LOVELY PROFESSIONAL UNIVERSITY

Transforming Education Transforming India

Final Report

Real-Time Multi-threaded Application Simulator

Operating Systems (CSE 316)

By

Sr. No.	Registration No	Name of Students	Roll No
1.	12501247	KUMAR ABHINASH	56
2.	12528614	AJAY UPADHYAY	47

SECTION : K24LL

Submitted To : Sachin Kumar Chawla, Assistant Professor
School of Computer Science and Engineering
Lovely Professional University

Abstract

This project presents a Real-Time Multi-threaded Application Simulator designed to demonstrate the core concepts of multithreading, CPU scheduling, and synchronization in operating systems. The simulator visually represents thread states, execution timelines, and resource interactions using semaphores and monitors. It allows users to experiment with threading models such as Many-to-One, One-to-One, and Many-to-Many while observing real-time scheduling behavior. By integrating interactive controls, event logs, and dynamic CPU allocation, the system provides an intuitive understanding of thread management. This project serves as an effective educational tool for learning and analyzing real-time multi-threaded environments.

Acknowledgement

I would like to express my sincere gratitude to Lovely Professional University for providing me with the opportunity to work on this project titled *"Real-Time Multi-threaded Application Simulator."* This project has been an enriching learning experience that allowed me to practically apply the theoretical concepts of Operating Systems, including multithreading, scheduling algorithms, and synchronization mechanisms.

I extend my heartfelt thanks to my faculty for assigning this meaningful project and for offering continuous guidance, motivation, and valuable feedback throughout the development process. Their support helped me explore complex technical concepts with clarity and confidence.

I would also like to acknowledge the collaborative spirit and contribution of my group member, Ajay Upadhyay, for his support and cooperation during various stages of the project. Working as a team enhanced our problem-solving abilities, communication skills, and understanding of real-time system simulation.

Finally, I am grateful to all digital tools, documentation resources, and development environments that assisted me in building a functional and visually interactive simulator. This project has significantly strengthened my programming abilities, conceptual understanding, and practical implementation skills.
I deeply appreciate the opportunity to learn, explore, and innovate through this assignment.

Project Report: Real-Time Multi-threaded Application Simulator

1. Project Overview

This project demonstrates how different multithreading models work (Many-to-One, One-to-One, Many-to-Many).

It simulates thread creation, scheduling, CPU time-sharing, and synchronization using semaphores and monitors.

The simulator visually shows thread states: NEW, READY, RUNNING, BLOCKED, TERMINATED.

The **Real-Time Multi-threaded Application Simulator** is an educational and interactive software system designed to visually demonstrate how operating systems manage threads, CPU scheduling, and synchronization mechanisms. Modern computers rely heavily on multithreading to achieve parallelism, improve performance, and increase responsiveness. However, the internal working of thread models, schedulers, and resource synchronization is often invisible to students. This project solves that problem by providing a **fully visual, real-time simulation** of how threads behave inside an operating system.

The primary goal of this simulator is to make complex OS concepts easy to understand by showing actual thread behavior through animations, color-coded states, and step-by-step execution. Users can create multiple threads, configure their burst times and priorities, select a threading model, choose a CPU scheduling algorithm, and then watch the simulation unfold in real time. This results in a hands-on learning experience where theoretical OS concepts come alive as dynamic visual interactions.

The simulator supports **three major multithreading models** used in real operating systems:

1. **Many-to-One (User-Level Threads):**
   In this model, many user-level threads are mapped to a single kernel thread. Blockage in one thread results in the entire process getting blocked. Our simulator visualizes this behavior by showing how all threads freeze when a single thread performs a blocking wait operation.

2. **One-to-One (Kernel-Level Threads):**
   Each user thread corresponds to one kernel thread. This allows real parallelism, better CPU utilization, and individual thread blocking without affecting others. The

simulator displays multiple CPU cores (kernel threads), and each can run a thread independently.

3. **Many-to-Many:**
   A flexible hybrid model where multiple user-level threads are mapped to multiple kernel threads. It balances performance and resource utilization. The simulator demonstrates how kernel pools work and how thread interference or blocking affects shared kernel resources.

Along with threading models, the simulator implements three widely used **CPU scheduling algorithms**:

- **FCFS (First Come First Serve):** The earliest arriving thread in the ready queue runs first.
- **Priority Scheduling:** Threads with higher priority values are selected first.
- **Round Robin:** Each thread gets a time slice (quantum), ensuring fairness and preventing starvation.

Users can dynamically switch the scheduler and observe how it affects thread execution order and completion time.

Another highlight of this project is its strong focus on **synchronization**, a crucial aspect of operating systems. The system implements:

- **Semaphores**, which regulate access to shared resources using counters and queues.
- **Monitors**, which enforce mutual exclusion and condition-based waiting.

Users can create semaphores/monitors and assign threads to wait or signal them. The simulator visualizes blocking, unblocking, ownership, and waiting queues in real time.

The output visualization panel includes:

- Live **Gantt chart** representing state history (Running, Ready, Blocked, Terminated).
- **CPU panel** showing which thread is running.
- Per-thread history tiles showing the last 40 state transitions.
- Event logs that explain every action, such as “T1 acquired semaphore S1” or “T3 blocked on monitor M1”.

The entire project is built using **pure HTML, CSS, and JavaScript**, without relying on any external libraries. The logic runs completely in the browser, guaranteeing portability and ease of use. This eliminates the need for backend servers while ensuring 100% transparency of how the simulation works.

Overall, the Real-Time Multi-threaded Application Simulator is not just a software tool but a comprehensive educational aid. It combines theory with visualization to enhance conceptual clarity, practical understanding, and debugging skills. The project demonstrates real OS behavior in an intuitive and interactive manner, making it beneficial for students, educators, and anyone interested in understanding multithreading and parallel processing systems.

2. Module-Wise Breakdown

The **Real-Time Multi-threaded Application Simulator** is structured into three major modules. Each module plays a specific role in simulating how operating systems manage threads, scheduling, and synchronization. The division ensures clarity, modularity, easier debugging, and improved understanding of how each component contributes to the whole system.

Module 1: User Interface (UI) Module

This module is responsible for all visuals, user controls, and interaction components. Its purpose is to make the simulation understandable and intuitive for the user.

Responsibilities:

✓ Display thread states and transitions
✓ Provide controls to create threads, semaphores, and monitors
✓ Allow selection of threading models and scheduling algorithms
✓ Show CPU visualization, thread lanes, Gantt chart, and event logs

How it Works:

The UI is created using **HTML and CSS**, including a card-based layout that visually separates controls, CPU view, and logs. It dynamically updates whenever the simulation engine changes the state of a thread.

Every UI element (buttons, sliders, dropdowns) is linked to a JavaScript function.
For example:

- Clicking **Add Thread** calls makeThread()
- Clicking **Play** starts the simulation timer
- Changing model updates modelSelect and re-renders the UI

The visualization includes:

- **Thread Rows:** Each thread shows its current state, burst time, and recent history tiles.
- **CPU Slots:** Colors show whether the CPU/core is idle or busy.
- **Gantt Chart:** Displays color-coded tiles:
  🟢 Running • 🟡 Ready • 🔴 Blocked • ⚪ Terminated

This module presents an OS-like environment without requiring real multitasking, making it easy to learn concepts visually.

Module 2: Simulation Engine (Core Logic Module)

This is the heart of the project. It handles all logical operations like thread scheduling, state transitions, kernel-slot allocation, and clock ticks.

Responsibilities:

✓ Manage thread lifecycle: New → Ready → Running → Blocked → Terminated
✓ Implement scheduling algorithms (FCFS, Priority, Round Robin)
✓ Manage CPU time slices (quantum)
✓ Handle threading models (Many-to-One, One-to-One, Many-to-Many)
✓ Generate a global clock (ticks)
✓ Update thread histories for Gantt chart

How it Works:

The engine runs inside a function called **tickOnce()**, triggered every few milliseconds by a timer when simulation is running.

