Programming Assignment 3:

Investigating the Linux Scheduler

CSCI 3753 – Operating Systems

University of Colorado Boulder

Fall 2014

**ABSTRACT**

In this lab, we want to compare the effectiveness of the hierarchically organized Real Time and Completely Fair (CFS) scheduler classes

**INTRODUCTION**

In this lab, we want to compare the effectiveness of the hierarchically organized Real Time and Completely Fair (CFS) scheduler classes contained in the Linux kernel. To do so, we are benchmarking and analyzing the behavior of three Linux Scheduling policies: SCHED\_OTHER (included in the CFS scheduler), SCHED\_FIFO and SCHED\_RR (both included in the Real Time scheduler). The latter policies implement respectively real-time first-in-first-out and real-time round-robin scheduling ones whereas the former one implements the standard CFS time-sharing scheduling parameters. Tests were undergone on the CUCS Virtual Image for Fall 2014, running a 64-bit Ubuntu with base memory of 2048 MB and 2 processors.

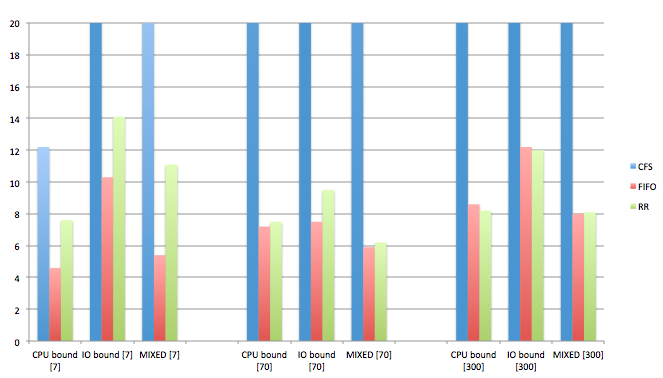
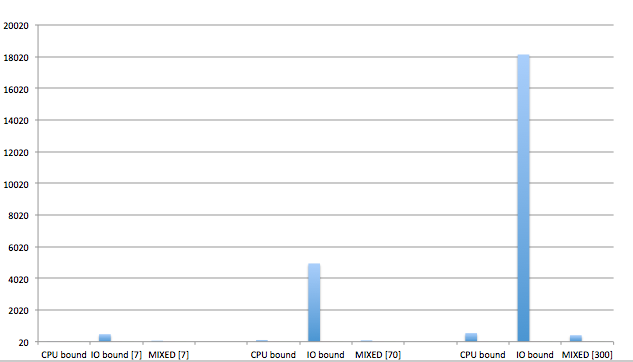
**METHOD**

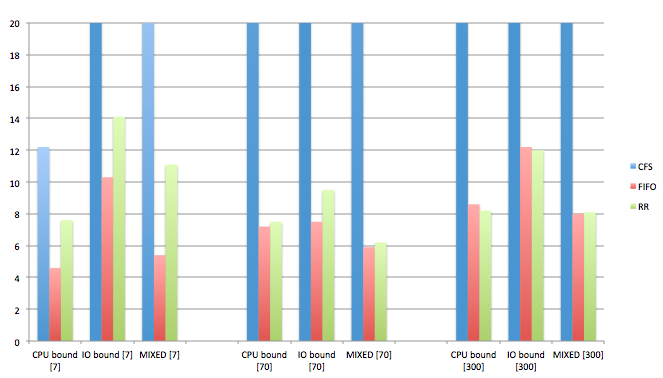
In order to compare those policies we are running their behaviors through a set three different code implementation across three representative process types adequately. Firstly, a CPU-bound process type where we implement a source code for a statically based pi calculator, repeating 100,000 times to effectively hog the CPU. Secondly, an I/O-bound process type where we implement a source code for a program that writes N bytes in blocks of K bytes from an input file to an output file, using the low-level read and write system calls in the O\_SYNC mode to minimize the effects of filesystem buffering and maximize I/O delays. In order to generate the required number of N bytes when the size of the output file is smaller than N, the former input file will be read multiple times by the program. Thirdly, a mixed process type where the pi and piCircle calculator results are written in a “junk.txt” output file thus utilizing both the CPU-bound and I/O-bound process types.

Furthermore, we are investigating how each process type will be affected under each scheduling policy scales across three levels of system utilization: a low scale of 7, a medium scale of 70 and lastly and high scale of 300 simultaneous process instances.

**RESULT**

In order to compare those policies we are running their behaviors through a set three different code implementation across three representative process types adequately.





