Project 1 (Warm Up) Report
Names: Saleem Khan (sk2304), Chirantan Patel (csp126)

Part 1: Signal Handler and Stacks

1. What are the contents in the stack?

The stack is a region of memory where local variables, function parameters, return addresses, and saved registers are stored. When a function is called a new frame is pushed onto the stack containing all the details. When a signal handler is called due to an interrupt which is our segmentation fault, that state of interruption is also pushed onto the stack.

- All the local variables which are stored in stack:
    - From main function: argc, argv[], r2
    - From signal_handle: signalno, pSig, offset, bLen
- Return addresses are also stored in stack. When 'signal_handle' is executed after the segmentation fault, the return address pointing to the line in 'main' that causes the fault will be on the stack. So that is 'r2 = *(int *) 0);' will be on stack.
- Other content is control data for the signal like signal number will also be stored on the stack.

2. Where is the program counter, and how did you use GDB to locate the PC?

The program counter is a CPU register that holds the address of the next instruction to be executed. You can see it in GDB.

- We first compile the program like this: gcc -g stack.c -o stack
- Then we run using GDB: gdb stack
- We set the breakpoint at the beginning of the 'signal_handle': break signal_handle
- We can run the program
- Now when the breakpoint is hit, this will be after the segmentation fault occurs and the handler is called. Now we do: info registers. The program counter will be shown in the list of registers. From our run we can see 'eip' (Program counter is also known as instruction pointer(ip), so register name is eip in 32bit) which is the program counter holding an address and the <signal_handle+17> means program counter is within the 'signal_handle' function, specifically 17 bytes offset from the start of that function.

```
(gdb) info registers
eax            0x56558fd0          1448447952
ecx            0x0                 0
edx            0x0                 0
ebx            0x56558fd0          1448447952
esp            0xffffc890          0xffffc890
ebp            0xffffc8a8          0xffffc8a8
esi            0xffffd144          -11964
edi            0xf7ffcb80          -134231168
eip            0x565561ce          0x565561ce <signal_handle+17>
eflags         0x212               [ AF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

3. What were the changes to get the desired result?

The changes were in the ‘signal_handle’ function. We have to be able to make the program skip the line that caused the segmentation fault and proceed with the next line of code. So first we have to find a location on the stack where the return address (from the ‘signal_handle’ back to the main function) is stored. This in our program is done using ‘pSig’ and ‘offset’. After we increment the return address by 2 using ‘*bLen +=2;’. This modifies the return address to point to the instruction after ‘r2 = *(int *) 0);’ We got the offset value to be 15 and bLen += 2 by using gdb.

```
csp126@candle:~/cs416$ gcc -m32 -g -o stack stack.c
csp126@candle:~/cs416$ gdb stack
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...
(gdb) break signal_handle
Breakpoint 1 at 0x11ce: file stack.c, line 17.
(gdb) run
Starting program: /common/home/csp126/cs416/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
main (argc=1, argv=0xffffd144) at stack.c:38
38          r2 = *( (int *) 0 ); // This will generate segmentation fault
(gdb) continue
Continuing.

