Chapter 1
Linking and Loading

$Revision: 2.3 $
$Date: 1999/06/30 01:02:35 $

What do linkers and loaders do?

The basic job of any linker or loader is simple: it binds more abstract *
names to more concrete names, which permits programmers to write code *
using the more abstract names. That is, it takes a name written by a pro- *
grammer such as `getline` and binds it to ‘‘the location 612 bytes from *
the beginning of the executable code in module `iosys`.’’ Or it may take a *
more abstract numeric address such as ‘‘the location 450 bytes beyond the *
beginning of the static data for this module’’ and bind it to a numeric ad- *
dress. *

Address binding: a historical perspective

A useful way to get some insight into what linkers and loaders do is to look at their part in the development of computer programming systems.

The earliest computers were programmed entirely in machine language. Programmers would write out the symbolic programs on sheets of paper, hand assemble them into machine code and then toggle the machine code into the computer, or perhaps punch it on paper tape or cards. (Real hotshots could compose code directly at the switches.) If the programmer used symbolic addresses at all, the symbols were bound to addresses as the programmer did his or her hand translation. If it turned out that an instruction had to be added or deleted, the entire program had to be hand-inspected and any addresses affected by the added or deleted instruction adjusted.

The problem was that the names were bound to addresses too early. Assemblers solved that problem by letting programmers write programs in terms of symbolic names, with the assembler binding the names to machine addresses. If the program changed, the programmer had to reassemble it, but the work of assigning the addresses is pushed off from the programmer to the computer.

Libraries of code compound the address assignment problem. Since the basic operations that computers can perform are so simple, useful programs are composed of subprograms that perform higher level and more complex operations. computer installations keep a library of pre-written and debugged subprograms that programmers can draw upon to use in new programs they write, rather than requiring programmers to write all their own subprograms. The programmer then loads the subprograms in with the main program to form a complete working program.

Programmers were using libraries of subprograms even before they used assemblers. By 1947, John Mauchly, who led the ENIAC project, wrote about loading programs along with subprograms selected from a catalog of programs stored on tapes, and of the need to relocate the subprograms' code to reflect the addresses at which they were loaded. Perhaps surprisingly, these two basic linker functions, relocation and library search, appear to predate even assemblers, as Mauchly expected both the program and subprograms to be written in machine language. The relocating loader allowed the authors and users of the subprograms to write each subprogram as though it would start at location zero, and to defer the actual address binding until the subprograms were linked with a particular main program.

With the advent of operating systems, relocating loaders separate from linkers and libraries became necessary. Before operating systems, each program had the machine's entire memory at its disposal, so the program could be assembled and linked for fixed memory addresses, knowing that all addresses in the computer would be available. But with operating systems, the program had to share the computer's memory with the operating system and perhaps even with other programs, This means that the actual addresses at which the program would be running weren't known until the operating system loaded the program into memory, deferring final address binding past link time to load time. Linkers and loaders now divided up the work, with linkers doing part of the address binding, assigning relative addresses within each program, and the loader doing a final relocation step to assign actual addresses.

As systems became more complex, they called upon linkers to do more and more complex name management and address binding. Fortran programs used multiple subprograms and common blocks, areas of data shared by multiple subprograms, and it was up to the linker to lay out storage and assign the addresses both for the subprograms and the common blocks. Linkers increasingly had to deal with object code libraries. including both application libraries written in Fortran and other languages, and compiler support libraries called implcitly from compiled code to handle I/O and other high-level operations.

Programs quickly became larger than available memory, so linkers provided overlays, a technique that let programmers arrange for different parts of a program to share the same memory, with each overlay loaded on demand when another part of the program called into it. Overlays were widely used on mainframes from the advent of disks around 1960 until the spread of virtual memory in the mid-1970s, then reappeared on microcomputers in the early 1980s in exactly the same form, and faded as virtual memory appeared on PCs in the 1990s. They're still used in memory limited embedded environments, and may yet reappear in other places where precise programmer or compiler control of memory usage improves performance.

