Programming Assignment 3: Investigating the Linux Scheduler

Stephen Bennett
CSCI 3753 - Operating Systems
University of Colorado at Boulder
Spring 2013

Due Date: Sunday, March 31, 2013 11:55pm

Abstract

1 Introduction

2 Method

2.1 Benchmarks

Three benchmarks were written, each testing one of the three possible program types: CPU bound, I/O bound, and a mix of the two. In creating the CPU bound benchmark `pi.c`, some computationally intensive algorithm had to be chosen such that no time would be spent waiting on I/O operations or blocked on any other resource. As a result of these constraints, the statistical algorithm for calculating the value of pi based on the Monte Carlo method was chosen[1]. This algorithm is relatively slow and very CPU intensive. It creates an imaginary quarter circle of radius RAND_MAX, generates a random pair (x,y) of coordinates $\{x, y \in \mathbb{R} \mid 0 \leq x, y \leq RAND\_MAX\}$, then calculates whether the (x,y) coordinate is within the quarter circle. Additionally, there are two counters: one that keeps track of the total number of iterations and another that keeps track of the number of times the random (x,y) coordinate is found to be within the quarter circle. Finally, once all iterations are complete, all that must be done is to calculate the probability of being in the quarter circle by dividing the two counters (inside circle divided by number of iterations) and multiplying the result by 4 to create a full circle rather than a quarter circle.

The I/O bound benchmark `rw.c` was written in such a way as to "minimize the effects of filesystem buffering and maximize I/O delays"[2]. In order to do this, the low level-level `read()` and `write()` system calls were used in conjunction with files opened in O_SYNC mode. O_SYNC causes `write()` operations to block until the data is physically written to the disk rather than until the data is simply copied to a kernel buffer[3]. An input file and an output file are given to the program which then reads blocks of data from the input file and writes said data to the output file. This process occurs multiple times with minimal CPU involvement, thus creating the I/O bound benchmark. There is only a small amount of CPU use involved, primarily in setting up the input/output files and verifying the passed parameters.

The third and final benchmark program was the mixed benchmark `mix.c`. I wrote this benchmark as a combination of the ideas from the previous two benchmarks. The program performs the same computation as the CPU bound process statistically calculating the value of pi, but every so many iterations it writes the current values of the counters and the estimated value of pi to a file. This `write()` operation once again uses O_SYNC mode to maximize I/O delays.

For each benchmark, I wrote a separate program (e.g. `rw-sched.c` for I/O benchmark, etc.) that took care of setting the correct scheduling policy and `fork()`ing the desired number of child processes. Each time a process needs to read from or write to a file, it needs its own input or output file in order to prevent additional waiting due to mutual exclusions and not actual I/O as is desired; the housekeeping programs ensure that each process has its own unique input/output file.

2.2 Testing and Data

I wrote a Bash script to take care of automation for running the 27 different test cases. Each test case was run three times and the results were averaged in order to get more accurate data. While compiling the source code with the Makefile, a single 1 MB file called `rwinput` is created. It is created by reading 1024 blocks of size 1024 kB of random data from `/dev/urandom` and writing the data to `rwinput`. To ensure that each instance of the I/O bound benchmark program had its own input file, the Bash script then copies `rwinput` as many times as necessary and gives each copy a unique name. The `rw-sched.c` program then passes the proper unique input file name to each child process `fork()`ed. Through the use of the `time` command and nested `for` loops iterating over both the number of child processes to `fork()` and the scheduling algorithms, I gathered results for each of the test cases. From the `time` command I gathered the following aggregate information for each set of processes:

- Wall time (turnaround time) - the real time the process took to complete from when it entered the system to when it completed
- User time - the amount of CPU-seconds the process spent executing in user mode
- System time - the amount of CPU-seconds the process spent executing in kernel mode
- The percentage of the CPU that the process got
- The number of times the process was context switched
- The number of times the process was blocked on I/O

Since the `time` command was used on the scheduler process which spawns the appropriate number of child processes, all information collected from the `time` command is the sum of the information of all child processes. As a result, in order to get more useful information on a per process basis, the aggregate results will be divided by the number of child processes spawned.

