SYSC4001 Assignment 2 Part 3 Report

11/7/2025

Kushal Poudel - 101298706
Amit Kunkulol - 101260631

Link to repository:

https://github.com/KushalPoudel6/SYSC4001_A3_P1

Explanation of implemented code:

First, the template code initializes everything and then populates the ready queue with incoming jobs.
Then, in the wait queue, it checks if I/O is done and the process can move to ready state, and if it can, it adds it to the ready queue and removes it from the wait queue.

Then the scheduler is called, different depending on which one is demanded:

External priorities: Sorted the jobs based on the PID, lower PID = higher priority, so the function that implemented this was similar to the given FCFS function but instead of arrival time, used PID.
Round robin: The scheduler for this one was essentially the same as FCFS but now implemented a time quantum where if the time ran out, it would proceed to the next job and kick out the previous job.
Round robin with external priorities: Same as regular round robin but instead uses the EP as the scheduler, but still implements the time quantums.

Then, it executes where if the CPU is idle and there are processes in the ready queue, it runs the next process.
If the cpu is already running, it decrements the amount of time left for the process and checks if the job is complete (and for the two round robin cases, checks if the quantum is complete). It also then checks if its time for I/O in which case the state goes to WAITING and pushes it to the wait queue.
The time is incremented as the loop continues and the status is added at each step, then after the loop is done, some metrics (which I added a function for in .hpp) are added as well.

Simulation results

For the simulations, I tested various things. The first 4, I used the ones given to the class, testing a number of processes with I/O or no I/O. The next 6 follow this trend with odd numbers being no I/O and even numbers being with I/O with increasing number of processes.
For 11-14, I had varying arrival times, (all at once, none for a while, different orders etc.) The remaining 6 are semi random.
Note. 1-10 have too large of quantums to see differences in schedulers, but 11-20 have longer total cpu time so there will be some preemption with the quantum of 100ms.
Here's the table for the first 4 cases:

Case	Throughput	Turnaround	Wait	Response
1	0.1	10	0	0
2	0.091	11	1	0
3	0.133	11	3.5	3.5
4	0.142	9.5	4.5	0

The cases were the same across the 3 schedulers as the large time quantum made it so there was no need for preemption.

Here are some random sample cases:
Case 12:

Scheduler	Throughput	Turnaround	Wait	Response
EP	0.00349	455	170	58.5
RR	0.00351	558.5	273.5	58.5
EP + RR	0.00347	446.5	161.5	48.5

Case 20:

Scheduler	Throughput	Turnaround	Wait	Response
EP	0.0028	956.5	599	455
RR	0.0028	970	612.5	430
EP + RR	0.0028	970	612.5	430

Analysis of results

The results show various things:
Throughput, which measures the number of processes per unit time is generally not affected by the schedulers, some of the results show slightly varying throughputs but they are due to the final completion times varying but overall not much of a difference.
This indicates they *generally* all use the CPU equally well over the long run.

Turnaround time, which measures the total time the process stays in the system seems to vary between different cases.
In case 12, for example, EP +RR is much lower due to the lower PID (higher priority) job being shorter and being completed quickly before doing the next one. RR is the worst in case 12 since the forced rotation delays the shorter job being finished, since there is no priority system based on job length. In case 20, EP performed the best since there is less context switching allowing whichever process has started to finish faster. Generally to optimize turnaround time, you want the shortest jobs to finish faster, but since the different priorities care more about arrival time or PID, which aren't affected by the job length, it's more up to luck for which process finishes the fastest.

For wait time, (the total time a process waits in the ready queue before completion), we can see EP and EP + RR performing better in case 12 since the priorities happen to correspond with job length, and adding the non-preemptive aspect of RR, makes EP+RR the best. but in case 20, EP+RR performs equally bad as RR since the priorities don't align with anything useful. Generally, RR performs better than non-premtive tasks because they prevent the jobs from blocking the CPU entirely, however if you can prioritize the shortest jobs to complete first, then that's best for the average wait time.

For average response time, RR and EP+RR are almost always better, since it measures the time from when a job enters the ready queue to when it first gets to the CPU and the rotation of RR allows for more jobs to get started faster. This can be seen in the 2 sample test cases above as well as others.

In terms of CPU-Bound vs I/O bound processes, the first 4 cases do not help us in finding the effectiveness of the different schedulers as the quantum is too large. However when comparing the results, I/O bound workloads seem to be better for performance like turnaround time due to them giving up the CPU for other tasks.

Generally, the other cases also follow this analysis.

Ultimately, which scheduler you use will depend on your use case, where if you have high priority tasks, then you might want EP, and if you are also concerned about starvation then EP + RR would be used. RR could be used if you have a system where all jobs are equally important and want it to be more responsive.

Bonus

Although I did not implement it, I can expect some external fragmentation where the total amount of unused memory space that is broken into small blocks, with it increasing over the course of the simulation.
Since we are using a fixed-partition scheme, when a process arrives, it must be loaded entirely into one partition that is equal to or larger than its size, meaning it could also cause some internal fragmentation issues. (when a process is smaller than the partition it is loaded into) The wasted space inside the partition can’t be used by any other process.

In terms of the schedulers, the non-preemptive EP may cause the system to reserve a large block for it which could potentially cause starvation from other jobs not being able to fit in smaller spaces. The preemptive schedulers would allow small processes to get memory and finish faster which could lead to reduced fragmentation.

Conclusion

In conclusion, the three different schedulers have been implemented, and with 20 cases different strengths and weaknesses of the schedulers and I/O bound vs CPU bound cases have been explored.