Assignment 3 - Part I - Report
SYSC 4001

Student 1: Tharsan Sivathasan (101081721)
Student 2: Samy Kaddour (101258499)

GitHub Repository:

https://github.com/tharsan18/SYSC4001_A3_P1

Introduction

The objective of this assignment was to create and analyze three different CPU scheduling algorithms, the three being; Enhanced Priority (***EP***), Round Robin (***RR***), Enhanced Priority Round Robin (***EP RR***). After the creation of these CPU algorithms, we ran several tests and analyzed the schedulers' performances under a variety of different use cases (*the use cases can be found under input_files, the results to analyze are under output_files*).

Setting up the CPU Schedulers

Each scheduler has a similar build structure, as they each followed the following flow:

> Every millisecond, the simulation would
>
> > Update the CPU burst time → Decrement the I/O timers for processes in WAITING State → Return the completed I/O requests to READY states→ Apply each of the scheduling algorithm rules (*this is the part that differed between each scheduler*) → Log every state transition → and tracks/calculate metrics

For I/O handling, the process enters WAITING state on an I/O request, and will return to READY state after *io_duration.* This was used to calculate response time using the average time between consecutive I/O requests per every process.

The metrics tracked and calculated were the following, and was done through code our code;

> **Turnaround Time**, which was found through ***completion time - arrival time***
> **Waiting Time**, which was found through ***the use of READY state tracking loops***
> **Throughput**, which was found through ***total processes / total simulation time***
> **Response Time (*explained above*)**

RESULTS

A cumulative of 60 tests were run, 20 tests per CPU Scheduler. For this report, we will focus on just 5 of these tests:

> ***Test 1 being a CPU-Bound scenario***
> ***Test 10 being a mix workload with moderate amounts of CPU burst scenario***
> ***Test 15 being heavy CPU-Bound scenario, with different arrivals***
> ***Test 20 being a multiprocess workload scenario***

Below are the tables comparing the outputs from the different schedulers for each test listed above.

Test 1 Results

Scheduler	Avg Turnaround	Avg Waiting	Throughput	Avg Resp Time
EP	*85.25*	*53.75*	*0.031008*	*0*
RR	*85.25*	*53.75*	*0.031008*	*0*
EP_RR	*85.25*	*53.75*	*0.031008*	*0*

Test 10 Results

Scheduler	Avg Turnaround	Avg Waiting	Throughput	Avg Resp Time
EP	*271.80*	*192.20*	*0.012376*	*0*
RR	*288.50*	*187.50*	*0.009780*	*0*
EP_RR	*313.00*	*214.40*	*0.010020*	*0*

Test 15 Results

Scheduler	Avg Turnaround	Avg Waiting	Throughput	Avg Resp Time
EP	*313.00*	*214.00*	*0.010020*	*0*
RR	*288.50*	*187.50*	*0.009780*	*0*
EP_RR	*313.00*	*214.40*	*0.010020*	*0*

Test 20 Results

Scheduler	Avg Turnaround	Avg Waiting	Throughput	Avg Resp Time
EP	*271.80*	*191.80*	*0.012376*	*0*
RR	*271.80*	*191.80*	*0.012376*	*0*
EP_RR	*302.80*	*223.20*	*0.012376*	*0*

Analysis

First let's take a look at the test 1 results. Every single metric across all the scheduling types holds the same value. This is because for this test, we used CPU bursts that are lower than 100 ms (which was the quantum we chose). There were also no I/Os in this test. This just shows that when we have no I/Os, and bursts shorter than the quantum, all schedulers will act as a FCFS. Thus no scheduling type here is more efficient than the other.

The test 10 results tell a different story. Every single metric had a different value. Here the CPU bursts were longer, and job overlapping while still maintaining the no I/O concept. This helps us observe how the schedulers actually work (priority wise).

