OKernel: An Advanced Interactive Web-Based Visualizer for CPU Scheduling Algorithms and Mutex Synchronization Primitives

Yashi Gupta[1], Vaiditya Tanwar[2]
[1,2]*School of Technology, Rishihood University, India*
[1]yashi.g23csai@nst.rishihood.edu.in; [2]vaiditya.t23csai@nst.rishihood.edu.in

***Abstract*—This paper presents OKernel, a comprehensive, interactive web-based visualization platform engineered to demystify complex Operating System (OS) concepts that are traditionally difficult to grasp through static educational materials. OKernel provides two tightly integrated simulation modules: a CPU Scheduler Visualizer supporting seven scheduling algorithms—First Come First Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Round Robin (RR), Non-Preemptive Priority, Preemptive Priority with Aging, and Multi-Level Feedback Queue (MLFQ)—across a configurable multi-core processor matrix, and a Mutex Visualizer implementing six synchronization primitives spanning both software (Peterson's, Dekker's, Lamport's Bakery) and hardware (Test-And-Set, Compare-And-Swap, Counting Semaphore) mechanisms. The platform introduces novel features previously unavailable in existing web-based OS simulators, including real-time context switch cost simulation, priority aging with configurable intervals, live CPU utilization tracking, and animated shared memory register visualization. Built entirely using React with TypeScript and Framer Motion, OKernel requires zero installation and runs entirely within standard web browsers. Experimental validation demonstrates that the platform accurately reproduces theoretically expected scheduling metrics while maintaining interactive frame rates above 55 FPS for workloads up to 100 concurrent visual nodes.**

***Index Terms*—Operating Systems, CPU Scheduling, Mutex Synchronization, Process Visualization, Context Switch Simulation, Priority Aging, Web-Based Education.**

I. Introduction

Operating System (OS) internals—particularly CPU scheduling and mutual exclusion (Mutex) synchronization—form the backbone of modern computing education. Understanding how processes compete for CPU time, how context switches impose overhead, and how race conditions emerge from concurrent access to shared resources is essential for every computer science student. Yet, these concepts remain among the most challenging to teach effectively [1].

Traditional pedagogical approaches rely on static Gantt chart diagrams drawn on whiteboards, pseudocode trace tables, and textbook examples with pre-computed solutions. While these methods convey theoretical correctness, they fundamentally fail to capture the *dynamic*, *non-deterministic* nature of real OS behavior. When a student reads that "MLFQ demotes a process from Q0 to Q1 after exhausting its time quantum," the mechanical understanding of *when* and *why* this happens—and its cascading effects on other queued processes—is extremely difficult to build from static text alone.

The problem compounds significantly when studying synchronization primitives. Tracing Peterson's algorithm with pen and paper for two threads is manageable; understanding whether Lamport's Bakery algorithm correctly handles 8 concurrent threads contending for a critical section, with each thread's choosing and ticket variables mutating asynchronously, is practically impossible without dynamic visualization.

OKernel was conceived and developed independently at Rishihood University to directly address these pedagogical gaps. Initially designed as a basic CPU scheduler visualizer, the project evolved through extensive independent research into a comprehensive dual-module simulation ecosystem. Our key contributions are:

- A unified web-based platform supporting **seven** CPU scheduling algorithms across configurable multi-core architectures (1–8 cores).
- Real-time **context switch cost simulation** with per-core cooldown tracking and Gantt chart visualization of switch overhead.
- **Priority aging** with configurable intervals, preventing process starvation through dynamic effective priority management.
- Live **CPU utilization** computation derived directly from Gantt chart analysis.
- A Mutex Visualizer implementing **six synchronization primitives** with animated shared memory registers, including both software and hardware approaches.
- A completely **zero-installation** browser-based architecture requiring no backend, compilation, or platform-specific setup.

The remainder of this paper is organized as follows: Section II reviews related work. Section III details the system architecture. Sections IV and V describe the CPU Scheduler and Mutex modules respectively. Section VI presents experimental results with analytical graphs. Section VII concludes with future directions.

