Sean Moss
CSCI 3753
Operating Systems
Spring 2015
Dr. Richard Han

Programming Assignment #4 – Scheduler Benchmarking

For this project, I worked on gathering runtime statistics for three different program types: CPU, IO, and mixed, with three different amounts of child processes, and with three different scheduler policies. Specifically, I gathered the amount of voluntary and involuntary context switches, as well as the average time that each process spent in both user time and real time, the difference between those two being discussed later in this paper. I wrote and ran tests using bash scripting, and generated the graphs using python. By looking at the graphs, I was able to draw conclusions as to the relative performances and recognize patterns that were expected to happen when mixing certain process types and scheduling policies.

Introduction

The programs that I wrote to test ended up being very similar to the suggestions given by the assignment write-up. The IO application was nearly identical, but I rolled my own "inefficient" random number generator to increase the CPU use for the CPU and mixed programs. I used the /use/bin/time program to gather runtime metrics for the various test cases, and output those metrics to files that I later parsed with python and used to generate graphs of the timing and context switch data. All of the requirements for running my code (minus the python graphing library, which is explained in the README), is included in the submission, and the full set of files I used is included in the Github repo I have been using for this assignment[1]. I have designed the submission to be runnable with only 2 commands (once you have installed plot.ly), which are outlined in the README.

Method

For the CPU bound program, I wrote my own implementation of the mersenne twister algorithm used to find random numbers, but I made it slightly more compute intensive to give an extra layer of CPU use. The code for this can be found in `programs/mersenne.h` and `programs/mersenne.c`. The standard implementation uses a cached list of 624 random numbers, and only regenerates these numbers every 624 times, as the list is used up[2], but my modified system forces the algorithm to regenerate the entire list every time a random number is needed. This increases the CPU use by the random number generator by a factor of 624.

Using my modified generator, I got 20,000 random numbers to make 10,000 pairs, and used the sqrt() function to calculate the distances between each successive pair of numbers, since the square root function is another CPU intensive function.

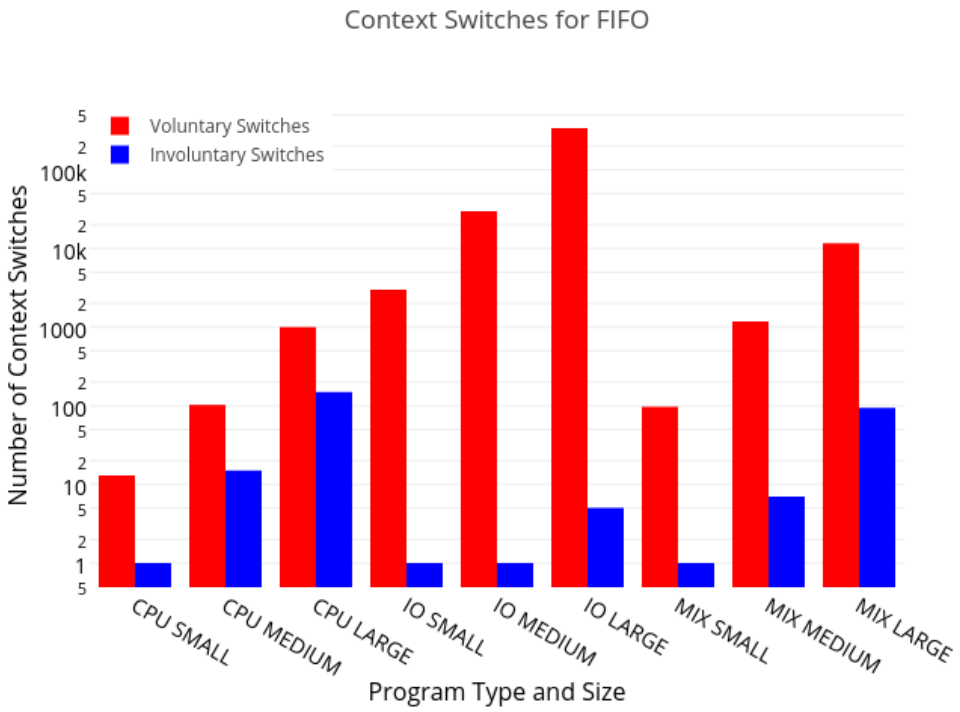
For the IO bound program, I simply chose to do what the write-up suggested, and wrote an application that read data from a common input file, and copied it a set number of times (in my case 10) to a specific output file. I drew heavily on the code provided in `rw.c` provided in the assignment files, but I modified it so it could set its scheduling policy, spawn children, and allow those children to each write to a specific output file, given by what number child they were spawned by the parent. I also took from the provided Makefile to generate a file filled with 1024 bytes of random data, which I used as the input file for the io application.

