**Lab 3 - Process Schedulers**

Christian Poynter and

John Frederick

[poyntecd@uwec.edu](mailto:poyntecd@uwec.edu)

[frederja@uwec.edu](mailto:frederja@uwec.edu)

**Abstract**

*In this lab, we set forward to implement three different process schedulers in C++. While we were not forging any new ground in the field of Computer Science, it was our first interaction with schedulers and our second program using the language of C++. We successfully accomplished our objective, and implemented a Real-Time Scheduler, a Multi-Level Feedback Queue Scheduler, and a Hybrid Windows Scheduler. Our Hybrid Windows scheduler does not handle large sets of processes, however.*

**1. Introduction**

The purpose of this document is to evaluate the results of the three schedulers. We will look at the average waiting time, average turnaround time, and total time taken for each scheduler and compare them. We will then draw our conclusions on which scheduler is most efficient and which is most fair.

**2. Details/Process**

**2.1. Real Time Scheduler (RTS)**

The RTS uses the deadlines of the processes in queue to determine which process to run at any given time. The process with the nearest deadline is run until it either finishes normally or is interrupted by another process with a sooner deadline.

We tested the RTS with three different input files, shown in section 6. The first was *testfile1*, the test file provided with the assignment. This is a very basic set of 10 processes, only 7 of which are valid processes (3 have negative values in them). These processes do not overlap, so it provided a good test of the scheduler being able to simply run the processes in order.

The second test file was *testfile2*, which is based off of the slides for the class, with 2 extra processes

added. These processes demonstrate the capabilities of the scheduler to dynamically change what process is running based on the process’ deadline, and also the

ability to terminate processes that run out of time to complete.

The third and final test file was *1m\_processes*, a list of 1,000,000 processes provided by Dr. Jack Tan. These processes were randomly generated, and they cover every case. They also place a large load on the program, demonstrating its efficiency and exposing any memory leaks (of which we found none).

**2.2. Multi-Level Feedback Queue Scheduler (MFQS)**

The MFQS uses between 2 and 5 queues in order to run the set of processes, with all queues being round-robin besides the final queue, which runs as a first-come-first-served queue.

This scheduler runs each process for a set time quantum according to the queue it is in. When the process has run for the time quantum, it is demoted to the next queue down.

This scheduler also allows processes to age up if it has not been run within a certain user-specified time. The process is allowed to age up to the second highest queue, where it will sit until it is given a time to run.

MFQS was tested with the same 3 input files specified under the RTS section, as well as the *MFQS starving* test file to specifically test the starvation handling of the program.

**2.3. Hybrid Windows Scheduler (HWS)**

The HWS puts processes into separate queues based on priority, with one queue for each priority level. A process can have a priority between 1 and 99, inclusive. Throughout the scheduler, each process’ priority is dynamically changed so as to make sure all processes run in a fair way.

The scheduler runs the highest priority queue using Round Robin. A process either goes into IO one clock tick before the time quantum, or, if it doesn’t do IO, breaks at the time quantum. Its priority is decremented by the amount of time spent running in its current time quantum. If it does not do IO, it simply gets added to a new priority queue. If it does do IO, it goes into the IO queue and stays there the duration of its IO. Its priority is then increased by the amount of time spent in IO, and it comes back into the appropriate priority queue. At no time does a process’ priority go below its original priority, and a process that starts with a priority less than 50 can never go above 50.

Every 100 clock ticks, the scheduler checks for any processes that haven’t run in the last 100 ticks, and increments their priority by 10 in the hopes that they will be able to run.

We ran this scheduler using the same 3 test files as the previous schedulers. We also ran the test file *HWS starving* to test the scheduler’s ability to handle starving processes. All results are shown in section 3.

**3. Results**

**3.1. Testfile**

**MFQS (3 queues, tq = 4, aging = 10):**

850 <- Process 1 -> 851 | 3980 <- Process 2 -> 3995 | 5148 <- Process 7 -> 5157 | 6529 <- Process 5 -> 6629 | 7262 <- Process 4 -> 7358 | 8095 <- Process 3 -> 8178 | 9068 <- Process 6 -> 9126 |

AWT: 0

ATT: 51.7143

Processes scheduled: 7

Time elapsed: 0

**MFQS (4 queues, tq = 5, aging = 11):**

850 <- Process 1 -> 851 | 3980 <- Process 2 -> 3995 | 5148 <- Process 7 -> 5157 | 6529 <- Process 5 -> 6629 | 7262 <- Process 4 -> 7358 | 8095 <- Process 3 -> 8178 | 9068 <- Process 6 -> 9126 |

