COMPUTER SYSTEMS
PROJECT 2
Assembly procedure calls

1 *x86* procedure calls

This project involves writing two procedures. Each uses the stack, although minimally; both require passing arguments, calculating something, and returning a result. While not a comprehensive experience with method writing, it will make you familiar with the form and capable of reading the assembly generated by a compiler.

Some things you are likely to need to know in order to write these procedures, building on material covered during lectures:

- **SP pre-call alignment**: The *stack pointer* (`rsp`) must be aligned on a double-word boundary before a CALL instruction is performed. That is to say, $rsp \ mod \ 16 \equiv 0$.

  If you need to align `rsp` just before a procedure call, you can simply *subtract* the needed value from this register, thus pushing *padding* (unused space) onto the top of the stack in order to align its top. Be sure to then *add* this same value back to `rsp` when the call has returned, restoring the top of the stack to its original position.

- **The CALL opcode**: Call a procedure. The use of this opcode looks like this:

  ```
  call  some_procedure
  ```

  When executed, it will do the following:

  1. *Push the return address:* Allocate a word-sized space on the stack by decrementing the stack pointer (`rsp`) by 8; and, copy the address of the next instruction (based on the current instruction pointer (`rip`) into this space (`[rsp]`).
  2. *Jump to the labeled address*.

- **SP post-call alignment**: Given the description above, the stack pointer will **not** be double-word aligned after the call. Therefore, all procedures must assume that, at the moment the procedure begins, `rsp` is word-aligned, but not double-word aligned.

- **The RET opcode**: Return from a procedure. It doesn't look like much...

  ```
  ret
  ```

  ...but it does a couple of important things:

  1. *Pop the return address:* Grab the return address stored at the top of the stack (`[rsp]`), then deallocate it (`rsp <- rsp + 8`).

2. *Jump to the return address*.

- **Passing arguments**: Arguments are passed into parameters first using six of the registers, and then (if there are more than six parameters) pushing additional arguments onto the stack. The registers are, in order:[1]

```
arg #:   0,   1,   2,   3,  4,  5
  reg: rdi, rsi, rdx, rcx, r8, r9
```

  It's a wacky order, but it is the standard for this instruction set architecture.

- **Returning a value**: The return value is placed in `rax`. Easy peasy.

- **Preserving registers**: There is a subset of registers that are *callee preserved*—that is, when a procedure is complete and returns, the calling procedure should be able to rely on the values in those registers being unchanged. Those registers are, in no particular order:

```
rbp, rbx, r12, r13, r14, r15
```

  If your procedure uses any one of these registers, then you must *preserve* its original value at the beginning of the procedure (by pushing its value onto the stack), and then you must *restore* that original value just before returning (by popping its value from the stack and back into the register).[2]

There are likely other things worth knowing, but these will, I hope, be helpful.

2 Getting started

1. Login to `remus/romulus` and open a terminal window.

2. Make a new directory for this class and project, and change into it:

```
$ mkdir -p systems/project-2
$ cd systems/project-2
```

3. Download the project's source code:

```
$ wget -nv -i https://bit.ly/COSC-171-project-2-source
```

4. Open a mostly familiar bit of assembly:

```
$ emacs neo-hello.asm &
```

[1] These are the registers used for integers and pointers; if floating point values are passed, there is another set of registers, `xmm0` to `vmm7`, for those. We won't worry about floating point values in this course.

[2] For this assignment, it is quite possible not to need to use any of these registers, thus avoiding this issue entirely.

3 A string-length procedure

The code that you will see should look mostly familiar. However, there are a few changes worth noting.

Starting with `main()`: The *C compiler*, `gcc` has two basic jobs: first, it translates *C* code into machine code;[3] and, then it *links* (with `ld`) that object code with *library code* to form an executable file. *Libraries* are collections of pre-written object code for procedures that a programmer can call upon.

Part of the standard *C* library code includes *stub code*—a pre-written `_start` quasi-procedure that initializes the stack, calls the procedure named `main()`, and when that procedure returns, performs the EXIT system call. That is how, just as with Java, `main()` is made the starting point of any *C* program.

We can leverage this behavior of `gcc` without writing any *C* code. Specifically, our assembly programs can begin with a `main` procedure instead of `_start`. We then don't have to worry about performing the EXIT system call. Better yet, *we will be able to call standard C procedures*. If we want to print something to the console, instead of performing a WRITE system call, we can call the *C* procedure `printf()`. Doing so will allow us to print not just static strings, but formatted strings into which numeric values are inserted.[4]

You will therefore notice that `neo-hello.asm` begins with `main`, and that the `main` procedure ends with a `ret` instruction.

Null-terminated strings: In the original `hello.asm`, we simply hand-calculated the length of the message to be written to the console and passed that value to the WRITE system call. However, the standard for assembly and *C* programs is to use *null-terminated character arrays* to represent strings. That is, a *string* is a sequence of characters that starts at some given address (i.e., a pointer marks its beginning) and ends at the *first zero-valued character*. Here, characters are byte-sized values, so the first *null character* (often written as `'\0'`) is the byte whose value is zero.

Notice, in the *data* section of our program, that the message is a string of characters, followed by the newline character (`10`) and *then* followed by the null character (`0`). That explicit zero value marks the end of the string. Without it, *C* functions won't know where the string ends.

Your assignment: Notice that there is a label, `string_length`, followed by no code. You must **write that procedure**. Specifically, this procedure has one parameter—a pointer to a string—and it returns one value—the length of that string. Write this procedure to count the number of bytes in the string, using the zero-valued byte as the marker of the string's end. If the procedure were declared in *C*, it would look like this:[5]

```
long string_length (char* string)
```

[3] Actually, it pre-processes the C code, then translates that into assembly, then assembles that into object code. For our purposes, we can just condense those steps.

[4] This first part of the assignment will use the old-fashioned WRITE system call, keeping things a little more familiar. The second part will use `printf()`.

[5] The *C* type `char*` denotes *a pointer to an array of characters*. More on that when we start using *C* itself.

When you have written this program, you can test it mostly in the normal way, but there is a change (given the above) in how it is linked. Specifically, using `gcc` instead of `ld` is a simple substitution:

```
$ nasm -felf64 -g neo-hello.asm
$ gcc -o neo-hello neo-hello.o
$ gdb neo-hello
[or]
$ ./neo-hello
```

4 An exponentiation procedure

Now open `exp.asm` in an *Emacs* window. Although there are again some differences from what you've seen (e.g., the actual use of the standard *C* procedure `printf()`), all of what is there has been addressed in previous work and in the foregoing sections.

Your assignment: Write the `exp()` procedure **recursively**. Assuming 64-bit integer parameters x and y, calculate and return x^y. Do not worry about error handling (e.g., negative values for y). You can rely on the following recursive definition of integer exponentiation:

$$x^y = \begin{cases} 1 & \text{if } y = 0 \\ x \times x^{y-1} & \text{if } y > 0 \end{cases} \tag{1}$$

The declaration of this procedure, in *C*, would look like so:

```
long exp (long x, long y)
```

5 How to submit your work

Submit your `neo-hello.asm` and `exp.asm` files using one of the two usual tools:

- **Web-based:** Visit the submission system web page.
- **Command-line based:** Use the `cssubmit` command at the shell prompt on `remus/romulus`.

This assignment is due on Wednesday, Sep-19, 11:59 pm.