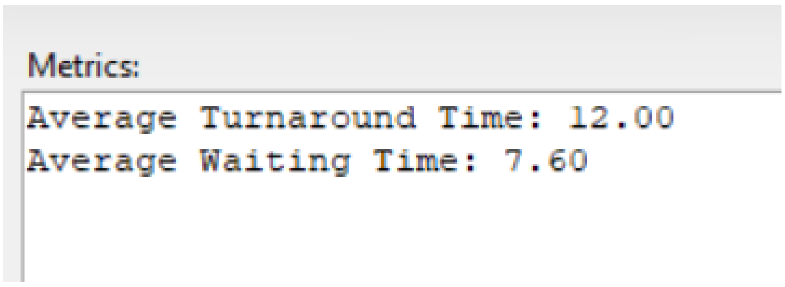
PID	Arrival Time	Burst Time	Priority
1	0	8	3
2	2	5	1
3	4	3	4
4	5	2	2
5	6	4	5

Processes loaded successfully.

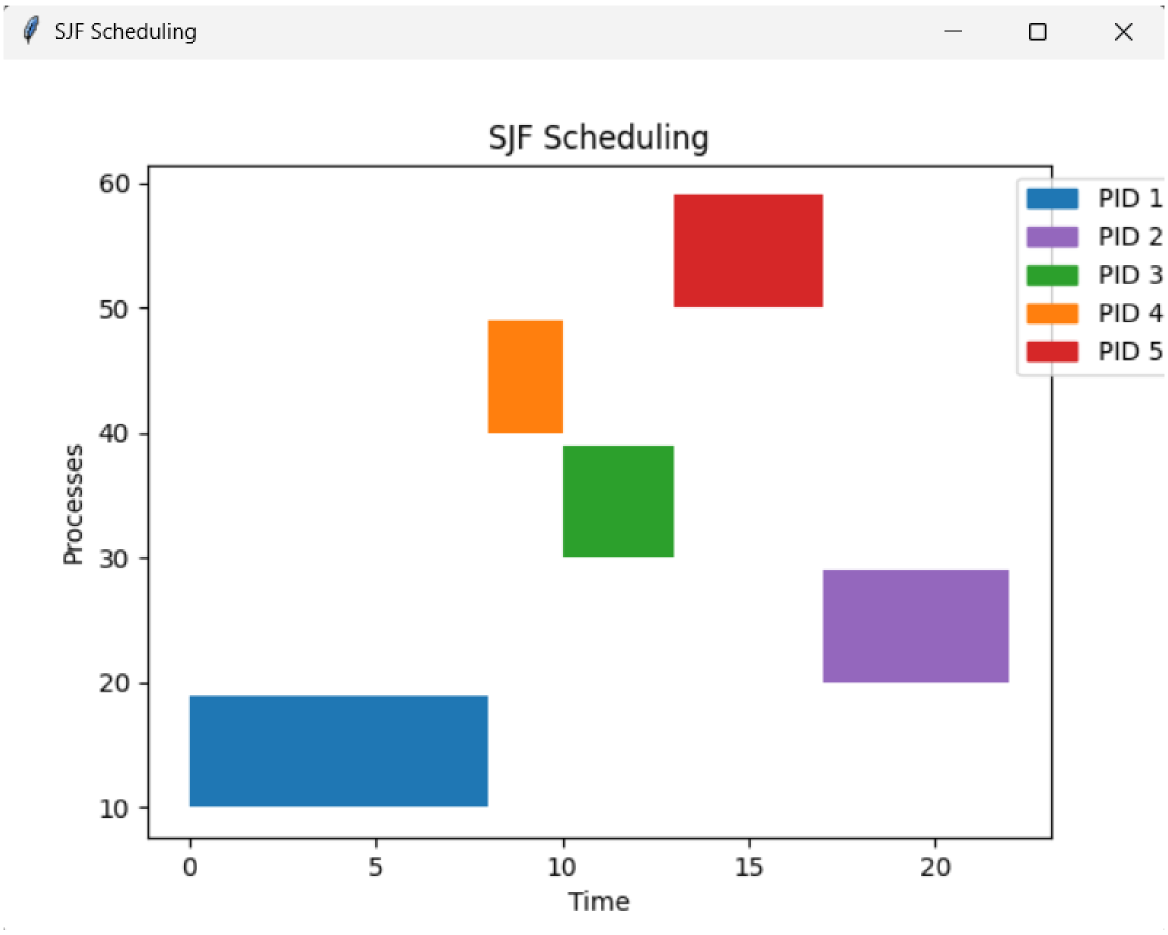
Time Quantum: 2

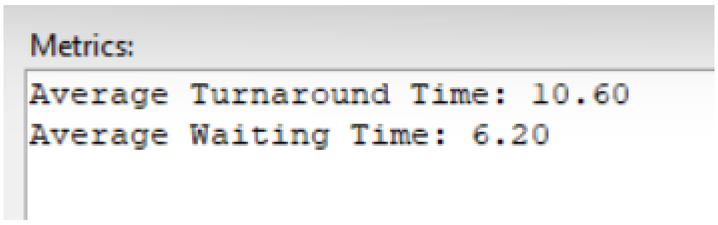
Metrics:



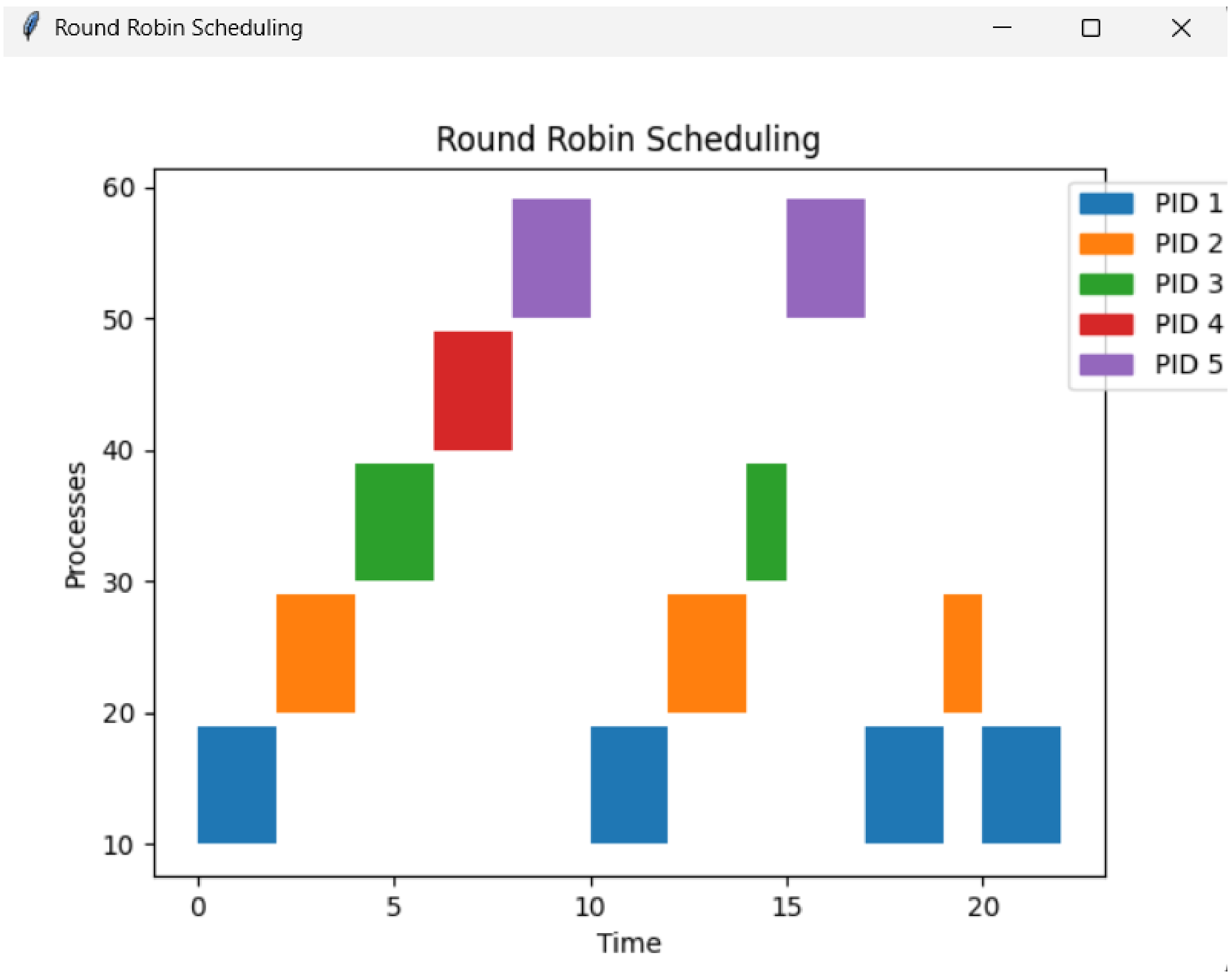


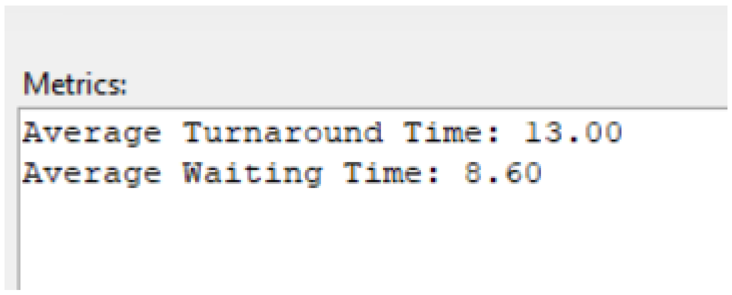
```
Metrics:
Average Turnaround Time: 12.00
Average Waiting Time: 7.60
```



