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 Unix linkers make when they resolve symbol references can silently affect the correctness of your programs. Programs that incorrectly define multiple global variables 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, most Web servers rely on dynamic linking of shared libraries to serve dynamic content.

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 IA32 machine running a version of Unix, such as Linux or Solaris, that uses the standard 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 consists of two source files, `main.c` and `swap.c`. Function `main()` calls `swap`, which swaps the two elements in the external global array `buf`. Granted, this is a strange way to swap two numbers, but it will serve as a small running example throughout this chapter that will allow us to make some important points about how linking works.

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:

```
unix> gcc -O2 -g -o p main.c swap.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`), which translates the C source file `main.c` into an ASCII intermediate file `main.i`:

```
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 main.c -O2 [other arguments] -o /tmp/main.s
```

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

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

code/link/main.c

```
1 /* main.c */
2 void swap();
3
4 int buf[2] = {1, 2};
5
6 int main()
7 {
8     swap();
9     return 0;
10 }
```

code/link/main.c

(a) `main.c`

code/link/swap.c

```
1 /* swap.c */
2 extern int buf[];
3
4 int *bufp0 = &buf[0];
5 int *bufp1;
6
7 void swap()
8 {
9     int temp;
10
11     bufp1 = &buf[1];
12     temp = *bufp0;
13     *bufp0 = *bufp1;
14     *bufp1 = temp;
15 }
```

code/link/swap.c

(b) `swap.c`

Figure 7.1: **Example program 1:** The example program consists of two source files, `main.c` and `swap.c`. The `main` function initializes a two-element array of ints, and then calls the `swap` function to swap the pair.

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

```
ld -o p [system object files and args] /tmp/main.o /tmp/swap.o
```

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

```
unix> ./p
```

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

7.2 Static Linking

Static linkers such as the Unix `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. 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:

- *Symbol resolution.* Object files define and reference *symbols*. The purpose of symbol resolution is to associate each symbol reference with exactly one symbol definition.

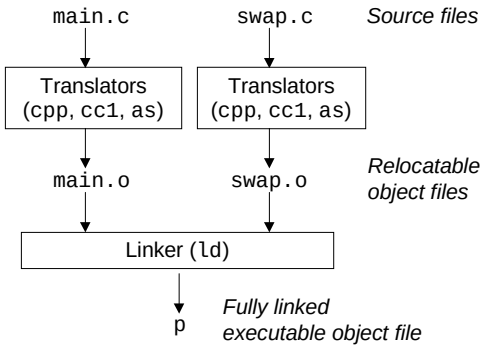

Figure 7.2: **Static linking.** The linker combines relocatable object files to form an executable object file `p`.

- *Relocation.* Compilers and assemblers generate code and data sections that start at address zero. 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 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 file formats 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.) Early versions of System V Unix used the Common Object File format (COFF). Windows NT uses a variant of COFF called the Portable

Executable (PE) format. Modern Unix systems — such as Linux, later versions of System V Unix, BSD Unix variants, and Sun Solaris — use the Unix *Executable and Linkable Format (ELF)*. Although our discussion will focus on ELF, the basic concepts are similar, regardless of the particular format.

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., IA32) 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 sized entry for each section in the object file.

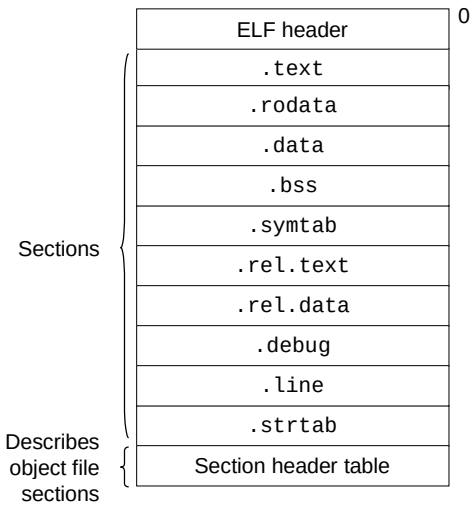

Figure 7.3: **Typical ELF relocatable object file.**

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 (see Problem 7.14).

`.data`: *Initialized* global 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 C variables. This section occupies no actual space in the object file; it is merely a place holder. 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.

`.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`. 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.

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 Storage Start" instruction 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!". **End Aside.**

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 that are defined *without* the C `static` attribute.
- Global symbols that are referenced by module m but defined by some other module. Such symbols are called *externals* and correspond to C functions and variables that are defined in other modules.
- *Local symbols* that are defined and referenced exclusively by module m. Some local linker symbols correspond to 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. The sections in an object file and the name of the source file that corresponds module m also get local symbols.

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 allocates space for two integers in `.bss` and exports a pair of unique local linker symbols 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`.

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++. 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. **End.**

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 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.

code/link/elfstructs.c