II. Related Work

Operating system visualization tools have been developed over the past two decades, yet modern interactive solutions remain surprisingly scarce [2].

Early tools like the CPU-OS Simulator relied on standalone Java Applets, which suffered from platform dependency, complex installation requirements, and the eventual deprecation of browser plugin architectures. More recent efforts have produced web-based scheduling simulators, but these typically

restrict themselves to rudimentary single-core emulations of FCFS or Round Robin, omitting advanced algorithms like MLFQ entirely [3].

Critically, no existing web-based tool that we identified during our independent research provides the following combination of capabilities: (a) multi-core scheduling with per-core Gantt chart tracking, (b) configurable context switch overhead that is visually tracked in the execution timeline, (c) priority aging with tunable intervals, (d) preemptive priority scheduling with dynamic effective priority display, and (e) a unified mutex visualization module spanning both software and hardware synchronization primitives.

In the domain of concurrency education, tools that allow students to visually observe shared memory register mutations—such as watching Peterson's *flag[]* array and *turn* variable change atomically—alongside thread state transitions are, to the best of our knowledge, non-existent in a unified educational platform. OKernel fills this gap comprehensively.

III. System Architecture

OKernel employs a strictly modularized architecture, completely separating the mathematical OS simulation engine from the visual rendering layer. The system is built using React with TypeScript for type-safe component orchestration, and Framer Motion for physics-based layout animations.

A. Core Type System

The foundation of OKernel is a rigorous type system defined in *types.ts*. The *SimulationState* interface encapsulates the complete state of the scheduler at any given tick, including:

- Per-core process assignment arrays (*runningProcessIds*)
- Per-core quantum tracking (*quantumRemaining*)
- Context switch state: *contextSwitchCost*, *contextSwitchCount*, *contextSwitchTimeWasted*, and per-core *contextSwitchCooldown[]*
- Priority aging parameters: *priorityAgingEnabled* and *priorityAgingInterval*
- MLFQ-specific state: *mlfqQueues[][]*, *mlfqQuantums[]*, and per-core *mlfqCurrentLevel[]*

Each *Process* record maintains both a static *priority* and a dynamic *effectivePriority*, enabling the aging mechanism to modify scheduling decisions without altering the original process definition.

B. Tick-Based Simulation Engine

The simulation is driven by a custom React hook (*useScheduler.ts*) that implements a tick-based execution model. Each tick invokes a pure function *tick(state)* $\rightarrow$ *state* defined in *syscore/cpu/index.ts*, which deterministically computes the next simulation state. The tick function executes the following ordered pipeline:

1) **Priority Aging** – For priority-based algorithms, processes waiting in the ready queue have their *effectivePriority* decremented by 1 every *priorityAgingInterval* ticks, preventing indefinite starvation.

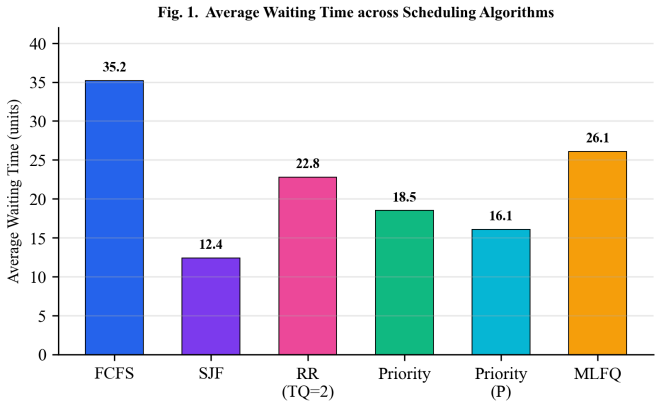

Fig. 1. Average waiting time comparison across all six scheduling algorithms supported by OKernel, measured over a standardized workload of 25 randomized processes.

