OS Technical Report: Deep Dive into Scheduling Algorithms

Greg Martin (j64h434), Dave Miller (c51c869),
Anthony Nardiello (f47f762), Alex Sutherland (g59q569)

November 2020

1 Introduction

The goal of this project was to examine several different scheduling algorithms by providing a "Proof of Concept" and implementing them ourselves in C. By comparing the algorithms we hoped to gain some intuition and evidence of potential benefits and drawbacks of each. Moreover, an in-depth examination of what could lead some of the algorithms to be more effective/practical in application was foremost in our minds.

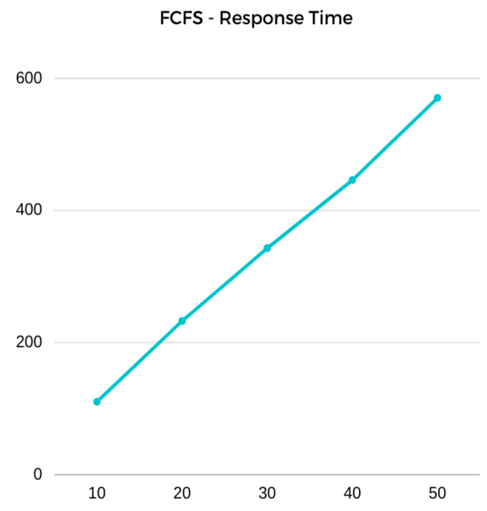
2 Background and Related Work

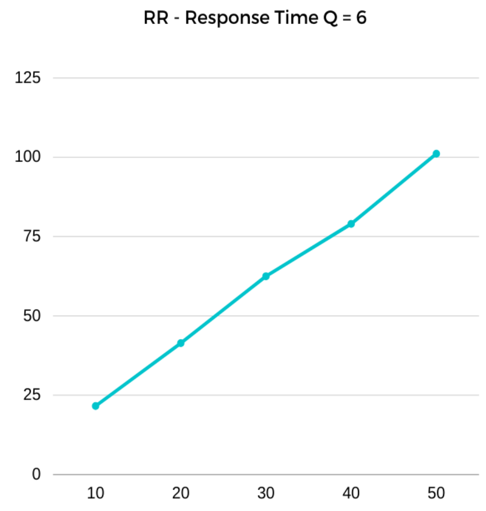
In order to get any work done on these algorithms, we had to have a deeper understanding on what different types of scheduling algorithms there were out there. Thus, as a group, we all decided to look deeper into different algorithms, using the basic knowledge given to us by the book and lecture slides that focus on the topic of scheduling. As the research went on, we realized that it would be necessary to pick one of these algorithms that hadn't gotten discussed in depth in class, so that way we could actually analyze alternative solutions to the problems we discussed within those lectures. Thus, we looked at alternative algorithms, and came together to discuss which algorithms we were going to pick.

3 Our Work

3.1 First Come First Served / Round Robin

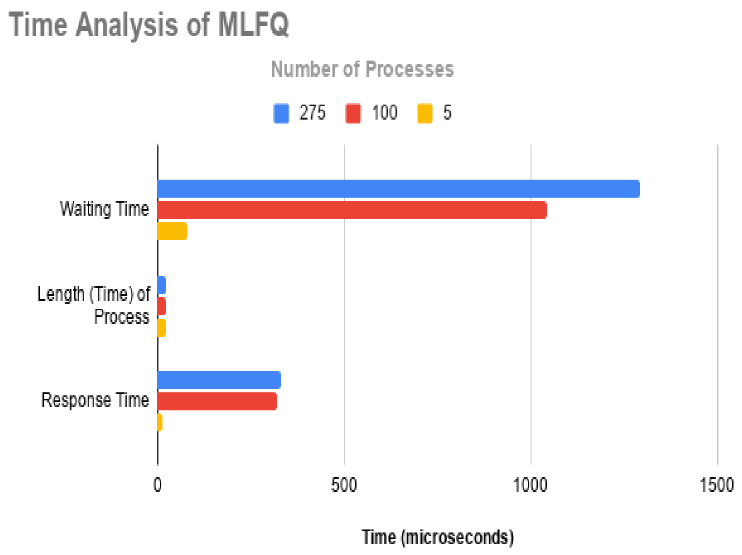
First Come First Served (FCFS) and Round Robin (RR) are relatively simple scheduling algorithms that we include here in order to form a basis for comparisons of the high level schedulers below. FCFS works exactly how it sounds, by placing processes on a queue as they arrive and letting the CPU run each process in its entirety before moving to the next. Advantages of this scheduler include ease of implementation and low overhead, but it can be unfair with non-uniformly sized processes and wait time is usually high. RR is a widely used, easy to implement and understand scheduler that works by assigning each process in the waiting queue a certain amount of time (usually called a time quantum) on the CPU. Thus, the advantages include fairness, good response time, and it avoids starvation. However, RR does not allow priority scheduling and the overhead of switching processes often can reduce performance. Pictured below are graphs depicting response time of these two schedulers, with number of processes on the x-axis and time (ms) on the y-axis. These results show a linear relationship for both schedulers, but it is quite obvious that FCFS has much higher response time overall.

