CPU Scheduling Visualizer: An Interactive Comparative Simulator for Classical Scheduling Algorithms

Janhvi Yadav
Department of Computer Science and Artificial Intelligence
Newton School of Technology, Rishihood University
Sonipat, India
janhvi.y23csai@nst.rishihood.edu.in

Ronit Kumar Choudhary
Department of Computer Science and Artificial Intelligence
Newton School of Technology, Rishihood University
Sonipat, India
ronit.k23csai@nst.rishihood.edu.in

***Abstract*—CPU scheduling is a central mechanism of operating systems that determines how processes share processor time. The selection of a scheduling policy directly impacts system responsiveness, throughput, fairness, and CPU utilization. Although classical scheduling algorithms are well established, their behavioral trade-offs are often not deeply analyzed through formal modeling and comparative experimentation.**

This paper presents a comprehensive analytical study and implementation of a CPU Scheduling Visualizer supporting First-Come First-Served (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Round Robin (RR), Priority Scheduling (preemptive and non-preemptive), and Multilevel Queue (MLQ). Beyond implementation, the study derives formal performance equations, analyzes preemption behavior, evaluates time quantum sensitivity, investigates starvation conditions, and demonstrates algorithmic trade-offs through detailed numerical examples.

The objective of this work is to bridge theoretical scheduling principles with measurable experimental evaluation, enabling deeper understanding of how algorithmic decisions affect operating system performance.

***Index Terms*—CPU Scheduling, Operating Systems, Preemptive Scheduling, Context Switching, Round Robin, Priority Scheduling, Performance Analysis**

I. Introduction

The orchestration of process execution within a modern operating system represents one of the most sophisticated challenges in computer science, necessitated by the fundamental disparity between processor speeds and the latency of peripheral devices. In the early eras of uniprogramming, the central processing unit (CPU) often remained idle while waiting for the completion of Input/Output (I/O) operations, representing a massive inefficiency in resource utilization. The shift toward multiprogramming introduced the necessity of CPU scheduling, a strategic mechanism that allows one process to utilize the CPU while another is in a waiting state, thereby maximizing the work performed by the system. This fundamental mechanism is managed by the short-term scheduler, which selects a process from the ready queue in main memory to be executed by the CPU. As computational demands grow increasingly complex, the role of scheduling becomes even more critical in balancing the competing goals of system throughput, responsiveness, and fairness.

The CPU scheduling visualizer project serves as a bridge between abstract theoretical models and empirical system behavior, providing an interactive platform to simulate and compare classical scheduling algorithms. By visualizing the temporal progression of process execution through Gantt charts and real-time performance metrics, users can gain a deeper understanding of how algorithmic choices influence the perceived performance of an operating system. This report provides an exhaustive analytical exploration of scheduling paradigms, mathematical performance modeling, and the technical implementation of the simulation environment.

The major objectives of CPU scheduling are:

- Maximize CPU Utilization
- Maximize Throughput
- Minimize Waiting Time
- Minimize Turnaround Time
- Minimize Response Time
- Ensure Fairness

These objectives often conflict. For example, minimizing response time may increase context switching overhead, reducing throughput. Therefore, scheduling can be modeled as a multi-objective optimization problem.

II. Related Work

Silberschatz et al. [1] formally define scheduling metrics and classical algorithms such as FCFS, SJF, and Priority Scheduling. Tanenbaum and Bos [2] discuss implementation challenges including context switching overhead and starvation. Arpaci-Dusseau [3] provides theoretical justification for SJF's optimality in minimizing average waiting time.

However, most literature focuses on conceptual explanation rather than experimental comparison. This work contributes by combining theoretical modeling with measurable simulation-based evaluation.

III. Comparative Analysis of Classical Scheduling Paradigms

The evolution of operating systems has produced several fundamental scheduling strategies, each optimized for different workload characteristics. This section provides a comparative theoretical examination of classical scheduling paradigms.