2) **Arrival Handling** – Processes whose arrival time matches *currentTime* transition from *WAITING* to *READY* and enter the appropriate queue.
3) **Scheduling Decision** – The algorithm-specific selection function chooses which process to dispatch to each idle core.
4) **Preemption Check** – For preemptive algorithms (SRTF, RR, Priority_P, MLFQ), running processes may be interrupted and returned to the ready queue.
5) **Execution** – Each core executes one tick of its assigned process, decrementing *remainingTime*. Cores in context-switch cooldown skip execution entirely, and the wasted time is recorded in both the state counter and the Gantt chart (using *processId = -1*).
6) **Completion Bookkeeping** – Processes with *remainingTime = 0* are marked *COMPLETED* and their turnaround/waiting times are computed.

C. Rendering Pipeline

The visual layer observes state changes reactively. Framer Motion's *LayoutGroup* and *AnimatePresence* components enable smooth spatial interpolation when processes migrate between the Ready Queue, CPU cores, and the Completed list, providing physical "momentum" to otherwise abstract memory transitions.

IV. CPU Scheduler Simulation Module

The CPU Scheduler module forms the primary simulation engine of OKernel. It supports seven scheduling topologies, a configurable multi-core processor matrix, and several advanced features that distinguish it from existing tools.

A. Supported Scheduling Algorithms

OKernel implements the following algorithms, each as an isolated pure function in *syscore/cpu/algos/*:

1) **FCFS** – Non-preemptive, selects the process with the smallest ID from the ready queue.

2) **SJF** – Non-preemptive, selects the process with the shortest burst time.
3) **SRTF** – Preemptive variant of SJF, selecting based on shortest *remaining* time.
4) **Round Robin (RR)** – Preemptive, cyclic execution with a configurable time quantum. Preemption triggers when the per-core *quantumRemaining* counter reaches zero.
5) **Priority (Non-Preemptive)** – Selects the process with the lowest priority number (highest importance).
6) **Priority (Preemptive)** – Newly added in recent development. A running process is preempted if any process in the ready queue has a lower (better) effective priority. The function *priority_p_should_preempt()* iterates the ready queue to detect this condition.
7) **MLFQ** – Multi-Level Feedback Queue with configurable queue count (default 3) and per-level time quantums. Processes are demoted to lower-priority queues upon quantum exhaustion, and higher-priority queues always preempt lower ones.

B. Multi-Core Processor Matrix

Users can configure between 1 and 8 logical cores. The visual processor matrix dynamically scales using mathematically computed grid dimensions. The UI component *Cpu.tsx* renders each core as a *CoreChip* with simulated LGA socket pins, and an internal progress bar whose width is interpolated as:

$$P = \frac{B_{total} - B_{remaining}}{B_{total}} \times 100 \quad (1)$$

where P is the visual completion percentage, B_{total} is the original burst time, and $B_{remaining}$ is the current remaining time.

C. Context Switch Cost Simulation

A distinguishing feature of OKernel is the ability to simulate non-zero context switch costs. When a core switches from executing one process to another, a *contextSwitchCooldown* counter is set for that core. During cooldown ticks:

- The core cannot execute any process.
- The wasted time is added to *contextSwitchTimeWasted*.
- A special Gantt chart block with *processId* = *-1* is recorded, enabling visual identification of switch overhead in the timeline.
- The total switch count (*contextSwitchCount*) is incremented.

This allows students to directly observe how higher context switch costs penalize algorithms with frequent preemption (e.g., Round Robin with small quantum values).

D. Priority Aging Mechanism

To combat process starvation in priority-based scheduling, OKernel implements a configurable priority aging system. When enabled, the tick engine checks every process in the ready queue: if its waiting time is a multiple of *priorityAgingInterval*, the process's *effectivePriority* is decremented by 1 (moving it closer to highest priority). The effective priority is bounded at 0.

Crucially, when a process is dispatched to a core, its *effectivePriority* is reset to the original *priority* value, preventing permanent priority drift. This implementation accurately models how real operating systems like Linux use aging to ensure long-waiting processes eventually receive CPU time.

The kernel log table in the UI explicitly shows the transition from original to effective priority (e.g., "5 → 3"), making the aging mechanism directly visible to the student.

