Lab 4

Objective:

- Calling functions in x86-64 assembly language and managing the stack, i.e., their stack frames.

  The purpose of this lab is to examine the x86-64 function call mechanisms in action. Given a program, i.e., a collection of functions that call each other, you will identify variable allocation — both in registers and on the stack (i.e., stack frame) — and draw the resulting stack as well as the register table as this program executes.

Part 1: Getting Started

- Login to the CSIL machines in the Linux environment, download and unzip `Lab4-files.zip` into `sfuhome/CMPT295`, and change directory to `Lab4`. A directory listing should reveal a `makefile` and the source files `main.c`, `p1.c` and `p2.c`.
- Start by opening `main.c` in your favourite editor. The program contains three variable declarations: a 40-byte `char` array and two 4-byte `int`s. It displays the original values of two of these variables, and then calls the function `proc1()`, which is defined in `p1.c`. The parameters are passed using *call by reference*, which means that each parameter is a pointer to the data, rather than [a copy of] the data itself (*call by value*). Call by reference allows the callee to modify the data. This is actually what is going to happen: the values of all three variables will change, which the subsequent calls to `printf()` and `puts()` will confirm.
- Now open `p1.c`. It uses the two local variables `v` and `t`, calls `proc2()`, re-computes the values of `s`, `a` and `b`, and returns.
- Make sure you understand what the code contained in these three files (`main.c`, `p1.c` and `p2.c`) does.
- On the command line, run `make` and then execute the program. Make sure you agree with its results.

Part 2: `main()` calling `proc1()`

Your task in this part of the lab is to draw the stack diagram (or stack segment) that is produced as `main()` executes and calls `proc1()`. In our Lectures 15, 18 and 19, we have drawn such stack diagrams. The drawing of the stack diagram is done by hand tracing the execution of code. Therefore, in this part of the lab, you draw the stack diagram produced by the execution of `main()` function until it calls `proc1()` (including the `call` instruction itself) by hand tracing this code.

The following instructions will help you with this task:

- Open `main.s`, the caller. This file contains a lots of directive statements and assembly code instructions. The first assembly code instruction of `main` we shall start with is `pushq %rbp` at line 18. The effect of this instruction on the stack has already been illustrated (drawn) on our stack diagram on the last page of this lab. Have a look! As you can see, the content of the register `%rbp` has already been pushed at the top of the stack. Since we do not know the value stored in this register, we shall simply use the name of the register to indicate this value.

  Why is `main` pushing the content of `%rbp` on the stack? You may not be able to answer this question just yet. As you step through the assembly instructions (hand tracing them), it will become clear to you. When it becomes clear to you, indicate your answer under the column named **Purpose** on the right-hand side.

- Over the next four instructions (line 21 to 24), `main` is preparing the parameters to the function `printf(…)` which it calls on line 33.

- At line 25, 64 is subtracted from the current stack pointer `%rsp`. This has the effect of reserving 64 bytes of memory space for `main()`'s local variables on the stack. So, on the stack diagram, mark these 64 bytes, or 8 quad words, as "space for `main()`'s local variables" under the column named **Purpose**.

- The effect of subtracting 64 bytes from the stack pointer `%rsp` (from the memory address the stack pointer `%rsp` contains) is reflected on the stack diagram by moving the label "`%rsp`". This has already been done for you, but contrary to what we have seen so far in our lectures, this location has been labelled "`%rsp + ` 0_{10}" (instead of simply this "`%rsp`"). Why? Let's explain:

  - Addresses within the "space for `main()`'s local variables" are expressed, on this stack diagram, relative to the current value (or location) of the stack pointer `%rsp`. This is to say that the current location of the stack pointer `%rsp` (i.e., the top of the stack) is 0_{10} (0 byte) away from its current location on the stack.

  - This format "`%rsp + ` 0_{10}" is composed of a **base register** (here: `%rsp`) to which a **displacement** (a factor of 8 bytes) is added. So, the next quad word will have the memory address `%rsp + ` 8_{10} (displacement of 8 bytes from the current stack pointer `%rsp`), and the next will have `%rsp + ` 16_{10} (displacement of 16 bytes – 2 x 8 bytes - from the current stack pointer `%rsp`), and so on.

  - Fill in all the other labels (i.e., "`%rsp` + displacement value") to the left of each stack cell (or row). There are already three of them on the diagram (`%rsp + ` 0_{10}, `%rsp + ` 8_{10}, `%rsp + ` 16_{10}).

- On lines 27 and 28 of `main( )`, the instruction pair

  ```
  movq %fs:40, %rax
  movq %rax, 56(%rsp)
  ```

  loads the ***canary value*** to detect ***buffer overruns***. Therefore, write "canary value", in the stack at `%rsp + 56`$_{10}$.

  We will talk about ***canary value*** and ***buffer overruns*** in our lectures very soon.

- Are the values of the local variables `x` and `y` stored on the stack? To answer this question, search for the appropriate two instructions by having a look at the next few assembly instructions (from line 29 to line 32). Where on the stack are they stored? Remember, `x` is initialized to the value `6` and `y` to the value `9`. Place these variables (names and values) on the stack diagram in the **correct** location.

