WoahACPU: CPU Scheduler Simulator

Othmane AZOUBI
College of Computing
Mohammed VI Polytechnic University
Benguerir, Morocco

Mohammed-Rida ELHANI
College of Computing
Mohammed VI Polytechnic University
Benguerir, Morocco

***Abstract*—This report is a brief summary of the project, written with love & care. We aimed to keep it concise and focused on core ideas and clarity rather than excessive length.**

The project simulates five CPU-scheduling algorithms—FCFS, SJF, Priority, Round Robin, and Priority + RR—using the Unity engine. Workloads are defined by arrival time, burst time, and priority. Input can come from a JSON file or be randomly generated. The system measures metrics such as waiting time and turnaround time so you can compare each algorithm's trade-offs in throughput, fairness, and responsiveness.

I. Project Architecture

A. Overall Architecture

Figure 1 shows that the system is divided into two main components:

- **Unity Client:** The core simulator and user-interface. It animates processes on the CPU, lets the user start, pause, or adjust the simulation speed, and displays which processes are executing.
- **Flask Backend:** Responsible for generating processes (either randomly or from a file), collecting benchmarking metrics. Process are exposed via RESTful endpoints to the Unity client.

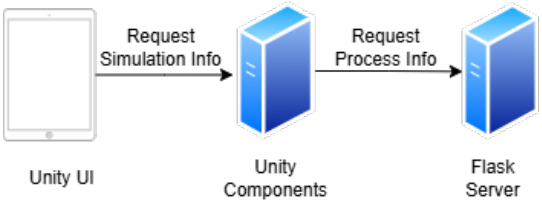

Fig. 1. System Architecture

B. Motivation for Design Choices

Basically, we had two ideas. The first was to implement the scheduler in an object-oriented language so we could leverage design patterns and apply what we've learned in our software-development course. The second was to create an interactive visualization—so you can actually see which process is occupying the CPU and watch the "hardware" timer discussed in Section II.C

We considered many languages: Python, LUA, C++, Java, and Csharp. We ruled out Python because it would make the project too easy and we wouldn't learn the problem-solving skills this project requires. We didn't want Java, since we've already 'had enough' with Spring Boot. Although LUA (via CoronaSDK or LÖVE) would be fast and would support our interactive-visualization idea, it's hard to simulate true OOP in Lua (No real OOP—it's just simulated using tables and metatables). C++ is a strong choice—but I expect most of our classmates will pick it, and we'd like our project to stand out. That leaves Csharp with the Unity engine for visualization.

C. Class Diagram

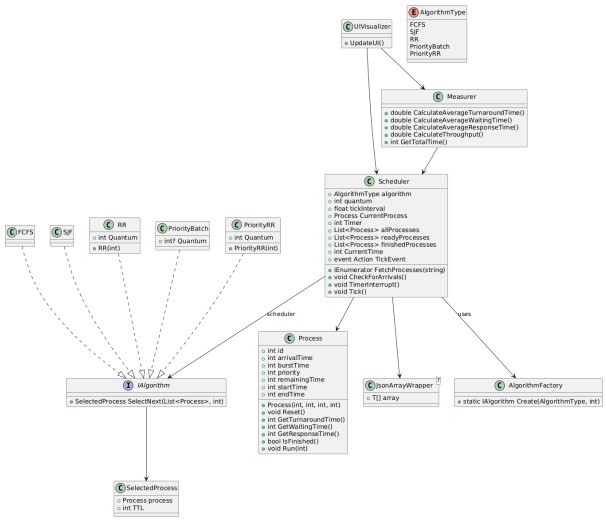

Fig. 2. Class Diagram

D. End Points

- **GET /**
  - Reads `processes.json` and returns its contents.
  - **Responses:**
    - 200 OK: JSON array of process objects
    - 500 Internal Server Error: if the file cannot be read or parsed
- **POST /generate**
  - Creates a list of random processes based on the JSON payload.
  - **Example Request JSON:**

```
{
  "minburst": 1,
  "maxburst": 10,
  "minarrival": 0,
  "maxarrival": 20,
```

```
  "minpriority": 1,
  "maxpriority": 5,
  "count": 5
}
```

  - **Responses:**
    - 200 OK: JSON array of generated processes, sorted by arrival time
    - 400 Bad Request: missing body, wrong types, or min $>$ max
- **POST /benchmark**
  - Stores performance results for one algorithm in server memory.
  - **Example Request JSON:**

```
{
  "algorithm": "RR",
  "metrics": {
    "avg_wait": 12.3,
    "avg_turnaround": 20.5,
    other metrics etc....
  }
}
```