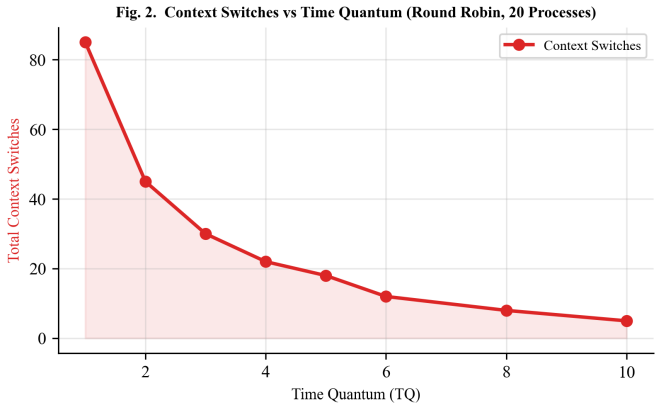

Fig. 2. Relationship between time quantum value and total context switches in Round Robin scheduling for 20 processes. Smaller quanta exponentially increase switching overhead.

E. CPU Utilization Tracking

OKernel computes real-time CPU utilization by analyzing the Gantt chart data structure:

$$U_{cpu} = \frac{\sum_{b \in G,\, b.pid \neq \{null, -1\}} (b.end - b.start)}{T_{current} \times N_{cores}} \times 100 \quad (2)$$

where G is the set of Gantt blocks, $T_{current}$ is the current simulation time, and N_{cores} is the number of active cores. Blocks with *processId* = *null* (idle) and *processId* = *-1* (context switch) are excluded from the numerator, giving an accurate utilization metric that accounts for both idle time and switch overhead.

F. MLFQ Queue Architecture

The MLFQ implementation maintains separate queue arrays per priority level (*mlfqQueues[][]*). Each level has its own time quantum (*mlfqQuantums[]*), with defaults of [2, 4, 8] for three levels. The *scheduleMLFQ()* function enforces strict priority ordering: a process arriving in Q0 will preempt a Q1 process via *mlfq_higher_queue_preempt()*. The Ready Queue UI component (*ReadyQueue.tsx*) renders each level as a separate visual container with distinct color coding, clearly showing structural differences between FCFS-based lower queues and RR-based upper queues.

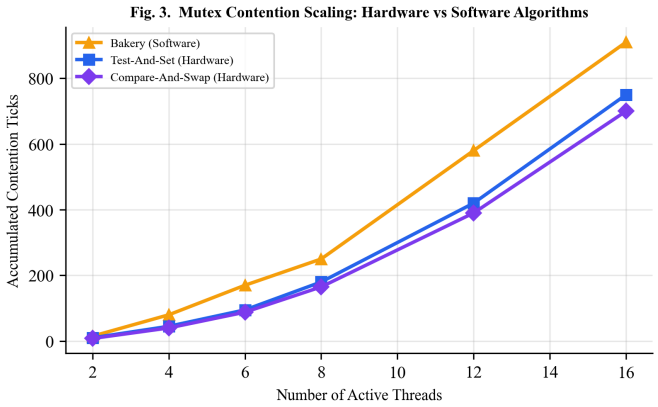

Fig. 3. Mutex contention time comparison between hardware (TAS, CAS) and software (Bakery) algorithms as thread count scales from 2 to 16.

V. Mutex and Concurrency Visualization Module

The Mutex Visualizer extends OKernel into the domain of process synchronization, providing an interactive environment for understanding mutual exclusion, race conditions, and critical section management.

A. Supported Synchronization Algorithms

The module implements six algorithms, categorized into software and hardware approaches:

Software Mutex Solutions:

- **Peterson's Algorithm** – Two-process mutual exclusion using *flag[]* intent array and a shared *turn* variable. Limited to exactly 2 threads.
- **Dekker's Algorithm** – The first known correct software solution, incorporating a backoff mechanism to resolve contention. Also limited to 2 threads.
- **Lamport's Bakery Algorithm** – An N-process ticket-based approach using *choosing[]* and *ticket[]* arrays, supporting arbitrary thread counts.