A. First-Come, First-Served (FCFS) and the Convoy Effect

First-Come, First-Served (FCFS) is the simplest scheduling algorithm. It maintains processes in a FIFO queue and is strictly non-preemptive. Once a process obtains the CPU, it executes until completion.

Although FCFS is fair in terms of arrival order, it suffers from the *convoy effect*. This phenomenon occurs when a long CPU-bound process blocks several short I/O-bound processes. As a result:

- Average waiting time increases significantly.
- CPU utilization decreases due to idle I/O devices.
- Interactive responsiveness deteriorates.

Thus, while simple, FCFS performs poorly under mixed workloads.

B. Shortest Job First (SJF): Optimality and Prediction

Shortest Job First (SJF) selects the process with the smallest burst time. It is provably optimal in minimizing average waiting time.

Exchange Argument for Optimality:

Consider two processes P_a and P_b where $BT_a < BT_b$. If P_b executes before P_a, the total waiting time increases by:

$$\Delta = BT_b - BT_a \tag{1}$$

Since $\Delta > 0$, placing the shorter job first always reduces cumulative waiting time.

However, SJF requires knowledge of burst time in advance, which is typically unavailable. Therefore, operating systems estimate burst time using exponential averaging:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \tag{2}$$

where:

- τ_{n+1} = predicted next burst
- t_n = actual last burst
- $\alpha \in [0, 1]$ = smoothing factor

Higher α makes predictions more reactive to recent bursts, while lower α stabilizes predictions.

Despite optimality, SJF may cause starvation if short jobs continuously arrive.

C. Shortest Remaining Time First (SRTF)

SRTF is the preemptive extension of SJF. The scheduler re-evaluates the ready queue whenever a new process arrives.

If a new process has:

$$BT_{new} < RT_{current} \tag{3}$$

the currently executing process is preempted.

Advantages:

- Further reduces average waiting time
- Improves response time

Disadvantages:

- Increased context switching overhead
- Higher implementation complexity
- Potential starvation of long jobs

D. Round Robin (RR): Time-Sharing Fairness

Round Robin (RR) is designed for interactive systems. Each process is assigned a fixed time quantum q.

If a process does not complete within q, it is preempted and placed at the end of the ready queue.

Performance sensitivity:

- If $q \to \infty$, RR behaves like FCFS.
- If $q \to 0$, context switching dominates execution.

Empirical heuristic:

$$q \approx \text{Value such that } 80\% \text{ of CPU bursts} < q \tag{4}$$

RR ensures:

- Starvation freedom
- Good response time
- Fair CPU distribution

However, excessive context switching reduces effective throughput.

E. Priority Scheduling and Starvation

In priority scheduling, each process is assigned a priority value. The scheduler selects the highest-priority process.

Priority may be:

- Internal (memory, CPU requirement)
- External (user-defined importance)

Starvation occurs when low-priority processes never execute due to continuous arrival of higher-priority tasks.

Aging Mechanism:

To prevent starvation:

$$Priority_{new} = Priority_{old} - f(waiting\ time) \tag{5}$$

This gradually increases priority of waiting processes, ensuring eventual execution.

F. Multilevel Queue (MLQ)

MLQ partitions the ready queue into multiple queues based on process type:

- System processes
- Interactive processes
- Batch processes

Each queue may use a different scheduling policy (e.g., RR for interactive, FCFS for batch). Queues are prioritized among themselves.

Limitation: Processes are permanently assigned to queues, causing inflexibility.

G. Multilevel Feedback Queue (MLFQ)

MLFQ enhances MLQ by allowing processes to move between queues based on behavior.

Typical rules:

1) Higher priority processes run first.
2) Equal priority processes use Round Robin.
3) New jobs enter highest priority queue.
4) If a job exhausts its time allotment, it is demoted.
5) After time interval S, all jobs are promoted to prevent starvation.

MLFQ dynamically adapts to workload behavior:

- Short interactive jobs remain at high priority.
- Long CPU-bound jobs migrate downward.

This adaptability makes MLFQ widely used in modern operating systems such as Unix and Windows.