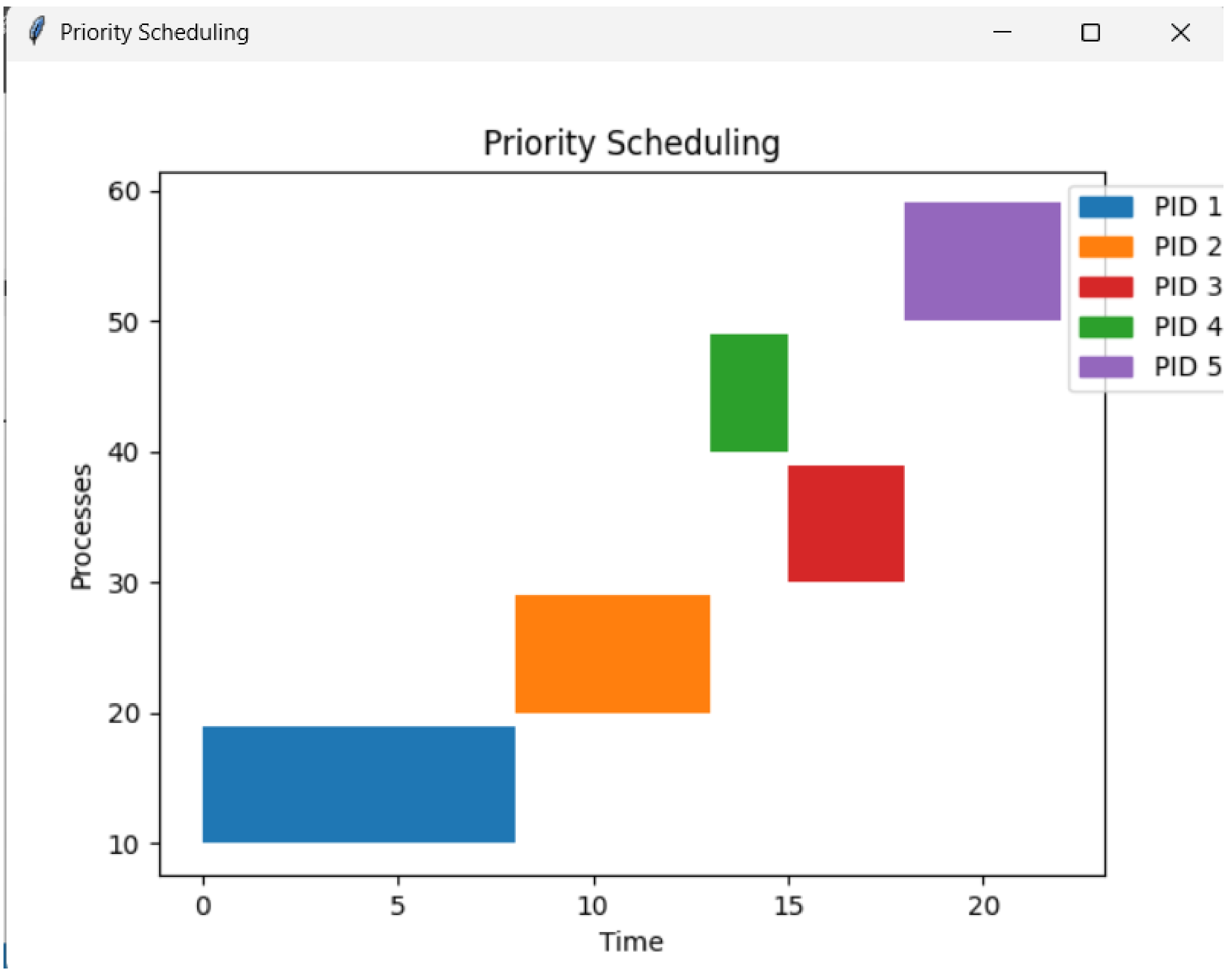


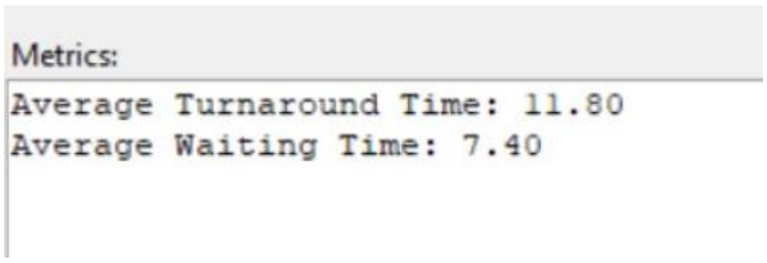
```
Metrics:
Average Turnaround Time: 10.60
Average Waiting Time: 6.20
```