**ANALYSIS**

The three different processes (CPU, IO, Mixed) in terms of wall (turnaround) time, each benefit from different schedulers:

For CPU bound processes both the CFS and FIFO schedulers perform about the same for each benchmark scaling: For the small and medium scaled benchmarks, there is very little discrepancy between the wall times, but at large process scaling, FIFO performed .41 seconds faster than the CFS scheduler. For IO bound processes the best two schedulers were the real time schedulers (FIFO and RR). FIFO performed best at small/medium process scaling with the lowest average time, but at large scale, the RR scheduler was clearly faster (on average 26.266 seconds faster than FIFO). Mixed processes on the other hand benefited best from CFS at small and large scale processing, and best from RR at medium scale processing.

Even though RR and FIFO perform better than CFS  in terms of wall time, it is significant to note that CFS *consistently* performs better than the other two in terms of CPU Utilization. For every test performed, on average CFS used approximately 9.09444% less CPU than the other two schedulers.

Furthermore, each scheduling policy performed differently during scaling. For the CFS scheduler, the performance was fairly reasonable for every type of process except for IO bound processes. For IO bound processes, the number of context switches grew seemingly exponentially as the number of processes were increased. Likewise, the turnaround time for CFS also went up as the number of processes was being scaled. For the FIFO scheduler, it performed the best in terms of wall times for small and medium scale processing but not at the large scale. Similarly in terms of context switches, FIFO seemed to have the least amount of context switching in small and medium scale computing but at large scale, it kept on par with the RR scheduler. Finally, for the RR scheduler, it performed the best at large scale computing (i.e. had the least amount of context switches overall and lowest overall wall times at large scale). In terms of CPU utilization, all of these had consistent data points as they scaled upwards (i.e. there was not a lot of variance in the data as the processing was scaled).

To summarize the three different schedulers, the following will explain situations in which each scheduler would perform well and situations in which each scheduler would perform poorly:

The CFS scheduler would perform the best when CPU performance is a priority. When wall times are a priority or minimizing involuntary context switches are a priority at scale then the CFS would perform poorly. The FIFO scheduler would perform the best when wall times and minimizing involuntary context switches are a priority at small or medium scale computing. At large scale computing the FIFO scheduler will perform worse than the RR scheduler but better than the CFS scheduler if wall times and involuntary context switches are a priority. In terms of the RR scheduler it will perform the best at large scale computing if wall time or minimizing involuntary context switches are a priority. It will perform poorly if utilized at small or medium scale computing.

**CONCLUSION**

From the results, it is apparent that the CFS scheduler performs the best in terms of CPU utilization, but at large scale it performs the worst in terms of IO bound processes. Otherwise, the CFS seems to be the most reliable because of its consistency in CPU utilization and fairly consistent wall time. In terms of the other two schedulers, RR performs the best at large scale computing (wall time) and FIFO performs the best at small and medium scale computing (in terms of involuntary context switching and wall times). Yet these two schedulers use a pretty significant amount of CPU more than the CFS scheduler.

**REFERENCES**

<http://www.cs.duke.edu/courses/spring01/cps110/slides/interleave/sld008.htm>

**APPENDIX A – Raw Data**

kerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvor33refh343erlfpwegrfipwrfikerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvorekerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvorekerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvorekerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvore kerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvor33refh343erlfpwegrfipwrfikerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvorekerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvorekerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvorekerfhersivfroefousehgouhseourghsofrosehfouhseofvoehvoeshfroaehrvore

**APPENDIX B – Description of Code**

**rw.c** – This file has been modified from the original rw.c where the first two parameters taken from the user are now <number of processes> and <scheduling policy> (This pushed the original parameters back by two). This was also done for the load testing of the different benchmarks.

**pi-sched.c** – This file has been modified from the original pi-sched.c, where the user specifies how many processes will be forked when the calculation of pi starts. This was done for load testing the different benchmarks.

**mixed.c –** This file was created by using the reworked pi-sched.c file. The difference is that this file writes values (pi and piCircle) calculated by pi-sched to a file. This makes it such that the process is now mixed bound in that the CPU is being used to calculate pi and there is an IO factor with writing the values to a text file.

**Makefile –** The only edit to this file was to add mixed.c to the compilation so that it can be used as an executable for the runscript.

**runscript –** This file is a shell script that runs the benchmark tests for the small, medium, and large (7, 70, 300 processes respectively) load testing for each of the three schedulers and three different processes. This comes out to 27 benchmarks, which were each written to separate files in the data folder (note: this folder is cleared out every time the runscript is called again). These benchmarks were ran 10 times each (see Appendix A for more information).

**data directory** - This directory contains all the data that was outputted by the runscript.

**saved\_data directory** -This directory was created to save all the data that was put in the data directory, since it gets cleared out every time the runscript is executed.