Simulating Process Scheduling with Real Time, Multilevel Feedback Queue, and Windows Hybrid Schduler

Nick Pientka and Brad Schroeder

Operating Systems

University of Wisconsin-Eau Claire

Eau Claire, WI 54702

{pientknk, schroebs7395}@uwec.edu

Abstract

Computers must run processes as fast as possible in order to keep the user from waiting too long when running and loading programs. This is an exploration to see how well various schedulers handle process management and completion. With a better understanding of how various process schedulers work on various computers, one can better understand why having many applications open at once can slow down the computer. Schedulers must handle incoming processes and complete as many as it can as quickly as possible. By comparing the run times of three different schedulers running a set amount of processes and various burst times, we can gather useful data needed to find the better scheduler for various burst times. The data we will be looking at includes average wait time (AWT), average turnaround time (ATT), and number of processes completed (NP).

1 Introduction

Processes can be scheduled in multiple different ways and many different operating systems have a variety of algorithms that they use in order to schedule all the processes they receive. For this project we used three different algorithms to handle process scheduling. They are Real Time processing which is an earliest deadline first algorithm, Multilevel feedback queue which allows processes to run and moves them down queues to wait if they don't finish on time, and the windows hybrid scheduler which focuses on the priority and PID of a process and puts it into a queue accordingly.

2 Real Time Scheduler

The first algorithm we implemented was the Real Time Scheduler. Real time has two different aspects that you can go through which are hard time and soft time. Hard time means that when a process doesn't finish on time the entire scheduler fails and shuts down. We implemented this but it's not very viable when there are five hundred thousand processes that arrive and are supposed to finish within ten thousand seconds. If you take the first select few say the first time out of the massive file this is not a problem since all the arrival times are so spread out it doesn't make much of a difference but when this comes to the 50 processes arriving in any given second it fails within seconds. Below in Figure 1 we display our process of handling our hard time.

```
if (rtsFlag == "h") {
          if ((time > waitVector.front().getDeadline())
|| (waitVector.front().getDeadline() - time <
waitVector.front().getTimeLeft())) {
           if (waitVector.front().getTimeLeft() <
waitVector.front().getBurstTime()) {
                waitingTime += time -
waitVector.front().getArrival() -
(waitVector.front().getBurstTime() -
waitVector.front().getTimeLeft());
           }
           done = true;
           cout << "Scheduler exiting due to Hard Real-
Time scheduler dropping a process" << endl;
          }
       }
```

Figure 1

This is a simple check to determine whether the simulator should stop at a time when a process is past its deadline.

Implementing soft time is just as easy. This is just a simple loop handling all processes arriving at any given time and checking to see what processes has the earliest deadline and running that first. After running this we realized that the average wait time and average turnaround time will be extremely low because our simulator likes to pick up those processes with a burst time of zero to 1 since those

generally have the earliest deadlines. This allows our simulator to get done 3795 processes within the little over ten thousand seconds that the last deadline reaches. Also during our simulator for RTS we make sure that we don't even bother running a process that we absolutely know won't finish on time. Comparing our program with other students in the class we have found that our simulator finishes more processes within the time because we don't run those that can't finish. Even though we finish more than others some algorithms we have seen from others look for the processes that always have the lowest burst times generally at 1 or 2 because these would almost finish a process every second allowing more to be completed. For the five hundred thousand processes passed into our simulator originally our final product for the Real Time Scheduler is a program that completes in 30s real time, has an Average Waiting time of 0.247 and an average Turnaround Time of 2.64 seconds showing that our program likes to target those processes with a lower burst time.

3 Multi-level Feedback Queue Scheduler

The multi-level feedback queue scheduler allows for every process to be completed but creates a large waiting time and large turnaround time for those simulations with a large amount of process in a short period time times. The multilevel feedback queue allows for a select amount of queues to be created based on user input and halves the quantum burst of the queue every time it gets demoted until it gets to the final queue which allows it to finish. Allowing the original quantum burst on the very first queue allows a majority of the processes to be completed within a short amount of time while those processes with a very long burst time get pushed back a few queues and get aged up eventually. We ran multiple speed tests on processes ranging between ten and five hundred thousand. We ran our speed tests on a file with two thousand five hundred processes with multiple queues, aging times, and original quantum burst times. In Figure 2 below we can see that as the original quantum burst goes higher the time it takes for the simulator to finish 2500 process goes down.

