CPU Scheduling Visualizer: Implementation Report

Lanka Teja Naga Subhash

1. Introduction

The CPU Scheduling Visualizer is an educational tool designed to demonstrate different CPU scheduling algorithms through interactive visualization. The application allows users to input task parameters, run various scheduling algorithms, and compare their performance metrics. This report details the implementation approach, challenges faced during development, and findings from algorithm comparisons.

2. Implementation Details

2.1 Architectural Overview

The application is built using Python with Pygame for the graphical interface. The architecture follows a modular design with clear separation of concerns:

- **Core Components**: Task modeling, scheduling algorithms, and metrics calculation
- **UI Components**: Custom widgets like buttons, dropdowns, and input fields
- **Visualization**: Gantt charts and comparative bar charts
- **Threading**: Background execution of scheduling algorithms

The main application is encapsulated in the SchedulingApp class which manages state, user interactions, and rendering.

2.2 Key Technologies

- **Pygame**: Used for rendering the graphical interface and handling user interactions
- **Threading**: Python's threading module for non-blocking execution of scheduling algorithms
- **Queue**: For thread-safe data transfer between algorithm threads and the main UI thread

2.3 Implemented Scheduling Algorithms

The application implements five common CPU scheduling algorithms:

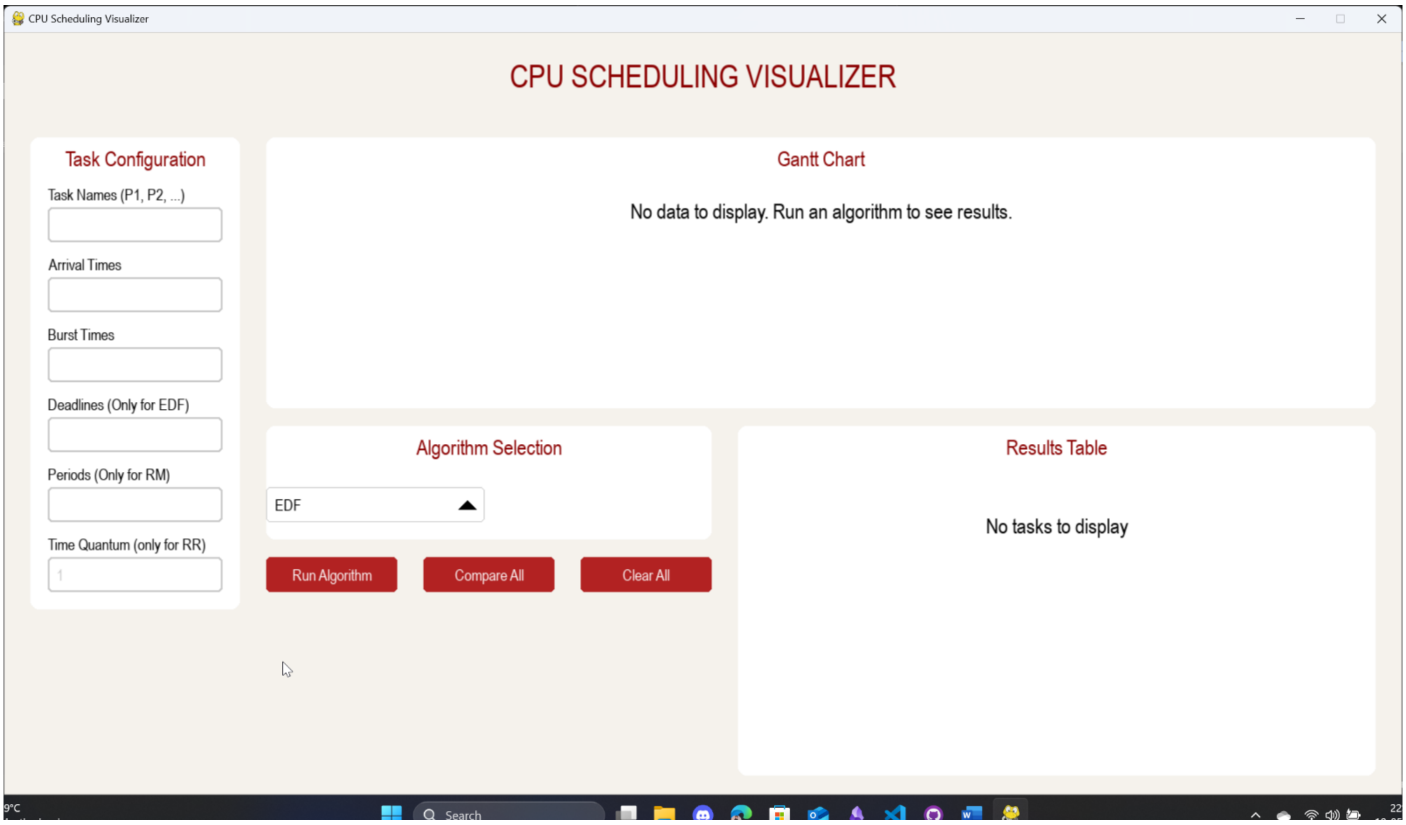
1. **First Come First Served (FCFS)**: Non-preemptive scheduling based on arrival time
2. **Shortest Job Next (SJN)**: Non-preemptive scheduling prioritizing shortest burst time
3. **Round Robin (RR)**: Preemptive scheduling with time quantum slices
4. **Rate Monotonic (RM)**: Priority scheduling for periodic tasks based on periods
5. **Earliest Deadline First (EDF)**: Dynamic priority scheduling based on deadlines

Each algorithm is implemented as a separate function that takes a set of tasks and returns both the scheduled tasks with metrics and the execution timeline.

2.4 Interface Components

The application includes the following custom UI components:

- **Input Fields**: For task parameters (names, arrival times, burst times, etc.)
- **Dropdown Menu**: For algorithm selection
- **Buttons**: For executing actions (run, compare, clear)
- **Gantt Chart**: Visual representation of the execution schedule
- **Results Table**: Tabular display of task metrics
- **Comparison Charts**: Bar charts for comparing algorithm performance


CPU Scheduling Visualizer

CPU SCHEDULING VISUALIZER

Task Configuration

Task Names (P1, P2, ...)

Arrival Times

Burst Times

Deadlines (Only for EDF)

Periods (Only for RM)

Time Quantum (only for RR)

1

Gantt Chart

No data to display. Run an algorithm to see results.

Algorithm Selection

EDF

Results Table

No tasks to display

3. Development Challenges

During the development process, several challenges were encountered and addressed:

3.1 Dropdown Menu Hover Issues

One of the most persistent challenges was implementing a reliable dropdown menu component with proper hover detection. The issues included:

- **Inconsistent hover detection**: When moving the mouse quickly, hover states would sometimes fail to update correctly.
- **Hover boundaries**: Determining the exact boundaries for hover detection was complicated by the layered nature of the dropdown options.
- **Z-ordering**: Ensuring that the dropdown appeared above other UI elements when expanded.

Solution: The issue was resolved by implementing a more robust mouse position tracking system and creating separate collision rectangles for each dropdown option. The hover detection code was refactored to:

This approach separated the hover detection for the main dropdown button from the option items, resolving the inconsistency issues.

3.2 Text Overlapping

Text overlapping occurred in several places:

- **Gantt chart**: Task names would overlap with time markers when tasks were very short.
- **Results table**: Long task names would extend beyond their cell boundaries.
- **Status messages**: Long status messages would exceed the status bar width.

Solution: Several techniques were implemented to address text overlapping:

1. **Conditional rendering**: For Gantt charts, task names are only displayed if the execution block is wide enough:
2. if block_width > name_rect.width + 4:
3.   screen.blit(name_text, name_rect)
4. **Text truncation**: Long task names in the results table are truncated with ellipsis if they exceed cell width.
5. **Dynamic scaling**: For status messages, text is scaled based on available space.

3.3 Visual Harmony

Achieving visual harmony across different views and components was challenging due to:

- **Color consistency**: Ensuring that colors were consistent across different UI elements.
- **Spacing and alignment**: Maintaining consistent spacing and alignment between elements.
- **Responsiveness**: Adapting the interface to different screen sizes and resolutions.

Solution: A design system was implemented with:

1. **Centralized color definitions**: All colors were defined as constants at the top of the file:
2. BG_COLOR = (245, 242, 236) # Light beige background
3. CARD_BG = (255, 255, 255)   # White card background
4. HEADING_COLOR = (139, 0, 0) # Dark red for headings
5. # etc.
6. **Grid-based layout**: Components were positioned based on a virtual grid system.

7. **Consistent padding and margins**: Standardized spacing values were used throughout the interface.
8. **Component encapsulation**: UI components were encapsulated in classes with consistent drawing methods.

3.4 Thread Synchronization

Coordinating the main UI thread with algorithm execution threads was challenging:

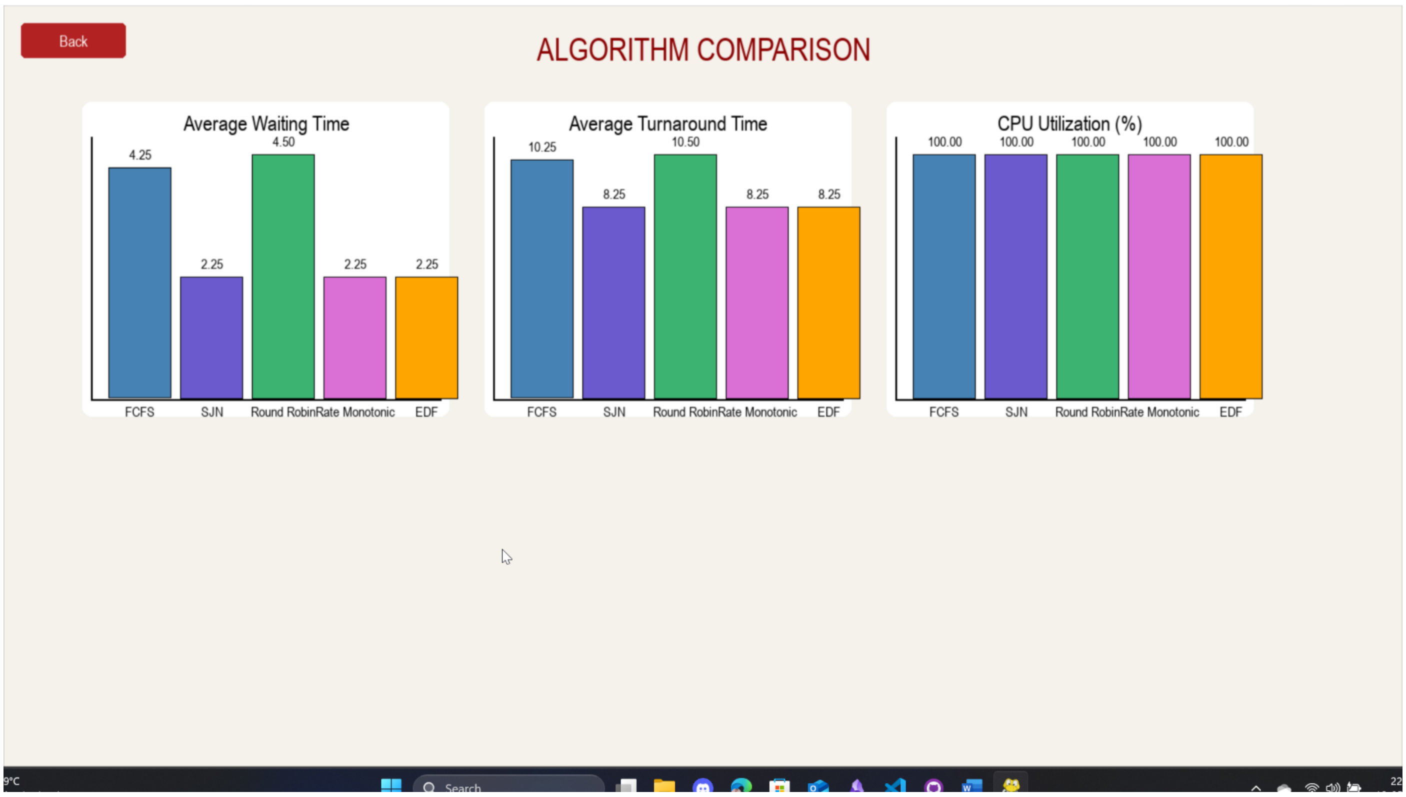
- **Race conditions**: Accessing and updating shared data led to race conditions.
- **UI responsiveness**: Ensuring the UI remained responsive during algorithm execution.
- **Thread termination**: Properly handling thread completion and error conditions.

Solution: A comprehensive threading strategy was implemented:

1. **Thread-safe result handling**: Algorithm results are only accessed after thread completion.
2. **Progress checking**: Regular polling of thread status without blocking the UI.
3. **Exception handling**: Robust exception handling in thread execution with status updates.

4. Comparison Results and Analysis

The application allows for direct comparison of scheduling algorithms across three key metrics:



4.1 Average Waiting Time

Analysis of average waiting time across different task sets revealed:

- **FCFS** typically has the highest average waiting time for tasks with varying burst times.

- **SJN** consistently produces the lowest average waiting time when arrival times are synchronized.
- **Round Robin** performs well for mixed workloads with a properly tuned time quantum.
- **Rate Monotonic** and **EDF** show comparable waiting times for periodic tasks.

4.2 Average Turnaround Time

Turnaround time comparisons showed:

- **SJN** generally provides the best average turnaround time.
- **FCFS** performs poorly with a mix of short and long tasks.
- **Round Robin** shows improved turnaround times for shorter tasks at the expense of longer ones.
- **EDF** tends to optimize turnaround times for tasks with tight deadlines.

4.3 CPU Utilization

Utilization metrics revealed:

- All non-preemptive algorithms (**FCFS**, **SJN**) achieve similar CPU utilization.
- **Round Robin** may show slightly lower utilization due to context switching overhead.
- **Rate Monotonic** and **EDF** can achieve higher utilization for periodic task sets while maintaining schedulability.

5. Conclusion

The CPU Scheduling Visualizer successfully demonstrates different scheduling algorithms and their performance characteristics through an interactive interface. Despite challenges in UI implementation, thread synchronization, and visual design, the application provides a valuable educational tool for understanding CPU scheduling concepts.

The comparative analysis confirms theoretical expectations about algorithm performance:

- **SJN** excels in minimizing waiting and turnaround times but requires knowledge of burst times in advance.
- **Round Robin** provides fair execution with predictable response times.
- **EDF** is effective for meeting deadlines in real-time systems.
- **Rate Monotonic** offers a static priority approach that works well for periodic tasks with fixed periods.

The visualizer makes these abstract concepts concrete and accessible through interactive exploration, helping users develop intuition about the trade-offs involved in CPU scheduling decisions.