



Project title: Intelligent CPU Scheduler Simulator

Submitted to: Dr. Anudeep Goraya

Subject Code: CSE316 Group

Details:

1.Venkata Manikanta Padarthi/12322263

2.Narendra Yalla/ 12324803

2.Prema Sai Kimmi/12306000

Section:K23GD

Topic: Intelligent CPU Scheduler Simulator

1.Overview:

The Intelligent CPU Scheduler Simulator is a software system that simulates and visualizes CPU scheduling algorithms interactively so that users can comprehend and examine their

performance. The major objective is real-time visualization of scheduling activities through Gantt charts along with computation of key performance metrics such as Average Waiting Time (AWT) and Turnaround Time (TAT).

2. Module-Wise Breakdown

Module 1: User Interface

Purpose: Provide an interactive User Interface for users to input process details and visualize scheduling algorithms.

Key Functions:

- Processes user input (algorithm choice, process details).
- Supports real-time Gantt chart visualization.
- Displays performance indicators (Turnaround Time, Waiting Time).
- The control panel enables the users to begin and stop the simulation, providing them with flexibility in examining various scheduling algorithms.

Module 2: Algorithm Processing

Purpose: Implement scheduling logic, process user input, and return computed results to the frontend.

Key Functions:

- The API server (Flask) handles frontend requests, runs scheduling logic, and provides calculated results in a seamless manner.
- The Scheduling Algorithms Module has FCFS, SJF, Round Robin, and Priority Scheduling implemented to ensure correct order of execution and time calculation.
- The backend keeps updating process queues, calculates Gantt chart data, and updates waiting & turnaround times in real time.
- The Data Processing & JSON Formatter transforms results into a wellformatted JSON structure, allowing seamless real-time visualization on the frontend.

Module 3: Real-Time Visualization

Purpose: Show dynamic Gantt chart updates as processes execute.

Key Functions:

- JavaScript updates the Gantt chart in real time as processes are being executed, reflecting scheduling activity in real-time.

- Users can see process execution in real-time, control speed, and dynamically analyze scheduling behavior.

Module 4: User Management & Authentication Module

Purpose: Allows users to create accounts, log in, and track past simulations.

Key Functions:

- User authentication using Flask authentication.
- Store and retrieve previously executed scheduling simulations for analysis.
- Role-based access for managing simulations.

3. Functionalities

User Input & Process Management:

- Users can enter process details such as Process ID, Arrival Time, Burst Time, Priority and Time Quantum.

Algorithm Selection & Execution:

- Users can select one of the following CPU scheduling algorithms:
 1. First Come First Serve
 2. Shortest Job First (SJF) – Preemptive & Non-Preemptive
 3. Round Robin
 4. Priority Scheduling – Preemptive & Non-Preemptive

Real-Time Gantt Chart Visualization:

- As it executes, a Gantt chart visually represents process execution.

Performance Measures Calculation & Representation:

- Computes and shows:
 1. Completion Time (CT) of every process.
 2. Turnaround Time ($TAT = CT - AT$).
 3. Waiting Time ($WT = TAT - BT$).
 4. Average Waiting Time (AWT) & Average Turnaround Time (ATAT).

Backend Processing & API Communication:

- Flask backend processes the scheduling logic.
- Manages preemptive and non-preemptive scheduling logic efficiently.

4. Technology Utilized

Programming Language:Python **Libraries**

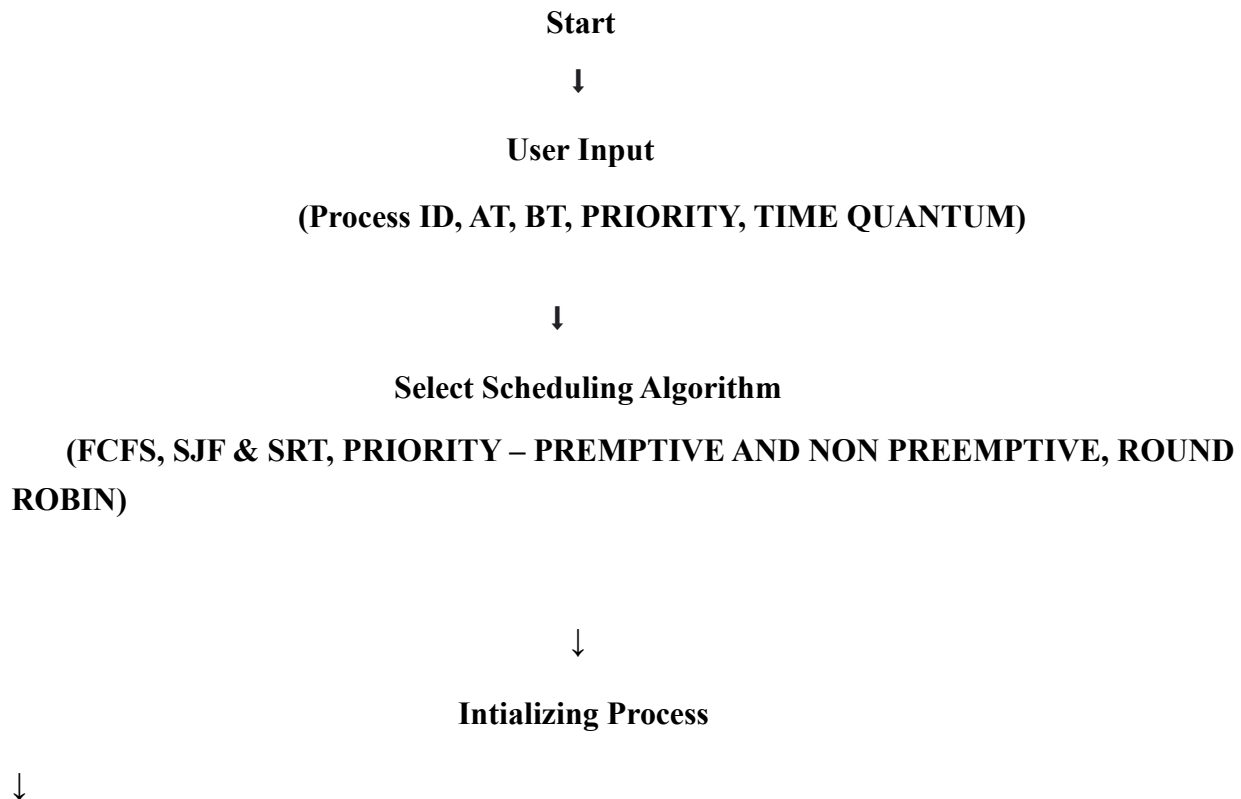
and Tools:

- **Matplotlib:** Used for Visual Representation of Gantt Chart
- **HTML, CSS, JavaScript:** For designing the user interface.
- **Pandas & NumPy:** For handling and processing scheduling data.
- **Flask API:** To implement scheduling algorithms and process API requests.

Miscellaneous Tools:

- **GitHub:** For version and collaboration control.
- **VS Code / PyCharm:** For development.
- **Postman (optional):** For API testing.

Flow chart:



Executing Scheduling Algorithms
(Generating CT, TAT, WT & AVERAGE TAT, AVERAGE WT)



Updating Gantt charts



End

6. Revision Tracking on GitHub

- **Repository Name:** Intelligent of CPU Scheduler Simulator
- **GitHub Link:** <https://github.com/manikanta12322263/Intelligent-of-CPU-Scheduler-Simulator.git>

7. Conclusion and Future Scope Conclusion:

Intelligent CPU Scheduler Simulator is an interactive, real-time visualizer for several CPU scheduling algorithms such as FCFS, SJF, Round Robin, and Priority Scheduling. Since it enables the user to insert processes, look at Gantt charts, and analyze performance factors (waiting time, turnaround time, CPU usage), the simulator is a highly effective learning as well as analyzing tool.

Future Scope:

Support for Other Scheduling Algorithms: Implement Multilevel Queue Scheduling, MultiLevel Feedback Queue (MLFQ), and Earliest Deadline First (EDF) for further advanced analysis.

AI-Based Smart Scheduler: Implement Machine Learning for recommending the best scheduling algorithm based on process patterns and workload conditions.

Multi-Core & Parallel Processing Simulation: Extend the simulator for multi-core CPU scheduling, displaying how processes execute concurrently.

Cloud-Based Deployment & User Profiles: Host the simulator on the cloud, enabling users to save and load previous simulations remotely.

Gamification & Learning Features: Integrate interactive tutorials, quizzes, and challenges to enhance learning scheduling principles.

Enhanced Visualization & Animation: Enrich Gantt chart visualization with more fluid animations, color-coded priority levels, and zoom-in analysis capabilities.

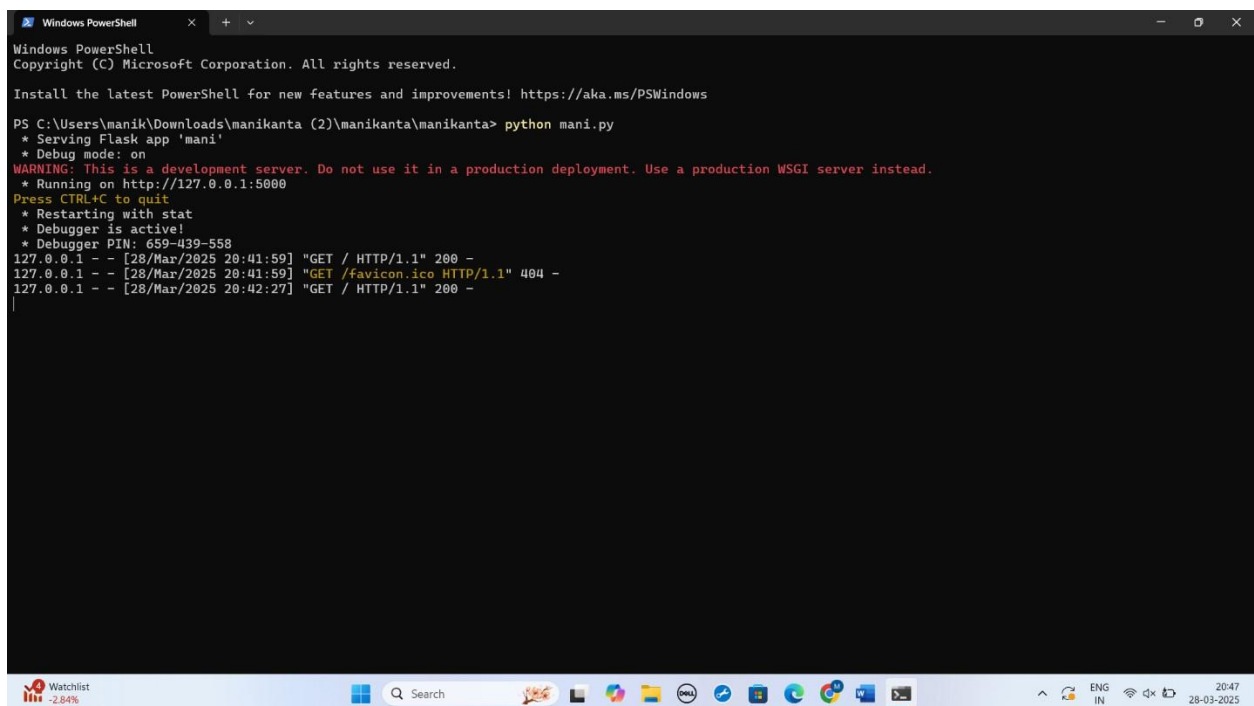
Mobile Application Development: Create a mobile-compatible version or standalone app for students and professionals to perform simulations on mobile devices.

8. References

- Flask API Documentation
- Python Library for Data Handling
- GitHub for Version Control
- Operating System Concepts

Appendix

A. AI-Generated Project Elaboration/Breakdown Report 1.Opening the Project by using Command Prompt:

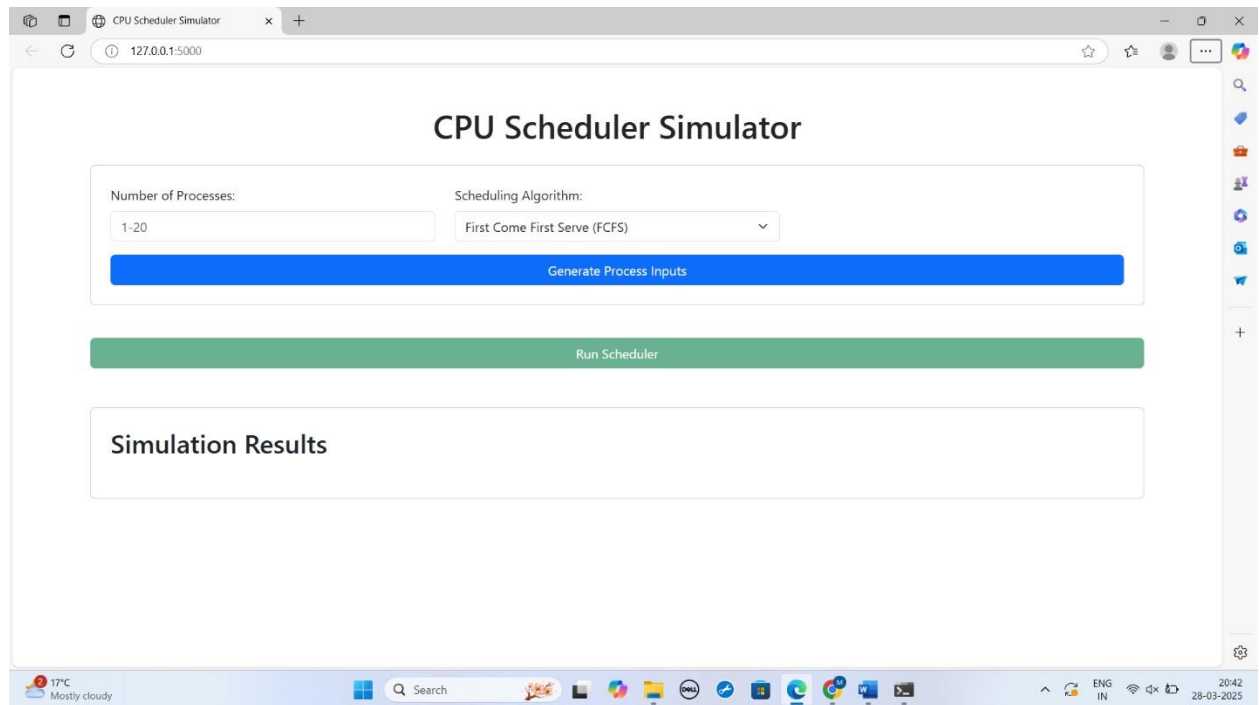


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

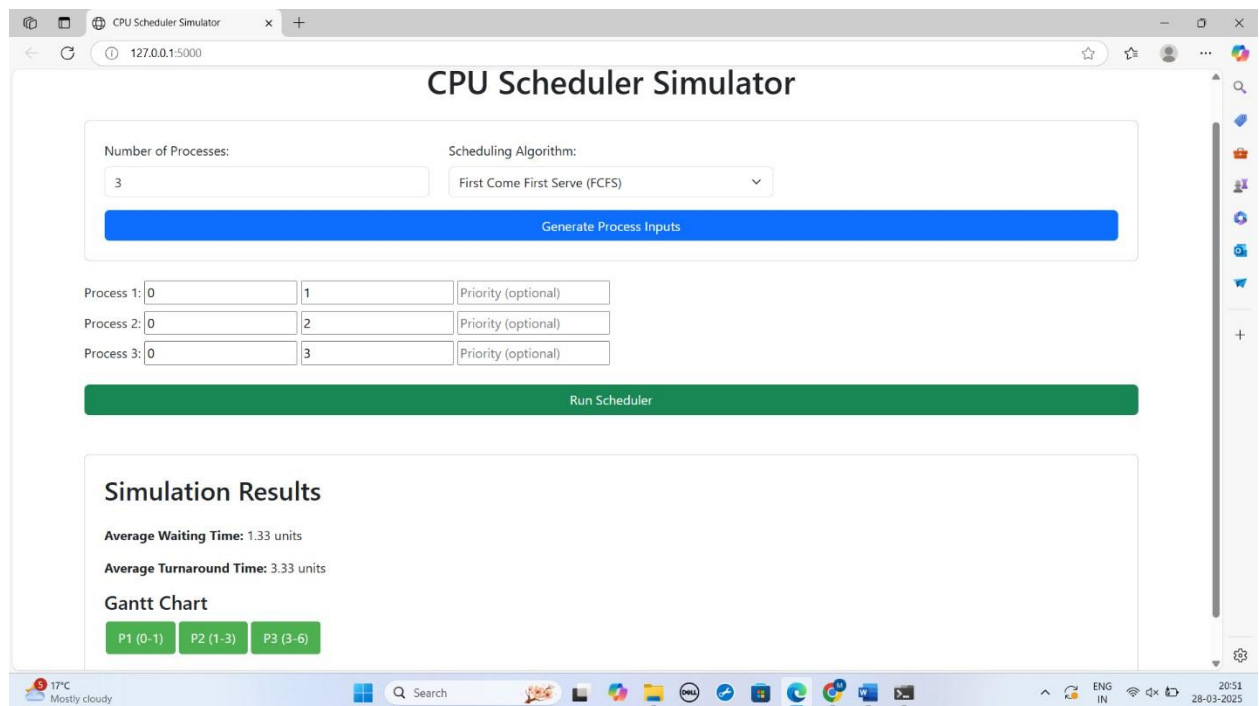
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\manik\Downloads\manikanta (2)\manikanta\manikanta> python mani.py
* Serving Flask app 'mani'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 659-439-558
127.0.0.1 - - [28/Mar/2025 20:41:59] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [28/Mar/2025 20:41:59] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [28/Mar/2025 20:42:27] "GET / HTTP/1.1" 200 -
```

2.Before Giving Input:



3.After Generating Output:



B. Problem Statement:

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:

```
from flask import Flask, request, jsonify, render_template from
flask_cors import CORS
```

