CPU Scheduling Simulator: Design, Implementation, and Comparative Study

Samia Lachgar Mounia Baddou

Prof. Youssef Iraqi
College Of Computing
Mohammed VI Polytechnic University, Ben Guerir

May 2, 2025

Quick Navigation

Contents

Abstract

This report describes a Python-based tool built to simulate how a computer decides which processes to run on its CPU. We tested five different methods, **First Come First Serve** (FCFS), **Shortest Job First** (SJF), **Static Priority**, **Round Robin** (RR), and **Round Robin with Priority** to see how they handle tasks. The tool records how tasks are processed, creates visual timelines (Gantt chart, CPU usage), and measures performance, such as how long tasks wait or how fair the method is. Through our comparison tests, we get some very interesting results in regards to waiting time, fairness and balance between the two.

1 Introduction

Imagine a busy restaurant kitchen with only one chef who can cook one dish at a time. Many orders arrive, but the chef must decide which to cook first. This is similar to how a computer CPU (Central Processing Unit) handles tasks, called *processes*. CPU scheduling is the method the computer uses to choose which process gets to use the CPU next when multiple processes are ready. Different scheduling methods prioritize different goals, such as finishing tasks quickly, ensuring fairness, or focusing on urgent tasks.

Our project created a *CPU scheduling simulator*, a software tool that mimics how a computer schedules processes. It lets us test and compare a number of scheduling methods in a controlled way. We built this tool to help understand how these methods work, see their strengths and weaknesses, and visualize their performance with charts and graphs. This is useful for students, researchers, or anyone curious about how computers manage multiple tasks efficiently.

1.1 Objectives

Our simulator was designed with clear goals to make it useful and user-friendly:

- **Create a flexible tool**: Build a simulator that can easily show the properties of the scheduling algorithms.
- **Offer easy access**: Provide two ways to use it: a command-line interface (through a terminal) and a web interface (using a browser).
- **Show results visually**: Generate timelines (Gantt charts) and graphs to make visualization of how processes are handled easy.
- **Compare methods fairly**: Test the five scheduling methods on different sets of tasks to measure their performance and understand their differences.

2 System Architecture

The simulator is a well-organized toolbox, with different parts working together to run simulations and show results. Figure 1 shows a diagram of how these parts connect. The project was coded in **Python and JavaScript** and is organized into folders and files, as shown in Listing 2, to keep everything tidy and easy to maintain.

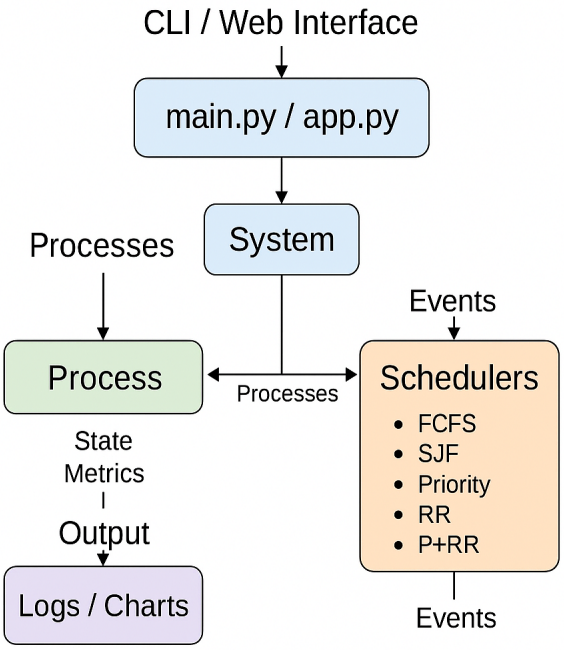


Figure 1: High-level architecture of the simulator, showing how different parts (like process management, schedulers, and interfaces) work together.

Project Tree

```
OS-ASSIGN-1-CC
main.py   # Terminal-based interface
app.py   # Flask web app (UI-based)
static/   # Static files (CSS, JS, images)
templates/   # HTML templates for Flask UI
system/   # Helper functions for the UI
schedulers/   # Scheduling algorithm implementations
ProcessClass/   # Process model and input logic
utils/   # Utility modules (e.g., file I/O)
README.md   # Project overview
```

The simulator has three main components:

- **Process Management**: Handles processes and tracks their details, such as when they arrive or how long they need the CPU.
- **Scheduling Methods**: Each algorithm is coded separately, thus maintaining modularity.
- **User Interfaces**: Users can interact with the simulator through a command line or a web app, thus favoring user experience.

