CS 518 Assignment 1

Abhinay Reddy Vongur (av730)
Anmol Sharma (as3593)

Contents of the Stack

`info frame` command in gdb was used to examine the contents of the stack. Below is a snapshot of the output from the command.

```
Stack level 0, frame at 0xffffcab0:
 eip = 0x56556223 in segment_fault_handler (signal.c:11); saved eip = 0xf7fd0b60
 called by frame at 0xffffd150
 source language c.
 Arglist at 0xffffca8c, args: signum=11
 Locals at 0xffffca8c, Previous frame's sp is 0xffffcab0
 Saved registers:
  ebx at 0xffffcaa4, ebp at 0xffffcaa8, eip at 0xffffcaac
```

It was observed that the stack consisted of the following information about the current process.

- eip - Program counter of the current function, values is the address of the instruction being executed right now.
- saved eip - This is the location where control will return after execution of the current function. It is the saved program counter value of the caller function.
- Caller info - The location of the caller function stack frame.
- Argument list - List of arguments passed to the function.
- Local variables - Variables in the scope of the function.
- Saved registers - Values of registers that are saved in the stack, used to restore the register values after the completion of the current function. These register values are of the previous function that called this function.
    - ebp - This is the location where the ebp of the caller function is saved. ebp represents the base address of the stack. It is used as an offset while accessing variables.
    - eip - This is the location where the saved eip(return address) is at.

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

- Using `info frame` in gdb, the stack was examined when the control is in the segment_fault_handler function. After observing the stack, it was observed that the saved eip holds the return address for the function.
- Further analysis of the stack determined that the eip location in the saved registers is where the saved eip is located at.
- It was found that in the case of an exception there was another stack frame between the fault handler and the main function while in the case of a normal function call there were only two stacks one for the main and the other for the called function.

- This stack frame is of the OS function that catches the fault and calls the corresponding fault handler. The actual return address of the main function is saved in this stack frame at the address 0xffffcaec.

- This observation was confirmed by verifying the locations of the instructions in the main function using disassemble utility in gdb.

What were the changes to get the desired result?

- A pointer to the signum argument received by the fault handler function was used to manipulate the values in the stack.
- After finding the location of the return address, the pointer to the argument was offset to point to the location of the return address. It was found that the return address is saved at 0xffffcaec. The location of the argument signum is 0xffffcab0.
- Since signum is of type int, adding 15 to the pointer incremented its location by 60 bytes making it 0xffffcaec.
- Now, the disassemble command was used to check the instructions of the main function and their addresses in memory. This information was used to determine the new return address and also check where the current return address is pointing to.
- The return address saved in the stack was 0x565562a5. It was determined that the instruction we want to skip to is at 0x565562aa. So the return address value was incremented by 5 to make the process skip the faulty instruction. This was achieved by dereferencing the pointer created above and incrementing by 5.
- After determining this information, the offset required to change the return address was calculated and the value at the pointer location was offset by the same.
- This resulted in the processor skipping the line of code that caused the fault and completing the execution successfully.

The code for handling segmentation faults was compiled using the **-m32** flag to compile as a **32-bit object**. This was tested on the machine **ilabu3 (ilabu3.cs.rutgers.edu)**.

Compilation flags used for the second question: gcc -o <object_name> <file_name> -lpthread