Mourya Vulupala (MV638)
Naman Bajaj (NB726)
CS416 – Operating Systems
10/3/23

Part 1 Report

In terms of our understanding, the stack (from top to bottom) was comprised of the local variables, and then the return address of the function, and before that you are able to see the pushed parameters of the function (*QUESTION/PART 1* – contents in the stack). Breaking this down, the local variables included the 'stack_ptr' variable that we had made within the signal_handle function to use as a pointer while navigating and modifying the stack. The return address tells the function which assembly instruction/address to return to once it is done executing. Finally the parameters are pushed upon the stack before a new stack frame gets created for the function being entered.

Since we did not directly enter the signal_handle function, finding the PC became a little bit confusing but not difficult. Using a bunch of breakpoints, we were able to 'backtrace' within GDB to see that there were 3 stackframes (one for main, one for the segmentation fault, and one for the signal_handle function). We then realized that the immediate return address found before the signal_handle stack frame would not take us back to main but to an internal function dealing with the segmentation error. We then were able to use "x/20xw $sp" to display items in the stack, at the current frame (signal_handle) and the previous frame (when segmentation fault occured). Our guess was right, the stack frame of signal_handle only contained the return address of another internal function, and that internal function (which was located earlier on the stack because it was called from main) contained the return address back to the line causing the segmentation fault in main. We verified this by using 'layout split'/'layout asm' to see that this value falls within the scope of the main function. Knowing this, we were able to count the number of 4-byte (or integer length) words between the signal number param (which we directly have the address of) and the proper return address that we discussed earlier. We now have the location of the PC (*QUESTION/PART 2* – where is the PC and how did we use GDB to locate the PC).

To find the length of the bad instruction that we needed to skip, we were able to 'disassemble main' within GDB. We were able to set a breakpoint at the faulty instruction line and GDB would give us a debug statement with the address, finding the address immediately after and adjacent to this instruction we were able to calculate the distance. This was the value that you add to the PC so that it successfully skips the faulty instruction upon its return (*QUESTION/PART 3* – changes necessary to get desired result).

Part 2 Report

For the function get_top_bits, we first wanted to make sure that the number of bits we wanted to extract was valid, so that was our first if check. This would prevent the program from crashing if the num_bits value was larger than the size of unsigned int or was negative. Then we created a shift variable that used the size of an unsigned int and the number of bits we wanted. We would then use this shift to right bitwise shift value. This effectively gets us the leftmost num_bits bits.

For the function set_bit_at_index, we first extract the specific byte index and the bit index within that byte that we want to set. We then add byte_index to the bitmap array argument to extract the byte from the array that we want to manipulate. We then create a mask where the first index is 1, followed by a bit_index number of 0s. Then we do a bitwise OR with byte and the mask. This means the bit at bit_index will be change from a 0 to 1 and everything else will remain the same, since we are doing or (1 or 0 is just 1).

For the function get_bit_at_index, we do the same thing as set_bit_at_index, where we get the byte and bit location, the value itself, and create a mask. Then we do a bitwise AND with the mask and the byte. Since the mask has a 1 only at the index we want to check, and 0s everywhere else, the value of the whole thing will be 0 if the value at the index is 0, and 1 otherwise. We use an if to check if the whole thing is 0 or 1 and return accordingly.

Part 3 Report
N/A