2.3 Test System

All of the tests were run on a desktop running 64-bit Linux Mint 14 Nadia with version `3.5.0-26-generic` of the Linux kernel. This all runs on a quad-core Intel Core i5-2500K CPU running at 3.30GHz with 8GB of RAM. Each core is capable of executing a single hardware thread; thus four processes may be running concurrently in this system. The primary disk for the system is a 1TB 7200 RPM Western Digital with 32MB of cache which uses the SATA II interface. All of the programs were compiled using `GNU C Compiler version 4.7.2 (Ubuntu/Linaro 4.7.2-2ubuntu1)`. The time slice for the Round Robin scheduling algorithm for this setup is 0.1000006 seconds.

3 Results

Using the methodology described above, I gathered the following results. Figure 1 on page 5, Figure 2 on page 6, and Figure 3 on page 6 show the average wall time per child process averaged over three trial runs for the three different benchmarks. Each figure shows the results for a specific benchmark and compares the performance of each scheduling policy based on number of child processes.

The CPU bound benchmark results in Figure 1 show that while the number of processes remains relatively low the SCHED_OTHER scheduling policy out-paces the other policies tested; at high numbers of processes the advantage disappears for all intents and purposes. Both of the Real-time scheduling algorithms perform relatively equally regardless of number of processes. Additionally, as the number of processes increases, each process takes more time to complete.

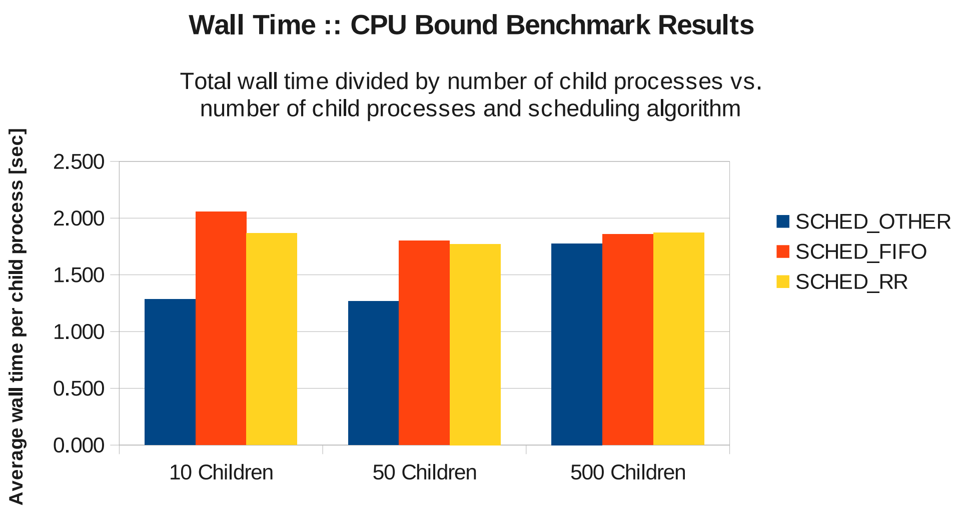

Figure 1

The I/O bound benchmark results in Figure 2 show that a higher priority scheduling algorithm improves wall time. As the number of processes increases, the performance of the different scheduling algorithms converges and the amount of time for each process to complete decreases.

