【ICS】链接 - Linking

1. What Linking?

- 将各种代码和数据片段收集并组合为一个单一文件。
- 这个文件可以被加载（复制）到内存并执行。
- 链接可以发生在编译时（compile time）——源代码被翻译为机器代码时
- 也可以发生于加载时（load time）——程序被加载器加载到内存并执行时
- 甚至可以发生于运行时（run time）——由应用程序来执行

在现代计算机系统中，链接由链接器自动执行。

2. Why Linking?

链接器使得分离编译（seperate compilation）成为可能——这意味着我们不用将一个大型应用程序组织为一个巨大的源文件。我们可以将其分解成为更小、更好管理的模块，并且独立修改、编译它们。

因此，我们修改了一个模块，我们只需要重新编译一个模块，然后重新做链接即可。

3. How Linking?

3.1 From .cpp to executable file

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

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

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

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

code/link/main.c

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

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

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

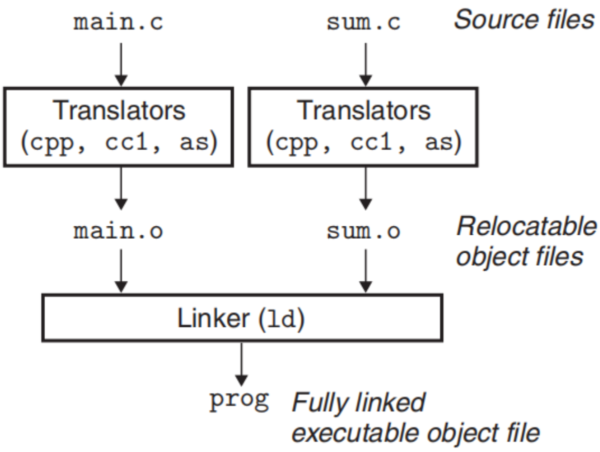
code/link/sum.c

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

1. 首先运行C预处理器（cpp)。它将C的原程序main.c翻译成一个ASCII码的中间文件main.i

2. 接下来运行C编译器（cc1），将main.i翻译成ASCII汇编语言文件main.s
3. 然后驱动程序运行汇编器（as），将main.s翻译为可重定位目标文件（relocatable object file）main.o
4. 驱动通过相同方法生成sum.o；然后链接器程序ld（注：ld是linux下的一种静态链接器）将main.o和sum.o以及一些其他的必要系统目标文件组合起来，创建一个executable object file。

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



如何执行?

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.

3.2 Basic Concepts

3.2.1 我们要干什么?

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*.

总结：

- 目标文件就是纯粹的字节块的集合
- 这些块中，有些是指令，有些是数据
- 链接器将这些块连接起来，确定被连接块的运行时位置，修改代码和数据块中的各种位置

3.2.2 Object File 目标文件

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.

- 编译器和汇编器生成relocatable object file（包括了shared object file）
- 链接器生成executable object file

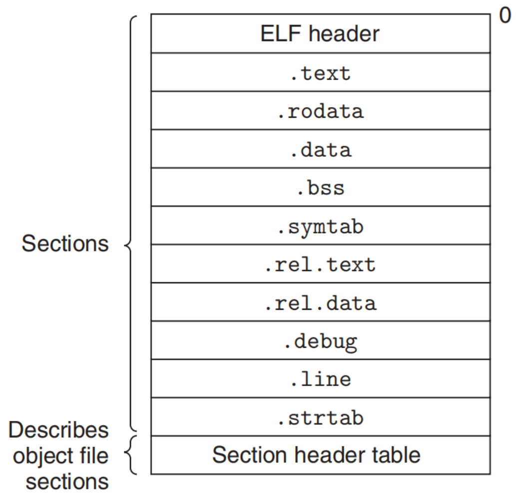
ELF：

- 现代X86-64架构linux/unix使用的可执行可链接格式（executable and linkable format）

3.2.3 Relocatable Object File 可重定位目标文件（注意啊！这是编译器+汇编器干的）

典型ELF的格式：

Figure 7.3
Typical ELF relocatable object file.



`.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.

`.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.

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!"

3.2.4 Symbols and Symbol Table 符号、符号表

重点来看看ELF中的符号表。这里的需要关注的其实就是：

- 全局符号：存在符号表里
- 非全局符号：
  - 不带static：符号表不care，这玩意存在栈上
  - 带static：符号表care，存在我这里

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.