AWT: 0

ATT: 51.7143

Processes scheduled: 7

Time elapsed: 0

**RTS:**

850 <- Process 1 -> 851 | 3980 <- Process 2 -> 3995 | 5148 <- Process 7 -> 5157 | 6529 <- Process 5 -> 6629 | 7262 <- Process 4 -> 7358 | 8095 <- Process 3 -> 8178 | 9068 <- Process 6 -> 9126

AWT: 0

ATT: 51.7143

Processes scheduled: 7

Processes finished: 7

Time elapsed: 0

**HWS (tq = 4):**

850 <- Process 1 -> 851 | 3980 <- Process 2 -> 4003 | 5148 <- Process 7 -> 5165 | 6529 <- Process 5 -> 6629 | 7262 <- Process 4 -> 7358 | 8095 <- Process 3 -> 8178 | 9068 <- Process 6 -> 9126

AWT: 0

ATT: 53

Processes scheduled: 7

Time elapsed: 0

**HWS (tq = 8):**

850 <- Process 1 -> 851 | 3980 <- Process 2 -> 3999 | 5148 <- Process 7 -> 5161 | 6529 <- Process 5 -> 6629 | 7262 <- Process 4 -> 7358 | 8095 <- Process 3 -> 8178 | 9068 <- Process 6 -> 9126 |

AWT: 0

ATT: 51.8571

Processes scheduled: 7

Time elapsed: 1

**3.2. Testfile2**

**RTS:**

0 <- Process 1 -> 1 <- Process 6 -> 2 <- Process 2 -> 3 <- Process 3 -> 4 | 4 <- Process 2 -> 8 | 8 <- Process 4 -> 12 | 12 <- Process 6 -> 13 | 13 <- Process 5 -> 20 |

AWT: 5.6

ATT: 7.2

Processes scheduled: 6

Processes finished: 5

Time elapsed: 0

**MFQS (3 queues, tq = 4, aging = 10):**

0 <- Process 1 -> 4 | 4 <- Process 6 -> 6 | 6 <- Process 2 -> 10 | 10 <- Process 3 -> 11 | 11 <- Process 5 -> 15 | 15 <- Process 4 -> 19 | 19 <- Process 1 -> 20 | 20 <- Process 2 -> 21 | 21 <- Process 5 -> 24 |

AWT: 10.1667

ATT: 14.1667

Processes scheduled: 6

Time elapsed: 0

**MFQS (4 queues, tq = 5, aging = 11):**

0 <- Process 1 -> 5 | 5 <- Process 6 -> 7 | 7 <- Process 2 -> 12 | 12 <- Process 3 -> 13 | 13 <- Process 5 -> 18 | 18 <- Process 4 -> 22 | 22 <- Process 5 -> 24 |

AWT: 7.16667

ATT: 11.1667

Processes scheduled: 6

Time elapsed: 0

**HWS (tq = 4):**

0 <- Process 1 -> 1 | 1 <- Process 6 -> 2 | 2 <- Process 2 -> 5 | 5 <- Process 4 -> 7 | 7 <- Process 2 -> 9 | 9 <- Process 4 -> 10 | 10 <- Process 6 -> 11 | 11 <- Process 3 -> 12 | 12 <- Process 1 -> 14 | 14 <- Process 5 -> 15 | 15 <- Process 4 -> 16 | 16 <- Process 5 -> 17 | 17 <- Process 1 -> 19 | 19 <- Process 5 -> 36 |

AWT: 7.66667

ATT: 13.5

Processes scheduled: 6

Time elapsed: 0

**HWS (tq = 8):**

0 <- Process 1 -> 1 | 1 <- Process 6 -> 2 | 2 <- Process 2 -> 7 | 7 <- Process 4 -> 11 | 11 <- Process 6 -> 12 | 12 <- Process 3 -> 13 | 13 <- Process 1 -> 17 | 17 <- Process 5 -> 24 |

