(a) `main.c`

code/link/main.c

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

code/link/main.c

(b) `sum.c`

code/link/sum.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

code/link/sum.c

Figure 7.1 Example program 1. The example program consists of two source files, `main.c` and `sum.c`. The `main` function initializes an array of `int`s, and then calls the `sum` function to sum the array elements.

This chapter provides a thorough discussion of all aspects of linking, from traditional static linking, to dynamic linking of shared libraries at load time, to dynamic linking of shared libraries at run time. We will describe the basic mechanisms using real examples, and we will identify situations in which linking issues can affect the performance and correctness of your programs. To keep things concrete and understandable, we will couch our discussion in the context of an x86-64 system running Linux and using the standard ELF-64 (hereafter referred to as ELF) object file format. However, it is important to realize that the basic concepts of linking are universal, regardless of the operating system, the ISA, or the object file format. Details may vary, but the concepts are the same.

7.1 Compiler Drivers

Consider the C program in Figure 7.1. It will serve as a simple running example throughout this chapter that will allow us to make some important points about how linkers work.

Most compilation systems provide a *compiler driver* that invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user. For example, to build the example program using the GNU compilation system, we might invoke the GCC driver by typing the following command to the shell:

```
linux> gcc -Og -o prog main.c sum.c
```

Figure 7.2 summarizes the activities of the driver as it translates the example program from an ASCII source file into an executable object file. (If you want to see these steps for yourself, run GCC with the `-v` option.) The driver first runs the C preprocessor (`cpp`),[1] which translates the C source file `main.c` into an ASCII intermediate file `main.i`:

1. In some versions of GCC, the preprocessor is integrated into the compiler driver.

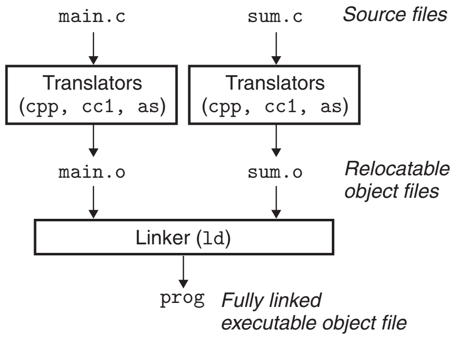
main.c
sum.c
Source files
Translators (cpp, cc1, as)
Translators (cpp, cc1, as)
main.o
sum.o
Relocatable object files
Linker (ld)
prog
Fully linked executable object file

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.