Schedulix: An Educational CPU Scheduling Visualizer Based on the Illusion of Parallelism

1st Abhay Rana
3rd year B.Tech CS and AI
Newton School of Technology
Haryana, India
abhay.r23csai@nst.rishihood.edu.in

2nd Saloni Sharma
3rd year B.Tech CS and AI
Newton School of Technology
Haryana, India
saloni.s23csai@nst.rishihood.edu.in

3rd Belal Raza
3rd year B.Tech CS and AI
Newton School of Technology
Haryana, India
belal.r23csai@nst.rishihood.edu.in

***Abstract*—This paper presents the design, implementation, and educational value of Schedulix, an interactive web-based CPU scheduling visualizer that demonstrates the illusion of parallelism in operating systems. The system allows users to observe how a single processor creates the appearance of concurrent execution through rapid context switching and time slicing. The visualizer supports four scheduling algorithms: First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Multi-Level Feedback Queue (MLFQ), and provides real-time metrics including average waiting time, turnaround time, CPU utilization, and throughput. The architecture employs the Strategy design pattern to allow dynamic selection of scheduling policies without modifying core scheduler logic. Key operating system concepts such as preemption, context switching overhead, starvation, and the convoy effect are visualized through an intuitive interface built with React and Framer Motion. The project serves as an educational tool for students and instructors in operating systems courses and demonstrates the trade-offs between different scheduling objectives in batch, interactive, and real-time computing environments.**

***Index Terms*—CPU scheduling, context switching, FCFS, SJF, Round Robin, MLFQ, illusion of parallelism, operating systems education, strategy pattern, process states, preemption, turnaround time, waiting time.**

I. Introduction

A. Motivation

Understanding how an operating system schedules processes on a single CPU is fundamental to computer science and systems programming. Although a single processor can execute only one instruction at a time, users perceive multiple applications running simultaneously. This "illusion of parallelism" is achieved through rapid context switching, time slicing, and scheduling algorithms that determine which process runs next and for how long [1]. Teaching these concepts purely through theory or static diagrams often leaves gaps in intuition. An interactive visualizer that allows students to add processes, inject I/O interrupts, and observe scheduling decisions in real time can bridge this gap.

B. Objectives

The primary objectives of this project are: (1) to build an accurate, step-by-step simulation of CPU scheduling that reflects standard OS behavior; (2) to support multiple scheduling algorithms (FCFS, SJF, Round Robin, MLFQ) with a pluggable architecture; (3) to visualize process states (New, Ready, Running, Waiting, Terminated), context switches, and a Gantt-style execution timeline; (4) to compute and display standard performance metrics (waiting time, turnaround time, CPU utilization, throughput); and (5) to serve as an educational resource that illustrates preemption, starvation, and the convoy effect.

C. Scope

The project focuses on single-CPU scheduling. Multi-core or multiprocessor scheduling is out of scope. The simulation uses discrete time steps and configurable context-switch overhead. The implementation is a client-side web application requiring no backend server.

D. Report Organization

The remainder of this paper is organized as follows. Section II summarizes relevant background and related work. Section III describes the system design and architecture. Section IV details the implementation. Section V discusses results and educational outcomes. Section VI concludes and suggests future work.

II. Background and Related Work

A. Process States and Scheduling

A process moves through well-defined states: New, Ready, Running, Waiting, and Terminated [2]. The scheduler selects from the ready queue which process to run next. When a process blocks for I/O, it moves to a waiting queue until the I/O completes. Scheduling algorithms differ in how they select the next process and whether they allow preemption—interrupting a running process to run another [3].

B. Scheduling Algorithms

FCFS (First-Come-First-Served): Non-preemptive; processes run in arrival order. Simple but can cause the convoy effect when a long job blocks shorter ones [2].

SJF (Shortest Job First): Selects the process with the smallest remaining burst time. Minimizes average waiting time in theory but can cause starvation of long jobs; may be preemptive (Shortest Remaining Time First) or non-preemptive [3].

Round Robin (RR): Preemptive; each process gets a fixed time quantum. Fair and responsive; performance depends on quantum size [2].

MLFQ (Multi-Level Feedback Queue): Uses multiple priority queues with different time quanta. Processes move between queues based on behavior (e.g., I/O-bound get higher priority), balancing response time and throughput [4].

C. Educational Visualizers

Several CPU scheduling visualizers exist for teaching [5], [6]. This project differentiates by combining: (1) a clean Strategy-pattern design for multiple algorithms; (2) explicit modeling of context-switch overhead and I/O; (3) real-time metrics and Gantt chart; and (4) a modern React-based UI with animations to improve engagement.

III. System Design

A. Architecture Overview

The system is divided into a **core** scheduling engine (algorithm-agnostic) and a **presentation** layer (React components). The core maintains process lists, ready and waiting queues, and the current simulation time; it delegates "which process runs next" to a pluggable **SchedulingStrategy**. This follows the Strategy design pattern [7], enabling new algorithms to be added without changing the scheduler logic.

B. Strategy Pattern

Context: The Scheduler class holds the ready queue, running process, and simulation state. It calls `strategy.selectNext(readyQueue, currentTime)` to choose the next process and `strategy.shouldPreempt(...)` to decide preemption.

Strategy interface: `SchedulingStrategy` defines `selectNext()`, `shouldPreempt()`, and optional `onProcessAdmit()` (e.g., for MLFQ queue assignment).

Concrete strategies: `FCFSStrategy`, `SJFStrategy`, `RoundRobinStrategy`, and `MLFQStrategy` implement the interface. Each encapsulates one algorithm and can be swapped at runtime.

