CS 416 Project 1: Understanding the Stack and Basics (Warm-up Project)

Due: 02/15/2021

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

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 system call cost.

- 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 handler and stack manipulation.

1.1 Description

In the skeleton code (*project1.c*), in the *main()* function, look at the line that dereferences memory address 0. This statement will cause a segmentation fault.

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

The first goal 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 segment fault.

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

Goal: You must, in the signal handler, 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 statement *printf("I live again!")* after the offending instruction gets executed. No other shortcuts are acceptable for this assignment.

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.

1.2 Desired Output

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

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 (25 points)

2.1 Description

In the skeleton code, there is a global variable x, which is initialized to be 0. You are required to use pThread to create 2 threads to increment global variable by 1 for totally 10 times (5 times for each thread);

Use *pthread_create* to create two threads and each of them will execute *inc_shared_counter*. Inside *inc_shared_counter*, increment x 5 times.

Because x is a shared variable, you need a mutex to guarantee the exclusive access and order. After each increment, you are also required to print the current value of x to the console.

After two threads finish incrementing counters, you must print the final value of x to the console. 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.

2.2 Desired Output

The desired output of the program should be the following:

```
x is incremented to 1
x is incremented to 2
x is incremented to 3
x is incremented to 4
x is incremented to 5
x is incremented to 6
x is incremented to 7
x is incremented to 8
x is incremented to 9
x is incremented to 10
The final value of x is 10
```

Hint: Because we are using mutex, the program output is deterministic. Therefore, your program MUST show the exact same output as shown above.

2.3 Tips and Resources

- 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

Part 3: Measure System Call Cost (10 points)

The last part of this project is to measure the system call cost. Write code in the **syscall_benchmark()** by invoking some Linux system call few thousand times and averaging the cost.

- Man Page for measuring time: https://linux.die.net/man/7/time

FAQs

Q1: Project 1 Submission

Should we submit the source code or the binary?

A:

- You should be submitting the source code *only* (signal.c, thread.c, syscall.c).
- Please add all group member names and NetID and the iLab machine you tested your code as a comment at the top of the code file.
- Your code must work on one of the iLab machines. Your code must use the attached C code as a base and the functions. Feel free to change the function signature for Part 2 and Part 3 if required.

Q2: Using signum

Should we use and modify the address of signum (or a local pointer to signum)? Can we use other addresses (ie. the main function stack pointer)?

A:

1. Yes, we expect you to use signum (though we prefer using signum itself, using a local pointer to signum in the handler does the same thing).
2. No, you cannot. You should use **signum as your pointer to the stack**, and subsequently, manipulate the stack.

Q3: Using Other Packages

Can we use other packages such as asm.h to manipulate the registers?

A:

No, we expect you to use the packages included in the file. **The goal of the project is to understand the stack, learn how registers are stored on the stack, and using GDB to inspect stack frames** (as well as getting used to using it in general). Although using asm is one way of doing this, the solution we expect is not to through asm (or any other packages that we do not include), so a submission using this will have points deducted.