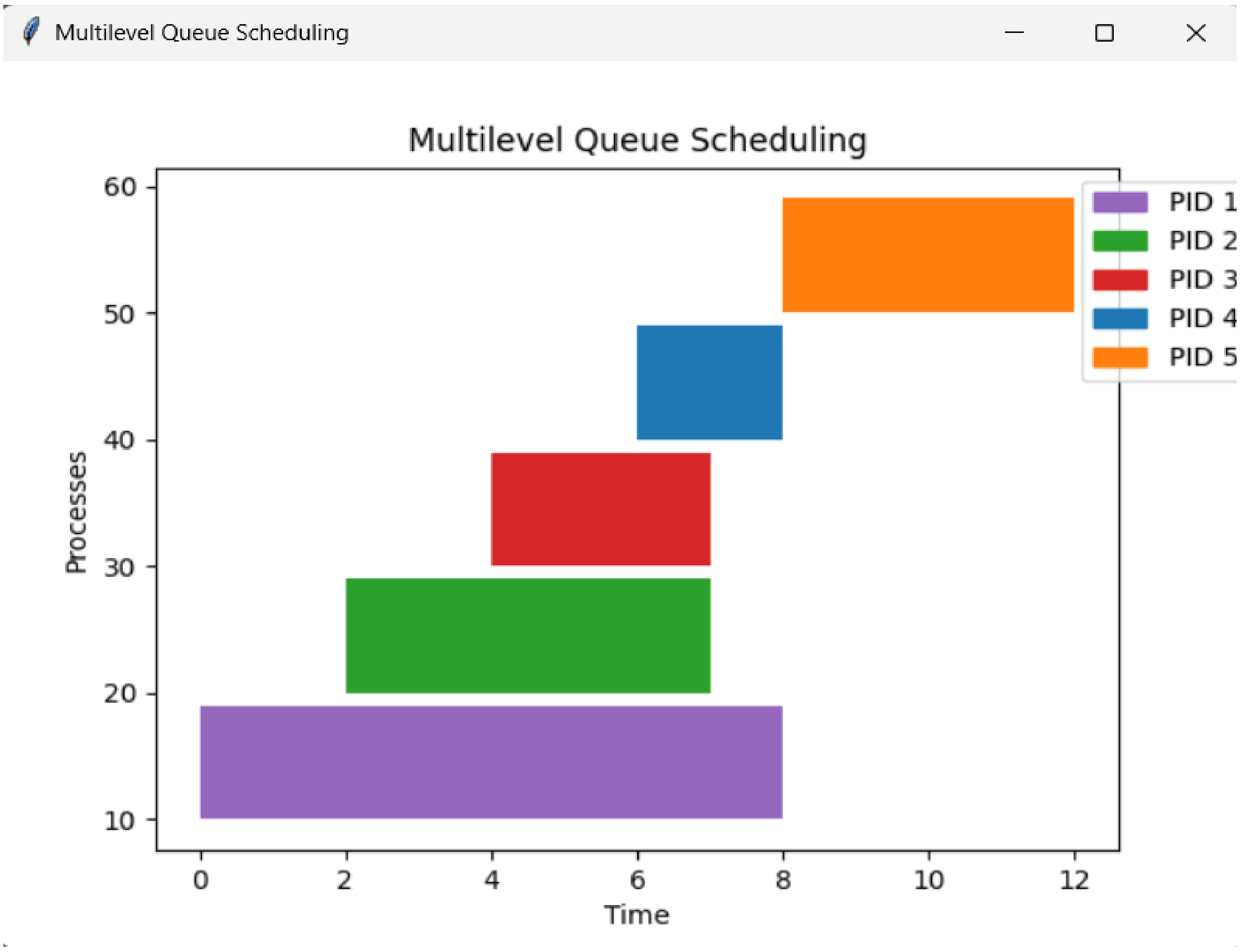

Metrics:

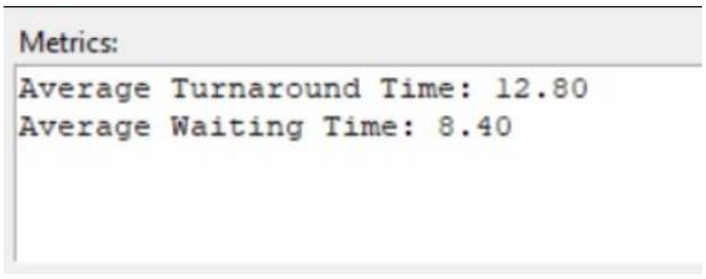
```
Average Turnaround Time: 13.00
Average Waiting Time: 8.60
```




Metrics:

```
Average Turnaround Time: 11.80
Average Waiting Time: 7.40
```





```
Average Turnaround Time: 12.80
Average Waiting Time: 8.40
```

- ## User Manual

Introduction

Welcome to the Process Scheduling Simulator! This simulator is designed to help you understand and compare different process scheduling algorithms used in operating systems. The simulator provides a graphical user interface (GUI) for loading process details, selecting scheduling algorithms, and viewing results in a visual and intuitive manner.

System Requirements

- Python 3.x
- Tkinter library
- Matplotlib library

Installation

1. Python Installation

Ensure that Python 3.x is installed on your system. You can download it from python.org.

2. Tkinter Installation

Tkinter is usually included with Python. If not, you can install it using the package manager for your operating system:

- For Windows: Tkinter is included with Python installation.
- For Linux: sudo apt-get install python3-tk
- For macOS: Tkinter is included with Python installation.

3. Matplotlib Installation

Install Matplotlib using pip:

```
pip install matplotlib
```

Getting Started

Launching the Application

Run the Application

- Navigate to the directory where the simulator script is located.
- Execute the script using Python
- The main window of the Process Scheduling Simulator will appear.

Main Interface

The main interface consists of the following components:

1. **Process Table**

- Displays the loaded processes in a tabular format.
- Columns include Process ID (PID), Arrival Time, Burst Time, and Priority.

2. **Buttons**

- Load Processes: Click this button to load process details from a file.
- Algorithm Buttons: Buttons for each scheduling algorithm (FCFS, SJF, RR, Priority Scheduling, Multilevel Queue Scheduling).
- Display Stack: Click this button to view the stack of loaded processes.

3. **Time Quantum Entry**

- An input field for entering the time quantum for the Round Robin scheduling algorithm.

4. **Status Label**

- Displays the current status and feedback messages.

5. **Metrics Display**

- A text area where performance metrics such as average turnaround time and average waiting time are displayed.

Loading Processes

1. Click **"Load Processes"**

- A file dialog will appear.
- Select a text file containing process details. The file format should be as follows:

```
PID   ArrivalTime   BurstTime    Priority
1 0 5 2
2 1 3 1
3 2 8 4
4 3 6 3
```

2. Verify Loaded Processes

- The loaded processes will appear in the Process Table.
- Click the "Display Stack" button to view the stack of loaded processes.

Running Scheduling Algorithms

1. **Select Algorithm**
- Click the button corresponding to the desired scheduling algorithm.
- If Round Robin is selected, ensure the Time Quantum Entry field is filled with the desired time quantum.

2. **View Gantt Chart**
- A new window will display the Gantt chart, showing the execution timeline of processes.
- Each process is represented with a unique color for clarity.

3. **View Metrics**

- Performance metrics will be displayed in the Metrics Display text area.
- Metrics include average turnaround time and average waiting time.

Scheduling Algorithms

1. **FCFS (First Come First Serve)**

- Schedules processes in the order they arrive.

2. **SJF (Shortest Job First)**

- Schedules processes with the shortest burst time first.

3. **RR (Round Robin)**

- Schedules processes in a cyclic order with a fixed time quantum.
- Ensure to input the time quantum in the Time Quantum Entry field.

4. **Priority Scheduling**

- Schedules processes with the highest priority first.

5. **Multilevel Queue Scheduling**

- Divides processes into multiple queues based on priority levels.
- Each queue has different scheduling policies.

Troubleshooting

1. **No Processes Loaded**

- Ensure the input file is correctly formatted and try loading again.

2. **Gantt Chart Not Displaying**

- Ensure a scheduling algorithm is selected and processes are loaded.

3. **Incorrect Metrics**

- Verify the correctness of the input file and ensure all processes have valid attributes.

Contact and Support

For any issues, suggestions, or feedback, please contact:

Email: m.ahmadsultan123mas@gmail.com

Phone: +92-306-1611301

Future Enhancements

To further enhance the capabilities and usability of the Process Scheduling Simulator, several future enhancements can be considered. These enhancements aim to provide advanced features, improve user experience, and expand the scope of the simulator.

Distributed Scheduling

Objective:

Simulate scheduling algorithms in a distributed system environment.

Details:

- Implement scheduling across multiple CPUs or nodes.
- Address communication delays and resource sharing among distributed components.
- Evaluate the performance and efficiency of distributed scheduling algorithms in comparison to single-system scheduling.

Integration with Real Systems

Objective:

Extend the simulator to interface with actual operating systems or hardware.

Details:

- Allow the simulator to run on real hardware to observe actual performance metrics.
- Analyze the impact of different scheduling algorithms on real-time processes and overall system responsiveness.
- Provide insights into the practical applicability and effectiveness of the implemented algorithms.

User-defined Algorithms

Objective:

Provide a framework for users to define and test their custom scheduling algorithms.