With the advent of hardware relocation and virtual memory, linkers and loaders actually got less complex, since each program could again have an entire address space. Programs could be linked to be loaded at fixed addresses, with hardware rather than software relocation taking care of any load-time relocation. But computers with hardware relocation invariably run more than one program, frequently multiple copies of the same program. When a computer runs multiple instances of one program, some parts of the program are the same among all running instance (the executable code, in particular), while other parts are unique to each instance. If the parts that don't change can be separated out from the parts that do change, the operating system can use a single copy of the unchanging part, saving considerable storage. Compilers and assemblers were modified to create object code in multiple sections, with one section for read only code and another section for writable data, the linker had to be able to combine all of sections of each type so that the linked program would have all the code in one place and all of the data in another. This didn't delay address binding any more than it already was, since addresses were still assigned at link time, but more work was deferred to the linker to assign addresses for all the sections.

Even when different programs are running on a computer, those different programs usually turn out to share a lot of common code. For example, nearly every program written in C uses routines such as `fopen` and `printf`, database applications all use a large access library to connect to the database, and programs running under a GUI such as X Window, MS Windows, or the Macintosh all use pieces of the GUI library. Most systems now provide *shared libraries* for programs to use, so that all the programs that use a library can share a single copy of it. This both improves runtime performance and saves a lot of disk space; in small programs the common library routines often take up more space than the program itself.

In the simpler static shared libraries, each library is bound to specific addresses at the time the library is built, and the linker binds program references to library routines to those specific addresses at link time. Static libraries turn out to be inconveniently inflexible, since programs potentially have to be relinked every time any part of the library changes, and the details of creating static shared libraries turn out to be very tedious. Systems added dynamically linked libraries in which library sections and symbols aren't bound to actual addresses until the program that uses the library starts running. Sometimes the binding is delayed even farther than that; with full-fledged dynamic linking, the addresses of called procedures aren't bound until the first call. Furthermore, programs can bind to libraries as the programs are running, loading libraries in the middle of program execution. This provides a powerful and high-performance way to extend the function of programs. Microsoft Windows in particular makes extensive use of runtime loading of shared libraries (known as DLLs, Dynamically Linked Libraries) to construct and extend programs.

Linking vs. loading

Linkers and loaders perform several related but conceptually separate actions.

- *Program loading:* Copy a program from secondary storage (which since about 1968 invariably means a disk) into main memory so it's ready to run. In some cases loading just involves copying the data from disk to memory, in others it involves allocating storage, setting protection bits, or arranging for virtual memory to map virtual addresses to disk pages.
- *Relocation:* Compilers and assemblers generally create each file of object code with the program addresses starting at zero, but few computers let you load your program at location zero. If a program is created from multiple subprograms, all the subprograms have to be loaded at non-overlapping addresses. Relocation is the process of assigning load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses. In many systems, relocation happens more than once. It's quite common for a linker to create a program from multiple subprograms, and create one linked output program that starts at zero, with the various subprograms relocated to locations within the big program. Then when the program is loaded, the system picks the actual load address and the linked program is relocated as a whole to the load address.
- *Symbol resolution:* When a program is built from multiple subprograms, the references from one subprogram to another are made using symbols; a main program might use a square root routine called `sqrt`, and the math library defines `sqrt`. A linker resolves the symbol by noting the location assigned to `sqrt` in the library, and patching the caller's object code to so the call instruction refers to that location.

Although there's considerable overlap between linking and loading, it's reasonable to define a program that does program loading as a loader, and one that does symbol resolution as a linker. Either can do relocation, and there have been all-in-one linking loaders that do all three functions.

The line between relocation and symbol resolution can be fuzzy. Since linkers already can resolve references to symbols, one way to handle code relocation is to assign a symbol to the base address of each part of the program, and treat relocatable addresses as references to the base address symbols.