Every tick performs the steps:

1. Increase the global time tick
2. Find READY threads
3. Use the selected scheduler to pick threads
4. Assign threads to CPU slots depending on the model
5. Reduce their burst time by 1
6. Change state to TERMINATED if finished
7. Update their history for visualization

Thread Models Inside Engine:

Many-to-One:

- All user threads mapped to ONE kernel thread
- If one thread blocks → entire process blocks
- Simulator shows this by blocking all threads together

One-to-One:

- Each user thread mapped to one kernel thread
- Real parallelism possible
- Each thread processed independently

Many-to-Many:

- Many user threads mapped to a limited pool of kernel threads
- Balanced performance
- Blocking releases kernel slot for another thread

The engine ensures realistic OS-like behavior while keeping logic simple and efficient.

Module 3: Synchronization Module

This module simulates critical OS synchronization mechanisms: **Semaphores** and **Monitors**. It teaches how threads coordinate to share resources without conflict.

Responsibilities:

✓ Manage semaphore values and waiting queues
✓ Implement P (wait) and V (signal) operations
✓ Handle monitor lock/unlock and wait queues
✓ Apply blocking rules based on threading model

How it Works:

Semaphores:

- Created with a name (S1, S2...) and initial value
- sem_wait() decreases value or blocks the thread
- sem_signal() wakes the next waiting thread

- UI updates queue instantly

Monitors:

- Ensure only one thread enters the critical section
- If locked, entering threads go to wait queue
- monitor_exit() releases lock or wakes next waiting thread

Effect of Threading Models:

If a thread blocks while holding a semaphore or monitor:

- In Many-to-One → all threads block
- In One-to-One → only that thread blocks
- In Many-to-Many → kernel slot is freed

This makes synchronization behavior realistic and educational.

Summary of Module Interaction:

- **User Interface** triggers actions
- **Simulation Engine** executes logic
- **Synchronization Module** manages resource control
- UI **updates dynamically** every tick

All modules work together to create a complete, real-time OS simulation environment.

3. Functionalities

The **Real-Time Multi-threaded Application Simulator** provides a wide range of functionalities designed to help users understand multithreading, scheduling algorithms, CPU management, and synchronization techniques in operating systems. Each functionality is built to replicate how real OS kernels behave, while still keeping the simulation simple, visual, and interactive. Below is a detailed explanation of all key features.

---

1. Thread Creation and Configuration

The simulator allows users to create any number of threads with customizable properties.

Features:

- Assign a **burst time**, representing how long the thread needs CPU.
- Set a **priority value** for priority scheduling.
- Automatically generate threads using the **Add Random** button.

Purpose:

This helps students see how different priorities and burst times affect scheduling decisions and thread lifecycle.

---

2. Multiple Threading Models

Users can switch between three major threading models:

a) Many-to-One:

Multiple user threads mapped to a single kernel thread.
Blocking one thread blocks all.

b) One-to-One:

Each user thread has its own kernel thread.
Allows parallel execution.

c) Many-to-Many:

Multiple user threads share a pool of kernel threads.
Efficient and flexible.

Purpose:

This functionality shows how real operating systems like Linux or Solaris handle threads differently based on their threading model.

---

3. CPU Scheduling Algorithms

The simulator supports three scheduling policies:

a) FCFS (First Come First Serve):

Threads are executed in the order they arrive.

b) Priority Scheduling:

Higher-priority threads run first.

c) Round Robin:

Each thread gets a time slice (quantum), ensuring fairness.

Purpose:

Students can observe differences in thread execution, waiting time, and starvation issues.

---

4. Real-Time CPU Visualization

A CPU panel displays kernel threads (CPU cores) and shows which thread is currently running.

Features:

- Active threads highlight dynamically.
- Idle CPUs show "Idle".
- Color-coded states for clarity:
  - 🟢 Running
  - 🟡 Ready
  - 🔴 Blocked
  - ⚪ Terminated

Purpose:

Gives a clear, real-time view of how the OS assigns CPU time.

---

5. Gantt Chart (Thread State History)

Each thread has a row showing its state changes over time.

Features:

- Latest 40 ticks displayed
- Color-coded tiles
- Updates every tick

Purpose:

Helps in understanding scheduling patterns, CPU utilization, and thread competition.

---

6. Semaphores and Monitors (Synchronization)

The simulator fully implements **synchronization mechanisms** used in OS:

Semaphores:

- Create semaphore with a custom name and initial value
- Threads can perform wait() and signal()
- Waiting queue is displayed

Monitors:

- Create monitors
- Only one thread enters at a time
- Others wait in queue

Purpose:

Shows how synchronization prevents race conditions and handles shared resources safely.

---

7. Step-by-Step Simulation Control

Users have fine control over the simulation:

- **Play:** Start continuous execution
- **Pause:** Stop without losing progress
- **Step:** Execute exactly one tick
- **Reset:** Clear all threads and restart simulation

Purpose:

Allows detailed debugging and learning, similar to stepping through OS code.

---

8. Import & Export Scenarios

The tool allows saving and loading simulations using JSON files.

Features:

- Export all threads, semaphores, monitors, settings
- Import previously saved scenarios

Purpose:

Helps in reproducibility and sharing simulation setups.

---

9. Event Log System

Every action performed is recorded in an event log, including:

- Thread creation
- Blocking/unblocking
- Semaphore operations
- Monitor entry/exit
- Termination

Purpose:

Provides transparency and helps track system behavior.

4. Technology Used

The **Real-Time Multi-threaded Application Simulator** is built entirely using web-based technologies to ensure maximum accessibility, portability, and ease of visualization. The project does not require any backend server or external libraries, making it lightweight, fast, and completely self-contained. The following technologies form the core foundation of this simulator:

---

1. Programming Languages Used

a) HTML (HyperText Markup Language)

HTML is used to structure all visual elements of the simulator. It defines the overall layout, including the control panel, CPU visualization area, thread lanes, Gantt chart, event logs,

buttons, inputs, and text fields. The HTML document follows a clean and semantic structure, making it easy to modify and extend.

b) CSS (Cascading Style Sheets)

CSS provides styling, animations, and visual aesthetics.
Key styling features used include:

- Gradient backgrounds
- Glass-like card layouts
- Rounded UI components
- Color-coded thread states
- Responsive grid design for mobile and desktop

The carefully designed UI helps users quickly interpret complex multithreading behaviors.

c) JavaScript (ES6+)

JavaScript forms the **core logic** of the entire simulation.
It manages:

- Thread creation
- Scheduling algorithms
- CPU tick updates
- Thread state transitions
- Synchronization (semaphores & monitors)
- UI rendering and dynamic updates

JavaScript acts as the "engine" of the operating system simulation, mimicking how real kernels schedule and manage threads.

---

2. Libraries and Tools

This project uses **no third-party libraries**, making the simulation:
✓ Lightweight
✓ Fast
✓ Fully customizable
✓ Offline-friendly

All logic is coded manually using vanilla JavaScript to help students learn the internal behavior clearly rather than relying on pre-built libraries.

---

3. Other Tools Used

a) GitHub (Version Control System)

GitHub is used to track project revisions, maintain code history, and manage contributions from group members.

Students use GitHub to:

- Create a project repository
- Maintain at least 7 commits
- Use branches for new features
- Merge branches after testing
- Upload final project code and documentation

This promotes good software engineering practices and teamwork.

b) Browser Developer Tools

Tools like Chrome DevTools or Firefox Inspector are used for debugging JavaScript, adjusting layouts, and monitoring console logs.

c) Code Editors (VS Code, Sublime, etc.)

