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

**Programming Assignment #4 – Scheduler Benchmarking**

**Abstract**

**Introduction**

**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 up2, 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.

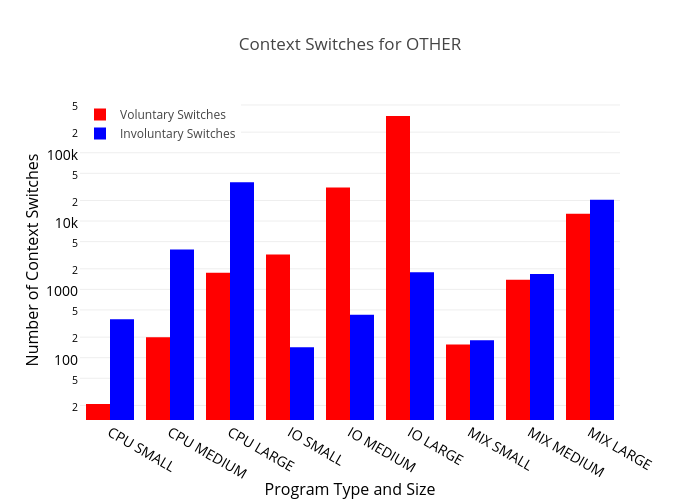
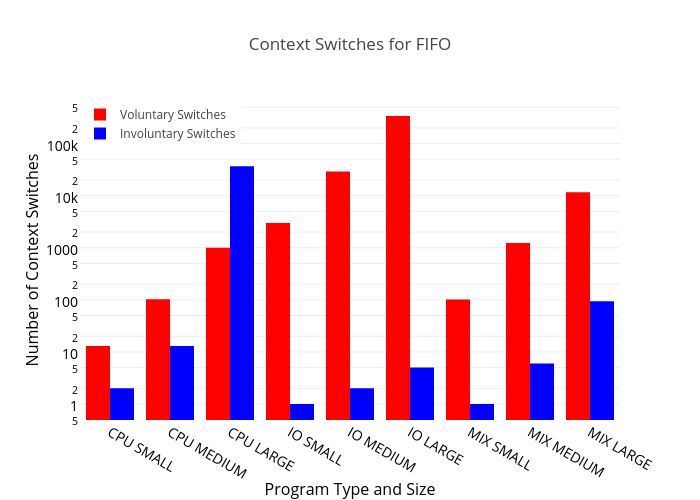
For the IO bound program, I simply chose to do what the write-up suggested, and wrote an application that read data from a common input file, and copied it a set number of times (in my case 10) to a specific output file. I drew heavily on the code provided in rw.c provided in the assignment files, but 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.

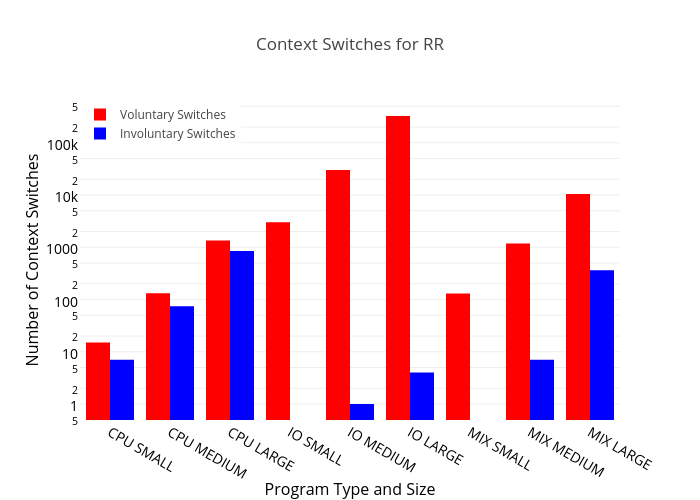
Finally, for the mixed program, I drew from both of the other programs. Each child process ran in two steps, the first one being to calculate a list of 1024 distances between the randomly generated points, like in my CPU program. A list of 1024 floats ends up being 4098 bytes, which I then proceeded 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.