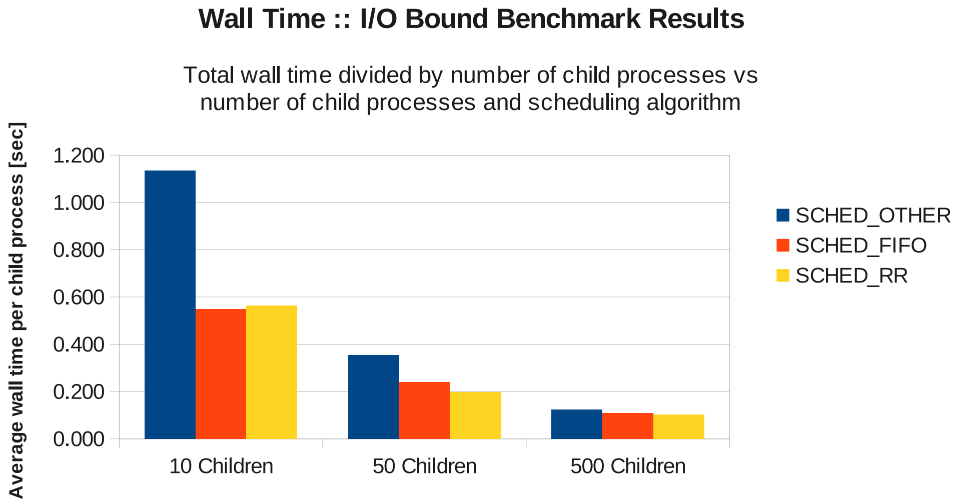

Figure 2

The mixed benchmark results in Figure 3 show that the scheduling policy does not have a significant impact on the amount of time each process takes to complete execution. As the number of processes increases, the amount of time each process takes to complete its execution decreases.

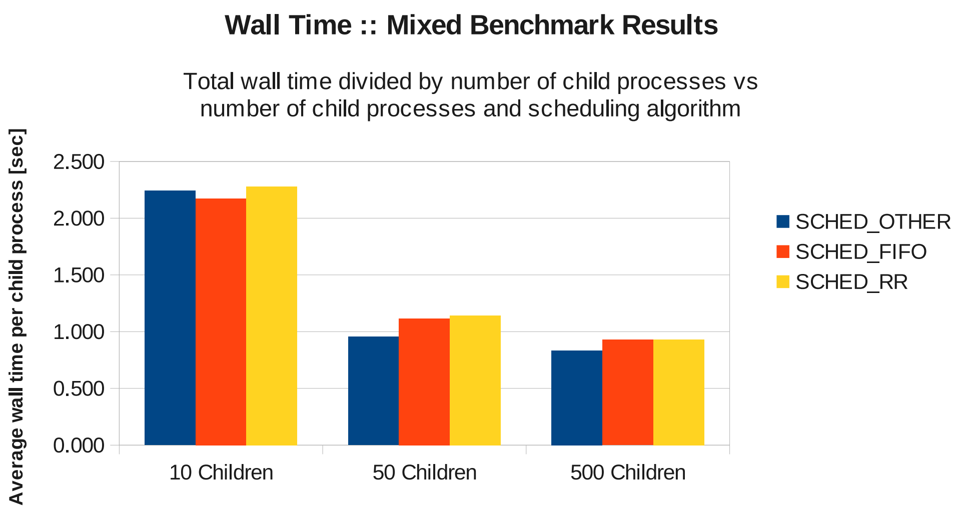

Figure 3

Figure 4 on page 7 and Figure 5 on page 7 show that the number of times a process is context switched under a Real-time scheduling policy remains constant as the number of processes increases. On the other hand, with the SCHED_OTHER scheduling policy as the number of child processes increases, so does the amount of context switches per process.

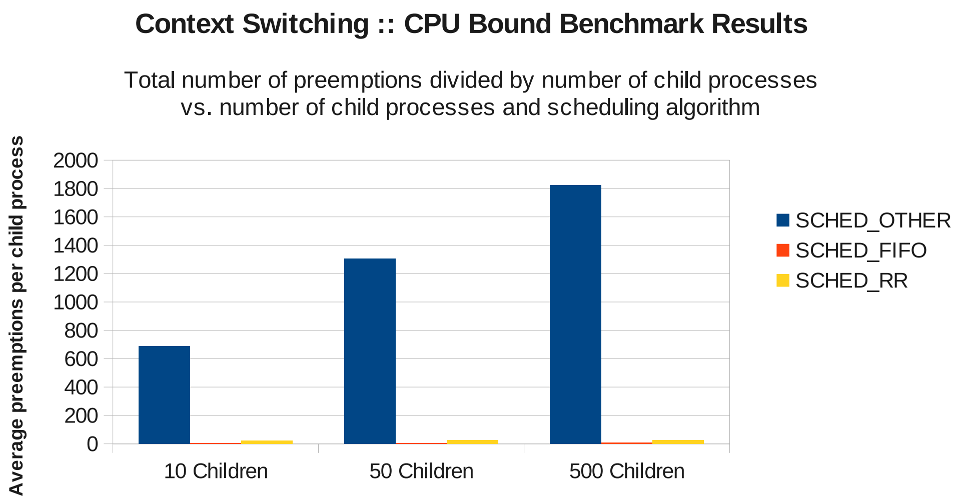

Figure 4

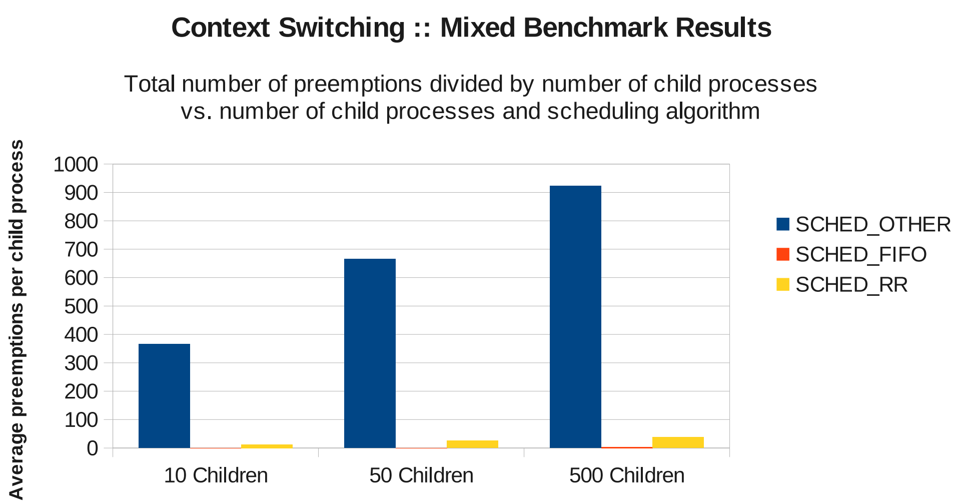

Figure 5

As can be seen in Figure 6 on page 8 showing the throughput of the CPU bound benchmark, the highest throughput is achieved using the SCHED_OTHER scheduling algorithm at relatively low quantities of child processes. As the number of processes increases, all of the scheduling algorithms converge on a throughput of roughly one process completed every two seconds.