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.

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`:

```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

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`.

3.2.5 符号解析

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,

```c
void foo(void);

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

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'
```

这里可能会想到一个比较tricky的问题：

- 编译器虽然保证了每个模块中的局部符号只有一个定义；
- 但是如果多个模块中，全局符号重名了怎么办呢？

3.3 🌶️链接器如何解析多重定义的全局符号

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.

3.4 Linking with Static Libraries 与静态库链接

3.4.1 What 静态库?

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.

You

C++中的静态库是什么

ChatGPT

在C++中，静态库（Static Library）是一种预编译的库，其中包含了一组函数和数据，它们在编译时被链接到程序中，以便在程序运行时使用。静态库的扩展名通常是`.lib`（在Windows平台）或`.a`（在Unix/Linux平台）。

静态库与动态库（Dynamic Library）相对应。与动态库不同，静态库在编译时会被完整地链接到目标程序中，因此目标程序在运行时不需要额外的库文件，所有必要的函数和数据都包含在目标程序中。这意味着目标程序相对较大，但也更加独立，不需要依赖外部库文件。

静态库通常由开发者提供，并包含了一组相关的函数和数据，用于特定的功能或模块。开发者可以将这些库与他们的程序一起分发，确保程序在其他计算机上的运行，而无需担心缺少所需的库文件。

创建静态库通常包括以下步骤：

1. 编写源代码文件，并编译成目标文件（`.obj`文件或`.o`文件）。
2. 将目标文件归档到静态库中，这可以通过`ar`（Unix/Linux平台）或`lib`（Windows平台）等工具完成。
3. 将静态库与开发的程序一起链接，以生成最终的可执行文件。

在使用静态库时，开发者需要确保将其链接到他们的项目中，并在编译时指定正确的链接选项，以便编译器能够找到并将库文件包含到最终的可执行文件中。

You

静态库是很多很多文件吗

ChatGPT

静态库通常是由多个目标文件（Object Files）组成的。每个目标文件包含一个或多个相关的函数或数据的定义。这些目标文件经过归档（Archiving）处理后，打包成一个单独的文件，这个文件就是静态库。

静态库的好处之一是，它可以将多个相关的功能组织在一个文件中，使得程序的组织和管理更加方便。同时，开发者可以在不同的项目中重复使用这些静态库，从而避免了重复编写相同功能的代码的需求。

因此，尽管静态库在逻辑上可以看作是一个文件，但实际上它是由多个目标文件组成的，每个目标文件包含了相关的函数或数据的定义。

You

打包成一个单独的文件，那么它不是很大吗

ChatGPT

静态库在物理上是一个文件，但是它可以包含多个目标文件（Object Files）。虽然这个文件在磁盘上可能很大，但是当这个静态库被链接到程序中时，链接器（Linker）会只选择程序所需的那些目标文件，并将它们复制到最终的可执行文件中。

这意味着，即使静态库很大，但只有程序中用到的部分会被包含在最终的可执行文件中。因此，尽管静态库本身可能很大，但最终的可执行文件通常只包含所需部分，不会因此变得过大。

此外，对于不同的平台和编译器，通常会有一些优化策略用于减小静态库的大小，例如去除未使用的函数、数据等。因此，即使静态库包含很多文件，最终的可执行文件大小也会受到控制。

3.4.2 Why 静态库?

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.

3.4.3 How 静态库?

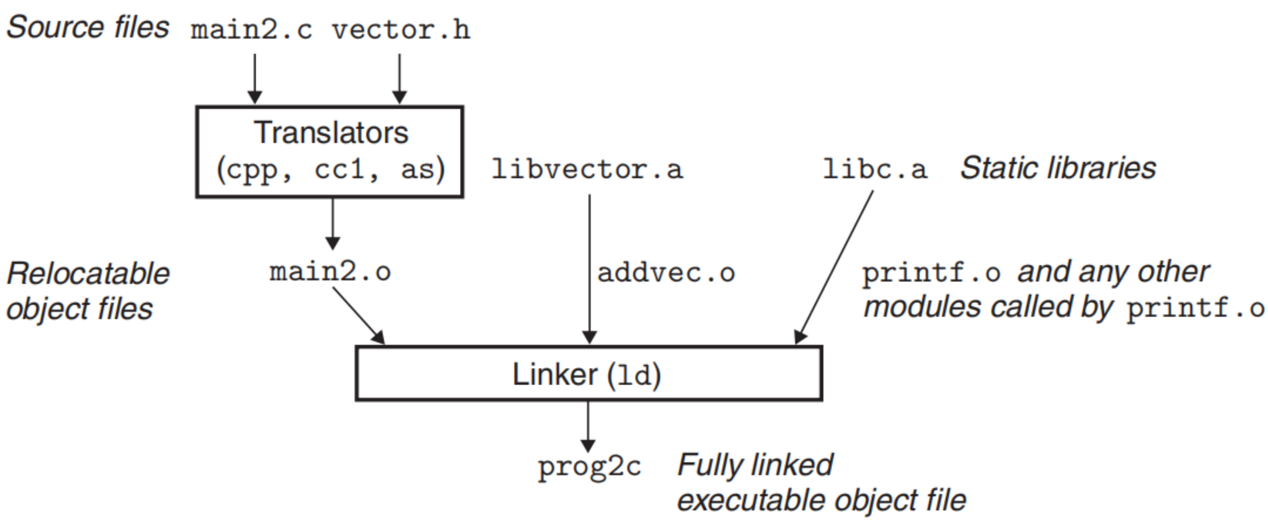


Figure 7.8 **Linking with static libraries.**

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.

3.5 🌶️加载可执行目标文件

这一块其实感觉和理解链接倒是没什么大关系，主要是帮助我们理解整个计算机系统跑一段代码的流程。

这一段真的讲的好综合啊，感觉把全部的东西都串起来了：

- 异常控制流
- 虚拟内存

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

