SYSC4001 Assignment 3 - Part 1 (Report)

Authors: Aj Donald 101259149, Jayven Larsen 101260364

Information about test traces:

Trace 1 (input\_data\_1.txt) : Given trace from assignment

Trace 2 (input\_data\_2.txt): Given trace from assignment

Trace 3 (input\_data\_3.txt): Small set (5 processes)

Trace 4 (input\_data\_4.txt): Medium set (10 processes)

Trace 5 (input\_data\_5.txt): Large set (15 processes)

Trace 6 (input\_data\_6.txt): High I/O Intensity (8 processes)

Trace 7 (input\_data\_7.txt): High CPU Intensity (6 processes)

Trace 8 (input\_data\_8.txt): Staggered Arrival Times for processes (10 processes)

Trace 9 (input\_data\_9.txt): Small I/O Durations (7 processes)

Trace 10 (input\_data\_10.txt): Mixed (9 processes)

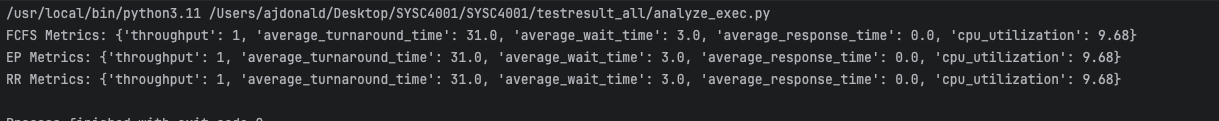
**Background**: For the purposes of analysis, 10 input traces were tested, each test varies in their number of processes, how CPU or I/O intensive they are. Additionally, to investigate the different schedulers we added test cases with staggered arrival times for processes as well input data with a mix of processes with large IO frequencies or more CPU execution time.

**Analysis of Execution Results**: In terms of analysis for the output execution files, a few metrics were considered. Namely, throughput, average turnaround time, average response time, average wait time and CPU utilization. The following screen captures from the resulting calculations of each performance metric done via python, this is to help visualize the results of each scheduler for each input trace. Note, given that schedulers were all subjected to testing using the same input trace at the same time, the throughput of each scheduler is identical in all scenarios.

1. Input\_data\_1.txt



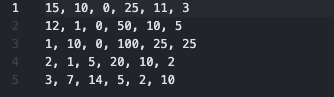
*Figure 1: input\_data\_1.txt contents*



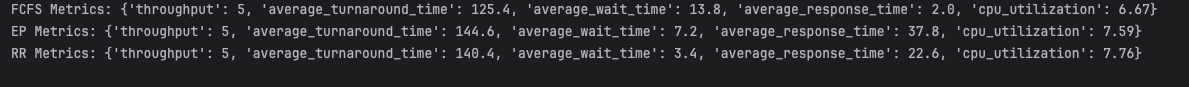
*Figure 2: Results from analyze\_execution.py for input\_data\_1.txt*

Given that the input trace in this scenario is quite simple, consisting of only 1 process, the output of each implementation is identical. As a result, each of their performance metrics are identical regardless of the scheduler.

2. Input\_data\_2.txt



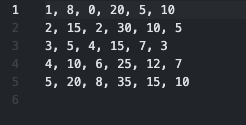
*Figure 3: input\_data\_2.txt contents*

**

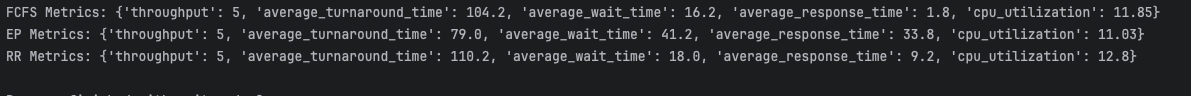
*Figure 4: Results from analyze\_execution.py for input\_data\_2.txt*

As expected, the resulting metrics from each scheduler abides by the expectation. Where, a First Come First Serve (FCFS) scheduler performs well for relatively shorter traces, with a short turnaround time and response time, but is outperformed in terms of CPU utilization and wait time for each process. Similarly, the Round Robin (RR) scheduler limits the wait time experienced by each process, while also promoting a strong CPU utilization. Lastly, for this particular trace the external priority (EP) scheduler is outperformed by its two counterparts in all metrics with the exception of CPU utilization, where it slightly outperformed the FCFS scheduler.

