CHAPTER 7

Linking

Linking is the process of collecting and combining various pieces of code and data into a single file that can be *loaded* (copied) into memory and executed. Linking can be performed at *compile time*, when the source code is translated into machine code; at *load time*, when the program is loaded into memory and executed by the *loader*; and even at *run time*, by application programs. On early computer systems, linking was performed manually. On modern systems, linking is performed automatically by programs called *linkers*.

Linkers play a crucial role in software development because they enable *separate compilation*. Instead of organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately. When we change one of these modules, we simply recompile it and relink the application, without having to recompile the other files.

Linking is usually handled quietly by the linker and is not an important issue for students who are building small programs in introductory programming classes. So why bother learning about linking?

- *Understanding linkers will help you build large programs.* Programmers who build large programs often encounter linker errors caused by missing modules, missing libraries, or incompatible library versions. Unless you understand how a linker resolves references, what a library is, and how a linker uses a library to resolve references, these kinds of errors will be baffling and frustrating.
- *Understanding linkers will help you avoid dangerous programming errors.* The decisions that Linux linkers make when they resolve symbol references can silently affect the correctness of your programs. Programs that incorrectly define multiple global variables can pass through the linker without any warnings in the default case. The resulting programs can exhibit baffling run-time behavior and are extremely difficult to debug. We will show you how this happens and how to avoid it.
- *Understanding linking will help you understand how language scoping rules are implemented.* For example, what is the difference between global and local variables? What does it really mean when you define a variable or function with the `static` attribute?
- *Understanding linking will help you understand other important systems concepts.* The executable object files produced by linkers play key roles in important systems functions such as loading and running programs, virtual memory, paging, and memory mapping.
- *Understanding linking will enable you to exploit shared libraries.* For many years, linking was considered to be fairly straightforward and uninteresting. However, with the increased importance of shared libraries and dynamic linking in modern operating systems, linking is a sophisticated process that provides the knowledgeable programmer with significant power. For example, many software products use shared libraries to upgrade shrink-wrapped binaries at run time. Also, many Web servers rely on dynamic linking of shared libraries to serve dynamic content.

(a) `main.c` *code/link/main.c*

```
1   int sum(int *a, int n);
2
3   int array[2] = {1, 2};
4
5   int main()
6   {
7       int val = sum(array, 2);
8       return val;
9   }
```

code/link/main.c

(b) `sum.c` *code/link/sum.c*

```
1   int sum(int *a, int n)
2   {
3       int i, s = 0;
4
5       for (i = 0; i < n; i++) {
6           s += a[i];
7       }
8       return s;
9   }
```

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.

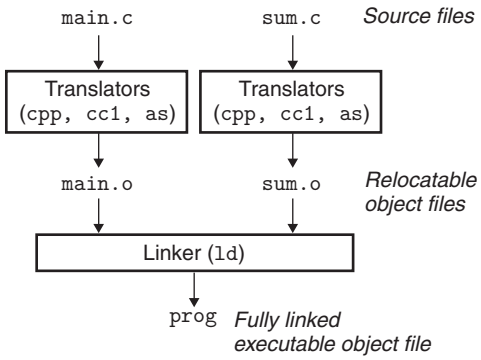


Figure 7.2
Static linking. The linker combines relocatable object files to form an executable object file `prog`.

```
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.

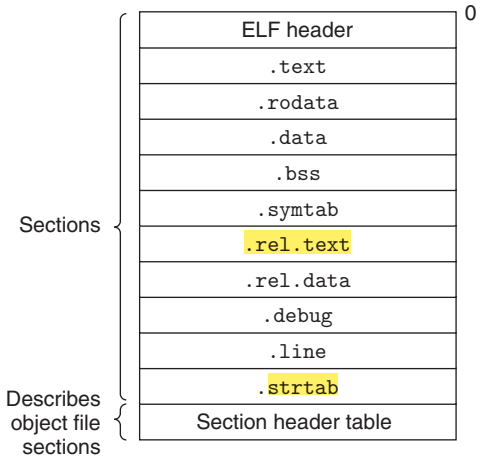


Figure 7.3
Typical ELF relocatable object file.

7.4 Relocatable Object Files

Figure 7.3 shows the format of a typical ELF relocatable object file. The *ELF header* begins with a 16-byte sequence that describes the word size and byte ordering of the system that generated the file. The rest of the ELF header contains information that allows a linker to parse and interpret the object file. This includes the size of the ELF header, the object file type (e.g., relocatable, executable, or shared), the machine type (e.g., x86-64), the file offset of the section header table, and the size and number of entries in the section header table. The locations and sizes of the various sections are described by the *section header table*, which contains a fixed-size entry for each section in the object file.

Sandwiched between the ELF header and the section header table are the sections themselves. A typical ELF relocatable object file contains the following sections:

`.text` The machine code of the compiled program.

`.rodata` Read-only data such as the format strings in `printf` statements, and jump tables for switch statements.

`.data` *Initialized* global and static C variables. Local C variables are maintained at run time on the stack and do *not* appear in either the `.data` or `.bss` sections.

`.bss` *Uninitialized* global and static C variables, along with any global or static variables that are initialized to zero. This section occupies no actual space in the object file; it is merely a placeholder. Object file formats distinguish between initialized and uninitialized variables for space efficiency: uninitialized variables do not have to occupy any actual disk space in the object file. At run time, these variables are allocated in memory with an initial value of zero.

Aside Why is uninitialized data called `.bss`?

The use of the term `.bss` to denote uninitialized data is universal. It was originally an acronym for the "block started by symbol" directive from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the `.data` and `.bss` sections is to think of "bss" as an abbreviation for "Better Save Space!"

`.symtab` A *symbol table* with information about functions and global variables that are defined and referenced in the program. Some programmers mistakenly believe that a program must be compiled with the `-g` option to get symbol table information. In fact, every relocatable object file has a symbol table in `.symtab` (unless the programmer has specifically removed it with the STRIP command). However, unlike the symbol table inside a compiler, the `.symtab` symbol table does not contain entries for local variables.

`.rel.text` A list of locations in the `.text` section that will need to be modified when the linker combines this object file with others. In general, any instruction that calls an external function or references a global variable will need to be modified. On the other hand, instructions that call local functions do not need to be modified. Note that relocation information is not needed in executable object files, and is usually omitted unless the user explicitly instructs the linker to include it.

`.rel.data` Relocation information for any global variables that are referenced or defined by the module. In general, any initialized global variable whose initial value is the address of a global variable or externally defined function will need to be modified.

`.debug` A debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. It is only present if the compiler driver is invoked with the `-g` option.

`.line` A mapping between line numbers in the original C source program and machine code instructions in the `.text` section. It is only present if the compiler driver is invoked with the `-g` option.

`.strtab` A string table for the symbol tables in the `.symtab` and `.debug` sections and for the section names in the section headers. A string table is a sequence of null-terminated character strings.

7.5 Symbols and Symbol Tables

Each relocatable object module, *m*, has a symbol table that contains information about the symbols that are defined and referenced by *m*. In the context of a linker, there are three different kinds of symbols:

- *Global symbols* that are defined by module *m* and that can be referenced by other modules. Global linker symbols correspond to *nonstatic* C functions and global variables.
- Global symbols that are referenced by module *m* but defined by some other module. Such symbols are called *externals* and correspond to nonstatic C functions and global variables that are defined in other modules.
- *Local symbols* that are defined and referenced exclusively by module *m*. These correspond to static C functions and global variables that are defined with the `static` attribute. These symbols are visible anywhere within module *m*, but cannot be referenced by other modules.

It is important to realize that local linker symbols are not the same as local program variables. The symbol table in `.symtab` does not contain any symbols that correspond to local nonstatic program variables. These are managed at run time on the stack and are not of interest to the linker.

Interestingly, local procedure variables that are defined with the C `static` attribute are not managed on the stack. Instead, the compiler allocates space in `.data` or `.bss` for each definition and creates a local linker symbol in the symbol table with a unique name. For example, suppose a pair of functions in the same module define a static local variable `x`:

```
1   int f()
2   {
3       static int x = 0;
4       return x;
5   }
6
7   int g()
8   {
9       static int x = 1;
10      return x;
11  }
```

In this case, the compiler exports a pair of local linker symbols with different names to the assembler. For example, it might use `x.1` for the definition in function `f` and `x.2` for the definition in function `g`.

Symbol tables are built by assemblers, using symbols exported by the compiler into the assembly-language `.s` file. An ELF symbol table is contained in the `.symtab` section. It contains an array of entries. Figure 7.4 shows the format of each entry.

The `name` is a byte offset into the string table that points to the null-terminated string name of the symbol. The `value` is the symbol's address. For relocatable modules, the `value` is an offset from the beginning of the section where the object is defined. For executable object files, the `value` is an absolute run-time address. The `size` is the size (in bytes) of the object. The `type` is usually either `data` or `function`. The symbol table can also contain entries for the individual sections

New to C? Hiding variable and function names with `static`

C programmers use the `static` attribute to hide variable and function declarations inside modules, much as you would use *public* and *private* declarations in Java and C++. In C, source files play the role of modules. Any global variable or function declared with the `static` attribute is private to that module. Similarly, any global variable or function declared without the `static` attribute is public and can be accessed by any other module. It is good programming practice to protect your variables and functions with the `static` attribute wherever possible.

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

Figure 7.4 ELF symbol table entry. The `type` and `binding` fields are 4 bits each.

and for the path name of the original source file. So there are distinct types for these objects as well. The `binding` field indicates whether the symbol is local or global.

Each symbol is assigned to some section of the object file, denoted by the `section` field, which is an index into the section header table. There are three special pseudosections that don't have entries in the section header table: ABS is for symbols that should not be relocated. UNDEF is for undefined symbols—that is, symbols that are referenced in this object module but defined elsewhere. COMMON is for uninitialized data objects that are not yet allocated. For COMMON symbols, the `value` field gives the alignment requirement, and `size` gives the minimum size. Note that these pseudosections exist only in relocatable object files; they do not exist in executable object files.

The distinction between COMMON and `.bss` is subtle. Modern versions of GCC assign symbols in relocatable object files to COMMON and `.bss` using the following convention:

COMMON	Uninitialized global variables
`.bss`	Uninitialized static variables, and global or static variables that are initialized to zero

The reason for this seemingly arbitrary distinction stems from the way the linker performs symbol resolution, which we will explain in Section 7.6.

The GNU READELF program is a handy tool for viewing the contents of object files. For example, here are the last three symbol table entries for the relocatable object file `main.o`, from the example program in Figure 7.1. The first eight entries, which are not shown, are local symbols that the linker uses internally.

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  8: 0000000000000000    24 FUNC    GLOBAL DEFAULT    1 main
  9: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
 10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

In this example, we see an entry for the definition of global symbol `main`, a 24-byte function located at an offset (i.e., `value`) of zero in the `.text` section. This is followed by the definition of the global symbol `array`, an 8-byte object located at an offset of zero in the `.data` section. The last entry comes from the reference to the external symbol `sum`. READELF identifies each section by an integer index. `Ndx=1` denotes the `.text` section, and `Ndx=3` denotes the `.data` section.

Practice Problem 7.1 (solution page 753)

This problem concerns the `m.o` and `swap.o` modules from Figure 7.5. For each symbol that is defined or referenced in `swap.o`, indicate whether or not it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `m.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, `.bss`, or COMMON) it is assigned to in the module.

(a) `m.c` — *code/link/m.c*

