Abstract

*We had to create three different schedulers for this lab. These schedulers can take in generic processes in text form and get the information that could be used to create a Gantt chart. It should be noted that we do not actually schedule real processes. Creating these schedulers helped us to learn about how they work and learn about some of the things that our computers do without us even thinking about it. This paper is meant to be a dive into the performance of these three schedulers and to compare and contrast these along with some other schedulers that have been used in the path and present.*

# 1. Introduction

Lab three was to create a Multilevel Feedback Queue Scheduler, Real-Time Scheduler, and a Windows Hybrid Scheduler. The main ideas of these three queues is that in Multilevel Feedback Queue Scheduler, processes are given a small amount of time to run in the order of priority, then loops around. In Real-Time Scheduler the idea is to run processes in order of who needs to be done the soonest, regardless of the remaining burst time. Windows Hybrid Scheduler is designed to run kernel processes, then run user processes, based on a dynamic priority system. In running these schedulers Real-Time Scheduler actually runs very quickly, but this is due to the fact that many processes cannot finish due to deadlines. These schedulers all have many situations where they are better than each other and also places where they are extremely inefficient at scheduling.

While creating these schedulers there were some hurdles we had to overcome. One of the major ones was that we had to handle simulating our own clock ticks, which means using many loops in the code, which are not very efficient when the number of processes begins to get large. Another hurdle that we had to overcome is simply managing the queues and what processes are present in each one and where they are supposed to move to.

In this paper we are going to talk about some of the performance issues that the processes have and what situations each scheduler is more suited for. We are also going to describe how the scheduler handles processes and manages itself, as well as some other schedulers that are out there so that we can compare what we did to the standards of current computing.

# 2. Previous Work

Looking around on the internet we found some previous schedulers we wanted to take a look into.

## 2.1 Linux Completely Fair Scheduler

The Linux Completely Fair Scheduler was merged into the October 2007 release of the Linux kernel and is now the default scheduler. This scheduler works by using a red black tree to create a timeline of future task execution. This scheduler also uses an idea called “sleeper fairness” which ensures that processes that are sleeping or waiting on the stack are given the same priority as those that are executing, meaning processes that are waiting to get input from a user or waiting on something else to happen will still get an adequate amount of resources from the CPU.

The scheduler uses an idea of maximum execution time to ensure processes get a similar amount of resources. The way this is used is to get an idea of how much time each process should run, and if it hits that number it is stopped and then inserted back into the tree based on its new execution time. The benefit of this is that processes that sleep a lot, which user processes usually do, maintain a low execution time and therefore have a higher priority and are able to easily get the resources needed to run when the time comes, and the user will have a more responsive feel.[2]

## 2.2 O(1) Scheduler

The O(1) Scheduler was the precursor to the Completely Fair Scheduler. This scheduler was designed so that it could schedule processes within a constant amount of time. The advantage of this is that it can handle a massive amount of processes with no troubles, since the time is guaranteed to be constant. The way this works is that all the processes are in one of two queues, and as they finish their time quantum they are moved to the other queue. Once the first queue is empty, the pointers simply swap so that the expired array is now the active array, and execution continues. This swap is what makes the scheduler so fast. [1]

An interesting thing to note is that this scheduler is focused on using two data structures, run queues and priority arrays. We didn’t know this before we started our schedulers, but we ended up using these data structures in our own schedulers. Part of the performance of our schedulers is that we do use these data structures that are very efficient.

# 3. Description of Scheduler Models

The schedulers are run by a main driver that simulates clock ticks. Within this driver processes are added to the queue if they arrive, the next process to run executes, and then the clock is incremented. The details of execution relating to a specific scheduler is described in each schedulers subsection.

## 3.1. Multilevel Feedback Queue Scheduler

The way the Multilevel Feedback Queue Scheduler works is by creating a set of queues which all work the same except the last. The first queues are all set up as round robin queues, which means that all processes run in order of arrival to the first queue for a set time quantum. When that time quantum expires the process is demoted to the next queue down. If a process finishes within the time quantum it is simply removed and the next process runs. Once the top queue is empty, the processes in the next queue down run. Each queue below the first has a time quantum that is double of the previous. If a queue is running and a process enters the queue above, if one exists, the current process finishes its time quantum, but the next process to run is the one in the higher queue, even if process execution was in the lower queue.

The last queue in the group is a First Come, First Serve queue. This queue does not have a time quantum and processes will simply run until they are finished in order of their arrival to the queue. This queue will not run unless there are no processes in the queues above, so process in this queue can easily be starved. This queue will handle aging up if a process is in the queue for too long without running. This is the only queue that a process can age up from, so a process that ages up to the next queue may sit for quite some time before it gets a chance to run. Once it does, it is still restricted to the time quantum of that queue so it many be demoted back to the First Come, First Serve queue quite quickly.

