CS 416 Project 1: Understanding the Stack and Basics

Names: Michael Liu msl196

Part 1

Questions:

1. **What are the contents in the stack? Feel free to describe your understanding.**
   When a function is called, which is my case where I observed the stack when *signal_handle* is called, the stack contains stuff such as the function's return address, local variables, the function's parameters and the program counter.

2. **Where is the program counter, and how did you use GDB to locate the PC?**
   In 32-bit x86 architecture, the program counter is pushed onto the stack when calling a function, and since the program counter is stored inside the EIP register, using the command **info frame** in GDB will show the address where the EIP is stored, in my case
   eip = 0x56556298. Thus, now we know the location of the PC.
   Next, I showed the contents of the current stack by using the command **x/32x $esp**

```
(gdb) x/32x $esp
0xffffc350:     0x3afcf44b      0x00000000      0x00000000      0x00000000
0xffffc360:     0x00000000      0x56558fd0      0xffffd078      0xf7fc4560
0xffffc370:     0x0000000b      0x00000063      0x00000000      0x0000002b
0xffffc380:     0x0000002b      0xf7ffcb80      0xffffd144      0xffffd078
0xffffc390:     0xffffd060      0x56558fd0      0x00000000      0x00000000
0xffffc3a0:     0x00000000      0x0000000e      0x00000004      0x56556298
0xffffc3b0:     0x00000023      0x00010286      0xffffd060      0x0000002b
0xffffc3c0:     0xffffc650      0x00000000      0x00000000      0x00000000
```

   Next, printing the local variable by using **print signalno** results in 11, which is exactly **SIGSEGV** and in hex that is **0xb**
   As we can see from my stack above, our local variable **signalno** and the **program counter** is easily locatable, and their offset by counting is **15.**
   Therefore, all we need to do now is to offset our existing signalno pointer by adding 15, making it point to the PC, and incrementing that value by 1 will successfully bypass the segmentation fault.

   What were the changes to get the desired result?
   In order to bypass the program causing a segmentation fault, I first constructed a pointer based on signalno, then incremented it by the offset I observed by look at the stack, which makes it points to the program counter, and lastly by incrementing that pointer, we change the program counter so it skips the instruction that triggers the segmentation fault. And also in the main method, I registered my custom custom signal_handle function to handle the SEGV signal so that it will be called before the segmentation fault is triggered.

Part 2

Note: please compile bitops.c with the -lm compiler option to link the math library since I used it in get_top_bits

2.1.1 Extracting Top-Order Bits

In order to extract the top n bits from a value, my idea was to just shift the top n bits all the way to the right (lower bits), thus "extracting" it.

For instance, if we wanted to extract the top 4 bits of a 32-bit number 4026544704, we would just shift it 28 times to the right, resulting in the top bits 1111

$$11110000000000000011001001000000$$

$$\downarrow$$

$$00000000000000000000000000001111$$

To calculate how many bits to shift I just subtracted n from the total number of digits of the original number $(calculated\ by: \log(x) + 1)$. So in this case $(\log(4026544704) + 1) - 4 = 32 - 4 = 28$

2.1.2 Setting and Getting Bits at a Specific Index

set_bit_at_index

In order to set a bit at a specific index, my idea was to use the bitwise **or** operation, since

$0 \mid 0 = 0\ and\ 1 \mid 0 = 1$, meaning that if you **or** any binary number with 0 it will result in that original number, and

$0 \mid 1 = 1\ and\ 1 \mid 1 = 1$, meaning that if you **or** any binary number with 1 it will result in 1 (setting the bit)

We can create a mask of the same length as the original number, where the bit at the target index will be 1 and the rest will all be 0, and if we or that mask with the original number, we would "set" the target bit to 1 and leaving the rest unchanged.

For instance, if we wanted to set the 17th bit in a 32-bit number:

$$00000000\ 00000000\ 00000000\ 00000000 \mid 00000000\ 00000000\ 10000000\ 00000000$$

Where the left-hand side can be any number, this will mask the 17th bit making it 1.

get_bit_at_index

For getting a bit, I used the bitwise **and** operation since, $0\ \&\ 1 = 0\ and\ 1\ \&\ 1 = 1$, meaning that for a single digit if you **and** it with 1 you get the digit, so now all we need to do is shift the target bit all the way to the right, anding it would "get" the value of the single target bit.

The number of times to shift the target bit can just be calculated by first getting the current byte num $(index\ /\ 8)$ and then get the current bit num in that targeted byte $(index\ \%\ 8)$, and the number of times to right shift the current byte is just the current bit num.