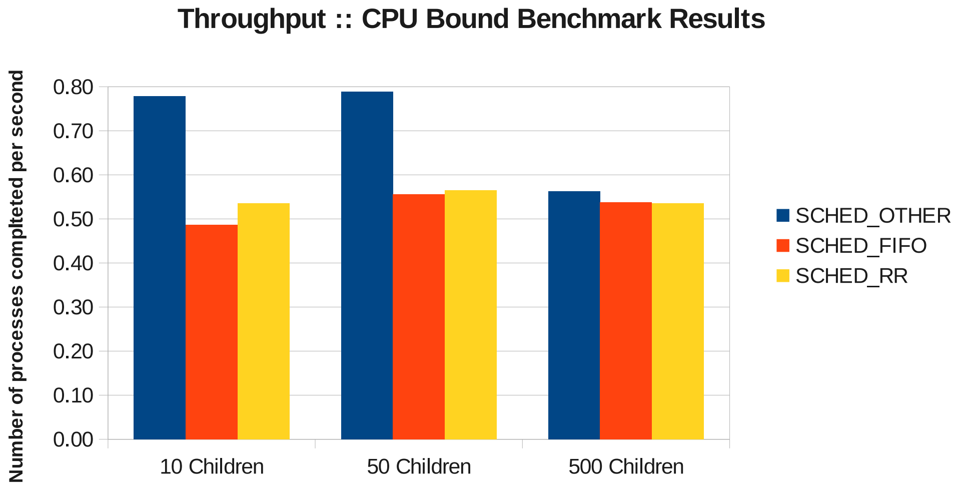

Figure 6

The throughput results in Figure 7 on page 8 show that increasing the number of concurrently executing I/O bound processes increases the overall throughput of the system. Additionally, the higher the priority of the scheduling algorithm, the higher the throughput.

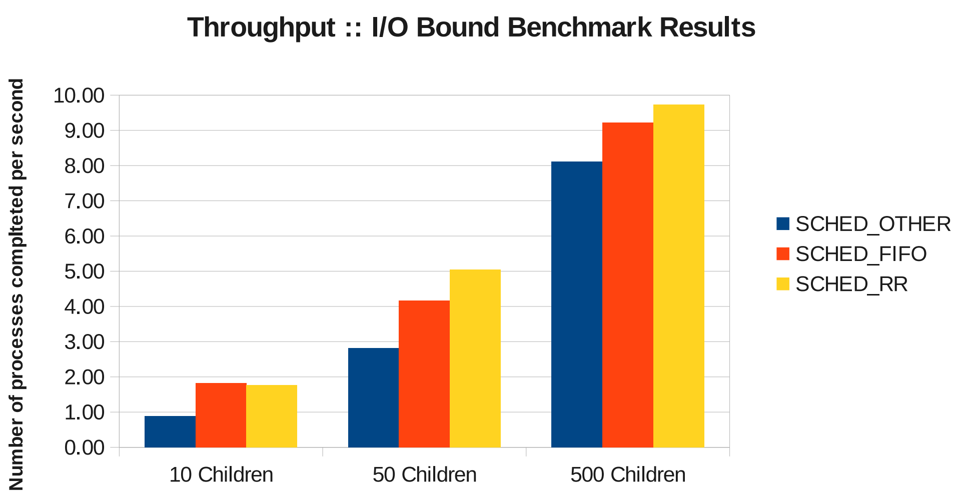

Figure 7

The throughput results in Figure 8 on page 9 show a similar overall positive correlation between number of processes and throughput to the I/O throughput results. However, at higher numbers of processes, the SCHED_OTHER scheduling algorithm seems to show a slight lead, similar to what is seen at lower system utilization in Figure 6 on page 8.

