Shared libraries

$Revision: 1.1 $
$Date: 1998/12/30 03:00:06 $

Program libraries date back to the earliest days of computing, but with the advent of compiled languages including Fortran and COBOL, libraries became an integral part of programming since compiled languages need libraries to work. They use libraries explictly when a program calls a standard procedure such as sqrt(), and they use libraries implicitly for I/O, conversions, sorting, and a variety of other higher level features. As languages have gotten more complex, libraries have gotten correspondingly more complex. When I wrote a Fortran 77 compiler twenty years ago, the runtime library was already more work than the compiler itself.

The growth of language libraries means not only that all programs include library code, but that programs include a lot of the same library code. Every C program, for example, uses the system call library, nearly all use the standard I/O library routines such as printf, and many use other popular libraries for math, networking, and other common functions. This means that in a typical Unix system with a thousand compiled programs, there's close to a thousand copies of printf. If all those programs could share a single copy of the library routines they use, the savings in disk space would be substantial. (On a Unix system without shared libraries, there's five to ten megabytes of copies of printf alone.) Even more important, if running programs could share a single in-memory copy of the libraries, the main memory savings could be very significant, both saving memory and improving paging behavior.

All shared library schemes work essentially the same way. At link time, the linker searches through libraries as usual to find modules that resolve otherwise undefined external symbols. But rather than copying the contents of the module into the output file, the linker makes a note of what library the module came from, and puts a list of the libraries in the executable. When the program is loaded, startup code finds those libraries and maps them into the program's address space before the program starts, Figure 1. Standard operating system file mapping semantics automatically share pages that are mapped read-only or copy-on-write. The startup code that does the mapping may be in the operating system, the executable, in a special dynamic linker mapped into the process' address space, or some combination of the three.

Figure 1: Program with shared libraries

Picture of executable, shared libraries
main excutable, app library, C library
files from different places
arrows show refs from main to app, main to C, app to C

In this chapter, we look at static linked shared libraries, that is, libraries where program and data addresses in libraries are bound to executables at link time. In the next chapter we look at the considerably more complex dynamic linked libraries. Although dynamic linking is more flexible and more "modern", it's also a lot slower than static linking because a great deal of work that would otherwise have been done once at link time is redone each time a dynamically linked program starts. On systems that support both, unless programs need the extra flexibility of dynamic linking, they're faster and smaller with static linked libraries.

Binding time

Shared libraries raise binding time issues that don't apply to conventionally linked programs. A program that uses a shared library depends on having that shared library available when the program is run. One kind of error occurs when the required libraries aren't present. There's not much to be done other than printing a cryptic error message and exiting.

A much more interesting problem occurs when the library is present, but the library has changed since the program was linked. In a conventionally linked program, symbols are bound to addresses and library code is bound to the executable at link time. With static shared libraries, symbols are still bound to addresses at link time, but library code isn't bound to the executable until run time. (With dynamic shared libraries, they're both delayed until runtime.) A static linked share library can't change very much without breaking the programs that it is bound to. Since the addresses of routines and data in the library are bound into the program, any changes in the addresses to which the program is bound will cause havoc.

In a few cases, a static shared library can be updated without breaking the programs that use it, if the updates can be made in a way that don't move any addresses in the library that programs depend on. This permits "minor version" updates, typically for small bug fixes. Larger changes unavoidably change program addresses, which means that a system either needs multiple versions of the library, or forces programmers to relink all their programs each time the library changes. In practice, the solution is invariably multiple versions, since disk space is cheap and tracking down every executable that might have used a shared library is rarely possible.

Shared libraries in practice

In the rest of this chapter we concentrate on the static shared libraries provided in UNIX System V Release 3.2 (COFF format), older Linux systems (a.out format) the BSD/OS derivative of 4.4BSD (a.out and ELF formats.) All three work nearly the same, but some of the differences are instructive. The SVR3.2 implementation required changes in the linker to support searching shared libraries and extensive operating system support to do the runtime startup required. The Linux implemention required one small tweak to the linker and added a single system call to assist in library mapping. The BSD/OS implementation made no changes at all to the linker or operating system, using a shell script to provide the necessary arguments to the linker.

