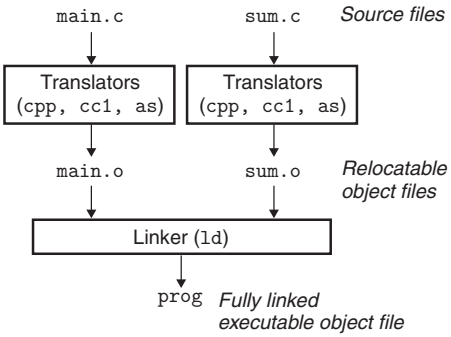

Figure 7.2
Static linking. The linker combines relocatable object files to form an executable object file `prog`.

```
cpp [other arguments] main.c /tmp/main.i
```

Next, the driver runs the C compiler (`cc1`), which translates `main.i` into an ASCII assembly-language file `main.s`:

```
cc1 /tmp/main.i -Og [other arguments] -o /tmp/main.s
```

Then, the driver runs the assembler (`as`), which translates `main.s` into a binary *relocatable object file* `main.o`:

```
as [other arguments] -o /tmp/main.o /tmp/main.s
```

The driver goes through the same process to generate `sum.o`. Finally, it runs the linker program `ld`, which combines `main.o` and `sum.o`, along with the necessary system object files, to create the binary *executable object file* `prog`:

```
ld -o prog [system object files and args] /tmp/main.o /tmp/sum.o
```

To run the executable `prog`, we type its name on the Linux shell's command line:

```
linux> ./prog
```

The shell invokes a function in the operating system called the *loader*, which copies the code and data in the executable file `prog` into memory, and then transfers control to the beginning of the program.

7.2 Static Linking

Static linkers such as the Linux LD program take as input a collection of relocatable object files and command-line arguments and generate as output a fully linked executable object file that can be loaded and run. The input relocatable object files consist of various code and data sections, where each section is a contiguous sequence of bytes. Instructions are in one section, initialized global variables are in another section, and uninitialized variables are in yet another section.

To build the executable, the linker must perform two main tasks:

Step 1. Symbol resolution. Object files define and reference *symbols*, where each symbol corresponds to a function, a global variable, or a *static variable* (i.e., any C variable declared with the `static` attribute). The purpose of symbol resolution is to associate each symbol *reference* with exactly one symbol *definition*.

Step 2. Relocation. Compilers and assemblers generate code and data sections that start at address 0. The linker *relocates* these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location. The linker blindly performs these relocations using detailed instructions, generated by the assembler, called *relocation entries*.

The sections that follow describe these tasks in more detail. As you read, keep in mind some basic facts about linkers: Object files are merely collections of blocks of bytes. Some of these blocks contain program code, others contain program data, and others contain data structures that guide the linker and loader. A linker concatenates blocks together, decides on run-time locations for the concatenated blocks, and modifies various locations within the code and data blocks. Linkers have minimal understanding of the target machine. The compilers and assemblers that generate the object files have already done most of the work.

7.3 Object Files

Object files come in three forms:

Relocatable object file. Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file.

Executable object file. Contains binary code and data in a form that can be copied directly into memory and executed.

Shared object file. A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time.

Compilers and assemblers generate relocatable object files (including shared object files). Linkers generate executable object files. Technically, an *object module* is a sequence of bytes, and an *object file* is an object module stored on disk in a file. However, we will use these terms interchangeably.

Object files are organized according to specific *object file formats*, which vary from system to system. The first Unix systems from Bell Labs used the `a.out` format. (To this day, executables are still referred to as `a.out` files.) Windows uses the Portable Executable (PE) format. Mac OS-X uses the Mach-O format. Modern x86-64 Linux and Unix systems use *Executable and Linkable Format (ELF)*. Although our discussion will focus on ELF, the basic concepts are similar, regardless of the particular format.