3. Input\_data\_3.txt



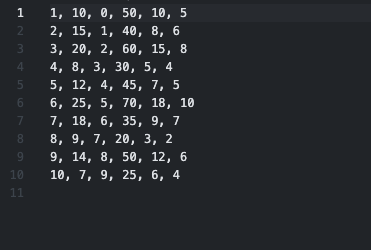
*Figure 5: input\_data\_3.txt contents*

**

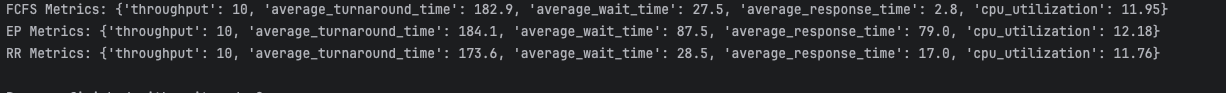
*Figure 6: Results from analyze\_execution.py for input\_data\_3.txt*

In this particular scenario, the goal was to choose a trace of the same length as the previous example to show how different factors can impact performance metrics of different schedulers. For instance, in this example we see the EP scheduler yielded the best turnaround time but a poor response time. Additionally, the CPU utilization can vary depending on the test case.

4. Input\_data\_4.txt



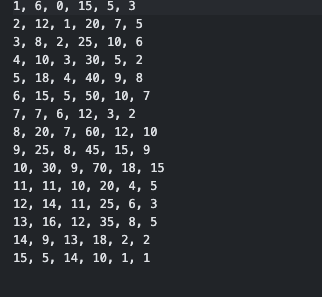
*Figure 7: input\_data\_4.txt contents*



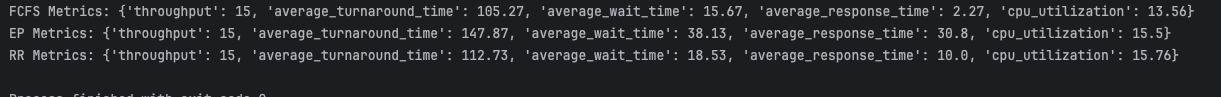
*Figure 8: Results from analyze\_execution.py for input\_data\_4.txt*

This particular trace was designed to be of “medium” length, with a total of 10 processes, varying in IO frequency and overall cpu duration. Nonetheless, this particular example showcases once again the variability in which scheduler performs the best. Each scheduler having their strengths, the RR implementation was particularly performant in terms of turnaround time (TAT). Which FCFS and EP schedulers performed well in terms of average wait time (AWT), average response time (ART) and CPU utilization.

5. Input\_data\_5.txt



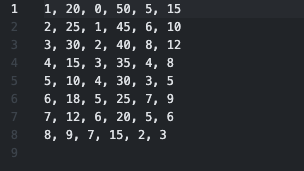
*Figure 9: input\_data\_5.txt contents*

**

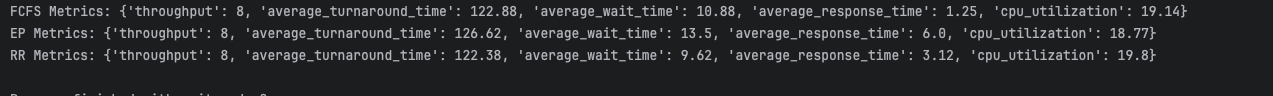
*Figure 10: Results from analyze\_execution.py for input\_data\_5.txt*

This trace was designed to be a large set of processes (15) with varied characteristics to showcase how different schedulers perform with large volume. In this particular scenario, the EP scheduler had the weakest performance across all of the implementations. While the RR scheduler and the FCFS had comparable performance with the RR having the strongest CPU utilization but slightly weaker metrics than the FCFS scheduler otherwise.

6. Input\_data\_6.txt



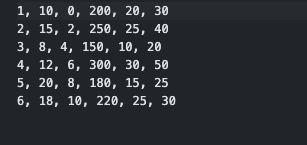
*Figure 11: input\_data\_6.txt contents*

**

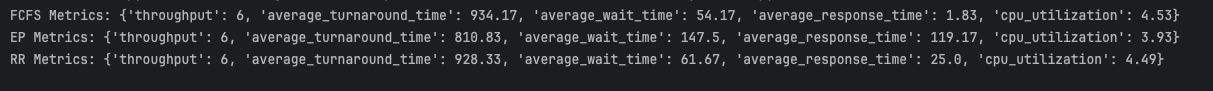
*Figure 12: Results from analyze\_execution.py for input\_data\_6.txt*

This trace was intended to highlight how different schedulers behave given a set of processes which are I/O intensive. As expected, the RR scheduler with a 100ns time quantum performed ideally for this test case. As seen in the results, the TAT, WT and CPU utilizations from the RR scheduler outperformed its counterparts while also remaining competitive for the ART. The benefit of the RR implementation is very much felt in cases where processes are constantly swapping between I/O since their time quantum allows them to use the CPU when needed and frequently swap to IO.