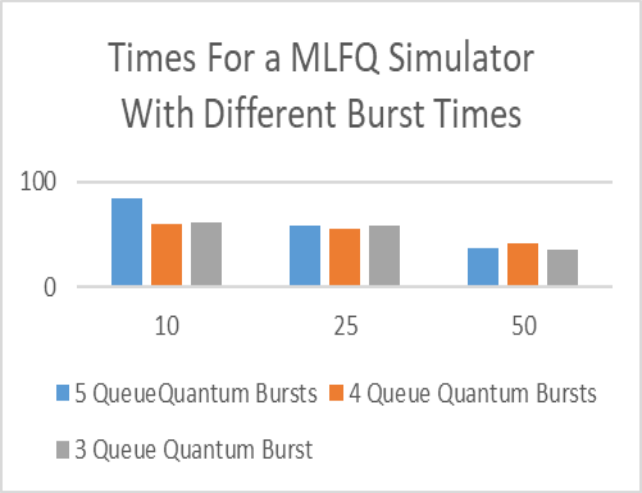

Figure 2

For a quantum of 10 seconds we find that the 5 queue simulator takes a much longer time while the 4 queue and 3 queue are realitivily close. Every single simulation at the different burst times has a front runner which means that having a variety of different queues and a different quantum burst time can achieve different results. We then tried to see if we could predict the time given an amount of burst. In Figure 2.1 we have create a polynomial equation to determine if we can trace the timeing of this.

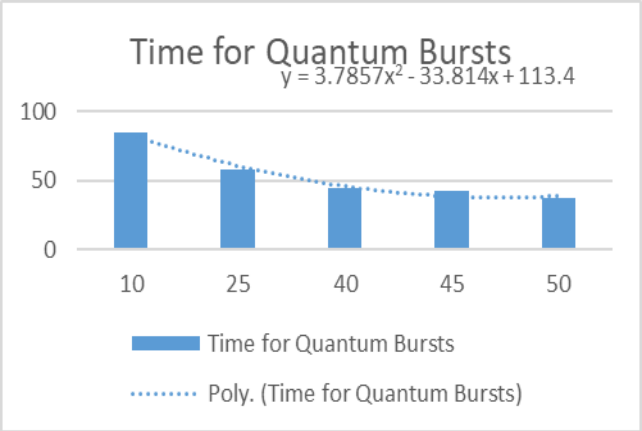

Figure 2.1

During the predicition we found that because the traffic on the thingys altered our time but we found the closest equation given under the title. As we increased the quantum burst everything only had to run once and thus was just able to run a basic loop without moving up and down queues.

We found the hardest part about the MFQS was handling the aging processes after a given time when the last queue was full of processes since it required to go through every process and add to it's aging number. In Figure 3 we have

the code regarding our aging process. The biggest problem with this code we have was having to go through the aging process of every processes in the last queue and then when it's been there for a certain amount of time move it up a process.

```
if (deq.size() > 0) {
                    for (n = 0; n < (int)deq.size(); n++) {

          deq.at(n).setWaitedTime(deq.at(n).getWaitedTime() - 1);
                        if (deq.at(n).getWaitedTime() == 0) {
                            deq.at(n).setQueue(numQueues - 2);

      queueList.at(numQueues.push(deq.at(n));
#ifdef DEBUG
                            cout << "Aged Process " << deq.at(n).getPid() << " at time " << time << endl;
#endif
                            deq.erase(deq.begin() + n);
                        }
                    }
          }
```

Figure 2.2

Running the simulator MFQS for a file the size of five hundred thousand processes takes only about 15 minutes as the total finish time is a little over twenty five million with all the bursts added together. Running the simulator gives us a final average wait time of around 12 million seconds and a turnaround time very similar to that as well. This makes sense as well since the vast majority of processes have to wait a long time to run at twenty five million seconds. I find that spreading out the arrival times over a period greater than ten thousand and over a couple million speeds up the simulator dramatically since it's not trying to handle so many at one time.

4 Windows Hybrid Scheduler

The Windows Hybrid Scheduler is a process scheduler which is comprised of 100 priority queues. This scheduler dynamically changes the priorities of processes. Queues 50-99 are the "kernel" processes with higher priorities being run first. Queues 0-49 are the "user" processes with lower priorities. Processes cannot switch between the lower and upper bands. Highest priority followed by lowest process ID is used to decide which process should run next. The current running process will continue to run until it's either finished or it's time quantum runs out. When it's time quantum runs out, the process's priority is decreased by its quantum. A process's priority will be increases when it runs I/O or when its aging timer expires while in the bottom 10 percent of queues. The priority is increased by either the amount of I/O clock ticks the process will take, or by 10 if the process is aged. A process's age timer is 100 clock ticks and is reset when the process is promoted. This scheduler must also take into account a process's original priority. The priority will not be dynamically promoted if the process is already in the maximum queue for its band. The priority will not be dynamically demoted if the process is already in the minimum queue for its band, or if the process will be demoted below its original priority. When a process does I/O, it is started on the second to last clock tick for that time quantum and process is sent to a wait queue where it is run simultaneously with a new process from the regular queues. When the I/O is finished, the process is sent to its new queue, which is determined by its dynamically increased priority.

