University of Amsterdam

Assignment 2: Scheduler

Comparative Analysis of Scheduling Algorithms

May 10, 2024

Lecturer:
Arno Bakker & Taco Walstra

Course:
Operating Systems

Course code:
5062BEST6Y

Student:
Daan Rosendal
15229394

1 Introduction

Efficient process scheduling is crucial for optimal system performance. This report deterministically evaluates various scheduling algorithms to identify their strengths and weaknesses across different configurations. Gradually improving the scheduling, each improvement is based on and/or contains the changes of the previous enhancement. The base algorithm used for comparison is the first-come, first-served algorithm. Strategies such as round robin scheduling, round robin scheduling with improved memory efficiency, and priority scheduling are explored. The goal is to identify optimal scheduling algorithms that balance processing efficiency and fairness across diverse workloads.

2 Method

This report uses deterministic modeling, which is a type of analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload. Deterministic modeling is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. (Abraham Silberschatz 2018)

The goal of the schedulers described in this report is not to perform well with one specific workload, but to perform well across several possible workloads.

2.1 Finding the optimal N-tries value

In managing memory allocation for incoming processes, the scheduler attempts to assign available memory based on the process's requirements. If a process requires more memory than is available, the allocation fails. Nevertheless, this does not rule out the possibility of successfully allocating memory to other, smaller processes. Thus, the concept of N-tries is introduced, which specifies the number of consecutive allocation attempts the scheduler should make for the next processes in the ready queue, irrespective of the success or failure of a particular attempt.

The strategy to determine the most effective N-tries value is based on two key metrics: (1) The number of successful allocations achieved per nth try, which helps to evaluate whether the additional effort of extending tries up to N is justified. (2) The overall average processing time associated with different N-tries values, which assesses the performance impact of varying the

N-tries value. Analysing these metrics allows for the identification of an optimal static N-tries value that balances allocation success rate and processing efficiency.

2.2 Round robin scheduling

A fixed time slice known as the "time quantum" or "time slice" is central to round robin scheduling. This quantum determines how long each process gets to run before it is preempted and the next process is scheduled. The length of the time quantum is crucial because it needs to balance between context switching overhead and fairness. A quantum that is too short causes high overhead due to frequent context switches, and a quantum that is too long reduces responsiveness.

A time quantum is generally from 10 to 100 milliseconds in length (Abraham Silberschatz 2018). To determine the optimal time quantum for the general use case of this scheduler, three key performance indicators shall be used: (1) turnaround time (total processing time), (2) waiting time, and (3) CPU utilisation.

2.3 Round robin scheduling with improved memory efficiency

The traditional round robin scheduling algorithm operates with a fixed time quantum. To enhance memory efficiency, implementing a dynamic time quantum might be more effective. This approach assumes that processes requiring less memory generally have shorter total processing times and could thus benefit from fewer preemptions, especially when they are close to completion. Therefore, a dynamic time quantum could be adjusted according to the process's memory needs relative to the total memory available.

2.4 Priority scheduling

The objective of the priority scheduling algorithm is to achieve a better average total processing time compared to the basic round-robin scheduling approach. Similar to round-robin scheduling with improved memory efficiency, this algorithm operates on the assumption that smaller processes tend to have shorter processing times. Therefore, a process's priority is determined based on its relative memory requirement.

The scheduler has 20 priority levels, each representing a 5% increase in relative memory needs, with level 0 having the highest priority and level 19 the lowest. For instance, if a process requires 11% of memory, it will be assigned priority level 3.

This scheduling approach is preemptive, allowing high-priority processes to interrupt and preempt lower-priority processes that are currently executing. To prevent starvation, where a low-priority process never gets a chance to run due to the continuous presence of higher-priority processes in the ready queue (Abraham Silberschatz 2018), an aging mechanism is implemented. The mechanism has two configurable parameters: (1) aging time interval and (2) aging factor. The interval is set to the time slice, and at each TIME_EVENT, the aging factor increases the age of non-executing processes in the ready queue.

When selecting the next process to execute, the scheduler looks through the ready queue for the highest-priority process and moves it to the front of the queue. The aging mechanism is accounted for by subtracting each process's aging value from its priority level.

3 Results

This chapter outlines the results from the deterministic evaluation of the algorithms.

3.1 Finding the optimal N-tries value

The utilised command uses the maximum amount of processes (-p 40960) and a high memory utilisation (-m 0.9) to test the system under the most extreme conditions. Note that the observed pattern holds true for less extreme conditions. The command:

```
./bin/ntries -c 0.5 -i 0.5 -m 0.9 -p 40960
```

Figure 1 shows the number of successful allocations per Nth try for the above command. The x-axis represents the Nth try from 1 to 30, and the y-axis shows the count of successful allocations. The histogram indicates exponential decay; most successes occur within the initial few tries, with a significant drop after the second try and a slower decline following. The diminishing bar heights highlight the decreasing likelihood of successful allocations as the number of attempts increases, suggesting that early tries are most likely to succeed.

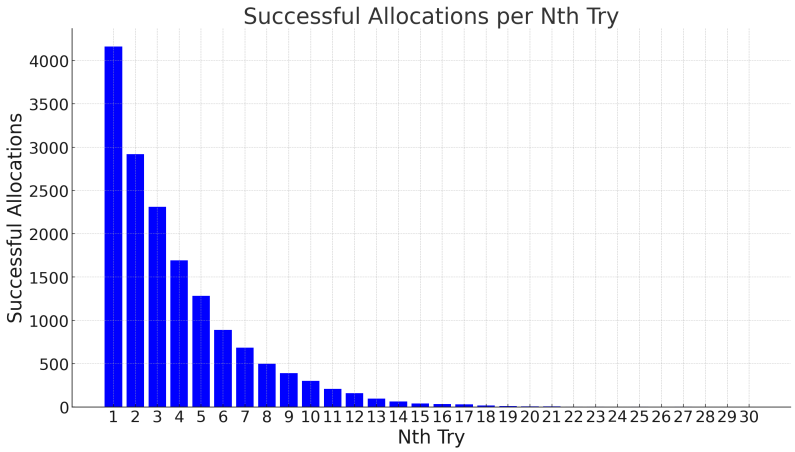

Figure 1: Successful memory allocations per Nth try

While knowing the most probable times for successful allocations is insightful, a comprehensive evaluation requires analysis of performance across different configurations. This was achieved using the command below, which was executed for each N-tries value up to 30—the maximum observed in Figure 1. Performance was analysed under various memory utilisation settings, as indicated by the -m array of arguments:

```
./bin/ntries -c 0.5 -i 0.5 -m [0.1, 0.3, 0.5, 0.7, 0.9] -p 40960
```

Figure 2 presents a line graph illustrating the relative change in total processing time for different memory configurations as the used N-tries value increases. The x-axis represents the used N-tries value from 1 to 30, and the y-axis shows the relative change in processing time as a percentage of the initial value for five memory configurations.

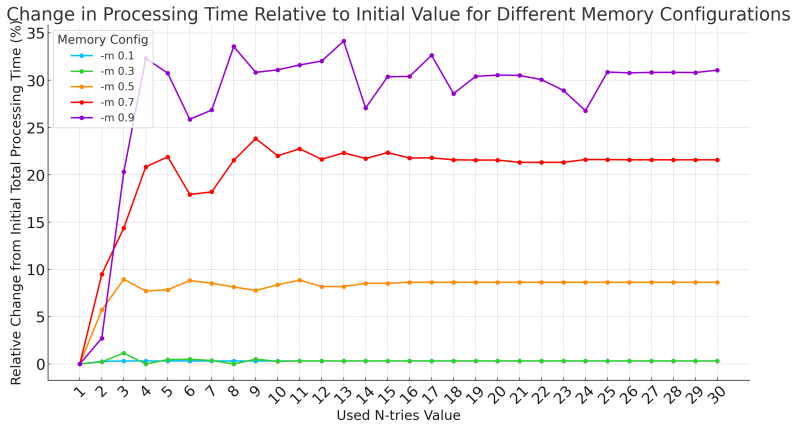

Figure 2: Change in relative processing time by used N-tries value

To determine an ideal static N-tries value based on the chart, a point must be found where the percentage differences stabilise or reach a minimal variance from the initial value across all memory configurations. This would suggest that after this number of tries, further iterations do not significantly change processing times, indicating a stable state. From the chart, it appears that, for -m 0.1 and -m 0.3, the percentage differences are consistently low and flat throughout the tries, suggesting these configurations stabilise almost immediately. For -m 0.5, there is noticeable variance early on, but it starts to stabilise closer to the 10th try. For -m 0.7 and -m 0.9, a sharper incline can be observed initially with stabilisation occurring around the 10th to 12th tries.

Given these observations, choosing a try number around 10 to 12 would be a balanced decision to ensure stability across all tested memory configurations while minimising the need for excessive iterations. This number ensures that most configurations have settled into a stable processing time by then, making it a practical choice for general use.

Looking back at figure 1, an N-tries value around 12 is logical as most successful allocations occur before the 12th try. Moreover, with the processing times stabilising around 10-12 tries and with no negative impact observed from increasing N-tries within this range, setting it to 12 optimally covers most use cases.