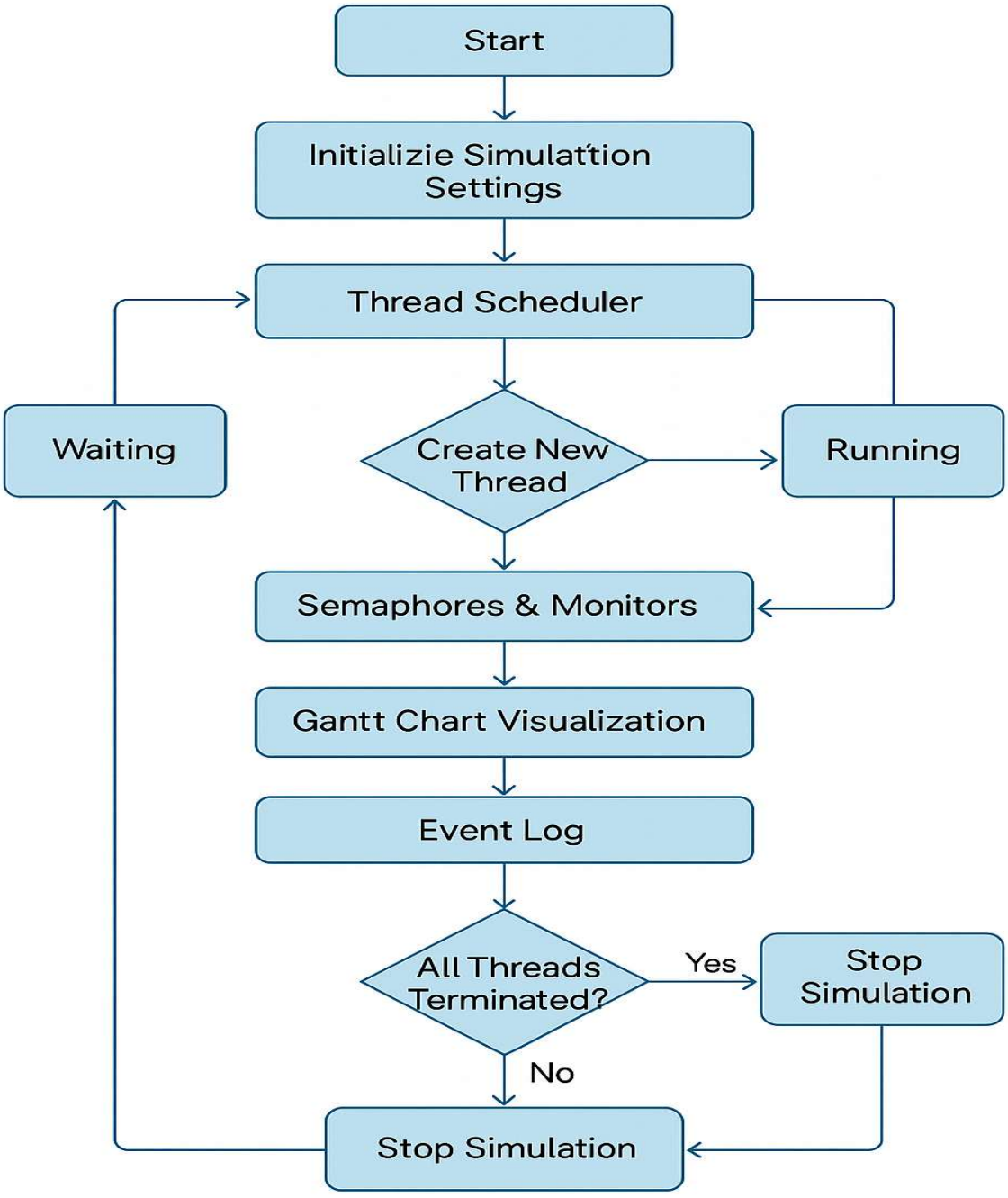
A modern text editor is used to write code efficiently with syntax highlighting and quick formatting.

Summary

The simulator uses **pure HTML, CSS, and JavaScript**, combined with **GitHub** for version control. No external libraries ensures full transparency and helps students understand how operating systems manage threads, schedule tasks, and handle synchronization in real time.

---

5. Flow Diagram



The flow diagram represents the entire working pipeline of the **Real-Time Multi-threaded Application Simulator**. It visually explains how user actions interact with the simulation engine and how thread states evolve over time. Below is the step-by-step flow:

---

1. Start

The simulation begins when the user loads the webpage or resets the system.

2. Initialize Simulation Settings

Default threading model, scheduler, tick counter, and empty thread lists are prepared.

3. Thread Scheduler

The scheduler selects which thread will run based on:

- Round Robin
- FCFS
- Priority Scheduling

The flow moves depending on thread availability.

---

4. Create New Thread (Decision Point)

If the user clicks **Add Thread**, the simulator creates a new thread and adds it to the READY queue.

If not, the scheduler proceeds with existing threads.

---

5. Running State

The selected thread executes for one CPU tick.
Burst time decreases, and state changes get recorded.

If the thread blocks, flow goes to the **Waiting** box.

---

6. Waiting State

Threads that:

- Wait on semaphores,
- Wait on monitors, or
- Are blocked due to Many-to-One model limitations

are placed here.

---

7. Semaphores & Monitors Module

The synchronization module checks:

- Semaphore wait/signal
- Monitor enter/exit
- Waiting queues

Threads may unblock or remain waiting.

---

8. Gantt Chart Visualization

The thread's state history is updated:

- Running → 🟢
- Ready → 🟡
- Blocked → 🔴
- Terminated → ⚪

---

9. Event Log Update

Every action is recorded:

- "T1 created"
- "T2 blocked on semaphore"
- "T3 terminated"

---

10. All Threads Terminated? (Decision Point)

If YES → Stop simulation
If NO → Loop back to Thread Scheduler

---

11. Stop Simulation

The simulation stops when:

- User pauses
- All threads terminate
- User clicks reset

6. Revision Tracking on GitHub

Repository Name: multi-threaded-simulator

GitHub Link: **https://github.com/abhiraj66369-commits/multi-threaded-simulator**

At least 7 commits with meaningful messages should be added:

Commit No.	Commit Message
1	Added project structure and initial HTML
2	Added CSS styling and responsive design
3	Implemented JavaScript simulation engine
4	Added semaphore and monitor logic
5	Added UI visual timeline and CPU section
6	Added README.md and documentation
7	Added final report PDF & DOCX

7. Conclusion and Future Scope

Conclusion:

The **Real-Time Multi-threaded Application Simulator** successfully demonstrates the fundamental concepts of multithreading, CPU scheduling, and synchronization in operating systems through an interactive and visually rich environment. Modern operating systems perform numerous operations simultaneously, and understanding how threads behave internally is often challenging for students. This project simplifies these concepts by simulating them in a controlled, easy-to-understand, and visually appealing format.

Through this simulator, users can observe how different threading models—**Many-to-One, One-to-One, and Many-to-Many**—impact the behavior of threads and their interaction with CPU cores. The visualization of these models highlights how thread blocking, parallel execution, and kernel-level mapping differ across systems. By switching between these models, users gain a deeper understanding of how real operating systems schedule and manage threads at both user and kernel levels.

Additionally, the implementation of CPU scheduling algorithms—**FCFS, Priority Scheduling, and Round Robin**—provides a practical demonstration of how each algorithm influences execution order, waiting time, and overall efficiency. Seeing how these algorithms work in real time helps learners comprehend the advantages and limitations of each approach.

The inclusion of **synchronization mechanisms**, such as **semaphores and monitors**, further enhances the simulator's educational value. By visualizing blocking, signaling, waiting queues, and lock ownership, users can better understand how operating systems prevent race conditions and ensure safe access to shared resources. The synchronization module showcases practical scenarios where threads compete for shared objects, making theoretical concepts more relatable and easier to grasp.

The simulator's intuitive user interface and real-time Gantt chart visualization make thread execution transparent. Users can follow thread histories, state transitions, CPU assignments, and event logs step-by-step. The ability to import and export scenarios also encourages experimentation and reproducibility for learning and teaching purposes.

