Note 10: CALL- Compiler, Assembler, Linker, Loader

Kevin Moy

Abstract

This note will cover the basics of how programs actually run, specifically with CALL- the compiler, assembler, linker, and loader. The goal, of course, is conversion of higher-level (C) code to machine code that the CPU actually understands and can execute.

1 Interpreters and Translators

There are multiple ways to run a program. Languages such as Lisp and Python use **interpreters**, which directly execute the program from the source. Other languages like Java and C, however, utilize a **translator** (compiler), which translates higher-level source code to another language.

We use interpreters when we want more simplicity (interpreters are far easier to write and look nicer) but don't care as much about performance. An added bonus is that interpreters provide **instruction set independence**: they can run on any machine, not specific to any architecture.

However, for higher performance (speed), compilers are the way to go. Translated or compiled code almost always overpowers interpreted code in efficiency and performance. This forms the base of many applications, such as operating systems and CPUs, that depend on maximal performance heavily.

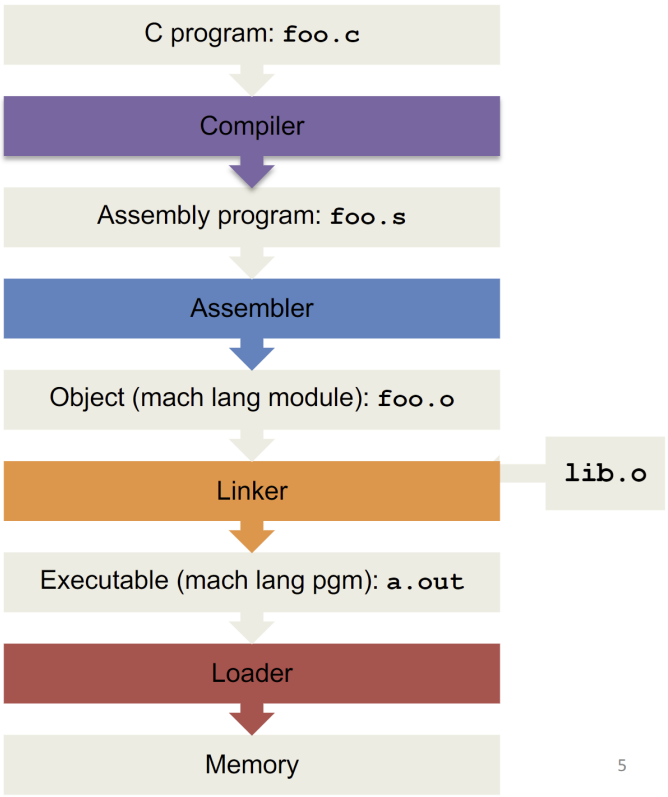
There is an additional bonus with compiled code- it serves as a form of source code encryption.

1.1 Translation in C

One of C's features was that we only needed to re-compile modified files in a project, i.e. we can compile files *separately* instead of all at once.

What can be accessed across files in a project? Functions and global variables. Think about what we need to do (with CALL) to make them accessible.

Here is a diagram of CALL:



We see the compiler translates C code into assembly (a `.s` file) code. The assembler takes the assembly code and turns it into an object file (`.o` file)-binary. The linker takes the object file as well as several other appropriate library object files and links them all together to create an executable: (a `.out` file or `.exe` file on windows). Finally, the loader takes the executable and loads it into memory for our CPU to execute.

The next four sections will describe each of the four machines in CALL with detail.

2 The Compiler

The compiler takes in some higher-level language (HLL) code and outputs assembly language code. Such output may contain pseudoinstructions. There is technically a preprocessor step to handle directives in our HLL code but it's kind of trivial. As far as CS61C goes that's literally all we need to know about the compiler- other courses like CS164 cover them in more detail.

3 The Assembler

This is probably the most complicated machine in CALL- so pay attention!

The assembler takes in assembly code and outputs **object code** (a `.o` file), known as *true assembly*. Some of the tasks that the assembler takes care of is handling directives (`.text`, `.data`,etc.), converting pseudoinstructions to actual instructions, and producing machine language.

3.1 Assembler Directives

Assembler directives are instructions to the assembler that do NOT count as part of the machine instructions at the end. Here are some examples:

- `.text`: Place (instructions) below into text (code) segment in memory.
- `.data`: Place (data) below into static (data) segment in memory.
- `.globl Label`: Declare `Label` global (other files can access).
- `.asciiz string`: Store `string` in memory and null-terminate.
- `.word w1 w2 ... wn`: Store `n` 32-bit values into successive word-length memory cells.

There are more assembly directives than this, of course, but these are the ones relevant to this course.

3.2 PseudoInstruction Replacement

The assembler also replaces pseudoinstructions with their real-instruction equivalent (sets).

3.3 Producing Machine Language

Finally, the assembler will take all of its assembly code and produce machine language in the form of an object file. This concerns converting each instruction into machine code, or binary.

So for arithmetic and logical instructions this seems simple enough- each instruction contains exactly the data needed for a full conversion. However, what about *dependent* instructions like branches and jumps- all of which require a relative address? Once we replace all our pseudoinstructions though, we know the exact number of instructions to branch to, so we are good.