- After calling `printf()` at line 33, the three instructions that follow are the setup for calling function `proc1()`. Remember that the x86-64 function call protocol dictates that the first argument be placed in `%rdi`, the second in `%rsi` and the third in `%rdx`. Using this information and the instruction at line 30, deduce the location of `char buf[40]`, and add it to your stack diagram. Recall: The `leaq` instruction computes an effective address, but does not dereference it. Thus each of `%rdx`, `%rsi` and `%rdi` contain addresses, i.e., they are pointers. Here is yet another example of how we create pointers using the stack.

- The call to `proc1()`, at line 37, pushes the return address (address of assembly instruction at line 38), thus ending the stack frame for the caller `main()`. Add this return address to your stack diagram. For the purpose of this lab, simply write in the appropriate stack cell (or row) the words "return address for `main` at line 38".

- As we saw in our lecture, a stack frame is the section of the stack segment utilized by a function as it executes. To clearly indicate the extend of the stack frame for the function `main()`, i.e., where it ends, draw a thick line just "below" this stack frame. Also, under the column named **Purpose** on the right-hand side, label this stack frame using the name of its associated function.

- Make sure you update the **register table** as you hand trace the code, listing all the registers used so far and their values.

- At this point in our Lab 4, you may wish to have the TA verify your stack diagram.

Part 3: `proc1()` calling `proc2()` and `proc2()` itself.

Your task for this part of the lab is to draw the stack frame for `proc1()` until it calls `proc2()` and the stack frame for `proc2()` if there is one allocated, i.e., if `proc2()` actually uses the stack.

Important: Your final stack diagram and register table should reflect the execution of the `main()`, `proc1()` and `proc2()` functions up until `proc2()` has reached its `ret` instruction, **but has not yet executed it**.

Bottom line: we are not hand tracing all the code of `main()` and `proc1()`!

The following instructions will help you with this task:

- Open `p1.s`, and have a look at `proc1()`. Within its first few lines (line 15 to 26), it saves four registers on the stack (on its stack frame). Which registers are they? Why are they being saved? Are they ***callee saved*** or ***caller saved*** registers? Add them to your stack diagram and add a note under the column **Purpose** describing what `proc1( )` is doing.
- Where are the variables for `proc1()` stored? Hand trace the code on line 21, line 25, line 29 and line 32 to line 34, just before the call to `proc2()` at line 35, and deduce which registers hold which values. Update the register table as you go. In other words, which register is holding each of the following values:
    - char *s (the pointer)
    - int *a (the pointer)
    - *a (the dereferenced pointer)
    - int *b (the pointer)
    - *b (the dereferenced pointer)
    - v (the integer result of `proc2()`)

  The variable **t** wasn't actually necessary. The compiler optimized it away.
- Have a look at `proc2()` in `p2.s` to determine which parameters are passed, which registers are saved, and where these functions' local variables are stored. Add the relevant information to your stack diagram and the register table. Do not forget about the locations of `proc2()`'s local variables `m` and `n`.
- Draw a thick line just "below" each stack frame to clearly indicate where it ends. Also, under the column named **Purpose** on the right-hand side, label this stack frame using the name of its associated function.
- Make sure you update the **register table** as you hand trace the code, listing all the registers used so far and their values.
- Again, at this point in our Lab 4, you may wish to have the TA verify your stack diagram.

Challenge

There are two things you should experiment with, both tied to the size of the string `buf`.

1. If you reduce the size of the string buffer to say, `buf[39]` and re-make the code, you'll notice that `main.s` did not change This is probably because of word boundary optimizations: it is more efficient for the machine to access the canary value (and perhaps other 8-byte values as well) if these values do not cross a word boundary, i.e., if they are stored in stack cells with addresses that are multiple of 8. Thus, the effective size of `buf` (and other oddly sized data types) will usually get rounded up to the nearest multiple of 8.

2. But here is the really strange thing: if you reduce the size of the `buf` to 32, there will still be no change in `main.s`. Try it! This has to do with stack optimization. For some portion of the instruction set that deals with streaming applications (see the Aside at page 276 in our textbook), it is critical that the stack pointer be a multiple of 16. This is called *stack alignment*, and it is not a big deal for the applications we are pursuing. However, because the compiler pays attention to stack alignment issues, you will not see any effect on `main.s` until you reduce the size of `buf` to < 25.

3. Try it by reducing the size of `buf` to 24 and re-making the code. Then have a look at `main.s`. Notice that it now allocates a stack "space" of 48 bytes instead of the original 64 bytes, i.e., a difference of 16!

   Lastly, re-run your modified program and observe an unusual program exception. Hum… What do you suppose happened here? Again, we will talk about this interesting situation in our lectures very soon. Stay tuned!

Last name:

First name:

Student number:

base + displacement	M[ ] Stack Segment	Purpose
	%rbp	
$\%rsp + 16_{10}$ ->		
$\%rsp + 8_{10}$ ->		
$\%rsp + 0_{10}$ ->		

Register Table :