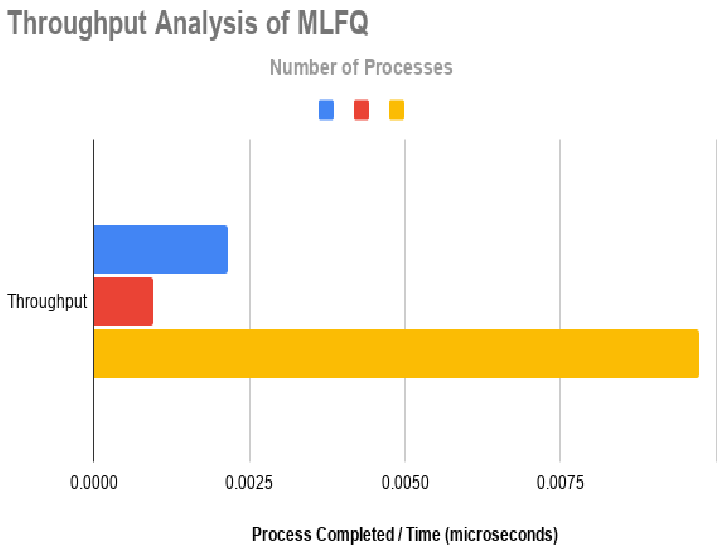




3.2 Multilevel Feedback Queue

Multilevel Feedback Queue, from here on referred to as MLFQ, is a priority based scheduling algorithm composed of several different ready queues and time quantums. In MLFQ, processor time is first allocated to processes in the highest priority queue (Queue 1),if the process does not finish with its task in the given time quantum (for our experiments this was seven time units) than the process is booted off of the processor and placed in a lower priority queue[1]. This process is repeated, with larger time quantums the lower in priority, until a base level queue is reached. This base queue is implemented with First-Come First-Served (FCFS), so that eventually, no matter the size of the process, it will be able to finish its task. In our experiment, three queues were implemented in MLFQ, two were priority-based with time quantums of seven and fifteen respectively, with the third queue being (FCFS). Our experimental results were quite telling of the characteristics of this algorithm for a single processor system. Below are listed our results for several different runs with different metrics:





Here we can see that the algorithm was run with 275 processes, 100 processes, and 5 processes, with average metrics like: waiting time, length of process, response time, and throughput being evaluated. One of the most obvious conclusions to be drawn from the results is that the higher the number of processes, the greater the waiting time and response time along with a lower throughput. It's clear that this algorithm greatly favors shorter (and lower number) processes, with the decreased waiting time, response time, and increased throughput being evidence of that. Because of the nature of the experiment, starvation was not possible in our confined setting because there were only a certain number of processes and eventually every process would get a chance to complete. However, in applied use of this algorithm, it is important to note that starvation would be possible if the process' length is long enough because new processes could continually be added at a higher priority, leaving lower priority processes no chance at the processor.

3.3 Deadline Monotonic

Deadline Monotonic is a fixed priority pre-emptive scheduling algorithm, focusing on calculations designed to determine what the actual deadline is to complete each process. It focus on a simple relationship: computation time <= deadline <= period, where computation time is a process' calculated computation time, the deadline is a the amount of time a process needs to run by, and the period is the total runtime from start and including up to the deadline. Additionally, Deadline Monotonic is governed by seven restrictions which exist to strengthen the relationship used for the calculations. These restrictions are as follows:

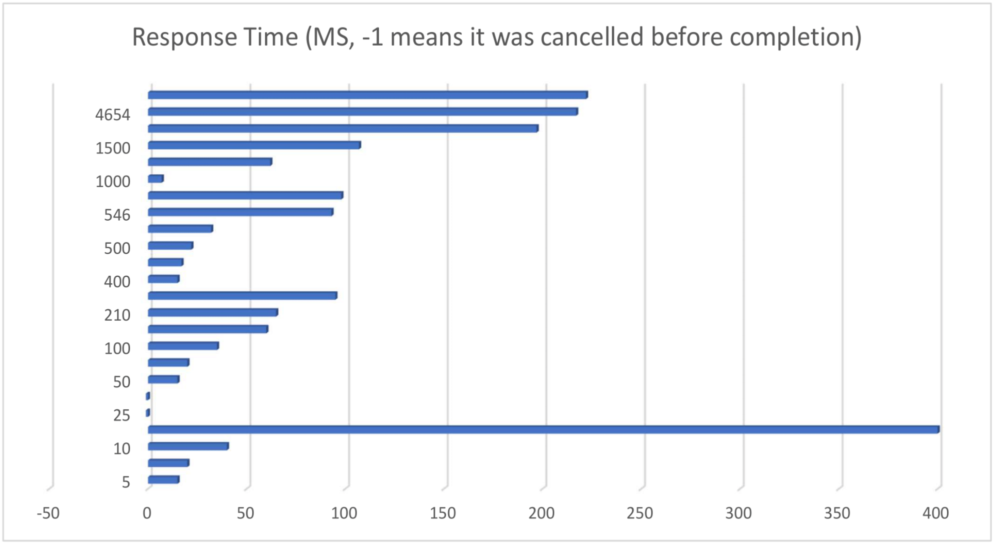
1. All Processes have deadlines less than or equal to their minimum inter-arrival times(periods).
2. All processes have worst-case execution times that are less than or equal to their deadlines
3. All processes are independent, and so do not block each other's execution
4. No process voluntarily suspends itself
5. There is some point in time, referred to as a critical instant, where all of the processes become ready to execute simultaneously.
6. Scheduling overheads are zero.
7. All processes have zero release jitters (the time from the task arriving to it becoming ready to execute.

There are ways to relax these seven requirements, but to do so transforms the algorithm into variations of deadline monotonic scheduling, or into entirely new algorithms. For true deadline monotonic, both the relationship and these seven requirements must hold true.

With these basic details about deadline monotonic scheduling, it was decided that using deadline monotonic as one of our unique scheduling algorithms would work best to draw a contrast from the algorithms we learned in class. That led us into our implementation, which was a rather interesting event. Due to the nature of the calculations, we needed to code the relationship into our implementation, or else the system would not work as intended. Thus, we codified this relationship using the following code example of Insertion Sort:

```
int i;

    // first element
    if (taskNum == 0){
        taskInfo[0][0] = computation;
        taskInfo[0][1] = deadline;
        taskInfo[0][2] = period;
    }

    // compare and insert
    else{
        for (i=taskNum; i>0; i--){
            if (deadline < taskInfo[i-1][1]){
                taskInfo[i][0] = taskInfo[i-1][0];
                taskInfo[i][1] = taskInfo[i-1][1];
                taskInfo[i][2] = taskInfo[i-1][2];
            }

            else
                break;
        }

        taskInfo[i][0] = computation;
        taskInfo[i][1] = deadline;
        taskInfo[i][2] = period;
    }
```

Once we codified the relationship, our final deadline calculations were easy to create. With this in mind, we were able to achieve the final results:



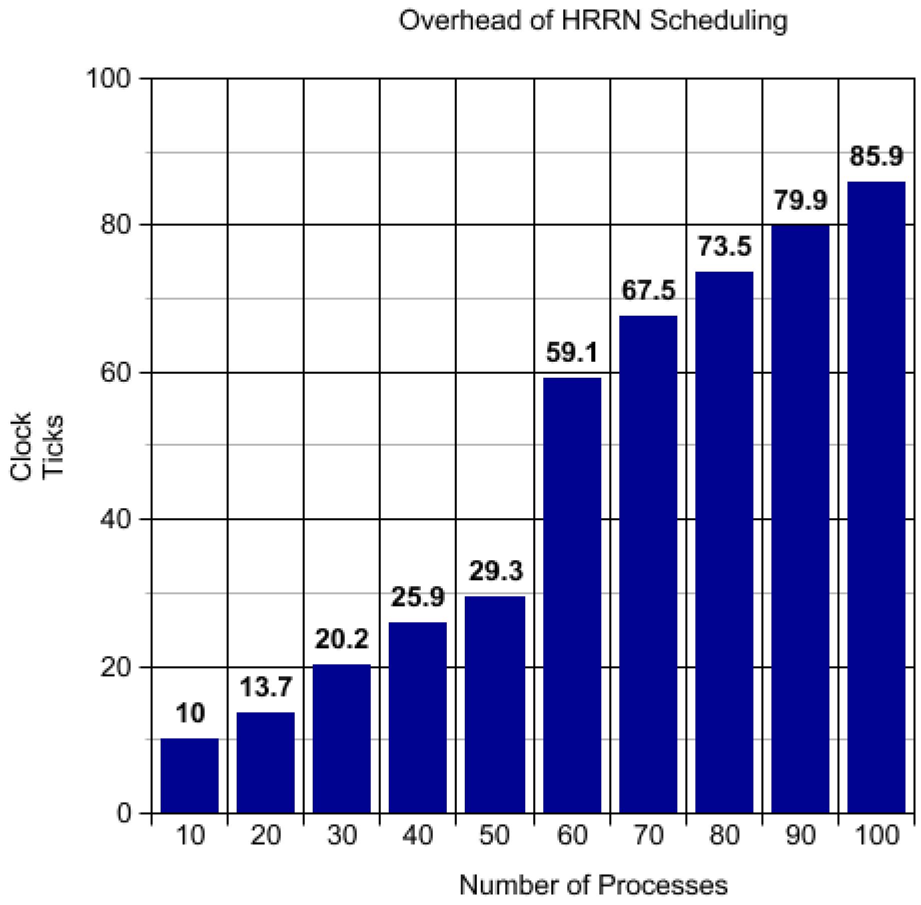
As you can see from the results, for the most part as the deadlines for our processes increased, so to did the response time for how quickly they finished. Now, there were a couple of outside data points, mostly coming from what the stated computation time for the process was. If the stated computation time went up, but not proportional to anything, then we received the data points that gave us the -1s. Similarly, if the computation time went up, but stayed proportional to the other two values, then the response time could be lowered, even if the deadline was high. In summary, the results prove that the relationship is critical for the success of the algorithm, as when the relationship is broken, the response time either do not happen, or they grow exponentially. Thus, the process does not get scheduled, which is not what we want from the system.