However, there is a problem with "forward referencing"- branch instructions referring to labels at an address ahead of it (declared below the branch instruction). How would the assembler know how many instructions to branch? To solve this, **the assembler makes two passes:**

- Pass 1: The first pass converts pseudoinstructions and notes label positions. Now the assembler will know the relative offsets It also does some more housekeeping stuff like removing comments and checking for syntax errors.
- Pass 2: Use the pass 1 label positions (differences) to generate the relative addresses needed for branches and jumps. We output the final `.o` file after doing this.

A few exceptions, though: jump instructions need ABSOLUTE addresses of labels. So, forward or not, the assembler can't convert a jump instruction to machine language without knowing instruction positions in memory. We probably won't know this during this stage!

Additionally, what about for instructions like `la` where we'd need to know the address of some symbol? We won't know this at this stage either.

3.4 The Two Tables

Our solution is to create two tables: a symbol table and a relocation table.

3.4.1 The Symbol Table

The symbol table is a list of "items" that can be used by other files. Each file makes their own symbol table. Remember that global values for each file are functions and specifically defined global variables. So these would be, in assembly, `Label`s and anything in the `.data` section.

Our symbol table maps relevant symbols/labels to their addresses; this subsequently solves the forward reference problem. We fill in the table (symbol, address) during the first pass.

3.4.2 The Relocation Table.

The relocation table is the list of items (instructions) we need to find addresses of **later on**- specifically, when linking. Examples of these items include jumping to an internal/external label (`jal` or `jalr`) or data referenced in the `.data` section of code.

3.5 Object File Format

An object file is formatted as below:

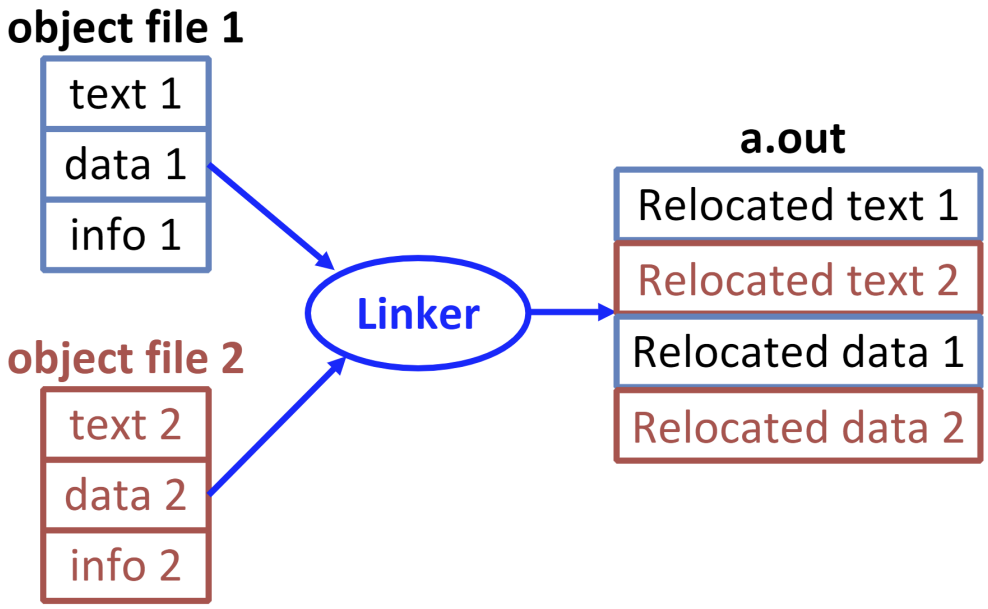
1. Object File Header: Lists size + position of other parts in `.o` file
2. Text Segment: Machine coded instructions
3. Data Segment: Data from source file
4. Relocation Table
5. Symbol Table
6. Debugging Info

4 The Linker

The linker takes in multiple `.o` files and outputs an executable (`a.out`). This combination of `.o` files is known as "linking".

This is what allows separate file compilation- we only need to recompile modified files. The linker edits the "links" (return address registers) in the jump-and-link instructions.

Below is a diagram of how the linker generally works to combine object files:



It relocates each segment of the object file, specifically text and data, such that they are contiguous in the executable.

The linker also **resolves all assembler reference issues** by going through the relocation table and filling in absolute addresses where needed.

Remember that PC-relative address instructions like branching instructions never have to be relocated. However, external function references (`jal`) WILL have to be relocated. Static data references (`auipc` and `addi`) will also always have to be relocated as well, as the offsets will probably have changed after linking.

Loads and stores to static memory (relative to the global pointer `gp`) also need relocation.

4.1 Resolving References

The linker assumes the first word of text is addressed at `0x10000`. The linker knows the length of each text and data segment from each object file, as well as their ordering. Linker calculates

5 The Loader

The loader takes an executable, loads it into memory, and runs it. **The loader is actually the operating system**, as one of its many tasks.

The loader first reads the executable's header to determine the size of text and data segments. Then, it creates a new address space for programs large enough such that the space can hold these text and data segments, *along with a stack segment*. We copy over the instructions/data from the executable into our newly made space. We pass the initial (command line) program arguments onto the stack. We initialize our (32) registers, zeroing out all except the stack pointer `sp` (it instead points to first free stack location). Finally, we run the executable, copying stack arguments to registers and setting `PC`. This is the "start routine" that is responsible for both starting and ending our program.