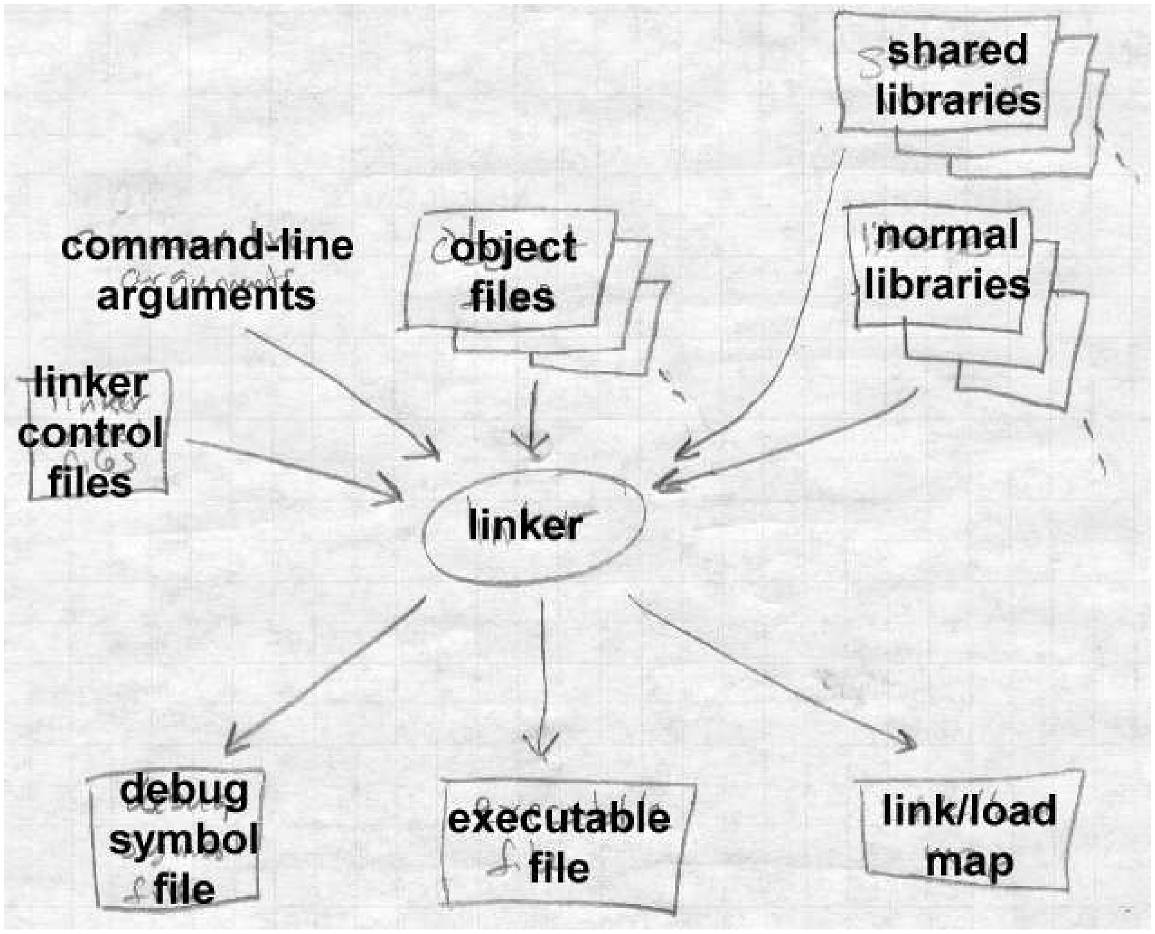
One important feature that linkers and loaders share is that they both patch object code, the only widely used programs to do so other than perhaps debuggers. This is a uniquely powerful feature, albeit one that is extremely machine specific in the details, and can lead to baffling bugs if done wrong.