```
1 typedef struct {
2     int name;        /* string table offset */
3     int value;       /* section offset, or VM address */
4     int size;        /* object size in bytes */
5     char type:4,     /* data, func, section, or src file name (4 bits) */
6          binding:4;  /* local or global (4 bits) */
7     char reserved;   /* unused */
8     char section;    /* section header index, ABS, UNDEF, */
9                      /* or COMMON  */
10 } Elf_Symbol;
```

code/link/elfstructs.c

Figure 7.4: **ELF symbol table entry.** `type` and `binding` are four bits each.

Each symbol is associated with some section of the object file, denoted by the `section` field, which is an index into the section header table. There are three special pseudo sections 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.

For example, here are the last three entries in the symbol table for `main.o`, as displayed by the GNU READELF tool. The first eight entries, which are not shown, are local symbols that the linker uses internally.

```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     8 OBJECT  GLOBAL  0    3 buf
  9:        0    17 FUNC    GLOBAL  0    1 main
 10:        0     0 NOTYPE  GLOBAL  0  UND swap
```

In this example, we see an entry for the definition of global symbol `buf`, an 8-byte object located at an offset (i.e., `value`) of zero in the `.data` section. This is followed by the definition of the global symbol `main`, a 17-byte function located at an offset of zero in the `.text` section. The last entry comes from the reference for the external symbol `swap`. READELF identifies each section by an integer index. `Ndx=1` denotes the `.text` section, and `Ndx=3` denotes the `.data` section.

Similarly, here are the symbol table entries for `swap.o`:

```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     4 OBJECT  GLOBAL  0    3 bufp0
  9:        0     0 NOTYPE  GLOBAL  0  UND buf
 10:        0    39 FUNC    GLOBAL  0    1 swap
 11:        4     4 OBJECT  GLOBAL  0  COM bufp1
```

First, we see an entry for the definition of the global symbol `bufp0`, which is a 4-byte initialized object starting at offset 0 in `.data`. The next symbol comes from the reference to the external `buf` symbol in the

initialization code for `bufp0`. This is followed by the global symbol `swap`, a 39-byte function at an offset of 0 in `.text`. The last entry is the global symbol `bufp1`, a 4-byte uninitialized data object (with a 4-byte alignment requirement) that will eventually be allocated as a `.bss` object when this module is linked.

Practice Problem 7.1:

This problem concerns the `swap.o` module from Figure 7.1(b). 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 `main.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`				

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`:

```
unix> gcc -Wall -O2 -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function ‘main’:
/tmp/ccSz5uti.o(.text+0x7): undefined reference to ‘foo’
collect2: ld returned 1 exit status
```

Symbol resolution for global symbols is also tricky because the same symbol might be defined by multiple object files. 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 Unix 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 *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. **End Aside.**

7.6.1 How Linkers Resolve Multiply Defined Global Symbols

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. For the example program in Figure 7.1, `buf`, `bufp0`, `main`, and `swap` are strong symbols; `bufp1` is a weak symbol.

Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiply defined symbols:

- Rule 1: Multiple strong symbols are not allowed.
- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.
- Rule 3: Given multiple weak symbols, 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):

```
unix> gcc foo1.c bar1.c
/tmp/cca015022.o: In function ‘main’:
/tmp/cca015022.o(.text+0x0): multiple definition of ‘main’
/tmp/cca015021.o(.text+0x0): first defined here
```

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

```
/* foo2.c */
int x = 15213;

int main()
{
    return 0;
}
```

```
/* bar2.c */
int x = 15213;

void f()
{
}
```

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

```
/* foo3.c */
#include <stdio.h>
void f(void);

int x = 15213;

int main()
{
    f();
    printf("x = %d\n", x);
    return 0;
}
```

```
/* bar3.c */
int x;

void f()
{
    x = 15212;
}
```

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

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./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 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 x = 15213;
6 int y = 15212;
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 IA32/Linux machine, `doubles` are 8 bytes and `ints` are 4 bytes. 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 one!