Overall, this project demonstrates how complex OS-level multitasking and synchronization concepts can be modeled in a simple, accessible, and interactive way using just HTML, CSS, and JavaScript. It not only strengthens conceptual clarity but also enhances analytical thinking by allowing users to explore different operating system designs. The simulator proves to be a valuable educational tool for students, teachers, and anyone interested in understanding OS internals.

Future Scope:

The **Real-Time Multi-threaded Application Simulator** is already a powerful educational tool, but it has significant potential for future improvements and expansion. These enhancements can make the simulator more realistic, feature-rich, and suitable for advanced learning or research in operating systems and concurrent computing.

One major area for expansion is the integration of **actual parallel execution** using technologies such as **Web Workers** in JavaScript. This would allow threads to execute truly in parallel, just like actual operating system threads, rather than being simulated on a single event loop. It would make the behavior of the simulator even closer to a real kernel.

Another potential improvement is the addition of **advanced scheduling algorithms**, such as Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Multilevel Feedback

Queue Scheduling (MLFQ), and Completely Fair Scheduler (CFS) used in Linux. This would give learners a broader understanding of real-world scheduling strategies and their trade-offs.

The project could also introduce **deadlock detection and avoidance mechanisms**, such as:

- Banker's Algorithm
- Resource Allocation Graph (RAG)
- Deadlock prevention rules

This would allow users to simulate more complex synchronization scenarios and learn how deadlocks occur and how operating systems handle them.

Another promising area of expansion is **performance monitoring**, where metrics like CPU utilization, waiting time, turnaround time, and throughput are calculated dynamically. These metrics would help users compare different threading models and scheduling algorithms in a more analytical way.

The simulator can also be extended to support **process-level simulations**, allowing users to visualize process creation, context switching, memory allocation, and inter-process communication (IPC). This would turn the tool into a more complete operating system simulator, covering even more core OS concepts.

A dedicated **mobile-friendly app** or **desktop application** could also be developed using frameworks like React, Electron, or Flutter to improve accessibility.

Finally, the project can be improved with **AI-based recommendations**, where the system analyzes thread behavior and suggests better scheduling algorithms or detects anomalies in execution.

Overall, the simulator has extensive room for future improvements, making it a scalable and evolving platform for OS education and experimentation.

8. References

1. Operating System Concepts – Abraham Silberschatz

2. Modern Operating Systems – Andrew S. Tanenbaum

3. Mozilla Developer Docs (JavaScript)

4. W3C HTML/CSS Standards

Other References

1. **Silberschatz, Abraham; Galvin, Peter B.; Gagne, Greg.**
   Operating System Concepts. 10th Edition. Wiley, 2018.
   (Widely used textbook explaining threads, scheduling, and synchronization.)

2. **Tanenbaum, Andrew S.; Bos, Herbert.**
   Modern Operating Systems. 4th Edition. Pearson, 2015.
   (Detailed coverage of threading models and OS architecture.)

3. **Stallings, William.**
   Operating Systems: Internals and Design Principles. 9th Edition. Pearson, 2018.
   (Explains semaphores, monitors, and process synchronization.)

4. **Mozilla Developer Network (MDN).**
   JavaScript Documentation.
   https://developer.mozilla.org

Appendix A: AI-Generated Project Elaboration / Breakdown Report

(Full detailed breakdown generated exactly as required by LPU guidelines)

1. Project Overview (AI-Generated Detailed Explanation)

The *Real-Time Multi-threaded Application Simulator* is a fully interactive, browser-based tool that demonstrates how multithreading works inside an Operating System. The simulator visually shows how threads are created, scheduled, executed, blocked, terminated, and synchronized using semaphores and monitors.

This project aims to simplify concepts like CPU scheduling, thread states, blocking, and concurrency — which are normally difficult to understand without visualization. The simulator uses animations, colored indicators, Gantt timelines, CPU slot boxes, state tiles, and runtime logs to show exactly what is happening inside the "Operating System".

The simulator supports 3 major threading models:

1. **Many-to-One:** Multiple user threads mapped to a single kernel thread. Entire process blocks if one thread blocks.

2. **One-to-One:** Each user thread mapped to one kernel thread. True parallelism possible.

3. **Many-to-Many:** Multiple user threads mapped to a flexible number of kernel threads. Scalable & efficient.

Along with threading models, the simulator includes 3 scheduling algorithms:

- **Round Robin (RR)**
- **First Come First Serve (FCFS)**
- **Priority Scheduling**

The project also demonstrates **process synchronization** using:

- **Semaphores** (wait/signal)
- **Monitors** (enter/exit with wait queues)

This helps students visualize how concurrency control works.

Appendix

2. Module-Wise Breakdown (AI-Generated)

Module 1: User Interface (UI) Module

This module creates the complete visual layout of the simulator.
It includes:

- Control panel (add thread, add semaphore, add monitor)
- Visualization panel (thread lanes, CPU slots, history tiles)
- Event logs, Gantt logs, state displays
- Responsive design for all screen sizes

It interacts with the simulation engine and updates the UI after every tick.

Module 2: Simulation Engine Module

This is the "brain" of the project.
It handles:

- Thread creation
- State transitions
- Scheduling logic (RR/FCFS/Priority)
- Burst time execution
- Per-tick update (tickOnce function)

- Kernel thread allocation
- Applying threading model rules (M2O, O2O, M2M)

It runs on an internal timer and updates thread states in real time.

Module 3: Synchronization Module

This module simulates OS-level synchronization:

- **Semaphores:** value, queue, wait(), signal()
- **Monitors:** lock, owner, queue, enter(), exit()
- Applying blocking rules
- Integrating with models (M2O blocks entire process)

This gives a complete visualization of concurrency control.

---

3. Functionalities (AI-Generated)

The simulator provides the following core functionalities:

Threading Features

- Create user threads with burst time & priority
- Change model at runtime (M2O, O2O, M2M)
- Real-time thread state transitions
- Visual CPU allocation

Scheduling Features

- Round Robin time slicing
- Priority scheduling
- FCFS scheduling
- Running threads get tiles colored according to state

Synchronization Features

- Create semaphores and monitors

- Wait & signal on semaphores
- Enter & exit monitors
- Blocks threads realistically
- Many-to-One blocks entire process

User Interaction Features

- Play/Pause simulation
- Step-by-step debugging
- Reset simulation
- Export & import JSON scenarios
- Event logs + Gantt timeline

---

4. Technology Recommendations (AI-Generated)

Programming Languages:

- HTML5
- CSS3
- JavaScript (ES6)

Libraries:

- No external libraries required (lightweight)
- ReportLab (only for generating PDF)
- python-docx (for DOCX generation)

Tools:

- GitHub for revision tracking
- VS Code for development
- Browser console for debugging

---

5. Execution Plan (AI-Generated)

Step 1 — Build UI layout
Step 2 — Implement thread object structure
Step 3 — Build scheduling engine
Step 4 — Add threading models
Step 5 — Add semaphore & monitor simulation
Step 6 — Create visualization
Step 7 — Add logs, controls, presets
Step 8 — Testing + GitHub commits
Step 9 — Report + documentation

Appendix B: Problem Statement

As provided by the university:

Project Title: Real-Time Multi-threaded Application Simulator

Project Description:
Develop a simulator to demonstrate multithreading models (Many-to-One, One-to-One, Many-to-Many) and thread synchronization using semaphores and monitors. The simulator should visualize thread states and interactions, providing insights into thread management and CPU scheduling in multi-threaded environments.