Breakpoint 1, signal_handle (signalno=11) at stack.c:17
17          printf("handling segmentation fault!\n");
(gdb) backtrace
#0  signal_handle (signalno=11) at stack.c:17
#1  <signal handler called>
#2  main (argc=1, argv=0xffffd144) at stack.c:38
(gdb) info frame
Stack level 0, frame at 0xffffc8b0:
 eip = 0x565561ce in signal_handle (stack.c:17); saved eip = 0xf7fc4560
 called by frame at 0xffffd060
 source language c.
 Arglist at 0xffffc8a8, args: signalno=11
 Locals at 0xffffc8a8, Previous frame's sp is 0xffffc8b0
 Saved registers:
  ebx at 0xffffc8a4, ebp at 0xffffc8a8, eip at 0xffffc8ac
(gdb) x/20x $sp
0xffffc890:     0x00000000      0x00f80dac      0xffffc830      0xf7fcc1f3
0xffffc8a0:     0xf7d71ac9      0x56558fd0      0xffffd078      0xf7fc4560
0xffffc8b0:     0x0000000b      0x00000063      0x00000000      0x0000002b
0xffffc8c0:     0x0000002b      0xf7ffcb80      0xffffd144      0xffffd078
0xffffc8d0:     0xffffd060      0x56558fd0      0x00000000      0x00000000
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x56556214 <+0>:     lea    0x4(%esp),%ecx
   0x56556218 <+4>:     and    $0xfffffff0,%esp
   0x5655621b <+7>:     push   -0x4(%ecx)
   0x5655621e <+10>:    push   %ebp
   0x5655621f <+11>:    mov    %esp,%ebp
   0x56556221 <+13>:    push   %ebx
   0x56556222 <+14>:    push   %ecx
   0x56556223 <+15>:    sub    $0x10,%esp
   0x56556226 <+18>:    call   0x565560c0 <__x86.get_pc_thunk.bx>
   0x5655622b <+23>:    add    $0x2da5,%ebx
   0x56556231 <+29>:    movl   $0x0,-0xc(%ebp)
   0x56556238 <+36>:    sub    $0x8,%esp
   0x5655623b <+39>:    lea    -0x2e13(%ebx),%eax
   0x56556241 <+45>:    push   %eax
   0x56556242 <+46>:    push   $0xb
   0x56556244 <+48>:    call   0x56556060 <signal@plt>
   0x56556249 <+53>:    add    $0x10,%esp
   0x5655624c <+56>:    mov    $0x0,%eax
   0x56556251 <+61>:    mov    (%eax),%eax
--Type <RET> for more, q to quit, c to continue without paging--
   0x56556253 <+63>:    mov    %eax,-0xc(%ebp)
   0x56556256 <+66>:    addl   $0x1e,-0xc(%ebp)
   0x5655625a <+70>:    sub    $0x8,%esp
   0x5655625d <+73>:    push   -0xc(%ebp)
   0x56556260 <+76>:    lea    -0x1fa8(%ebx),%eax
   0x56556266 <+82>:    push   %eax
   0x56556267 <+83>:    call   0x56556050 <printf@plt>
   0x5655626c <+88>:    add    $0x10,%esp
   0x5655626f <+91>:    mov    $0x0,%eax
   0x56556274 <+96>:    lea    -0x8(%ebp),%esp
   0x56556277 <+99>:    pop    %ecx
   0x56556278 <+100>:   pop    %ebx
   0x56556279 <+101>:   pop    %ebp
   0x5655627a <+102>:   lea    -0x4(%ecx),%esp
   0x5655627d <+105>:   ret
```

Part 1 References:

https://www.geeksforgeeks.org/signals-c-language/
https://www.geeksforgeeks.org/gdb-step-by-step-introduction/
https://www.geeksforgeeks.org/segmentation-fault-sigsegv-vs-bus-error-sigbus/

Part 2: Bit Manipulation

Extracting Top Order Bits

To extract the top order bits from the parameter “value” we computed the number of bits that were not needed to be stored, in other words we computed the number of bits to shift right such that only the number of top bits equal to the argument “num_bits” remained in the 32 bits where an integer value would be stored. To compute the number of bits to shift right we got the number of bits in an integer using ‘sizeof(integer) * 8;’ and subtracted the argument “num_bits” from that value. After we know the number of bits to shift right we return the argument value shifted right by the number we had previously calculated.

Set Bit At Index

To implement set_bit_at_index function we had computed two values byte_index(the byte at which the bit we want to set is located in bitmap) and bit_index(the index of the bit in the byte). How we calculated the two values specified is shown below:

- byte_index = index / 8;
- bit_index = index % 8;

Afterwards we created a variable “byte” with type char * to directly access the byte within bitmap and a variable “mask” of type char which is equal to 1 << bit_index(value of 1 shifted left by bit_index number of bits). Using the dereferenced pointer *byte(which references the byte in bitmap that we want to update) we assigned to it the value *byte | mask. A more specific example shown below:

- char *byte = bitmap + byte_index;
- char mask = 1 << bit_index;
- *byte = *byte | mask;

Bitwise or will work as mask contains all zeros except for a single one which is located at the same index we are trying to set, hence when we assign the value obtained from the bitwise or to the byte where the bit is located in bitmap we can assume that the bit is set if it was previously a zero otherwise it doesn’t need to be updated.

Get Bit At Index

To implement get_bit_at_index we used a similar approach as before where we get the byte in which the bit is located in the bitmap, the index of the bit within the byte that is in the bitmap, and the value of the byte/char which contains the bit. The code is specified below to clarify any confusion relating to previous wording:

- byte_index = index / 8;
- bit_index = index % 8;
- char byte = bitmap[byte_index];

We used bitwise and with byte and 1 << bit_index(1 shifted left to the index of the bit we want to get the value of). Since the 1 is shifted left to the index of the bit we are checking, the resulting value will only contain a single one(in binary), hence any bit at any other position aside from the position/bit we are checking will result in a zero from bitwise and so we can assume if the bit at bit_index was set it will result in a single one at the exact index in the result of the bitwise and. Furthermore any non-zero value in C is interpreted as true so we used this to determine the return value of the function. That is if the statement:

- byte & (1 << bit_index)

results in a non-zero value we return 1 otherwise we return 0.

Part 2 : References

https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/