2024SP - OPER SYSTEM DESIGN 16:198:518:01

PROJECT-1 REPORT

Members: Krishna Sathyamurthy ks2025, Abhinav Bharadwaj Sarathy ab2812

1. Signal Handler and Stacks

1. What are the contents in the stack? Feel free to describe your understanding.

Based on what we have learned so far, the stack is a set memory space that acquires memory from the top down to the middle of the memory allocated for the program being executed. The stack contains what we call a stack pointer, which usually consists of all the contents inside a function, starting with the arguments, local or static variables, and the return address. The stack pointer begins at the main function and grows as more functions are invoked. But the stack needs to keep track of where the current function, or the stack pointer on top of the stack, should return when it is done with its work. This is done with the help of a return address. The return address is pushed onto the stack whenever a new function is invoked. The act of pushing the address onto the stack makes the stack grow. Thus, each function has its own stack pointer, which contains the details of all the variables, arguments, and return address.

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

As mentioned above, when a function call is made, the stack grows. When a function call is made the program counter is incremented, and the return address is stored in the stack pointer of the new function.

The return address is usually stored close to the base pointer address (EBP or RBP). After identifying the memory address of EBP, we printed the memory of the next 32 levels in the stack. Later, we disassembled the main function to identify the address of the program counter for the next instruction. Finally, we compared the PC of the main instruction with the list of addresses printed using the GDB command to identify the location of the return address in the stack pointer.

As shown below, the next instruction in the main function is **0x56556245**, and it is stored at the address **0xffffc56c** in the stack.

```
edx            0x1                 1
ebx            0x56558fd0          1448447952
esp            0xffffc510          0xffffc510
ebp            0xffffc528          0xffffc528
esi            0xffffd414          -11244
edi            0xf7ffcb80          -134231168
eip            0x565561e2          0x565561e2 <signal_handle+37>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) x/32x $ebp
0xffffc528:     0xffffd348      0xf7fc4560      0x00000008      0x00000063
0xffffc538:     0x00000000      0x0000002b      0x0000002b      0xf7ffcb80
0xffffc548:     0xffffd414      0xffffd348      0xffffd330      0x56558fd0
0xffffc558:     0x00000000      0x00000000      0x00000005      0x00000000
0xffffc568:     0x00000000      0x56556245      0x00000023      0x00010286
0xffffc578:     0xffffd330      0x0000002b      0xffffc810      0x00000000
0xffffc588:     0x00000000      0xf7fce1a6      0xf7f82000      0x000097bc
0xffffc598:     0x00000003      0x00000032      0xffffffff      0x00000000
(gdb) disas main
Dump of assembler code for function main:
   0x565561fb <+0>:     lea    0x4(%esp),%ecx
   0x565561ff <+4>:     and    $0xfffffff0,%esp
   0x56556202 <+7>:     push   -0x4(%ecx)
   0x56556205 <+10>:    push   %ebp
   0x56556206 <+11>:    mov    %esp,%ebp
   0x56556208 <+13>:    push   %ebx
   0x56556209 <+14>:    push   %ecx
   0x5655620a <+15>:    sub    $0x10,%esp
   0x5655620d <+18>:    call   0x565560c0 <__x86.get_pc_thunk.bx>
   0x56556212 <+23>:    add    $0x2dbe,%ebx
   0x56556218 <+29>:    movl   $0x5,-0x14(%ebp)
   0x5655621f <+36>:    movl   $0x0,-0x10(%ebp)
   0x56556226 <+43>:    movl   $0x4,-0xc(%ebp)
   0x5655622d <+50>:    sub    $0x8,%esp
   0x56556230 <+53>:    lea    -0x2e13(%ebx),%eax
   0x56556236 <+59>:    push   %eax
   0x56556237 <+60>:    push   $0x8
   0x56556239 <+62>:    call   0x56556060 <signal@plt>
   0x5655623e <+67>:    add    $0x10,%esp
   0x56556241 <+70>:    mov    -0x14(%ebp),%eax
   0x56556244 <+73>:    cltd
   0x56556245 <+74>:    idivl  -0x10(%ebp)
   0x56556248 <+77>:    mov    %eax,-0xc(%ebp)
   0x5655624b <+80>:    sub    $0x8,%esp
   0x5655624e <+83>:    push   -0xc(%ebp)
   0x56556251 <+86>:    lea    -0x1fb6(%ebx),%eax
   0x56556257 <+92>:    push   %eax
   0x56556258 <+93>:    call   0x56556050 <printf@plt>
   0x5655625d <+98>:    add    $0x10,%esp
   0x56556260 <+101>:   mov    $0x0,%eax
   0x56556265 <+106>:   lea    -0x8(%ebp),%esp
   0x56556268 <+109>:   pop    %ecx
```

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

To get the desired result, I had to modify the return address to **0x5655624b**, i.e., increment the current return address by **6** bytes. To achieve this, we first found the address of the **signal_no (0xffffc530)** by using the command `x/x &signal_no` (**&** since signal_no is not a pointer). By doing a little bit of math, we identify that we need to increment the **signal_no** address by 15 ((**0xffffc56c-0xffffc530)/4**) to reach the return address. Finally, we increment the value stored in the return address.

