CS3753 (Operating Systems)
Spring 2014
University of Colorado Boulder
Programming Assignment 3

1 ABSTRACT

The experiment that was conducted was about the efficiency of Linux process scheduling policies in relation to the type process, io-bound, cpu-bound, or a mix of both, used and the number of simultaneous processes.

I found that for CPU-bound processes the scheduler policy of *SCHED_FIFIO* had the fewest number of context switches over the different numbers of simultaneous processes and *SCHED_OTHER* had the smallest runtime. For I/O-bound processes, I found that *SCHED_FIFO* had the fewest number of context switches and *SCHED_FIFO* had the lowest runtime overall. Finally, for the mixed-process that does a combination of both CPU and I/O tasks, I found that *SCHED_FIFO* has the fewest number of context switches and *SCHED_OTHER* has the lowest runtime.

2 INTRODUCTION

Linux implements three different scheduling policies within two different scheduling classes. The CFS class has the *SCHED_OTHER* policy while the RT class contains the *SCHED_FIFO* and *SCHED_RR*. *SCHED_OTHER* is a Completely Fair Scheduler policy that uses time-slicing to manage processes. *SCHED_RR* is a round-robin scheduler policy while *SCHED_FIFO* is a first-in-first-out scheduler policy.

The experiment that was conducted looked at and measured the behavior of those three scheduling policies with three different kinds of processes and three different amounts of instances of the processes. The three different processes in this experiment were CPU-bound, a process that does mainly tasks using the CPU, I/O bound, a process that does mainly I/O tasks and does not require the CPU, and a mixed-process that does both kinds of tasks. The number of instances of those three different processes were broken into three categories: Low-10 simultaneous process instances; Medium-10s of simultaneous process instances; High-100s of simultaneous process instances.

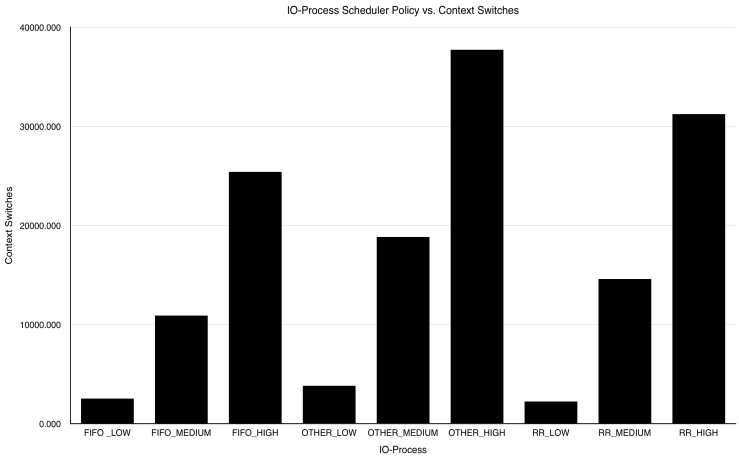
3 METHOD

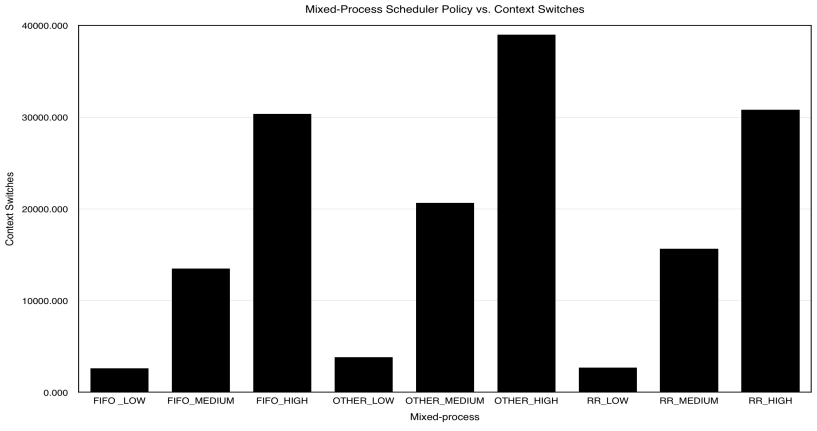
Three programs were used in this experiment. The first program performs a CPU-bound task by calculating the value of pi a certain amount of times. The second program performs an I/O bound task by reading bytes from an input file and outputting those bytes to a different file, almost imitating reading and writing from and to a disk. The third program does a combination of both. Each program takes in as arguments the scheduling policy to be used and the number of simultaneous processes to be spawned per test. Fork() is used to create multiple child processes to run at the same time to give the Low, Medium, and High simultaneous process instances. The parent process waits for all child processes to complete and when that happens the test is completed.