3 Process Model

A *process* in our simulator is the main class representing the unit of information. Each process has its own details:

- **Arrival Time**: When the process arrives.
- **Burst Time**: CPU Time needed by the process.
- **Priority**: Urgency of task (we work in this code with the convention that smaller number means higher priority).
- **Metrics**: Such as awaiting time, start time and end time.
- **State**: Expresses the current state of the process: *NEW*, *READY*, *RUNNING*, *BLOCKED* and *COMPLETED*

4 Implemented Algorithms

We were interested in implementing five scheduling methods, each with its own properties and logic. In the following, we explain each method, how it works, and what its best points are, using simple analogies.

4.1 First Come First Serve (FCFS)

What it does: Processes are handled in the order they arrive, like a queue at a ticket counter where the first person in line is served first.

How it works: The simulator sorts processes by arrival time. It picks the earliest one, runs it until it is done (called *non-preemptive*), then moves to the next. If no process is ready (e.g., the next process hasn't arrived yet), the CPU waits.

Pros and Cons: *FCFS* is simple and fair for arrival order, but if a long task arrives first, later tasks wait a long time, causing delays (called the "convoy effect").

Example: If three orders arrive at 0, 1, and 2 minutes needing 5, 2, and 3 minutes to cook, *FCFS* cooks them in that order, even if the 5-minute dish delays the quicker ones.

4.2 Shortest Job First (SJF)

What it does: Picks the process that takes the least time to run,similar to choosing the quickest dish to cook first to clear orders faster.

How it works: The simulator looks at all ready processes and selects the one with the shortest burst time. It runs it to completion, then picks the next shortest. If no processes are ready, the CPU waits.

Pros and Cons: *SJF* reduces average waiting time by finishing quick tasks first, but long tasks might wait forever if short tasks keep arriving (called "starvation").

Example: For the same three orders (5, 2, 3 minutes), *SJF* cooks the 2-minute dish first, then the 3-minute, and finally the 5-minute, minimizing wait times for most orders.

4.3 Static Priority

What it does: Chooses the process with the highest priority (lowest priority number), such as prioritizing VIP orders.

How it works: The simulator checks ready processes and picks the one with the highest priority. It runs it to completion, then selects the next highest-priority process. The "aging" feature increases a process's priority if it waits too long, preventing starvation.

Pros and Cons: Great for urgent tasks, but low-priority tasks might wait a long time without aging.

Example: If the three orders have priorities 3, 1, and 2 (lower is better), the chef cooks the priority-1 order first, then priority-2, and finally priority-3, even if the priority-1 order takes longer.

4.4 Round Robin (RR)

What it does: Gives each process a short time slot (quantum, set to 3 time units), like a chef working on each dish for a few minutes before switching to the next.

How it works: The simulator puts processes in a queue. Each process gets up to, let us say, 3 time units on the CPU. If it finishes, it is done; if not, it goes back to the queue's end. This continues until all processes are complete.

Pros and Cons: RR is fair, as every process gets regular CPU time, but switching between processes (context switching) adds overhead, slowing down completion for long tasks.

Example: For the three orders, the chef might cook the first for 3 minutes, switch to the second for 2 minutes (finishing it), then work on the third, cycling until all are done.

4.5 Round Robin with Priority

What it does: Combines priority and RR by grouping processes by priority and using RR within each group, similar to prioritizing VIP orders but cooking each VIP dish in short bursts.

How it works: The simulator organizes processes into priority queues. It picks the highest-priority queue and uses RR for its processes. If a higher-priority process arrives, it takes over. Aging adjusts priorities for long-waiting processes.

Pros and Cons: Balances urgency and fairness but is more complex and has context-switching overhead.

Example: If the three orders are in priority groups, the chef uses RR for the highest-priority group first, then moves to lower-priority groups, ensuring VIPs are served quickly but fairly.