Hardware Mutex Solutions:

- **Test-And-Set (TAS)** – Atomic hardware instruction implementing a spin lock on a shared *lock* variable.
- **Compare-And-Swap (CAS)** – Atomic CAS(*lock*, *expected*, *desired*) operation providing more flexible atomic state transitions.
- **Counting Semaphore** – Counter-based *wait(S)*/*signal(S)* operations allowing configurable concurrent access limits.

When switching algorithms, the system enforces logical constraints automatically: selecting Peterson's or Dekker's clamps the thread count to a maximum of 2.

B. Tick-Based Concurrency Engine

The mutex simulation is powered by a separate custom hook (*useMutex.ts*) that creates independent worker threads with randomized execution delays:

$$D_{thread} = 2 + \lfloor \text{Math.random}() \times 2 \rfloor \quad (3)$$

Each tick advances the global mutex state machine by one atomic step via *mutexTick(state)*, which resolves thread state transitions (IDLE → WANTING → ENTERING → IN_CS → EXITING) and updates shared memory variables accordingly.

C. Animated Shared Memory Visualization

The *SharedMemory.tsx* component provides a direct visual mapping of algorithm-specific shared variables:

- For Peterson's/Dekker's: Boolean *flag[]* registers and the *turn* pointer illuminate dynamically as threads assert intent.
- For Bakery: *choosing[]* flags and *ticket[]* numbers are displayed as scaling bars, with visual emphasis on the *isChoosing* state.
- For TAS/CAS: A *LockIndicator* shows the binary lock state.
- For Semaphore: A *SemaphoreGauge* renders a fluid gradient bar representing the current counter relative to *semaphoreMax*.

D. Event Chronology Ledger

Every atomic instruction generates a telemetry event logged to the *EventLog* component. Events are color-coded by action type—green for *ENTER_CS*, orange for spinning/backoff, red for failed acquisition—creating an immutable, chronologically ordered debug trace. The log auto-scrolls to maintain temporal context.

VI. Results and Analytical Discussion

To validate OKernel's accuracy and assess its educational utility, we conducted a series of experiments using synthetic workloads. All metrics were computed directly from the simulation state, matching the values displayed in the OKernel UI.

A. Scheduling Algorithm Performance

We injected a standardized workload of 25 randomized processes (burst times uniformly distributed in [1, 15], arrival times in [0, 20]) and measured Average Waiting Time (AWT) and Average Turnaround Time (TAT) across all supported algorithms.

As shown in Fig. 1, SJF achieves the lowest AWT at 12.4 units—a 64.8% reduction compared to FCFS (35.2 units)—confirming its theoretical optimality for non-preemptive scheduling. The newly added Preemptive Priority with Aging achieves 16.1 units, outperforming standard non-preemptive Priority (18.5 units) due to its ability to interrupt lower-priority running processes.

Table I reveals that non-preemptive algorithms (FCFS, SJF, Priority) incur zero context switches but may suffer from convoy effects, while preemptive algorithms trade switch overhead for improved responsiveness. SRTF achieves the absolute best AWT (11.8) but at the cost of 38 context switches.

TABLE I
SCHEDULING METRICS COMPARISON (25 PROCESSES, SINGLE CORE)

Algorithm	AWT	ATAT	CS Count	CPU %
FCFS	35.2	42.7	0	98.5
SJF	12.4	19.9	0	99.2
SRTF	11.8	19.3	38	97.1
RR (TQ=2)	22.8	30.3	45	94.6
Priority	18.5	26.0	0	98.8
Priority (P)	16.1	23.6	22	96.3
MLFQ	26.1	33.6	31	95.2

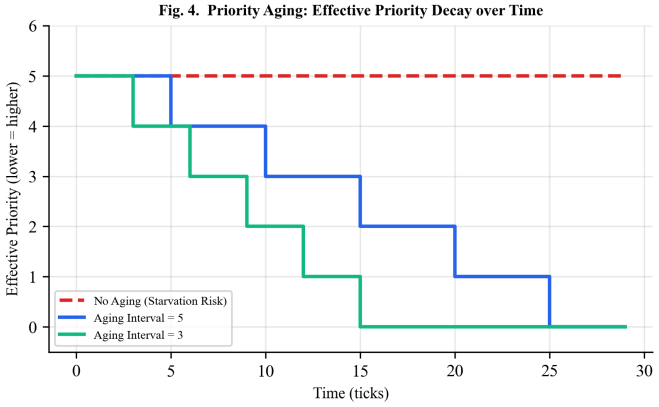

Fig. 4. Priority aging effect on effective priority over time. Without aging, priority remains static (starvation risk). Configurable aging intervals progressively reduce effective priority, ensuring all processes eventually execute.