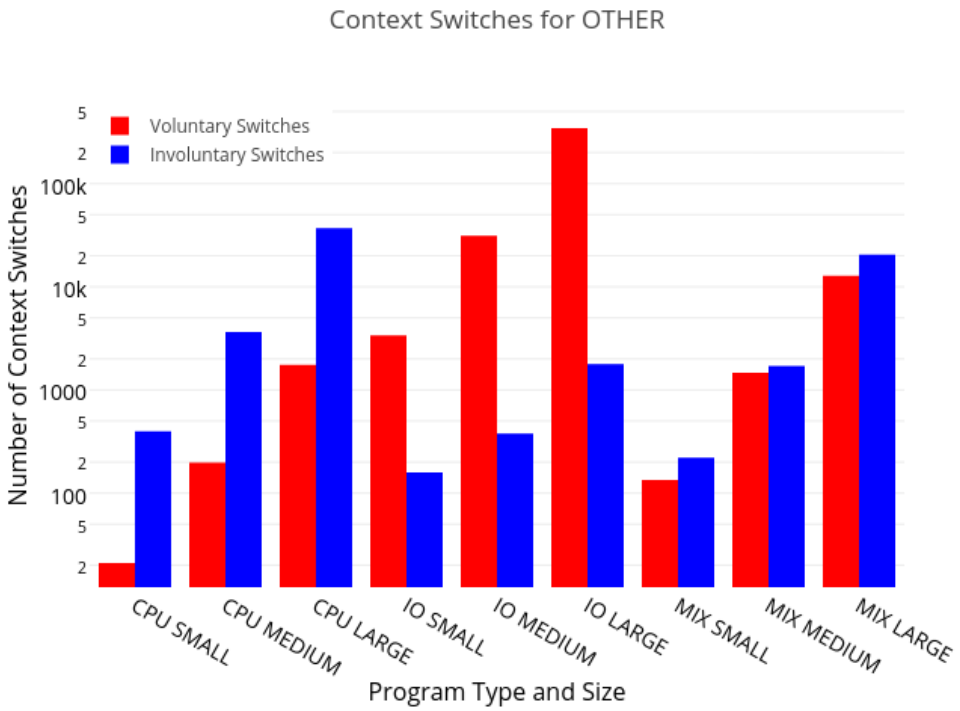
Finally, for the mixed program, I drew from both of the other programs. Each child process ran in two steps, the first one being to calculate a list of 1024 distances between the randomly generated points, like in my CPU program. A list of 1024 floats ends up being 4098 bytes, which I then proceeded to write out to a unique output file 4 times each, with the output file being named using the same process as the IO program.

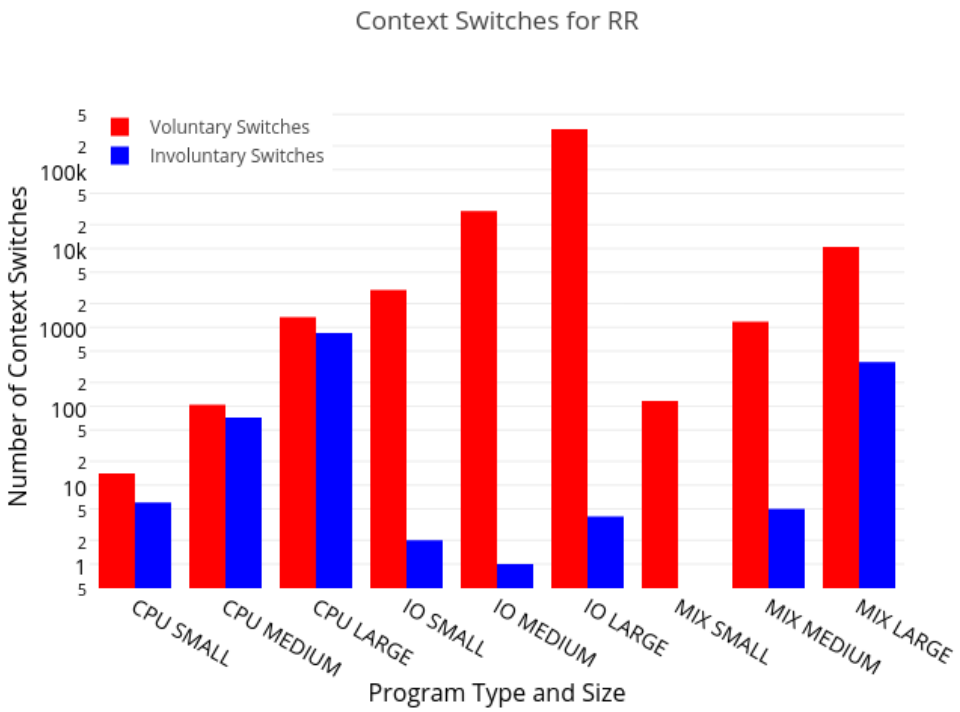
For my test suite, I wrote a shell script that generates all of the necessary folders and files for all of the tests, then runs each of the tests in order. This shell script can be found in my submission in the root folder, named `test.sh`, and instructions on running it can be found in the README provided. It works by first generating the output folders that the intermediate and output files go into during and after the tests, respectively. It then runs the Makefile found in the `programs` folder, which actually builds the test programs themselves, and is also in charge of generating the 1024 file full of random data used by the IO program. It then runs the test programs first by type: cpu, io, then mixed, then by size: small, medium, then large, and finally by scheduler policy: OTHER, FIFO, RR. For the sizes of the tests, I used 10 children for small, 100 children for medium, and 1000 children for large, as any more than 1000 children took an inordinate amount of time to finish, and anything more than about 2000 caused my VM to halt, and my host system to freeze up. As it was, running all 27 tests took a little over 20 minutes to finish. After finishing the tests, the script cleaned up all of the temporary files, and called the clean rule on the Makefile in `programs`. The output from /usr/bin/time was piped directly into files.