5 Visual Comparison

To understand how these methods work, we use *Gantt charts*, which are timelines showing when each process runs on the CPU.

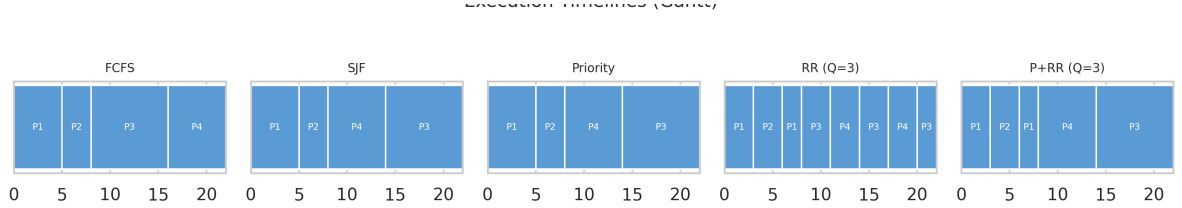
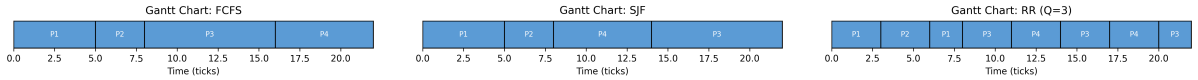


Figure 2: A combined timeline showing how all five methods schedule processes over 22 time units. Each colored bar represents a process running on the CPU.



(a) FCFS: Processes run in arrival order.

(b) SJF: Shortest tasks run first.

(c) RR: Processes take turns every 3 units.

Figure 3: Detailed Gantt charts for three methods, showing how processes are scheduled over time.

We also use a *boxplot* to show how long processes wait before running. A narrow box means most processes wait similar times (fair), while a wide box shows uneven waiting times.

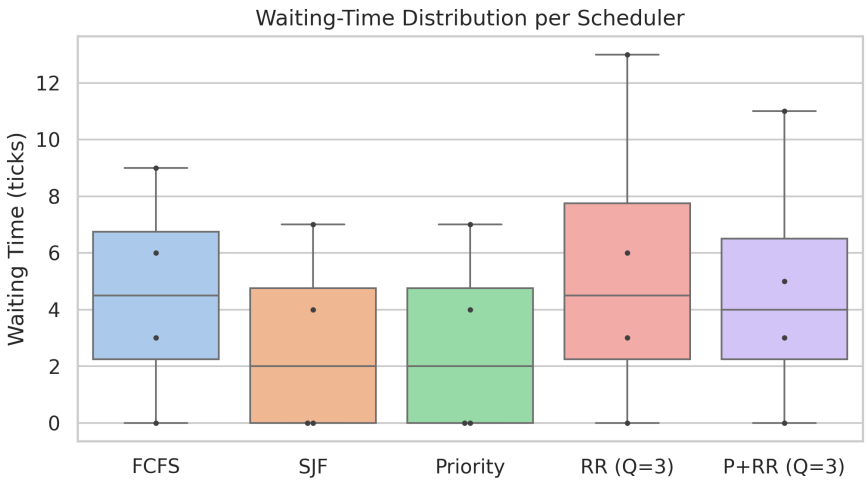


Figure 4: Boxplot of waiting times for each method. Shorter boxes mean fairer waiting times across processes.

For RR, we tested different quantum sizes (time slots) to see how they affect waiting time and throughput (tasks completed per unit time). A smaller quantum means more fairness but slower completion due to switching.