7. Input\_data\_7.txt



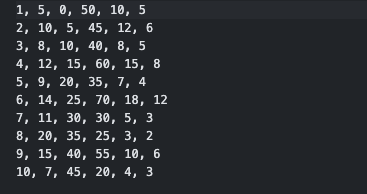
*Figure 13: input\_data\_7.txt contents*

**

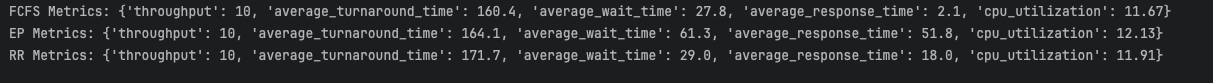
*Figure 14: Result from analyze\_execution.py for input\_data\_7.txt*

Contrary to the previous example, this trace was designed to show of these schedulers handle CPU sets of CPU intensive processes. The resulting output from the program demonstrates much of the expected behaviour from each scheduler. The EP implementation yields the lowest TAT but also corresponds to the highest AWT and ART and poorer CPU utilization. Additionally, the FCFS scheduler provides quickest response time and strong CPU utilization. Lastly, the RR scheduler performs more comparably to the FCFS scheduler with a few discrepancies in ART and AWT while remaining very similar in terms of CPU utilization and TAT.

8. Input\_data\_8.txt



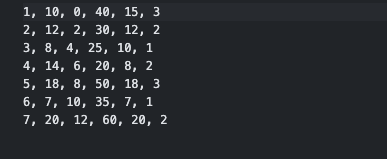
*Figure 15: input\_data\_8.txt contents*

**

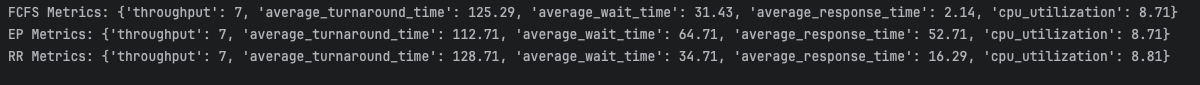
*Figure 16: Results from analyze\_execution.py for input\_data\_8.txt*

This trace has 10 processes with staggered arrival times to showcase how different schedulers will handle this via the resulting response times. As seen in the figure above, the TAT and CPU utilization for each scheduler is remarkably similar, while the AWT for RR and FCFS defer from the EP scheduler.

9. Input\_data\_9.txt



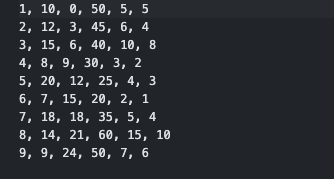
*Figure 17: input\_data\_9.txt contents*

**

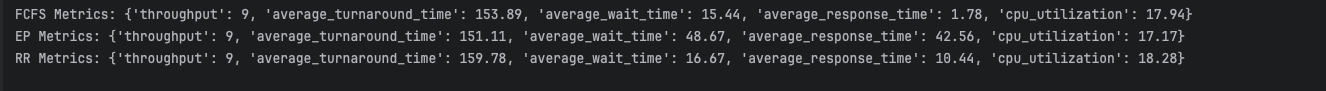
*Figure 18: Result from analyze\_execution.py for input\_data\_9.txt*

This particular trace had 7 processes with short IO durations. As, we see that the CPU utilization of each scheduler is nearly identical, which is to be expected. Nonetheless, the EP implementation featured the best TAT but poor AWT and ART while the RR and FCFS had similar metrics compared to one and other.

