CS 416 Project 1: Warm-up Project

Due: February 13th, 2020

Points: 100 (5% of the overall course points.)

Introduction

This is a simple warm-up project that will help you to recall basic systems programming and get into the second project. In Part 1 of this project, you will recall details of the stack, processes, and how the OS executes code. In Part 2 you will use Linux pThread library to write a simple multi-threaded program. In Part 3, you will write a benchmark to measure the cost of a simple system call.

- Remember: You can discuss the logic but do not share the code!

Part 1: Signal Handler and Stacks (50 points)

In this part, you will learn about signal handling and stack manipulation.

In the skeleton code (*signal.c*), in the *main()* function, look at the line that dereferences memory address 0.

```
r2 = *( (int *) 0 );
```

If you compile and run the program, the statement will cause a segmentation fault:

```
$ gcc -g signal.c -o signal
$ ./signal
segmentation fault
```

In this case, a segmentation fault occurs and a signal is sent to the program. By default, a segmentation fault signal causes the program to terminate. However, we don't want this to happen. We want to use a signal handler to redefine our program's behaviour and allow it to run.

The goals of this part are to (1) handle the segmentation fault by registering a signal handler and (2) manipulate the stack to allow the program to continue.

1.1 Registering a Signal Handler

The first part is to handle the segmentation fault by installing a signal handler in the main function (*marked as Part 1 - Step 1 in project1.c*). If you register the following function correctly with the segmentation handler, Linux will first run your signal handler to give you a chance to address the segmentation fault.

```
**void segment_fault_handler(int signum)**
```

If your signal handler (*segment_fault_handler()*) is registered correctly, you will see the print out of the print statement within the signal handler:

```
$ gcc -g signal.c -o signal
$ ./signal
I am slain!
I am slain!
I am slain!
I am slain!
.....
```

You'll notice that it prints out forever. This is because upon the return signal handler, the program will continue execution starting from the offending instruction. In our case, the segmentation fault will occur and our signal handler will be ran over and over again indefinitely. We'll handle this next.

Note: In order to stop your program stuck running indefinitely, use *ctrl+c*.

1.2 Handling Segmentation Fault

The second part is to make sure the segmentation fault does not occur a second time. To achieve this goal, you must change the stack frame of the main function; else, Linux will attempt to rerun the offending instruction after returning from the signal handler. Specifically, you must change the program counter of the caller such that the below statement after the offending instruction gets executed. No other shortcuts are acceptable for this assignment.

```
printf("I live again!\n")
```

If the program counter is changed correctly, the main function should continue executing, printing out *"I live again!"*:

```
$ gcc -g signal.c -o signal
$ ./signal
I am slain!
I live again!
```

More details: 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 2. Dereferencing memory address 0 will cause a segmentation fault. Thus, you also need to figure out the length of this bad instruction.

Step 3. 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 4. Construct a pointer inside segmentation fault hander based on signum, pointing it to the program counter by incrementing the offset you figured out in Step 3. Then add the program counter by the length of the offending instruction you figured out in Step 1.

1.3 Tips and Resources

- 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/

Part 2: Recall of pThread Programming (30 points)

Provided is a program template *threads.c*. This program takes a single argument which determines how many times the global variable x should be incremented. Note how the incrementing of x occurs in the function *inc_shared_counter()*. The goal of this part is to create two threads to update the global counter x concurrently.

2.1 Threads

In part 2.1, you have use the pThread library to create 2 threads that runs the *inc_shared_counter()* function. Use the *pthread_create()* function to create two threads that will execute *inc_shared_counter*. After two threads finish incrementing counters, the final value of x will be printed. Remember, the main thread may terminate earlier than those 2 threads; hence, make sure to use *pthread_join* to let main thread wait for the threads to finish before exiting.