```
linux> ./prog
```

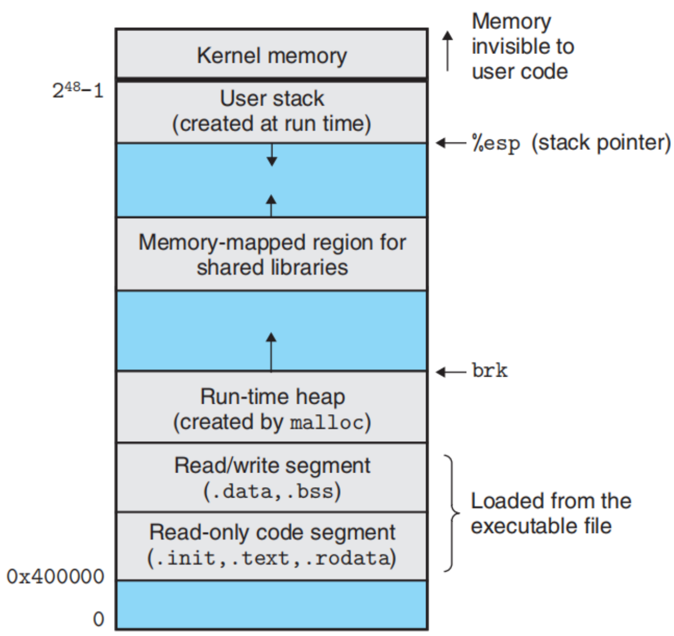
Since `prog` does not correspond to a built-in shell command, the shell assumes that `prog` is an executable object file, which it runs for us by invoking some memory-resident operating system code known as the `loader`. Any Linux program can invoke the loader by calling the `execve` function, which 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 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.

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.



3.6 Dynamic Linking with Shared Libraries 动态链接共享库

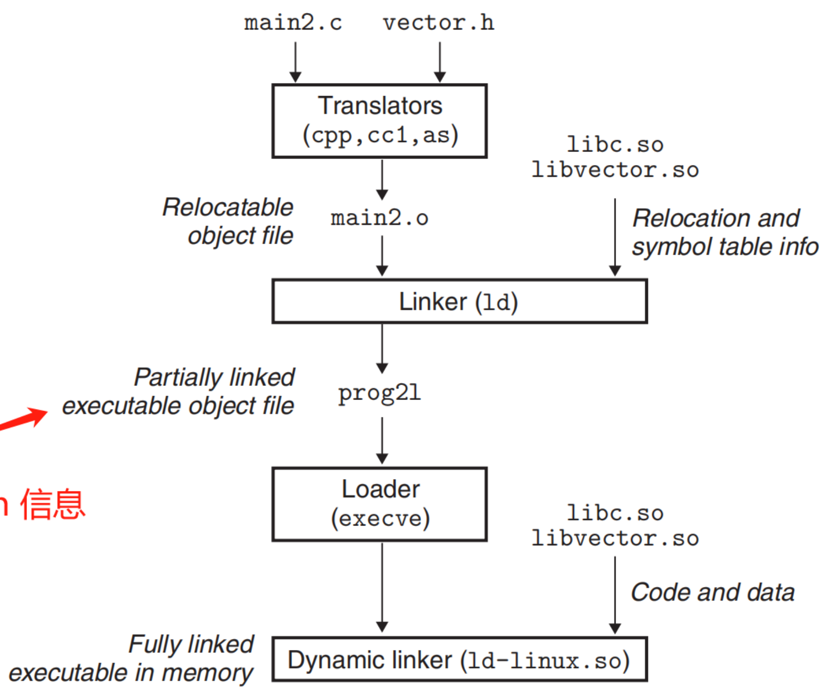
3.6.1 Why 动态链接?

静态链接的问题:

- 几乎每个C程序都使用标准IO函数,在运行的时候,这些函数的代码会被复制到每个运行的进程的文本段中;在一个运行上百个进程的系统上,这是很大的浪费。
- 所以我们引入了“共享库”的概念:
  - Shared library:一个目标模块,在运行或者加载的时候,可以加载到任意的内存地址,并且和一个在内存中的程序链接起来。这个过程就是动态链接。
  - linux下叫共享库,windows下叫动态链接库。

3.6.2 How 动态链接?

Figure 7.16
Dynamic linking with shared libraries.



数据、代码没有进去;
有symbol table和relocation 信息