Details:

- Allow users to input custom logic and parameters for scheduling.
- Facilitate experimentation with new or experimental scheduling policies.
- Enable users to compare custom algorithms with standard algorithms in terms of performance metrics.

Advanced Visualization Techniques

Objective:

Enhance the visualization module to support more interactive and detailed graphical representations.

Details:

- Support interactive Gantt charts with zoom, filter, and manipulation capabilities.
- Provide additional visualizations such as heatmaps and timelines to represent process states and transitions.
- Improve user experience by offering deeper analysis and insights through advanced visualization tools.

Enhanced User Interface

Objective:

Develop a more intuitive and feature-rich GUI.

Details:

- Implement drag-and-drop functionality for process management.
- Provide customizable themes and layouts for the GUI.
- Enhance real-time interaction with the simulator, allowing users to modify parameters and observe immediate effects.

Cloud-based Simulation

Objective:

Enable cloud-based simulation for scalable and remote access to the simulator.

Details:

- Deploy the simulator on cloud platforms to handle large-scale simulations.
- Facilitate remote access, collaboration, and sharing of simulation results.
- Leverage cloud computing resources for enhanced performance and scalability.

Extended Performance Metrics

Objective:

Incorporate additional performance metrics for comprehensive evaluation.

Details:

- Include metrics such as CPU utilization, context switch count, and throughput.
- Provide detailed reports and analysis for each scheduling algorithm.
- Enable users to generate custom performance reports based on selected metrics.

▪ Applications

Educational Tool

Purpose: Helps students understand and compare different scheduling algorithms.

Description: The simulator provides a visual and interactive way for students to learn about scheduling algorithms by running simulations and observing how each algorithm handles different scenarios. Students can experiment with varying process arrival times, burst times, and priorities to see the impact on scheduling outcomes.

Performance Analysis

Purpose: Assists in analyzing the performance of various scheduling policies.

Description: Users can simulate different scheduling algorithms and compare metrics such as average turnaround time and average waiting time. This allows system administrators and developers to evaluate which scheduling algorithm performs best under specific workload conditions and make informed decisions about system optimizations.

Algorithm Development

Purpose: Provides a platform for developing and testing new scheduling algorithms.

Description: The simulator supports user-defined algorithms, enabling researchers and developers to implement and test new scheduling policies. By interfacing with the simulator, users can validate the efficiency and fairness of their algorithms before deployment in real-world systems.

Resource Allocation Studies

Purpose: Facilitates studying resource allocation strategies in computing systems.

Description: Researchers and system designers can use the simulator to investigate how different scheduling algorithms distribute CPU resources among processes. This includes studying the impact of preemptive vs. non-preemptive scheduling, varying quantum sizes in round-robin scheduling, and optimizing resource utilization in multilevel queue systems.

Real-time System Simulation

Purpose: Simulates scheduling algorithms in real-time systems for analysis and testing.

Description: The simulator can be adapted to model real-time operating systems, allowing users to experiment with scheduling policies that prioritize tasks based on deadlines and time constraints. This capability supports industries such as telecommunications, where responsiveness and predictability are critical.

▪ Potential Questions

How does the choice of scheduling algorithm impact system performance?

How can scheduling algorithms be adapted for real-time and distributed systems?

What are the trade-offs between different scheduling algorithms?

▪ Conclusion

In conclusion, the Process Scheduling Simulator developed in this project provides a robust platform for studying and comparing various scheduling algorithms essential to efficient CPU resource management. By offering comprehensive visualization through Gantt charts and performance metrics calculation, the simulator supports both educational exploration and practical performance analysis. Its user-friendly interface and extensible architecture lay a strong foundation for future enhancements and algorithmic innovations in scheduling, addressing critical challenges in operating system design and optimization.

▪ Reference Citations:

- https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/
- https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm
- https://www.geeksforgeeks.org/python-tkinter-tutorial/
- https://unacademy.com/content/cbse-class-11/difference-between/turnaround-time-and-waiting-time-in-cpu-scheduling/
- https://www.w3schools.com/python/
- https://www.javatpoint.com/os-scheduling-algorithms

THE END