SYSC4001 - Assignment 3 Part I Report

Zaineb Ben Hmida - 101302936
Sohaila Haroun - 101297624
https://github.com/sohaila-cu/SYSC4001_A3_P1.git

Bonus Memory Recordings

We did implement the bonus memory use/status recordings/logs in our simulator. Every time a change was made in the memory allocations (new process added, memory freed, memory error) our simulator logged it and put it in the memory.txt file. We also made sure to handle the case in which a process enters the system and there is no partition that it will fit in. We did that by setting that process's state to NEW, adding it to the job_list and outputting an error message in the memory log, and implementing a segment that checks for any NEW processes in the job_list and if they can be allocated memory after every iteration. Our test case 7 was a very good and simple example to exemplify these additions.

test8.txt:
```
10, 40, 0, 8, 0, 0
14, 40, 3, 5, 0, 0
```

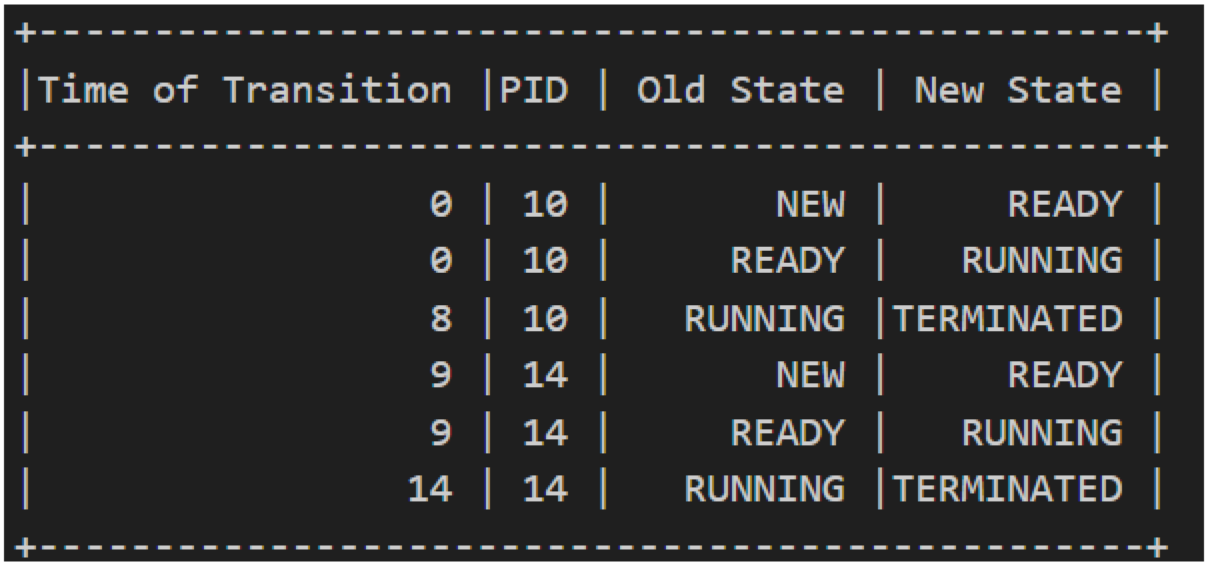
Memory_test8.txt: (copy pasted cuz its long, but you can check it out)

```
Current Time: 0
+----------------------------------+
|    # | Size |  PID | Used |Unused |
+----------------------------------+
|    1 |   40 |   10 |   40 |     0 |
|    2 |   25 |   -1 |    0 |    25 |
|    3 |   15 |   -1 |    0 |    15 |
|    4 |   10 |   -1 |    0 |    10 |
|    5 |    8 |   -1 |    0 |     8 |
+----------------------------------+
Total free memory available for use (without fragmentation): 58

Current Time: 3
+----------------------------------+
|    # | Size |  PID | Used |Unused |
+----------------------------------+
|    1 |   40 |   10 |   40 |     0 |
|    2 |   25 |   -1 |    0 |    25 |
|    3 |   15 |   -1 |    0 |    15 |
|    4 |   10 |   -1 |    0 |    10 |
|    5 |    8 |   -1 |    0 |     8 |
+----------------------------------+
Total free memory available for use (without fragmentation): 58
Memory Allocation Failure Occured, Process will wait until memory is available.

Previous Memory Allocation Failure Resolved, memory was available.
Current Time: 9
```

```
+----------------------------------+
|   # | Size |  PID | Used |Unused |
+----------------------------------+
|   1 |  40 |  14 |  40 |    0 |
|   2 |  25 |  -1 |   0 |  25 |
|   3 |  15 |  -1 |   0 |  15 |
|   4 |  10 |  -1 |   0 |  10 |
|   5 |   8 |  -1 |   0 |   8 |
+----------------------------------+
Total free memory available for use (without fragmentation): 58
```

As you can see in the execution file, the time that memory became available and process 14 could fit is the time that process 10 terminated.

