CS-118-02 Week 6

Varun Narravula

2021-02-28

You can follow along with the examples provided in the folders. I use Nix, and got a bit lazy when uploading, so you can look at instructions to build in the buildPhase portions of those files and ignore everything else.

Working With More Than 1 Assembly File

Last week, you used `printf` in assembly code by calling to it using `extern` and `call`. There is an important caveat: once you call, there's no actual way to go back. There's a series of hardcoded registers for the number of arguments: the first argument is in `rdi`, the next in `rsi`, `rdx`, and after 6 registers, they get stored on the stack.

Then, how do you create these functions in assembly? Or create two files and call code from it?

If you were to put something in a separate file called `comp` with a label called `comp`, and you were to return something from it, how would you use it? Say it needs to compare two numbers and returns 1 or 0 depending on if the first number is bigger than the second number. Something equal to this C function:

```c
int compare(int a, int b) {
    return a > b ? 1 : 0;
}
```

How will this work? Let's see a code example; This is `comp.asm`:

```
section .text
global  comp

comp:
    cmp rdi, rsi
    jg  .greater
    jle .lesser

.greater:
    mov eax, 1
```

```
    ret

.lesser:
    xor eax, eax
    ret
```

Since you've seen jump instructions before, this is simple. But there is no "main" function here; how do you call it from another file? The solution is to make `comp` global, and then during the linking process, `ld` will resolve the two together at link time. Let's see how `main.asm` will call this:

```
extern comp

section .text
global  _start

_start:
    mov  rdi, 2
    mov  rsi, 1
    call comp
    mov  ebx, eax

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

This looks like the way you used `printf` as an external function; that's because it works in the same way. By default, all labels in an assembly file are private to the file unless you export it using the `global` directive; it's like the concept of `public` and `private` in C++, Java, and other OOP languages. When you use `extern`, it's essentially equal to `import` or `#include`. Make sure to move the first and second arguments into their respective registers, `rdi` and `rsi`. Then, you can call `comp`, This will move to the instructions in `comp`, and you'll have your result in `eax`.

How to compile this? `ld` can take more than 1 object file and link both together into a single executable. Assuming the same file names, run these commands:

```
nasm -g -f elf64 ./comp.asm
nasm -g -f elf64 ./main.asm
ld -g -o main main.o comp.o
```

And there you have it; You now have a program that calls a label in a separate file and gets its results, like the C function above.

Call Assembly From C

Let's take this puzzle of calling functions from different places further. Now, what if you want to call assembly in C? There's a couple of different ways to do this. One is inline assembly, where you can give raw strings filled with assembly functions to a macro called `__asm__`. Here's an example in C:

```
int foo(void) {
    __asm__( // Assembly function body
        "  mov $100,%eax \n"
        "  ret \n"
    );
    return 1; // never gets here, due to ret
}
```

Notice some glaring problems: one, the code to do this looks horrifying; it's pretty ugly. Second, the syntax used is different from the Intel syntax that `nasm` or `yasm` uses; it's called AT&T syntax, and this is the syntax for the GNU `gas` assembler, which is different from the Intel syntax in a significant number of ways besides readability. Third, the flow of execution can be hard to trace because if you use `jmp` or `ret` or anything else that changes the flow of execution inline, then function bodies will never finish properly. A more common way to do this is using a separate `asm` file and linking them together. Take a look at this C++ file:

```
#include <iostream>

extern "C" int comp(int a, int b);

int main() {
  std::cout << comp(2, 1) << std::endl;
  return 0;
}
```

The same thing in C:

```
#include <stdio>

extern int comp(int a, int b);

int main(int argc, char *argv[]) {
    printf("%d\n", comp(2, 1));
}
```

And their corresponding `comp.s`:

```
.intel_syntax noprefix

.global comp
```

```
.text

comp:
    cmp rdi, rsi
    jg  .greater
    jle .lesser

.greater:
    mov eax, 1
    ret

.lesser:
    xor eax, eax
    ret
```

There's glaring differences here, too. Why does the top of the assembly file look different? Why is it called `comp.s`? And why is this `extern` thing different in the C++ file and the C file? The explanation gets a bit lengthy.

First, the difference between `extern "C"` and `extern`. Why is it different for C and C++? The reason is the way C++ works. C++ is object-oriented, so it needs to perform "name-mangling" to be able to compile objects that have the same name in code but have different actions, unlike C. GCC will not recognize this name mangling by default and will get confused when linking, and it will throw an error. Because of this, in a C++ file, you have to specify `extern "C"` to create normal C symbols that can get linked to assembly instead of C++ name-mangled symbols. In a C file, this specification is not needed; after all, C doesn't understand C++! That's why the difference exists.

Second, why is the assembly file named `comp.s` and not `comp.asm`? This difference lies in the difference between GNU `gas` and `nasm`/`yasm`. By default, GCC will take files in assembly and link them with C files if they have the `.s` extension, but will refuse to recognize `.asm` files.

The reason for the different header from a normal `asm` file is also because of this quirk. Since GCC uses `gas` to compile files, and not `nasm` or `yasm`, `gas` needs to understand Intel syntax. It does, but will do so if you specify it as a directive. If you don't, it will attempt to interpret it as AT&T syntax and horribly fail. Let's go through it together: `.intel_syntax noprefix` is pretty self-explanatory. Despite that, `gas` does not understand `section` as a keyword of any sort, and will throw an error; instead, you have to use `gas` syntax and replace `global` with `.global` and `section .text` with `.text`.

What's the reason for all this? It's solely to be able to compile in one go. To compile into a single executable, the one command you need to run is `gcc -o main main.c comp.s` for C or `g++ -o main main.cpp comp.s` for C++.

To be fair, you don't have to do this at all. You could compile the C/C++ file to an object file using `gcc -c -o main.o main.c` or `g++ -c -o main.o`

`main.cpp` and then compile the assembly using the normal `nasm/yasm` command and link the two object files together like you did in the first example. For example:

```
gcc -c -o main.o main.c
nasm -g -f elf64 comp.asm
ld -g -o main comp.o main.o
```

This will work fine. If you want to compile in one step, use the above trick where you change the file extension and the assembly header. I wanted you to be aware of both methods because both are commonly used.

Calling Custom C Function From Assembly

You've seen a variant of this before while calling `printf` from the standard library. Imagine if you need to compile a custom C function you made yourself and link it, without using anything from the standard library; how would you do this?

It's almost the same process. Here, let's stick with a C function, to avoid the name-mangling in C++, though if you remember to make it `extern "C"`, you should be fine. Assume the name of the C file is `call-from-assembly.c`:

```
extern int comp(int a, int b);

int comp(int a, int b) {
    return a > b ? 1 : 0;
}
```

It's the same as the initial example, except now there's the `extern` to expose it. The assembly to call it looks like this (assume it's named `call-from-assembly.asm`):

```
extern comp

section .text
global  _start

_start:
    mov  rdi, 2
    mov  rsi, 1
    call comp
    mov  ebx, eax

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

It's essentially the same as from before, nothing too special. How to compile? Also the same as before.

```
nasm -g -f elf64 ./call-from-assembly.asm
gcc -c -g call-from-assembly.c -o lib.o
ld -g -o main call-from-assembly.o lib.o
}
```

Keep in mind that I'm specifying a different name (`lib.o`) for the object file created by the C compiler to avoid overwriting the `call-from-assembly.asm` file's object code. I named it `lib.o` to show that it's a library containing a single C function. Not too bad, right? It's simple compared to before.

In most of this class (and also in the real world), you are going to call assembly code from C, so the second example is most important to note. The others are not as important to care about, but they are still important to know because they still can have some use cases.