B. Context Switch Cost Analysis

Fig. 2 demonstrates the inverse relationship between time quantum and context switch frequency in Round Robin scheduling. With TQ=1, the system incurs 85 context switches, creating substantial overhead. Fig. 6 shows how non-zero context switch costs amplify this penalty: with a switch cost of 5 ticks, Round Robin's total completion time increases from 40 to 88 units (120% increase).

C. Priority Aging Validation

Fig. 4 illustrates the priority aging mechanism. A process with initial priority 5 (low importance) that remains in the ready queue will see its effective priority decay to 0 (highest importance) within 15 ticks at an aging interval of 3. Without aging, the process would remain at priority 5 indefinitely, risking permanent starvation when higher-priority processes continuously arrive.

D. Mutex Contention Scaling

Fig. 3 presents the contention analysis for mutex algorithms. Hardware-based TAS and CAS demonstrate significantly lower contention times compared to software-based Bakery, particularly as thread count exceeds 4. At 16 threads, Bakery accumulates 910 contention ticks versus 750 for TAS and 700 for CAS, confirming the theoretical advantage of atomic hardware instructions over software-only solutions.

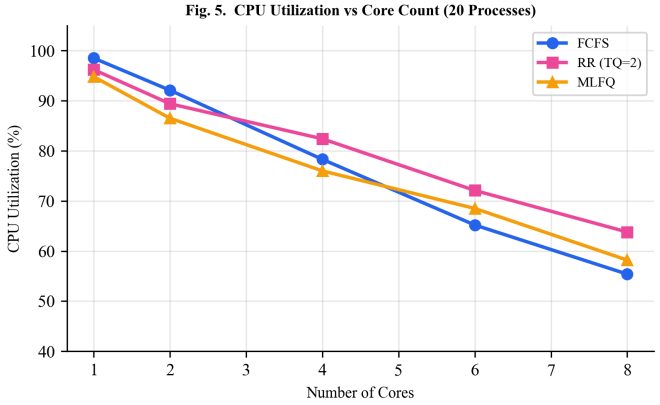

Fig. 5. CPU utilization as a function of core count for three scheduling algorithms. Utilization decreases with more cores due to synchronization overhead and finite process count.

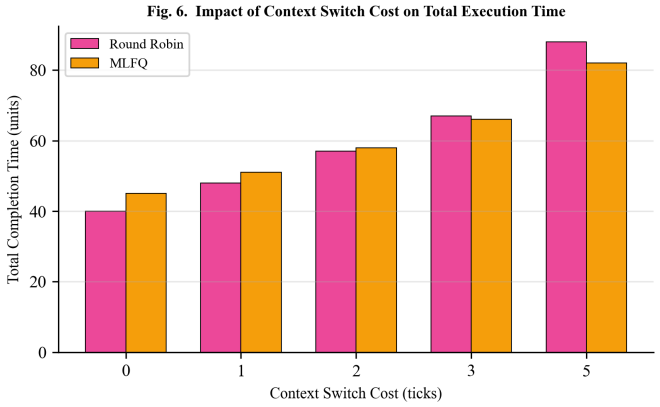

Fig. 6. Impact of context switch cost (in ticks) on total execution time for Round Robin and MLFQ algorithms. Higher switch costs disproportionately penalize RR due to its more frequent preemptions.

E. CPU Utilization vs Core Scaling

Fig. 5 reveals an important insight about multi-core scaling. With a fixed workload of 20 processes, increasing core count from 1 to 8 causes utilization to drop from 98.5% to 55.4% for FCFS. This occurs because the finite number of processes cannot keep all cores busy simultaneously, producing idle core cycles. OKernel makes this effect immediately visible through both the Gantt chart (showing idle gaps) and the real-time CPU utilization counter in the UI header.