Two-pass linking

Now we turn to the general structure of linkers. Linking, like compiling or assembling, is fundamentally a two pass process. A linker takes as its input a set of input object files, libraries, and perhaps command files, and produces as its result an output object file, and perhaps ancillary information such as a load map or a file containing debugger symbols, Figure 1.

Figure 1-1: The linker process

picture of linker taking input files, producing output file, maybe also other junk



Each input file contains a set of *segments*, contiguous chunks of code or data to be placed in the output file. Each input file also contains at least one *symbol table*. Some symbols are exported, defined within the file for use in other files, generally the names of routines within the file that can be called from elsewhere. Other symbols are imported, used in the file but not defined, generally the names of routines called from but not present in the file.

When a linker runs, it first has to scan the input files to find the sizes of the segments and to collect the definitions and references of all of the symbols It creates a segment table listing all of the segments defined in the input files, and a symbol table with all of the symbols imported or exported.

Using the data from the first pass, the linker assigns numeric locations to symbols, determines the sizes and location of the segments in the output address space, and figures out where everything goes in the output file.

The second pass uses the information collected in the first pass to control the actual linking process. It reads and relocates the object code, substituting numeric addresses for symbol references, and adjusting memory addresses in code and data to reflect relocated segment addresses, and writes the relocated code to the output file. It then writes the output file, generally with header information, the relocated segments, and symbol table information. If the program uses dynamic linking, the symbol table contains the info the runtime linker will need to resolve dynamic symbols. In many cases, the linker itself will generate small amounts of code or data in the output file, such as "glue code" used to call routines in overlays or dynamically linked libraries, or an array of pointers to initialization routines that need to be called at program startup time.

Whether or not the program uses dynamic linking, the file may also contain a symbol table for relinking or debugging that isn't used by the program itself, but may be used by other programs that deal with the output file.

Some object formats are relinkable, that is, the output file from one linker run can be used as the input to a subsequent linker run. This requires that the output file contain a symbol table like one in an input file, as well as all of the other auxiliary information present in an input file.

Nearly all object formats have provision for debugging symbols, so that when the program is run under the control of a debugger, the debugger can use those symbols to let the programmer control the program in terms of the line numbers and names used in the source program. Depending on the details of the object format, the debugging symbols may be intermixed in a single symbol table with symbols needed by the linker, or there may be one table for the linker and a separate, somewhat redundant table for the debugger.

A few linkers appear to work in one pass. They do that by buffering some or all of the contents of the input file in memory or disk during the linking process, then reading the buffered material later. Since this is an implementation trick that doesn't fundamentally affect the two-pass nature of linking, we don't address it further here.

