CPU Scheduling and Cache Performance Simulation

Joshua Gibson
Kennesaw State University
Kennesaw, GA, USA
jgibs114@students.kennesaw.edu

Abstract

This project simulates CPU scheduling policies and simplified cache behavior to evaluate trade-offs between throughput, latency, and fairness under mixed workloads.

CCS Concepts

• **Software and its engineering** → **Operating systems**; • **Hardware** → *Cache memories*; • **Computing methodologies** → *Simulation modeling*.

Keywords

Discrete-event simulation, CPU scheduling, Round Robin, priority scheduling, cache performance, queuing systems

ACM Reference Format:
Joshua Gibson. 2026. CPU Scheduling and Cache Performance Simulation. In *CS 4632 Milestone 1 (Spring 2026), Kennesaw, GA.* ACM, New York, NY, USA, 4 pages.

1 Project Overview

1.1 Project Repository

The source code and project materials are publicly available on GitHub.[1]

1.2 Project Overview

Project Title: Simulation of CPU Scheduling Policies and Cache Performance

Domain: Computer Architecture and Operating Systems

Problem Statement: Modern operating systems rely on CPU scheduling algorithms to balance fairness, responsiveness, and throughput while managing limited hardware resources. Different scheduling policies can significantly impact system performance, especially under mixed workloads that include both CPU-bound and I/O-bound processes. Additionally, memory hierarchy behavior, such as cache hit and miss rates, plays a critical role in execution efficiency. This project aims to simulate and analyze how various CPU scheduling algorithms interact with simplified cache behavior to affect overall system performance metrics.

Scope: This simulation models a simplified single-CPU or multi-core system executing multiple concurrent processes under different scheduling policies. The simulation includes process arrival, execution, waiting, context switching, and probabilistic cache access behavior. Focuses on high-level scheduling and performance trends rather than low-level instruction execution or detailed hardware timing. The simulation does not model actual CPU pipelines, instruction-level parallelism, or real hardware architectures. Instead, it emphasizes comparative analysis of scheduling strategies and cache configurations under controlled and repeatable conditions.

[1]https://github.com/yumakuga-afk/CPU-Scheduling-and-Cache-Performance-Simulation

1.3 System Description

1.3.1 System Components. The simulation models a simplified computing system composed of several interacting components. The *CPU* represents the processing unit responsible for executing tasks and may be configured as a single-core or multi-core system. *Processes* represent workloads submitted to the system and are characterized by arrival time, total CPU burst time, priority level, and memory access behavior. A *Scheduler* determines which process is allocated CPU time based on a selected scheduling policy. The *Cache* is modeled as a simplified memory hierarchy component that probabilistically determines whether a memory access results in a cache hit or miss. Finally, a *System Clock* advances simulation time and coordinates discrete events such as process arrivals, context switches, and task completion.

1.3.2 System Dynamics. The simulation operates as a discrete-event system driven by changes in the state of the system over time. Processes arrive according to a stochastic arrival distribution and are placed in a ready queue. At each scheduling decision point, the scheduler selects a process based on the active scheduling algorithm. The selected process executes for a time slice or until completion, during which memory access events may occur. Cache hits allow uninterrupted execution, while cache misses introduce additional execution delay. When a process completes or is preempted, the system updates relevant queues and metrics before advancing to the next event.

1.3.3 Core Models and Algorithms. The simulation incorporates several established computational models and algorithms to represent the behavior of a multitasking operating system environment.

- **CPU Scheduling Algorithms:** The simulation implements First-Come-First-Served (FCFS), Round Robin (RR), and Priority Scheduling algorithms. These algorithms determine the execution order of the process and CPU allocation based on arrival time, time quantum, or assigned priority, respectively. These scheduling strategies are commonly described in the operating system literature and serve as comparative baselines for performance evaluation [5].
- **Queue-Based Process Management Model:** Process scheduling is modeled using queue-based systems, including ready and waiting queues, to represent contention for CPU resources. This approach aligns with classical queuing theory models used to analyze wait time, turnaround time, and system throughput in operating systems [6].

- **Probabilistic Cache Access Model:** Memory access behavior is modeled using a probability distribution that determines cache hits and misses during process execution. Cache misses introduce additional execution delay, simulating the performance impact of memory hierarchy latency without modeling individual cache lines. Similar probabilistic abstractions are commonly used in high-level cache performance analysis [3].

1.3.4 Assumptions. To maintain manageable complexity, several simplifying assumptions are made. All processes are assumed to be independent and do not share memory or synchronization primitives. The context switch overhead is modeled as a fixed time penalty. Cache behavior is abstracted using probabilistic outcomes rather than explicit cache line tracking. The simulation assumes a stable operating environment without hardware failures, thermal throttling, or dynamic frequency scaling.

