Project 1 Report

- Sanjay Kethineni (sk2425) and Adhit Thakur (at1186)
- CS416
- ilab machine used: pascal.cs.rutgers.edu
- Compile and running commands for each file are commented at the top of the file

Part 1: Signal Handler and Stacks

- **What are the contents in the stack? Feel free to describe your understanding**
    - The stack is a region of memory, with each method (in this case, main and signal_handle) containing its own stack frame. Within a stack frame, there are the function parameters, return addresses, and local variables for the frame's method. Within GDB there are base addresses provided when viewing the stack so that an offset can be used to reach a desired location.
- **Where is the program counter, and how did you use the GDB to locate the PC?**
    - The program counter for a method is a pointer that is used to locate the current executing instruction for a method. When performing a system call such as signal(), there is a mode switch, with the system call number and program counter being stored on the stack for the function. In the specific signal() system call that was used in this program, once the system call was made, the stack frame for the signal_handle can be viewed. Within the stack frame for the signal_handle method, the program counter for the main method resides. Through GDB, the "disassemble" method was used prior to the divide by 0 error to locate the address for the incorrect division. This memory address was noted, and once the stack frame for the signal_handle method was entered, the "x/40x $sp" command was used to view the stack frame for the method. The program counter was found within this display, with its value being the same address as the location of the divide by 0 error. This makes sense, since once the signal_handle method is exited, the next instruction that would be executed under normal circumstances would be the divide by 0 statement.
        - Using disassemble to locate the program counter

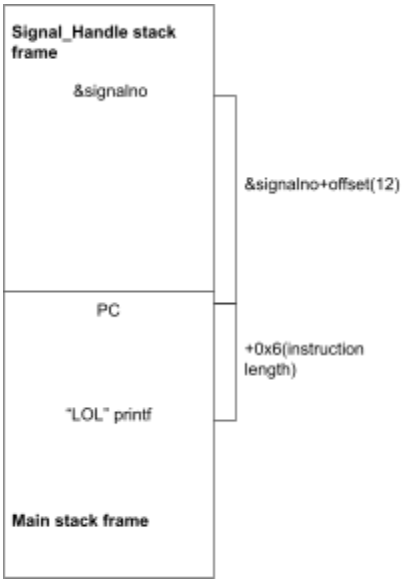
```
   0x5655623e <+73>:    cltd
   0x5655623f <+74>:    idivl  -0x10(%ebp)
--Type <RET> for more, q to quit, c to continue without paging--
   0x56556242 <+77>:    mov    %eax,-0xc(%ebp)
   0x56556245 <+80>:    sub    $0x8,%esp
   0x56556248 <+83>:    push   -0xc(%ebp)
   0x5655624b <+86>:    lea    -0x1fb6(%ebx),%eax
   0x56556251 <+92>:    push   %eax
   0x56556252 <+93>:    call   0x56556050 <printf@plt>
   0x56556257 <+98>:    add    $0x10,%esp
   0x5655625a <+101>:   mov    $0x0,%eax
   0x5655625f <+106>:   lea    -0x8(%ebp),%esp
   0x56556262 <+109>:   pop    %ecx
   0x56556263 <+110>:   pop    %ebx
   0x56556264 <+111>:   pop    %ebp
   0x56556265 <+112>:   lea    -0x4(%ecx),%esp
   0x56556268 <+115>:   ret
End of assembler dump.
(gdb) n

Program received signal SIGFPE, Arithmetic exception.
0x5655623f in main (argc=1, argv=0xffffd0c4) at stack.c:24
24          z=x/y;
(gdb) n
signal_handle (signalno=8) at stack.c:9
9       void signal_handle(int signalno) {
(gdb)
10          printf("OMG, I was slain!\n");
(gdb) x/40x $sp
0xffffc1e0:     0x00000000      0x56558fd0      0xffffcff8      0xf7fc4560
0xffffc1f0:     0x00000008      0x00000063      0x00000000      0x0000002b
0xffffc200:     0x0000002b      0xf7ffcb80      0xffffd0c4      0xffffcff8
0xffffc210:     0xffffcfe0      0x56558fd0      0x00000000      0x00000000
0xffffc220:     0x00000005      0x00000000      0x00000000      0x5655623f
0xffffc230:     0x00000023      0x00010282      0xffffcfe0      0x0000002b
0xffffc240:     0xffffc4d0      0x00000000      0x00000000      0x00000032
```