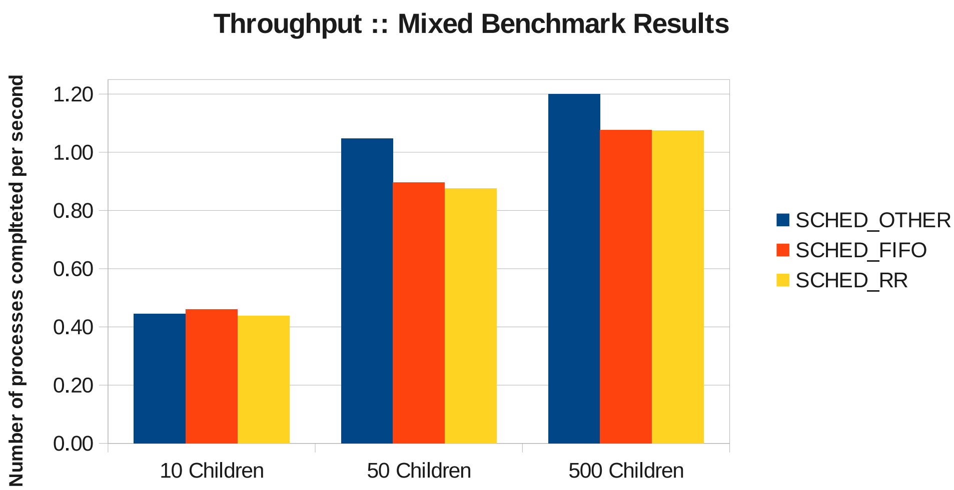

Figure 8

Figure 9 on page 9 and Figure 10 on page 10 show that as the system utilization increases, so does the CPU utilization; the mixed benchmark shows this more drastically. The CPU bound benchmark results show that the SCHED_OTHER scheduling algorithm makes the most efficiency use of the CPU, nearly maxing out all four cores regardless of system utilization.