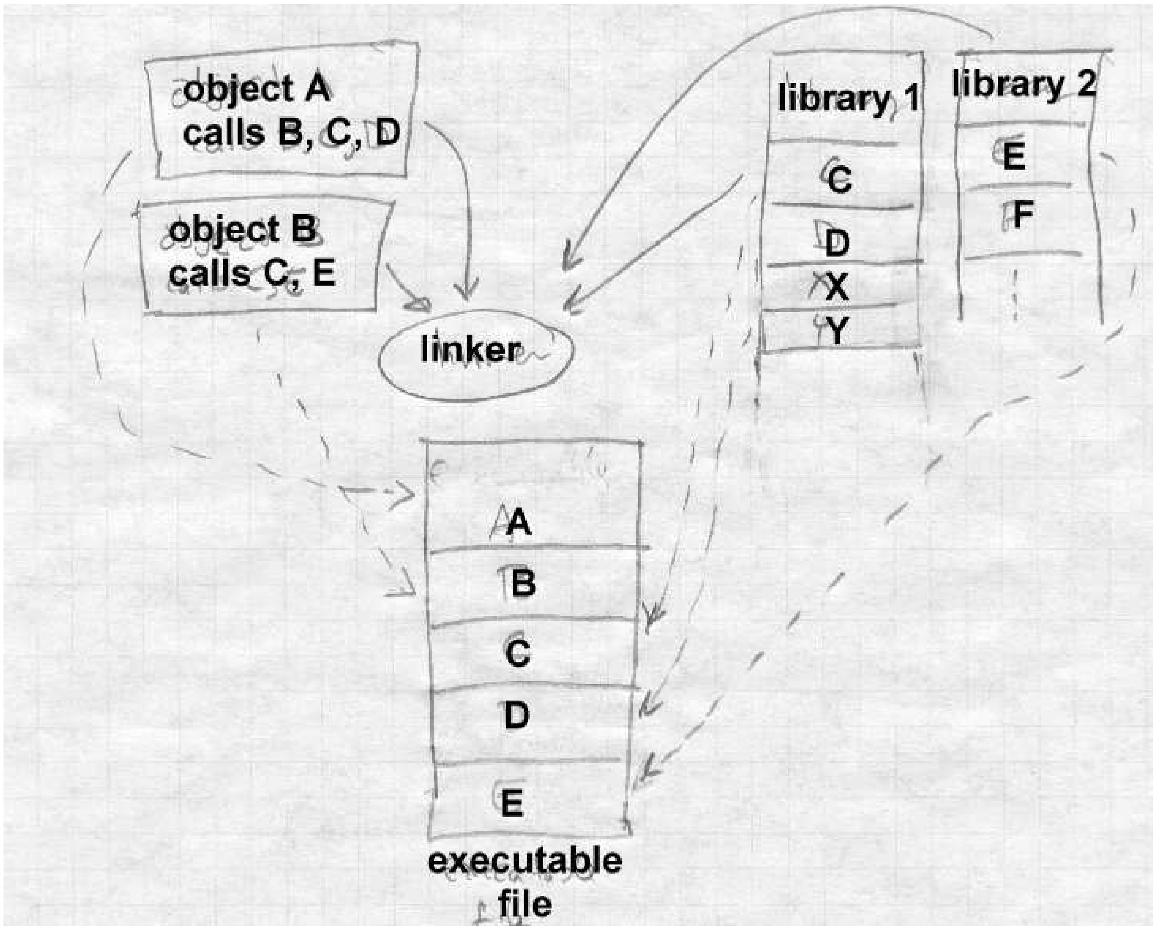
Object code libraries

All linkers support object code libraries in one form or another, with most also providing support for various kinds of shared libraries.

The basic principle of object code libraries is simple enough, Figure 2. A library is little more than a set of object code files. (Indeed, on some systems you can literally catenate a bunch of object files together and use the result as a link library.) After the linker processes all of the regular input files, if any imported names remain undefined, it runs through the library or libraries and links in any of the files in the library that export one or more undefined names.

Figure 1-2: Object code libraries

Object files fed into the linker, with libraries containing lots of files following along.



Shared libraries complicate this task a little by moving some of the work from link time to load time. The linker identifies the shared libraries that resolve the undefined names in a linker run, but rather than linking anything into the program, the linker notes in the output file the names of the libraries in which the symbols were found, so that the shared library can be bound in when the program is loaded. See Chapters 9 and 10 for the details.

Relocation and code modification

The heart of a linker or loader's actions is relocation and code modification. When a compiler or assembler generates and object file, it generates the code using the unrelocated addresses of code and data defined within the file, and usually zeros for code and data defined elsewhere. As part of the linking process, the linker modifies the object code to reflect the actual addresses assigned. For example, consider this snippet of x86 code that moves the contents of variable `a` to variable `b` using the eax register.

```
mov a,%eax
mov %eax,b
```

If `a` is defined in the same file at location 1234 hex and b is imported from somewhere else, the generated object code will be:

```
A1 34 12 00 00 mov a,%eax
A3 00 00 00 00 mov %eax,b
```

Each instruction contains a one-byte operation code followed by a four-byte address. The first instruction has a reference to 1234 (byte reversed, since the x86 uses a right to left byte order) and the second a reference to zero since the location of `b` is unknown.