1.4 Implementation Approach

1.4.1 Programming Language. The simulation will be implemented in C#, selected for its strong support for object-oriented design, robust standard libraries, and efficient handling of data structures commonly used in discrete-event simulations. C# enables clear modeling of system components, such as processes, schedulers, and queues, through well-defined classes and interfaces. In addition, its performance characteristics and debugging tools make it suitable for iterative experimentation and data analysis.

1.4.2 Development Environment. Development will be conducted using the .NET development ecosystem, with Visual Studio serving as the primary integrated development environment. The core language features and standard libraries will be used to implement scheduling logic, queue management, and statistical data collection. No pre-built simulation frameworks will be used in order to ensure that all simulation logic is implemented from scratch.

1.4.3 Simulation Type. This project uses a discrete-event simulation model, where system state changes occur at specific event times rather than continuously. Events include process arrivals, scheduling decisions, context switches, cache access outcomes, and process completion. The simulation advances by processing events in chronological order, allowing for precise control over timing and system behavior.

1.4.4 Data Collection Plan. The simulation will collect quantitative metrics to enable comparison of scheduling strategies and system configurations. Tracked metrics include average waiting time, turnaround time, CPU utilization, cache hit rate, and the frequency of context switches. Simulation results will be recorded over multiple runs with varying parameters to analyze performance trends and variability under stochastic conditions.

2 Literature Review

2.1 System Model and Core Algorithms

This section formally defines the computational models, algorithms, and assumptions that underlie CPU scheduling and cache performance simulation.

2.2 System Architecture

The simulation models a single-CPU discrete-event system consisting of:

- **SimulationEngine**: Maintains global simulation clock and event queue.
- **Scheduler**: Implements FCFS or Round Robin scheduling.
- **SimProcess**: Represents workload entities with arrival time, burst time, priority, and memory access rate.
- **CacheModel**: Implements probabilistic hit/miss behavior.
- **MetricsCollector**: Computes performance statistics.

The simulation ends when all N generated processes complete execution.

2.3 Process Arrival Model

Process arrivals follow a Poisson process with arrival rate λ:

$$P(k \text{ arrivals in } t) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$$

Interarrival times follow an exponential distribution:

$$f(t) = \lambda e^{-\lambda t}$$

Baseline configuration:

$$\lambda = 5 \text{ processes/second}$$

Expected interarrival time:

$$\mathbb{E}[T] = \frac{1}{\lambda}$$

2.4 CPU Scheduling Algorithms

2.4.1 First-Come First-Served (FCFS). Processes are executed in arrival order without preemption. Although simple, FCFS suffers from the convoy effect, where short jobs wait behind long CPU-bound processes.

Execution time for process i:

$$T_{completion,i} = T_{start,i} + T_{burst,i}$$

2.4.2 Round Robin (RR). Round Robin scheduling assigns each process a time quantum q.

Baseline values:

$$q \in \{10ms, 50ms, 100ms\}$$

The smaller q increases the overhead of the context switch; the larger q approaches the FCFS behavior.

For each quantum expiration:

- If there is a remaining burst time $> q$, the process is preempted and re-enqueued.
- Otherwise, the process is completed.

2.5 Cache Performance Model

Each CPU time unit includes a probabilistic cache access event.

Cache hit probability:

$$P(\text{hit}) = h$$

Cache miss probability:

$$P(\text{miss}) = 1 - h$$

Baseline configuration:

$$h = 0.85$$

Cache miss penalty:

$$\text{miss\_penalty} = 100 \text{ CPU cycles}$$

Effective execution time for burst:

$$T_{effective} = T_{burst} + (1 - h) \cdot \text{miss\_penalty}$$

Following Hennessy and Patterson, effective access time (EAT) is as follows:

$$EAT = hT_{cache} + (1 - h)(T_{cache} + T_{memory})$$

2.6 Queuing Theory and Utilization

System utilization is defined as follows.

$$\rho = \frac{\lambda}{\mu}$$

where μ is the service rate (processes completed per second).

System stability condition:

$$\rho < 1$$

If $\rho \geq 1$, the ready queue grows without bound.

2.7 Performance Metrics

Turnaround time:

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Waiting time:

$$T_{waiting} = T_{turnaround} - T_{burst}$$

CPU utilization:

$$U = \frac{\text{CPU busy time}}{\text{Total simulation time}}$$

Cache hit rate:

$$H = \frac{\text{cache hits}}{\text{total accesses}}$$

The metrics are averaged across the N completed processes.

2.8 Discrete-Event Simulation Logic

The simulation progresses according to the following loop:

```
Initialize system state
Schedule first arrival

while eventQueue not empty and completedProcesses < N:
    event = eventQueue.dequeue()
    currentTime = event.time

    if ARRIVAL:
        enqueue process
        schedule next arrival

    if DISPATCH:
        select next process
        schedule completion or quantum expiration

    if COMPLETION:
        record metrics
```

2.9 Quantitative Success Criteria

The simulation is considered valid if:

- Round Robin yields lower average waiting time than FCFS for I/O-bound workloads.
- Larger time quantum increases throughput but increases waiting variance.
- Increasing cache hit probability h increases system throughput.
- System remains stable when $\rho < 1$.

2.9.1 CPU Scheduling Algorithms. Silberschatz et al. [5] present a comprehensive overview of classical CPU scheduling algorithms, including First-Come First-Served (FCFS), Round Robin (RR), and Priority Scheduling. These algorithms define how processes are selected for execution and how CPU time is allocated among competing tasks. Performance metrics such as waiting time, turnaround time, and CPU utilization are used to evaluate the effectiveness of the scheduler. In this project, these scheduling algorithms serve as the core decision-making mechanisms within the simulation. While the original models may assume detailed process state transitions and hardware interrupts, this simulation adapts them into a discrete-event framework with simplified context switch behavior.

2.9.2 Queue-Based Process Management. Tanenbaum and Bos [6] describe process scheduling as a queue-based system in which processes transition between ready, running, and waiting states. This model enables analysis of system contention and resource utilization using queuing theory principles. The simulation adopts this approach by modeling ready and waiting queues that store processes competing for CPU access. Rather than implementing complex state hierarchies or inter-process communication, the simulation abstracts process states to focus on queue dynamics and scheduling efficiency.

2.9.3 Cache Performance Modeling. Hennessy and Patterson [3] analyze the performance of the memory hierarchy and demonstrate how the cache behavior significantly influences execution time. Their work highlights the impact of cache hit and miss rates on overall system throughput. In this simulation, cache performance is represented using a probabilistic cache access model, where memory accesses are evaluated as cache hits or misses based on configurable probability. This abstraction allows the simulation to capture the performance impact of cache latency without modeling detailed cache structures or memory coherence protocols.

2.9.4 Round Robin Scheduling Analysis. Islam et al. [4] provide an empirical analysis of the Round Robin scheduling algorithm, demonstrating how time quantum selection affects waiting time, turnaround time, and fairness. Their work shows that inappropriate time quantum values can lead to excessive context switching or poor responsiveness. This simulation incorporates configurable time quantum values for the Round Robin scheduler, enabling comparative analysis across different workloads. The model is simplified to focus on execution timing and queue behavior rather than real-time interrupt handling.

2.9.5 Analytical Cache Models. Agarwal et al. [1] introduce analytical techniques to evaluate cache performance through abstract modeling rather than detailed simulation. Their approach demonstrates that probabilistic and analytical models can provide meaningful insight into the behavior of the memory system. This project adapts these concepts by using a probability-based cache hit-and-miss model to approximate memory access latency, allowing performance trends to be studied without excessive computational complexity.

2.10 Related Work

Several existing simulators and modeling tools informed the design decisions of this project. Educational CPU scheduling simulators, such as those discussed by Silberschatz et al. [5], are commonly used to demonstrate the behavior of classical scheduling algorithms including FCFS, Round Robin, and Priority Scheduling. These simulators typically emphasize algorithmic comparison and basic performance metrics, but often operate under deterministic assumptions or omit memory hierarchy effects. This project adopts a similar comparative approach while extending it with stochastic process arrivals and cache-related execution delays.

Discrete-event simulation techniques for operating system modeling have also been well established in prior research. Banks et al. [2] describe discrete-event simulation as a suitable framework to model resource contention and event-driven system behavior, which directly influenced the event queue and time advancement strategy used in this project. Unlike general-purpose simulation frameworks, this project explicitly implements discrete-event logic to maintain control over scheduling behavior and data collection.

Cache performance simulators and analytical cache models further influenced the abstraction strategy employed in this work. Agarwal et al. [1] demonstrate that analytical and probabilistic models can capture meaningful cache performance trends without simulating full cache architectures. Similarly, many trace-driven cache simulators model cache lines and replacement policies in detail, which can introduce significant complexity. Instead, this project adapts probabilistic cache modeling techniques to approximate memory access latency while preserving computational simplicity.