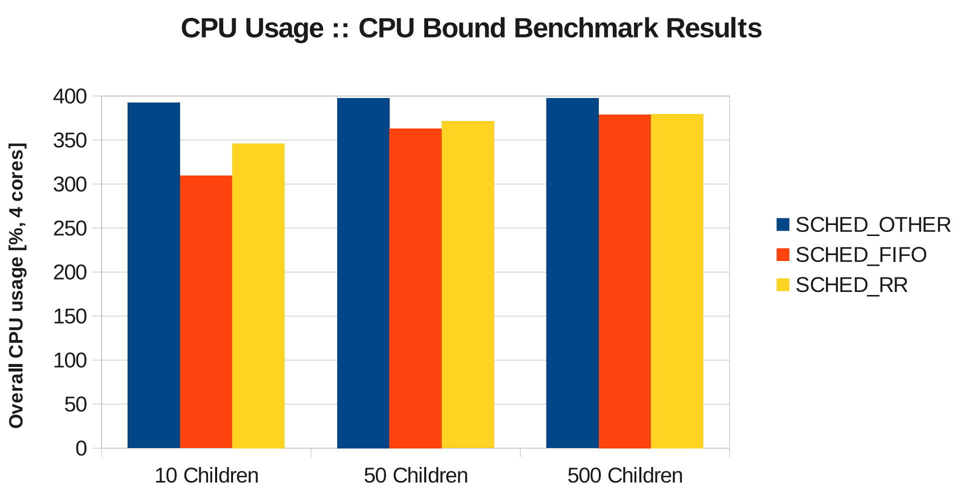

Figure 9

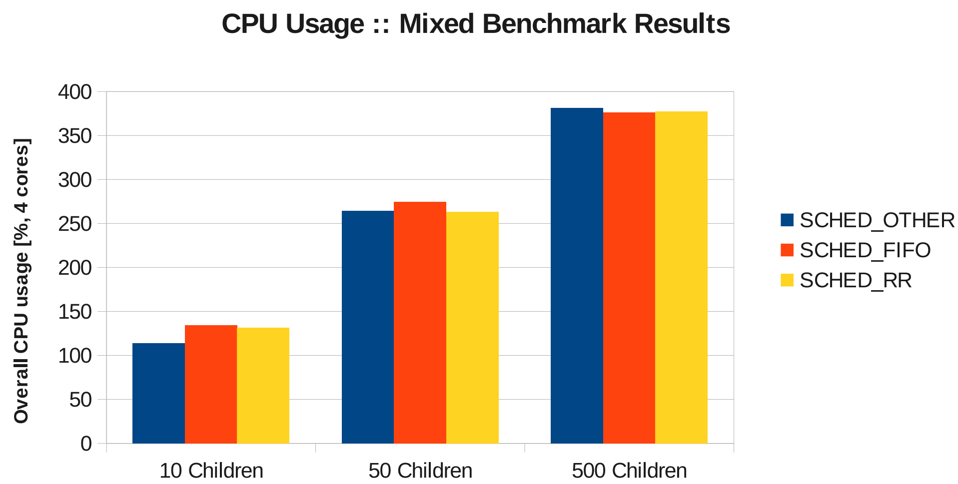

Figure 10

4 Analysis

The ideal scheduling policy in terms of minimizing turnaround time varies depending on the type of process. For CPU bound processes, it is clear the SCHED_OTHER algorithm reduced overall run-time, regardless of the system load and in spite of the greater amount of context switching that occurred with this scheduling policy. I believe this was a result of the scheduler's ability to maximize CPU usage at all levels of system utilization, unlike what was done with either SCHED_FIFO or SCHED_RR. There does appear to be a trend where increasing the number of processes reduces SCHED_OTHER's advantage. In fact, it is quite possible that if the system utilization was increased further, SCHED_OTHER might eventually lose out to the real-time policies.

For I/O bound processes, there is no best scheduler, but SCHED_OTHER is clearly outperformed by both of the real-time scheduling algorithms. As system utilization was increased though, the gap between the CFS and real-time algorithms was reduced. This can most likely be attributed to a more efficient use of the I/O subsystem. Since SCHED_RR and SCHED_FIFO processes have higher priorities than SCHED_OTHER processes, they are more likely to be given CPU time as soon as they need it which allows them to more quickly make their next I/O request. As more processes are added to the system, the I/O subsystem spends less time sitting idle. This idea can also be seen in Figure 7; the I/O subsystem is able to support a large amount of data transfer which is not efficiently put to use until enough I/O bound processes are present in the system. Using the information from the throughput figure does seem to indicate that the SCHED_RR algorithm is actually the best for completely I/O bound processes.

As for mixed processes, there is simply no best scheduling policy directly in terms of wall time. The combination of computation and I/O counteract each other and as a result each scheduling algorithm performs essentially identically, although a slight edge may be given to SCHED_OTHER. This advantage can also be seen in the throughput results at higher levels of system utilization. It is entirely likely that when I wrote the mixed benchmark, I unknowingly gave the program more of a CPU bound tendency. As described above, this would cause the processes to perform best under the SCHED_OTHER scheduling policy. I would think that mixed processes would lend themselves to SCHED_FIFO as the best scheduling policy since that is somewhere between SCHED_OTHER and SCHED_RR in terms of priority, but that does not seem to be entirely the case.

If minimizing time wasted on context switching is desired, SCHED_FIFO is the ideal scheduling algorithm. Since a SCHED_FIFO process can only be preempted by a higher priority process and all processes were given the same priority, the SCHED_FIFO were almost never preempted (any preemptions were outside the control of my test system setup).

5 Conclusion

References

[1] Sayler, Andy. *pi.c*. 03/09/2012. Forked 03/17/2013. https://github.com/asayler/CU-CS3753-PA3/blob/master/pi.c.

[2] Sayler, Andy. *Programming Assignment 3: Investigating the Linux Scheduler*. 03/06/2012 https://github.com/cgartrel/CU-CS3753-PA3/blob/master/handout/pa3.pdf.

[3] Linux man-pages project. *open(2)*. 2013/02/18. Access 03/23/2013. http://man7.org/linux/man-pages/man2/open.2.html.

[4] Jones, M. Tim. *Inside the Linux 2.6 Completely Fair Scheduler*. IBM developerWorks: 2009. Accessed 06/01/12. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[5] Kernighan, Brian and Dennis, Ritchie. *The C Programming Language*. Second Edition: 2009. Prentice Hall: New Jersey.

[6] Kolivas, Con. *BFS - The Brain Fuck Scheduler*. 2009. Accessed 06/01/12. http://ck.kolivas.org/patches/bfs/sched-BFS.txt.

[7] Kolivas, Con. *FAQS about BFS*. v0.330: 2009. Accessed 06/01/12. http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

[8] Kumar, Avinesh. *Multiprocessing with the Completely Fair Scheduler*. IBM developerWorks: 2008. Accessed 06/01/12. http://www.ibm.com/developerworks/linux/library/l-cfs/.

[9] Le, Thang Minh. *A Study on Linux Kernel Scheduler: version 2.6.32*. 2009. Accessed 06/01/12. http://www.scribd.com/thangmle/d/24111564-Project-Linux-Scheduler-2-6-32.

[10] Molnar, Ingo. *This is the CFS scheduler*. 2007. Accessed 06/01/12. http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.

A Appendix - Raw Data

Figure 11

Process Type	Scheduler	Children	wall [sec]	user [CPU-sec]	system [CPU-sec]	CPU [%]	preempted [count]	blocked [count]
CPU Bound	SCHED_OTHER	10	12.84	50.40	0.03	392	6871	22
		50	63.39	252.20	0.10	398	65376	104
		500	888.47	3532.61	1.15	397	911979	1014
	SCHED_FIFO	10	20.55	63.56	0.02	309	50	15
		50	89.99	326.81	0.08	363	326	55
		500	928.95	3523.70	0.98	379	3685	505
	SCHED_RR	10	18.68	64.82	0.01	346	215	18
		50	88.57	329.68	0.07	371	1240	76
		500	934.63	3552.97	0.76	380	13501	791
I/O Bound	SCHED_OTHER	10	11.36	0.02	0.03	0	19	2687
		50	17.72	0.02	0.06	0	114	16957
		500	61.60	0.04	0.07	0	544	163474
	SCHED_FIFO	10	5.50	0.00	0.00	0	1	2012
		50	12.01	0.02	0.02	0	1	12494
		500	54.24	0.02	0.06	0	1	142977
	SCHED_RR	10	5.64	0.00	0.00	0	1	2012
		50	9.89	0.00	0.00	0	1	10052
		500	51.40	0.02	0.04	0	1	133495
Mixed	SCHED_OTHER	10	22.42	25.30	0.06	114	3657	4969
		50	47.76	126.03	0.22	264	33244	25253
		500	416.60	1588.08	2.95	381	461419	236701
	SCHED_FIFO	10	21.73	29.24	0.05	134	1	4262
		50	55.80	152.80	0.24	274	3	21425
		500	464.28	1747.49	2.67	376	1732	186482
	SCHED_RR	10	22.79	30.00	0.05	131	123	4343
		50	57.12	150.48	0.19	263	1258	21701
		500	464.80	1753.86	2.27	377	18655	172993

Figure 12

Process Type	Scheduler	wall/child [sec]	user/child [CPU-sec]	system/child [CPU-sec]	preempted/child [count]	blocked/child [count]
CPU Bound	SCHED_OTHER	1.284	5.040	0.00333	687	2
		1.268	5.044	0.00200	1308	2
		1.777	7.065	0.00229	1824	2
	SCHED_FIFO	2.055	6.356	0.00167	5	2
		1.800	6.536	0.00167	7	1
		1.858	7.047	0.00195	7	1
	SCHED_RR	1.868	6.482	0.00133	21	2
		1.771	6.594	0.00140	25	2
		1.869	7.106	0.00153	27	2
I/O Bound	SCHED_OTHER	1.136	0.002	0.00267	2	269
		0.354	0.000	0.00113	2	339
		0.123	0.000	0.00014	1	327
	SCHED_FIFO	0.550	0.000	0.00000	0	201
		0.240	0.000	0.00047	0	250
		0.108	0.000	0.00013	0	286
	SCHED_RR	0.564	0.000	0.00000	0	201
		0.198	0.000	0.00000	0	201
		0.103	0.000	0.00009	0	267
Mixed	SCHED_OTHER	2.242	2.530	0.00567	366	497
		0.955	2.521	0.00433	665	505
		0.833	3.176	0.00591	923	473
	SCHED_FIFO	2.173	2.924	0.00533	0	426
		1.116	3.056	0.00487	0	428
		0.929	3.495	0.00534	3	373
	SCHED_RR	2.279	3.000	0.00500	12	434
		1.142	3.010	0.00373	25	434
		0.930	3.508	0.00453	37	346