System Calls and Signals:
Understanding the cost of OS Invocations

Introduction

The operating system is responsible for managing computer resources and ensures user applications use resources in a fair and secure manner. This means when applications want to use resources, they have to go through the operating system, more specifically the **kernel**. However when applications want to use resources such the CPU, memory, or I/O devices, the kernel must be scheduled to run and interrupt normal user program execution in order to carry out the request or handle other various issues that come up when utilizing such resources.

The goal of this assignment is to get an idea of potential overhead of invoking the kernel by using system calls and signaling and measuring their average cost of invocation. Please read the entire assignment description before starting and make sure you follow the requirements.

1 System Calls

User applications use system calls to request some service or resource from the operating system. When system calls are used, the kernel is invoked via a special trap, or software interrupt. Some common system calls include *read()* and *write()* which are used when reading a writing a file. The kernel needs to be involved as is has to ensure that the user application has the proper permissions needed to read and write that file. The system call *getpid()* is another simple system call that requests from the the process id of the calling process. This sort of functionality is handled by the kernel as the kernel is responsible for managing processes and assigning their IDs.

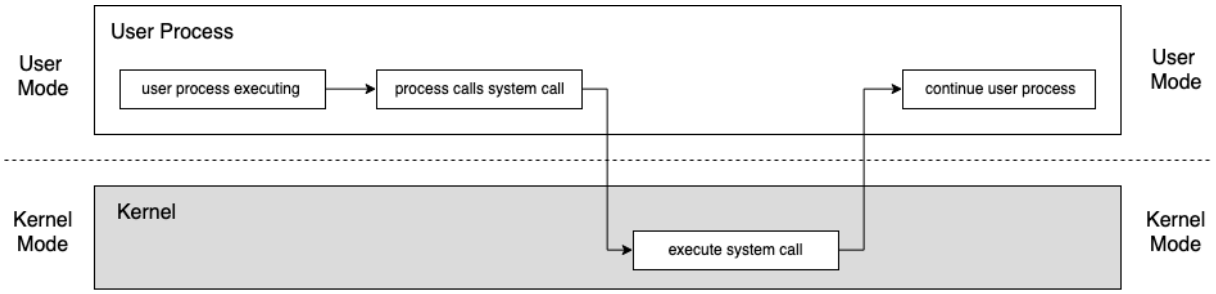

Figure 1: Execution flow when a system call is used

Given that system calls require the kernel to carry out the request before the user application can continue executing, it can incur significant overheads if we use system calls often. In this part of the assignment we will understand how system calls are used as well as understand how much time a system call can take.

Implementation

In order to get an idea of how long a system call can take, we want to time how long it takes to perform a system call. For this assignment we'll try and time the simple and fast system call *getpid()*. In order to get

an accurate reading, you should run the system call many times (e.g. 100,000 times) and then calculate the average time it took for each system call. In order to time the program execution you should use the function *gettimeofday()* to get the time before you start executing the system calls and the time after you stop executing the system calls so you can calculate the total time it took to execute all the system calls.

Supplied should be a *Makefile* and bare C program template called *TimeSysCall.c*, Your job to is to complete the implementation in *TimeSysCall.c*, running *getpid()* 100,000 times and timing it in order to calculate the following statistics: (1) how many times the system call was used, (2) the total time it took to execute all the system calls, and (3) the average time it took per system call. **The output of your program should be formatted exactly the same as in the example run** shown later on in this section **and show the time as microseconds**.

Note: Use floating point arithmetic during calculations. The *getpid()* system call will execute very fast (in order of fractions of a microsecond) since it's simply fetching the process id number from the kernel. When calculating the average time per system call, ensure to use floating point arithmetic (double or float) in order to correctly calculate the average time per system call.

Example Run

Command Line

```
$ make clean                              # Remove any old executable made
rm -f TimeSysCall TimeFork TimeSignal
$ make TimeSysCall                        # Compile using make
gcc -Wall TimeSysCall.c -o TimeSysCall
$ ./TimeSysCall                           # Run the program
Syscalls Performed: XXXX
Total Elapsed Time: XXXX microseconds
Average Time Per Syscall: XXXXX microseconds
```

Helpful References

The following are helpful references to help get you started

- **What is a System Call?**
  Link: https://en.wikipedia.org/wiki/System_call
- **The getpid() man page**
  Link: http://man7.org/linux/man-pages/man2/getpid.2.html
- **How to use getpid()**
  Link: https://www.geeksforgeeks.org/getppid-getpid-linux
- **How to use time functions using gettimeofday() (*Look at method #3*)**
  Link: https://www.techiedelight.com/find-execution-time-c-program/

2 System Calls Continued

The operating system has many system calls, each used to help user applications request services or resources from the operating system. Not all system calls would be as simple as *getpid()*. In order to get an understanding of the overhead of more complex system calls, we want to time a more involved system call such as *fork()*. The *fork()* system call is responsible for duplicating the calling process and the kernel needs to handle this as the operating system is responsible for keeping track of every process running on the system.