AWT: 6.5

ATT: 10.3333

Processes scheduled: 6

Time elapsed: 0

**3.3. 1m\_processes**

**MFQS (3 queues, tq = 4, aging time = 10):**

AWT: 3.34451e+07

ATT: 3.34452e+07

Processes scheduled: 997000

Time elapsed: 88

**MFQS (4 queues, tq = 5, aging = 11):**

AWT: 3.3423e+07

ATT: 3.3423e+07

Processes scheduled: 997000

Time elapsed: 90

**RTS:**

AWT: 1030.69

ATT: 5.64057

Processes scheduled: 997000

Processes finished: 5517

Time elapsed: 58

**HWS (tq = 100):**

**3.4. HWS starving**

**HWS (tq = 4):**

0 <- Process 1 -> 1 | 1 <- Process 6 -> 2 | 2 <- Process 2 -> 5 | 5 <- Process 4 -> 7 | 7 <- Process 2 -> 9 | 9 <- Process 4 -> 10 | 10 <- Process 6 -> 12 | 12 <- Process 3 -> 15 | 15 <- Process 4 -> 16 | 16 <- Process 6 -> 19 | 19 <- Process 3 -> 23 | 23 <- Process 6 -> 26 | 26 <- Process 3 -> 30 | 30 <- Process 6 -> 33 | 33 <- Process 3 -> 37 | 37 <- Process 6 -> 40 | 40 <- Process 3 -> 44 | 44 <- Process 6 -> 47 | 47 <- Process 3 -> 51 | 51 <- Process 6 -> 53 | 53 <- Process 3 -> 100 | 100 <- Process 5 -> 103 | 103 <- Process 3 -> 109 | 109 <- Process 5 -> 112 | 112 <- Process 3 -> 118 | 118 <- Process 5 -> 119 | 119 <- Process 3 -> 200 | 200 <- Process 1 -> 202 | 202 <- Process 3 -> 205 | 205 <- Process 1 -> 207 | 207 <- Process 3 -> 241 |

AWT: 63.8333

ATT: 103.833

Processes scheduled: 6 6

Time elapsed: 0

**HWS (tq = 8):**

0 <- Process 1 -> 1 | 1 <- Process 6 -> 2 | 2 <- Process 2 -> 7 | 7 | 7 <- Process 4 -> 11 | 11 <- Process 6 -> 17 | 17 <- Process 3 -> 21 | 21 <- Process 6 -> 28 | 28 <- Process 3 -> 32 | 32 <- Process 6 -> 38 | 38 <- Process 3 -> 100 | 100 <- Process 5 -> 107 | 107 <- Process 3 -> 200 | 200 <- Process 1 -> 204 | 204 <- Process 3 -> 241 |

AWT: 57.6667

ATT: 97.6667

Processes scheduled: 6 6

Time elapsed: 0

**3.5. MFQS starving**

**MFQS (3 queues, tq=4, aging = 10):**

0 <- Process 1 -> 4 | 4 <- Process 2 -> 8 | 8 <- Process 6 -> 12 | 12 <- Process 5 -> 16 | 16 <- Process 4 -> 20 | 20 <- Process 3 -> 24 | 24 <- Process 1 -> 25 | 25 <- Process 2 -> 26 | 26 <- Process 6 -> 34 | 34 <- Process 5 -> 37 | 37 <- Process 3 -> 45 | 45 <- Process 6 -> 53 | 53 <- Process 3 -> 91 | 91 <- Process 6 -> 101 |

AWT: 29.5

ATT: 46.3333

Processes scheduled: 6

Time elapsed: 0

**MFQS (4 queues, tq=5, aging = 11):**

0 <- Process 1 -> 5 | 5 <- Process 2 -> 10 | 10 <- Process 6 -> 15 | 15 <- Process 5 -> 20 | 20 <- Process 4 -> 24 | 24 <- Process 3 -> 29 | 29 <- Process 6 -> 39 | 39 <- Process 5 -> 41 | 41 <- Process 3 -> 51 | 51 <- Process 6 -> 61 | 61 <- Process 3 -> 81 | 81 <- Process 6 -> 86 | 86 <- Process 3 -> 101 |

AWT: 24

ATT: 40.8333

Processes scheduled: 6

Time elapsed: 0

**4. Comparison/Analysis**

For the first test file, all of the schedulers run in a decent amount of time. The overall differences were negligible at most, making them all useable for small cases with no conflicts. The processes had no overlap, so the only differences were when the Hybrid scheduler was run with a higher time quantum, thus using less IO and resulting in slightly faster turnaround times.

As for the second test file, the differences in the schedulers start to become apparent. RTS, for example, has better wait and turnaround times at the expense of dropping a process because it couldn’t finish before the respective deadline. MFQS and HWS still run decently well. Both have better wait and turnaround times when using a longer time quantum, and more queues helps the HWS. Overall, the Hybrid scheduler does the best on this set of processes, with shorter wait and turnaround times than the MFQS, but completing all the processes, something that the RTS did not do.