10. Input\_data\_10.txt



*Figure 19: input\_data\_10.txt contents*

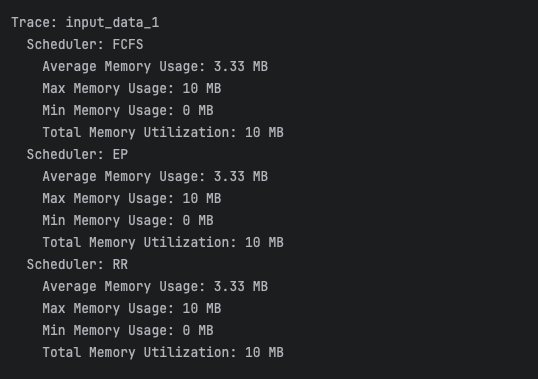
**

*Figure 20: Result from analyze\_execution.py for input\_data\_10.txt*

This trace was designed to a mix of different characteristics such as various IO frequencies, durations and CPU durations. As a result of having arbitrarily chosen these data points, we expect to see the true behavior of these schedulers. The results received remain consistent with both previous experiments but also the expectation for these implementations. For instance, the CPU utilizations of these schedulers are all quite comparable for the given trace, while also yielding a similar TAT. Lastly, we see the EP implementation provides the largest average waiting time, follows the logical expectation of External Priority.

**Analysis of Memory Results (bonus):** Similarly to when performance metrics for execution were evaluated, a python script was developed (analyze\_memory.py) which can be found in the submission. Nonetheless, the metrics considered were Average Memory Usage (AMU), Max Memory Usage (MAU), Minimum Memory Usage (MIU) and Total Memory Utilization (TMU). It is important to note that since each scheduler was subject to testing using the same input trace simultaneously, the MAU and MIU are identical in each scenario.

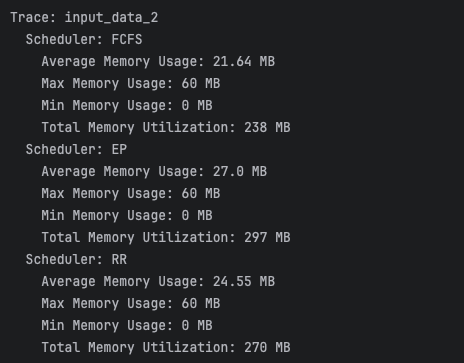
1. Input\_data\_1.txt



*Figure 21: Result from analyze\_memory.py for input\_data\_1.txt*

As seen in the figure above and similarly to the execution file, memory usage metrics are identical for each of the different schedules. Since there’s only 1 process within the input trace, the Total, Average, Maximum and Minimum memory utilization does not vary between implementations.

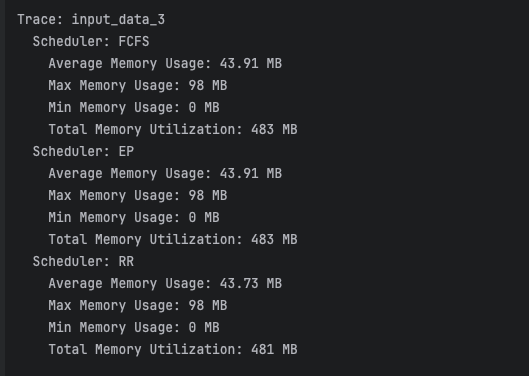
2. Input\_data\_2.txt



*Figure 22: Result from analyze\_memory.py for input\_data\_2.txt*

In this second test case, we see the maximum memory consumption is consistent between each process but FCFS performs the best in terms of average memory usage (AMU) and total memory utilization (TMU). Additionally we understand that external priority schedulers will consume more memory overall and on an average basis.

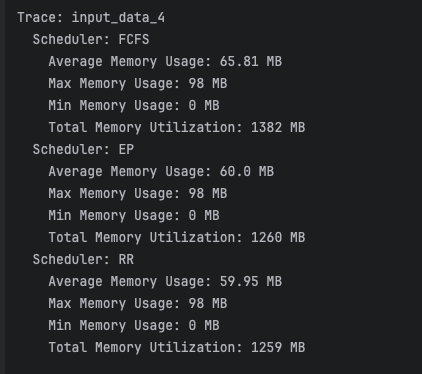
3. Input\_data\_3.txt



*Figure 23: Result from analyze\_memory.py for input\_data3.txt*

In this example, where a small set of small processes was considered, each scheduler performs quite similarly, all yielding a very comparable if not nearly identical result for all metrics. This is to be expected for small traces and similar behavior was observed in subsequent testing.