Implementation

In order to get a sense of the time it takes to fork a process, we want capture the time it takes to fork a process multiple times and calculate the average time it takes per fork. In order to fork the process, the program should call fork(). The original process (parent process) should then call *wait()* or *waitpid()* to wait for the the forked process (child process) to successfully be created, run, and terminate. Instead of running code, the forked process (child process) should immediately terminate by either calling exit() or returning from the main function. When the child process terminates, the parent process should continue to run and then fork the process again.

In order to get an accurate reading, you should *fork()* and *wait()/waitpid()* many times (e.g. 5,000 times) and then calculate the average time it took for each fork. Like the previous section, you should use the function *gettimeofday()* to get the time before you start forking the process and the time after you stop forking the process so you can calculate the total time it took to fork the process a certain amount of times.

Supplied should be a *Makefile* and bare C program template called *TimeFork.c*, Your job to is to complete the implementation in *TimeFork.c*, forking the process and waiting for the child process to terminate 5,000 times and timing it in order to calculate the following statistics: (1) how many times the process was forked, (2) the total time it took to execute all the forks, and (3) the average time it took per fork. **The output of your program should be formatted exactly the same as in the example run** shown later on in this section **and show the time as microseconds**.

Example Run

Command Line

```
$ make clean                              # Remove any old executable made
rm -f TimeSysCall TimeFork TimeSignal
$ make TimeFork                        # Compile using make
gcc -Wall TimeFork.c -o TimeFork
$ ./TimeFork                           # Run the program
Forks Performed: XXXX
Total Elapsed Time: XXXX microseconds
Average Time Per Fork: XXXXX microseconds
```

Helpful References

The following are helpful references to help get you started

- **The fork() System Call**
  Link: https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
- **Fork() in C**
  Link: https://www.geeksforgeeks.org/fork-system-call/
- **Wait() in C**
  Link: https://www.geeksforgeeks.org/wait-system-call-c/
- **Waitpid() in C**
  Link: https://www.tutorialspoint.com/unix_system_calls/waitpid.htm

3 Signals

Signaling is important aspect of Interprocess Communication (IPC). A signal is a mechanism used for delivering an asynchronous event or notification to a process. Common reasons for a process to receive a signal are memory protection violations (SIGSEGV), divide by zero error (SIGFPE), and an expired timer (SIGALRM). Signals interrupt the normal flow of execution of a program and are generally handled by a

signal handler function. User programs can also register a signal handler, which gives the user program the ability to try and handle a signal in a specific way.

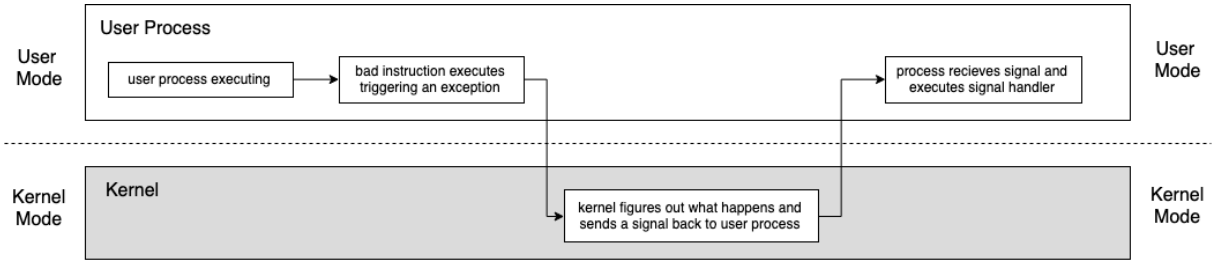

Figure 2: Execution flow when a signal handler is registered and a signal occurs

Given that signals often involve the kernel and interrupt normal user process execution, it can incur significant overheads if our user program is forced to handle many signals. In this part of the assignment we will understand how signals are handled as well as understand how much time handling a signal can incur.

Implementation

In order to get an idea of how long a signal handling can take, we want to measure the amount of time it takes to incur a signal and handle it. For this assignment we'll try and invoke a signal by simply dividing by zero. When dividing by zero, an exception is delivered to the kernel, and the kernel will deliver a floating point exception signal, also known as **SIGFPE**, to the user program where the signal will be handled. In order to get an accurate reading, we want to incur the signal many times (e.g. 100,000 times) and then calculate the average time it took for each signal. However, this is an issue as typically the default action to handle of these types of signals such as SIGFPE are to terminate the user process. In order to incur a signal multiple times and time it, we can override the policy to terminate the user process by registering our own signal handler for the SIGFPE signal.