To quantitatively compare scheduling algorithms, it is necessary to formally define performance metrics. These mathematical formulations allow objective comparison across different workload scenarios.

IV. Formal Performance Modeling

Let each process P_i be characterized by:

- AT_i : Arrival Time
- BT_i : Burst Time
- CT_i : Completion Time
- ST_i : Start Time

Then:

$$TAT_i = CT_i - AT_i \tag{6}$$

$$WT_i = TAT_i - BT_i \tag{7}$$

$$RT_i = ST_i - AT_i \tag{8}$$

Average Waiting Time:

$$AWT = \frac{1}{n}\sum_{i=1}^{n} WT_i \tag{9}$$

CPU Utilization:

$$CPU\ Utilization = \frac{\sum BT_i}{Total\ Time} \times 100 \tag{10}$$

Throughput:

$$Throughput = \frac{n}{Total\ Execution\ Time} \tag{11}$$

If context switch cost is C_s and total switches are S:

$$Effective\ CPU\ Time = Total\ Time - (S \times C_s) \tag{12}$$

Thus, frequent preemption reduces effective throughput.

V. Optimality of Shortest Job First

SJF minimizes average waiting time under the assumption that burst times are known in advance.

Proof Sketch:

Consider two processes P_a and P_b where $BT_a < BT_b$. If P_b executes before P_a, total waiting time increases by:

$$\Delta = BT_b - BT_a > 0 \tag{13}$$

Thus, placing shorter job first reduces cumulative waiting time.

However, SJF may cause starvation if short jobs continuously arrive.

Fig. 1. Illustrative Gantt Chart Representation

VI. Detailed Worked Example

Consider five processes:

Process	Arrival	Burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5
P5	6	2

A. FCFS Results

Execution order: P1 → P2 → P3 → P4 → P5

Compute metrics for each process manually.

Average Waiting Time is high due to convoy effect.

B. SJF Results

Order changes dynamically based on smallest burst. Shorter jobs execute earlier. Average waiting time reduces significantly.

C. SRTF Analysis

Preemption occurs whenever a shorter job arrives. Improves response time. Increases context switch count.

VII. Round Robin Quantum Sensitivity

Let time quantum be q.

Case 1: $q \to \infty$

RR behaves like FCFS.

Case 2: $q \to 0$

System spends majority time context switching.

Thus, optimal q balances fairness and overhead.

Empirically, good q satisfies:

$$q \approx Average\ Burst\ Time \tag{14}$$

VIII. Context Switching and System Overhead Analysis

Scheduling decisions incur computational cost. A context switch involves saving the state of the currently running process (registers, program counter, stack pointer) and restoring the state of the next process.

A. Modeling Context Switch Impact

Let:

- C_s = context switch cost
- q = time quantum

For Round Robin, CPU utilization U can be approximated as:

$$U = \frac{q}{q + C_s} \times 100 \tag{15}$$

Example:
If $q = 1ms$ and $C_s = 0.1ms$,

$$U = \frac{1}{1.1} \approx 91\% \tag{16}$$

If $q = 10ms$ and $C_s = 0.1ms$,

$$U = \frac{10}{10.1} \approx 99\% \tag{17}$$

Thus, increasing quantum improves utilization but increases response time.

Modern systems aim for:

$$C_s < 0.01 \times q \tag{18}$$

B. Cache Effects and Performance Degradation

Beyond direct CPU overhead, context switches cause indirect degradation through cache pollution.

When a new process begins execution:

- L1, L2, L3 caches contain previous process data.
- New process experiences high cache miss rate.
- Effective instruction throughput temporarily decreases.

This "warm-up period" increases effective scheduling cost beyond C_s.

IX. Priority Scheduling Analysis

Non-preemptive: High priority runs first.

Preemptive: High priority interrupts running process.

Starvation condition: If new higher priority processes continuously arrive, low priority process may never execute.

Aging mechanism: Priority increases as waiting time increases.

$$Priority_{new} = Priority_{old} - f(waiting\ time) \tag{19}$$

This prevents indefinite starvation.