If done correctly, you should be able to threads print out that they are running and the final printout of x to change:

```
$ gcc -g threads.c -o threads -lpthread   // compile
$ gcc ./threads 100                       // run with threads incrementing x 100 times
Going to run two threads to increment x up to 100
Thread Running
Thread Running
The final value of x is 100
```

However you should ensure both threads are running simultaneously. One way to do that is to pass a large argument to the program. If you run the program with a large enough argument (over 100,000), you should see that the final value of x turns out to be lower than expected (we'll go over why in the next part):

```
$ gcc -g threads.c -o threads -lpthread   // compile
$ gcc ./threads 100000                   // run with threads incrementing x 100k times
Going to run two threads to increment x up to 100
Thread Running
Thread Running
The final value of x is 59302
```

Note: If you have other print statements within the loop, printing out the x variable you may not see this. It is recommended you remove any print statements with.

If you don't have any print statements within the for-loop and running the program with over 100000 still yields x with a value of 100000, then the threads may not be running simultanously and you should double check how you created and joined your threads.

2.2 Mutual Exclusion

In the previous part, you were able to see that if you pass a large enough argument, the final value of x turns out to be lower than expected. This is because the two threads are simultanously trying to modify the shared variable x. The two threads may read the value of x at the same time, resulting in both incrementing x by one and writing the same value back to memory, so instead of the two thread incrementing x twice, they really only incremented x once.

In order to fix this, you must use mutexes to ensure the incrementing of x within the main thread loop is only done by one thread at a time.

If implemented correctly, you should be able to pass in any argument to the *threads* program and the final value of x should always match the value you gave:

```
$ gcc -g threads.c -o threads -lpthread   // compile
$ gcc ./threads 100000                   // run with threads incrementing x 100k times
Going to run two threads to increment x up to 100000
Thread Running
Thread Running
The final value of x is 100000
```

2.3 Tips and Resources

- Make sure to compile with `-lpthread` to ensure the pthread library is linked and the pthread functions can be found
- Man Page for pthread_create: http://man7.org/linux/man-pages/man3/pthread_create.3.html
- Man Page for pthread_join: http://man7.org/linux/man-pages/man3/pthread_join.3.html
- Mutexes for thread synchronization: https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

Part 3: Measure System Call Cost (20 points)

The last part of this project is to measure the system call cost. We want to see how long it would take to run a fast system call and get an idea of how long it takes to go to kernel mode and back to user mode.

(1) How to invoke a system call and (2) time the average time to invoke a system call.

3.1 Doing a system call

One way to invoke a system call is with syscall() function. The *syscall()* function requires at least a single argument specifying the number of the system call to run. In this case we want to run a relatively fast system call. For this project, use *getuid*, The *getuid* system call simply returns the user ID of the calling process. Since this system call simply fetches the user ID, it should be fast. Your task is to use *syscall()* to call run the *getuid* system call.

Hint: Look at the man page for *syscall()* and look at the example of how to use it.

3.2 Finding the average time of a system call

In order to time a system call we can use the *gettimeofday()* function to record the time before the system call and record the time after the system call. The time it took for the system call to execute would be the difference between the starting and ending time. However on modern computers, system calls are pretty fast, so timing just one system call may not give us accurate or precise results.

In order to get more accurate idea of how long a single system call should take, you should measure the time it takes to run the system call many, many times and calulcate the average time to run it.

For this part you should time how long it takes to run the system call 5,000,000 times and use the result to calcualte the average time per system call. You should be able to be see consistent times with precision up to hundreths of a microsecond.

Take the final value and place it in the *avg_time* variable so it prints out in the end.

The output should look like the following:

```
$ gcc -g syscall.c -o syscall   // compile
$ ./syscall                     // run
Average time per system call is X.XXXX microseconds
```

3.3 Some tips and references

- Man page for syscall: http://man7.org/linux/man-pages/man2/syscall.2.html
- Man Page for measuring time: https://linux.die.net/man/7/time

---

4 Submission

Your project submission will consist of the following files: - signal.c - threads.c - syscall

Before uploading, prepare each of your source code files by do the following: - add all group member names and NetID and the iLab machine you tested your code as a comment at the top of each source file. - remove any extra print statments or output modifications

When your files are ready to submit, upload them directly to Sakai as is (**Do not compress them**)

Other things to note

- Your code must work on one of the iLab machines. Your code must use the provided C code as a base of your implementations. Feel free to change the function signature for Part 2 and Part 3 if required.
- Points will be removed for any extra print statements or output modifications so make sure revert any changes to output before submission.