To generate the graphs, I used a python program instead of shell scripting so I had access to plot.ly[3], a graphing library for python that I have used before with good results. This python program can be found in the root directory of my submission, called `graph.py`, with instructions on how to run it in the README. This program only takes about 10 seconds to execute, and parses the data from the output files from the test script and generates the graphs found in the `report` folder. It generated 6 graphs, 3 with context switching data for each scheduler policy, and 3 with timing data for each scheduler policy.

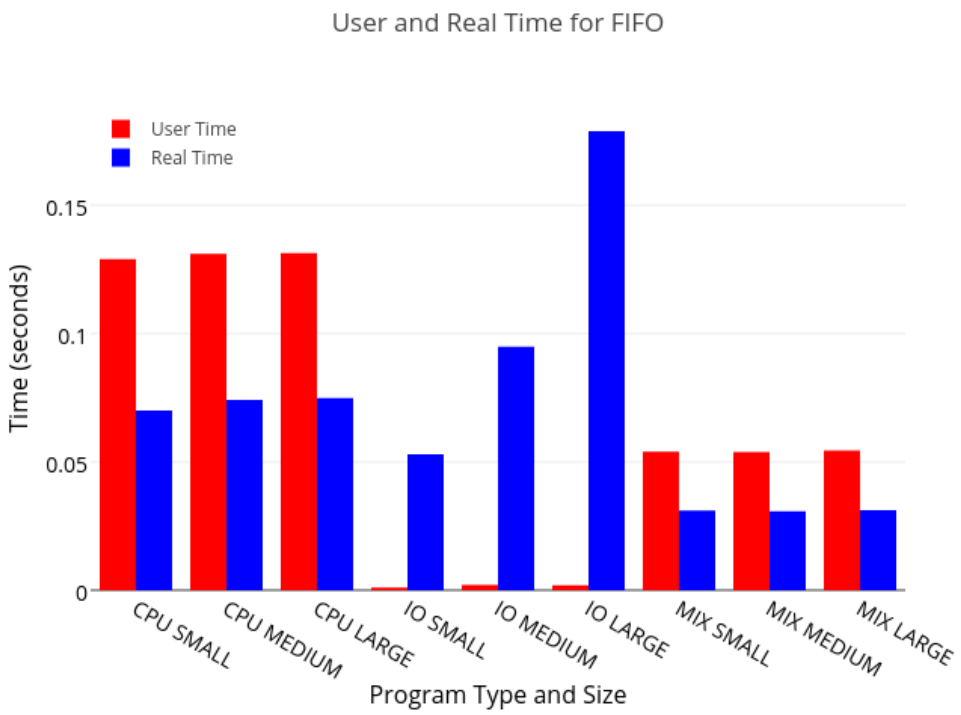
Results (Context Switches)