This scheduler is done using priority queues. Each queue is a priority queue that is sorted in the order the processes arrive. The ready queue that is used to hold the processes until they should technically “arrive” is a priority queue that is sorted with a comparator that so Real-Time Scheduler by arrival time.

Lastly, our processes have some helper methods that allow us to compare them and work with them, as well as some variables that can be edited to allow for keeping track of original values when others are mutated during scheduling.

## 3.2. Real-Time Scheduler

The Real-Time scheduler works by creating a ready queue of processes sorted by arrival time just like the Multilevel Feedback Queue Scheduler. The way the scheduler actually works is that there is a single queue that is sorted by earliest deadline first. When a process is ready to arrive it is added to the queue. The queue is then sorted by deadline, and in the case of a tie sorted by priority and then process ID. Once the process is added to the queue, it begins to run. If at any point the process finishes, it is removed and the next process runs. If a process is running and a process is added with an earlier deadline, then the process is stopped and the new process begins to run.

There are two modes that this scheduler can be run in, soft real time and hard real time. If the scheduler is run in soft real time mode, processes that fail to meet their deadline are acceptable. What this means is that if a process becomes unable to finish by its deadline, determined by checking if the current tick plus the amount of burst time the process has left is greater than the deadline. This does mean that a process can finish on its deadline, which is a semantics problem that could be easily changed to force the process to finish before the deadline. If a process is unable to meet its deadline, the process is removed from the queue and a message is displayed to let the user know a process failed.

In hard real time mode, the scheduler does not accept any processes failing to meet the deadline. In this case, the ability to meet the deadline is checked in the same way, but if there is even a single process that fails, the entire scheduler shuts down. This can be advantageous for things like rocket launches, where a single thing wrong can be catastrophic. In normal situations, it would be silly to run in hard real time, since a single process can ruin the entire run.

Real Time scheduling does not have a concept of aging, since the processes all have deadlines they have to meet, it would not help execution to have the process jumping around. There are also no other queues in this scheduler, so moving a process would require removing at a spot in the queue, and then inserting at a new point and shifting the rest of the queue. This is a very time expensive operation, and the more time spent moving processes is less time for actually running them.

## 3.3. Windows Hybrid Scheduler

The Windows Hybrid Scheduler works using a large amount of, in this case one hundred, queues that are separated into two bands. The lower band is queues zero through forty-nine and the upper band is fifty through ninety-nine. These queues are essentially priority, so the scheduler finds whichever process is in the highest queue to run.. A process with priority five would be put into queue five. The lower band is reserved for kernel processes and the upper band is reserved for user processes. The Windows Hybrid Scheduler runs processes starting in the kernel and moving downwards towards the user processes. The Windows Hybrid Scheduler has a concept of aging and demotion, as well as promotion due to running I/O. It also uses a time quantum to ensure that a single process does not run for too long without stopping.

The aging in the Windows Hybrid Scheduler is done in the bottom ten percent of queues only, which means kernel processes do not age. When a process in the bottom queues reaches its aging timer, the process is moved up ten queues so that its priority increases. Since the bottom ten percent of queues can age, there needs to be a check to make sure that increasing the priority by ten does not move the process above the cut off line for user processes, if there are fewer queues.

Demotion in the Windows Hybrid Scheduler is done whenever a process finishes running its time quantum. The process is then demoted by the amount of time it ran in the CPU. So if a process runs for four ticks, the process will be sent to a queue four lower, assuming it doesn’t cross the band border. Another consideration is that the Windows Hybrid Scheduler does not allow a process to be demoted lower than its initial priority, so if a process staReal-Time Scheduler at three, and runs for four ticks, it will not be put lower than three. A process must be promoted before it can be demoted for the first time.

Promotion in the Windows Hybrid Scheduler is only done with aging and I/O. I/O is sent to be done on the tick before the end of the time quantum. When the process is sent to the I/O queue it continually waits for its I/O to finish before it can be sent back to the main queues to finish its burst time. When a process is sent back from I/O it increases its priority by the amount of I/O ticks it did, so doing three ticks of I/O will move the priority up three. Again this is bound by the border between the bands, so a user process cannot become a kernel process. Once a process returns from I/O it will continue to run for a time quantum at a time until it is done, but will now experience demotion when it finishes.

# 4. Performance of the Schedulers

Each scheduler has its own advantages and disadvantages regarding the sets of processes being scheduled. Depending on whether a set of processes has larger or smaller burst times, whether or not they have I/O operations to perform, if deadlines need to be met or many other factors, they may perform quite differently.

## 4.1 Multilevel Feedback Queue Scheduler

The Multilevel Feedback Queue Scheduler can perform quite differently depending on the amount of queues specified, the maximum time quantum set for each queue and the aging interval for the final First Come First Serve Queue portion of it.

The Multilevel Feedback Queue Scheduler performs greater with smaller burst processes that have burst times that can complete quickly within the specified time quantum of earlier queue levels since longer running processes will be demoted to make room for those that can run and complete quicker.

Multilevel Feedback Queue Scheduler is sub-optimal when dealing with longer running processes that have burst times which exceed that of the time quantum of the queue it is running in as this results in processes taking much longer to complete. These processes will need to complete many more cycles within lower priority queues causing them to take much longer to run especially if new processes arrive which consist of shorter burst times.

## 4.2 Real-Time Scheduler

Since the Real-Time Scheduler operates on a queue with processes being sorted by their shortest deadline first there are scenarios in which this works well and ones in which it does not.

Scenarios when the Real-Time Scheduler works best are where processes not only need to consist of shorter burst times but also require more relaxed deadlines in order to have the best opportunity to fully complete their runtime.

The situations where the Real-Time Scheduler performs suboptimal are when processes consist of longer burst times and really tight deadline as they will usually not encounter the ability to finish processing as other processes with shorter deadlines will be ran instead.

## 4.3 Windows Hybrid Scheduler

Windows Hybrid Scheduler performs quite differently from the other two. In the Windows Hybrid Scheduler processes not only have burst times to run but also have I/O operations to complete. Since Windows Hybrid Scheduler operates using a larger set of queues based on the priorities of the processes which determine the queue they start in.

The I/O queue adds a whole extra layer of complexity on top of what already mimics a more complex Multilevel Feedback Queue Scheduler as the Windows Hybrid Scheduler can also promote processes based on completion of I/O work.

Processes that have lower priorities and a lot of I/O work to be done, once they are complete, they will be promoted up based on how much I/O they accomplished, giving them a much higher priority and if they have little actual burst time left to run, they will complete faster than those without any I/O as their new higher priority will allow them to complete sooner due to the new queue it has been placed in.

Where Windows Hybrid Scheduler does not perform well is when processes consist of both longer running burst times, little to none I/O work to be done and the lower the priority, the increased turnaround and wait times for that process.

# 5. Future Work

There is some work that can be done in the future on this project specifically but also in the area of schedulers by itself. Something that would be interesting to work at would be to speed up our schedulers. With the one hundred thousand process sample file Real-Time Scheduler and Windows Hybrid Scheduler take about five seconds, but Multilevel Feedback Queue Scheduler takes around a minute. These are not the greatest in terms of performance and there are definitely some areas where some optimizations can be made. Another thing that would be useful is some more research into the programming standards of schedulers so that those can be used in these schedulers. Since this is a first attempt at process scheduling there is much to be learned.

Another thing that may be interesting to work on is attempting to develop a newer type of scheduler. The three that were implemented are all schedulers that have made the rounds and are documented, so making a new one could be fun. It would be interesting to try and combine a few types of schedulers to balance the flaws in each other and improve the algorithm. Eventually creating a new one completely from scratch may be a possibility if enough experience with schedulers is gained.

# 6. Conclusions

Overall, this lab was an interesting look into the way operating systems schedule their processes and handle things. It is definitely a part of computing that is not as well known or glamorous. There were challenges that presented themselves just as usual when programming a new concept, where half the challenges is learning enough about how it works to be able to adapt some of those ideas for yourself.

Of the three schedulers implemented, Multilevel Feedback Queue Scheduler was the most challenging since it was the first one. Real-Time Scheduler was fairly simple once it was broken down into its parts and some thought was put into how best to go about it. Windows Hybrid Scheduler was also challenging due to a lot of the constraints with priority and I/O, but since it behaved a lot like Multilevel Feedback Queue Scheduler with more queues and I/O, the general outline of how it worked came together pretty quickly. Writing this technical paper was certainly a new experience and is something that requires some forethought, but ultimately is more writing than a typical computer science major hopes for.

# 

# 7. References

1. “Completely Fair Scheduler,” Wikipedia, 31-May-2018. [Online]. Available: https://en.wikipedia.org/wiki/Completely\_Fair\_Scheduler. [Accessed: 04-Dec-2018].
2. “O(1) scheduler,” Wikipedia, 07-Nov-2018. [Online]. Available: https://en.wikipedia.org/wiki/O(1)\_scheduler. [Accessed: 04-Dec-2018].