C. Process Model

Each process is represented by a Process object with: unique id, name, arrival time, burst time, remaining burst time, state (NEW, READY, RUNNING, WAITING, TERMINATED), and timing fields (waiting time, turnaround time, response time). For MLFQ, processes also have a priority-queue level and support promote/demote. State transitions are triggered by the scheduler and recorded for the Gantt chart and metrics.

D. Simulation Loop

One step of the simulation: (1) check for newly arrived processes and admit them to the ready queue; (2) check for I/O completions and move processes from waiting back to ready; (3) if no process is running, use the strategy to select the next process, perform a context switch (with configurable overhead), and dispatch it; (4) execute the running process for one time unit, update burst remaining, and handle completion, I/O request, or preemption; (5) if no process is running and the ready queue is empty, record CPU idle time. The loop runs repeatedly with an optional delay for visualization.

E. Metrics

The following are computed from completed processes and simulation time: average waiting time (mean time spent in the ready queue); average turnaround time (mean time from arrival to completion); average response time (mean time from arrival to first execution); CPU utilization as (total time − idle time − context-switch time) / total time × 100%; throughput (processes completed per unit time); and total context switches.

IV. Implementation

A. Technology Stack

Frontend: React 18, with hooks for state management. Animation: Framer Motion for transitions and layout animations. Build: Create React App (React Scripts 5.x). Utilities: UUID for process identifiers. The application runs entirely in the browser; no backend or database is required.

B. Core Modules

`Process.js`: Defines ProcessState enum and Process class with `execute()`, `startIO()`, `processIO()`, `reset()`, and state transition logic. `Scheduler.js`: Implements the main scheduler (queues, step loop, context switch, preemption handling, arrival and I/O completion checks, and `calculateMetrics()`). `strategies/SchedulingStrategy.js`: Abstract base with `selectNext()`, `shouldPreempt()`, `onProcessAdmit()`, `getQuantum()`. Concrete strategy files implement FCFS, SJF, Round Robin, and MLFQ.

C. React Components

`App.jsx`: Root layout with header (title, simulation time, status), left panel (control panel), center (live view or comparison), right panel (metrics); integrates `useScheduler` hook. `ControlPanel`: Add process (random/custom burst), select algorithm, inject I/O, kill process, start/pause/step/reset. `CPUVisualizer`: Displays the CPU and the currently running process. `ReadyQueue`, `WaitingQueue`: List processes with color coding. `GanttChart`: Timeline of execution, idle, and context-switch segments. `MetricsDashboard`: Real-time computed metrics. `AlgorithmComparison`, `AlgorithmInfo`: Educational explanations. `ContextSwitchOverlay`: Indicates when a context switch is in progress.

D. Custom Hook

`useScheduler.js`: Maintains scheduler instance, current strategy, and React state derived from `scheduler.getState()`. Exposes actions (add process, set strategy, step, run, pause, reset, inject I/O, kill process)

and state (currentTime, running process, queues, metrics, ganttChart, isRunning, isComplete). Ensures the UI updates after each step or state change.

E. Visual Design

Process states are color-coded: green (Running), yellow (Ready), red (Waiting), gray (Terminated). The interface uses a grid background and clear typography. Animations (e.g., process blocks moving between queues) are kept short to avoid obscuring the underlying logic.

V. Results and Discussion

A. Functional Behavior

The visualizer correctly implements FCFS (convoy effect visible with one long job followed by short ones), SJF (shortest burst selected; starvation can be demonstrated with continuous short arrivals), Round Robin (fair time slices and preemption when quantum expires), and MLFQ (queue levels and promotion/demotion based on CPU vs. I/O behavior). Context-switch overhead is applied and reflected in the Gantt chart and utilization. I/O injection moves the running process to Waiting and allows other processes to run; on I/O completion, processes return to the ready queue.

B. Metrics Validation

Computed metrics align with standard definitions: waiting time as time in ready queue, turnaround as completion minus arrival, and utilization as busy time over total time. Users can compare metrics across algorithms for the same workload to see trade-offs (e.g., FCFS vs. RR vs. SJF on waiting and response times).

C. Educational Value

Instructors can use the tool to illustrate: **Preemption:** RR and MLFQ interrupting a running process. **Context switching:** Overhead and its effect on utilization. **Starvation:** e.g., long jobs under SJF with many short jobs. **Convoy effect:** One long FCFS job delaying many short ones. The Strategy pattern in the codebase also serves as a software-engineering example of design patterns in systems software.

D. Limitations

The simulation is discrete and single-CPU only. Real systems have more complex behavior (e.g., kernel vs. user mode, caching effects). The MLFQ implementation uses a fixed number of queues and simple promotion/demotion rules for educational clarity.

VI. Conclusion

Schedulix provides an accurate, interactive CPU scheduling visualizer that demonstrates the illusion of parallelism and supports FCFS, SJF, Round Robin, and MLFQ through a Strategy-based design. The implementation separates core scheduling logic from the React UI and delivers real-time metrics and a Gantt chart to support teaching and self-study. Future work could include additional algorithms (e.g., Priority Scheduling), configurable time quanta and context-switch cost, workload presets, and export of Gantt charts or metrics for reports.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[2] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Upper Saddle River, NJ, USA: Pearson, 2018.

[3] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Pearson, 2015.

[4] R. Arpaci-Dusseau and A. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018. [Online]. Available: http://ostep.org

[5] "CPU Scheduling Visualizer," various implementations. [Online]. Available: https://github.com/topics/cpu-scheduling-visualizer

[6] "OS Process Scheduling Simulators," educational resources. [Online]. Available: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.