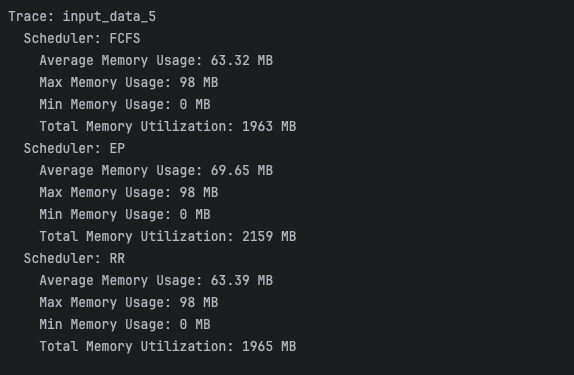
4. Input\_data\_4.txt



*Figure 24: Result from analyze\_memory.py for input\_data\_4.txt*

The trace used in this example consisted of 10 processes which is considered medium length. From the results show above, we can immediate start to observe the memory ineffiencies related to a FCFS scheduler, consuming considerably more memory on average and in total. On the other hand, we see the improved TMU and AMU in both the EP and RR schedulers which yield similar results.

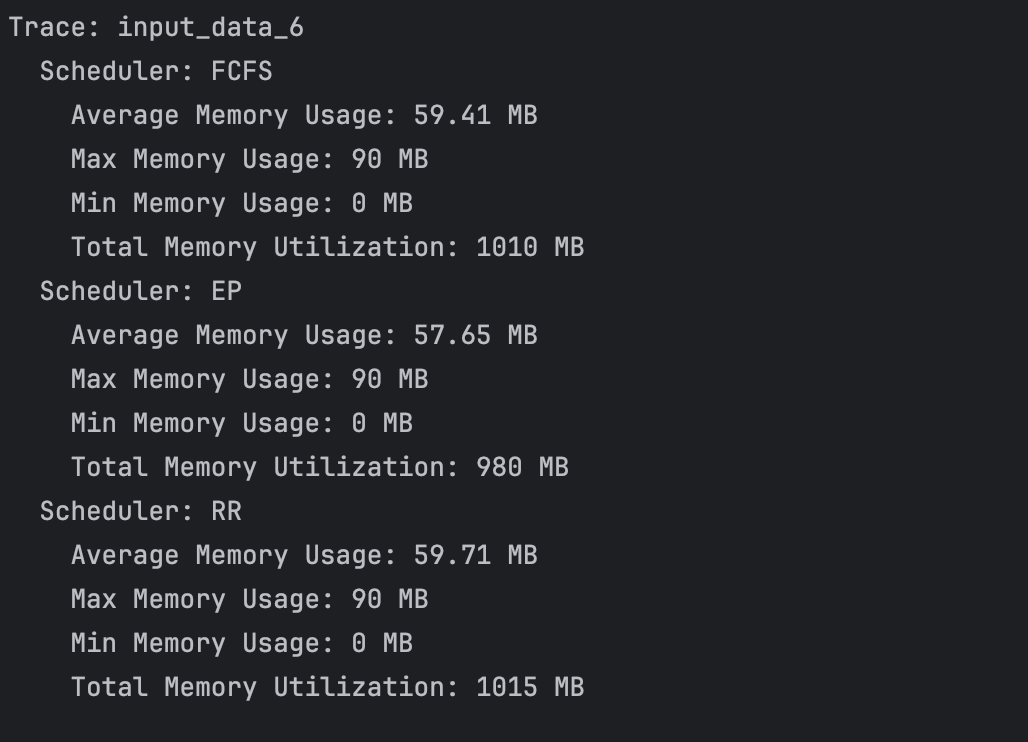
5. Input\_data\_5.txt



*Figure 25: Results from anaylze\_memory.py for input\_data\_5.txt*

Trace 5 is the largest trace amongst the test cases, as a result, the Maximum Memory usage reached its peak for each of the 3 schedulers (98MB/100MB). However, the AMUs are relatively similar for each scheduler with the EP implementation being the highest, but were all remarkably similar considering the TMU of this trace. As such, we see that for large data sets, RR and FCFS can perform quite similarly depending on the processes within the trace.