Appendix C: Complete Source Code of the Project

**The source code presented in this appendix forms the functional core of the Real-Time Multi-threaded Application Simulator.
It includes:

- User Interface layout (HTML)
- Styling and responsive design (CSS)
- Complete simulation engine for thread scheduling and synchronization (JavaScript)
- Event logging, resource visualization, and interactive controls

This code can be copied and executed as a standalone file, as it does not require any backend or external libraries.**

CODE

<!doctype html>

<html lang="en">

```
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width,initial-scale=1" />
<title>Real-Time Multi-threaded Application Simulator</title>
<style>
 :root{
  --bg:#0f1724; --card:#0b1220; --muted:#9aa7bf; --accent:#6ee7b7;
  --danger:#ff6b6b; --yellow:#ffd166; --glass: rgba(255,255,255,0.04);
  --tile-size:18px;
  font-family: Inter, ui-sans-serif, system-ui, -apple-system, "Segoe UI", Roboto, "Helvetica Neue", Arial;
 }
 *{box-sizing:border-box}
 body{margin:0;background:linear-gradient(180deg,#071021 0%, #0f1724 100%);color:#e6eef6;min-height:100vh;display:flex;flex-direction:column}
 header{padding:18px 20px;display:flex;align-items:center;gap:16px;border-bottom:1px solid rgba(255,255,255,0.04)}
 header h1{font-size:18px;margin:0}
 header .meta{font-size:13px;color:var(--muted)}
 .wrap{display:grid;grid-template-columns:320px 1fr 360px;gap:16px;padding:16px;align-items:start}
 /* responsive */
 @media (max-width:1100px){ .wrap{grid-template-columns:1fr; padding:12px} .right, .left{order:99} .center{order:1} }
 /* cards */
 .card{background:linear-gradient(180deg, rgba(255,255,255,0.02), rgba(255,255,255,0.01));border-radius:12px;padding:12px;border:1px solid rgba(255,255,255,0.03);box-shadow:0 6px 24px rgba(2,6,23,0.6)}
 .left .card, .right .card{margin-bottom:12px}
```

```
label{font-size:13px;color:var(--muted);display:block;margin-bottom:6px}

input, select, button{font-size:14px;padding:8px;border-radius:8px;border:1px solid rgba(255,255,255,0.06);background:transparent;color:inherit}

.row{display:flex;gap:8px;align-items:center}

button.primary{background:linear-gradient(90deg,#22c1c3,#6ee7b7);border:none;color:#062122;font-weight:600}

button.warn{background:linear-gradient(90deg,#ff8a65,#ff6b6b);border:none;color:white}

.controls{display:flex;flex-direction:column;gap:8px}

.small{font-size:12px;color:var(--muted)}

/* center visualization */

.center{display:flex;flex-direction:column;gap:12px}

.toolbar{display:flex;gap:8px;align-items:center;flex-wrap:wrap}

.lanes{background:linear-gradient(180deg, rgba(255,255,255,0.01), rgba(255,255,255,0.005));padding:12px;border-radius:10px;min-height:200px;overflow:auto}

.thread-row{display:flex;gap:8px;align-items:center;padding:6px;border-radius:8px;margin-bottom:6px}

.thread-label{width:90px;font-weight:700;color:#cfeef0}

.tiles{display:flex;gap:3px;flex-wrap:nowrap;min-width:120px;overflow:auto;padding:4px}

.tile{width:var(--tile-size);height:var(--tile-size);border-radius:4px;flex:0 0 auto;box-shadow:inset 0 -1px 0 rgba(0,0,0,0.2)}

.state-Running{background:#22c1c3}

.state-Ready{background:#ffd166}

.state-Blocked{background:#ff6b6b}

.state-Terminated{background:#94a3b8}

.state-New{background:#7c3aed}
```

```
 .cpu-box{display:flex;gap:8px;align-items:center;padding:8px;border-radius:8px;background:linear-gradient(90deg, rgba(255,255,255,0.01), rgba(255,255,255,0.005))}

 .cpu-slot{flex:1;padding:8px;border-radius:8px;border:1px dashed rgba(255,255,255,0.04);text-align:center;font-size:13px}

 .log{max-height:240px;overflow:auto;padding:8px;background:rgba(0,0,0,0.12);border-radius:8px;border:1px solid rgba(255,255,255,0.02)}

 .muted{color:var(--muted);font-size:13px}

 .sep{height:1px;background:rgba(255,255,255,0.02);margin:8px 0;border-radius:4px}

 /* right panel */

 .resource{display:flex;flex-direction:column;gap:8px}

 .resource div{display:flex;justify-content:space-between;align-items:center;padding:6px;border-radius:8px;background:var(--glass)}

 .badge{padding:6px 8px;border-radius:999px;background:rgba(255,255,255,0.03);font-weight:600}

 .grid-2{display:grid;grid-template-columns:1fr 1fr;gap:8px}

 .footer{padding:12px;margin-top:auto;color:var(--muted);font-size:13px;text-align:center}

 /* responsiveness for tiles scrollbars */

 .tiles::-webkit-scrollbar{height:8px}

 .tiles::-webkit-scrollbar-thumb{background:rgba(0,0,0,0.3);border-radius:8px}

 /* small screens adjustments */

 @media (max-width:600px){

  .thread-label{width:70px;font-size:13px}

  .badge{font-size:12px;padding:4px 6px}

 }

</style>

</head>
```

```
<body>
<header>
 <div style="display:flex;flex-direction:column">
  <h1>Real-Time Multi-threaded Application Simulator</h1>
  <div class="meta">Kumar Abhinash (12501247) • Ajay Upadhyay (12528614) • Lovely Professional University</div>
 </div>
</header>

<div class="wrap">
 <!-- LEFT: Controls -->
 <div class="left">
  <div class="card">
   <label>Threading Model</label>
   <div class="row">
    <select id="modelSelect">
     <option>Many-to-One</option>
     <option>One-to-One</option>
     <option>Many-to-Many</option>
    </select>
    <label style="margin-left:6px" class="small">Scheduler</label>
    <select id="schedSelect" style="margin-left:6px">
     <option>RoundRobin</option>
     <option>FCFS</option>
     <option>Priority</option>
    </select>
```

```
    </div>

    <div class="sep"></div>

    <label>Create Thread</label>
    <div class="row" style="gap:6px">
      <input id="burstInput" type="number" placeholder="Burst (ticks)" value="6"
style="width:120px" />
      <input id="prioInput" type="number" placeholder="Priority" value="1"
style="width:90px" />
    </div>
    <div style="display:flex;gap:8px;margin-top:8px">
      <button id="addThreadBtn" class="primary">Add Thread</button>
      <button id="addRandomBtn">Add Random</button>
    </div>

    <div class="sep"></div>

    <label>Synchronization</label>
    <div class="row" style="gap:6px;margin-bottom:6px">
      <input id="sname" placeholder="Semaphore name" value="S1"
style="width:120px" />
      <input id="sinit" type="number" placeholder="Init" value="1"
style="width:70px" />
      <button id="createSema">Create</button>
    </div>
    <div class="row" style="gap:6px">
```

```
      <input id="mname" placeholder="Monitor name" value="M1"
style="width:120px" />
      <button id="createMon">Create Monitor</button>
    </div>

    <div class="sep"></div>

    <label>Simulation Controls</label>
    <div class="row" style="margin-bottom:6px">
      <button id="playBtn" class="primary">Play</button>
      <button id="pauseBtn">Pause</button>
      <button id="stepBtn">Step</button>
      <button id="resetBtn" class="warn">Reset</button>
    </div>
    <div class="row" style="align-items:center;margin-top:8px">
      <label class="small">Quantum (ms)</label>
      <input id="quantum" type="number" value="300" style="width:100px;margin-
left:6px" />
      <label class="small" style="margin-left:8px">Speed</label>
      <input id="speed" type="range" min="80" max="1000" value="300"
style="flex:1;margin-left:6px" />
    </div>

    <div class="sep"></div>

    <label>Presets</label>
    <div style="display:flex;gap:8px;flex-wrap:wrap">
```

```
    <button id="preset1">Many-to-One Demo</button>
    <button id="preset2">Semaphore Pool</button>
    <button id="preset3">Producer/Consumer</button>
   </div>
  </div>

  <div class="card">
   <label>Save / Load Scenario</label>
   <div class="row" style="margin-bottom:8px">
    <button id="exportBtn">Export JSON</button>
    <input id="importFile" type="file" accept=".json" />
   </div>
   <div class="sep"></div>
   <label>Quick Tips</label>
   <ul class="small" style="margin:8px 0 0 18px;padding:0">
    <li>Use Step to debug tick-by-tick</li>
    <li>Use Many-to-One to see process-wide blocking</li>
    <li>Use semaphores to create realistic blocking</li>
   </ul>
  </div>

 </div>

 <!-- CENTER: Visualization -->
 <div class="center">
  <div class="card toolbar">
```

```
    <div style="display:flex;gap:12px;align-items:center;flex:1">
      <div><strong>Simulation</strong></div>
      <div class="muted">Tick: <span id="tickCounter">0</span></div>
      <div class="muted">Threads: <span id="countThreads">0</span></div>
      <div class="muted">Kernel threads: <span id="countKernel">0</span></div>
    </div>
    <div style="display:flex;gap:8px;align-items:center">
      <button id="clearLogs">Clear Logs</button>
    </div>
  </div>

  <div class="card lanes" id="lanesContainer" aria-live="polite">
    <!-- thread rows inserted here -->
    <div id="cpuArea" style="margin-bottom:12px">
      <label class="small" style="margin-bottom:6px;display:block">CPU / Kernel Threads</label>
      <div id="cpuSlots" class="cpu-box"></div>
    </div>

    <div id="threadsList"></div>
  </div>

  <div class="card">
    <label>Gantt (recent ticks)</label>
    <div class="muted" style="font-size:13px;margin-bottom:8px">Colors: 🟢 Running • 🟡 Ready • 🔴 Blocked • ⚪ Terminated</div>
    <div id="logArea" class="log"></div>
```

```
  </div>
 </div>

 <!-- RIGHT: Resources & Info -->
 <div class="right">
  <div class="card">
   <label>Semaphores</label>
   <div id="semaList" class="resource"></div>
   <div class="sep"></div>
   <label>Monitors</label>
   <div id="monList" class="resource"></div>
  </div>

  <div class="card">
   <label>Thread Controls</label>
   <div id="threadControls" class="grid-2"></div>
  </div>

  <div class="card">
   <label>Event Log</label>
   <div id="events" class="log"></div>
  </div>
 </div>
</div>

<div class="footer card" style="margin:12px 16px">
```

```
  <div style="font-weight:700">Made for: Real-Time Multi-threaded Application Simulator</div>

  <div class="muted">Demonstrates thread models, scheduling, semaphores, and monitors — fully client-side.</div>

</div>


<script>

/*

  Real-Time Multi-threaded Application Simulator

  Single-file HTML/CSS/JS educational simulator.

*/


/* ------------------------

  Simulation Data Structures

  ------------------------ */

let nextTid = 1;

let tick = 0;

let running = false;

let timer = null;

let quantum = 300;

let model = 'Many-to-One';

let scheduler = 'RoundRobin';

let kernelThreads = 1; // for mapping pool in Many-to-Many

const threads = []; // user threads

const semaphores = {}; // {name: {value, queue:[]}}

const monitors = {}; // {name: {locked:false, owner:null, waitq:[]}}

const events = [];
```

```
/* Thread object:
 {
  id, state, burstRemaining, priority, history[], waitingFor: {type:'sema'|'monitor', name}
 }
*/

/* ------------------------
  Helpers & UI references ye dom ka conneection banata hai
  ------------------------ */
const lanesEl = document.getElementById('threadsList');
const cpuSlotsEl = document.getElementById('cpuSlots');
const logEl = document.getElementById('logArea');
const eventsEl = document.getElementById('events');

function logEvent(msg){
 const time = tick;
 events.unshift(`[t=${time}] ${msg}`);
 if(events.length>200) events.pop();
 eventsEl.innerHTML = events.map(s=>`<div>${s}</div>`).join('');
}

/* ------------------------
  Simulation Engine
```

```
  ------------------------ */
function makeThread(burst=5, priority=1, name=null){
  const t = {
    id: nextTid++,
    name: name || `T${nextTid-1}`,
    state: 'Ready', // New, Ready, Running, Blocked, Terminated
    burstRemaining: burst,
    priority: priority,
    history: [],
    waitingFor: null,
    kernelSlot: null
  };
  threads.push(t);
  renderUI();
  logEvent(`Created ${t.name} (burst=${burst}, prio=${priority})`);
  return t;
}

function resetAll(){
  nextTid = 1;
  tick = 0;
  running = false;
  clearInterval(timer);
  timer = null;
  threads.length = 0;
  for(const k in semaphores) delete semaphores[k];
```

```
 for(const k in monitors) delete monitors[k];
 events.length = 0;
 renderUI();
 renderResources();
 updateStats();
}

function findReady(){
 return threads.filter(t=>t.state==='Ready');
}

function schedulerPick(){
 const ready = findReady();
 if(ready.length===0) return null;
 if(scheduler === 'FCFS') return ready[0];
 if(scheduler === 'Priority') {
  ready.sort((a,b)=> b.priority - a.priority || a.id - b.id);
  return ready[0];
 }
 // RoundRobin simulation: rotate by last run - we'll pick first ready to be simple
 return ready[0];
}

/* Apply model consequences when blocking occurs */
function applyModelOnBlock(blockingThread){
 if(model === 'Many-to-One'){
```

```
  // simulate entire process blocking: set non-terminated threads to Blocked
  threads.forEach(t=>{
   if(t.state !== 'Terminated'){
    if(t.id === blockingThread.id){
     t.state = 'Blocked';
     t.waitingFor = blockingThread.waitingFor;
    } else {
     if(t.state !== 'Terminated') t.state = 'Blocked';
    }
   }
  });
  logEvent(`(M2O) Process blocked due to ${blockingThread.name} waiting for
${blockingThread.waitingFor.type}:${blockingThread.waitingFor.name}`);
 } else if(model === 'One-to-One'){
  // only thread blocked -> handled elsewhere
 } else if(model === 'Many-to-Many'){
  // blocked thread returns kernel slot (if any)
  if(blockingThread.kernelSlot !== null){
   // free kernel slot
   blockingThread.kernelSlot = null;
   renderCPUSlots();
  }
 }
}

/* Semaphore operations */
```

```
function createSemaphore(name, init){
 if(!name) return;
 semaphores[name] = {value: init|0, queue: []};
 renderResources();
 logEvent(`Semaphore ${name} created (init=${init})`);
}
function sem_wait(name, tid){
 const s = semaphores[name];
 if(!s) return false;
 const t = threads.find(x=>x.id===tid);
 if(!t) return false;
 if(s.value > 0){
  s.value--;
  logEvent(`${t.name} acquired ${name}`);
  return true;
 } else {
  s.queue.push(t.id);
  t.state = 'Blocked';
  t.waitingFor = {type:'sema', name};
  applyModelOnBlock(t);
  logEvent(`${t.name} blocked on ${name}`);
  return false;
 }
}
function sem_signal(name){
 const s = semaphores[name];
```

```
  if(!s) return;
  if(s.queue.length>0){
    const wakeTid = s.queue.shift();
    const t = threads.find(x=>x.id===wakeTid);
    if(t){ t.state = 'Ready'; t.waitingFor = null; logEvent(`${t.name} unblocked from ${name}`); }
  } else {
    s.value++;
  }
  renderResources();
}

/* Monitor operations (simple) */
function createMonitor(name){
  monitors[name] = {locked:false, owner:null, waitq:[]};
  renderResources();
  logEvent(`Monitor ${name} created`);
}
function monitor_enter(name, tid){
  const m = monitors[name];
  const t = threads.find(x=>x.id===tid);
  if(!m || !t) return false;
  if(!m.locked){
    m.locked = true;
    m.owner = tid;
    logEvent(`${t.name} entered monitor ${name}`);