Now assume that the linker links this code so that the section in which `a` is located is relocated by hex 10000 bytes, and `b` turns out to be at hex 9A12. The linker modifies the code to be:

```
A1 34 12 01 00 mov a,%eax
A3 12 9A 00 00 mov %eax,b
```

That is, it adds 10000 to the address in the first instruction so now it refers to `a`'s relocated address which is 11234, and it patches in the address for `b`. These adjustments affect instructions, but any pointers in the data part of an object file have to be adjusted as well.

On older computers with small address spaces and direct addressing, the modification process is fairly simple, since there are only only one or two address formats that a linker has to handle. Modern computers, including all RISCs, require considerably more complex code modification. No single instruction contains enough bits to hold a direct address, so the compiler and linker have to use complicated addressing tricks to handle data at arbitrary addresses. In some cases, it's possible to concoct an address using two or three instructions, each of which contains part of the address, and use bit manipulation to combine the parts into a full address. In this case, the linker has to be prepared to modify each of the instructions appropriately, inserting some of the bits of the address into each instruction. In other cases, all of the addresses used by a routine or group of routines are placed in an array used as an ''address pool'', initialization code sets one of the machine registers to point to that array, and code loads pointers out of the address pool as needed using that register as a base register. The linker may have to create the array from all of the addresses used in a program, then modify instructions that so that they refer to the appropriate address pool entry. We address this in Chapter 7.

Some systems require position independent code that will work correctly regardless of where in the address space it is loaded. Linkers generally have to provide extra tricks to support that, separating out the parts of the program that can't be made position independent, and arranging for the two parts to communicate. (See Chapter 8.)

Compiler Drivers

In most cases, the operation of the linker is invisible to the programmer or nearly so, because it's run automatically as part of the compilation process. Most compilation systems have a *compiler driver* that automatically invokes the phases of the compiler as needed. For example, if the programmer has two C language source files, the compiler driver will run a sequence of programs like this on a Unix system:

- C preprocessor on file A, creating preprocessed A
- C compiler on preprocessed A, creating assembler file A
- Assembler on assembler file A, creating object file A
- C preprocceor on file B, creating preprocessed B
- C compiler on preprocessed B, creating assembler file B

- Assembler on assembler file B, creating object file B
- Linker on object files A and B, and system C library

That is, it compiles each source file to assembler and then object code, and links the object code together, including any needed routines from the system C library.

Compiler drivers are often much cleverer than this. They often compare the creation dates of source and object files, and only recompile source files that have changed. (The Unix *make* program is the classic example.) Particularly when compiling C++ and other object oriented languages, compiler drivers can play all sorts of tricks to work around limitations in linkers or object formats. For example, C++ templates define a potentially infinite set of related routines, so to find the finite set of template routines that a program actually uses, a compiler driver can link the programs' object files together with no template code, read the error messages from the linker to see what's undefined, call the C++ compiler to generate object code for the necessary template routines and re-link. We cover some of these tricks in Chapter 11.

Linker command languages

Every linker has some sort of command language to control the linking process. At the very least the linker needs the list of object files and libraries to link. Generally there is a long list of possible options: whether to keep debugging symbols, whether to use shared or unshared libraries, which of several possible output formats to use. Most linkers permit some way to specify the address at which the linked code is to be bound, which comes in handy when using a linker to link a system kernel or other program that doesn't run under control of an operating system. In linkers that support multiple code and data segments, a linker command language can specify the order in which segments are to be linked, special treatment for certain kinds of segments, and other application-specific options.

