Computer Systems
Project 2
Assembly procedure calls

1 *x86* procedure calls

This project involves procedure calls, each of which uses the stack, although minimally so. 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:

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

  Since the `call` instruction pushes one word onto the stack itself, you may need to *pad* the stack so that `rsp` will be aligned after the `call` completes. If you need to add such padding, you can simply *subtract* the needed value from `rsp`, thus pushing the unused space onto the top of the stack.

- **SP post-call alignment**: If you add padding to the stack to allow alignment after a `call`, then you need to remove this padding after the call returns. To do so, simply *add* the same value to `rsp`—the number of bytes of padding—that you subtracted before the `call`.

- **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).[1] The registers are, in order:[2]

  ```
  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).[3]

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

2 Getting started

1. **Login to the server:** Connect to the course server.

   ```
   $ ssh -Y yourusername@systems.aws.amherst.edu
   ```

2. **Login to GitLab:** From your browser, login to `https://gitlab.amherst.edu`

[1]Yes, this is different from what we described in class. This in the *Linux x86_64 calling convention*, whereas what we described in class was the *C declaration (cdecl) calling convention*. Both are used, and we are choosing here to use the one that is standard for the system on which we're coding our projects.

[2]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.

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

3. **Start a new project:** On the top toolbar of the GitLab window, click the little drop-down menu marked by a plus-sign. Select `New project`.

4. **Name and create the project:** Set the *Project name* to be `sysproj-2`, and leave the other default values. Click on the `Create project` button at the bottom.

5. **Clone the repository onto the course server:** Next, copy-and-paste the **first two commands** shown under *Create a new repository* by GitLab in your browser.

   ```
   $ git clone git@gitlab.amherst.edu:yourusername/sysproj-2.git
   Cloning into 'sysproj-2'...
   Enter passphrase for key '/home/yourusername/.ssh/id_rsa':
   warning: You appear to have cloned an empty repository.
   $ cd sysproj-2
   ```

6. **Download the source code:** After you download the files, use `ls -l` to list the directory and see what you have.

   ```
   $ wget -nv -i https://bit.ly/cosc-171-20F-p2
   $ ls -l
   ```

7. **Add the source code to the repository:**

   ```
   $ git add *
   ```

   Notice that **no news is good news**. This command generates no output when everything goes correctly.

8. **Commit and push the updated repository:** First, *commit* the changes—the addition of the `.asm` files. Then, *push* this update to GitLab.

   ```
     $ git commit -m "Starting code."
     $ git push
   ```

   You can go back to your browser and refresh the GitLab page. You should see the source code files appear as part of the repository there.

3 An exponentiation procedure

Now open `exp.asm` with a text editor (e.g., *Emacs*). This is a **completely written** example that we will go through during lab. It shows exponentiation implemented both *iteratively* and *recursively*. The *C* code for these functions would look something like this:

```
long exp_iterative (long x, long y) {
  long total = 1;
  while (y > 0) {
    total = total * x;
    y = y - 1;
  }
  return total;
}

long exp_recursive (long x, long y) {
  if (y == 0) return 1;
  return x * exp(x, y-1);
}
```

To try assemblying and running this code, do this:

```
$ make exp
$ ./exp
```

During lab, we will discuss the `make` command and its input, `Makefile`.

4 A string-length procedure

In order to work with the remaining parts of this assignment, we must address some changes to how we will write, assemble, and link our code.

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.

We will be **writing an assembly function that calculates the length of null-terminated strings**.

Starting with `main()`: The *C compiler*, `gcc` has two basic jobs: first, it translates *C* code into machine code; and, then it *links* (using `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. When `main()` returns, the program then ends.

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.

A tester program in C: By writing our string-length-calculating function using the standard Linux calling convention, we can write C code that calls on our assembly function. Doing so makes it easier to create test cases for our function.

The C source code in `tester.c` is a simple `main()` function that calls the `string_length` function that you wiill be writing, and then showing the results of its calculation. More on this later.

Your assignment, Part I: writing `string_length`: Open `string-length.asm`, and you will find the beginning of a function—the label, `string_length`. 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. Return the result in `eax`.

To test your code, use the C `tester` program, like so:

```
$ make tester
$ ./tester
```

Your assignment, Part II: calling `string_length`: Then open `neo-hello.asm`, in which you will see an incomplete `main()`. Specifically, where shown in the comments, you must write the steps to call `string_length()`. The result of your function call is then used by the existing code to perform a WRITE system call.

To test this code:

[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()`.

```
$ make neo-hello
$ ./neo-hello
```

5 How to submit your work

First, be sure that the most recent versions of your work are up-to-date on the GitLab server by performing an *add/commit/push* with `git`.

Then, go to GitLab with your browser. Navigate to the `sysproj-1` project. On the left side of the page, click on `Settings` to reveal a drop-down menu, from which you should select `Members`.

In this *Project members* window, under *Invite member*:

1. Under *Select members to invite*, enter `sfkaplan`. You will see me appear (*Prof. Scott Kaplan*) as a user; select me.
2. Under *Choose a role permission*, click the drop-down menu and select `Developer`.
3. Below, click the `Add to project` button.

This assignment is due on Sunday, Sep-20, 11:59 pm.