```

```
    return true;
  } else {
    m.waitq.push(tid);
    t.state = 'Blocked';
    t.waitingFor = {type:'monitor', name};
    applyModelOnBlock(t);
    logEvent(`${t.name} waiting to enter monitor ${name}`);
    return false;
  }
}
function monitor_exit(name){
  const m = monitors[name];
  if(!m) return;
  if(m.waitq.length>0){
    const nextTid = m.waitq.shift();
    m.owner = nextTid;
    const t = threads.find(x=>x.id===nextTid);
    if(t){ t.state='Ready'; t.waitingFor=null; logEvent(`${t.name} acquired monitor
${name}`) }
  } else {
    m.locked = false;
    m.owner = null;
  }
  renderResources();
}
```

```
/* ------------------------
  Tick / run loop ye pura simulator ka heart hai
   ye 300ms me chalti hai
  ------------------------ */
function tickOnce(){
 tick++;
 document.getElementById('tickCounter').textContent = tick;
 // choose running threads per kernel capacity (for One-to-One maybe each user thread can run -> but sim limited)
 // We'll maintain kernelSlots count:
 let activeKernel = (model==='One-to-One') ? threads.filter(t=>t.state==='Ready' || t.state==='Running').length : (model==='Many-to-Many' ? Math.max(1, kernelThreads) : 1);
 // ensure activeKernel limit reasonable
 activeKernel = Math.min(activeKernel, 10);

 // if model is Many-to-One, only one can be running
 if(model === 'Many-to-One') activeKernel = 1;

 // get eligible threads for running (Ready)
 let runCandidates = threads.filter(t=>t.state==='Running' || t.state==='Ready');

 // If nothing to run, just add histories
 if(runCandidates.length === 0){
  threads.forEach(t=>{
   t.history.push(t.state);
   if(t.history.length>120) t.history.shift();
```

```
  });
  renderUI();
  return;
 }

 // assign running slots (simple):
 // 1) If someone is already Running, let them finish quantum (simulate by making them run one tick)
 // 2) Else pick by scheduler
 let runningNow = threads.filter(t=>t.state==='Running');
 if(runningNow.length < activeKernel){
  // fill slots
  while(runningNow.length < activeKernel){
   const pick = schedulerPick();
   if(!pick) break;
   pick.state = 'Running';
   runningNow.push(pick);
  }
 }

 // Execute work for running threads (1 unit)
 for(const t of threads){
  // history before update
  t.history.push(t.state);
  if(t.history.length>120) t.history.shift();
 }
```

```
 // copy running threads and process
 const toProcess = threads.filter(t=>t.state==='Running');
 for(const t of toProcess){
  // simulate performing 1 unit
  t.burstRemaining--;
  // random demo: some running threads attempt to acquire semaphores/monitors (disabled by default in stable)
  // If we want deterministic, comment randomness.
  // If burst completes
  if(t.burstRemaining <= 0){
   t.state = 'Terminated';
   t.waitingFor = null;
   logEvent(`${t.name} terminated`);
  } else {
   // For fairness/slice: put back to Ready for RR
   if(scheduler === 'RoundRobin'){
    t.state = 'Ready';
   } else {
    // FCFS or Priority will keep them Running until others
    t.state = 'Ready';
   }
  }
 }

 // occasionally auto-signal semaphores to show progress (demo)
```

```
 for(const name in semaphores){
  const s = semaphores[name];
  // if a queue exists and value==0, try to release occasionally
  if(s.queue.length > 0 && Math.random() < 0.05){
   sem_signal(name);
  }
 }
 // also try to release monitors occasionally for demo
 for(const name in monitors){
  const m = monitors[name];
  if(m.locked && Math.random() < 0.04){
   monitor_exit(name);
  }
 }

 renderUI();
 updateStats();
}

/* ------------------------
  UI Rendering ye backend data ko screen par dikhata hai
  ------------------------ */
function renderUI(){
 // CPU slots and threads
 renderCPUSlots();
 // thread list
```

```
 lanesEl.innerHTML = '';
 // threads sorted by id
 const sorted = threads.slice().sort((a,b)=>a.id - b.id);
 sorted.forEach(t=>{
  const row = document.createElement('div');
  row.className = 'thread-row';
  row.style.background = 'linear-gradient(90deg, rgba(255,255,255,0.01),
rgba(255,255,255,0.005))';
  row.innerHTML = `
   <div class="thread-label">${t.name}</div>
   <div style="flex:1" title="State: ${t.state}; Burst: ${t.burstRemaining}; Prio:
${t.priority}">
    <div class="tiles" id="tiles-${t.id}"></div>
   </div>
   <div style="width:110px;text-align:right">
    <div class="small muted">State: ${t.state}</div>
    <div class="small muted">Burst: ${t.burstRemaining}</div>
   </div>
  `;
  lanesEl.appendChild(row);
  const tiles = document.getElementById(`tiles-${t.id}`);
  // render last 40 history entries
  const last = t.history.slice(-40);
  for(const s of last){
   const d = document.createElement('div');
   d.className = `tile state-${s}`;
   d.title = s;
```

```
    tiles.appendChild(d);
  }
  // ensure trailing filler to show future ticks
  for(let i=0;i<8;i++){
    const d = document.createElement('div');
    d.className = `tile`;
    d.style.opacity = 0.03;
    tiles.appendChild(d);
  }
 });

 // thread controls (per thread actions)
 renderThreadControls();
 renderResources();
 renderLogs();
}

function renderCPUSlots(){
 cpuSlotsEl.innerHTML = '';
 let slots = (model === 'Many-to-One') ? 1 : (model === 'One-to-One' ? Math.max(1,
Math.min(threads.length, 8)) : Math.max(1, kernelThreads));
 document.getElementById('countKernel').textContent = slots;
 for(let i=0;i<slots;i++){
  const slot = document.createElement('div');
  slot.className = 'cpu-slot';
  // show which thread is running on this slot (naive mapping)
```

```
  const running = threads.find(t=>t.state==='Running');
  if(running){
    slot.innerHTML = `<div style="font-
weight:700;color:#062122">${running.name}</div><div
class="muted">Running</div>`;
  } else {
    slot.innerHTML = `<div class="muted">Idle</div>`;
  }
  cpuSlotsEl.appendChild(slot);
 }
}

function renderResources(){
 // semaphores
 const sl = document.getElementById('semaList');
 sl.innerHTML = '';
 for(const name in semaphores){
  const s = semaphores[name];
  const div = document.createElement('div');
  div.innerHTML = `<div><strong>${name}</strong><div class="small
muted">val=${s.value} |
queue=[${s.queue.map(id=>threads.find(t=>t.id===id)?.name||id).join(',
')}]</div></div><div class="badge">S</div>`;
  sl.appendChild(div);
 }
 // monitors
 const ml = document.getElementById('monList');
 ml.innerHTML = '';
```

```
 for(const name in monitors){
   const m = monitors[name];
   const ownerName = m.owner ? (threads.find(t=>t.id===m.owner)?.name || m.owner) : 'none';
   const div = document.createElement('div');
   div.innerHTML = `<div><strong>${name}</strong><div class="small muted">locked=${m.locked} owner=${ownerName} wait=[${m.waitq.map(id=>threads.find(t=>t.id===id)?.name||id)}]</div></div><div class="badge">M</div>`;
   ml.appendChild(div);
 }
}

function renderThreadControls(){
 const box = document.getElementById('threadControls');
 box.innerHTML = '';
 threads.forEach(t=>{
   const d = document.createElement('div');
   d.style.display = 'flex';
   d.style.flexDirection = 'column';
   d.style.gap = '6px';
   d.innerHTML = `
    <div style="display:flex;gap:6px;align-items:center;justify-content:space-between">
      <div style="font-weight:700">${t.name}</div>
      <div style="font-size:12px;color:var(--muted)">${t.state}</div>
    </div>
    <div style="display:flex;gap:6px">
```

```
