Lecture Exercise: Calling Conventions

In class, we discussed calling conventions for 32-bit x86. This exercise challenges you and your group to reverse engineer the 64 bit x86 calling conventions.

How to Work in a Group

The main rule I want each group to abide by is this:

> Your group should *not* proceed with additional implementation or debugging until *everyone* understands all steps and logic up till now.

In short, unanimity and shared understanding are required. I highly recommend that you should feel like you're in the "teacher seat" 60% of the time. This is not only about finding the solution, but also about having your team arrive at a solution. You learn material much better when teaching it, so I want you to demonstrate patience and willingness to help your peers.

The Files

The Makefile includes the following commands:

```
$ make obj
$ make bin
$ make clean
```

Note that `make obj` is identical to a simple `make`.

The `main.c` file is boring and, since every executable binary requires a `main` function, it is there just to allow you to make an executable program. `calls.c` is going to be the main focus of today. Currently it only has one function that calls another.

Calling Conventions

Recall that a calling convention is just a set of assumptions made by the calling function (`caller`) and the called function (`callee`) when a function is called. Calling conventions exist so that

1. when a function is called, the caller knows which of the registers it was using it should save, and which the callee saves, and
2. how to pass arguments and get return values.

The calling convention assumptions codify:

- how arguments should be passed (in registers, or on the stack?),
- how return values are returned (registers or stack?), and

- which registers should be saved by caller, and which by callee.

Our tool: `objdump`

`objdump` is a program that allows us to look at the inner deals of an object (i.e. a `.o` file) or binary. It has a boat-load of features, but we're going to be using `objdump -S` which dumps out the assembly code for your object/binary. On most systems, it does something magical: it normally outputs the C code interspersed with the assembly![1] This is a great way to learn assembly. For example, check this out:

```
$ make
gcc -Wall -Wextra -Werror -g -O0 -c -o obj.o calls.c

$ objdump -S obj.o
obj.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <callee>:
#include <stdio.h>

void
callee(void)
{
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp

}
   4:   90                      nop
   5:   5d                      pop    %rbp
   6:   c3                      retq

0000000000000007 <caller>:

void
caller(void)
{
   7:   55                      push   %rbp
   8:   48 89 e5                mov    %rsp,%rbp
    callee();
   b:   e8 00 00 00 00          callq  10 <caller+0x9>
}
```

[1] Except on repl.it because we don't get nice things.

```
  10:   90                      nop
  11:   5d                      pop    %rbp
  12:   c3                      retq
```

Notice the `callee()` above instruction `b`! That's essentially saying that the `callq` instruction corresponds to the function call `callee()`. Cool!

Note, in that output, you have three columns.

```
   b:   e8 00 00 00 00          callq  10 <caller+0x9>
|-1-| |-----------2----------| |---------3---------|
```

Column 1 is the address of the instruction, 2 is the machine code for the instruction, and 3 is the "human readable" instruction.

Your Goal

Your goal is to answer the initial set of questions below: Think of this all as a puzzle. You have to come up with ideas about how to change the code to derive the answers. The questions:

- Can you explain how the assembly behaves as you'd expect the C to in the provided code?
- How does the prologue for the `callee` change when you add the following local variables? Why?

  ```
  int a = 2, b = 4;
  ```

- How is one argument passed into a function?
- How is a *second argument* passed into a function?
- How about the *6th argument*?
- How is the return value passed from `callee` to `caller`?
- Run `make bin` which creates an executable binary (run it with `./bin`). Use `objdump` on `bin`. Why is there so much extra stuff? What is all of that stuff?

Much harder questions:

- Change the local variables from `int a = 2, b = 4;` to `int a, b;` for a simple invocation between caller and callee. Why does their allocation disappear from the prologue?
- Change the `-O0` (note that's a capital `o`, followed by the digit `0`) to `-O3`. That enables strong compiler optimizations. Explain what happens.
- When you look at the executable binary, can you guess what `_start` is? What is its relationship to `main`?

- How does the function return the second return value? (Recall that in C, you return the second value by passing a pointer as an argument that the function dereferences to set the return value.)

Assembly Cheatsheet

A few pieces of information:

- `%r*x` - register `*` (e.g. the `a` register for `%rax`)
- `%r#` - where `#` is a number (starting at `8`) – the `#`-th register, just another name for more registers
- `%rsp` - stack pointer (the bottom of the stack – remember a function call expands the stack *downwards* from high addresses to lower addresses)
- `%rbp` - the frame pointer that points to the start of the invocation frame for the current function
- `$0x#` - is a hexdecimal constant value `#`.
- `0x#(%r)` - assuming `%r` is a register that holds a memory address and `0x#` is a hexdecimal constant value, this is very strange syntax for array lookup. It is equivalent to `%r[0x#]`, or `a[i]` if `a` is a `char *`, and holds the value in `%r`, while `i` is equal to `0x#`.

Most functions will have a pretty uniform *prologue* and *epilogue* which are the instructions at the start of each function, and at the end. They often are concerned with setting up the stack, and returning to the previous function.

Most numerical constants in this assembly are hexdecimal (base-16), not base-10.

Some common instructions follow. Note that the `q` postfix on instructions means that it is a 64-bit variant, so the `q` can be ignored for now.

- `call fn` pushes the instruction *after* this `call` onto the stack, and jumps to the function `fn`. Note that `fn` might be a little strange (a relative address directly below the call instruction?).
- `ret` returns from a function. It pops the value off the top of the stack (which was likely pushed by `call`), and jumps to it. This has the effect of resuming execution in the calling function.
- `push %r` and `pop %r` push the register `%r` onto the stack (i.e. store it to `%rsp` and do `%rsp -= sizeof(i)`), and pop the value off the top of the stack into `%r`.
- `mov a, b` with move `a` to `b`. If `a` is a register, and `b` is an address in memory, it executes a *store*. If `a` is an address in memory, and `b` is a register, it executes a *load*. If both `a` and `b` are registers, it copies the first register into the second.
- `nop` is a "no operation" or "no op". Often these simply take space and allow the following instructions to be aligned in a specific way. Deep compiler

and architecture magic here.

- `sub` and `add` subtract and add!
- `leave` is one I'm only lightly going to touch on. Part of the prologue, it effective restores the previous frame pointer for the caller.