CS 61C   Lecture Notes   Week 7

Topic: Compilation, Assembly, Linkage

Reading: Patterson & Hennessy, Section 3.9, A.1–A.5

The topic this week is to understand the various programs that get from a source program you write to an executable machine language program, especially when multiple source files are involved. Here's an overall picture for what happens when you type the shell command

```
% gcc a.c b.c -o foo
```

The C compiler translates each file into assembly language. Then the assembler translates each assembly language file into machine language. Then the loader (also called the linker) combines the machine language files, along with parts of the C library, to form a single executable program.

(The reason there are two names for the loader/linker is that it used to be "loader," but there's an ambiguity because to load a program can mean either what the loader does, namely to combine modules into a single program, or what the operating system does to get a program into memory when you say to run it. The newfangled idea is to reserve the word "load" for this operating system function, and call the combining of modules "linking.")

Details differ a lot in different systems, and sometimes even a single system may have more than one set of conventions; Linux, for example, supports three or four different executable file formats. The details presented here are a simplified example; no actual current system runs exactly this way. But the ideas are faithfully represented.

We're not going to talk about the compiler; you'll learn about that part of the process in CS 164.

The first thing we have to understand about the job of the assembler is how it can handle forward references, like this:

```
        beqz  $8, zero
        ...
zero:   ...
```

To translate the `beqz` instruction into machine language, the assembler must know the distance between that instruction and its target, the instruction labelled `zero`. But that label comes later in the file.

The usual solution is a *two-pass* assembler. That is, the assembler reads the source (assembly language) file twice. The first time, it just counts how many bytes of machine language will have been produced before each line, and when a label is seen, it adds that label and the byte count to its *symbol table*. Then, on the second pass, it actually generates the machine language instructions; when a label is used, the assembler can look up the corresponding address in its table.

Actually the assembler keeps *two* counters, one for text (machine language instructions) and one for data. The reason is that when the program is finally loaded, we're going to want all the text in one place, and all the data in another place, because the operating system will arrange to mark the text part as *read-only* so that the program can't mistakenly modify itself. (The reason we don't want to allow self-modification is that if several people are running the same program at the same time, the system can keep only one copy of the text part as long as it can guarantee that it won't change. If the program modified its own instructions while running, then a modification made for one user might not be appropriate for another user, so the text couldn't be shared.) Both of these counters start at zero. Of course the text and the data can't *really* both start at zero, but as you'll see, this will be straightened out by the linker.

The text comes first in the machine language file that the assembler generates. Since text and data can be intermixed in the source file, how does the assembler know where to start the data? After the first pass, it knows the complete length of the text, so it can start the data after that.

(The *real* truth is that there are *three* counters, one for text, one for nonzero data, and one for initially-zero data. The reason is to save space in executable program files; instead of having a gazillion zero words, the executable file just has a count of how many zeros there should be, and the system fills the space with zeros while loading the program. But from now on I'll talk as if there were just two counters, because it simplifies the examples and doesn't lose any interesting ideas.)

Relocation

The linker's job is to combine several `.o` (object) files. Each file contains a header, some text words, and some data words. In addition to your `.o` files, some additional ones are taken from the C library, a file called `libC.a`, which is just a bunch of `.o` files strung together into one big file (by a program called `ar` for "archive"). The linker will extract from the C library only those parts that your program actually needs. The C library has things like `printf()` and `exit()` and so on.

It's common these days that the first few memory locations aren't used in an executable program, so that an accidental reference to location zero (from a pointer that doesn't point anywhere) will signal an error instead of using whatever happened to be at location zero. Let's say the first actual program address is `0x400`. To simplify things, let's also leave out the library procedures and assume that everything we need is in `a.o` and `b.o`. Here's a table of how things might be arranged in memory for a particular case:

	file a.o	file b.o
assembler thinks text starts at	0	0
assembler thinks data starts at	0	0
length of text (bytes)	0x100	0x50
length of data (bytes)	0x244	0x300
text really starts at	0x400	0x500
data really starts at	0x800	0xa44

The linker looks first at the text segment. If we've decided it should start at `0x400`, then that's where `a.o`'s text will start. To find the starting address of the text for `b.o`, we add the length of `a.o`'s text to that starting address, getting `0x500`.

In principle we could put `a.o`'s data segment right after `b.o`'s text, but for reasons you'll see when we discuss virtual memory, we round up to the next multiple of `0x400`. So `a.o`'s data segment starts at `0x800`. The start address of `b.o`'s data is the start of `a.o`'s data plus the length of `a.o`'s data; that's how we compute the value `0xa44`.

If the program isn't where the assembler thought it would be, then the addresses that are used within the program may have to be changed. For example, let's say that `a.s` contained the instruction

```
jal foo
```

and according to the assembler's symbol table for that file, the label `foo` is at address `0x24`. Then the assembler turned that source instruction into the machine language instruction `0x0c000009`. (The first six bits are the opcode for `jal`; the 9 at the end is the jump target address `0x24` divided by four, since what goes in the J-format instruction is missing the last two (always zero) bits.)

But the *actual* address of `foo` is `0x424`, because what's in the assembler's symbol table is the distance from the start of the text, and `a.o`'s text starts at `0x400`. So the linker has to change this machine language instruction to `0x0c000109`. This is called *relocation*.

