CS416 Project 1: Operating System and Architecture Basics

Due: 09/28/2021

Points: 100

Description

The operating system is responsible for manage computer hardware resouces like CPU, DRAM and ensure applications, users and system administrators use those 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. In this project, you will need to measure the overhead of different system calls in Linux and hack the signal handler code to manipulate the execution flow of a program.

Part 1 System Call Overhead Measurement

1.1 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 (depending on the CPU architecture). Some common system calls include *read()* and *write()* which are used when reading and writing from/to 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. 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.

1.2 Measure System Call Costs

In the first part of this assignment, you will need to measure the costs of the following system calls: *getpid()*, *read()*. In order to get an accurate measurement, 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.

You will be given 2 separate source files **timeGetpid.c** and **timeRead.c** as a skeleton code. You need to add your own implementation of measuring the cost as described above. The read() system call is a little more complex than the getpid(), you will need to create a file of 512MB in your ilab home folder in advance, and each *read()* will read 4KB from the file sequentially, and iterate 512MB/4KB = 128K times. The desired output should be like the following when running your program:

```
Syscalls Performed: XXXX
Total Elapsed Time: XXXX microseconds
Average Time Per Syscall: XXXX microseconds
```

1.3 Useful Links

The getpid() man page

`http://man7.org/linux/man-pages/man2/getpid.2.html`

The read() man page

`https://man7.org/linux/man-pages/man2/read.2.html`

How to use time functions using gettimeofday() (Look at method #3)

`https://www.techiedelight.com/find-execution-time-c-program/`

Part 2 Signal Handler Hacking

2.1 Signal and Signal Handlers

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. 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.

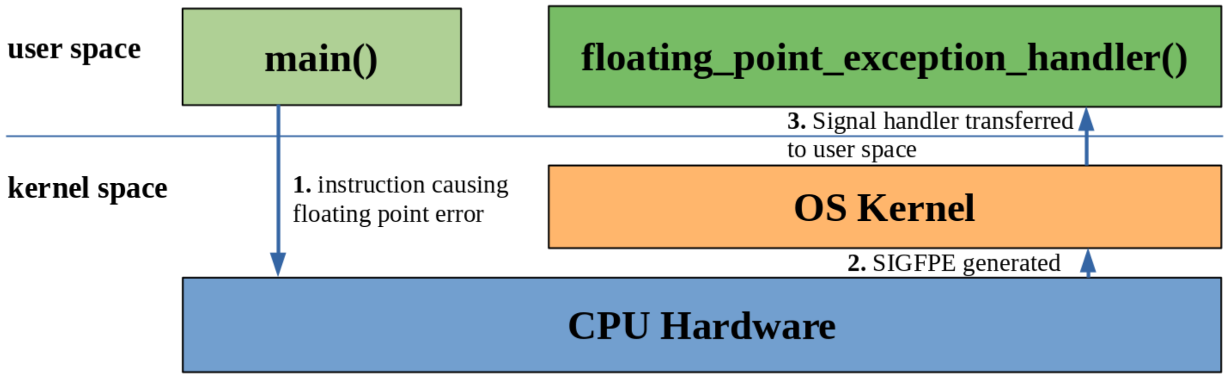
user space
main()
floating_point_exception_handler()
3. Signal handler transferred to user space
kernel space
1. instruction causing floating point error
OS Kernel
2. SIGFPE generated
CPU Hardware

Figure 1: How signal handler works

As it shown in the figure, once the signal hander is registered with OS (SIGFPE in this example), the user-defined signal handler callback function (floating_point_exception_handler() in this case) will be invoked for any floating point exception generated in the main() function in this example.

2.2 Hacking with Signal Handlers

In the skeleton code - **sigHandler.c**, void floating_point_exception_handler(int signum) – the signal handler will be called on any segmentation violation in this program. divided by zero is pretty much guaranteed to do that. Linux will first run your signal handler to give you a chance to address the floating point exception. If you return from the signal handler and the same signal occurs, then Linux will stop your code with a floating point exception. You must, in the signal handler only, make sure the floating point exception does not occur a second time. The problem is that Linux will attempt to run the offending instruction again. So you must somehow make it not run that instruction. The key to doing this is the signal number. What you need to do in this programming assignment is adding codes in the signal handler that causes the program to run to completion. You CANNOT change any code BUT the signal handler.

When your code hits the segment fault, it asks the OS what to do. The OS notices you have a signal handler declared, so it hands the reins over to your signal handler. In the signal handler, you get a signal number - signum as input to tell you which type of signal occurred. That integer is sitting in the stored stack of your code that had been running. If you grab the address of that int, you can then build a pointer to your code's stored stack frame, pointing at the place where the flags and signals are stored. You can now manipulate ANY value in your code's stored stack frame. Here are the suggested steps:

Step 1. divided by zero will cause a floating point exception. Thus, you also need to figure out 1) what exact instruction is causing the floating point exeception, and 2) the length of this bad instruction. (Hint, use GDB disassemble command to disassemble the main function)

Step 2. According to x86 calling convention, the program counter is pushed on stack frame before the subroutine is invoked. So, you need to figure out where is the program counter located on stack frame. (Hint, use GDB to show stack)

Step 3. Construct a pointer inside floating point exeption hander based on signum, pointing it to the program counter by incrementing the offset you figured out in Step 2. Then adding the program counter by the length of the bad instruction you figured out in Step 1.

The desired output should be like the following when you successfully manipulated the executing sequence of this program:

```
I am slain!
I live again!
```

2.3 Useful Links

Man Page of Signal

`http://www.man7.org/linux/man-pages/man2/signal.2.html`

Basic GDB tutorial

`http://www.cs.cmu.edu/~gilpin/tutorial/`

Submission

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

```
1. timeGetpid.c
2. timeRead.c
3. sigHandler.c
4. Makefile (even if you didn't modify it)
```