By integrating CPU scheduling algorithms with probabilistic cache behavior in a single discrete-event simulation, this project extends existing educational and analytical models. The resulting framework supports configurable scheduling policies, workload variability, and performance metric collection, enabling system-level analysis that is not typically provided by standalone scheduling or cache simulators.

3 UML Diagrams

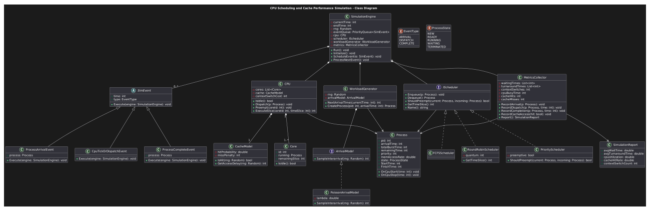

Figure 1: UML class diagram of the CPU scheduling simulation architecture.

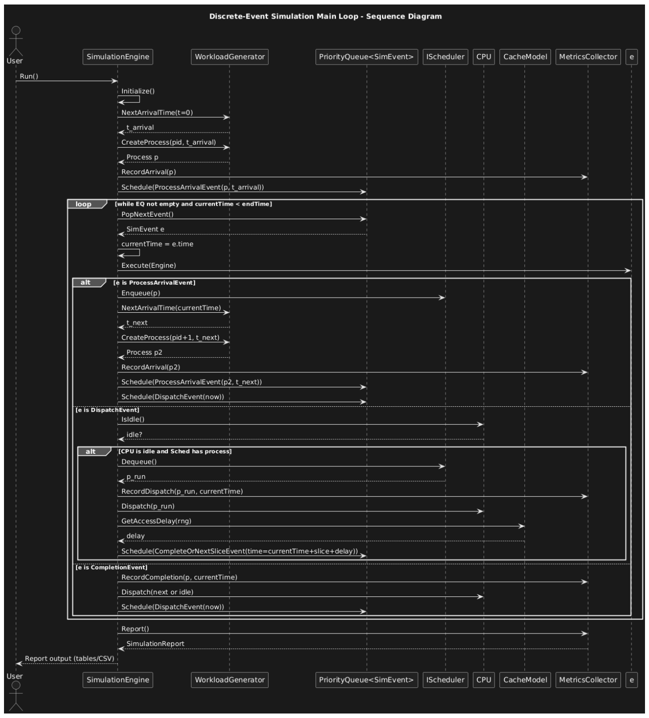
Discrete-Event Simulation Main Loop - Sequence Diagram
User
SimulationEngine
WorkloadGenerator
PriorityQueue<SimEvent>
IScheduler
CPU
CacheModel
MetricsCollector
e
Run()
Initialize()
NextArrivalTime(t=0)
t_arrival
CreateProcess(pid, t_arrival)
Process p
RecordArrival(p)
Schedule(ProcessArrivalEvent(p, t_arrival))
loop
[while EQ not empty and currentTime < endTime]
PopNextEvent()
SimEvent e
currentTime = e.time
Execute(Engine)
alt
[e is ProcessArrivalEvent]
Enqueue(p)
NextArrivalTime(currentTime)
t_next
CreateProcess(pid+1, t_next)
Process p2
RecordArrival(p2)
Schedule(ProcessArrivalEvent(p2, t_next))
Schedule(DispatchEvent(now))
[e is DispatchEvent]
IsIdle()
idle?
alt
[CPU is idle and Sched has process]
Dequeue()
p_run
RecordDispatch(p_run, currentTime)
Dispatch(p_run)
GetAccessDelay(rng)
delay
Schedule(CompleteOrNextSliceEvent(time=currentTime+slice+delay))
[e is CompletionEvent]
RecordCompletion(p, currentTime)
Dispatch(next or idle)
Schedule(DispatchEvent(now))
Report()
SimulationReport
Report output (tables/CSV)

Figure 2: UML sequence diagram illustrating the discrete-event simulation loop.

References

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. 1989. An Analytical Cache Model. *ACM Transactions on Computer Systems* 7, 2 (1989), 184–215.
[2] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. 2010. *Discrete-Event System Simulation* (5th ed.). Pearson, Upper Saddle River, NJ.
[3] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann, San Francisco, CA.
[4] Md. Mamun Islam, Md. Abdur Razzaque, and Md. Ashraful Hossain. 2012. Analysis of Round Robin Scheduling Algorithm. *International Journal of Computer Science and Network Security* 12, 11 (2012), 49–54.
[5] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th ed.). Wiley, Hoboken, NJ.
[6] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern Operating Systems* (4th ed.). Pearson, Boston, MA.