Assignment 3

For part 1 of this assignment we built a C++ program that simulates an Operating System scheduler. The goal was to manage processes by moving them between states such as New->Ready->Running->Waiting->Terminated and to assign them fixed memory partitions. We implemented three different scheduling algorithms to see how they differ from one another and which one would be the most optimal after analysing several results.

The algorithms that were implemented:

1. **External Priorities (EP):** A non preemptive algorithm where the lowest PID has the highest priority.

2. **Round Robin (RR):** A fair scheduling algorithm with a fixed time quantum of 100ms.

3. **EP with Preemption & RR:** A combined approach where the high-priority processes can kick out running processes, and equal-priority processes will share time slices.

During this part of the assignment, we executed 20 different simulations per algorithm which are represented by the test cases in the github input_files directory. We used a Python script to calculate all of the metrics such as throughput, average wait time, average turnaround time, and average response time to compare all of the algorithms numerically and see which would be most optimal according to the variety of tests we ran.

Comparison 1: CPU-Bound Processes (Fairness vs Speed)

In a self developed test scenario, we wanted to find out how the algorithms handle heavy computational loads. Our primary for this scenario was test case 6 in which the input was (**Figure 1**), two processes each 250ms and both arrived at 0ms. We see that the first process has a PID of 20 and the second process has a PID of 30. With the EP algorithm, it followed strict priority meaning that the PID 20 ran from start to end without ever stopping, while the PID 30 waited 250ms until it was able to start running. For RR, the scheduler treated both processes as equal and switched between both processes every 100ms as that was the time quantum. This resulted in both processes staying in the system longer because they were flip flopping turns every time.

```
20, 10, 0, 250, 0, 0
30, 10, 0, 250, 0, 0
```

(Figure 1: test_case_6.txt)

EP avg turnaround time: 375 ms
RR avg turnaround time: 475 ms

We can see that Round Robin increases the turnaround time for long cpu-bound processes. The overhead of performing context switches keeps both processes alive longer, whereas in

EP, it allows the first process to finish quickly, which reduces the turnaround time even though it is unfair to the lower priority process.

Time of Transition	PID	Old State	New State
0	20	NEW	READY
0	30	NEW	READY
0	20	READY	RUNNING
250	20	RUNNING	TERMINATED
250	30	READY	RUNNING
500	30	RUNNING	TERMINATED

(Figure 2: execution_case_6.txt [EP])

Time of Transition	PID	Old State	New State
0	20	NEW	READY
0	30	NEW	READY
0	20	READY	RUNNING
100	20	RUNNING	READY
100	30	READY	RUNNING
200	30	RUNNING	READY
200	20	READY	RUNNING
300	20	RUNNING	READY
300	30	READY	RUNNING
400	30	RUNNING	READY
400	20	READY	RUNNING
450	20	RUNNING	TERMINATED
450	30	READY	RUNNING
500	30	RUNNING	TERMINATED

(Figure 3: execution_case_6.txt [RR])

Comparison 2: Testing Responsiveness for Preemption

In another important scenario we decided to test how the system would handle the concept of urgent tasks, in this case we implemented a test case scenario that covers this (**Figure 4)**. In this case, a low priority process with PID 100 started running at time = 0ms, but a high priority process of PID 10 arrived later at time = 50ms. In EP, since it is non-preemptive, the high priority task was blocked until the low priority task completed its CPU burst. The RR improved compared to EP as the average wait time was 50 ms. Even though RR ignores priority, the 100ms time quantum interrupted the running processes at time = 100ms which resulted in a task to start sooner than in EP, but still forced the urgent task to be idle for 50ms while waiting for the time slice to expire. The EP+RR algorithm proved to be optimal since the average wait time was 25ms. This scheduler recognized the priority difference at time = 50ms and preempted the PID 100 right away.

```
100, 5, 0, 200, 0, 0
10, 5, 50, 50, 0, 0
```

(Figure 4: test_case_7.txt)

EP avg wait time: 75ms
RR avg wait time: 50ms
EP+RR avg wait time: 25ms

Time of Transition	PID	Old State	New State
0	100	NEW	READY
0	100	READY	RUNNING
50	10	NEW	READY
200	100	RUNNING	TERMINATED
200	10	READY	RUNNING
250	10	RUNNING	TERMINATED

(Figure 5: execution_case_7.txt [EP])

Time of Transition	PID	Old State	New State
0	100	NEW	READY
0	100	READY	RUNNING
50	10	NEW	READY
50	100	RUNNING	READY
50	10	READY	RUNNING
100	10	RUNNING	TERMINATED
100	100	READY	RUNNING
200	100	RUNNING	READY
200	100	READY	RUNNING
250	100	RUNNING	TERMINATED

(Figure 6: execution_case_7.txt [EP+RR])

Time of Transition	PID	Old State	New State
0	100	NEW	READY
0	100	READY	RUNNING
50	10	NEW	READY
100	100	RUNNING	READY
100	10	READY	RUNNING
150	10	RUNNING	TERMINATED
150	100	READY	RUNNING
250	100	RUNNING	TERMINATED

(Figure 7: execution_case_7.txt [RR])

Comparison 3: Response Time

Another metric we wanted to cover was prominent within test case 17, which simulated a workload where a high priority CPU-bound process competes with a lower priority I/O bound process. We measured the average response time as a critical metric in this case.

Our results concluded that both EP & EP+RR completely ignored the lower priority process until the higher one finished or timed out. This resulted in a poor **average response time** of 100ms. The RR provided the best performance in this case with an **average response time** of 50ms. This showcases a major strength with RR, which is that it treats all processes equally regardless of priority. Since RR gave the I/O bound process a time slice early, it allowed it to issue an I/O request and enter the waiting state while the CPU worked on the other process. This confirms the statement that RR is superior for working with workloads where avoiding starvation is important than sole priority.

Memory Management Analysis (BONUS)

In addition to the scheduling logic, we implemented a memory tracking feature that logs the status of the six fixed partitions of memory every time a process is successfully added to the system.

For our sample analysis we decided to discuss test_case_20.txt. In this specific test case there are five processes each of 1MB and they all arrive at time = 0ms. The memory logs show that the simulator successfully loaded all five processes at the same time. According to the results, we can clearly see that the partitions were filled in reverse order, which means that we correctly implemented a best-fit strategy in this case. When we check the smallest partitions first, the system preserves space for potential larger processes that may arrive later.

Conclusion

In this project we successfully simulated three different scheduling algorithms and analyzed their performances across 20 different simulations. We analysed each output to see how each algorithm performed and whether there was a clear best algorithm to use. From analyzing this data we can clearly see that the EP algorithm provides the best throughput for high priority tasks but suffers from starvation, since there are a lot of low priority processes that are forced to wait extended periods of time. The RR algorithm is the most fair and provides the best response time for tasks, but it is inefficient for long CPU jobs due to a lot of overhead from context switching. The EP+RR algorithm offers a combined balance. By allowing there to be preemption it solves the blocking issue of the EP algorithm, and by using priorities it avoids the inefficiency of pure RR for important tasks.

Overall, the EP+RR algorithm proved to be the best for more complex scenarios as we have discussed in the comparisons above.

Github Repo: https://github.com/yuvraajbains/SYSC4001_A3_P1