  - **Responses:**
    - 201 Created: {"status":"ok"}
    - 400 Bad Request: if "algorithm" or "metrics" is missing
- **GET /benchmark**
  - Renders an HTML page with a table of all stored results. And displays them as nice graphs and charts for comparaison.
  - **Response:**
    - 200 OK: HTML page showing algorithms and their metrics

II. Design Choices

The system is designed carefully to respect Modularity, Seperation of concerns and SOLID principles here is how it is done:

A. Structs & Interfaces

Before we proceed we'd like to defend our structure choices from Section I.C:

- It is not needed to repeat the definition of a process entry; the Class Diagram already makes it clear.
- **Enum `AlgorithmType`**
  Lists the possible algorithm types. This avoids using messy string comparisons.
- **Interface `IAlgorithm`**
  Holds any scheduling algorithm as a "strategy." This lets us use the Strategy pattern and keeps the core scheduler simple. The scheduler only calls the method `SelectNext` and does not need to know which algorithm runs. This also follows the Single Responsibility Principle: the scheduler moves processes and handles timer interrupts, and each algorithm class only picks the next process.
- **Struct `SelectedProcess`**
  Contains the data that an algorithm returns. At first, we thought about returning only the `Process`, but then the scheduler would not know how long that process should run before the next interrupt. This is important because the scheduler does not know if the algorithm is preemptive or not.
- **Class `AlgorithmFactory`**
  Uses the Factory pattern to make `IAlgorithm` objects. This keeps the scheduler from having to change when we add or remove algorithms.

B. Flask Server

We'll cover the Flask server now, since the next subsection refers back to it. The endpoints are defined in Section I.D, where their functionality is already explained.

We chose this approach to keep Unity from handling file I/O on our systems—it would complicate the production build process. With a Flask server, we can change the I/O handling at any time without rebuilding or updating the Unity code.

A second reason is that, with our experience using Unity, we know that its random-number generation can become predictable and is tricky to fix. Python's random-number libraries are simpler and more reliable for this task.

Finally, this approach will simplify stress testing.

C. Scheduler

In this subsection, we describe the simulation's main component: the scheduler. The scheduler class is a state machine with three main states summarized in Figure 3

a) State 1: Begin: In this phase, the system fetches processes from the Flask server and caches them. All scheduler attributes are reset to their initial values (empty or zero). The processes remain in cache until State 3 loads them into the process table. Then the scheduler moves to State 2.

b) State 2: Scheduling Decision: Here the scheduler is faced with two decisions:

- If the timer expires and the process still has remaining time (remaining time $>$ 0), the scheduler returns it to the ready queue.
- Otherwise, it moves the process to the finished queue for later benchmarking.

Then scheduler applies the `IAlgorithm.SelectNext` strategy to choose the next process. It updates its timer and gives CPU control to that process (Moves to State 3)

c) State 3: CPU Tick: This state simulates the CPU itself. We combine it with the scheduler code to avoid overcomplicating the implementation. The `void Tick()` function does the following:

1) Checks for new arrivals by scanning the cache from State 1. Any process whose arrival time equals the current time is added to the ready queue.

2) Executes one CPU instruction (one "tick"). The tick frequency is defined by a class attribute.
3) If the hardware timer expires (timer = 0) or the process finishes executing (remaining time = 0), the simulated hardware issues a `TimerInterrupt()` event and returns control to the scheduler (Moves to State 2).
4) After each tick, a `TickEvent` is invoked to update the UI so users can see what is happening.

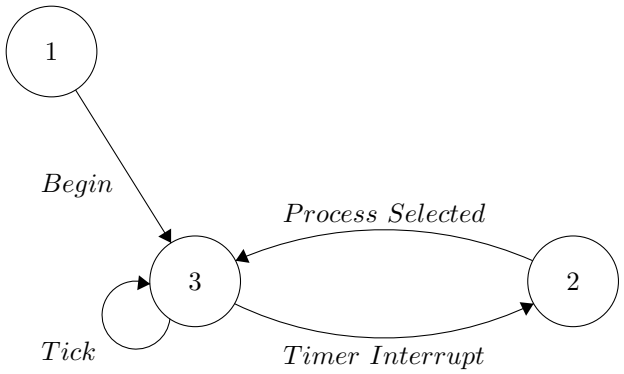

Fig. 3. Scheduler Class as a State Machine

D. Measurer

This class is simple: it reads from the finished queue and collects measurements like arrival time, start time, finish time, etc.. And then compute the evaluation metrics that the UI component will display in its text boxes.