Execution_test8.txt:



```
+------------------------------------------------+
|Time of Transition |PID | Old State | New State |
+------------------------------------------------+
|                 0 | 10 |       NEW |     READY |
|                 0 | 10 |     READY |   RUNNING |
|                 8 | 10 |   RUNNING |TERMINATED |
|                 9 | 14 |       NEW |     READY |
|                 9 | 14 |     READY |   RUNNING |
|                14 | 14 |   RUNNING |TERMINATED |
+------------------------------------------------+
```

There were other interesting observations relating to the memory management portion of the simulator.
Overall in most of our testcases, for all of our algorithms, there are almost no differences to the overall management of memory, especially when looking at the overall metrics like how much space was used vs unused and how many errors occurred. There were slight differences in timings just due to the fact that they are scheduling algorithms so their schedules were different. We actually had to put it in effort to get differing results, but because our memory management and scheduling algorithms only really do anything after processes enter the ready_queue and there is no long-term or mid-term scheduler, this portion was very simple and had no major deviations or observations that were notable enough to make. I did definitely find it interesting how little it made a difference.
There were obviously very big differences in the memory logs when comparing different test cases that have differing numbers of processes and/or differing sizes of programs. For example the above example 8, is comparable to its predecessors 6 and 7, whom either have the same

exact process or the same number of processes with just one size difference. They had very different memory logs as they did not have any memory errors, and this was the case for all 3 of them for all three (or four if you include my slightly different EP_RR) of the scheduling algorithms.

(note that I will try to reference the specific test case numbers or files in this report, however I am running out of time and they are named descriptively including whether they are IO/CPU -bound or mixed and which algorithms were used)

Another note: it wasn't very clear in the assignment text if "A combination of both: external Priorities *including preemption* and a 100ms timeout, in Round-Robin fashion", was instructing us to have EP that has the preemption and 100ms timeout properties of RR or also has its own preemption, so I coded both. EP_RR is assuming both algorithms are preemptive, meaning if a process is in the middle of executing during its quantum, and a higher priority process joins the ready queue, it will be preempted, and if its quantum is over, it will also be preempted. In EP_RR1, processes are only preempted when their quantum is over.

That on its own led to interesting results, as this seemingly small detail led to differing execution files from the same test cases. Notably, testcase 7 for example, where the lower priority task kept getting interrupted by the higher priority task finishing its IO, which was a lot more frequent than the quantum interrupting it in the other variation of the algorithm. This led to higher wait times, as expected from a round-robin algorithm, as well as high overhead because the IO frequency is not carefully selected like the quantum so that it is not so often that the overhead caused by the interrupt is minimizing the other benefits of this algorithm.

In some of the IO-bound test cases, all of the processes were IO bound (like IO_3.txt) and so the overall performance …
However I also included test cases where most processes were long IO bound processes and only 1 or 2 were short IO-bound or even balanced or CPU bound(IO_2,4,5). It was very interesting to see how very differently the algorithms dealt with the same scenarios, especially when EP had the same set of processes but their PIDs were shuffled around, which ended up drastically impacting the wait times and turn around times for the processes. This was due to the fact that I purposely created a scenario where the longer processes had higher priority than the short ones, intentionally creating the convoy effect which led to higher average wait times and higher average turnaround times.

For test cases with a mix of both IO-bound and CPU-bound processes, the average response times drastically improved when using RR compared to EP because the IO bound processes kept getting delayed for long periods of time behind the CPU-bound processes that would not leave the CPU often (or at all for those without IO). For CPU-bound processes with very long process times however, this caused a lot of overhead, especially when I made the quantum shorter, because they kept getting interrupted. The EP-RR combination was more fair to both kinds of processes when they were together, assuming I did not intentionally manipulate the priorities as mentioned above.

In scenarios where there were many processes and I intentionally made the first few have very low priorities (high PID), they experienced considerable starvation when using the EP algorithm, despite this not being a system that gets continuous input, which I did expect but not to the extent I saw. (could be related to the extreme numbers I was putting tho…) Nevertheless, it exemplified well the limitations of EP algorithms or other algorithms that depend on prioritization like SJF. This was, however, well balanced by the EP-RR algorithm, in the case that those processes' processing time were larger and on a similar scale to the quantum.

Overall Analysis:

Across the 20+ simulation scenarios for each scheduler, several consistent patterns emerged regarding throughput, fairness, and process responsiveness:

External Priority (Non-Preemptive)

This scheduler produced the highest turnaround and waiting times for low-priority processes. In CPU-bound scenarios, high-priority long jobs monopolized the CPU, causing severe starvation for lower priority tasks. Throughput decreased when high-priority jobs were long, since lower-priority processes could not begin execution. Response time was also poor for most processes, as no preemption meant processes had to wait until the full CPU burst of a higher-priority process completed.

Round Robin (100 ms Time Quantum)

Round Robin provided the best overall response time due to frequent preemption. This made it especially effective for I/O-bound workloads, where processes frequently blocked for I/O and returned quickly to the ready queue. Throughput was stable across all workloads, though slightly lower for CPU-bound tasks because of context-switch overhead. Wait and turnaround times were moderate—better than External Priority, but not as optimized for high-priority long tasks. RR also eliminated starvation entirely.

External Priority + Round Robin (Preemptive)

The combined scheduler balanced fairness with priority control. High-priority tasks received preferred access to the CPU, while preemption prevented lower-priority tasks from starving. Across mixed workloads, this scheduler consistently achieved better throughput than pure External Priority and better turnaround/wait times than Round Robin for high-priority processes. Response time was not as fast as plain RR but significantly improved over non-preemptive EP. Overall, this algorithm handled all workload types more evenly, making it the most balanced of the three.

Overall Comparison

Best for CPU-bound workloads: External Priority + RR

Best for I/O-bound workloads: Round Robin

Worst for fairness: External Priority

Best response time: Round Robin

Most balanced overall: External Priority + RR

This analysis indicates that no single algorithm dominates in every category, but the combined approach offers the most consistent performance across diverse workloads.