```
1  void swap();
2
3  int buf[2] = {1, 2};
4
5  int main()
6  {
7      swap();
8      return 0;
9  }
```

code/link/m.c

(b) `swap.c` — *code/link/swap.c*

```
1  extern int buf[];
2
3  int *bufp0 = &buf[0];
4  int *bufp1;
5
6  void swap()
7  {
8      int temp;
9
10     bufp1 = &buf[1];
11     temp = *bufp0;
12     *bufp0 = *bufp1;
13     *bufp1 = temp;
14 }
```

code/link/swap.c

Figure 7.5 **Example program for Practice Problem 7.1.**

Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf				
bufp0				
bufp1				
swap				
temp				

7.6 Symbol Resolution

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Symbol resolution is straightforward for references to local symbols that are defined in the same module as the reference. The compiler allows only one definition of each local symbol per module. The compiler also ensures that static local variables, which get local linker symbols, have unique names.

Resolving references to global symbols, however, is trickier. When the compiler encounters a symbol (either a variable or function name) that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle. If the linker is unable to find a definition for the referenced symbol in any of its input modules, it prints an (often cryptic) error message and terminates. For example, if we try to compile and link the following source file on a Linux machine,

```
1  void foo(void);
2
3  int main() {
4      foo();
5      return 0;
6  }
```

then the compiler runs without a hitch, but the linker terminates when it cannot resolve the reference to `foo`:

```
linux> gcc -Wall -Og -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function ‘main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to ‘foo'
```

Symbol resolution for global symbols is also tricky because multiple object modules might define global symbols with the same name. In this case, the linker must either flag an error or somehow choose one of the definitions and discard the rest. The approach adopted by Linux systems involves cooperation between the compiler, assembler, and linker and can introduce some baffling bugs to the unwary programmer.

Aside Mangling of linker symbols in C++ and Java

Both C++ and Java allow overloaded methods that have the same name in the source code but different parameter lists. So how does the linker tell the difference between these different overloaded functions? Overloaded functions in C++ and Java work because the compiler encodes each unique method and parameter list combination into a unique name for the linker. This encoding process is called *mangling*, and the inverse process is known as *demangling*.

Happily, C++ and Java use compatible mangling schemes. A mangled class name consists of the integer number of characters in the name followed by the original name. For example, the class `Foo` is encoded as `3Foo`. A method is encoded as the original method name, followed by `__`, followed by the mangled class name, followed by single letter encodings of each argument. For example, `Foo::bar(int, long)` is encoded as `bar__3Fooil`. Similar schemes are used to mangle global variable and template names.

7.6.1 How Linkers Resolve Duplicate Symbol Names

The input to the linker is a collection of relocatable object modules. Each of these modules defines a set of symbols, some of which are local (visible only to the module that defines it), and some of which are global (visible to other modules). What happens if multiple modules define global symbols with the same name? Here is the approach that Linux compilation systems use.

At compile time, the compiler exports each global symbol to the assembler as either *strong* or *weak*, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols.

Given this notion of strong and weak symbols, Linux linkers use the following rules for dealing with duplicate symbol names:

Rule 1. Multiple strong symbols with the same name are not allowed.

Rule 2. Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol.

Rule 3. Given multiple weak symbols with the same name, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

```
1  /* foo1.c */
2  int main()
3  {
4      return 0;
5  }
```

```
1  /* bar1.c */
2  int main()
3  {
4      return 0;
5  }
```

In this case, the linker will generate an error message because the strong symbol `main` is defined multiple times (rule 1):

```
linux> gcc foo1.c bar1.c
/tmp/ccq2Uxnd.o: In function ‘main':
bar1.c:(.text+0x0): multiple definition of ‘main'
```

Similarly, the linker will generate an error message for the following modules because the strong symbol x is defined twice (rule 1):

```
1   /* foo2.c */
2   int x = 15213;
3
4   int main()
5   {
6       return 0;
7   }
```

```
1   /* bar2.c */
2   int x = 15213;
3
4   void f()
5   {
6   }
```

However, if x is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (rule 2):

```
1   /* foo3.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x = 15213;
6
7   int main()
8   {
9       f();
10      printf("x = %d\n", x);
11      return 0;
12  }
```

```
1   /* bar3.c */
2   int x;
3
4   void f()
5   {
6       x = 15212;
7   }
```

At run time, function `f` changes the value of `x` from 15213 to 15212, which might come as an unwelcome surprise to the author of function `main`! Notice that the linker normally gives no indication that it has detected multiple definitions of `x`:

```
linux> gcc -o foobar3 foo3.c bar3.c
linux> ./foobar3
x = 15212
```

The same thing can happen if there are two weak definitions of `x` (rule 3):

```
1   /* foo4.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x;
6
7   int main()
8   {
9       x = 15213;
10      f();
11      printf("x = %d\n", x);
12      return 0;
13  }
```

```
1   /* bar4.c */
2   int x;
3
4   void f()
5   {
6       x = 15212;
7   }
```

The application of rules 2 and 3 can introduce some insidious run-time bugs that are incomprehensible to the unwary programmer, especially if the duplicate symbol definitions have different types. Consider the following example, in which `x` is inadvertently defined as an `int` in one module and a `double` in another:

```
1   /* foo5.c */
2   #include <stdio.h>
3   void f(void);
4
5   int y = 15212;
6   int x = 15213;
7
8   int main()
9   {
10      f();
```

```
11      printf("x = 0x%x y = 0x%x \n",
12             x, y);
13      return 0;
14  }
```

```
1   /* bar5.c */
2   double x;
3
4   void f()
5   {
6       x = -0.0;
7   }
```

On an x86-64/Linux machine, `double`s are 8 bytes and `int`s are 4 bytes. On our system, the address of `x` is `0x601020` and the address of `y` is `0x601024`. Thus, the assignment `x = -0.0` in line 6 of `bar5.c` will overwrite the memory locations for `x` and `y` (lines 5 and 6 in `foo5.c`) with the double-precision floating-point representation of negative zero!

```
linux> gcc -Wall -Og -o foobar5 foo5.c bar5.c
/usr/bin/ld: Warning: alignment 4 of symbol `x' in /tmp/cclUFK5g.o
is smaller than 8 in /tmp/ccbTLcb9.o
linux> ./foobar5
x = 0x0 y = 0x80000000
```

This is a subtle and nasty bug, especially because it triggers only a warning from the linker, and because it typically manifests itself much later in the execution of the program, far away from where the error occurred. In a large system with hundreds of modules, a bug of this kind is extremely hard to fix, especially because many programmers are not aware of how linkers work, and because they often ignore compiler warnings. When in doubt, invoke the linker with a flag such as the GCC `-fno-common` flag, which triggers an error if it encounters multiply-defined global symbols. Or use the `-Werror` option, which turns all warnings into errors.

In Section 7.5, we saw how the compiler assigns symbols to COMMON and `.bss` using a seemingly arbitrary convention. Actually, this convention is due to the fact that in some cases the linker allows multiple modules to define global symbols with the same name. When the compiler is translating some module and encounters a weak global symbol, say, `x`, it does not know if other modules also define `x`, and if so, it cannot predict which of the multiple instances of `x` the linker might choose. So the compiler defers the decision to the linker by assigning `x` to COMMON. On the other hand, if `x` is initialized to zero, then it is a strong symbol (and thus must be unique by rule 2), so the compiler can confidently assign it to `.bss`. Similarly, static symbols are unique by construction, so the compiler can confidently assign them to either `.data` or `.bss`.

Practice Problem 7.2 (solution page 754)

In this problem, let REF(x.i) → DEF(x.k) denote that the linker will associate an arbitrary reference to symbol x in module i to the definition of x in module k. For each example that follows, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN".

A.
```
/* Module 1 */          /* Module 2 */
int main()              int main;
{                       int p2()
}                       {
                        }
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

B.
```
/* Module 1 */          /* Module 2 */
void main()             int main = 1;
{                       int p2()
}                       {
                        }
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

C.
```
/* Module 1 */          /* Module 2 */
int x;                  double x = 1.0;
void main()             int p2()
{                       {
}                       }
```

(a) REF(x.1) → DEF(_____._____)

(b) REF(x.2) → DEF(_____._____)

7.6.2 Linking with Static Libraries

So far, we have assumed that the linker reads a collection of relocatable object files and links them together into an output executable file. In practice, all compilation systems provide a mechanism for packaging related object modules into a single file called a *static library*, which can then be supplied as input to the linker. When it builds the output executable, the linker copies only the object modules in the library that are referenced by the application program.

Why do systems support the notion of libraries? Consider ISO C99, which defines an extensive collection of standard I/O, string manipulation, and integer math functions such as `atoi`, `printf`, `scanf`, `strcpy`, and `rand`. They are available

to every C program in the `libc.a` library. ISO C99 also defines an extensive collection of floating-point math functions such as `sin`, `cos`, and `sqrt` in the `libm.a` library.

Consider the different approaches that compiler developers might use to provide these functions to users without the benefit of static libraries. One approach would be to have the compiler recognize calls to the standard functions and to generate the appropriate code directly. Pascal, which provides a small set of standard functions, takes this approach, but it is not feasible for C, because of the large number of standard functions defined by the C standard. It would add significant complexity to the compiler and would require a new compiler version each time a function was added, deleted, or modified. To application programmers, however, this approach would be quite convenient because the standard functions would always be available.

Another approach would be to put all of the standard C functions in a single relocatable object module, say, `libc.o`, that application programmers could link into their executables:

```
linux> gcc main.c /usr/lib/libc.o
```

This approach has the advantage that it would decouple the implementation of the standard functions from the implementation of the compiler, and would still be reasonably convenient for programmers. However, a big disadvantage is that every executable file in a system would now contain a complete copy of the collection of standard functions, which would be extremely wasteful of disk space. (On our system, `libc.a` is about 5 MB and `libm.a` is about 2 MB.) Worse, each running program would now contain its own copy of these functions in memory, which would be extremely wasteful of memory. Another big disadvantage is that any change to any standard function, no matter how small, would require the library developer to recompile the entire source file, a time-consuming operation that would complicate the development and maintenance of the standard functions.

We could address some of these problems by creating a separate relocatable file for each standard function and storing them in a well-known directory. However, this approach would require application programmers to explicitly link the appropriate object modules into their executables, a process that would be error prone and time consuming:

```
linux> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

The notion of a static library was developed to resolve the disadvantages of these various approaches. Related functions can be compiled into separate object modules and then packaged in a single static library file. Application programs can then use any of the functions defined in the library by specifying a single filename on the command line. For example, a program that uses functions from the C standard library and the math library could be compiled and linked with a command of the form

```
linux> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

(a) `addvec.o` — *code/link/addvec.c*

```
1   int addcnt = 0;
2
3   void addvec(int *x, int *y,
4               int *z, int n)
5   {
6       int i;
7
8       addcnt++;
9
10      for (i = 0; i < n; i++)
11          z[i] = x[i] + y[i];
12  }
```

code/link/addvec.c

(b) `multvec.o` — *code/link/multvec.c*

```
1   int multcnt = 0;
2
3   void multvec(int *x, int *y,
4                int *z, int n)
5   {
6       int i;
7
8       multcnt++;
9
10      for (i = 0; i < n; i++)
11          z[i] = x[i] * y[i];
12  }
```

code/link/multvec.c

Figure 7.6 Member object files in the `libvector` library.

At link time, the linker will only copy the object modules that are referenced by the program, which reduces the size of the executable on disk and in memory. On the other hand, the application programmer only needs to include the names of a few library files. (In fact, C compiler drivers always pass `libc.a` to the linker, so the reference to `libc.a` mentioned previously is unnecessary.)

On Linux systems, static libraries are stored on disk in a particular file format known as an *archive*. An archive is a collection of concatenated relocatable object files, with a header that describes the size and location of each member object file. Archive filenames are denoted with the `.a` suffix.

To make our discussion of libraries concrete, consider the pair of vector routines in Figure 7.6. Each routine, defined in its own object module, performs a vector operation on a pair of input vectors and stores the result in an output vector. As a side effect, each routine records the number of times it has been called by incrementing a global variable. (This will be useful when we explain the idea of position-independent code in Section 7.12.)

To create a static library of these functions, we would use the AR tool as follows:

```
linux> gcc -c addvec.c multvec.c
linux> ar rcs libvector.a addvec.o multvec.o
```

To use the library, we might write an application such as `main2.c` in Figure 7.7, which invokes the `addvec` library routine. The include (or header) file `vector.h` defines the function prototypes for the routines in `libvector.a`,

To build the executable, we would compile and link the input files `main2.o` and `libvector.a`:

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o ./libvector.a
```

code/link/main2.c

```
1   #include <stdio.h>
2   #include "vector.h"
3
4   int x[2] = {1, 2};
5   int y[2] = {3, 4};
6   int z[2];
7
8   int main()
9   {
10      addvec(x, y, z, 2);
11      printf("z = [%d %d]\n", z[0], z[1]);
12      return 0;
13  }
```

code/link/main2.c

Figure 7.7 Example program 2. This program invokes a function in the `libvector` library.

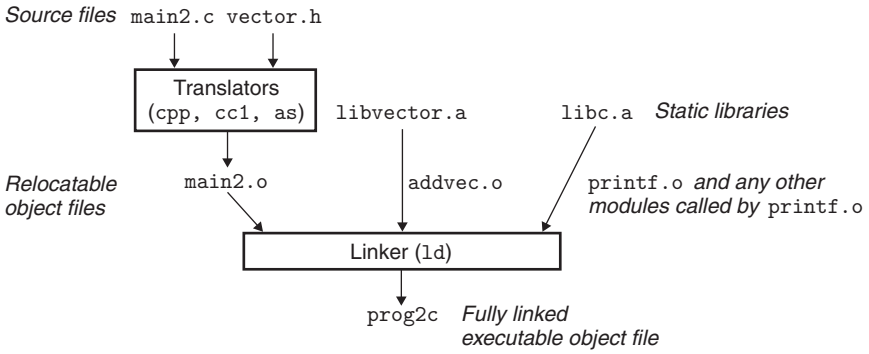


Figure 7.8 Linking with static libraries.

or equivalently,

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o -L. -lvector
```

Figure 7.8 summarizes the activity of the linker. The `-static` argument tells the compiler driver that the linker should build a fully linked executable object file that can be loaded into memory and run without any further linking at load time. The `-lvector` argument is a shorthand for `libvector.a`, and the `-L.` argument tells the linker to look for `libvector.a` in the current directory.

When the linker runs, it determines that the `addvec` symbol defined by `addvec.o` is referenced by `main2.o`, so it copies `addvec.o` into the executable.

Since the program doesn't reference any symbols defined by `multvec.o`, the linker does *not* copy this module into the executable. The linker also copies the `printf.o` module from `libc.a`, along with a number of other modules from the C run-time system.

7.6.3 How Linkers Use Static Libraries to Resolve References

While static libraries are useful, they are also a source of confusion to programmers because of the way the Linux linker uses them to resolve external references. During the symbol resolution phase, the linker scans the relocatable object files and archives left to right in the same sequential order that they appear on the compiler driver's command line. (The driver automatically translates any `.c` files on the command line into `.o` files.) During this scan, the linker maintains a set E of relocatable object files that will be merged to form the executable, a set U of unresolved symbols (i.e., symbols referred to but not yet defined), and a set D of symbols that have been defined in previous input files. Initially, E, U, and D are empty.

- For each input file f on the command line, the linker determines if f is an object file or an archive. If f is an object file, the linker adds f to E, updates U and D to reflect the symbol definitions and references in f, and proceeds to the next input file.
- If f is an archive, the linker attempts to match the unresolved symbols in U against the symbols defined by the members of the archive. If some archive member m defines a symbol that resolves a reference in U, then m is added to E, and the linker updates U and D to reflect the symbol definitions and references in m. This process iterates over the member object files in the archive until a fixed point is reached where U and D no longer change. At this point, any member object files not contained in E are simply discarded and the linker proceeds to the next input file.
- If U is nonempty when the linker finishes scanning the input files on the command line, it prints an error and terminates. Otherwise, it merges and relocates the object files in E to build the output executable file.

Unfortunately, this algorithm can result in some baffling link-time errors because the ordering of libraries and object files on the command line is significant. If the library that defines a symbol appears on the command line before the object file that references that symbol, then the reference will not be resolved and linking will fail. For example, consider the following:

```
linux> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function ‘main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to ‘addvec'
```

What happened? When `libvector.a` is processed, U is empty, so no member object files from `libvector.a` are added to E. Thus, the reference to `addvec` is never resolved and the linker emits an error message and terminates.

The general rule for libraries is to place them at the end of the command line. If the members of the different libraries are independent, in that no member references a symbol defined by another member, then the libraries can be placed at the end of the command line in any order. If, on the other hand, the libraries are not independent, then they must be ordered so that for each symbol *s* that is referenced externally by a member of an archive, at least one definition of *s* follows a reference to *s* on the command line. For example, suppose `foo.c` calls functions in `libx.a` and `libz.a` that call functions in `liby.a`. Then `libx.a` and `libz.a` must precede `liby.a` on the command line:

```
linux> gcc foo.c libx.a libz.a liby.a
```

Libraries can be repeated on the command line if necessary to satisfy the dependence requirements. For example, suppose `foo.c` calls a function in `libx.a` that calls a function in `liby.a` that calls a function in `libx.a`. Then `libx.a` must be repeated on the command line:

```
linux> gcc foo.c libx.a liby.a libx.a
```

Alternatively, we could combine `libx.a` and `liby.a` into a single archive.

Practice Problem 7.3 **(solution page 754)**

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references.

A. `p.o` → `libx.a`

B. `p.o` → `libx.a` → `liby.a`

C. `p.o` → `libx.a` → `liby.a` *and* `liby.a` → `libx.a` → `p.o`

7.7 Relocation

Once the linker has completed the symbol resolution step, it has associated each symbol reference in the code with exactly one symbol definition (i.e., a symbol table entry in one of its input object modules). At this point, the linker knows the exact sizes of the code and data sections in its input object modules. It is now ready to begin the relocation step, where it merges the input modules and assigns run-time addresses to each symbol. Relocation consists of two steps:

1. *Relocating sections and symbol definitions.* In this step, the linker merges all sections of the same type into a new aggregate section of the same type. For example, the `.data` sections from the input modules are all merged into one section that will become the `.data` section for the output executable object

file. The linker then assigns run-time memory addresses to the new aggregate sections, to each section defined by the input modules, and to each symbol defined by the input modules. When this step is complete, each instruction and global variable in the program has a unique run-time memory address.

2. *Relocating symbol references within sections.* In this step, the linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses. To perform this step, the linker relies on data structures in the relocatable object modules known as relocation entries, which we describe next.

7.7.1 Relocation Entries

When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory. Nor does it know the locations of any externally defined functions or global variables that are referenced by the module. So whenever the assembler encounters a reference to an object whose ultimate location is unknown, it generates a *relocation entry* that tells the linker how to modify the reference when it merges the object file into an executable. Relocation entries for code are placed in `.rel.text`. Relocation entries for data are placed in `.rel.data`.

Figure 7.9 shows the format of an ELF relocation entry. The `offset` is the section offset of the reference that will need to be modified. The `symbol` identifies the symbol that the modified reference should point to. The `type` tells the linker how to modify the new reference. The `addend` is a signed constant that is used by some types of relocations to bias the value of the modified reference.

ELF defines 32 different relocation types, many quite arcane. We are concerned with only the two most basic relocation types:

`R_X86_64_PC32`. Relocate a reference that uses a 32-bit PC-relative address. Recall from Section 3.6.3 that a PC-relative address is an offset from the current run-time value of the program counter (PC). When the CPU executes an instruction using PC-relative addressing, it forms the *effective address* (e.g., the target of the `call` instruction) by adding the 32-bit value

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

Figure 7.9 ELF relocation entry. Each entry identifies a reference that must be relocated and specifies how to compute the modified reference.

encoded in the instruction to the current run-time value of the PC, which is always the address of the next instruction in memory.

`R_X86_64_32`. Relocate a reference that uses a 32-bit absolute address. With absolute addressing, the CPU directly uses the 32-bit value encoded in the instruction as the effective address, without further modifications.

These two relocation types support the x86-64 *small code model*, which assumes that the total size of the code and data in the executable object file is smaller than 2 GB, and thus can be accessed at run-time using 32-bit PC-relative addresses. The small code model is the default for GCC. Programs larger than 2 GB can be compiled using the `-mcmodel=medium` (*medium code model*) and `-mcmodel=large` (*large code model*) flags, but we won't discuss those.

7.7.2 Relocating Symbol References

Figure 7.10 shows the pseudocode for the linker's relocation algorithm. Lines 1 and 2 iterate over each section `s` and each relocation entry `r` associated with each section. For concreteness, assume that each section `s` is an array of bytes and that each relocation entry `r` is a `struct` of type `Elf64_Rela`, as defined in Figure 7.9. Also, assume that when the algorithm runs, the linker has already chosen run-time addresses for each section (denoted `ADDR(s)`) and each symbol (denoted `ADDR(r.symbol)`). Line 3 computes the address in the `s` array of the 4-byte reference that needs to be relocated. If this reference uses PC-relative addressing, then it is relocated by lines 5–9. If the reference uses absolute addressing, then it is relocated by lines 11–13.

