Family Name:______________________________

Given Names:______________________________

ID#:______________________________

CMPT 295 Lab 4 (0.5%)

The purpose of this lab is to examine the function call protocol in action. Given a collection of functions that call each other, you will identify variable allocation — both in registers and on the stack — and draw the call stack for the full program, stack frame by stack frame.

Part 1: Getting Started

- Login to the CSIL machines in the Linux environment, download and unpack the care package into `sfuhome`, and change directories into `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: two 4-byte `int`s and a 40-byte `char` buffer.
- Follow the execution of the mainline routine. It displays the original values, 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 reference allows the callee to modify the data, which actually happens in this case: the values of all three variables change, which the subsequent calls to `printf()` and `puts()` will indicate.
- Now open `p1.c`. It uses the two local variables `v` and `t`, re-computes the values of `s`, `a` and `b`, and returns.
- On the command line, run `make` and then run the program.

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

Your task for the first part of this lab is to draw the stack frame for `main()` calling `proc1()`.

- Open `main.s`, the *caller*. The first line subtracts 72 from the current stack pointer, thereby reserving 72 bytes of variable space for `main()`. On the stack frame diagram (last page of this lab) you can mark the top 72 bytes, or 9 quad words, as local variable space for `main()`.

- Addresses within the local variable space are expressed *relative* to the current value of the stack pointer. The top of the stack has address `rsp` + 0, but the next quad word has address `rsp` + 8, and the next has `rsp` + 16, and so on. Label all of the base + displacement values to the left of each box.

- 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 box for `rsp` + 56.

- Where are the values for the variables `x` and `y` stored? Place their variable names in the diagram in the correct location.

- After calling `printf()`, the three `leaq` instructions that follow are the setup for calling function `proc1()`. Remember that the function call protocol dictates that the first argument goes in `%rdi`, the second in `%rsi` and the third in `%rdx`. Using this information, deduce the location of `char buf[40]`, and add it to your diagram.

  Recall: The `leaq` instruction computes an effective address, but does not dereference it. Thus `%rdx`, `%rsi` and `%rdi` contain addresses, i.e., they are pointers.

- The `call` to `proc1()` pushes the return address, thus ending the caller's stack frame. Add the return address for `main()` to your stack diagram.

- Switching views to `p1.s`, the *callee*, the first six lines save some registers on the stack. Which registers are saved? Add them to your diagram.

- The next thing that happens is a deduction in the stack pointer by 24, which may seem like a confusing amount to allocate because for `proc1()`, no local variables are stored on the stack. So, where are they stored? Read through the code to deduce which registers hold which values. Make a list of the locations of each of:

  – `char *s` (the pointer)
  – `int *a` (the pointer)
  – `*a` (the dereferenced pointer)
  – `int *b` (the pointer)
  – `*b` (the dereferenced pointer)
  – `*b` − 2 (the dereferenced pointer minus 2)
  – `v` (the integer result of `proc2()`)

  Note: The variable `t` wasn't actually necessary. The compiler optimized it away.

- Call your TA to view your stack diagram before proceeding to Part 3.

Part 3: `proc1()` *calling* `proc2()`

- Your final job is to complete your diagram by adding the stack frames for `proc1()` calling `proc2()`. You will need to read the code for `p1.s` and `p2.s` to determine where parameters are passed, which registers are saved, and where the local variables are stored.

  Add the relevant information to your stack diagram, as well as the locations of the local variables `m` and `n`.

- This is the last step of Lab 4. Call the TA to your terminal to check your stack diagram. After verifying your work, the TA will then collect this sheet.

Thinking Ahead

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

If you reduce the size of the string to, say, `buf[39]` and re-`make` the code, you'll notice that `main.s` didn't change. This is probably because of word boundary optimizations: it is more efficient for the machine to access the canary value (or perhaps other 8-byte datatypes as well) if they don't cross a word boundary. Thus, the effective size of `buf` (and other oddly sized datatypes) will usually get rounded up to the nearest multiple of 8.

But here's the really strange thing: if you reduce the size of `buf` to 32, there will still be no change. (Try it.) This has to do with stack optimization. For some portion of the instruction set that deals with streaming applications, 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 to `buf[24]`. Try it and you will see an allocation of 56 bytes instead of the original 72.

You can re-run the program at this point to see an unusual program exception. What do you suppose happened here?

base + displacement	Stack Variables	Purpose
rsp + 64	Local variables main()	
rsp + 56		
rsp + 48		
rsp + 40		
rsp + 32		
rsp + 24		
`rsp` + 16		
`rsp` + 8		
`rsp` + 0		

Register Variable Map for `proc1()`:

`char *s` —

`int *a` —

`*a` —

`int *b` —

`*b` —

`*b` − 2 —

`v` —

Register Variable Map for `proc2()`:

`m` —

`n` —