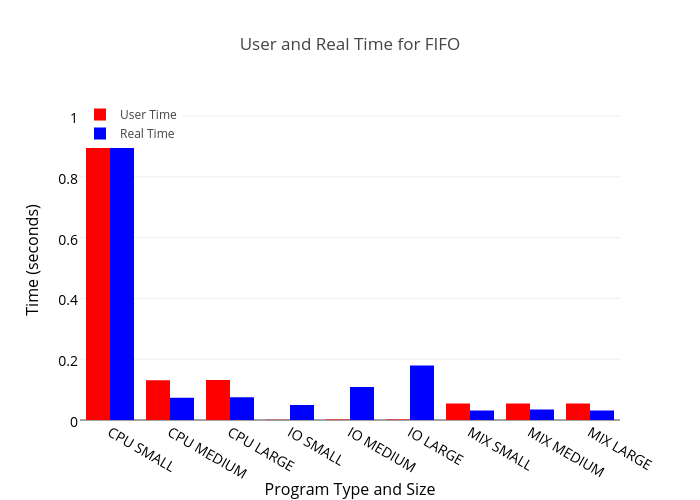
To generate the graphs, I used a python program instead of shell scripting so I had access to plot.ly3­, a graphing library for python that I have used before with good results. This python program can be found in the root directory of my submission, called graph.py, with instructions on how to run it 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.

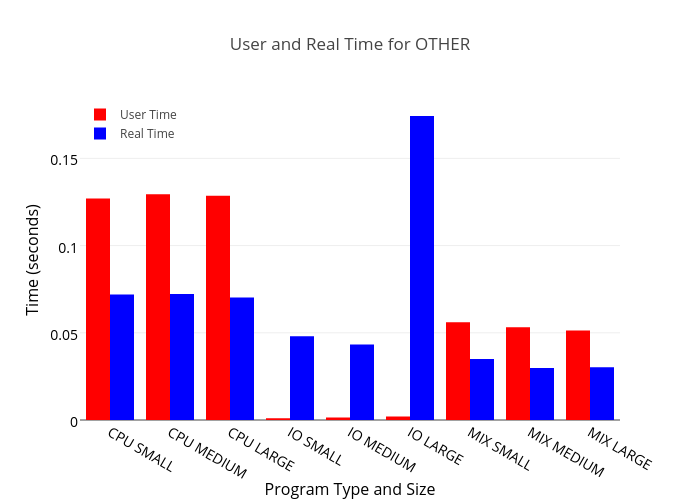
**Results (Context Switches)**

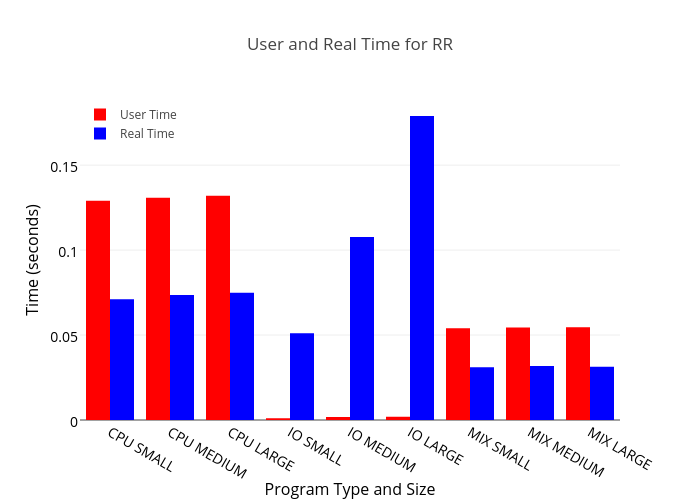




**Results (Timing)**







**Analysis**

**Conclusion**

**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.