```
1  foreach section s {
2      foreach relocation entry r {
3          refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5          /* Relocate a PC-relative reference */
6          if (r.type == R_X86_64_PC32) {
7              refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8              *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9          }
10
11         /* Relocate an absolute reference */
12         if (r.type == R_X86_64_32)
13             *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14     }
15 }
```

Figure 7.10 **Relocation algorithm.**

code/link/main-relo.d

```
1  0000000000000000 <main>:
2     0:   48 83 ec 08             sub    $0x8,%rsp
3     4:   be 02 00 00 00          mov    $0x2,%esi
4     9:   bf 00 00 00 00          mov    $0x0,%edi         %edi = &array
5                          a: R_X86_64_32 array            Relocation entry

6     e:   e8 00 00 00 00          callq  13 <main+0x13>    sum()
7                          f: R_X86_64_PC32 sum-0x4        Relocation entry
8    13:   48 83 c4 08             add    $0x8,%rsp
9    17:   c3                      retq
```

code/link/main-relo.d

Figure 7.11 **Code and relocation entries from** `main.o`. The original C code is in Figure 7.1.

Let's see how the linker uses this algorithm to relocate the references in our example program in Figure 7.1. Figure 7.11 shows the disassembled code from `main.o`, as generated by the GNU OBJDUMP tool (`objdump -dx main.o`).

The `main` function references two global symbols, `array` and `sum`. For each reference, the assembler has generated a relocation entry, which is displayed on the following line.[2] The relocation entries tell the linker that the reference to `sum` should be relocated using a 32-bit PC-relative address, and the reference to `array` should be relocated using a 32-bit absolute address. The next two sections detail how the linker relocates these references.

Relocating PC-Relative References

In line 6 in Figure 7.11, function `main` calls the `sum` function, which is defined in module `sum.o`. The `call` instruction begins at section offset `0xe` and consists of the 1-byte opcode `0xe8`, followed by a placeholder for the 32-bit PC-relative reference to the target `sum`.

The corresponding relocation entry `r` consists of four fields:

```
r.offset = 0xf
r.symbol = sum
r.type   = R_X86_64_PC32
r.addend = -4
```

These fields tell the linker to modify the 32-bit PC-relative reference starting at offset `0xf` so that it will point to the `sum` routine at run time. Now, suppose that the linker has determined that

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

2. Recall that relocation entries and instructions are actually stored in different sections of the object file. The OBJDUMP tool displays them together for convenience.

and

```
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
```

Using the algorithm in Figure 7.10, the linker first computes the run-time address of the reference (line 7):

```
refaddr = ADDR(s)  + r.offset
        = 0x4004d0 + 0xf
        = 0x4004df
```

It then updates the reference so that it will point to the `sum` routine at run time (line 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        = (unsigned) (0x4004e8       + (-4)     - 0x4004df)
        = (unsigned) (0x5)
```

In the resulting executable object file, the `call` instruction has the following relocated form:

```
4004de:  e8 05 00 00 00          callq  4004e8 <sum>      sum()
```

At run time, the `call` instruction will be located at address `0x4004de`. When the CPU executes the `call` instruction, the PC has a value of `0x4004e3`, which is the address of the instruction immediately following the `call` instruction. To execute the call instruction, the CPU performs the following steps:

1. Push PC onto stack
2. PC $\leftarrow$ PC + 0x5 = 0x4004e3 + 0x5 = 0x4004e8

Thus, the next instruction to execute is the first instruction of the `sum` routine, which of course is what we want!

Relocating Absolute References

Relocating absolute references is straightforward. For example, in line 4 in Figure 7.11, the `mov` instruction copies the address of `array` (a 32-bit immediate value) into register `%edi`. The `mov` instruction begins at section offset `0x9` and consists of the 1-byte opcode `0xbf`, followed by a placeholder for the 32-bit absolute reference to `array`.

The corresponding relocation entry `r` consists of four fields:

```
r.offset = 0xa
r.symbol = array
r.type   = R_X86_64_32
r.addend = 0
```

These fields tell the linker to modify the absolute reference starting at offset `0xa` so that it will point to the first byte of `array` at run time. Now, suppose that the linker has determined that

(a) Relocated `.text` section

```
1  00000000004004d0 <main>:
2    4004d0:  48 83 ec 08           sub    $0x8,%rsp
3    4004d4:  be 02 00 00 00        mov    $0x2,%esi
4    4004d9:  bf 18 10 60 00        mov    $0x601018,%edi    %edi = &array
5    4004de:  e8 05 00 00 00        callq  4004e8 <sum>      sum()
6    4004e3:  48 83 c4 08           add    $0x8,%rsp
7    4004e7:  c3                    retq

8  00000000004004e8 <sum>:
9    4004e8:  b8 00 00 00 00        mov    $0x0,%eax
10   4004ed:  ba 00 00 00 00        mov    $0x0,%edx
11   4004f2:  eb 09                 jmp    4004fd <sum+0x15>
12   4004f4:  48 63 ca              movslq %edx,%rcx
13   4004f7:  03 04 8f              add    (%rdi,%rcx,4),%eax
14   4004fa:  83 c2 01              add    $0x1,%edx
15   4004fd:  39 f2                 cmp    %esi,%edx
16   4004ff:  7c f3                 jl     4004f4 <sum+0xc>
17   400501:  f3 c3                 repz retq
```

(b) Relocated `.data` section

```
1  0000000000601018 <array>:
2    601018:  01 00 00 00 02 00 00 00
```

Figure 7.12 Relocated `.text` and `.data` sections for the executable file `prog`. The original C code is in Figure 7.1.

```
ADDR(r.symbol) = ADDR(array) = 0x601018
```

The linker updates the reference using line 13 of the algorithm in Figure 7.10:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend)
        = (unsigned) (0x601018       + 0)
        = (unsigned) (0x601018)
```

In the resulting executable object file, the reference has the following relocated form:

```
  4004d9:  bf 18 10 60 00         mov    $0x601018,%edi   %edi = &array
```

Putting it all together, Figure 7.12 shows the relocated `.text` and `.data` sections in the final executable object file. At load time, the loader can copy the bytes from these sections directly into memory and execute the instructions without any further modifications.

Practice Problem 7.4 (solution page 754)

This problem concerns the relocated program in Figure 7.12(a).

A. What is the hex address of the relocated reference to `sum` in line 5?

B. What is the hex value of the relocated reference to `sum` in line 5?

Practice Problem 7.5 (solution page 754)

Consider the call to function `swap` in object file `m.o` (Figure 7.5).

```
9:   e8 00 00 00 00          callq  e <main+0xe>      swap()
```

with the following relocation entry:

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

Now suppose that the linker relocates `.text` in `m.o` to address `0x4004d0` and `swap` to address `0x4004e8`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

7.8 Executable Object Files

We have seen how the linker merges multiple object files into a single executable object file. Our example C program, which began life as a collection of ASCII text files, has been transformed into a single binary file that contains all of the information needed to load the program into memory and run it. Figure 7.13 summarizes the kinds of information in a typical ELF executable file.

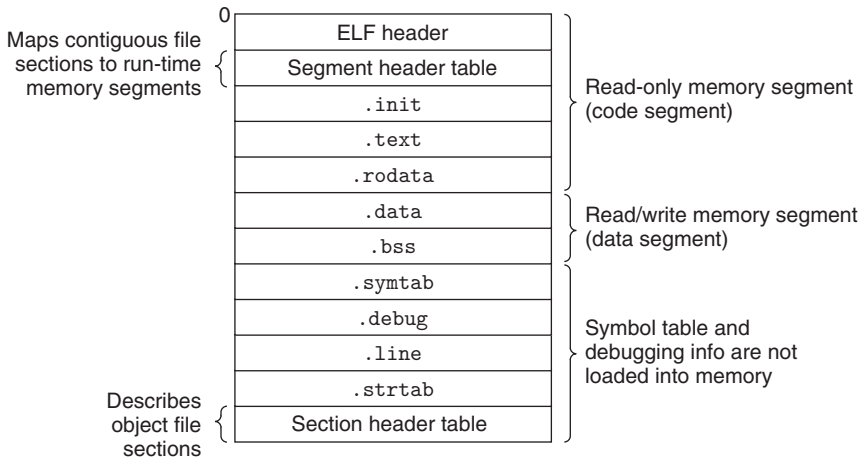


Figure 7.13 Typical ELF executable object file.

code/link/prog-exe.d

```
    Read-only code segment
1   LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
2        filesz 0x000000000000069c memsz 0x000000000000069c flags r-x

    Read/write data segment
3   LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
4        filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
```

code/link/prog-exe.d

Figure 7.14 Program header table for the example executable `prog`. `off`: offset in object file; `vaddr`/`paddr`: memory address; `align`: alignment requirement; `filesz`: segment size in object file; `memsz`: segment size in memory; `flags`: run-time permissions.

The format of an executable object file is similar to that of a relocatable object file. The ELF header describes the overall format of the file. It also includes the program's *entry point*, which is the address of the first instruction to execute when the program runs. The `.text`, `.rodata`, and `.data` sections are similar to those in a relocatable object file, except that these sections have been relocated to their eventual run-time memory addresses. The `.init` section defines a small function, called `_init`, that will be called by the program's initialization code. Since the executable is *fully linked* (relocated), it needs no `.rel` sections.

ELF executables are designed to be easy to load into memory, with contiguous chunks of the executable file mapped to contiguous memory segments. This mapping is described by the *program header table*. Figure 7.14 shows part of the program header table for our example executable `prog`, as displayed by OBJDUMP.

From the program header table, we see that two memory segments will be initialized with the contents of the executable object file. Lines 1 and 2 tell us that the first segment (the *code segment*) has read/execute permissions, starts at memory address `0x400000`, has a total size in memory of `0x69c` bytes, and is initialized with the first `0x69c` bytes of the executable object file, which includes the ELF header, the program header table, and the `.init`, `.text`, and `.rodata` sections.

Lines 3 and 4 tell us that the second segment (the *data segment*) has read/write permissions, starts at memory address `0x600df8`, has a total memory size of `0x230` bytes, and is initialized with the `0x228` bytes in the `.data` section starting at offset `0xdf8` in the object file. The remaining 8 bytes in the segment correspond to `.bss` data that will be initialized to zero at run time.

For any segment s, the linker must choose a starting address, `vaddr`, such that

$$\texttt{vaddr} \bmod \texttt{align} = \texttt{off} \bmod \texttt{align}$$

where `off` is the offset of the segment's first section in the object file, and `align` is the alignment specified in the program header ($2^{21} = \texttt{0x200000}$). For example, in the data segment in Figure 7.14,

$$\texttt{vaddr} \bmod \texttt{align} = \texttt{0x600df8} \bmod \texttt{0x200000} = \texttt{0xdf8}$$

and

$$\texttt{off} \bmod \texttt{align} = \texttt{0xdf8} \bmod \texttt{0x200000} = \texttt{0xdf8}$$

This alignment requirement is an optimization that enables segments in the object file to be transferred efficiently to memory when the program executes. The reason is somewhat subtle and is due to the way that virtual memory is organized as large contiguous power-of-2 chunks of bytes. You will learn all about virtual memory in Chapter 9.

7.9 Loading Executable Object Files

To run an executable object file `prog`, we can type its name to the Linux shell's command line:

```
linux> ./prog
```

Since `prog` does not correspond to a built-in shell command, the shell assumes that `prog` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the `loader`. Any Linux program can invoke the loader by calling the `execve` function, which we will describe in detail in Section 8.4.6. The loader copies the code and data in the executable object file from disk into memory and then runs the program by jumping to its first instruction, or *entry point*. This process of copying the program into memory and then running it is known as *loading*.

Every running Linux program has a run-time memory image similar to the one in Figure 7.15. On Linux x86-64 systems, the code segment starts at address `0x400000`, followed by the data segment. The run-time *heap* follows the data segment and grows upward via calls to the `malloc` library. (We will describe `malloc` and the heap in detail in Section 9.9.) This is followed by a region that is reserved for shared modules. The user stack starts below the largest legal user address ($2^{48} - 1$) and grows down, toward smaller memory addresses. The region above the stack, starting at address 2^{48}, is reserved for the code and data in the *kernel*, which is the memory-resident part of the operating system.

For simplicity, we've drawn the heap, data, and code segments as abutting each other, and we've placed the top of the stack at the largest legal user address. In practice, there is a gap between the code and data segments due to the alignment requirement on the `.data` segment (Section 7.8). Also, the linker uses address-space layout randomization (ASLR, Section 3.10.4) when it assigns run-time addresses to the stack, shared library, and heap segments. Even though the locations of these regions change each time the program is run, their relative positions are the same.

When the loader runs, it creates a memory image similar to the one shown in Figure 7.15. Guided by the program header table, it copies chunks of the

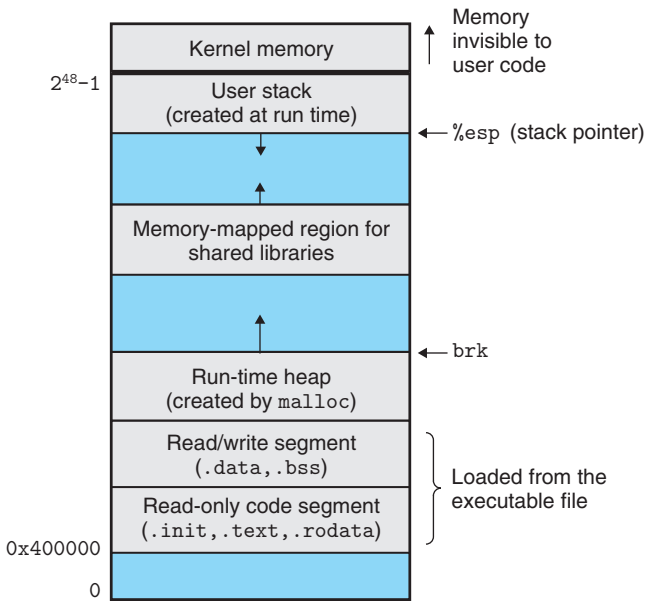


Figure 7.15
Linux x86-64 run-time memory image. Gaps due to segment alignment requirements and address-space layout randomization (ASLR) are not shown. Not to scale.

executable object file into the code and data segments. Next, the loader jumps to the program's entry point, which is always the address of the `_start` function. This function is defined in the system object file `crt1.o` and is the same for all C programs. The `_start` function calls the *system startup function*, `__libc_start_main`, which is defined in `libc.so`. It initializes the execution environment, calls the user-level `main` function, handles its return value, and if necessary returns control to the kernel.

7.10 Dynamic Linking with Shared Libraries

The static libraries that we studied in Section 7.6.2 address many of the issues associated with making large collections of related functions available to application programs. However, static libraries still have some significant disadvantages. Static libraries, like all software, need to be maintained and updated periodically. If application programmers want to use the most recent version of a library, they must somehow become aware that the library has changed and then explicitly relink their programs against the updated library.

Another issue is that almost every C program uses standard I/O functions such as `printf` and `scanf`. At run time, the code for these functions is duplicated in the text segment of each running process. On a typical system that is running hundreds of processes, this can be a significant waste of scarce memory system resources. (An interesting property of memory is that it is *always* a scarce resource, regardless

Aside How do loaders really work?

Our description of loading is conceptually correct but intentionally not entirely accurate. To understand how loading really works, you must understand the concepts of *processes*, *virtual memory*, and *memory mapping*, which we haven't discussed yet. As we encounter these concepts later in Chapters 8 and 9, we will revisit loading and gradually reveal the mystery to you.

For the impatient reader, here is a preview of how loading really works: Each program in a Linux system runs in the context of a process with its own virtual address space. When the shell runs a program, the parent shell process forks a child process that is a duplicate of the parent. The child process invokes the loader via the `execve` system call. The loader deletes the child's existing virtual memory segments and creates a new set of code, data, heap, and stack segments. The new stack and heap segments are initialized to zero. The new code and data segments are initialized to the contents of the executable file by mapping pages in the virtual address space to page-size chunks of the executable file. Finally, the loader jumps to the `_start` address, which eventually calls the application's `main` routine. Aside from some header information, there is no copying of data from disk to memory during loading. The copying is deferred until the CPU references a mapped virtual page, at which point the operating system automatically transfers the page from disk to memory using its paging mechanism.

of how much there is in a system. Disk space and kitchen trash cans share this same property.)

Shared libraries are modern innovations that address the disadvantages of static libraries. A shared library is an object module that, at either run time or load time, can be loaded at an arbitrary memory address and linked with a program in memory. This process is known as *dynamic linking* and is performed by a program called a *dynamic linker*. Shared libraries are also referred to as *shared objects*, and on Linux systems they are indicated by the `.so` suffix. Microsoft operating systems make heavy use of shared libraries, which they refer to as DLLs (dynamic link libraries).

Shared libraries are "shared" in two different ways. First, in any given file system, there is exactly one `.so` file for a particular library. The code and data in this `.so` file are shared by all of the executable object files that reference the library, as opposed to the contents of static libraries, which are copied and embedded in the executables that reference them. Second, a single copy of the `.text` section of a shared library in memory can be shared by different running processes. We will explore this in more detail when we study virtual memory in Chapter 9.

Figure 7.16 summarizes the dynamic linking process for the example program in Figure 7.7. To build a shared library `libvector.so` of our example vector routines in Figure 7.6, we invoke the compiler driver with some special directives to the compiler and linker:

```
linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c
```

The `-fpic` flag directs the compiler to generate *position-independent code* (more on this in the next section). The `-shared` flag directs the linker to create a shared

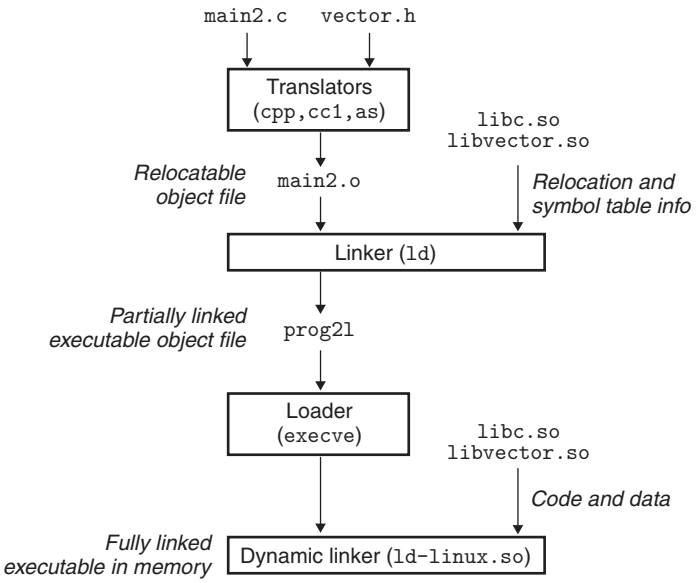


Figure 7.16
Dynamic linking with shared libraries.

object file. Once we have created the library, we would then link it into our example program in Figure 7.7:

```
linux> gcc -o prog2l main2.c ./libvector.so
```

This creates an executable object file `prog2l` in a form that can be linked with `libvector.so` at run time. The basic idea is to do some of the linking statically when the executable file is created, and then complete the linking process dynamically when the program is loaded. It is important to realize that none of the code or data sections from `libvector.so` are actually copied into the executable `prog2l` at this point. Instead, the linker copies some relocation and symbol table information that will allow references to code and data in `libvector.so` to be resolved at load time.

When the loader loads and runs the executable `prog2l`, it loads the partially linked executable `prog2l`, using the techniques discussed in Section 7.9. Next, it notices that `prog2l` contains a `.interp` section, which contains the path name of the dynamic linker, which is itself a shared object (e.g., `ld-linux.so` on Linux systems). Instead of passing control to the application, as it would normally do, the loader loads and runs the dynamic linker. The dynamic linker then finishes the linking task by performing the following relocations:

- Relocating the text and data of `libc.so` into some memory segment
- Relocating the text and data of `libvector.so` into another memory segment
- Relocating any references in `prog2l` to symbols defined by `libc.so` and `libvector.so`

Finally, the dynamic linker passes control to the application. From this point on, the locations of the shared libraries are fixed and do not change during execution of the program.

7.11 Loading and Linking Shared Libraries from Applications

Up to this point, we have discussed the scenario in which the dynamic linker loads and links shared libraries when an application is loaded, just before it executes. However, it is also possible for an application to request the dynamic linker to load and link arbitrary shared libraries while the application is running, without having to link in the applications against those libraries at compile time.

Dynamic linking is a powerful and useful technique. Here are some examples in the real world:

- *Distributing software.* Developers of Microsoft Windows applications frequently use shared libraries to distribute software updates. They generate a new copy of a shared library, which users can then download and use as a replacement for the current version. The next time they run their application, it will automatically link and load the new shared library.
- *Building high-performance Web servers.* Many Web servers generate *dynamic content*, such as personalized Web pages, account balances, and banner ads. Early Web servers generated dynamic content by using `fork` and `execve` to create a child process and run a "CGI program" in the context of the child. However, modern high-performance Web servers can generate dynamic content using a more efficient and sophisticated approach based on dynamic linking.

  The idea is to package each function that generates dynamic content in a shared library. When a request arrives from a Web browser, the server dynamically loads and links the appropriate function and then calls it directly, as opposed to using `fork` and `execve` to run the function in the context of a child process. The function remains cached in the server's address space, so subsequent requests can be handled at the cost of a simple function call. This can have a significant impact on the throughput of a busy site. Further, existing functions can be updated and new functions can be added at run time, without stopping the server.

Linux systems provide a simple interface to the dynamic linker that allows application programs to load and link shared libraries at run time.

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
                                    Returns: pointer to handle if OK, NULL on error
```