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.

"linking at running time":可执行文件加载到内存的时候去做链接。

3.7 🌶️位置无关代码 PIC: position independent code

共享库的目的就是让很多进程share内存中的同一份库代码。

我们就需要PIC，来使得多个进程share库代码的单一副本。它们可以加载到内存的任何位置而不需要链接器修改。为了做到这一点，我们引入GOT（global offset table）这种数据结构。

- 当然，与之相对应的，在x86里，如果我们引用的是当前模块里面的符号，直接用pc相对寻址就完事了。但是如果我们引用的是其他模块里的符号，就必须来点这种奇技淫巧了。

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.

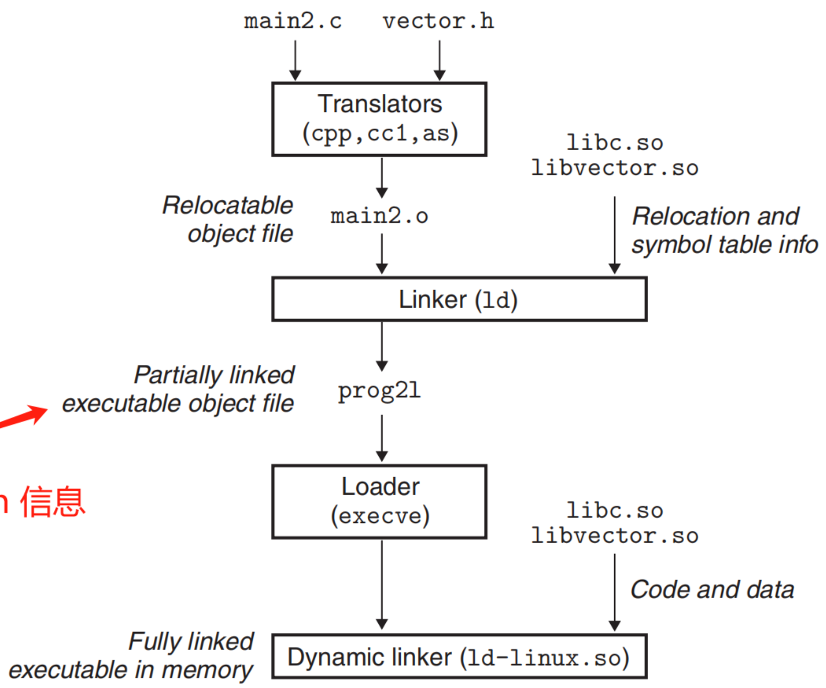
To avoid these problems, modern systems compile the code segments of shared modules so that they can be loaded anywhere in memory without having to be modified by the linker. With this approach, a single copy of a shared module's code segment can be shared by an unlimited number of processes. (Of course, 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.

TODO

4. Take away messages

Figure 7.16
Dynamic linking with shared libraries.



数据、代码没有进去；
有symbol table和relocation 信息