CS 416 + CS 518 Project 1: Stacks and Basics (Warm-up Project)

Due: 4th October, 2023, Time: 8:00 pm ET

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 write functions for simple bit manipulations. In part 3, you will use Linux pThread library to write a simple multi-threaded program. We have given three code files, *stack.c*, *threads.c*, *bitops.c*, for completing the project.

PLEASE NOTE: *We will perform sophisticated plagiarism checks on all projects submitted throughout the semester. We will compare your submission with those of other students from this semester, and also submissions from students across several previous offerings of this course. Please abide by Rutgers academic integrity policies. Consequences of violation range from failing the course to separation from your degree program. If you are ever in doubt, please ask us first.*

Part 1: Signal Handler and Stacks (35 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 (*stack.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 stack.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 signal_handle(int signalno)**
```

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 *r2 = r2 + 1* 30;* 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 - signalno 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 fault hander based on signalno, 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) To reduce the project's complexity, please compile the *stack.c* for 32-bit bypassing pass an additional flag to your gcc (*-m32*). 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.

1.3 Desired Output

```
handling segmentation fault!
result after handling segfault...
```

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: Bit Manipulation (35 points)

2.1 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 **bitops.c** Setting a bit refers to updating a bit to 1 if the bit is 0. Clearing a bit refers to changing a bit with 1 to 0.

2.1.1 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.

2.1.2 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().* Note that each byte has 8 bits.

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.

So, if one allocates a bitmap array of 4 characters, bitmap[0] could refer to byte 0, bitmap[1] could refer to byte 1, and so on.

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).

2.2 Report In your report, describe how you implemented the bit operations.

Part 3: pThread Basics (30 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*).

To use Linux pthread library, you must compile your C files by linking to pthread library using *-lpthread* as a compilation option.

3.1 Description In the skeleton code, there is a global variable x, which is initialized to 0. You are required to use pThread to create 4 threads to increment a global variable.

Use *pthread_create* to create four worker threads and each of them will execute *add_counter*. Inside *add_counter*, each thread increments the global variable x by *loop* times (e.g., 10000 times).

After the worker threads finish incrementing the global variable, 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.

Because x is a shared variable, you need to synchronize across threads to guarantee that each thread exclusively updates that shared variable. For synchronization, you can use pthread mutex (a locking mechanism, recollect from CS 214!).

You can read more about using pthread mutex in the link [3]. In later lectures and projects, we will discuss how to use other complex synchronization methods.

If you have implemented the thread synchronization correctly, the final value of x would be 4 times the loop value.

NOTE: When evaluating your projects, we will test your code for different loop values.

3.2 Tips and Resources

- [1] pthread_create: http://man7.org/linux/man-pages/man3/pthread_create.3.html
- [2] pthread_join: http://man7.org/linux/man-pages/man3/pthread_join.3.html
- [3] pthread_mutex: https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html

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

Project 1 Submission

For submission, create a Zip file of your submission and upload to Canvas. The zip file name should be **project1.zip**. The zip file must contain the following files. Please note that the report file should be a PDF (please do not submit word or text files).

project1.zip (in lower case) (1) stack.c (2) thread.c (3) bitops.c (4) report.pdf

- Only **one group member** submits the code. But in your project report and at the top of your code files, **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.

- **Cite your references:** If you consulted resources other than the references listed in this document, or discussed the project with people outside of your teammate and the course staff, please note them here and describe the nature of your consulation.

FAQs

Q1: Using signalno

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

(2) Can we use other addresses (ie. the main function stack pointer)?

A:

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

Q2: 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.