The dlopen function loads and links the shared library filename. The external symbols in filename are resolved using libraries previously opened with the RTLD_GLOBAL flag. If the current executable was compiled with the -rdynamic flag, then its global symbols are also available for symbol resolution. The flag argument must include either RTLD_NOW, which tells the linker to resolve references to external symbols immediately, or the RTLD_LAZY flag, which instructs the linker to defer symbol resolution until code from the library is executed. Either of these values can be ORed with the RTLD_GLOBAL flag.

```
#include <dlfcn.h>

void *dlsym(void *handle, char *symbol);
                                    Returns: pointer to symbol if OK, NULL on error
```

The dlsym function takes a handle to a previously opened shared library and a symbol name and returns the address of the symbol, if it exists, or NULL otherwise.

```
#include <dlfcn.h>

int dlclose (void *handle);
                                                    Returns: 0 if OK, −1 on error
```

The dlclose function unloads the shared library if no other shared libraries are still using it.

```
#include <dlfcn.h>

const char *dlerror(void);
            Returns: error message if previous call to dlopen, dlsym, or dlclose failed;
                                                    NULL if previous call was OK
```

The dlerror function returns a string describing the most recent error that occurred as a result of calling dlopen, dlsym, or dlclose, or NULL if no error occurred.

Figure 7.17 shows how we would use this interface to dynamically link our libvector.so shared library at run time and then invoke its addvec routine. To compile the program, we would invoke GCC in the following way:

```
linux> gcc -rdynamic -o prog2r dll.c -ldl
```

code/link/dll.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <dlfcn.h>
4
5  int x[2] = {1, 2};
6  int y[2] = {3, 4};
7  int z[2];
8
9  int main()
10 {
11     void *handle;
12     void (*addvec)(int *, int *, int *, int);
13     char *error;
14
15     /* Dynamically load the shared library containing addvec() */
16     handle = dlopen("./libvector.so", RTLD_LAZY);
17     if (!handle) {
18         fprintf(stderr, "%s\n", dlerror());
19         exit(1);
20     }
21
22     /* Get a pointer to the addvec() function we just loaded */
23     addvec = dlsym(handle, "addvec");
24     if ((error = dlerror()) != NULL) {
25         fprintf(stderr, "%s\n", error);
26         exit(1);
27     }
28
29     /* Now we can call addvec() just like any other function */
30     addvec(x, y, z, 2);
31     printf("z = [%d %d]\n", z[0], z[1]);
32
33     /* Unload the shared library */
34     if (dlclose(handle) < 0) {
35         fprintf(stderr, "%s\n", dlerror());
36         exit(1);
37     }
38     return 0;
39 }
```

code/link/dll.c

Figure 7.17 Example program 3. Dynamically loads and links the shared library `libvector.so` at run time.

Aside Shared libraries and the Java Native Interface

Java defines a standard calling convention called *Java Native Interface (JNI)* that allows "native" C and C++ functions to be called from Java programs. The basic idea of JNI is to compile the native C function, say, `foo`, into a shared library, say, `foo.so`. When a running Java program attempts to invoke function `foo`, the Java interpreter uses the `dlopen` interface (or something like it) to dynamically link and load `foo.so` and then call `foo`.

7.12 Position-Independent Code (PIC)

A key purpose of shared libraries is to allow multiple running processes to share the same library code in memory and thus save precious memory resources. So how can multiple processes share a single copy of a program? One approach would be to assign a priori a dedicated chunk of the address space to each shared library, and then require the loader to always load the shared library at that address. While straightforward, this approach creates some serious problems. It would be an inefficient use of the address space because portions of the space would be allocated even if a process didn't use the library. It would also be difficult to manage. We would have to ensure that none of the chunks overlapped. Each time a library was modified, we would have to make sure that it still fit in its assigned chunk. If not, then we would have to find a new chunk. And if we created a new library, we would have to find room for it. Over time, given the hundreds of libraries and versions of libraries in a system, it would be difficult to keep the address space from fragmenting into lots of small unused but unusable holes. Even worse, the assignment of libraries to memory would be different for each system, thus creating even more management headaches.

To avoid these problems, modern systems compile the code segments of shared modules so that they can be loaded anywhere in memory without having to be modified by the linker. With this approach, a single copy of a shared module's code segment can be shared by an unlimited number of processes. (Of course, each process will still get its own copy of the read/write data segment.)

Code that can be loaded without needing any relocations is known as *position-independent code (PIC)*. Users direct GNU compilation systems to generate PIC code with the `-fpic` option to GCC. Shared libraries must always be compiled with this option.

On x86-64 systems, references to symbols in the same executable object module require no special treatment to be PIC. These references can be compiled using PC-relative addressing and relocated by the static linker when it builds the object file. However, references to external procedures and global variables that are defined by shared modules require some special techniques, which we describe next.

PIC Data References

Compilers generate PIC references to global variables by exploiting the following interesting fact: no matter where we load an object module (including shared

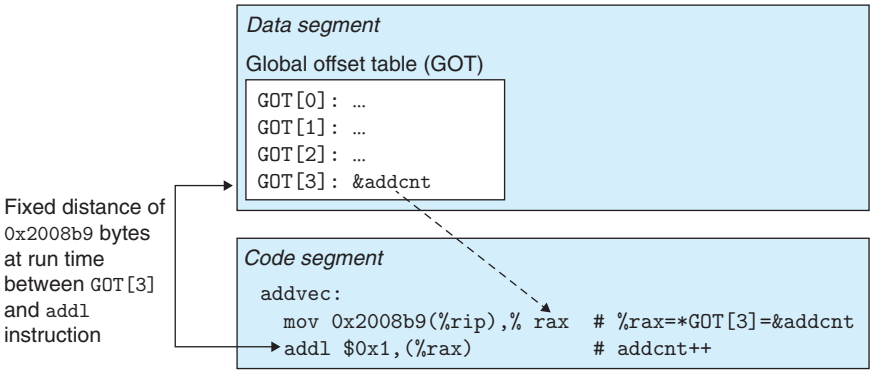


Figure 7.18 Using the GOT to reference a global variable. The `addvec` routine in `libvector.so` references `addcnt` indirectly through the GOT for `libvector.so`.

object modules) in memory, the data segment is always the same distance from the code segment. Thus, the *distance* between any instruction in the code segment and any variable in the data segment is a run-time constant, independent of the absolute memory locations of the code and data segments.

Compilers that want to generate PIC references to global variables exploit this fact by creating a table called the *global offset table (GOT)* at the beginning of the data segment. The GOT contains an 8-byte entry for each global data object (procedure or global variable) that is referenced by the object module. The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each GOT entry so that it contains the absolute address of the object. Each object module that references global objects has its own GOT.

Figure 7.18 shows the GOT from our example `libvector.so` shared module. The `addvec` routine loads the address of the global variable `addcnt` indirectly via `GOT[3]` and then increments `addcnt` in memory. The key idea here is that the offset in the PC-relative reference to `GOT[3]` is a run-time constant.

Since `addcnt` is defined by the `libvector.so` module, the compiler could have exploited the constant distance between the code and data segments by generating a direct PC-relative reference to `addcnt` and adding a relocation for the linker to resolve when it builds the shared module. However, if `addcnt` were defined by another shared module, then the indirect access through the GOT would be necessary. In this case, the compiler has chosen to use the most general solution, the GOT, for all references.

PIC Function Calls

Suppose that a program calls a function that is defined by a shared library. The compiler has no way of predicting the run-time address of the function, since the shared module that defines it could be loaded anywhere at run time. The normal approach would be to generate a relocation record for the reference, which

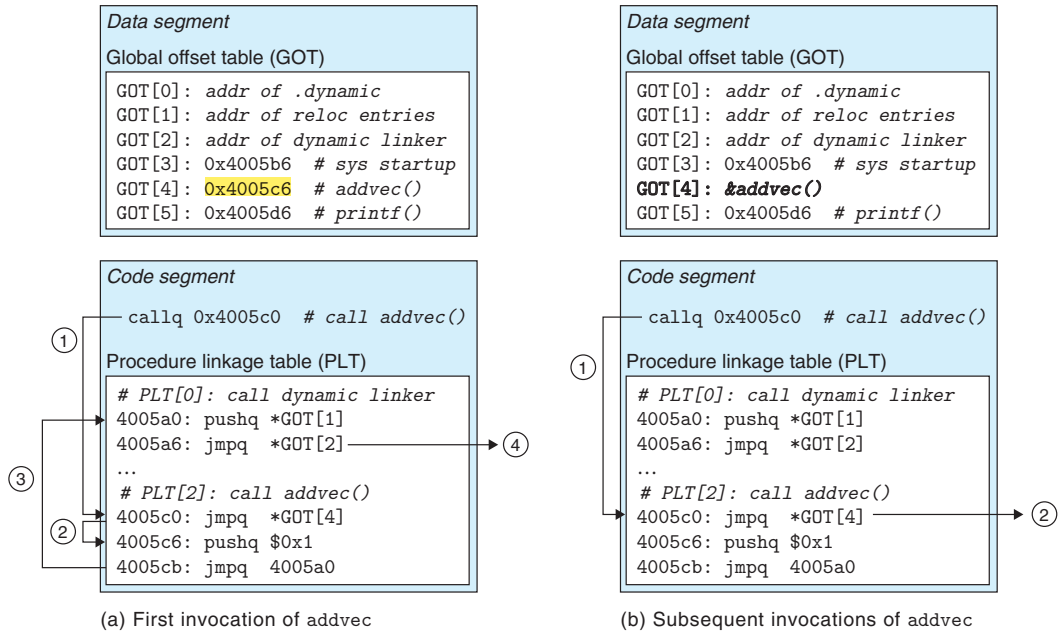
the dynamic linker could then resolve when the program was loaded. However, this approach would not be PIC, since it would require the linker to modify the code segment of the calling module. GNU compilation systems solve this problem using an interesting technique, called *lazy binding*, that defers the binding of each procedure address until the *first time* the procedure is called.

The motivation for lazy binding is that a typical application program will call only a handful of the hundreds or thousands of functions exported by a shared library such as `libc.so`. By deferring the resolution of a function's address until it is actually called, the dynamic linker can avoid hundreds or thousands of unnecessary relocations at load time. There is a nontrivial run-time overhead the first time the function is called, but each call thereafter costs only a single instruction and a memory reference for the indirection.

Lazy binding is implemented with a compact yet somewhat complex interaction between two data structures: the GOT and the *procedure linkage table (PLT)*. If an object module calls any functions that are defined in shared libraries, then it has its own GOT and PLT. The GOT is part of the data segment. The PLT is part of the code segment.

Figure 7.19 shows how the PLT and GOT work together to resolve the address of a function at run time. First, let's examine the contents of each of these tables.

Procedure linkage table (PLT). The PLT is an array of 16-byte code entries. `PLT[0]` is a special entry that jumps into the dynamic linker. Each shared library function called by the executable has its own PLT entry. Each of



(a) First invocation of `addvec` (b) Subsequent invocations of `addvec`

Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of `addvec` the first time it is called.

these entries is responsible for invoking a specific function. `PLT[1]` (not shown here) invokes the system startup function (`__libc_start_main`), which initializes the execution environment, calls the `main` function, and handles its return value. Entries starting at `PLT[2]` invoke functions called by the user code. In our example, `PLT[2]` invokes `addvec` and `PLT[3]` (not shown) invokes `printf`.

Global offset table (GOT). As we have seen, the GOT is an array of 8-byte address entries. When used in conjunction with the PLT, `GOT[0]` and `GOT[1]` contain information that the dynamic linker uses when it resolves function addresses. `GOT[2]` is the entry point for the dynamic linker in the `ld-linux.so` module. Each of the remaining entries corresponds to a called function whose address needs to be resolved at run time. Each has a matching PLT entry. For example, `GOT[4]` and `PLT[2]` correspond to `addvec`. Initially, each GOT entry points to the second instruction in the corresponding PLT entry.

Figure 7.19(a) shows how the GOT and PLT work together to lazily resolve the run-time address of function `addvec` the first time it is called:

Step 1. Instead of directly calling `addvec`, the program calls into `PLT[2]`, which is the PLT entry for `addvec`.

Step 2. The first PLT instruction does an indirect jump through `GOT[4]`. Since each GOT entry initially points to the second instruction in its corresponding PLT entry, the indirect jump simply transfers control back to the next instruction in `PLT[2]`.

Step 3. After pushing an ID for `addvec` (`0x1`) onto the stack, `PLT[2]` jumps to `PLT[0]`.

Step 4. `PLT[0]` pushes an argument for the dynamic linker indirectly through `GOT[1]` and then jumps into the dynamic linker indirectly through `GOT[2]`. The dynamic linker uses the two stack entries to determine the run-time location of `addvec`, overwrites `GOT[4]` with this address, and passes control to `addvec`.

Figure 7.19(b) shows the control flow for any subsequent invocations of `addvec`:

Step 1. Control passes to `PLT[2]` as before.

Step 2. However, this time the indirect jump through `GOT[4]` transfers control directly to `addvec`.

7.13 Library Interpositioning

Linux linkers support a powerful technique, called *library interpositioning*, that allows you to intercept calls to shared library functions and execute your own code instead. Using interpositioning, you could trace the number of times a particular

library function is called, validate and trace its input and output values, or even replace it with a completely different implementation.

Here's the basic idea: Given some *target function* to be interposed on, you create a *wrapper function* whose prototype is identical to the target function. Using some particular interpositioning mechanism, you then trick the system into calling the wrapper function instead of the target function. The wrapper function typically executes its own logic, then calls the target function and passes its return value back to the caller.

Interpositioning can occur at compile time, link time, or run time as the program is being loaded and executed. To explore these different mechanisms, we will use the example program in Figure 7.20(a) as a running example. It calls the `malloc` and `free` functions from the C standard library (`libc.so`). The call to `malloc` allocates a block of 32 bytes from the heap and returns a pointer to the block. The call to `free` gives the block back to the heap, for use by subsequent calls to `malloc`. Our goal is to use interpositioning to trace the calls to `malloc` and `free` as the program runs.

7.13.1 Compile-Time Interpositioning

Figure 7.20 shows how to use the C preprocessor to interpose at compile time. Each wrapper function in `mymalloc.c` (Figure 7.20(c)) calls the target function, prints a trace, and returns. The local `malloc.h` header file (Figure 7.20(b)) instructs the preprocessor to replace each call to a target function with a call to its wrapper. Here is how to compile and link the program:

```
linux> gcc -DCOMPILETIME -c mymalloc.c
linux> gcc -I. -o intc int.c mymalloc.o
```

The interpositioning happens because of the `-I.` argument, which tells the C preprocessor to look for `malloc.h` in the current directory before looking in the usual system directories. Notice that the wrappers in `mymalloc.c` are compiled with the standard `malloc.h` header file.

Running the program gives the following trace:

```
linux> ./intc
malloc(32)=0x9ee010
free(0x9ee010)
```

7.13.2 Link-Time Interpositioning

The Linux static linker supports link-time interpositioning with the `--wrap f` flag. This flag tells the linker to resolve references to symbol `f` as `__wrap_f` (two underscores for the prefix), and to resolve references to symbol `__real_f` (two underscores for the prefix) as `f`. Figure 7.21 shows the wrappers for our example program.

Here is how to compile the source files into relocatable object files:

```
linux> gcc -DLINKTIME -c mymalloc.c
linux> gcc -c int.c
```

(a) Example program `int.c`

code/link/interpose/int.c

```
1   #include <stdio.h>
2   #include <malloc.h>
3
4   int main()
5   {
6       int *p = malloc(32);
7       free(p);
8       return(0);
9   }
```

code/link/interpose/int.c

(b) Local `malloc.h` file

code/link/interpose/malloc.h

```
1   #define malloc(size) mymalloc(size)
2   #define free(ptr) myfree(ptr)
3
4   void *mymalloc(size_t size);
5   void myfree(void *ptr);
```

code/link/interpose/malloc.h

(c) Wrapper functions in `mymalloc.c`

code/link/interpose/mymalloc.c

```
1   #ifdef COMPILETIME
2   #include <stdio.h>
3   #include <malloc.h>
4
5   /* malloc wrapper function */
6   void *mymalloc(size_t size)
7   {
8       void *ptr = malloc(size);
9       printf("malloc(%d)=%p\n",
10              (int)size, ptr);
11      return ptr;
12  }
13
14  /* free wrapper function */
15  void myfree(void *ptr)
16  {
17      free(ptr);
18      printf("free(%p)\n", ptr);
19  }
20  #endif
```

code/link/interpose/mymalloc.c

Figure 7.20 Compile-time interpositioning with the C preprocessor.

code/link/interpose/mymalloc.c

```
1  #ifdef LINKTIME
2  #include <stdio.h>
3
4  void *__real_malloc(size_t size);
5  void __real_free(void *ptr);
6
7  /* malloc wrapper function */
8  void *__wrap_malloc(size_t size)
9  {
10     void *ptr = __real_malloc(size); /* Call libc malloc */
11     printf("malloc(%d) = %p\n", (int)size, ptr);
12     return ptr;
13 }
14
15 /* free wrapper function */
16 void __wrap_free(void *ptr)
17 {
18     __real_free(ptr); /* Call libc free */
19     printf("free(%p)\n", ptr);
20 }
21 #endif
```

code/link/interpose/mymalloc.c

Figure 7.21 **Link-time interpositioning with the `--wrap` flag.**

And here is how to link the object files into an executable:

```
linux> gcc -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
```

The `-Wl,option` flag passes `option` to the linker. Each comma in `option` is replaced with a space. So `-Wl,--wrap,malloc` passes `--wrap malloc` to the linker, and similarly for `-Wl,--wrap,free`.

Running the program gives the following trace:

```
linux> ./intl
malloc(32) = 0x18cf010
free(0x18cf010)
```

7.13.3 Run-Time Interpositioning

Compile-time interpositioning requires access to a program's source files. Link-time interpositioning requires access to its relocatable object files. However, there is a mechanism for interpositioning at run time that requires access only to the executable object file. This fascinating mechanism is based on the dynamic linker's `LD_PRELOAD` environment variable.

If the LD_PRELOAD environment variable is set to a list of shared library pathnames (separated by spaces or colons), then when you load and execute a program, the dynamic linker (LD-LINUX.SO) will search the LD_PRELOAD libraries first, before any other shared libraries, when it resolves undefined references. With this mechanism, you can interpose on any function in any shared library, including libc.so, when you load and execute any executable.

Figure 7.22 shows the wrappers for malloc and free. In each wrapper, the call to dlsym returns the pointer to the target libc function. The wrapper then calls the target function, prints a trace, and returns.

Here is how to build the shared library that contains the wrapper functions:

```
linux> gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
```

Here is how to compile the main program:

```
linux> gcc -o intr int.c
```

Here is how to run the program from the bash shell:[3]

```
linux> LD_PRELOAD="./mymalloc.so" ./intr
malloc(32) = 0x1bf7010
free(0x1bf7010)
```

And here is how to run it from the csh or tcsh shells:

```
linux> (setenv LD_PRELOAD "./mymalloc.so"; ./intr; unsetenv LD_PRELOAD)
malloc(32) = 0x2157010
free(0x2157010)
```

Notice that you can use LD_PRELOAD to interpose on the library calls of *any* executable program!

