CSC227 CPU Scheduler Simulator Report

1. Introduction

This report documents the design and evaluation of the CSC227 CPU Scheduler Simulator. The simulator models a single CPU executing workloads using three standard algorithms—Shortest Job First (SJF), Round Robin (RR), and Priority with aging—while coordinating supporting threads for job ingestion and memory admission. The code base is organized under a modular `simulator.*` package hierarchy.

The following sections address the required topics: development tooling, simulated system calls, strengths and weaknesses, threading behavior, output preferences, and considerations for expanding the simulator into a full operating-system model.

2. Software and Hardware Tools Used

- **Programming Language**: Java 17 (open-source distribution via OpenJDK).
- **Build and Runtime Environment**: Windows 11 Pro, 64-bit. The project is routinely executed from a PowerShell terminal using the Microsoft-provided JVM.
- **Editor / IDE**: Visual Studio Code with Microsoft's Java extensions for syntax support, linting, and project navigation.
- **Version Control**: Git (CLI) hosted on GitHub (`FaisalSWEN/CSC227-OS-Project`).
- **Ancillary Utilities**:
  - `find` (MSYS or Git for Windows shell) / PowerShell's `Get-ChildItem` for source discovery during compilation.
  - Markdown editing extensions inside VS Code for authoring documentation.
- **Hardware**: HP Envy x360 laptop (11th Gen Intel Core i7, 16 GB RAM, 1 TB NVMe SSD). The simulator does not depend on specialized hardware features beyond a multi-core CPU capable of running several user-level Java threads.

3. System Call Interface

The simulator uses a `SystemCallHandler` class to model kernel-visible operations that user processes (or the runtime) would invoke. Each method timestamps the event and records a human-readable log entry. Table 1 summarizes the interface.

System Call	Arguments	Return Value	Description
`createProcess(pcb)`	`ProcessControlBlock pcb`	`void`	Records the creation of a PCB, logging burst time and base priority.
`enqueueJob(pcb, queueSize)`	`ProcessControlBlock pcb`, `int queueSize`	`void`	Indicates that the PCB entered the job queue; captures current queue length.
`allocateMemory(pcb, usedMemory, totalMemory)`	`ProcessControlBlock pcb`, `int usedMemory`, `int totalMemory`	`void`	Logs a successful memory reservation for the process.
`admitToReady(pcb, readyTime, degree)`	`ProcessControlBlock pcb`, `int readyTime`, `int degree`	`void`	Notes that the process entered the ready queue, along with the current degree of multiprogramming.
`dispatch(pcb, dispatchTime)`	`ProcessControlBlock pcb`, `int dispatchTime`	`void`	Marks that the scheduler dispatched the process onto the CPU.
`yield(pcb, currentTime, remainingTime)`	`ProcessControlBlock pcb`, `int currentTime`, `int remainingTime`	`void`	Captures a preemptive context switch (used by Round Robin when a time quantum expires).
`complete(pcb, completionTime)`	`ProcessControlBlock pcb`, `int completionTime`	`void`	Records that the process finished execution.
`releaseMemory(pcb, usedMemory, totalMemory)`	`ProcessControlBlock pcb`, `int usedMemory`, `int totalMemory`	`void`	Logs the release of main-memory resources after termination.
`reportStarvation(pcb, waitingTime, degree)`	`ProcessControlBlock pcb`, `int waitingTime`, `int degree`	`void`	Notes a starvation incident and explains the wait duration prior to aging.
`boostPriority(pcb)`	`ProcessControlBlock pcb`	`void`	Records that aging raised the dynamic priority of a waiting process.

These calls collectively narrate the life cycle of each process and allow the simulator to print comprehensive execution traces without embedding logging logic inside the scheduling algorithms themselves.

4. Strengths and Weaknesses

Strengths

- **Modularity**: The simulator separates concerns across packages (`app`, `core`, `io`, `memory`, `scheduler`), simplifying maintenance and extension.
- **Concurrent Pipeline**: Dedicated threads for job reading and memory-aware loading mirror real OS behavior and keep the scheduler thread focused on CPU dispatch.
- **Comprehensive Reporting**: `SchedulingResult` produces Gantt charts, per-process statistics, and starvation notices, providing multiple analytical views.
- **Extensibility**: Implementing a new `Scheduler` requires satisfying a single interface and can reuse the shared `SimulationContext`.

Weaknesses

- **Single CPU Simplification**: The simulator assumes one CPU core.
- **Static Job File**: Workloads are static inputs read at startup. Real systems allow interactive process submission.
- **Limited Timing Model**: Context switches, IO wait, and cache/memory latencies are abstracted away, reducing realism.
- **Resource Modeling**: Only memory is modeled; other resources (IO devices, locks) are not simulated, which constrains scenario variety.

5. Threading Behavior and Performance Impact

The application creates three Java threads per simulation run:

1. **Main/Scheduler Thread** – executes the selected scheduling algorithm.
2. **Job Reader Thread** – parses `job.txt` and fills the job queue.
3. **Process Loader Thread** – blocks on memory availability and moves PCBs into the ready queue.

Multithreading primarily improves responsiveness rather than raw throughput. The reader and loader threads decouple IO-bound and memory-bound tasks from CPU scheduling, ensuring the scheduler is never blocked on file I/O or memory waits longer than necessary. However, because the workload is small and the scheduling is CPU-light, wall-clock speedup is modest. The main benefit is architectural clarity: the scheduler can focus on CPU policy while background threads emulate the OS's job and memory subsystems. On multi-core hardware, these threads may run in parallel, preventing the scheduler from idling when prerequisite work is pending.

6. Output Format Preference

I prefer combining the Gantt chart with tabular summaries rather than choosing a single format. The ASCII Gantt chart offers an intuitive, timeline-oriented visualization of context switches and CPU utilization, which is invaluable when diagnosing scheduling behavior. Tables, on the other hand, provide precise numeric metrics (waiting time, turnaround, response) that are better suited for quantitative comparisons. Presenting both allows quick pattern recognition (via the chart) alongside exact figures (via the table), giving stakeholders both a qualitative and quantitative understanding of performance.

7. Toward a Full Operating-System Simulation

To evolve this project into a richer operating-system simulator, several architectural enhancements are necessary:

- **Kernel Scheduler**: Generalize the `Scheduler` interface to handle multiple CPUs, interrupt queues, and process states beyond ready/running (e.g., blocked on IO).
- **Device and IO Modeling**: Introduce device drivers and an event subsystem so processes can issue IO requests, block, and resume based on device completion events.
- **Memory Management Extensions**: Implement paging or segmentation, page replacement policies, and per-process address spaces.
- **System Call Layer**: Formalize a syscall dispatcher that converts user-space requests into kernel actions, including file IO, process control, and synchronization primitives.

These modifications would require reworking `SimulationContext` to represent the global OS state, decomposing monolithic classes into service modules (e.g., VM manager, file system), and designing a richer event simulation engine. The current modular structure provides a solid foundation for incremental expansion.

8. Conclusion

The CSC227 CPU Scheduler Simulator fulfills its educational objective by demonstrating three classic scheduling policies within a concurrent, memory-aware environment. Its modular architecture, detailed logging, and clear separation of responsibilities make it a strong platform for experimentation. The identified weaknesses and future enhancements outline a roadmap toward a comprehensive operating-system simulation should the project continue to evolve.