Finally, when it comes to the million process test, times definitely show a difference. RTS drops a lot of processes when running, which can become very problematic, but has reasonable wait and turnaround times for the processes that do finish. MFQS, though it takes more time than RTS and has much longer wait and turnaround times (due to a queue having to be empty before moving to the next queue), finishes all processes in a relatively reasonable amount of time. HWS also runs all the processes, but takes significantly longer than RTS to do so.

An area of this assignment that was somewhat experimental in nature was the way we implemented the MFQS. When implementing the MFQS, there are different algorithms for aging processes. The more common way is to only age up from the last queue. However, we implemented our MFQS so as to age a process up from any queue, except from the second to first queue. The advantage of this is that processes that have been in the queue longer get a better chance of finishing faster. However, when they do this, they are keeping new processes coming into that queue from running. This could happen to an extent that new processes wait for significant amounts of time before they can run. In the sake of fairness, This is not the best way of implementing aging. If given the choice (the way we did it was given by the assignment specifications), we would limit aging to be from the last queue to the queue above it only.

**5. Conclusion**

In conclusion, each scheduler has its strengths and weaknesses. The Real Time Scheduler ensures very fast total runtime, and does its best to complete each process before its deadline. If a process cannot do so, it is aborted. This can obviously cause problems.

The Multi-Level Feedback Queue Scheduler is a very fair scheduler. Every process is given the same amount of time to run, and if they can not finish in that time, they are given another chance in the next queue with more time. If a process gets ignored at the end of the line, so to speak, it gets to move back up a level to get a better chance. This fairness comes at the cost of speed and efficiency, and area in which this scheduler is lacking.

The Hybrid Windows Scheduler is priority-based. It gives processes with higher priority preference over those with lower priority. This scheduler also has to take IO into consideration. The end result varies considerable depending on the amount of time processes spend in IO. Overall the HWS performs similarly to the MFQS, spending much of its time in IO in the same way MFQS spends much of its time waiting for processes in large queues.

If given more time, we would work more on the Hybrid Scheduler. It does not work correctly if given large sets of processes, and we are sure it is a simple fix.

**6. Test Files**

**testfile:**

Pid Bst Arr Pri Dline I/O

1 1 850 61 852 3

2 15 3980 82 3998 2

3 83 8095 51 8179 0

4 96 7262 66 7365 0

5 100 6529 1 6630 0

6 58 9068 24 9132 0

7 9 5148 35 5166 4

8 -8 5924 3 5925 0

9 100 -1 34 101 0

10 72 1918 43 -5 0

**testfile2:**

Pid Bst Arr Pri Dline I/O

1 5 0 10 15 3

2 5 2 56 8 2

3 1 3 24 6 10

4 4 5 33 12 5

5 7 5 7 26 6

6 2 1 24 14 4

**HWS starving:**

Pid Bst Arr Pri Dline I/O

1 5 0 10 15 3

2 5 2 56 8 2

3 200 3 24 6 0

4 4 5 33 12 5

5 7 5 20 26 6

6 20 1 24 14 4

**MFQS starving:**

Pid Bst Arr Pri Dline I/O

1 5 0 10 15 3

2 5 2 56 8 2

3 50 7 24 6 0

4 4 5 33 12 5

5 7 5 20 26 6

6 30 3 24 14 4

**1m\_processes:**

Pid Bst Arr Pri Dline I/O

1 1 1179 72 1187 1

2 78 543 2 621 0

3 3 2158 71 2170 0

4 30 6270 25 6309 0

5 3 7367 50 7373 4

6 20 7807 27 7831 0

7 42 3589 22 3638 0

8 -8 5923 82 5919 5

9 13 -1 14 16 3

10 17 8063 7 -5 0

11 82 6245 33 6334 4

12 43 9888 69 9932 0

13 51 2854 8 2914 0

14 71 3453 86 3527 0

…

999986 25 4520 15 4549 0

999987 59 2970 42 3031 0

999988 76 2425 24 2510 3

999989 68 7033 26 7109 0

999990 95 4775 47 4874 0

999991 91 2788 8 2886 0

999992 17 6090 80 6114 5

999993 37 7098 58 7142 5

999994 88 9397 60 9486 0

999995 24 6594 25 6627 0

999996 72 4348 11 4422 4

999997 85 6823 11 6916 5

999998 46 2246 17 2296 4

999999 85 7497 46 7590 0

1000000 1 7680 16 7688 0