```
linux> LD_PRELOAD="./mymalloc.so" /usr/bin/uptime
malloc(568) = 0x21bb010
free(0x21bb010)
malloc(15) = 0x21bb010
malloc(568) = 0x21bb030
malloc(2255) = 0x21bb270
free(0x21bb030)
malloc(20) = 0x21bb030
malloc(20) = 0x21bb050
malloc(20) = 0x21bb070
malloc(20) = 0x21bb090
malloc(20) = 0x21bb0b0
malloc(384) = 0x21bb0d0
 20:47:36 up 85 days,  6:04,  1 user,  load average: 0.10, 0.04, 0.05
```

3. If you don't know what shell you are running, type printenv SHELL at the command line.

code/link/interpose/mymalloc.c

```
1  #ifdef RUNTIME
2  #define _GNU_SOURCE
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <dlfcn.h>
6
7  /* malloc wrapper function */
8  void *malloc(size_t size)
9  {
10     void *(*mallocp)(size_t size);
11     char *error;
12
13     mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get address of libc malloc */
14     if ((error = dlerror()) != NULL) {
15         fputs(error, stderr);
16         exit(1);
17     }
18     char *ptr = mallocp(size); /* Call libc malloc */
19     printf("malloc(%d) = %p\n", (int)size, ptr);
20     return ptr;
21 }
22
23 /* free wrapper function */
24 void free(void *ptr)
25 {
26     void (*freep)(void *) = NULL;
27     char *error;
28
29     if (!ptr)
30         return;
31
32     freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
33     if ((error = dlerror()) != NULL) {
34         fputs(error, stderr);
35         exit(1);
36     }
37     freep(ptr); /* Call libc free */
38     printf("free(%p)\n", ptr);
39 }
40 #endif
```

code/link/interpose/mymalloc.c

Figure 7.22 **Run-time interpositioning with** `LD_PRELOAD`.

7.14 Tools for Manipulating Object Files

There are a number of tools available on Linux systems to help you understand and manipulate object files. In particular, the GNU *binutils* package is especially helpful and runs on every Linux platform.

AR. Creates static libraries, and inserts, deletes, lists, and extracts members.

STRINGS. Lists all of the printable strings contained in an object file.

STRIP. Deletes symbol table information from an object file.

NM. Lists the symbols defined in the symbol table of an object file.

SIZE. Lists the names and sizes of the sections in an object file.

READELF. Displays the complete structure of an object file, including all of the information encoded in the ELF header. Subsumes the functionality of SIZE and NM.

OBJDUMP. The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the `.text` section.

Linux systems also provide the LDD program for manipulating shared libraries:

LDD: Lists the shared libraries that an executable needs at run time.

7.15 Summary

Linking can be performed at compile time by static linkers and at load time and run time by dynamic linkers. Linkers manipulate binary files called object files, which come in three different forms: relocatable, executable, and shared. Relocatable object files are combined by static linkers into an executable object file that can be loaded into memory and executed. Shared object files (shared libraries) are linked and loaded by dynamic linkers at run time, either implicitly when the calling program is loaded and begins executing, or on demand, when the program calls functions from the `dlopen` library.

The two main tasks of linkers are symbol resolution, where each global symbol in an object file is bound to a unique definition, and relocation, where the ultimate memory address for each symbol is determined and where references to those objects are modified.

Static linkers are invoked by compiler drivers such as GCC. They combine multiple relocatable object files into a single executable object file. Multiple object files can define the same symbol, and the rules that linkers use for silently resolving these multiple definitions can introduce subtle bugs in user programs.

Multiple object files can be concatenated in a single static library. Linkers use libraries to resolve symbol references in other object modules. The left-to-right sequential scan that many linkers use to resolve symbol references is another source of confusing link-time errors.

Loaders map the contents of executable files into memory and run the program. Linkers can also produce partially linked executable object files with unresolved references to the routines and data defined in a shared library. At load time, the loader maps the partially linked executable into memory and then calls a dynamic linker, which completes the linking task by loading the shared library and relocating the references in the program.

Shared libraries that are compiled as position-independent code can be loaded anywhere and shared at run time by multiple processes. Applications can also use the dynamic linker at run time in order to load, link, and access the functions and data in shared libraries.

Bibliographic Notes

Linking is poorly documented in the computer systems literature. Since it lies at the intersection of compilers, computer architecture, and operating systems, linking requires an understanding of code generation, machine-language programming, program instantiation, and virtual memory. It does not fit neatly into any of the usual computer systems specialties and thus is not well covered by the classic texts in these areas. However, Levine's monograph provides a good general reference on the subject [69]. The original IA32 specifications for ELF and DWARF (a specification for the contents of the `.debug` and `.line` sections) are described in [54]. The x86-64 extensions to the ELF file format are described in [36]. The x86-64 application binary interface (ABI) describes the conventions for compiling, linking, and running x86-64 programs, including the rules for relocation and position-independent code [77].

Homework Problems

7.6 ◆

This problem concerns the `m.o` module from Figure 7.5 and the following version of the `swap.c` function that counts the number of times it has been called:

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   static int *bufp1;
5
6   static void incr()
7   {
8       static int count=0;
9
10      count++;
11  }
12
13  void swap()
14  {
```

```
15      int temp;
16
17      incr();
18      bufp1 = &buf[1];
19      temp = *bufp0;
20      *bufp0 = *bufp1;
21      *bufp1 = temp;
22  }
```

For each symbol that is defined and referenced in `swap.o`, indicate if it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the module that defines the symbol (`swap.o` or `m.o`), the symbol type (local, global, or extern), and the section (`.text`, `.data`, or `.bss`) it occupies in that module.

Symbol	`swap.o .symtab` entry?	Symbol type	Module where defined	Section
`buf`	_____	_____	_____	_____
`bufp0`	_____	_____	_____	_____
`bufp1`	_____	_____	_____	_____
`swap`	_____	_____	_____	_____
`temp`	_____	_____	_____	_____
`incr`	_____	_____	_____	_____
`count`	_____	_____	_____	_____

7.7 ◆

Without changing any variable names, modify `bar5.c` on page 719 so that `foo5.c` prints the correct values of `x` and `y` (i.e., the hex representations of integers 15213 and 15212).

7.8 ◆

In this problem, let `REF(x.i)` → `DEF(x.k)` denote that the linker will associate an arbitrary reference to symbol `x` in module `i` to the definition of `x` in module `k`. For each example below, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN".

A.
```
/* Module 1 */
int main()
{
}
```
```
/* Module 2 */
static int main=1[
int p2()
{
}
```

(a) `REF(main.1)` → `DEF(`_____`.`_____`)`

(b) `REF(main.2)` → `DEF(`_____`.`_____`)`

B.
```
/* Module 1 */          /* Module 2 */
int x;                  double x;
void main()             int p2()
{                       {
}                       }
```

(a) REF(x.1) → DEF(_____._____)
(b) REF(x.2) → DEF(_____._____)

C.
```
/* Module 1 */          /* Module 2 */
int x=1;                double x=1.0;
void main()             int p2()
{                       {
}                       }
```

(a) REF(x.1) → DEF(_____._____)
(b) REF(x.2) → DEF(_____._____)

7.9 ◆

Consider the following program, which consists of two object modules:

```
1  /* foo6.c */
2  void p2(void);
3
4  int main()
5  {
6      p2();
7      return 0;
8  }
```

```
1  /* bar6.c */
2  #include <stdio.h>
3
4  char main;
5
6  void p2()
7  {
8      printf("0x%x\n", main);
9  }
```

When this program is compiled and executed on an x86-64 Linux system, it prints the string `0x48\n` and terminates normally, even though function `p2` never initializes variable `main`. Can you explain this?

7.10 ◆◆

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is

referenced by a. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references:

A. `p.o` → `libx.a` → `p.o`

B. `p.o` → `libx.a` → `liby.a` *and* `liby.a` → `libx.a`

C. `p.o` → `libx.a` → `liby.a` → `libz.a` *and* `liby.a` → `libx.a` → `libz.a`

7.11 ◆◆

The program header in Figure 7.14 indicates that the data segment occupies `0x230` bytes in memory. However, only the first `0x228` bytes of these come from the sections of the executable file. What causes this discrepancy?

7.12 ◆◆

Consider the call to function `swap` in object file `m.o` (Problem 7.6).

```
9:   e8 00 00 00 00          callq  e <main+0xe>      swap()
```

with the following relocation entry:

```
r.offset = 0xa
r.symbol = swap
r.type   = R_X86_64_PC32
r.addend = -4
```

A. Suppose that the linker relocates `.text` in `m.o` to address `0x4004e0` and `swap` to address `0x4004f8`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

B. Suppose that the linker relocates `.text` in `m.o` to address `0x4004d0` and `swap` to address `0x400500`. Then what is the value of the relocated reference to `swap` in the `callq` instruction?

7.13 ◆◆

Performing the following tasks will help you become more familiar with the various tools for manipulating object files.

A. How many object files are contained in the versions of `libc.a` and `libm.a` on your system?

B. Does `gcc -Og` produce different executable code than `gcc -Og -g`?

C. What shared libraries does the `gcc` driver on your system use?

Solutions to Practice Problems

Solution to Problem 7.1 (page 714)

The purpose of this problem is to help you understand the relationship between linker symbols and C variables and functions. Notice that the C local variable `temp` does *not* have a symbol table entry.

Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf	Yes	extern	m.o	.data
bufp0	Yes	global	swap.o	.data
bufp1	Yes	global	swap.o	COMMON
swap	Yes	global	swap.o	.text
temp	No	—	—	—

Solution to Problem 7.2 (page 720)

This is a simple drill that checks your understanding of the rules that a Unix linker uses when it resolves global symbols that are defined in more than one module. Understanding these rules can help you avoid some nasty programming bugs.

A. The linker chooses the strong symbol defined in module 1 over the weak symbol defined in module 2 (rule 2):
   (a) REF(main.1) → DEF(main.1)
   (b) REF(main.2) → DEF(main.1)

B. This is an ERROR, because each module defines a strong symbol `main` (rule 1).

C. The linker chooses the strong symbol defined in module 2 over the weak symbol defined in module 1 (rule 2):
   (a) REF(x.1) → DEF(x.2)
   (b) REF(x.2) → DEF(x.2)

Solution to Problem 7.3 (page 725)

Placing static libraries in the wrong order on the command line is a common source of linker errors that confuses many programmers. However, once you understand how linkers use static libraries to resolve references, it's pretty straightforward. This little drill checks your understanding of this idea:

A. `linux> gcc p.o libx.a`

B. `linux> gcc p.o libx.a liby.a`

C. `linux> gcc p.o libx.a liby.a libx.a`

Solution to Problem 7.4 (page 730)

This problem concerns the disassembly listing in Figure 7.12(a). Our purpose here is to give you some practice reading disassembly listings and to check your understanding of PC-relative addressing.

A. The hex address of the relocated reference in line 5 is `0x4004df`.

B. The hex value of the relocated reference in line 5 is `0x5`. Remember that the disassembly listing shows the value of the reference in little-endian byte order.

Solution to Problem 7.5 (page 731)

This problem tests your understanding of how the linker relocates PC-relative references. You were given that

```
ADDR(s) = ADDR(.text) = 0x4004d0
```

and

```
ADDR(r.symbol) = ADDR(swap) = 0x4004e8
```

Using the algorithm in Figure 7.10, the linker first computes the run-time address of the reference:

```
refaddr = ADDR(s)  + r.offset
        = 0x4004d0 + 0xa
        = 0x4004da
```

It then updates the reference:

```
*refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr)
        = (unsigned) (0x4004e8       + (-4)     - 0x4004da)
        = (unsigned) (0xa)
```

Thus, in the resulting executable object file, the PC-relative reference to `swap` has a value of `0xa`:

```
  4004d9:  e8 0a 00 00 00          callq  4004e8 <swap>
```