CS416 - Project 1

Sami Munir

February 12, 2024

1 Part I: Signal Handler and Stacks

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

The contents of the stack include the instruction pointer which points to the next instruction, the argument list to the program, locals, and saved registers. I found that utilizing GDB to track the stack was very helpful along with monitoring the ESP and EIP registers.

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

The program counter is stored within EIP. I used GDB to locate the PC by analyzing and tracking the values of the various registers. I learnt and knew that ESP was the stack pointer to the top of the stack frame. EBP was the base pointer to the stack frame. I then concluded that EIP was the program counter which was keeping a track of which instruction to execute using the [info line #] command and matching memory addresses. The PC was changing every time I moved to the next instruction in GDB.

1.3 What were the changes to get the desired result?

To get the desired result, I had to determine the length of the bad instruction. I did so by using GDB to create a break point at $z = x/y$ and used [info line #] to determine the starting and ending addresses of this instruction. I then

determined the return address by examining ESP and then computing the difference between the address of *signalno* and the return address at ESP. I was then able to manipulate the program counter of the main stack frame to skip the bad instruction and continue execution of code.

2 Part II: Bit Manipulation

2.1 function firstSetBit()

This function finds the index of the first set bit (from LSB) in an unsigned integer *num*.
It first checks if *num* is equal to 0. If it is, it returns 0 since there are no set bits.
It then checks the lowest 16 bits of *num* (using a bitmask of 0x0000FFFF). If all 16 bits are 0, it increments the position by 16 and shifts *num* by 16 bits.
It repeats this process for the next 8 bits, 4 bits, and 2 bits, and finally 1 bit, each time checking if all bits in the current range are 0 and updating the position accordingly.
Finally, it returns the position of the first set bit.

2.2 function setBitAtIndex()

To set a bit at a specific index, we first need to determine the byte index and bit index within that byte.
The byte index is obtained by dividing the bit index by 8 (since there are 8 bits in a byte). This gives us the byte containing the bit we want to set.
The bit index within the byte is the remainder when the bit index is divided by 8.
To set the bit, we use a bitwise OR operation between the byte at the byte index in the *bitmap* array and a bitmask. The bitmask has a 1 in the position corresponding to the bit index within the byte and 0s elsewhere.
After applying the bitwise OR operation, the bit at the specified index is set to 1 in the *bitmap* array.

2.3 function getBitAtIndex()

We first need to determine the byte index and the bit index within the byte. The byte index is obtained by dividing the bit index by 8.
The bit index within the byte is $bit_index = bit_index\%8$.
To extract the bit value, we perform a bitwise AND operation between the byte at the byte index in the *bitmap* array and a bitmask. The bitmask has a 1 only in the bit index position within the byte and 0s elsewhere.
The result of the bitwise AND operation will be non-zero if the bit at the specified index is set, and zero if not set.
Return this result as the value of bit at the specified index in the *bitmap* array.

3 Part III: pThread Basics

NO REPORT REQUIRED.