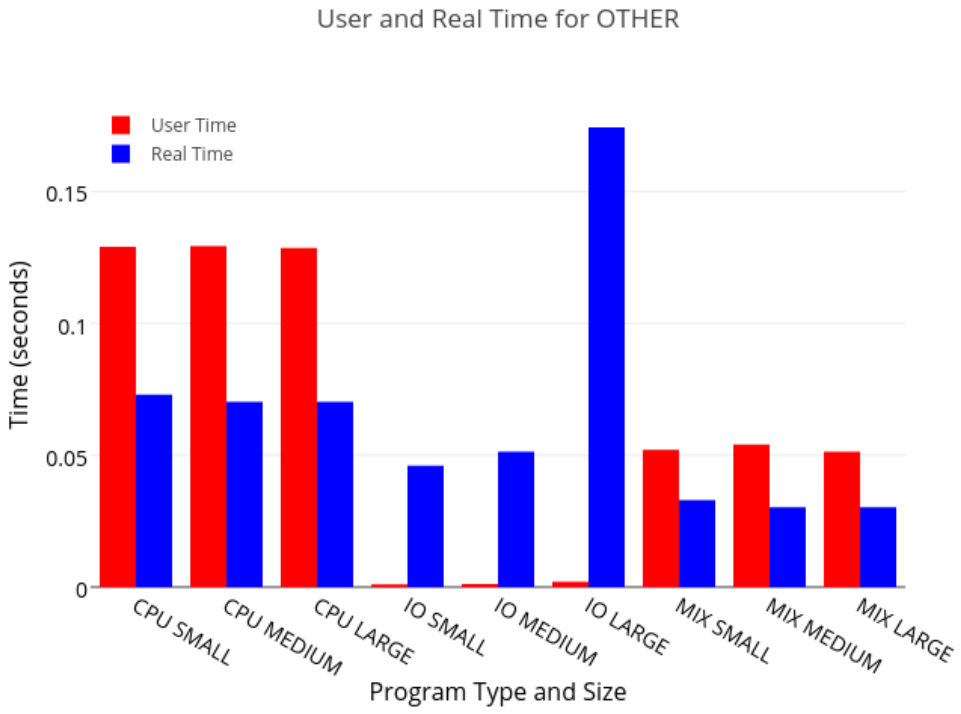


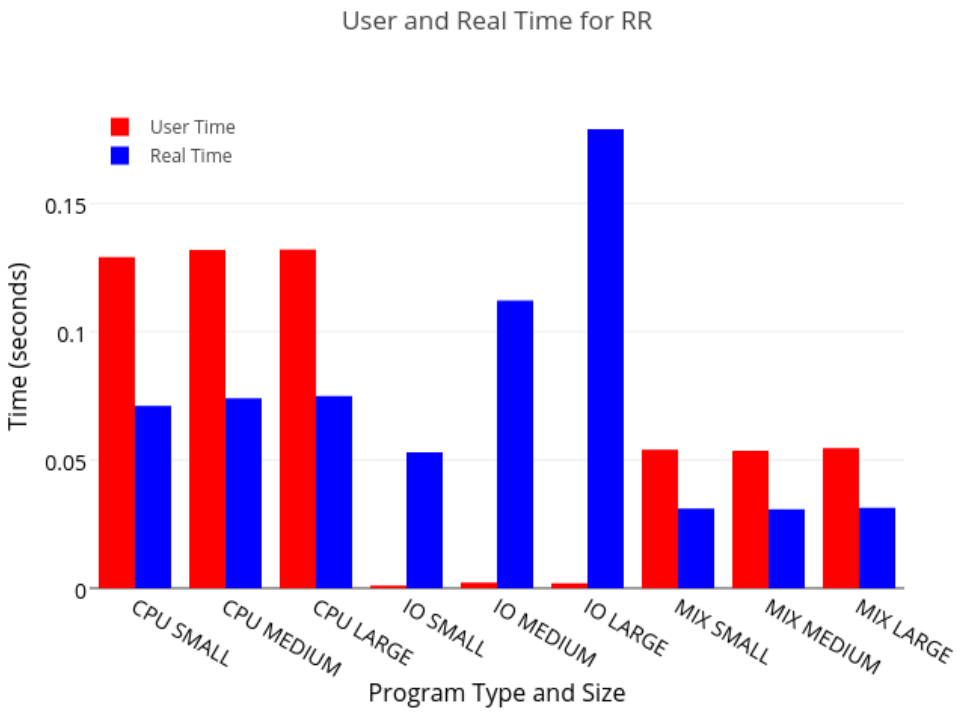



Context Switches for RR
Voluntary Switches
Involuntary Switches
Number of Context Switches
5
2
100k
5
2
10k
5
2
1000
5
2
100
5
2
10
5
2
1
5
CPU SMALL
CPU MEDIUM
CPU LARGE
IO SMALL
IO MEDIUM
IO LARGE
MIX SMALL
MIX MEDIUM
MIX LARGE
Program Type and Size

