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

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, 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 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. 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 another program that took care of setting the correct scheduling algorithm and `fork()`ing the desired number of 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.

Finally, I wrote a Bash script to take care of automation for running the 27 different test cases. Through the use of nested `for` loops and the `time` command, I gathered results for each of the test cases. From the `time` command I gathered the following information:

- 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

100000000

3 Results

4 Analysis

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

B Appendix - All Code