An Analysis between System Calls and Context Switching

Steven Faulkner
COP 4600
Project 1
sfaulkenr1@mail.usf.edu

Abstract*—To compare the overhead of system calls and context switches within a Linux operating system using programs written in C. With system calls and context switching being an integral piece of operating system core functions, it's important to understand the consequences of utilizing the right instructions to optimize operating system kernels. This ideology forms the framework of which our tests were made. After an introduction to the reasonings behind the creation of both the system call and context switching program, we discuss the testing methodologies that were used, finally, we analyze our results, discussing what anomalies, limitations and challenges that that were presented during our findings.

I. Objective

The goal of this project is to gain a deeper understanding of how system calls, and context switches work and the cost of their implementation. This was achieved by utilizing a program that would invoke a low level system call, and a separate program that would force a context switch between two forked processes.

II. Design and Implementation

Our design for this problem involves two separate programs. The first program invokes a system call using the C language function `read()`, which we measured using `clock_gettime()`. There exists a variety of system clocks and clock profiles to choose from for a variety of testing purposes. While the project parameters mentioned using **`rdtsc`** instructions, this was seemed too low level for our purposes, `CLOCK_MONOTONIC_RAW` was chosen as the preferred clock profile as it was precise to the nanosecond while also immune to NTP adjustments that `gettimeofday()` suffers from. The program iteratively calls the `read()` function while recording the monotonic time before and after the calls return. The `timespec` struct that stores the monotonic time contains two elements; a second element and a nanosecond element. In order to calculate the system call time the structs seconds element is converted to nanoseconds then the difference is taken to get the approximate value of the `read()` function. Finally, this the sum of this difference over all iterations is divided by the total number of iterations to give an approximation for an individual system call.

The second program builds upon the of the first program with some key differences. The C function `pipe()` allows for two processes to communicate and pass data between them. This function can also be manipulated to force a context switch away from a waiting function. This function in combination with the `fork()` function allows for the creation of a second process to which we can potentially for a context switch to. This is accomplished by calling the `read()` function on one pipe under the parent process while writing to that same pipe in the child process. Utilization of `sched_setaffitnity()` in each process also ensured that both parent and child process would execute on the same processor. The parent process contains an iterative loop of the same magnitude as the first program, while also calculating the time difference of the context switch to the child process and back to the parent process.

III. Testing and Results

Testing methodologies were very straightforward for both programs. We performed our tests on a remote server running Red Hat Linux with an Intel Xeon E5-2603 CPU running at 1.60 GHz with 16GB of DDR3 1333-MHz system memory as well as an NVidia GTX 960 GPU with 2 GB of on board memory(global) and 1048 KB of local cache running at 1.33 Ghz. As mentioned before both programs iteratively make a system call or force a context switch between two processes, outputting the average time of each action took in nanoseconds. An anomaly was discovered with the execution of the context switch where the recorded

time would return microseconds versus nanoseconds, This was concluded to be improper placement of the `clock_gettime()` functions, after the functions were moved outside of the iterative loop the time returned was in nanoseconds. The outputs can be in figure 1.

Figure 1.

Average Time for Read System Call

```
taking the average of 10000 system calls
2761 nanosecond average for system calls
```

Average Time for Context Switch

```
calculating 10000 context switches
14960 nanosecond average for context switches
```

The system call program performs a 0 byte reads from a file filled with random characters to not have to account for time to copy bytes from file to a character array. The average time for a system call was roughly 2684 nanoseconds with a maximum of 4003 ns and minimum of 2053 ns.

Accounting for the n number of reads for n number of iterations in the context switching problem brings the average down to roughly 8386 nanoseconds. Which still is not an accurate measurement of a pure context switch. While steps were taken to force a context switch between the parent process to the child process and back, it is impractical to assume that the operating system would context switch a different background process in-between the child and parents pipeline process. This becomes more apparent when looking at the average minima and maxima for the program: 4231 ns and 42930 ns respectively. This maxima value shows a high probability of separate processes executing in-between the context switch two processes.

Another limitation is the difficulty to measure just the time taken to perform a context switch, without some way to cause a process to wait, such as for I/O or even a simple wait function, a context switch is unlikely to happen. Including some necessary function is needed but also increases overall overhead of the entire process.

IV. Conclusion

Throughout the course of this paper we explored the overhead cost of system calls by measuring the `read()` function and forcing context switching using `pipe()`. After thorough testing, the high overhead cost of context switching over system calls becomes apparent, and this should be considered for if low latency operations are vital for whatever program might call for them. While both system calls and context switches serve their purpose, be it allowing user level access to operating systems or keeping cpu utilization as high as possible, they both suffer their drawback in overhead as seen by these test results.