Lab 4

Objective:

- Calling functions in x86-64 assembly language and managing the stack (and stack frames)

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.

Submission – Participation Activity #5:

- At the end of this lab session, hand-in the last page of this lab, i.e., the stack diagram once you have completed it.
- Make sure you write your name and student number on its top left corner (as the top right corner seems to be quite “busy”).

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 main function. 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 value). 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, calls proc2(), 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 this part of the lab is to draw the stack frame for main() calling proc1(). As we saw in class, a stack frame is the section of the stack occupied by a function when it executes.
• Open main.s, the caller. This file contains a lot of directive statements and assembly code instructions. The first assembly code instruction of main is `pushq %rbx` at line 17. The effect of this instruction on the stack has already been illustrated on our stack diagram on the last page of this lab. Have a look! As you can see, the content of the register rbx 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 rbx on the stack? You may not be able to answer this question just yet. As you step through the assembly instructions, it will become clear to you. At this point, indicate your answer under the column named Purpose.

• Over the next four instructions (line 20 to 23), main is preparing the parameters to the function `printf(...)` which it calls on line 32.

• At line 24, 64 is subtracted from the current stack pointer. This has the effect of reserving 64 bytes of variable space for main() on the stack. On the stack diagram, mark the next 64 bytes, or 8 quad words, as “local variable space for main()” under the column named Purpose.

• Addresses within the “local variable space for main()” are expressed relative to the current value of the stack pointer (think of it as the base register). The top of the stack (remember where the top of the stack is located) has address `rsp + 0₁₀`, where 0 byte is the displacement made from the base register, i.e., the stack pointer, but the next quad word has address `rsp + 8₁₀` (displacement of 8 bytes from stack pointer), and the next has `rsp + 16₁₀` (displacement of 16 bytes from the stack pointer), and so on. Label all of the base + displacement values (i.e., “rsp + displacement value”) to the left of each stack cell. There are already three of them on the diagram (`rsp + 0₁₀, rsp + 8₁₀, rsp + 16₁₀`).

• On lines 26 and 27 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 + 56₁₀.`

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

• Where are the values for the variables x and y stored? Remember, x is initialized to the value 6 and y to the value 9. Place their variable names in the stack diagram in the correct location.

• After calling `printf()`, 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 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 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 36, pushes the return address (address of assembly instruction at line 37), thus ending the “local variable space for main()” (the caller). Add the return address for main() to your stack diagram. For the purpose of this lab, simply write in the appropriate stack cell the words “return address for main at line 37”.

- Now that proc1() is being called, update the register table found to the left of the stack diagram.

- Switching views to p1.s, the callee, within the first few lines, four registers are saved on the stack. Which registers are saved? Why are they saved? Add them to your stack diagram.

- Where are the variables for proc1() stored? Read through the code (line 20, line 27 to line 51 –skip the directive statements) to deduce which registers hold which values and update the register table as you go:
  - 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.

- At this point in the lab, you may wish to have the TA verify your stack diagram before proceeding to Part 3.

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

- Your final job, if you have not done it already, is to add to your diagram the complete “stack space” for proc1(), i.e., its stack frame, as well as the stack frame for proc2() (if proc2() uses the stack). 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 and the register table. Do not forget about the locations of the local variables m and n.
• Add the note under the column named **Purpose** to describe what `proc1()` is doing when it pushed 4 registers onto 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.** This is to say, your final stack diagram and register table must be at their fullest since functions have not yet started to execute their `ret` and their `popq` instructions.

• This is the last step of Lab 4. Again, you may wish to have the TA verify your stack diagram then. When all is well, hand in your stack diagram to the TA.

---

**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. Try it and you will see a stack allocation of 48 bytes instead of the original 64 in `main.s`, a difference of 16.

Re-run the program at this point to see an unusual program exception. What do you suppose happened here?
<table>
<thead>
<tr>
<th>base + displacement</th>
<th>Stack Variables</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rbx</td>
<td></td>
</tr>
</tbody>
</table>

Register Table:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rsp + 16\text{\textsubscript{10}}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rsp + 8\text{\textsubscript{10}}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rsp + 0\text{\textsubscript{10}}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last name:  
First name:  
Student number: