Group Members: Ritwika Das and Siya Vyas
NetIDs: rd935, sv694
CS 416

1.4 Report

- What are the contents in the stack? Feel free to describe your understanding.
    - Stack is a data structure that inserts new elements on the top of the stack, and can only be removed by the order of what was pushed in last.
    - You can push and pop elements into the stack, in which all of these operations work on a O(1) time complexity.
    - Local variables, return addresses, function parameters, and other temporary data is stored in the stack.
- Where is the program counter, and how did you use GDB to locate the PC?
    - The program counter can be found in the CPU, in which it is used to set the memory address where the program's execution begins.
    - In GDB, the program counter is usually referred to as $pc or $rip, in which the compiler will then list off the value of the program counter at the memory address.
- What were the changes to get the desired result?
    - The changes that needed to be made to the program counter, was that there needed to be an offset on it in order to have the program skip the line where the exception is occurring. So by going into signal_handler, the program counter is being added by an offset, which then goes back to main and allows the program to skip the exception fault line.

2.2 Report

In this code, there are three main functions: finding the location of the leftmost set bit, setting a bit at the specified index, and getting the value of the bit at that specified index.

- first_set_bit:
    - We checked if input number, num, is 0 and in that case, we return a 0.
    - return log2(num & -num);
        - We use log2() because it computes the log base 2 of a number, so we can determine the position of the leftmost set bit.
        - The -num operation is the same as taking the two's complement of num, which flips all the bits and adds 1.
        - We used the bitwise AND (&) between num and -num so that we end up with a binary number that has only the rightmost set bit of num set to 1, and all other bits are 0. This is because the AND operation between corresponding bits is 1 only when both bits are 1.
- set_bit_at_index:

    - We calculated the byte index by dividing the index by 8, so that we know which byte in the array holds the bit we want to set. Then, we calculated the bit index within the byte by taking the remainder when dividing the index by 8. This gives us the position of the bit within the byte.
    - (1 << bit_index) :
        - << : is the bitwise left shift operator
            - (1 << bit_index) : represents shifting the binary representation of the number 1 by bit_index positions to the left
    - bitmap[byte_index] | (1 << bit_index) :
        - | : is the bitwise OR
            - bitmap[byte_index] represents the byte in the bitmap array at the specified byte_index
            - So when we use the OR, what we are doing is that if the bit at bit_index in bitmap[byte_index] is already 1, the result will remain 1 because 1 OR 1 is 1 and if the bit at bit_index in bitmap[byte_index] is 0, it will be set to 1 because 0 OR 1 is 1.
- get_bit_at_index:

    - >> : is the bitwise right shift operator
        - We perform a bitwise right shift (>>) operation on the byte in the bitmap array by the calculated bit index. This moves the bit of interest to the least significant bit (LSB) position.
        - We then used a bitwise AND (&) operation with the result of the shift operation and the number 1 so that it isolates the LSB and takes the value of the bit at the calculated bit index within the byte.
        - If the extracted value is 1, it means the bit at the specified index is set. If it is 0, the bit is not set.