CS 416 - Operating Systems Design

February 10, 2014
Due by February, 18 - Midnight

1 Introduction

This homework (HW01) is about understanding the performance of certain hardware and OS operations. It consists of three parts. In the first part, you will have to determine the parameters of the second-level memory cache through time measurements. In the second part, we will gauge the cost of a fork(..) system call. Finally, in the third part, we will evaluate the cost of data transfers between two processes using pipes, one of the methods that the OS offers for inter-process communication (IPC). From the last two parts, you will understand the impact of memory copy operations on the performance of the fork and pipe system calls, which in turn will allow you to guess their implementation in the OS kernel.

2 Part 1
Determining L2 Cache Parameters

[33 points] It was reviewed on class how the cache works to mitigate the time costly of main memory data access. The cache can be characterized by three parameters:

- Size of the Cache Block/Line
- Cache Size
- Associativity

As the cost to access data from the Main Memory is several orders of magnitude higher than from the cache, a good program design is one that most of the accessed data is available in the cache, avoiding several expensive accesses to the Main Memory. Cache's design exploits the *Principle of Locality*, which states that data accessed now has a higher chance to be accessed again in the near future (*temporal locality*), and that data blocks near the one accessed now have, also, higher chances to be accessed in the near future (*spacial locality*).

Different processors, even from the same manufacturer, have different Cache designs and Memory Hierarchies, in terms of Cache levels (L1, L2, L3, shared L3, etc). Refer to the Processor's Manual to know the Cache Parameters and Memory Hierarchy of your CPU.

For this part of the homework, you are asked to design a program that will allow you to extract the L2 Cache parameters. You should be able to determine:

(a) Cache Block/Line Size

(b) Cache Size

(c) Cache Miss Penalty - the time spent when the data is not on the cache, it might be on the Main Memory

The following list outlines some ideas that will help you to achieve this part:

- Since the cache is transparent to the application programmer, there is no direct method to measure the cache size. Therefore, the cache should be measured indirectly.
- The way we achieve this is by repeatedly running a program that accesses a large array with different patterns to lead to predictable capacity misses on the cache. By measuring the running time of the program we can infer the cache size.
- Assume that the cache line and size are a power of two.
- Make sure that the memory used in the program is already mapped into the physical memory before measuring the running time. This can be achieved by traversing the array once before starting the measurement.
- The array you traverse in your program is allocated at virtual addresses, while the L2 cache whose parameters you are measuring maps physical addresses. Your approach can ignore this since it will not significantly affect the result.

For this part, submit the program source codes with a makefile. You should also include a short report about your methodology and results (namely, the size of the cache line and the size of the cache, and how you determined them). This report should also contain a short description about each C file and the main function's input argument(s). Sometimes it is easier to analyze the data collected by your memory accesses in a chart, submit it if you have created one.

3 Part 2
Cost of fork()

[34 points] System Calls are the interface that allows user-level applications require Operating System services. It is a synchronous operation which gives an application a transparent, safe and encapsulated way to execute kernel-level operations. They have successfully been used by different Operating Systems. More about System Calls can be found on the text book, Chapter 2.

For this part, you will have to implement an user-level application that should measure the cost of the fork(..) system call. To do that, you can place a pair of gettimeofday(..), just before and after the fork(..). As we know, there is no deterministic way to know which one of the two processes (after the fork(..)) will execute first. You have to deal with that and find out a way to force a certain deterministic schedule of the two processes in order to perform meaningful measurements. Also, to diminish interference, you should compute the average of 10,000 fork(..)s calls. You can try to design different programs for these measurements to cross-validate the result.

Also, to understand the impact of the expected memory copy operation, you must repeat the measurements for different program sizes. You achieve this by allocating an array of the given size before fork()ing.

The block of memory corresponding to this array should range from 1MB to 1GB. As we know, this block will be copied to the child process after the fork(..), since both of them will have the same memory content after the system call. You should evaluate the average cost of the fork(..) for different block sizes, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB and 1GB. The size of the block has to be the input of your program, such as: ./part2 size-in-MB. For instance, if the user wants the average cost time of fork(..) system call with a block of 128MB, he would call your program: ./part2 128, the output will be the average time in usecs.

For this part, submit the program source code with a makefile. You should also include a short report about your methodology and the interpretation of the results. You have to submit a chart: time x data blocks.

4 Part 3 The cost of communication through pipes

[33 points] Pipes provide a unidirectional inter-process communication (IPC) channel. A pipe has a read and a write end. Data written to the write end can be read from the read end of the pipe. They are also blocking, i.e. if one tries to read from a empty pipe, it will block until there is data on the pipe. More about pipe, refer to pipe's Linux man page.

In this part, you will have to measure the cost of communicating a certain message size between two processes through a pipe. To achieve this, you should write a program that creates two pipes. This program will fork(..). One pipe will be used to send data from the parent to the child process, while the other one will be used to send data from the child to the parent. In this way, you can create a simple synchronization between the two processes, which will allow you to deterministically measure the cost of communication through pipe.

You will use the pipes to send/receive different data blocks between the processes. Given that you have two processes after the fork(..), P1 and P2, P1 will allocate a block of data and repeatedly send it to P2. Similarly, P2 will also allocate a data block of the same size to read the block sent by P1, and will send it back to P1. With a properly designed scheme, P1 should be able to measure the cost of sending and receiving data blocks of a given size through a pipe. The data blocks will range from 1MB-512MB, in the same strides as for Part2.

Also, you should get the average of 1,000 executions of each data block size, to diminish interference in your measures.

For this part, sumit the program source code with a makefile. You should also include a short report about your methodology and interpretation of the results. You have to submit a chart: time x data blocks.

To fully understand the results for part 2 and part 3 of this HW you could write a small program to repeatedly copy blocks of memory (defined as arrays) of the same sizes as the corresponding ones in the fork/pipe programs. By comparing the results, you could guess what copies the kernel does in these two cases.

5 Notes

- The code has to be done in **C** language, without using inline **asm**.
- The code of each part is small, it should have less than 20-50 lines. If you have more than that, it might have something wrong.
- Do not copy the solution from other students. Use Piazza to discuss ideas of how to implement it. Do not post your solution there. Use Sakai to submit it.