- **What were the changes to get the desired result?**
    - In order to get the desired result, a few calculations needed to be made. Firstly, the distance between the signalno variable and the program counter for the main function needed to be calculated. The “p /x &signalno” command was used in GDB to get the address for the signalno variable. After this was done, the “x/40x $sp” command was used to locate the program counter, as previously mentioned. Because the address location was 12 words away from the base address of 0xffffc220 (as can be seen in the image from the previous question), we performed the “p /x 0xffffc220+12” operation to get the address for the program counter. Becuase the program counter had to be updated in reference from the signalno variable, we also used “p /x &signalno” to get the variable’s address. A final calculation of (0xffffc220+12)-(&signalno) was done to get the distance between signalno and the program counter. If the calculated distance was added to the &signalno, we would reach the stack frame of the main method. However, instead of simply returning back to what the program was originally pointing to (the divide by 0 line), we want to update the address so that it points to the “LOL” print statement. Thus, we calculated the instruction length by looking at the “disassemble” information prior to the divide by 0 function call. As can be seen in the image below, we calculated the distance between the divide by 0 call (original PC location) and the sub call, since this was where space would be made within the stack pointer to perform the print operation. Ultimately, by taking the &signalno, adding the offset to get to the location of the PC, and adding on the length of the instruction to get to the printf function call, we were able to manipulate the stack frame to get to the print statement despite a divide by 0 error.

```
0x5655623e <+73>:    cltd
0x5655623f <+74>:    idivl  -0x10(%ebp)
0x56556242 <+77>:    mov    %eax,-0xc(%ebp)
0x56556245 <+80>:    sub    $0x8,%esp
0x56556248 <+83>:    push   -0xc(%ebp)
0x5655624b <+86>:    lea    -0x1fb6(%ebx),%eax
0x56556251 <+92>:    push   %eax
0x56556252 <+93>:    call   0x56556050 <printf@plt>
0x56556257 <+98>:    add    $0x10,%esp
0x5655625a <+101>:   mov    $0x0,%eax
0x5655625f <+106>:   lea    -0x8(%ebp),%esp
0x56556262 <+109>:   pop    %ecx
0x56556263 <+110>:   pop    %ebx
0x56556264 <+111>:   pop    %ebp
Type <RET> for more, q to quit, c to continue without paging--
0x56556265 <+112>:   lea    -0x4(%ecx),%esp
0x56556268 <+115>:   ret
d of assembler dump.
db) p /x 0x56556245-0x5655623f
 = 0x6
```

- Here is a diagram showing why the calculations that were made allowed for the proper change



Part 2: Bitops

Implementing bit operations was pretty straightforward. In the code, the bit index the original author wanted to edit was “17”, meaning they wanted to edit the 17th bit in the byte array. In a char array with size 4 (4 chars), that would mean the array size is 4 bytes. To figure out what byte contains the bit that we are planning to edit, we divide the bit number by 8, 17 divided by 8, which gives us 2. This means that the 17th bit is part of the 2nd byte (0 indexed). After that, we use modulo (17 % 8) to figure out which bit to edit. The modulo gives us 1, so it tells us to edit the first bit in the 2nd byte.

To get the first bit, first we use x & -x, which isolates the rightmost set bit in a binary number. Applying log2 to this result gives us the rightmost set bit index.

To set a bit, we use |=. To read a bit, we use &.