E. Algorithms

I left this part at the end since it just repeats the textbook. The implemented algorithm strategies are as follows:

- **FCFS**
  Processes are handled in the order they arrive. No preemption: once a process starts, it runs until completion. Simple but can lead to long waits (convoy effect).
- **SJF**
  Picks the process with the smallest total service time next. It is non-preemptive (runs to completion). Minimizes average wait time but requires knowing job lengths in advance. Also can lead to starving long length jobs.
- **RR (Round-Robin)**
  Each process runs for a fixed time slice (quantum). If it doesn't finish, it goes to the back of the ready queue. Ensures fairness and good responsiveness for time-sharing systems.
- **PriorityBatch**
  Processes have static priorities; the scheduler picks the highest-priority job next. Non-preemptive: once started, a job runs until completion. Can starve low-priority jobs.
- **PriorityRR**
  Combines priority scheduling with time-slicing. Within each priority level, processes share the CPU in round-robin fashion. Balances priority treatment with fairness.

III. Testing & Preview

A. Test Case

Simulation Screenshots are demonstrated in Figure 4 & 5
A list of processes used in the simulation (View `processes.json` (quantum = 2 for RR and Priority+RR):

TABLE I
Process Attributes

ID	Arrival Time	Burst Time	Priority	Remaining Time
1	0	8	3	8
2	2	4	1	4
3	4	9	2	9
4	6	5	4	5
5	8	2	1	2

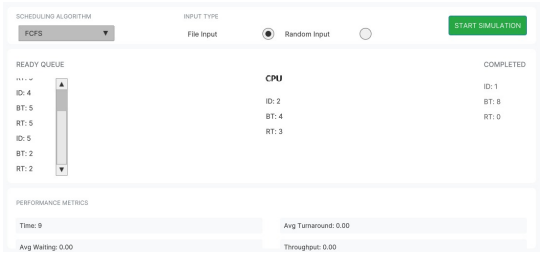

Fig. 4. During Simulation State.

Fig. 5. End-of-simulation state (all processes complete).

B. Benchmarking Results

We record average waiting time, average turnaround time, average response time, and throughput for each algorithm. Table II shows these results.

TABLE II
Summary of Results (Quantum = 2)

Algorithm	Avg. Waiting	Avg. Turnaround	Avg. Response	Throughput
FCFS	9.4	15.0	9.4	0.312
SJF	6.2	11.8	6.2	0.424
Priority	7.4	13.0	7.4	0.385
Round Robin	10.6	16.2	2.4	0.308
Priority+RR	7.2	12.8	3.8	0.390

- **FCFS** follows strict arrival order: average waiting time = 9.4, average turnaround time = 15.0, average response time = 9.4. Simple to implement, but short tasks arriving later must wait behind long ones, causing the "convoy effect."
- **SJF** has the lowest average waiting time (6.2) and turnaround time (11.8), but long processes may wait too long.
- **Priority** reduces waiting (7.4) and turnaround (13.0) compared to FCFS, yet low-priority tasks can still be delayed.

- **Round Robin** gives each process a fast start (avg response = 2.4) but raises average waiting (10.6) and turnaround (16.2).
- **Priority + RR** combines both: high-priority jobs start sooner (avg response = 3.8), while waiting (7.2) and turnaround (12.8) stay moderate.

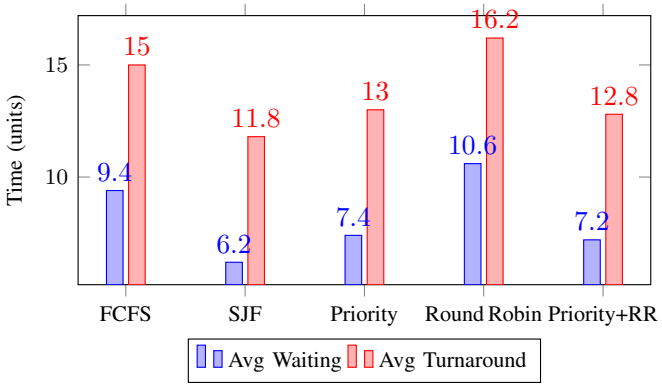

Fig. 6. Comparison of average waiting and turnaround times.