```
linux> gcc -o foobar5 foo5.c bar5.c
linux> ./foobar5
x = 0x0 y = 0x80000000
```

This is a subtle and nasty bug, especially because it occurs silently, with no warning from the compilation system, 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. When in doubt, invoke the linker with a flag such as the GCC `-warn-common` flag, which instructs it to print a warning message when it resolves multiply defined global symbol definitions.

Practice Problem 7.2:

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 ANSI C, which defines an extensive collection of standard I/O, string manipulation, and integer math functions such as `atoi`, `printf`, `scanf`, `strcpy`, and `random`. They are available to every C program in the `libc.a` library. ANSI C 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:

```
unix> 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 a typical system, `libc.a` is about 8 MB and `libm.a` is about 1 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:

```
unix> 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 file name on the command line. For example, a program that uses functions from the standard C library and the math library could be compiled and linked with a command of the form

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

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 Unix 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, suppose that we want to provide the vector routines in Figure 7.5 in a static library called `libvector.a`.

To create the library, we would use the AR tool as follows:

code/link/addvec.c

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

code/link/addvec.c

(a) `addvec.o`

code/link/multvec.c

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

code/link/multvec.c

(a) `multvec.o`

Figure 7.5: **Member object files in `libvector.a`.**

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

To use the library, we might write an application such as `main2.c` in Figure 7.6, which invokes the `addvec` library routine. (The include (header) file `vector.h` defines the function prototypes for the routines in `libvector.a`.)

code/link/main2.c

```
1 /* main2.c */
2 #include <stdio.h>
3 #include "vector.h"
4
5 int x[2] = {1, 2};
6 int y[2] = {3, 4};
7 int z[2];
8
9 int main()
10 {
11     addvec(x, y, z, 2);
12     printf("z = [%d %d]\n", z[0], z[1]);
13     return 0;
14 }
```

code/link/main2.c

Figure 7.6: **Example program 2:** This program calls member functions in the static `libvector.a` library.

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

```
unix> gcc -O2 -c main2.c
unix> gcc -static -o p2 main2.o ./libvector.a
```

Figure 7.7 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. When the linker runs, it determines that the `addvec` symbol defined by `addvec.o` is referenced by `main.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.

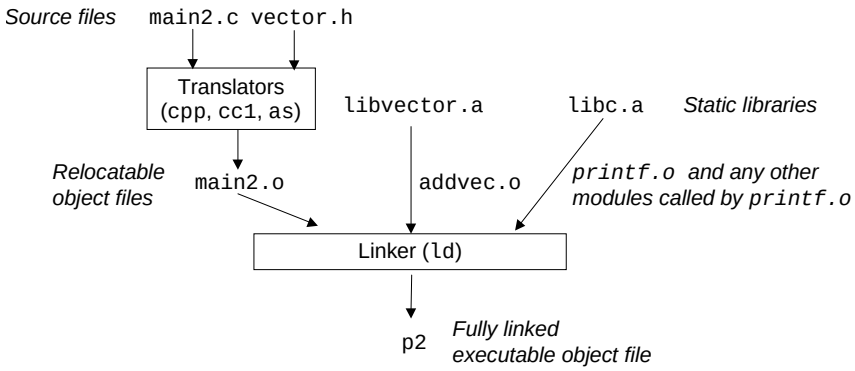

Figure 7.7: **Linking with static libraries.**

7.6.3 How Linkers Use Static Libraries to Resolve References

While static libraries are useful and essential tools, they are also a source of confusion to programmers because of the way the Unix 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:

```
unix> 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:

```
unix> 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:

```
unix> 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:

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 file 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:

- *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, every instruction and global variable in the program has a unique run-time memory address.
- *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 `.relo.text`. Relocation entries for initialized data are placed in `.relo.data`.

Figure 7.8 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 the modify the new reference.

code/link/elfstructs.c

```
1 typedef struct {
2     int offset;     /* offset of the reference to relocate */
3     int symbol:24,  /* symbol the reference should point to */
4         type:8;     /* relocation type */
5 } Elf32_Rel;
```

code/link/elfstructs.c

Figure 7.8: **ELF relocation entry.** Each entry identifies a reference that must be relocated.

ELF defines 11 different relocation types, some quite arcane. We are concerned with only the two most basic relocation types:

- `R_386_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 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_386_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.