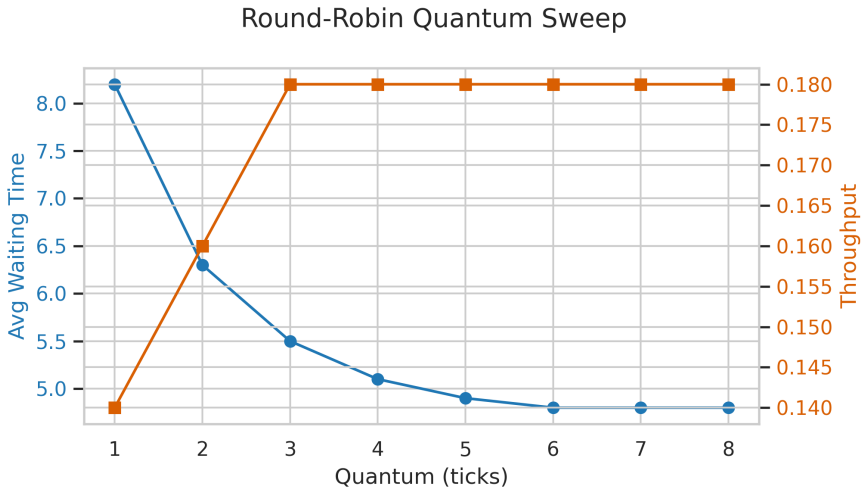


Figure 5: How RR's quantum size affects waiting time and throughput. Smaller quanta increase fairness but may reduce efficiency.

6 Quantitative Results

We tested the simulator with a set of four processes to measure performance. Table 1 shows three key metrics:

- **Average Waiting Time** ($\overline{W}$): How long processes wait before starting.
- **Average Turnaround Time** ($\overline{T}$): Total time from arrival to completion.
- **Throughput** (τ): How many processes finish per time unit.

Table 1: Performance metrics for a mixed workload with 4 processes.

Policy	$\overline{W}$ (Waiting Time)	$\overline{T}$ (Turnaround Time)	τ (Throughput)
FCFS	4.25	9.75	0.18
SJF	3.75	9.25	0.18
Priority	3.75	9.25	0.18
RR (Q=3)	5.50	11.00	0.18
P+RR (Q=3)	4.75	9.50	0.18

Figure 6 shows these metrics as bars, making it easy to compare methods. For example, SJF has the lowest waiting time, while RR has the highest turnaround time due to switching.

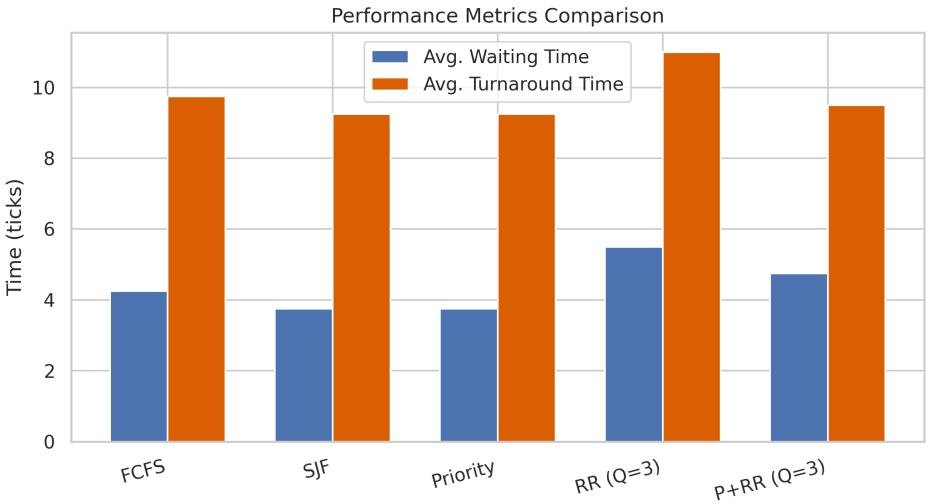


Figure 6: Bar chart comparing average waiting and turnaround times for each method.

7 Extended Metrics

We also measured *fairness*, which shows if all processes are treated equally. We used Jain's fairness index, where a score close to 1 means high fairness (everyone waits similar times). Figure 7 shows RR is the fairest, with scores above 0.9.