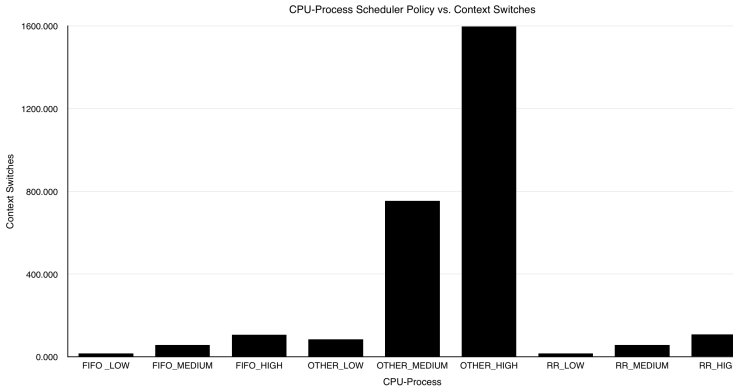
A bash script is used to automatically run all of the tests and output the values of three runs for each test to multiple files. The time benchmarks look at many different characteristics such as the number of context switches, time it took the processes to complete, time in kernel and user mode, and percentage of CPU the job got. The data that will be used in the analysis and results will pertain mostly to context switches and total time it took for the processes to complete as this data will help answer the questions found in section 4.2 of the write-up pdf.

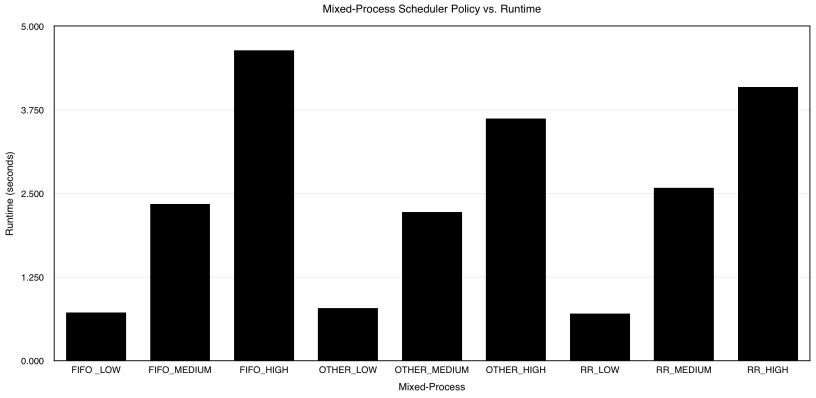
4 RESULTS

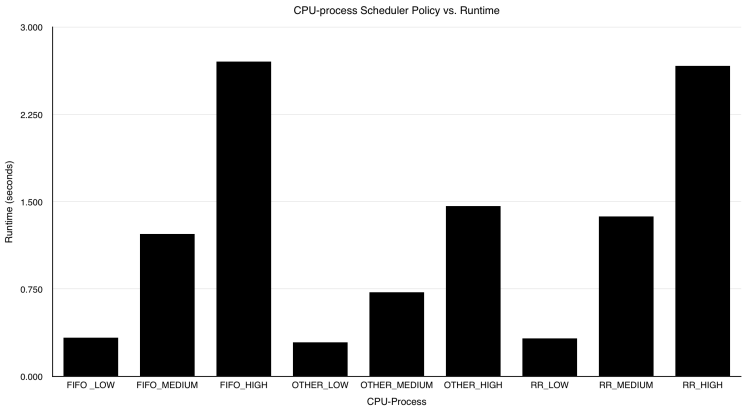
The below graphs are for each type of process showing how context switches and runtime are related to each policy and the number of simultaneous processes for each process.