A handler for SIGFPE can be registered in the following manner:

```
#include <signal.h>

void handle_sigfpe(int signum){
    // Handler code goes here
}

 int main(int argc, char **argv){
    int x = 5;
    int y = 0;
    int z = 0;
    signal(SIGFPE, handle_sigfpe); // Register signal handler
    z = x / y;                     // This causes the exception
    return 0;
 }
```

Registering Signal Handlers: As you go through the references, you will realize that there are two functions you can use to register a signal handler: (1) *signal()* and (2) *sigaction()*. Look at how both works. Using either one is acceptable.

Note on Signal Handling: Please note that if a signal handler function is allowed to return, the instruction that caused the signal will execute again, therefore causing the exception to occur again, ad infinitum. Your signal handler will need to detect when it has been invoked 100,000 times, then report the results and terminate the program. You can use a static/global variable for this purpose.

Supplied should be a *Makefile* and bare C program template called *TimeSignal.c*. Within *Timesignal.c* there will be a *handle_sigfpe()* function ready to be implemented. Your job to is to complete the implementation, incurring a SIGFPE and handling the signal with the signal handler 100,000 times and time it to calculate the following statistics: (1) how many exceptions occurred, (2) total time elapsed, and (3) the average time per exception. You should use the function *gettimeofday()* to time the execution like in the previous part of this assignment. **The output of your program should be formatted exactly the same as in the example run** shown later in this section **and show the time as microseconds**

Note: Measurement Precision. Like with measuring system call times in the previous section, you should use floating point arithmetic to show sub-microsecond precision. The average time to handle a signal should be a few microseconds.

Example Run

Command Line

```
$ make clean                              # Remove any old executable made
rm -f TimeSysCall TimeFork TimeSignal
$ make TimeSignal                         # Compile using make
gcc -Wall TimeSignal.c -o TimeSignal
$ ./TimeSignal                            # Run the program
Exceptions Occurred: XXXX
Total Elapsed Time: XXXX microseconds
Average Time Per Exception: XXXXX microseconds
```

Helpful References

The following are helpful references to help get you started in understanding Signals.

- **What is a Signal**
  Link: https://en.wikipedia.org/wiki/Signal_(IPC)
- **Signal Handling in Linux**
  Link: http://www.alexonlinux.com/signal-handling-in-linux#sigaction()
- **Signal Handling in Depth**
  Link: https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html
- **What happens when you dereference a null pointer? (How exceptions are handled)**
  Link: https://stackoverflow.com/questions/12645647/what-happens-in-os-when-we-dereference-a-null-pointer-in-c

Important Things to Note:

When completing your assignment make sure to keep the following things in mind:

- **Including other headers**
  You may need to include other headers to use certain functions like *gettimeofday()*, *wait()*, etc. This is okay, make sure you include these headers so that you can use these functions properly.
- **Make sure your programs can compile and run on the iLab machines**
  We will be grading your assignments on the iLab machines, so if your programs are unable to compile or run on the iLab machines, points will be deducted. No exceptions.
- **Make sure you can compile you programs with *make***
  We will be using the Makefile to compile your programs. If for some reason we can not compile your program with *make*, points will be deducted. No exceptions.

Submissions

To submit your assignment, simply submit the following files as is, directly to sakai as is. (**Do not compress the files**):

1. TimeSysCall.c
2. TimeFork.c
3. TimeSignal.c
4. Makefile (even if you didn't modify it)

Additional Tips and Tricks

When in doubt, Google: If you have an issue or your program is throwing an error that you don't know how to fix, Google It. Someone, somewhere, probably faced the same issue at some point.

Frequently Asked Questions

- **Can I use helper functions in my implementations?** Yes, as long as the program works as intended and the output is correct.
- **Can I use other *.c or *.h files in my implementation?** For the most part you shouldn't have to separate functionality into other files, but as long as the main files *TimeSysCall.c*, *TimeSignal.c*, and *Makefile* remain intact and as long as we will use the makefile to compile your programs before running.
- **Can I modify the makefile?** Yes as long as we can compile your programs via 'make TimeSysCall' and 'make TimeSignal'
- **Can we implement the programs on my own computer?** Yes as long as it can compile and run on the iLab machines as that is where they will be compiled, ran, and graded.

Additional Questions

If you have any questions about the assignment or are having any issues, email me at David.Domingo@rutgers.edu