CS 416 + CS 518 Project 1: Understanding the Stack and Basics (Warm-up Project)

Due: 02/14/2022, Time: 11:59pm EST

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

This simple warm-up project will help you recall basic systems programming before the second project. This project's first part will help you recollect how stacks work. In part 2, you will use Linux pThread library to write a simple multi-threaded program. In part 3, you will write functions for simple bit manipulations. We have given three code files, *signal.c*, *threads.c*, *bit-shifting.c*, for completing the project.

Part 1: Signal Handler and Stacks (50 points)

In this part, you will learn about signal handler and stack manipulation and also write a description of how you changed the stack.

1.1 Description

In the skeleton code (*signal.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 signal.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: In the signal handler, you must 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 Compiling for 32-bit (-m32) or 64-bit Compilation

To reduce the project's complexity, you can compile the *signal.c* for either 32-bit or 64-bit mode. When compiling for 32-bit mode, pass an additional flag to your gcc (*-m32*), whereas for the 64-bit, you don't have to pass any additional flag. The 32-bit compilation makes it a bit easier because, in an x86 32-bit mode, the function arguments are always pushed to the stack before the local variables, making it easier to locate *main()'s* program counter. In a 64-bit mode, when you add additional local variables in your segfault handler function, this might not always be true (which could be confusing to some students).

NOTE: We will accept a solution that works for either 32-bit or 64-bit mode. Please just mention what you used in your report.

1.3 Desired Output

```
handling segmentation fault!
result after handling seg fault 100!
```

1.4 Report

Please submit a report that answers the following question. Without the report, you will not receive points for this part. 1. What are the contents in the stack? Feel free to describe your understanding. 2. Where is the program counter, and how did you use GDB to locate the PC? 3. What were the changes to get the desired result?

1.5 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: pThread Basics (25 points)

In this part, you will learn how to use Linux pthreads and update a shared variable. We have given a skeleton code (*thread.c*).

2.1 Description

In the skeleton code, there is a global variable x, which is initialized to 0. You are required to use pThread to create 2 threads to increment the global variable by 1.

Use *pthread_create* to create two worker threads and each of them will execute *inc_shared_counter*. Inside *inc_shared_counter*, each thread increments x 10000 times.

After the two worker threads finish incrementing counters, you must print the final value of x to the console. Remember, the main thread may terminate before the worker threads; avoid this by using *pthread_join* to let the main thread wait for the worker threads to finish before exiting.

You do not have to synchronize updates across the threads. Therefore, you might notice inconsistency in the final result across multiple runs. In later projects, we will discuss how to use mutex and other synchronization methods.

2.2 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

2.3 Report

You do not have to describe this part in the project report.

Part 3: Bit Manipulation (25 points)

3.1 Description

Understanding how to manipulate bits is an important part of systems/OS programming and is required for subsequent projects. As a first step towards getting used to bit manipulation, you will write simple functions to extract and set bits. We have provided a template file **bit-shifting.c**

3.1.1 Extracting top order bits

Your first task is to write a program to find the top-order bits by completing the *get_top_bits()* function. For example, let's assume the global variable *myaddress* is set to 4026544704. Now let's say you have to extract just the top (outer) 4 bits (1111), which is decimal 15. Your function *get_top_bits()* that takes the value of *myaddress* and the number of bits (*GET_BIT_INDEX*) to extract as input arguments and would return 15 as a value.

3.1.2 Setting and Getting bits at at specific index

Setting bits at a specific index and extracting the bits is widely used across all OSes. You will be using these operations frequently in your projects 2, 3, and 4. You will complete two functions *set_bit_at_index()* and *get_bit_at_index().*

Before the *set_bit_at_index* function is called, we will allocate a bitmap array (i.e., an array of bits) as a character array, specified using the *BITMAP_SIZE* macro. For example, when the BITMAP_SIZE is set to 4, we can store 32 bits (8 bits in a character element). The *set_bit_at_index* function passes the bitmap array and the index (SET_BIT_INDEX) at which a bit must be set. The *get_bit_at_index()* tests if a bit is set at a specific index.

Note, you will not be making any changes to the main function, but just the three functions, *get_top_bits()*, *set_bit_at_index()*, and *get_bit_at_index().*

When evaluating your projects, we might change the values *myaddress* or other MACROS. You don't have to worry about dealing with too many corner cases for this project (for example, setting myaddress or other MACROS to 0).

3.2 Report

You do not have to describe this part in the project report.

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, *bit-shifting.c) and the project report for Part 1.
- Please add all group member names and NetID, course number (CS 416 or CS 518), 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 if required.

Q2: Using signum

(1) Should we use and modify the address of signum (or a local pointer to signum)?
(2) 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.