There are four common techniques to pass commands to a linker:

- *Command line:* Most systems have a command line or the equivalent, via which one can pass a mixture of file names and switches. This is the usual approach for Unix and Windows linkers. On systems with limited length command lines, there's usually a way to direct the linker to read commands from a file and treat them as though they were on the command line.

- *Intermixed with object files:* Some linkers, such as IBM mainframe linkers, accept alternating object files and linker commands in a single input file. This dates from the era of card decks, when one would pile up object decks and hand-punched command cards in a card reader.
- *Embedded in object files:* Some object formats, notably Microsoft's, permit linker commands to be embedded inside object files. This permits a compiler to pass any options needed to link an object file in the file itself. For example, the C compiler passes commands to search the standard C library.
- *Separate configuration language:* A few linkers have a full fledged configuration language to control linking. The GNU linker, which can handle an enormous range of object file formats, machine architectures, and address space conventions, has a complex control language that lets a programmer specify the order in which segments should be linked, rules for combining similar segments, segment addresses, and a wide range of other options. Other linkers have less complex languages to handle specific features such as programmer-defined overlays.

Linking: a true-life example

We complete our introduction to linking with a small but real linking example. Figure 3 shows a pair of C language source files, m.c with a main program that calls a routine named `a`, and a.c that contains the routine with a call to the library routines `strlen` and `printf`.

Figure 1-3: Source files

Source file m.c

```
extern void a(char *);
```

```
int main(int ac, char **av)
{
  static char string[] = "Hello, world!\n";

  a(string);
}
```

Source file a.c

```
#include <unistd.h>
#include <string.h>

void a(char *s)
{
  write(1, s, strlen(s));
}
```

The main program m.c compiles, on my Pentium with GCC, into a 165 byte object file in the classic a.out object format, Figure 4. That object file includes a fixed length header, 16 bytes of "text" segment, containing the read only program code, and 16 bytes of "data" segment, containing the string. Following that are two relocation entries, one that marks the pushl instruction that puts the address of the string on the stack in preparation for the call to `a`, and one that marks the call instruction that transfers control to `a`. The symbol table exports the definition of `_main`, imports `_a`, and contains a couple of other symbols for the debugger. (Each global symbol is prefixed with an underscore, for reasons described in Chapter 5.) Note that the pushl instruction refers to location 10 hex, the tentative address for the string, since it's in the same object file, while the call refers to location 0 since the address of `_a` is unknown.

Figure 1-4: Object code for m.o

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
```

```
  0 .text         00000010  00000000  00000000  00000020  2**3
  1 .data         00000010  00000010  00000010  00000030  2**3
Disassembly of section .text:

00000000 <_main>:
   0:  55             pushl  %ebp
   1:  89 e5          movl   %esp,%ebp
   3:  68 10 00 00 00 pushl  $0x10
     4: 32  .data
   8:  e8 f3 ff ff ff call   0
     9: DISP32 _a
   d:  c9             leave
   e:  c3             ret
  ...
```

The subprogram file a.c compiles into a 160 byte object file, Figure 5, with the header, a 28 byte text segment, and no data. Two relocation entries mark the calls to `strlen` and `write`, and the symbol table exports `_a` and imports `_strlen` and `_write`.

Figure 1-5: Object code for m.o

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000001c  00000000  00000000  00000020  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data         00000000  0000001c  0000001c  0000003c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
Disassembly of section .text:

00000000 <_a>:
   0:  55             pushl  %ebp
   1:  89 e5          movl   %esp,%ebp
   3:  53             pushl  %ebx
   4:  8b 5d 08       movl   0x8(%ebp),%ebx
   7:  53             pushl  %ebx
   8:  e8 f3 ff ff ff call   0
     9: DISP32 _strlen
   d:  50             pushl  %eax
   e:  53             pushl  %ebx
   f:  6a 01          pushl  $0x1
  11:  e8 ea ff ff ff call   0
     12: DISP32  _write
  16:  8d 65 fc       leal   -4(%ebp),%esp
  19:  5b             popl   %ebx
  1a:  c9             leave
  1b:  c3             ret
```

To produce an executable program, the linker combines these two object files with a standard startup initialization routine for C programs, and necessary routines from the C library, producing an executable file displayed in part in Figure 6.