7.7.2 Relocating Symbol References

Figure 7.9 shows the pseudo code for the linker's relocation algorithm.

```
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_386_PC32) {
7              refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8              *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
9          }
10
11         /* relocate an absolute reference */
12         if (r.type == R_386_32)
13             *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
14     }
15 }
```

Figure 7.9: **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 `Elf32_Rel`, as defined in Figure 7.8. 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.

Relocating PC-Relative References

Recall from our running example in Figure 7.1(a) that the `main` routine in the `.text` section of `main.o` calls the `swap` routine, which is defined in `swap.o`. Here is the disassembled listing for the `call` instruction, as generated by the GNU OBJDUMP tool:

```
6:   e8 fc ff ff ff          call   7 <main+0x7>    swap();
                             7: R_386_PC32 swap     relocation entry
```

From this listing we see that the `call` instruction begins at section offset `0x6` and consists of the 1-byte opcode `0xe8`, followed by the 32-bit reference `0xfffffffc` (-4 decimal), which is stored in little-endian byte order. We also see a relocation entry for this reference displayed on the following line. (Recall that relocation entries and instructions are actually stored in different sections of the object file. The OBJDUMP tool displays them together for convenience.) The relocation entry `r` consists of three fields:

```
r.offset = 0x7
r.symbol = swap
r.type   = R_386_PC32
```

These fields tell the linker to modify the 32-bit PC-relative reference starting at offset `0x7` so that it will point to the `swap` routine at run time. Now, suppose that the linker has determined that

```
ADDR(s) = ADDR(.text) = 0x80483b4
```

and

```
ADDR(r.symbol) = ADDR(swap) = 0x80483c8.
```

Using the algorithm in Figure 7.9, the linker first computes the run-time address of the reference (line 7):

```
refaddr = ADDR(s)   + r.offset
        = 0x80483b4 + 0x7
        = 0x80483bb
```

It then updates the reference from its current value (-4) to `0x9` so that it will point to the `swap` routine at run time (line 8):

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr)
        = (unsigned) (0x80483c8      + (-4)    - 0x80483bb)
        = (unsigned) (0x9)
```

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

```
80483ba:  e8 09 00 00 00          call   80483c8 <swap>          swap();
```

At run time, the `call` instruction will be stored at address `0x80483ba`. When the CPU executes the `call` instruction, the PC has a value of `0x80483bf`, which is the address of the instruction immediately following the `call` instruction. To execute the instruction, the CPU performs the following steps:

```
1. push PC onto stack
2. PC <- PC + 0x9 = 0x80483bf + 0x9 = 0x80483c8
```

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

You may wonder why the assembler created the reference in the `call` instruction with an initial value of -4. The assembler uses this value as a bias to account for the fact that the PC always points to the instruction following the current instruction. On a different machine with different instruction sizes and encodings, the assembler for that machine would use a different bias. This is powerful trick that allows the linker to blindly relocate references, blissfully unaware of the instruction encodings for a particular machine.

Relocating Absolute References

Recall that in our example program in Figure 7.1, the `swap.o` module initializes the global pointer `bufp0` to the address of the first element of the global `buf` array:

```
int *bufp0 = &buf[0];
```

Since `bufp0` is an initialized data object, it will be stored in the `.data` section of the `swap.o` relocatable object module. Since it is initialized to the address of a global array, it will need to be relocated. Here is the disassembled listing of the `.data` section from `swap.o`:

```
00000000 <bufp0>:
   0:   00 00 00 00                                    int *bufp0 = &buf[0];
                        0: R_386_32 buf                relocation entry
```

We see that the `.data` section contains a single 32-bit reference, the `bufp0` pointer, which has a value of `0x0`. The relocation entry tells the linker that this is a 32-bit absolute reference, beginning at offset 0, which must be relocated so that it points to the symbol `buf`. Now, suppose that the linker has determined that

```
ADDR(r.symbol) = ADDR(buf) = 0x8049454
```

The linker updates the reference using line 13 of the algorithm in Figure 7.9:

```
*refptr = (unsigned) (ADDR(r.symbol) + *refptr)
        = (unsigned) (0x8049454      + 0)
        = (unsigned) (0x8049454)
```

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

