SYSC4001
Assignment 3

Report

Eric Cui – 101237617

Aydan Eng – 101295820

Part 1 GitHub

Part 2 GitHub

Introduction

The objective of this simulation was to evaluate how different scheduling strategies handle various workload characteristics, including CPU-bound tasks, I/O-bound tasks, and memory-constrained scenarios. The simulation was run across 20 different test cases.

Metrics

To compare the algorithms, we computed different metrics for the simulation results:

- *Throughput:* Number of processes completed per unit of time
- *Wait Time:* The amount of time a process spends in the ready queue waiting for the CPU
- *Turnaround Time:* The amount of time passed from process arrival to termination
- *Response Time:* The amount of time a process waits between an IO completion and the next CPU execution

Results and Analysis

All raw data for each test can be viewed in testing directory of our GitHub repository and all metrics for each test is recorded in final_metrics.csv

CPU-Bound Workloads (Tests 1–5)

Processes with lengthy execution bursts and little input/output were the focus of tests 1 through 5. The EP typically performed better in terms of TAT than RR in situations with concurrent heavy arrivals.

General Observation:

- EP / EP_RR: Prioritized high-priority tasks, resulting in lower Wait Times for specific processes
- RR: Treated all processes equally, increasing the Average Turnaround Time as no single process completed early

Analysis: Because of time-slicing, RR makes all processes in purely CPU-bound tests move slowly in parallel. The highest priority process can exit the system earlier thanks to EP's ability to finish quickly.

I/O-Bound Workloads (Tests 6–10, 12)

Processes with frequent I/O operations were the subject of tests 6–10 and 12. In mixed I/O environments, the EP_RR and EP algorithms demonstrated interesting results.

Data Highlight (Test 12 - High Prio I/O vs Low Prio CPU):

- RR: Average Wait Time: 30.0 ms | Average TAT: 88.0 ms
- EP / EP_RR: Average Wait Time: 20.67 ms | Average TAT: 78.67 ms

Analysis The Priority-based schedulers (EP and EP_RR) performed better in this case because the I/O-bound processes were given a higher priority, even though RR is usually good for I/O responsiveness. Instead of waiting in a Round Robin queue, they could interrupt the background CPU tasks and make sure they went straight to I/O.

Pre-emption & Responsiveness (Tests 14, 15, 17)

Tests 14, 15, and 17 involved processes requiring immediate pre-emption or crowded system management. The EP_RR algorithm proved to be best in handling high priority arrivals and high-load batch scenarios.

Data Highlight (Test 17 – Pre-emption Efficiency):

- RR: Average Response Time: 40.0 ms
- EP: Average Response Time: 90.0 ms
- EP_RR: Average Response Time: 0.0 ms

Analysis: Test 17 confirms that EP_RR correctly implements pre-emption. The 0.0 ms response time indicates that the moment the high-priority process arrived, it immediately pre-empted the running task. In contrast, EP forced the new process to wait for the current one to finish, causing significant delays.

Memory Contention (Test 21)

The relationship between CPU scheduling and memory management was shown in Test 21. In this configuration, a third process (P3) waited in the NEW queue while a high-priority process (P1) and a low-priority process (P2) occupied memory.

Data Highlight (Test 21 - Memory Blockage):

- EP: Average TAT: 865.0 ms. P1 ran to completion immediately (0–500ms), freeing memory at 500ms.
- RR: Average TAT: 998.33 ms. P1 and P2 shared the CPU via time-slicing, delaying memory release until approx. 900–1000ms.

Analysis: Memory throughput is directly impacted by the CPU scheduler selection. EP decreased the effective wait time for the process waiting for memory by enabling the "blocker" process to complete more quickly. RR kept memory busy for almost twice as long by forcing the blocker to share CPU time.

Convergence & Edge Cases (Tests 11, 13, 18–20)

Tests 11, 13, and 18–20 were examples of generic control scenarios in which the benefits of any one algorithm were not seen because of the workload characteristics like serial memory constraints or non-conflicting arrival patterns.

Data Highlight (Test 18 - Serial Execution):

- All Schedulers: Average Wait Time: 150.0 ms | Average TAT: 275.0 ms

Analysis: In these tests, regardless of the scheduling logic, the system was forced into a predictable execution path by external constraints (such as memory bottlenecks in Test 18) or standard sequential arrivals (Tests 11, 13). As a result, all three schedulers generated the same performance metrics, providing a baseline confirmation that the fundamental logic of process management operates consistently in all implementations.

Table 1: Summary of information found in metrics.csv which can be found in our GitHub

Workload Type	**Metric**	**Round Robin (RR)**	**Effective Priority (EP)**	**EP_RR**	**Winner**
I/O Heavy (Test 10)	Avg Wait (ms)	50.0	50.0	40.0	EP_RR
	Avg TAT (ms)	195.0	195.0	185.0	EP_RR
Mixed Prio (Test 12)	Avg Wait (ms)	30.0	20.7	20.7	EP / EP_RR
	Avg TAT (ms)	88.0	78.7	78.7	EP / EP_RR
Memory Stress (Test 21)	Avg TAT (ms)	998.3	865.0	865.0	EP / EP_RR

Conclusion

Overall, each algorithm demonstrated distinct strengths and weaknesses throughout our testing:

- **Round Robin** is the most "fair" approach. It guarantees that every process gets regular attention from the CPU, ensuring that no task is ever left waiting indefinitely just because it isn't considered high priority. However, this fairness comes at a cost: because the CPU is constantly switching attention, every individual task takes longer to finish (High Turnaround Time). As we saw in Test 21, this creates a backlog where memory partitions stay occupied for longer than necessary, blocking new processes from entering the system.

- **Effective Priority** is excellent for raw speed (throughput) and efficiency. By letting important tasks run to completion immediately, it clears out memory partitions very quickly. The major downside is its lack of flexibility; once a task starts running, it cannot be stopped. This creates a risk where a critical, high-priority arrival might get stuck waiting behind a long, less important task, destroying system responsiveness (as seen in Test 17).

- **EP_RR** feels like the best middle ground for a realistic operating system. It handles priorities effectively, ensuring urgent work is done immediately via pre-emption. At the same time, the inclusion of time-slicing ensures that lower-priority background jobs still receive regular processing time and are not completely ignored by the system. For a general-purpose system handling mixed workloads of interactive and background tasks, this balance makes EP_RR our top choice.