Figure 1-6: Selected parts of executable

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000fe0  00001020  00001020  00000020  2**3
  1 .data         00001000  00002000  00002000  00001000  2**3
  2 .bss          00000000  00003000  00003000  00000000  2**3

Disassembly of section .text:

00001020 <start-c>:
  ...
    1092: e8 0d 00 00 00 call   10a4 <_main>
  ...
000010a4 <_main>:
    10a4: 55             pushl  %ebp
    10a5: 89 e5          movl   %esp,%ebp
    10a7: 68 24 20 00 00 pushl  $0x2024
    10ac: e8 03 00 00 00 call   10b4 <_a>
    10b1: c9             leave
    10b2: c3             ret
  ...

000010b4 <_a>:
    10b4: 55             pushl  %ebp
    10b5: 89 e5          movl   %esp,%ebp
    10b7: 53             pushl  %ebx
    10b8: 8b 5d 08       movl   0x8(%ebp),%ebx
    10bb: 53             pushl  %ebx
    10bc: e8 37 00 00 00 call   10f8 <_strlen>
    10c1: 50             pushl  %eax
    10c2: 53             pushl  %ebx
    10c3: 6a 01          pushl  $0x1
    10c5: e8 a2 00 00 00 call   116c <_write>
    10ca: 8d 65 fc       leal   -4(%ebp),%esp
    10cd: 5b             popl   %ebx
    10ce: c9             leave
    10cf: c3             ret
  ...
000010f8 <_strlen>:
  ...
0000116c <_write>:
  ...
```

The linker combined corresponding segments from each input file, so there is one combined text segment, one combined data segment and one bss segment (zero-initialized data, which the two input files didn't use). Each segment is padded out to a 4K boundary to match the x86 page size, so the text segment is 4K (minus a 20 byte a.out header present in the file but not logically part of the segment), the data and bss segments are also each 4K.

The combined text segment contains the text of library startup code called `start-c`, then text from m.o relocated to 10a4, a.o relocated to 10b4, and routines linked from the C library, relocated to higher addresses in the text segment. The data segment, not displayed here, contains the combined data segments in the same order as the text segments. Since the code for `_main` has been relocated to address 10a4 hex, that address is patched into the call instruction in start-c. Within the main routine, the reference to the string is relocated to 2024 hex, the string's final location in the data segment, and the call is patched to 10b4, the final address of `_a`. Within `_a`, the calls to `_strlen` and `_write` are patched to the final addresses for those two routines.

The executable also contains about a dozen other routines from the C library, not displayed here, that are called directly or indirectly from the startup code or from `_write` (error routines, in the latter case.) The executable contains no relocation data, since this file format is not relinkable and the operating system loads it at a known fixed address. It contains a symbol table for the benefit of a debugger, although the executable doesn't use the symbols and the symbol table can be stripped off to save space.

In this example, the code linked from the library is considerably larger than the code for the program itself. That's quite common, particularly when programs use large graphics or windowing libraries, which provided the impetus for shared libraries, Chapters 9 and 10. The linked program is 8K, but the identical program linked using shared libraries is only 264 bytes. This is a toy example, of course, but real programs often have equally dramatic space savings.

Exercises

What is the advantage of separating a linker and loader into separate programs? Under what circumstances would a combined linking loader be useful?

Nearly every programming system produced in the past 50 years includes a linker. Why?

In this chapter we've discussed linking and loading assembled or compiled machine code. Would a linker or loader be useful in a purely interpretive system that directly interprets source language code? How about in a interpretive system that turns the source into an intermediate representation like P-code or the Java Virtual Machine?