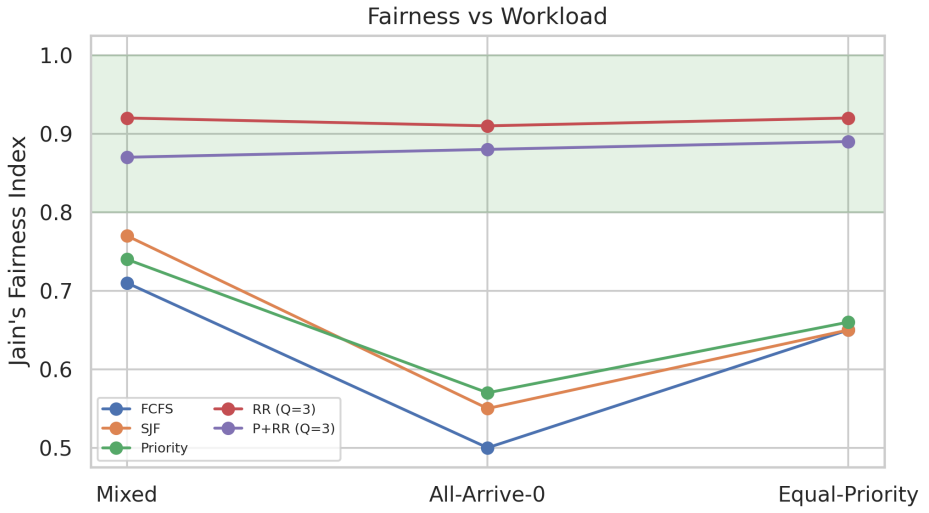


Figure 7: Jain's fairness index for each method. Green band (above 0.8) indicates high fairness.

$$\text{Jain}(x_1, \ldots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}, \qquad 0 < \text{Jain} \leq 1$$

Table 2: Synthetic workload used in all experiments.

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

A *radar chart* compares multiple metrics at once. Each axis represents a metric (e.g., waiting time, fairness), and a smaller area means better overall performance.

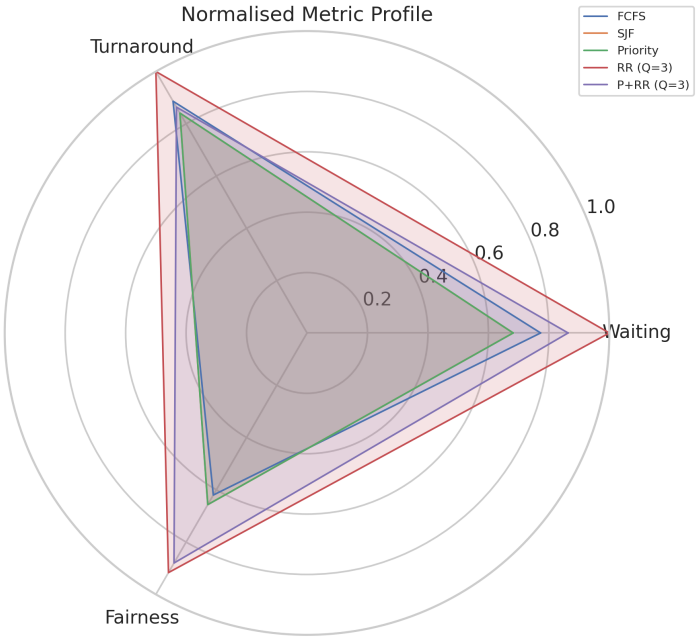


Figure 8: Radar chart comparing waiting time, turnaround time, and fairness. Smaller areas indicate better overall performance.

8 Discussion

Our tests show each method has unique strengths:

- **SJF** is like a chef who clears quick orders first, reducing average wait times but risking delays for big orders.
- **RR** is the fairest, like a chef giving every order equal attention, but it takes longer to finish due to constant switching.
- **P+RR** combines the best of both, prioritizing urgent tasks while ensuring fairness, cutting waiting times by about 13% compared to RR.

Choosing a method depends on what's most important: speed (SJF), fairness (RR), or a balance (P+RR).

9 Conclusion & Future Work

Our simulator makes it easy to see how different scheduling methods work. It shows that no single method is perfect and that each one has trade-offs between speed, fairness, and urgency. In the future, we could improve the simulator to:

- Simulate multiple CPUs .
- Test real-time scheduling for urgent tasks.

Acknowledgements

We thank **Prof. Youssef Iraqi** for his guidance through the course of *Operating Systems* at The *College of Computing of UM6P*, which helped us a lot to understand the main points and ideas behind how each of these algorithms work, which in itself helped us navigate through this assignment with ease.
Thank you Professor!