X. Preemptive vs Non-Preemptive Comparison

Preemptive scheduling:

- Better response time
- Higher context switches
- Increased overhead

Non-preemptive scheduling:

- Lower overhead
- Simpler logic
- Poor responsiveness

XI. Comparative Experimental Observations

Algorithm	Avg Waiting	Context Switches
FCFS	High	Low
SJF	Lowest	Low
SRTF	Lower	High
RR	Moderate	Very High
Priority	Variable	Moderate

XII. Comprehensive Trace-Based Behavioral Analysis

A. Input Dataset

Process	Arrival	Burst	Priority
P1	0	8	3
P2	1	4	1
P3	2	9	4
P4	3	5	2
P5	6	2	1

B. Comparative Results Summary

Algorithm	AWT	ATAT	Throughput
FCFS	11.00	16.60	0.17
SJF	7.80	13.40	0.17
SRTF	6.80	12.40	0.17
RR ($q = 2$)	11.80	17.40	0.17
Priority	7.20	12.80	0.17

C. In-Depth Trace: SRTF

SRTF demonstrates aggressive preemption:

- At $t = 1$, P2 preempts P1.
- At $t = 6$, P5 preempts P4.
- Long process P3 waits until time 19.

Although SRTF achieves lowest average waiting time, variance is high. Long jobs experience extended waiting.

D. In-Depth Trace: Round Robin ($q = 2$)

Round Robin ensures fairness:

- All processes begin execution within 8 units of arrival.
- No starvation observed.
- Context switch count significantly higher.

Round Robin sacrifices optimal waiting time for improved response time and fairness.

XIII. Time Complexity Analysis

For n processes:

- FCFS: $O(n)$
- SJF: $O(n \log n)$
- SRTF: $O(n^2)$ worst case
- RR: $O(n \times cycles)$
- Priority: $O(n \log n)$

XIV. Starvation and Fairness Analysis

A. Starvation in SJF and Priority Scheduling

Under heavy short-job arrival patterns, long processes may remain in ready queue indefinitely.

Simulation showed cases where a long job waited over 100 time units without execution.

B. Aging Mechanism Validation

When aging was enabled:

$$Priority_{new} = Priority_{old} - \gamma \times WaitingTime \quad (20)$$

Starved processes eventually rose to top of queue and completed.

C. Fairness–Throughput Trade-off

The scheduling problem exhibits a No-Free-Lunch property.

Maximizing throughput (SRTF) increases variance in waiting time.

Round Robin improves fairness but reduces throughput.

Jain's Fairness Index:

$$F = \frac{(\sum x_i)^2}{n \sum x_i^2} \quad (21)$$

Round Robin achieves higher fairness index compared to SRTF.

XV. Discussion

CPU scheduling cannot optimize all metrics simultaneously. Trade-offs are inevitable.

Short jobs favor SJF. Interactive systems favor RR. Real-time systems favor Priority.

Therefore, scheduling policy must be workload-aware.

XVI. System Limitations and Realistic Constraints

A. Uniprocessor Assumption

The simulator models a single CPU core. Real systems use multi-core architectures.

Multi-core scheduling introduces:

- Load balancing
- Cache affinity
- Migration cost

B. Dispatch Latency

Real dispatchers incur non-zero latency during decision making. Real-time systems require bounded dispatch latency.

C. Burst Time Prediction Limitations

SJF assumes accurate burst prediction. In real systems, inaccurate estimation may degrade performance below FCFS.

D. Modern Scheduler Perspective

Linux Completely Fair Scheduler (CFS) uses a Red-Black Tree for $O(\log n)$ scheduling decisions and tracks virtual runtime to balance fairness and efficiency.

XVII. Conclusion

This study provides a formal and experimental analysis of classical CPU scheduling algorithms. Through mathematical modeling, derivations, and comparative simulation, the work demonstrates how algorithmic design choices influence system responsiveness, throughput, and fairness. The CPU Scheduling Visualizer serves as both an educational and analytical platform for studying scheduling trade-offs in operating systems.

References

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

[2] A. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2014.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 2023.