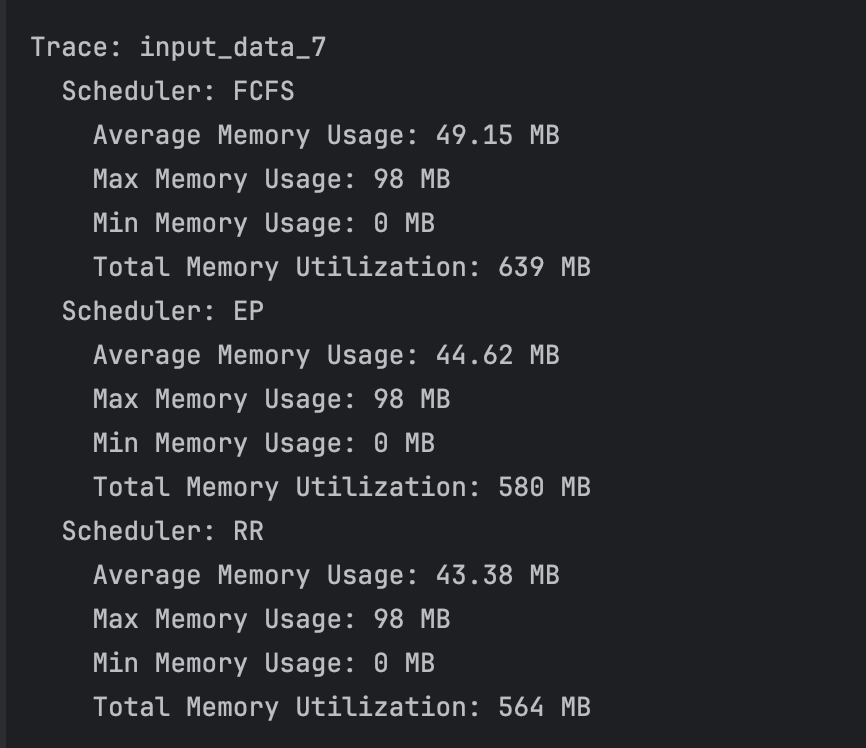
6. Input\_data\_6.txt



*Figure 26: Result from analyze\_memory.py for input\_data\_6.txt*

For this trace, the processes were IO intensive, unlike the output from the execution file, memory remained consistent regardless of the scheduler. Overall each metric remained relatively consistent with the EP scheduler performing the best in this particular scenario.

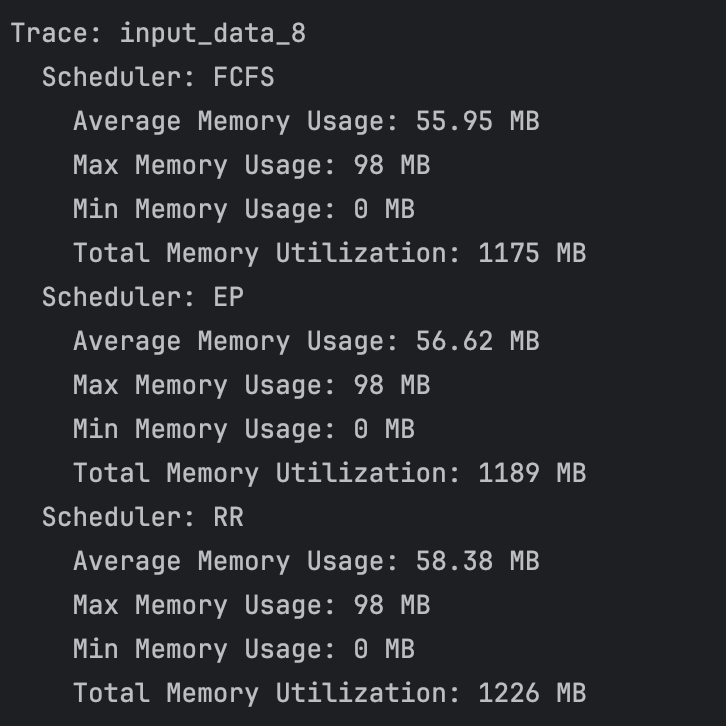
7. Input\_data\_7.txt



*Figure 27: Result from analyze\_memory.py for input\_data\_7.txt*

Contrary to the previous trace, this input provided CPU intensive processes, in order to examine the difference between schedulers and how memory is managed in all scenarios. This particular example showcases how FCFS schedulers can have significantly worse performance than more intricate implementations such as RR and EP. In addition, this example highlights how similarly performing RR and EP schedulers can behave.