      <button data-tid="${t.id}" class="btn-acquire">Acquire</button>
      <button data-tid="${t.id}" class="btn-release">Release</button>
      <button data-tid="${t.id}" class="btn-kill"
style="background:#ff6b6b;color:white">Kill</button>
    </div>
  `;
  box.appendChild(d);
 });
 // attach listeners
 document.querySelectorAll('.btn-acquire').forEach(b=>{
  b.onclick = ()=>{
   const tid = Number(b.getAttribute('data-tid'));
   // open a small prompt to choose semaphore or monitor
   const choice = prompt('Acquire: type "S:name" for semaphore or "M:name" for
monitor. Example: S:S1 or M:M1');
   if(!choice) return;
   if(choice.startsWith('S:')){
    const name = choice.slice(2);
    sem_wait(name, tid);
   } else if(choice.startsWith('M:')){
    const name = choice.slice(2);
    monitor_enter(name, tid);
   }
   renderUI();
  };
 });
 document.querySelectorAll('.btn-release').forEach(b=>{
```

```
  b.onclick = ()=>{
    const tid = Number(b.getAttribute('data-tid'));
    const choice = prompt('Release: type "S:name" to signal semaphore or "M:name" to exit monitor. Example: S:S1 or M:M1');
    if(!choice) return;
    if(choice.startsWith('S:')){
      const name = choice.slice(2);
      sem_signal(name);
    } else if(choice.startsWith('M:')){
      const name = choice.slice(2);
      monitor_exit(name);
    }
    renderUI();
  };
});
document.querySelectorAll('.btn-kill').forEach(b=>{
  b.onclick = ()=>{
    const tid = Number(b.getAttribute('data-tid'));
    const t = threads.find(x=>x.id===tid);
    if(!t) return;
    t.state = 'Terminated';
    logEvent(`${t.name} killed`);
    renderUI();
  };
});
}
```

```
/* render logs */
function renderLogs(){
 logEl.innerHTML = events.slice(0,200).map(s=>`<div>${s}</div>`).join('');
}

/* update stats on header */
function updateStats(){
 document.getElementById('countThreads').textContent = threads.length;
}

/* ------------------------
  UI events binding  buttons backend function ko call karte hai
  ------------------------ */
document.getElementById('modelSelect').onchange = (e)=>{
 model = e.target.value;
 logEvent(`Model set to ${model}`);
 renderUI();
};
document.getElementById('schedSelect').onchange = (e)=>{
 scheduler = e.target.value;
 logEvent(`Scheduler set to ${scheduler}`);
};

document.getElementById('addThreadBtn').onclick = ()=>{
```

```
  const burst = Math.max(1, Number(document.getElementById('burstInput').value)
|| 5);
  const pr = Math.max(0, Number(document.getElementById('prioInput').value) || 1);
  makeThread(burst, pr);
  updateStats();
};
document.getElementById('addRandomBtn').onclick = ()=>{
  const burst = Math.floor(Math.random()*8)+2;
  const pr = Math.floor(Math.random()*3)+1;
  makeThread(burst, pr);
  updateStats();
};

document.getElementById('createSema').onclick = ()=>{
  const name = document.getElementById('sname').value.trim();
  const init = Math.max(0, Number(document.getElementById('sinit').value) || 0);
  if(!name){ alert('Enter name'); return; }
  createSemaphore(name, init);
};

document.getElementById('createMon').onclick = ()=>{
  const name = document.getElementById('mname').value.trim();
  if(!name){ alert('Enter name'); return; }
  createMonitor(name);
};
```

```
document.getElementById('playBtn').onclick = ()=>{
 if(running) return;
 running = true;
 quantum = Number(document.getElementById('quantum').value) || 300;
 const speed = Number(document.getElementById('speed').value) || 300;
 clearInterval(timer);
 timer = setInterval(tickOnce, speed);
 logEvent('Simulation started');
};
document.getElementById('pauseBtn').onclick = ()=>{
 running = false;
 clearInterval(timer);
 timer = null;
 logEvent('Simulation paused');
};
document.getElementById('stepBtn').onclick = ()=>{
 tickOnce();
};
document.getElementById('resetBtn').onclick = ()=>{
 if(confirm('Reset simulation?')) resetAll();
};
document.getElementById('clearLogs').onclick = ()=>{
 events.length = 0;
 renderLogs();
};
```

```
document.getElementById('preset1').onclick = ()=>{
 resetAll();
 model = 'Many-to-One';
 document.getElementById('modelSelect').value = model;
 scheduler = 'RoundRobin';
 document.getElementById('schedSelect').value = scheduler;
 createSemaphore('S1', 0);
 makeThread(6,1,'T1');
 makeThread(8,1,'T2');
 makeThread(4,1,'T3');
 logEvent('Loaded Many-to-One Demo');
 renderUI();
};
document.getElementById('preset2').onclick = ()=>{
 resetAll();
 model = 'Many-to-Many';
 document.getElementById('modelSelect').value = model;
 scheduler = 'FCFS';
 document.getElementById('schedSelect').value = scheduler;
 kernelThreads = 3;
 createSemaphore('SPOOL', 2);
 for(let i=0;i<6;i++) makeThread(6,1);
 logEvent('Loaded Semaphore Pool Demo');
 renderUI();
};
document.getElementById('preset3').onclick = ()=>{
```

```
  resetAll();
  model = 'Many-to-Many';
  document.getElementById('modelSelect').value = model;
  scheduler = 'RoundRobin';
  createMonitor('BUF');
  // producer/consumer: producers will "enter" monitor manually via controls or we simulate some events
  makeThread(6,1,'Producer1');
  makeThread(6,1,'Consumer1');
  makeThread(6,1,'Producer2');
  logEvent('Loaded Producer/Consumer Demo (use controls to Acquire/Release monitor BUF)');
  renderUI();
};

/* Save / Load */
document.getElementById('exportBtn').onclick = ()=>{
  const data = {threads, semaphores, monitors, model, scheduler, tick};
  const blob = new Blob([JSON.stringify(data,null,2)],{type:'application/json'});
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a'); a.href = url; a.download = 'scenario.json'; a.click();
  URL.revokeObjectURL(url);
};
document.getElementById('importFile').onchange = (e)=>{
  const f = e.target.files[0];
  if(!f) return;
```

```
 const reader = new FileReader();
 reader.onload = (ev)=>{
  try{
   const data = JSON.parse(ev.target.result);
   resetAll();
   // restore basic structure
   if(data.semaphores) { for(const k in data.semaphores)
semaphores[k]=data.semaphores[k]; }
   if(data.monitors) { for(const k in data.monitors) monitors[k]=data.monitors[k]; }
   if(Array.isArray(data.threads)){
    data.threads.forEach(t=> {
     // reconstruct thread shallowly
     const nt = makeThread(t.burstRemaining||5,t.priority||1,t.name||null);
     nt.state = t.state || 'Ready';
     nt.history = t.history || [];
    });
   }
   if(data.model) { model = data.model;
document.getElementById('modelSelect').value = model; }
   if(data.scheduler) { scheduler = data.scheduler;
document.getElementById('schedSelect').value = scheduler; }
   logEvent('Scenario imported');
   renderUI();
  }catch(err){ alert('Invalid JSON'); }
 };
 reader.readAsText(f);
};
```

```
/* initialize UI */

renderUI();

updateStats();


</script>

</body>


<footer>

  <p>© 2025 Real-Time Multi-threaded Application Simulator<br>

  Developed by: Kumar Abhinash & Ajay Upadhyay — Lovely Professional University</p>

</footer>


</html>
```

The above code represents the final implementation of the Real-Time Multi-threaded Application Simulator. Through this project, all core concepts—threading models, scheduling, synchronization, and visualization—have been practically demonstrated.
Thank you for evaluating my work and giving me the opportunity to learn and build this project.

——— End of Project Documentation and Source Code ——

Thank you for reviewing my project.

END…….

THANK YOU

COLLEGE REPORT