Results (Timing)






User and Real Time for RR
User Time
Real Time
0.15
0.1
0.05
0
Time (seconds)
CPU SMALL
CPU MEDIUM
CPU LARGE
IO SMALL
IO MEDIUM
IO LARGE
MIX SMALL
MIX MEDIUM
MIX LARGE
Program Type and Size

Analysis

Many of the results that we see represented in the graph are results that are expected based on what we know about the implementations of the scheduling policies and the runtime habits of the program types. The places where these patterns were the most obvious was when comparing the runtime statistics of IO bound programs with the other program types, and when comparing FIFO statistics with the other policies. When looking at the graphs, it is immediately obvious that there is a large data in the graph shapes for IO bound programs than for the other ones. For context switches, nearly 100% of all context switches were voluntary, and for time, the IO runtime was much higher than the IO user time. The context switch results are consistent with what we would expect to see from a program designed to have mostly blocking IO calls, where it would release its time at the IO call to allow other processes to run. The time is consistent with what we would expect, a higher real time, because of the slowness of disk IO, and a very low user time, since most of the IO call is spent in kernel space. Combining the two, we see IO programs that use FIFO policies have all but a very small handful of their context switches being voluntary.

We also see many similarities between the OTHER and RR policies, particularly in the time graphs, which can be easily explained by the similarities between the two policies. They both work on a principal of fixed time slices, with RR using priorities. However, since all tested processes were given the same highest priority, it made the RR and OTHER policies end up working nearly identically from each other, on the same principle of “if everyone is special, then no one is”.

Conclusion

The timing graphs in particular can show us information pertaining to the relative scaling abilities of the policies, as well as which policies were most efficient for which program type. Overall, all of the policies turned out to have very similar runtimes for all of the program types, which makes sense because all policies are designed to try to keep every process getting the same overall amount of time. All polices seem to scale well for CPU and mixed program types, and IO does not seem to scale well with any policy type, because of its dependency on a resource much more limited than the CPU, and while OTHER seems to initially scale the best for IO, it gets just as bad as the other policies at the large program, making me think that the results for the medium IO program for OTHER are perhaps erroneous. However, for which policy is better for which program type, there seems to be no clear winner time-wise for any policy with any program, because all policies gave an almost identical average process time for each process type. This being said, I also tried doing other tasks in a separate test run (who’s results I did not post here because of their irrelevancy), and the OTHER policy gave the best feeling of interactivity while the tasks were running, meaning that if the system must be used with other tasks at the same time, I would recommend sticking with the default scheduler.

Because of the incredibly close results between the scheduler policies, most of their pros and cons come from outside of their efficiency at their relative tasks. The most obvious application of this is with the responsiveness of the system while the application is running. With OTHER and RR, the system was still remotely responsive, and could be used on other tasks with relative success. FIFO, however, rendered the system completely unusable for any other tasks, particularly on the CPU and mixed programs. I could categorize the responsiveness of the policies as a pro for OTHER and RR, and a con for FIFO. If the results for the IO medium OTHER program turn out to be accurate, that would also give to the idea that OTHER is a good scheduler, particularly for IO heavy programs that don't get too big.

References

[1] – Github Repository for this Project: http://www.github.com/SeMo810/SchedulerTester

[2] – Mersenne Twister Pseudo-random Number Generator Algorithm adapted for C from:

http://en.wikipedia.org/wiki/Mersenne_Twister#Pseudocode

[3] – Plot.ly Python Graphing Library: http://plot.ly/

Appendix A: Raw Data

All raw data output files generated by /use/bin/time and used to make the graphs can be found in the `results` folder in the repository or the assignment submission. They are named as such: "<type>_<size>_<scheduler>.res", and are plaintext files containing the raw output.

Appendix B: Code Files

All C code and header files that were used to make the test programs can be found in the `programs` folder. Namely, there is `mersenne.c/.h`, which contains my custom implementation of the mersenne twister pseudo-random number generator, `sched_util.c/.h`, which contains code for parsing and setting scheduler policies, and `cpuprog.c`, `ioprog.c`, and `mxprog.c`, which are the test application code files.

The shell script used to run the Makefile, generate and clean folders and files, and run the tests themselves can be found in the root directory, named `test.sh`.

The python code used to parse the data files, and generate the graphs, can be found in the root directory, named `graph.py`.