We will get a result, even if we increment the return address by 3 bytes. But doing that will make the variable z hold the value 5, due to the instruction *mov %eax,-0xc(%ebp)*. -0xc(%ebp) points to the memory address of z and %eax holds the value 5, thereby modifying the value of variable z. Thus, increasing the return address by 6 bytes gives us the desired result.

References :

- Smashing the Stack for Fun and Profit by Aleph One: https://insecure.org/stf/smashstack.html
- How to look at the stack with gdb: https://jvns.ca/blog/2021/05/17/how-to-look-at-the-stack-in-gdb/

2. BIT MANIPULATION

2.1.1 Locating first set order bit to the left of LSB (Least Significant Bit)

The program finds the index of the first set bit with respect to the least significant bit in a number. Going into the logical approach, the function `first_set_bit()` takes in the argument num. Since we need to find the bit position wrt the LSB, we are doing a right shift by 1 over the num and re-assign. This would get us rid of the LSB letting us proceed further. Now to find the first set bit, we are computing the bitwise AND between the num and its 2's complement (Refer Fig A).

In simple terms, 2's complement of a number is 1's complement plus 1. This leaves us to conclude, for any number and its 2's complement are going to have the same first set bit. By doing a bitwise AND operation between the both, we can arrive at a value where 1 is set only at the first set bit. To determine the position, we take the log2 of this value, and add one to negate the zero-indexing.

```
/*
 * Function 1: FIND FIRST SET (FROM LSB) BIT
 */
static unsigned int first_set_bit(unsigned int num)
{
    //Implement your code here
    if (num == 0)
        return 0;
    num >>= 1;
    int res = log2(num & -num) + 1;
    return res;
}
```

Fig A. Code Snippet of first_set_bit()

2.1.2 Setting and Getting bits at at specific index

The first part of the question requires setting a bit at a specific index. We are given a char pointer bitmap which points to a dynamically allocated memory of 4 bytes in heap. Initially the four-byte memory is set with zeros in the main(), the function set_bit_at_index is then executed, which sets the bit at the given index (Refer Fig B).

Function: `set_bit_at_index`
Arguments: `char *bitmap, int index (position where the bit needs to be set)`
Return: `void`

The function begins by finding the `byte_offset`. Significantly, with this we could identify the block of array where the bit needs to be updated. Further, we call the method `bit_masking()`. It takes the `index`, and creates a bit mask; This is done by left shifting a zero order set bit by an offset (Ref Fig C.). The offset is evaluated to find the bit position in the 8-bit(1 byte) width by using a mod operation. For eg, let's say the index=17, the byte_offset will be 2, i.e. the bit belongs to the third memory block; And the offset, `bit_position` namely, will be 1. One thing to note, we are using `uint8_t`, an unsigned 8 bit int to set the mask; This could support masking logic across any OS architecture. A native signed int won't defy our logic as such, however since we know that our mask only requires 8-bit, so we are being considerate. Back to the rundown, the result of the mask function after left shifting by 1 will be 2 (10 in binary).

To set the bit, we utilize the mask value and do a bitwise OR operation with the bitmap block (bitmap[i]). By doing OR operation with the mask(1XXX), it is going to set the index with 1 no matter which bit was stored earlier.

```
/*
 * Function 2: SETTING A BIT AT AN INDEX
 * Function to set a bit at "index" bitmap
 */
static void set_bit_at_index(char *bitmap, int index)
{
    //Implement your code here
    int byte_offset = index/8;
    bitmap[byte_offset] |= bit_masking(index);
}
```

Fig B. Snippet of set_bit_at_index()

```
int bit_masking(int index)
{
    int bit_position = index % 8;

    // Create a mask with 1 at the desired bit position
    uint8_t mask = 1 << bit_position;
    return mask;
}
```

Fig C: Code snippet of the masking function

The second part is implemented quite similarly to the first, where we will come across bit masking and offset logic to get the bit at a given index. We have,

Function: `get_bit_at_index`
Arguments: `char *bitmap, int index (position to get the bit)`
Return: `int`

The logic begins by taking the same hierarchy approach for memory. We find the `byte_offset` by doing a division eight operation over the index, corresponding to byte in memory. Then we get the bit at the index position, by doing an bitwise AND operation between the bitmap block and the mask. The AND operation with a mask will return:

1. The mask value(int), if the bit at index is set. For example, the mask and the bitmap block are 1000 and 10101000 in bits correspondingly. Doing bitwise AND, we get 00001000 (8 in decimal).

2. Zero, if the bit at index is zero.

Based on these two cases, we return 1 and 0 correspondingly to the main function.

```
/*
 * Function 3: GETTING A BIT AT AN INDEX
 * Function to get a bit at "index"
 */
static int get_bit_at_index(char *bitmap, int index)
{
    //Get to the location in the character bitmap array
    //Implement your code here
    int byte_offset = index/8;
    return (bitmap[byte_offset] & bit_masking(index)) ? 1 : 0;
}
```

Fig D. Code snippet of get_bit_at_index()

References:

- https://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm
- https://www.programiz.com/c-programming/bitwise-operators