In scenarios such as this one, where the simulation is predetermined, setting N-tries to infinity could be justified since the ready queue rarely exceeds tens of processes. However, if system load fluctuates with the arrival of resource-intensive processes, the number of processes in the ready queue could escalate, making it essential to choose a sensible N-tries value to prevent performance degradation from excessive processing overhead.

3.2 Round robin scheduling

To assess the performance metrics for round robin scheduling, the following command was executed with time slices ranging from 1 ms to 100 ms. This configuration aimed to optimise the system for general usage, maintaining a utilisation of 50% for all parameters and with the maximum amount of processes allowed in the simulation; 40,960 processes.

```
./bin/round-robin -c 0.5 -i 0.5 -m 0.5 -p 40960
```

Figure 3 depicts the results of the performance assessment, showing Total Processing Time and Waiting Time across every time slice. CPU utilisation was consistent at about 50% throughout all slices and is thus not displayed in the graph.

The Total Processing Time and Waiting Time both show an increasing trend as the time slice lengthens, which suggests that a minimal time slice results in optimal system performance. Given these results, further optimisation could be achieved by exploring the potential benefits of reducing the time slice even further, below the current minimum of 1 ms, if the system's capabilities permit.

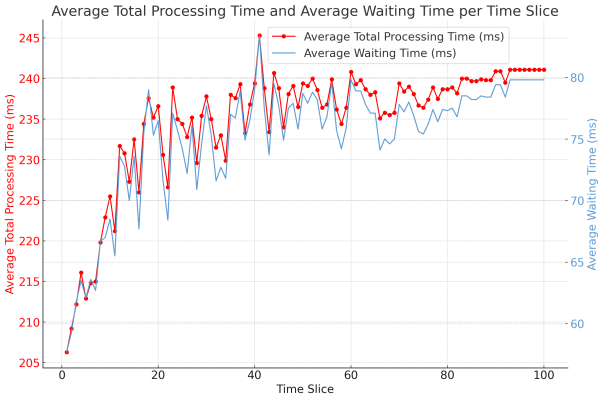

Figure 3: Average total processing time and waiting time for round robin per time slice

3.3 Round robin scheduling with improved memory efficiency

Efforts were made to improve memory efficiency within round robin scheduling by implementing a modified time quantum strategy. The base time quantum is set to 1 ms and increases tenfold for processes consuming less than 3% of the designated memory capacity, increases fivefold for those using between 3% and 15%, triples for those using between 15% and 25% and remains unchanged for processes whose memory consumption meets or exceeds 25%.

Figure 4 illustrates the adjustments in memory utilisation when the round robin scheduling algorithm is modified for better memory efficiency.

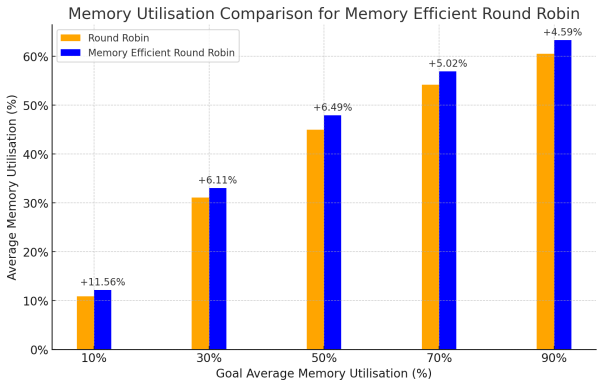

Figure 4: Memory utilisation comparison for memory efficient round robin

The modification shows an improvement in memory utilisation across all predefined memory utilisation goals, supporting the assumption that processes with lower memory demands benefit from reduced frequency of preemption.

3.4 Priority scheduling

The aging time interval and aging factor were made configurable so that many different configurations could be tried out. To find the optimal configuration, many possible combinations for the configurations tested. The aging time interval was tested from 1 to 100 and the aging factor from 1 to 20. All these configurations were tested using the following command with various amounts of processes, listed in the array for the -p argument:

```
./bin/priority -c 0.5 -i 0.5 -m 0.5 -p [100, 500, 1000, 5000, 10000, 20000,
40960]
```

The combination of an aging time interval of 2 and an aging factor of 10 consistently appeared in the top 20, even being in the top 5 for each ranking upwards of 5000. Hence, this configuration was chosen.

Figure 5 compares the performance of the priority scheduling algorithm to that of the round robin algorithm. The first data point, at 100 processes, is not shown because it stretches out the Y-axis, making it harder to see the performance variance for the other data points.

