CS-118-02 Week 5

Varun Narravula

2021-02-23

Assembly Labels

What is a label in assembly? You've already seen labels; one of them is the statrting label from which execution starts, called `_start`. Another is `exit`; you can define a lot of labels in your program; in fact, any number of them. Another is the `syscall` instruction, which calls a system service or subsystem in the Linux kernel, which is a simple function. It's important to note that a section is not a label; `.text` is not a label, it's a section.

Let's look at the normal exit routine again:

```
exit: ; Label
    mov rax, 60
    mov rdi, 0
    syscall
```

Here, `exit` is the label. If you look at `rdi`, you could technically put any number there, but the number that is present is important. Why? It's because it denotes the exit code of the program. The program may not look any different in its execution if you put, say, 10, there. But a lot of people rely on this number. If you put 10 there, and run the program, then run `echo $?` after running the program, you'll see 10. This is the error code. `echo ?` prints the status code that the program put into `rdi` here. Sysadmins and a lot of other people rely on this status code to be able to see what the status of the last ran program was, and certain numbers correspond to different statuses depending on how the program runs. You can look at a manual for the program for that most of the time. 0 is universally known as the success status code; all other status codes are errors of some sort, so you may make some poor sysadmin's life hell if you put anything else here when the program, in fact, was successful.

C and C++ have labels too; you don't see it as often; sometimes it's there when you need more advanced loop control. Here's an example:

```
int main() {
    label: while(true) {
        std::cout << "Hello world!" << "\n";
        if (true) {
            break label;
        }
    }
}
```

Why is this important or relevant to assembly? In assembly this allows for jumping around the code! it allows for conditional execution. There is an instruction that enables this, called `jmp`. This is the way to create what are essentially `if` statements. Let's see an example of this:

```
section .data
msg     db `Equal`, 0xA, "Bruh", 0xA ; What you should print and length
msgLen  dq 11 ; Length of the message, which includes null terminator

section .text
global  _start

_start:
    mov eax, 5
    cmp eax, 5
    je  .print_equal ; Jump if equal as shown by cmp
.print_equal:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, qword [msgLen]
    syscall

exit: ; Normal exit routine
    mov rax, 60
    mov rdi, 0
    syscall
```

There's some variants of `jmp`, which is actually a condition-less jump like `goto` in C. But what is this `cmp`? You haven't seen it before; it compares the two operands, but in a different way. Think of instructions like `sub`; they move the result into the first operand. `cmp` here sets the zero flag in `eflags` if subtracting the two operands results in zero (effectively comparing if they are equal). `je` is a variant of the `jmp` instruction that stands for `jump equal`, which jumps to a label given to it as an argument when the CPU set the zero flag to zero. There's lots of other variants of `jmp`; you can see them here.

Raw link: [http://unixwiz.net/techtips/x86-jumps.html]

Printing in Assembly Using `write`

Let's look at the `write` system call to solidify our understanding a little more on how system calls work. The code:

```
section .data
msg db `Hello world`, 0xA

; other code

_start:
    mov rax, 1; syscall number for `write`
    mov rdi, 1; STDOUT is 1
    mov rsi, `Hello world`; The actual message
    mov rdx, qword [msgLen]; The length of the message
    syscall ; Calls `write` in the kernel
```

The comments make this a bit clearer; `write` uses the register `rax` like the call to `exit` does (all system calls use this register for the call). Here, it uses `rdi` for the file handle number (standard output, or the screen, corresponds to 1), `rsi` for the char pointer array (aka string), and `rdx` for the length of that string. `syscall` does the same thing here that it does in the `exit` system call code; it calls the code in the Linux kernel with the specified arguments in the registers for you.

The Linux kernel hardcodes these values for which registers to use for the arguments for each system call, so you have to consult the documentation for your system call that you will use. A lot of code writing consists of reading the documentation and applying it, so get used to it!

C in Assembly

What if you want to call a C function from an assembly file? That's a bit harder, but not code-wise. It's much more involved in the compilation process. Here's a code example that calls `printf` from the C standard library:

```
extern printf

section .data
msg1    db "The value of var1 %d var2 %d", 10, 0
var1    dd 256
var2    dd 512

section .text
global  _start

_start:
    xor rax, rax
    mov rdi, msg1

    xor rsi, rsi
    mov esi, dword [var1]
    xor rdx, rdx
    mov edx, dword [var2]

    call printf
exit:
    mov rax, 60
    mov rdi, 0
    syscall
```

The code looks simple. There's a couple of important notes: one, `xor reg, reg` zeroes out a register faster than a `mov reg, 0`, because `mov` can be expensive. It's a shortcut for that. Second, the `extern printf` at the top allows the assembly file to use the `printf` function. But how do you tell the assembler where `printf` is? That's the weird part. The solution is to tell the assembler and linker where the program interpreter is.

The program interpreter for Linux OSes specifically is the way that the kernel launches binaries, and the assembly file in question needs to know where this is to find other libraries to call functions from. Almost all binaries using C code on Linux dynamically link to this file, hardcoded to `/lib/ld-linux-x86-64.so.2` on most systems, sometimes `/lib64/ld-linux-x86-64.so.2`, unless you use an arcane OS like NixOS or GuixSD like I do. Since it's hardcoded at this location, it's easy to link to it using these commands (assume the asm file is main.asm and you use `nasm`):

```
nasm -g -f elf64 ./main.asm -l main.lst
ld -g -o main main.o -lc --dynamic-linker /lib64/ld-linux-x86-64.so.2
```

Most of the arguments here are self explanatory. But what's this `lst` file? You don't need to know too much about it other than that it's a set of object code to compile, and allows the linker to link properly to the system interpreter and other libraries needed, such as the stdlib that actually contains printf.

Those two commands will allow you to compile your assembly file and be able to use C code from it. If you use other external libraries, you should provide them as such but you should not need to.