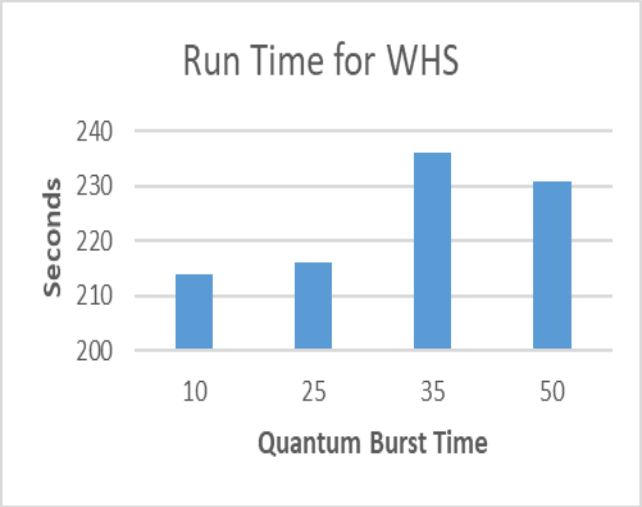

Figure 3

After simulating our WHS with 10,000 processes and various quantum burst times. We found that in general, increasing the quantum burst will increase the amount of time it takes to run the simulation. A different quantum burst drastically changes the amount of time needed to run. This is because I/O will require more time for that process to finish. I/O will also only be run one second before the end of the burst, so if a process finishes before the end of the burst, the I/O will never be run. All Processes must also wait 1 clock tick to context switch between background processes and I/O. Meaning that when there is more I/O, the time will drastically increase to complete those

processes. The total run time also increased when more processes are introduced. This is due to the fact that there will be as much as 10% of all the processes aging and getting rearrange, along with I/O processes getting rearranged. Dynamically altering so many processes slows down the speed of this scheduler significantly. Currently, we are unable to fully run 500,000 processes simply because it is too slow and would take half a day to complete.

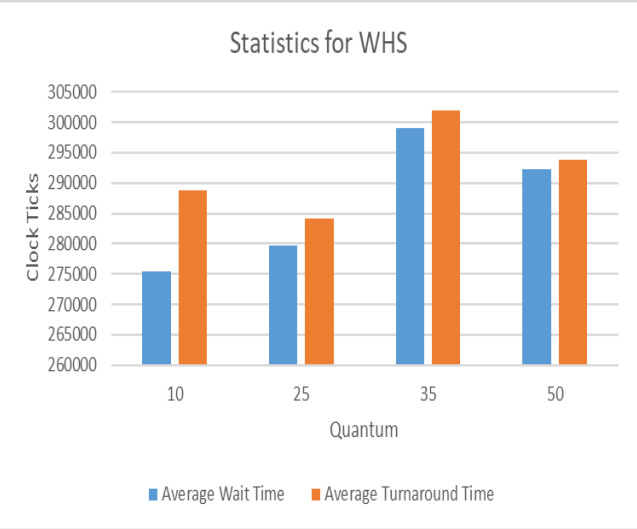

Figure 3.1

We also gathered stats on average wait time and average tournaround time for all processes that completed. For 10,000 processes running, the results can be seen in figure 3.1. We found that as the quantum is increased, the difference between the average wait time and the average turnaround time decreases. This makes sense because as the quantum increases, processes with longer burst times and I/O will not run their I/O because they will finish before their I/O runs. It's interesting why the average wait time and average tournaround time are much higher at a quantum of 35. We believe this is due to the burst times of a majority of processes being close to 35, meaning that any of these processes that have I/O will now be required to wait and will most likely get demoted to the same queue since they can't be demoted below their band of 50 priorities. This also aligns with our data for the total run time at each various burst time. The amount of time it took to run the simulation with a quantum of 35 was the longest and lines up with the longest waiting times for all processes.

5 Summary

The Real Time Scheduler is best for completing short processes very quickly. When running our RTS with 500,000 processes, it completes only 3795 of them, but it does this in a matter of seconds. Unfortunately this is not ideal because it simply throws away any process that cannot finish by its deadline, and so in this scenario where all processes arrive within 10,000 clock ticks, most of the processes are never run. Hard Real Time Scheduler is even worse and will stop completely as soon as any process will not complete. This occurs with our example after only 1 process is completed. For Multi-level Feedback Queue Scheduler, all processes are always run and the simulation isn't complete until all processes have finished. It is relatively fast for completing all 500,000 processes and is much more complex than RTS. The process to run in MFQS is more even due to the fact that processes that have run through their burst are demoted to a lower queue that will have a smaller burst. After processes are in the lowest queue, they can age and get promoted back up to a higher queue. This allows a more fair process allocation algorithm. The third scheduler, Windows Hybrid Scheduler, is similar to MFQS, but adds an extra layer of complexity. With 100 queues and processes within those queues being dynamically changed, this scheduler runs the slowest of the three. It was the most challenging to get to work right and also deals with I/O for any process having an I/O greater than 0. It finished all valid processes but has an even larger wait time than MFQS. Overall we think that the best scheduler in terms of both speed and processes finished is MFQS.