(Note: As we'll see later, there is another, unrelated process also called relocation, which was carried out by the hardware in some old computer systems. That one has to do with the way in which several independent programs can coexist in a timeshared computer. But what we're talking about this week is software relocation within a single program; at the end of this relocation process the program still thinks it has the computer's memory all to itself.)

What does the linker have to do for a `branch` instruction? Perhaps surprisingly, the answer is nothing. Since what's encoded in the machine language branch instruction is the *distance* from the instruction to its target, and that doesn't change if the entire chunk of text containing the branch and its target is moved to a different part of memory, the machine instruction is fine just as the assembler produced it. (What if the branch goes to a piece of text in a different file? That isn't allowed; the only way to get from one file to another is by calling a procedure, using `jal`.)

What other kinds of instructions must be relocated? One obvious possibility would be the load and store instructions, since they refer to memory addresses. But in fact loads and stores are not relocated, because there is no room in an I-format instruction for a complete address. Therefore, all load and store instructions use a small offset from a base address in a register, and it's the base address that must sometimes be relocated. There are three basic possibilities: local variables, global variables, and heap-allocated data.

Most data references are to local variables, which are in a stack frame, whose address isn't computed until the program is actually running. The load and store instructions have a fixed offset relative to base register `$29`. The offset is not affected by the linking process, so those instructions are fine. Dereferencing of pointers into the heap, in which you see an address such as `0($16)`, is also safe without relocation, since the address in the heap was provided by `malloc` at run time, not worked out at compile time. But a load or store of a global variable *may* require relocation in its address calculation.

If your program includes the assembly language instruction

```
lw    $18, x
```

where `x` is a variable name, the assembler may generate code like this:

```
lui   $1, left-half-of-x
ori   $1, right-half-of-x
lw    $18, 0($1)
```

It is the immediate values in the `lui` and `ori` instructions that must be relocated, not the `lw` instruction itself. This kind of relocation requires that the `lui` and `ori` instructions be consecutive in the program, since the linker must examine both of them to compute the correct 32-bit address. (There might be a carry from bit 15 to bit 16 during the relocation arithmetic.)

Since not all `lui` and `ori` instructions need relocation, how does the linker know which to relocate? The answer is that the assembler includes *relocation bits* in the `.o` file that it produces. There are a few bits for each word of the program; the linker can decode these bits to find out whether a given word needs relocation, and if so, what kind. (The relocation of jump instructions adds 1/4 of the text origin to the last 26 bits of the word; the relocation for `lui` and `ori`, if any, adds the data offset to the combined 32-bit address, and then splits the result into two 16-bit values.)

Are there other kinds of relocation? Yes, one more: a pointer in the data segment whose initial value is a label in either text or data. In this case, some offset must be added to the entire 32-bit word.

Furthermore, some relocation is with respect to the text segment (e.g., the target address of a jump) and some with respect to the data segment (the left-half and right-half kinds, generally). So the relocation bits must specify what kind of relocation (none, jump, `lui/ori`, fullword) and which segment (text, data) to use.

External relocation

So far we've been talking about references in a `.o` file to a label defined in the same file. In those cases, the assembler put the label's offset (relative to the beginning of the file's text or data) in the machine language instruction, so the linker can add what it knows — the actual starting address of the file's text or data — to what the assembler knows (the offset). What if one file contains a reference to something in another file?

In this case, the assembler has no information about the label's offset from the start of its file. (The assembler handles each file entirely separately; this is important so that when you change one part of a large program you just have to recompile one file, not the entire program.) Therefore, the assembler has to communicate the referenced symbol (the actual characters in its name) to the linker somewhere in the `.o` file; the *other* `.o` file must contain a symbol table showing the offset of the symbol's definition within that file's text or data.

We've already talked about a symbol table that the assembler makes for its own use, to handle forward references. We now see that the symbol table must also contain the names of *external* symbols that are used in the program. For these symbols there is no associated address, but there is a flag indicating that the symbol is supposed to be defined elsewhere.

When a reference to an external symbol is made, what the assembler puts in the actual machine word in the `.o` file is an *index* into its symbol table. So, if the program says

```
jal foo
```

and `foo` is entry number 5 in the symbol table, the assembler will put the number 5 where the jump address belongs in the instruction. The relocation bits for that word will indicate that the word needs J-format relocation, and that the symbol is external.

The `.o` file, we now see, contains five parts: a header, the words of the text segment, the words of the data segment, the relocation bits for both segments, and the symbol table.

The linker, when it finds a word whose relocation bits specify external relocation, uses the index in the word itself to find an entry in the same file's symbol table. Then it looks for a matching entry in some other file's symbol table. That symbol table contains an offset, which the linker can use to compute the final address, and then update the original instruction or pointer.

The header, by the way, contains the sizes of each piece of the file, and the location of each piece within the file. The header doesn't have to be very long, but in practice it occupies one complete disk record; every segment of the file starts at the beginning of a disk record, for reasons we'll discuss later when we study virtual memory.

The output from the linker is an executable file. This file has essentially the same format as a `.o` file, except that the header flags it as executable, and there are no relocation bits. There is usually still a symbol table, though. Even though the program itself has no need for the symbol table, a debugger (such as `gdb`) needs the table to be able to "disassemble" instructions and to allow the user to enter debugging commands using the names of procedures and variables.