Project 1 Report
Operating Systems
Oleg Tielushko, Andrew Canaan
February 2nd, 2020

Project 1 Report

Introduction

For Project 1 we were required to write two C programs: system_call.c and context_switch.c. The purpose of these is as follows:

a. system_call to measure the time-cost of a system call.
b. context _switch to measure the time-cost of a context switch using pipes to communicate.

The file 'system_call.c' measures the average time in nanoseconds of the system call 'getpid()' over one million iteration cases. The file 'context_switch.c' measures the average time in milliseconds of the context switch between two processes over one million iteration cases. This is accomplished through piping which would be explained more in the approach section of our report.

Problem Importance

The problem of measuring system calls is important because hardware and software engineers work to reduce the time-cost of a system call in their architectures. By reducing the time-cost of a system call, the system in which it is applied will become more efficient overall.

Similarly, reducing the time-cost of a context switch is vital to improving performance of a given architecture. However, context switches are also important because they enable processors to 'multi-task', greatly improving the performance of the architecture. Without context switching modern day processors would be *significantly* less efficient. Therefore, measuring and reducing the time-cost of such processes is vital to the development of classical computing.

Approach

System Call:

Project 1 Report
Operating Systems
Oleg Tielushko, Andrew Canaan
February 2nd, 2020

In order to measure the time-cost of a system call a simple approach was taken. Firstly, we decided to use 'getpid()' as our system call to measure. To measure the time cost of a single call we used the struct timeval start, end; and gettimeofday() function included in <sys/time.h> this allowed us to record the begin and the end of our processes. In order to create a significant amount of iterations, we created a for loop that would run for a million times and initiate the same getpid() call. After the loop execution, we called gettime of day to record the ending of the timer. After, we performed calculations to find out the average time in nanoseconds that it would take for our processor to run the system call and output the calculation on the screen. The output went as follows:

```
-bash-4.2$ gcc -std=c99 system_call.c
-bash-4.2$ ./a.out
Average time for system call in nanoseconds over million iterations: 2.679565907
-bash-4.2$
```

In the picture above, we see the bare output of the calculation of the average time for system_call.c which happens to be around 2.68 nanoseconds per call.

Context Switch:

The context switch was performed by imitating the lmbench benchmark test, in particular lat_ctx() function of the benchmark. The process involved initialising two pipes and connecting the forked parent and child process with the pipes. The first process would then issue a write to the first pipe, and would wait to read from the second pipe. Since the first process is waiting for the read from the second pipe, the OS puts it in the blocked state, and switches to the second process. That process in itself, reads the content in the first pipe that was passed by the first process, and writes it into the second pipe. Once this write is complete, the OS performs yet another context switch, finally executing the read from the first pipe. This allows us to make OS perform a context switch.

To begin with, first thing in our program, we used the GNU Source header functions to set the CPU to perform using only a single core. That was mostly achieved by the <sched.h> function sched_setaffinity which accepted our current process id, the set of cpu cores and which core we would like to use.

Then, we went ahead and instantiated the needed variables: struct timeval, 3 pipes that are used in communicating between the processes, a pid_t for forking and a communication int byte that

was to be passed in between processes. The 3rd pipe was necessary in order to allow for the struct timeval end to get transferred between the two processes for the proper time calculation.

After some pipe-checking, we entered the if else if statements to differentiate the execution of the parent and child processes. In the parent process, we started the timer by calling gettimeofday() and closed the opposite ends of pipe from the ones that we are going to use. After, we run a for loop that allows us to switch between processes by technique described way above. Context switch will make us jump to the child process, in which we initiate a read, and a write back. This is performed for one million iterations. After the loops, the program flow is in child, so then we get the timer for the end of operations and write it to the third pipe to get accessed by the parent process later. Later the child exists, and the control jumps to the parent. We close down all the pipes and get the value of struct timeval end by reading from the third pipe called timePipe. Finally, we compute the values and output the results in milliseconds. The output is presented below:

```
-bash-4.2$ gcc -std=c99 context_switch.c
-bash-4.2$ ./a.out
Average time for context switch in milliseconds over million iterations: 0.002714 ms
-bash-4.2$
```

In the picture above, we see that the output of the calculated average time it takes to perform a context switch over a million iterations. Remember, the result has to not be only divided by million, but also by 2, since with each iteration, context switch occurs twice.

Results and Analysis

Most important factor in determining the correct calculations was to make sure that the tests ran over a significant amount of iterations. This would allow for unskewed data and better averages.

It was shown that a context switch is far more expensive that a system call and for the most part depends on how still limited our single core CPUs are in executing multiple instructions and communicating the results back to the user. It is a segway into areas of improvement for computer architecture.

Project 1 Report
Operating Systems
Oleg Tielushko, Andrew Canaan
February 2nd, 2020

Challenges and Limitations

In regards to system_call.c, one of the biggest challenges was getting the function ‘gettimeofday’ to work properly. It would consistently return inaccurate results over a small number of trials. However, by increasing the number of trials to a more sufficiently large number we were able to raise the measurement accuracy overall.

For context_switch.c, the first challenge was to make sure the system and processes run on a single CPU. As having multiple cores would allow for much shorter context switch times and would not benefit our project. The second challenge was to have a way to communicate the time readings between processes. At first we had issues of having negative or very large times for the gettimeofday() function. The solution to inaccurate/false time readings was to introduce a 3rd pipe. This pipe was written the struct timeval end to by the child pipe, and then read from by the parent pipe to get the ending time value. Another technical challenge was properly communicating between pipes. After researching into file descriptors, and GNU piping manual pages, the syntax became more intuitive and implementation was possible.

For both programs one of the biggest limitations is the innate inaccuracy of all time measuring functions in software. In order to gain the most accurate insight, one would need to use a perfectly synced time clock. However since access to such software is unavailable we relied on the average over a very large number of trials. If the hardware available were more suited to computing over large iterations we could have increased the number of trials, furthering the accuracy of our measurements.

Final Thoughts

This project was vital in contributing to our understanding of how OS, hardware, and software interact in a dynamic way. In order to actually complete the project one must have a firm understanding of not only how the CPU operates on one process, but how multiple processes are handled by the Operating System. Measuring a system call was intriguing, but measuring a context switch is a lot more interesting endeavor to achieve as a student. These switches are occuring incredibly quickly under the hood, so setting out to identify and measure their real world effect was quite interesting.