```
0804945c <bufp0>:
 804945c:  54 94 04 08                          Relocated!
```

In words, the linker has decided that at run time, the variable `bufp0` will be located at memory address `0x804945c` and will be initialized to `0x8049454`, which is the run-time address of the `buf` array.

The `.text` section in the `swap.o` module contains five absolute references that are relocated in a similar way (See Problem 7.12). Figure 7.10 shows the relocated `.text` and `.data` sections in the final executable object file.

code/link/p-exe.d

```
080483b4 <main>:
 80483b4:  55                      push   %ebp
 80483b5:  89 e5                   mov    %esp,%ebp
 80483b7:  83 ec 08                sub    $0x8,%esp
 80483ba:  e8 09 00 00 00          call   80483c8 <swap>          swap();
 80483bf:  31 c0                   xor    %eax,%eax
 80483c1:  89 ec                   mov    %ebp,%esp
 80483c3:  5d                      pop    %ebp
 80483c4:  c3                      ret
 80483c5:  90                      nop
 80483c6:  90                      nop
 80483c7:  90                      nop

080483c8 <swap>:
 80483c8:  55                      push   %ebp
 80483c9:  8b 15 5c 94 04 08       mov    0x804945c,%edx          Get *bufp0
 80483cf:  a1 58 94 04 08          mov    0x8049458,%eax          Get buf[1]
 80483d4:  89 e5                   mov    %esp,%ebp
 80483d6:  c7 05 48 95 04 08 58    movl   $0x8049458,0x8049548    bufp1 = &buf[1]
 80483dd:  94 04 08
 80483e0:  89 ec                   mov    %ebp,%esp
 80483e2:  8b 0a                   mov    (%edx),%ecx
 80483e4:  89 02                   mov    %eax,(%edx)
 80483e6:  a1 48 95 04 08          mov    0x8049548,%eax          Get *bufp1
 80483eb:  89 08                   mov    %ecx,(%eax)
 80483ed:  5d                      pop    %ebp
 80483ee:  c3                      ret
```

code/link/p-exe.d

(a) Relocated `.text` section.

code/link/pdata-exe.d

```
08049454 <buf>:
 8049454:  01 00 00 00 02 00 00 00

0804945c <bufp0>:
 804945c:  54 94 04 08                    Relocated!
```

code/link/pdata-exe.d

(b) Relocated `.data` section.

Figure 7.10: **Relocated `.text` and `data` sections for executable file p** The original C code is in Figure 7.1.

Practice Problem 7.4:

This problem concerns the relocated program in Figure 7.10.

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

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

C. Suppose the linker had decided for some reason to locate the `.text` section at `0x80483b8` instead of `0x80483b4`. What would the hex value of the relocated reference in line 5 be in this case?

7.8 Executable Object Files

We have seen how the linker merges multiple object modules into a single executable object file. Our 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 7.11 summarizes the kinds of information in a typical ELF executable file.

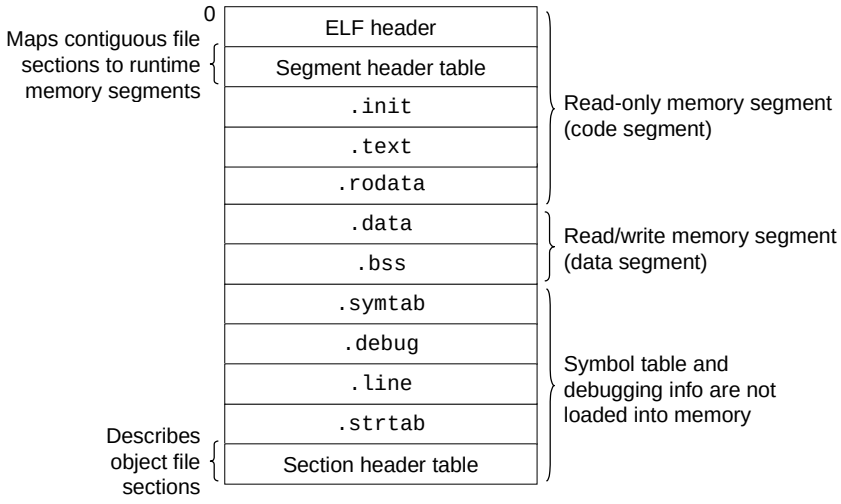

Figure 7.11: **Typical ELF executable object file**

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 `.relo` 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 *segment header table*. Figure 7.12 shows the segment header table for our example executable `p`, as displayed by OBJDUMP.