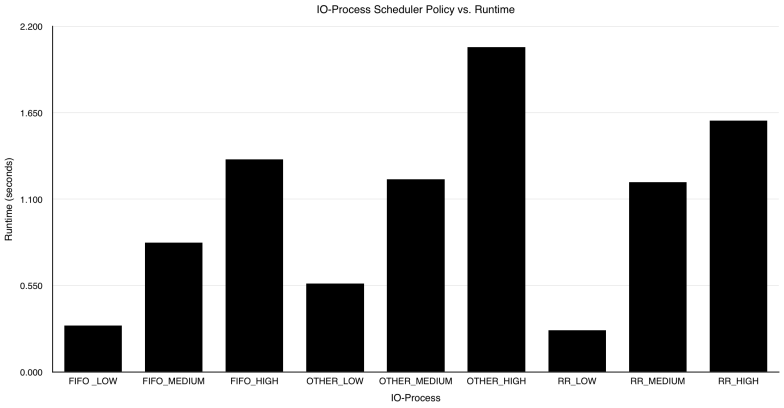












What can be seen in the graphs is that certain scheduling policies limit the number of context switches but don't necessarily lead to shorter overall runtime.

5 ANAYLYSIS

Based on the graphs and results found in appendix B, a few facts can be stated outright. For a CPU-process the best scheduling policy for overhead efficiency, the fewest amount of context switches, is *SCHED_FIFO* and the best scheduling policy for run-time is *SCHED_OTHER*. For an I/O-process the best scheduling policy for overhead efficiency is *SCHED_FIFO* and the best scheduling policy for run-time is *SCHED_FIFO*. For a mixed-process the best scheduling policy for overhead efficiency is *SCHED_FIFO* and the best scheduling policy for run-time is *SCHED_OTHER*.

As more and more simultaneous processes are run each of the scheduling policies goes up in terms of run-time and context switches. *SCHED_FIFO* is consistently lower than the other scheduling process when it comes to simultaneous processes and context switches. *SCHED_OTHER* did better than the other scheduling policies in the majority of the tests

when it came to run-time. For each test, *SCHED_RR* was always in-between *SCHED_OTHER* and *SCHED_FIFO* in terms of context switches and runtime.

SCHED_FIFO gets the lowest amount of context switches when it runs with CPU-bound processes. *SCHED_OTHER* has the fastest run-time when it runs with CPU-bound processes. If dealing with worst case run-times and context switches, *SCHED_RR* is the best policy because it is neither the worst nor the best for each policy but it gives consistent middle-of-the-road results.

SCHED_OTHER is at its worst in terms of context switches when dealing with CPU-bound or any process for that matter. *SCHED_FIFO* is the worst for run-time when using CPU-bound processes. Again *SCHED_RR* gives average results for all graphs concerning run-time and context switches.

6 CONCLUSIONS

Based on the data gathered from the experiment, it could be concluded that Linux defaults to the *SCHED_OTHER* scheduling policy because it typically has the shortest runtime of all the other scheduling policies. Additionally, the idea that *SCHED_OTHER* would have the most context switches out of any of the policies because it is a time sharing policy with time-slicing is confirmed by the data. In all cases, *SCHED_OTHER* had the most context switches out of any of other policies. It can be seen in the data that fewer context switches does not necessarily mean a lower run-time. In the case of a CPU-process, *SCHED_FIFO* had the fewest amount of context switches but had a higher run-time probably because processes were less likely to voluntarily give up the CPU making it harder for other processes to complete their tasks in a timely manner. A time-slicing policy like *SCHED_OTHER*, leads to a higher run-time for I/O bound processes but works best for CPU-bound process run-time. A first-in-first-out policy such as *SCHED_FIFO* works best for I/O-bound process run-time. It can be concluded that no scheduler policy works best for each process type. That is why it is necessary for an Operating System to have multiple scheduling policies to handle the the many different processes it may encounter in order to run most efficiently.