```
app = Flask(__name__)
```

```
CORS(app) # Enable CORS for API requests
```

```
class Process:    def __init__(self, pid, arrival,
burst, priority=0):
```

```
    self.pid = pid
```

```
    self.arrival = arrival
```

```
    self.burst = burst
```

```
    self.priority = priority
```

```
    self.remaining = burst
```

```
    self.start_time = None
```

```
    self.completion_time = None
```

```
    self.waiting_time = 0
```

```
    self.turnaround_time = 0
```

```
def fcfs_scheduling(processes):
```

```
    processes.sort(key=lambda p: p.arrival)    time = 0    gantt_chart = []    for
process in processes:        if time < process.arrival:            time = process.arrival
process.start_time = time        time += process.burst        process.completion_time =
time        process.turnaround_time = process.completion_time - process.arrival
```



```

process.waiting_time = process.turnaround_time - process.burst

gantt_chart.append((process.pid, process.start_time, process.completion_time))

return gantt_chart, processes

```

```

def sjf_preemptive(processes):
    time = 0    gantt_chart = []

    completed = []    remaining =
processes.copy()    while
remaining:

    available = [p for p in remaining if p.arrival <= time]

    if not available:

        time += 1

        continue

    current = min(available, key=lambda p: p.remaining)

    current.remaining -= 1

    gantt_chart.append((current.pid, time, time + 1))    if

    current.remaining == 0:

        current.completion_time = time + 1        current.turnaround_time
= current.completion_time - current.arrival        current.waiting_time =
current.turnaround_time - current.burst        completed.append(current)

    remaining.remove(current)    time += 1    return gantt_chart, completed

def priority_preemptive(processes):

```

```

    time = 0    gantt_chart = []

    completed = []    remaining =
    processes.copy()    while
    remaining:

        available = [p for p in remaining if p.arrival <= time]

    if not available:

        time += 1

    continue    current =
    min(available,
    key=lambda p:
    p.priority)

    current.remaining -= 1

    gantt_chart.append((cur
    rent.pid, time, time + 1))

    if current.remaining ==
    0:

        current.completion_time = time + 1        current.turnaround_time
        = current.completion_time - current.arrival        current.waiting_time =
        current.turnaround_time - current.burst        completed.append(current)

    remaining.remove(current)    time += 1    return gantt_chart, completed

def round_robin_scheduling(processes, quantum):

```

```

    queue = processes.copy()
time = 0    gantt_chart = []
completed = []    while
queue:
    process = queue.pop(0)    if time <
process.arrival:    time = process.arrival
start = time    exec_time =
min(process.remaining, quantum)    time +=
exec_time    process.remaining -= exec_time
gantt_chart.append((process.pid, start, time))

    if process.remaining > 0:
queue.append(process)
    else:
        process.completion_time = time    process.turnaround_time =
process.completion_time - process.arrival    process.waiting_time =
process.turnaround_time - process.burst    completed.append(process)
return gantt_chart, completed

@app.route('/') def
index():
    return render_template('manikanta.html')

```

```

@app.route('/schedule', methods=['POST']) def
schedule():

    try:

        data = request.json    if not data or 'algorithm' not in data
or 'processes' not in data:    return jsonify({"error": "Missing
required fields"}), 400    algorithm = data['algorithm']

    quantum = data.get('quantum', 2)    processes_data =
data['processes']

    processes = [Process(p['pid'], p['arrival'], p['burst'], p.get('priority', 0))
for p in processes_data]

    if algorithm == "FCFS":

        gantt_chart, result_processes = fcfs_scheduling(processes)

    elif algorithm == "SJF-Preemptive":

        gantt_chart, result_processes = sjf_preemptive(processes)

    elif algorithm == "Priority-Preemptive":

        gantt_chart, result_processes = priority_preemptive(processes)

    elif algorithm == "RR":

        if quantum < 1:

            return jsonify({"error": "Quantum must be positive"}), 400

        gantt_chart, result_processes = round_robin_scheduling(processes, quantum)

    else:

```

```

    return jsonify({"error": "Invalid algorithm"}), 400

    result = {

        "gantt_chart": gantt_chart,

        "avg_waiting_time": sum(p.waiting_time for p in result_processes) /
len(result_processes),

        "avg_turnaround_time": sum(p.turnaround_time for p in result_processes) /
len(result_processes),

        "processes": [{"pid": p.pid, "waiting_time": p.waiting_time,

                        "turnaround_time": p.turnaround_time} for p in result_processes]

    }

    return jsonify(result)

except Exception as e:

    return jsonify({"error": str(e)}), 500

if __name__ == "__main__":

    app.run(debug=True, host='127.0.0.1', port=5000)

```