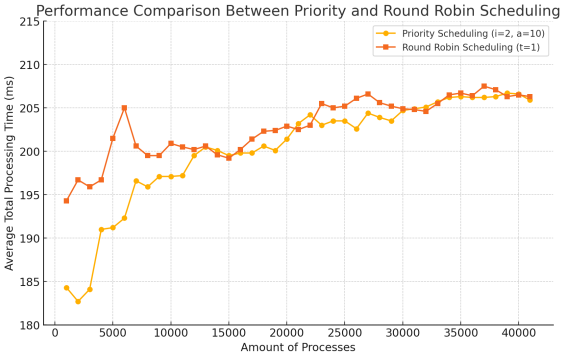

Figure 5: Performance comparison between priority and round robin scheduling

It is important to note that with 100 processes, round robin had an average total processing time of 152.3 ms, while it was 164.0 ms for priority scheduling, thus round robin performs better at the very lowest amount of processes, approximately from 100 to 500 processes. From 500 to around 12000 processes, priority scheduling outperforms round robin. From 12000 processes onwards, the performance between the algorithms is fairly equal, with priority scheduling mostly slightly outperforming round robin.

Although round-robin scheduling has an advantage at lower process counts, the consistent performance improvement of priority scheduling across a broader range of processes makes it a favourable choice for systems with varied and demanding workloads.

4 Discussion

This chapter provides a discussion and conclusion of the results derived from the deterministic evaluation, and highlights potential areas for future research.

4.1 N-tries

The analysis of the N-tries value indicates that a static value of 12 strikes a balance between successful allocations and processing efficiency. Further research could explore dynamic adjustments to the N-tries value, such as adjustments based on queue length. Additionally, this report does not address the impact of different N-tries values or approaches across different algorithms, only FCFS. Furthermore, the -c and -i parameters were both kept at 0.5; changing these values could affect the optimal N-tries value as well.

4.2 Round robin scheduling

The results for round robin scheduling show that reducing the time slice to the minimum of 1 ms optimises scheduling by minimising average waiting and average total processing times. Similar to the N-tries discussion, further research could look into how changing the simulation arguments affects this algorithm.

4.3 Round robin scheduling with improved memory efficiency

Implementing a dynamic time quantum strategy based on the relative memory usage of processes enhanced the memory efficiency of the round robin scheduler. Processes with lower memory needs benefit from reduced preemptions, allowing them to complete execution with fewer interruptions. One configuration for the dynamic time quantum strategy was presented in the results, further research could involve gathering and analysing statistics for other strategies to identify the optimal configuration.

4.4 Priority scheduling

Priority scheduling, with aging to prevent starvation, generally outperformed round robin scheduling across a broad range of process counts, particularly for mid-sized workloads ranging from 500 to 12.000 processes. However, round robin had a slight advantage with fewer than 500 processes.

For priority scheduling, performance with workloads under 500 processes varied significantly depending on the workload size, making it difficult to establish a consistently well-performing configuration. Therefore, alternating between priority scheduling and round robin based on workload size could potentially combine the strengths of both approaches, which warrants further investigation.

The aging time interval of 2 and aging factor of 10 were identified as the optimal configuration. Further refinements could include dynamically adjusting these parameters based on system load or process characteristics. Additionally, the priority mechanism could be modified to use alternative determinants, such as a process's number of preemptions or CPU time consumed.

It was a tough quest to find an algorithm that performs better than round robin. Many were tried such as using multiple queues for implementing Multilevel Feedback Queue (failed because the system does not allow extra queues), smallest job first (starves big processes), non-preemptive random selection (provides inconsistent and inferior results), round robin with aging, round robin with a time slice multiplier based on relative memory need, and priority scheduling with queue reordering/sorting.

Finding an algorithm that consistently outperforms round robin proved challenging. Several alternatives were explored, including a Multilevel Feedback Queue (MLFQ) (which failed due to system restrictions on additional queues), Smallest Job First (which lead to starvation of larger processes), non-preemptive random selection (which provided inconsistent and suboptimal results), round robin with aging (which increased overhead due to additional computations), round robin with a time slice multiplier based on relative memory needs (performed better only in niche situations), and priority scheduling with queue reordering (which had a more complex implementation than linear search and had equal performance). Further research could focus on developing optimised algorithms that operate within the system's limitations, taking inspiration from the algorithms mentioned in this paragraph.

4.5 Conclusion

This report has shown that while each scheduling algorithm has unique strengths and weaknesses, priority scheduling generally offers better performance across varied workloads. However, optimal performance requires careful calibration of parameters like aging interval and aging factor.

References

Abraham Silberschatz Peter Baer Galvin, Greg Gagne (2018). *Operating System Concepts (Tenth Edition)*. Wiley.

A Appendix schedulers code

Attached is a tarball containing the code for the schedulers (schedulers.tar.gz).