3.4 Highest Response Ratio Next

Highest response ratio next (HRRN) is a non-preemptive scheduling algorithm where the process with the highest response ratio is given cpu time. The response ratio of a process can be found by adding the waiting time of the process to the burst length and then dividing that resulting sum by the burst length. This scheduling algorithm favors shorter processes, but still allows longer processes to run if their waiting time is high enough. For example, if there are two processes one with a burst length of 5 and an arrival time of 2, and the other with a burst length of 2 and an arrival time of 7. If time is currently at 7, the longer process will have a response ratio of 2 and the shorter process will have a response ratio of 1. Therefore, the longer process will be selected to run instead of the shorter process. We chose to use a linked list to store processes for our HRRN algorithm, ordered by arrival time. This allows for processes to quickly be removed from our list of processes. The basic procedure for our HRRN algorithm is:

1. Calculate response ratio of processes that have arrived.
2. Run process with highest response ratio.
3. Remove process from linked list of processes.
4. Repeat the above steps until no more processes need to run.

Because this algorithm performs calculations in between running processes, overhead is introduced. Below is a figure showing how overhead increases relative to the number of process that need to be run.



Overhead was calculated using the clock function in C which returns the number of clock ticks since the program started. Clock is called twice, once before the algorithm starts scheduling and again after all processes have run. The overhead is calculated by subtracting the value returned by the first call of clock from the second.

4 Discussion

We present an analysis of common (and uncommon) scheduling algorithms, including: advantages and drawbacks, ease of implementation, uses, and example code. Making a scheduler from scratch in order to show the inner workings of an algorithm is somewhat difficult due to the multitude of different ways to accomplish the same thing. Furthermore, most schedulers are built with the purpose of scheduling a certain type of process on a certain machine, and creating a generic version requires some imagination. With this in mind, our experiences and results show that no one scheduler is objectively best, although certainly some are better than others. We propose that the algorithm should be tailored to the context that it will be used in. For example, if the task in not time sensitive, FCFS or RR might be used because they are easy to implement with low overhead and decent performance. However, when time is an issue or other restrictions are apparent, other schedulers would be better. MFLQ is great for priority scheduling, Deadline Monotonic works best with tasks that are potentially pre-emptive and of a fixed computational time, and HRRN as a good general scheduler that avoids starvation.

5 Conclusion and Future Work

The results of our experiment and the implementation of these algorithms gave us an understanding of why these algorithms were created, how they stack up to each other, and there use in general. While this experiment has been enlightening and has provided us with the intuition and evidence that we set out for, there are improvements that could still be made. For example, an implementation of our experiment with an open number of processes (allowing for the possibility of starvation) would provide an even greater understanding of the use of these algorithms in the practical world. Furthermore, featuring specific processes, ones that would be more common on certain systems, would also benefit the experiment because it would give an idea of how these algorithms would function on certain operating systems.

References

[1] Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. *Operating Systems: Three Easy Pieces*, http://pages.cs.wisc.edu/r̃emzi/OSTEP/cpu-sched-mlfq.pdf.

[2] Neil C. Audsley *Deadline Monotonic Scheduling* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.8928rep=rep1andtype=pdf