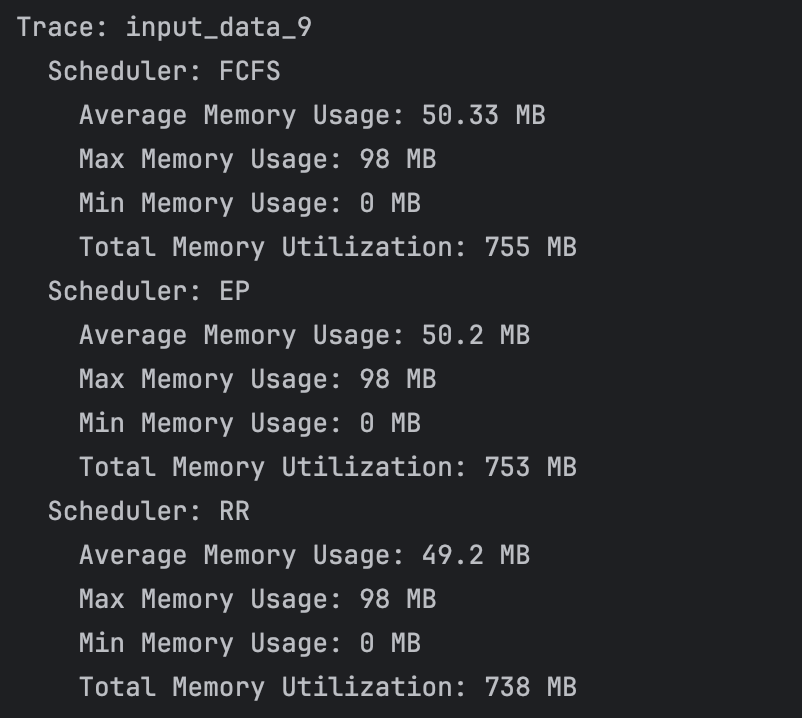
8. Input\_data\_8.txt



*Figure 28: Result from analyze\_memory.py for input\_data\_8.txt*

The eighth test case used a trace with staggered arrival times. In this scenario, the output from the execution file showed how well a RR scheduler can perform under the correct circumstance. However, as seen in the result from the memory utilization above, this can come at a price, memory. This circumstance indicates that strong performance metrics in the execution file are inversely proportional to the memory performance metrics of the same given trace of staggered arrivals.

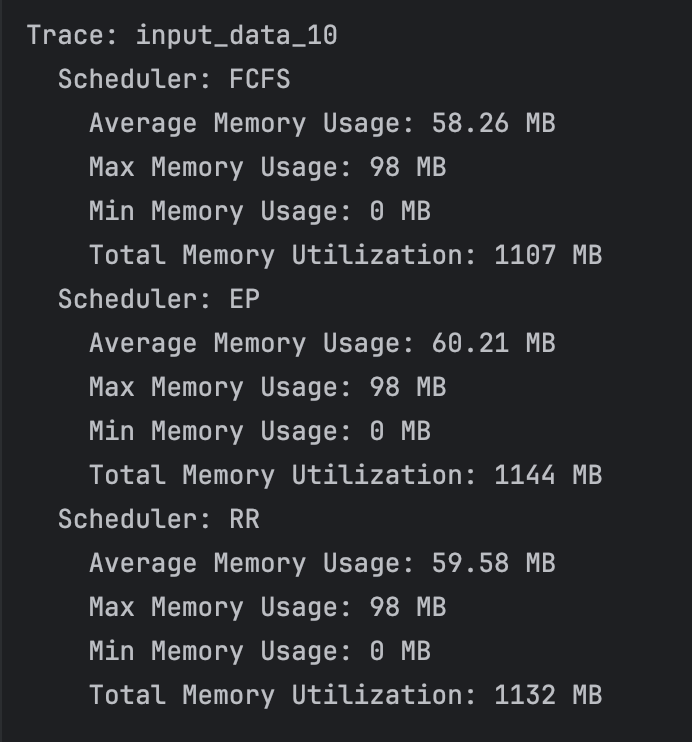
9. Input\_data\_9.txt



*Figure 29: Result from analyze\_memory.py for input\_data\_9.txt*

This input trace featured processed with short I/O time. As seen from the results above, for smaller input traces with processes with shorter IO, each scheduler performs quite similarly. In summary, none of the implemented schedulers yield particularly different results when considering AMU and TMU performance metrics.

10. Input\_data\_10.txt



*Figure 30: Results from analyze\_memory.py for input\_data\_10.txt*

The test trace analyzed above contained processes of varying CPU duration, IO duration and frequency. Yet, the calculations yielded performance metrics very similar in terms of TMU and AMU. It is to be said that for some traces, there is certainly an element of pseudo-randomness.