7 REFERENCES

[1] Krzyzanowski, Paul. *Operating Systems 07. Process Scheduling*. Rutgers University: Spring 2014. Accessed 26 March 2014. http://www.cs.rutgers.edu/~pxk/416/notes/content/07-scheduling-slides.pdf

8 APPENDIX A

CPU-PROCESS	SCHED_FIFO_LOW	SCHED_FIFO_MEDIUM	SCHED_FIFO_HIGH	SCHED_OTHER_LOW	SCHED_OTHER_MEDIUM	SCHED_OTHER_HIGH	SCHED_RR_LOW	SCHED_RR_MEDIUM	SCHED_RR_HIGH
Real time (%e)	0.333	1.223	2.703	0.293	0.723	1.463	0.327	1.373	2.667
User mode time (%U)	0.307	1.733	3.517	0.573	2.660	5.300	0.297	1.723	3.533
Kernel mode time (%S)	0.000	0.017	0.047	0.007	0.040	0.117	0.000	0.027	0.040
% CPU used (%P)	92.000	151.000	150.000	350.333	372.000	370.000	91.667	143.667	152.333
Involuntary context switches (%c)	2.000	2.000	1.333	63.667	654.333	1402.333	2.000	2.000	2.333
Voluntary context switches (%w)	14.000	55.333	106.000	21.000	100.000	197.667	14.000	55.000	106.000
IO-PROCESS	SCHED_FIFO_LOW	SCHED_FIFO_MEDIUM	SCHED_FIFO_HIGH	SCHED_OTHER_LOW	SCHED_OTHER_MEDIUM	SCHED_OTHER_HIGH	SCHED_RR_LOW	SCHED_RR_MEDIUM	SCHED_RR_HIGH
Real time (%e)	0.297	0.823	1.353	0.563	1.227	2.067	0.267	1.207	1.600
User mode time (%U)	0.033	0.037	0.097	0.003	0.027	0.043	0.000	0.033	0.107
Kernel mode time (%S)	0.053	0.203	0.333	0.333	0.927	1.637	0.090	0.367	0.503
% CPU used (%P)	23.000	29.000	32.000	60.667	77.333	81.000	35.333	34.000	37.667
Involuntary context switches (%c)	1.000	1.000	1.000	144.333	514.333	794.000	1.000	1.000	1.000
Voluntary context switches (%w)	2514.667	10914.333	25412.000	3682.333	18327.667	36927.333	2232.000	14574.000	31215.333
MIXED-PROCESS	SCHED_FIFO_LOW	SCHED_FIFO_MEDIUM	SCHED_FIFO_HIGH	SCHED_OTHER_LOW	SCHED_OTHER_MEDIUM	SCHED_OTHER_HIGH	SCHED_RR_LOW	SCHED_RR_MEDIUM	SCHED_RR_HIGH
Real time (%e)	0.723	2.343	4.640	0.787	2.220	3.617	0.707	2.583	4.090
User mode time (%U)	0.383	2.253	3.933	0.653	2.997	5.710	0.320	1.890	4.723
Kernel mode time (%S)	0.223	1.067	1.960	0.343	1.050	1.927	0.123	0.810	2.013
% CPU used (%P)	81.667	145.667	128.333	127.000	181.667	211.000	63.000	106.000	164.333
Involuntary context switches (%c)	1.000	1.000	1.000	499.333	2532.000	4138.667	1.000	1.000	1.333
Voluntary context switches (%w)	2606.667	13502.667	30360.667	3330.000	18112.333	34877.667	2699.333	15654.333	30803.333

9 APPENDIX B

1. **cpu-process.c** This file contains a simple program for statistically calculating pi using a specific scheduling policy and number or processes.
2. **io-process.c** This file contains a small I/O bound program to copy N bytes from an input file to an output file for a specific scheduling policy and number of processes.
3. **mixed-process.c** This file contains a simple program for calculating pi and reads and writes N bytes from an input file to an output file for a specific scheduling policy and number of processes.
4. **testscript** A simple bash script that runs and measures the performance of each test case three times.
5. **Makefile** A GNU Make makefile to build all of the code listed here.
6. **README** Describes how to execute the code listed here.
7. **tests_output/** The output from running ./tescript