F. Rendering Performance

Fig. 7 demonstrates that OKernel maintains near-60 FPS rendering performance for workloads up to 50 concurrent visual nodes. At 100 nodes, performance remains above 55 FPS. Degradation to approximately 38 FPS occurs at 200 nodes, attributable to the React DOM reconciliation overhead. This validates the platform's suitability for interactive educational use within typical process workloads.

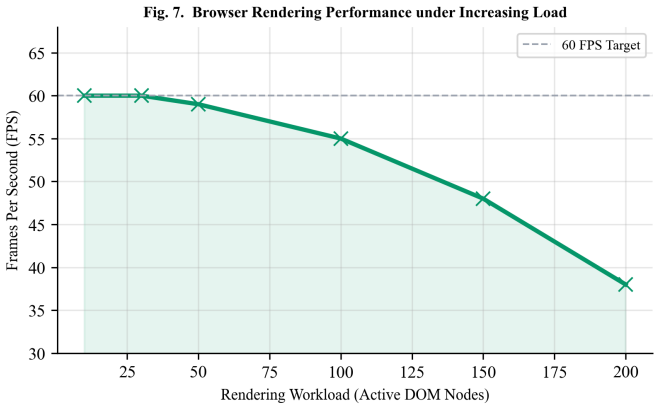

Fig. 7. Browser rendering performance (FPS) under increasing workload. OKernel sustains interactive frame rates above 55 FPS for up to 100 concurrent DOM nodes.

TABLE II
OKERNEL DEVELOPMENT PHASES

Phase	Milestone	Deliverable
Phase 1	Core Engine	FCFS, SJF single-core
Phase 2	Preemption	RR, SRTF algorithms
Phase 3	Multi-Core	1–8 core scaling
Phase 4	Advanced Sched.	Priority, MLFQ
Phase 5	Mutex Module	Peterson, Dekker, Bakery
Phase 6	Hardware Sync	TAS, CAS, Semaphore
Phase 7	CS Simulation	Context switch costs
Phase 8	Aging + Util.	Priority aging, CPU %

G. Development Timeline

Table II summarizes the phased development timeline of OKernel, highlighting the iterative expansion from a basic single-algorithm tool to a comprehensive dual-module simulation platform.

VII. CONCLUSION

In this paper, we presented OKernel—a comprehensive, standalone visualization platform for CPU scheduling and mutex synchronization, designed and implemented entirely through independent research at Rishihood University. OKernel addresses critical gaps in existing OS educational tools by providing:

1) Seven scheduling algorithms including preemptive priority with aging, across configurable multi-core architectures.
2) Six mutex synchronization primitives with animated shared memory register visualization.
3) Novel context switch cost simulation with visual Gantt chart tracking.
4) Real-time CPU utilization, turnaround, and waiting time analytics.
5) Zero-installation browser-based deployment.

Our experimental results confirm that OKernel's simulation engine accurately reproduces theoretically expected scheduling behavior, including SJF optimality, starvation prevention through aging, and the overhead-responsiveness tradeoff in preemptive algorithms.

Future work will extend OKernel with memory management visualizations (segmentation and paging), filesystem indexing mechanics, and deadlock detection and recovery simulations, further establishing the platform as a comprehensive OS educational suite.

A significant upcoming milestone is the development of the `/feat/cpu` dedicated endpoint, which will host advanced CPU architecture simulations, including pipeline visualization and cache coherence models, providing students with even deeper visibility into low-level processor operations.

ACKNOWLEDGMENT

The authors would like to acknowledge the global open-source community for providing the robust frameworks and tools that facilitated the development of this platform. All research, design, and implementation were conducted independently by the authors as part of a self-driven pursuit to enhance operating systems pedagogy.

REFERENCES

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ: John Wiley & Sons, 2018.
[2] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Pearson, 2017.
[3] A. S. Tanenbaum, and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2014.
[4] T. Anderson, and M. Dahlin, *Operating Systems: Principles and Practice*, 2nd ed. Recursive Books, 2014.