EP had the lowest turnover time → the higher priority processes finished earlier

Average response time was 0 → as we did not have an I/Os

RR had lowest wait time → CPUs time was spread

EP_RR had performed the worst, due to the fact that there was a lot of context switching being done.

For test 15, we aimed to test heavy CPU-Bound workload, meaning longer CPU Bursts, and a continuation and different arrivals of processes. What was noticed was that EP and EP_RR had very similar metrics, showing that EP AND EP_RR allowed the higher priority arrivals to run uninterrupted. RR did not enforce this, and instead showed a more fair priority spread, thus the lowest wait time. What these results show is how the Enhanced Priority scheduling create this starvation effect.

Lastly, there is test 20. This Test included a larger multi process workload. The results here indicate that EP and RR performed identically, as they had the same metrics throughout the board. The reason for this was because the quantums were larger than the bursts. This created a FCFS effect, similar to test 1. The reasons EP_RR did not behave like this was because the preemptions caused READY queue reshuffling which would interrupt and cause a longer waiting time.

In conclusion, EP optimizes turnaround for high priority jobs, but will cause starvation to other jobs. The RR minimizes waiting time, but will sacrifice the turnaround time for this to happen. Lastly EP_RR is very responsive, but will consume the most resources. I also performs the worst for mixed workloads.

BONUS

We used memory in the simulation using assign_memory() and free_memory(). The output system status files are located in the output_files folder on the repository called System_status_EP_test10.txt. You are able to see that once a process is added to the ready queue it is added to main memory (would be running queue if there were more processes than available memory frames).

```
time: 6: PID: 3, Mem Size: 10, Arrival T: 6, CPU T: 100, IO Freq: 400, IO Dur: 5

+-------------------------------------------------------------------------------+
| PID |  Partition | Size | Arrival Time | Start Time |Remaining Time |      State |
+-------------------------------------------------------------------------------+
|   1 |          3 |   15 |            0 |          0 |           120 |    RUNNING |
|   2 |          2 |   15 |            3 |         -1 |           110 |      READY |
|   3 |          4 |   10 |            6 |         -1 |           100 |      READY |
+-------------------------------------------------------------------------------+
```

Then once the process is terminated it is unallocated from the partition it was using freeing it up for use with another partition. With a test case that has tens of entries you would then be able to see the partition changing as the running program goes to the waiting or ready state not being used and another process needs the memory.

```
time: 299: PID: 3, Mem Size: 10, Arrival T: 6, CPU T: 100, IO Freq: 400, IO Dur: 5

+-------------------------------------------------------------------------------+
| PID |  Partition | Size | Arrival Time | Start Time |Remaining Time |      State |
+-------------------------------------------------------------------------------+
|   1 |          3 |   15 |            0 |          0 |           120 |      READY |
|   2 |          2 |   15 |            3 |        100 |           110 |      READY |
|   3 |         -1 |   10 |            6 |        200 |             0 | TERMINATED |
|   4 |          5 |    8 |            9 |         -1 |            90 |      READY |
|   5 |          6 |    2 |           12 |         -1 |            80 |      READY |
+-------------------------------------------------------------------------------+
```

We can see in the RR file that every 100 millisecond even though the process is pre-empted nothing happens to the memory as in the test case we have for that is only 5 processes so the frames are enough to hold all the processes at once. This simulator was using first fit and because of that there are no need for extra time as all of the processes found available memory frame.

What we noticed with this addition was the following:

CPU-Bound workloads will reflect a more steady memory usage

I/O-Bound workloads lead to a constant amount of deallocations and reallocations (if there are more processes than memory frames available). This causes an increase in fragmentation.

Memory turnover is increased when the amounts of active processes are increased. This is because there are more state transitions, thus we are allocating and deallocating more and more. According to Belady's Anomoly, paradoxically the page replacements increase as the number of available frames increase. We have 6 available frames to use and we are not putting multiple processes in one frame but we have no page replacements as our test case shown has sufficient memory.