Address space management

The most difficult aspect of shared libraries is address space management. Each shared library occupies a fixed piece of address space in each program in which it is used. Different libraries have to use non-overlapping addresses if they can be used in the same program. Although it's possible to check mechanically that libraries don't overlap, assigning address space to libraries is a black art. On the one hand, you want to leave some slop in between them so if a new version of one library grows a little, it won't bump into the next library up. On the other hand, you'd like to put your popular libraries as close together as possible to minimize the number of page tables needed. (Recall that on an x86, for example, there's a second level table for each 4MB block of address space active in a process.)

There's invariably a master table of shared library address space on each system, with libraries starting some place in the address space far away from applications. Linux's start at hex 60000000, BSD/OS at a0000000. Commercial vendors subdivide the address space further between vendor supplied libraries and user and third-party libraries which start at a0800000 in BSD/OS, for example.

Generally both the code and data addresses for each library are explicitly defined, with the data area starting on a page boundary a page or two after the end of the code. This makes it somewhat to create minor version updates, since the updates frequently don't change the data layout, but just add or change code.

Each individual shared library has exported symbols, both code and data, and usually also has imported symbols if the library depends on other libraries. Although it would work if one just linked routines together into a shared library in haphazard order, real libraries use some discipline in assigning addresses to make it easier, or at least possible, to update a library without changing bound addresses. For code addresses, rather than exporting the actual address of each routine, the library contains a table of jump instructions which jump to all of the routines, with the addresses of the jump instructions exported as the addresses of the routines. All jump instruction are the same size, so the addresses in the jump table are easy to compute and won't change from version to version so long as no entries are added or deleted. One extra jump per routine is an insignificant slowdown, but since the actual routine addresses are not visible, new versions of the library will be compatible even if routines in the new version aren't all the same sizes and addresses as in the old version.

For exported data, the situation is more difficult, since there's no easy way to add a level of indirection like the one for code addresses. In practice it turns out that exported data tends to be tables of known sizes that change rarely, such as the array of `FILE` structures for the C standard I/O library or single word values like `errno` (the error code from the most recent system call) or `tzname` (pointers to two strings giving the name of the current time zone.) With some manual effort, the programmer who creates the shared library can collect the exported data at the front of the data section in front of any anonymous data that are part of individual routines, making it less likely that exported addresses will change from one version to the next.

Structure of shared libraries

The shared library is an executable format file that contains all of the library code and data, ready to be mapped in, Figure 2.

Figure 2: Structure of typical shared library

File header, a.out, COFF, or ELF header
(Initialization routine, not always present)
Jump table
Code
Global data
Private data

Some shared libraries start with a small bootstrap routine used to map in the rest of the library. After that comes the jump table, aligned on a page boundary if it's not the first thing in the library. The exported address of each public routine in the library is the jump table entry. Following the jump table is the rest of the text section (the jump table is considered to be text, since it's executable code), then the exported data and private data. The bss segment logically follows the data, but as in any other executable file, isn't actually present in the file.

Creating shared libraries

A UNIX shared library actually consists of two related files, the shared library itself and a stub library for the linker to use. A library creation utility takes a conventional library and creates the two shared library files. The stub library contains no code or data at all (other than possibly a tiny bootstrap routine) but contains symbol definitions for programs linked with the library to use.

A library creation utility takes as input a normal library in archive format and some files of control information and uses them to create create the shared library and the stub library.

Creating the shared library involves these basic steps, which we discuss in greater detail below:

- Determine at what address the library's code and data will be loaded.
- Scan through the input library to find all of the exported code symbols. (One of the control files may be a list of some of symbols not to export, if they're just used for inter-routine communication within the library.)
- Make up the jump table with an entry for each exported code symbol.
- If there's an initialization or loader routine at the beginning of the library, compile or assemble that.
- Create the shared library: Run the linker and link everything together into one big executable format file.
- Create the stub library: Extract the necessary symbols from the newly created shared library, reconcile those symbols with the symbols from the input library, create a stub routine for each library routine, then compile or assemble the stubs and combine them into the stub library. In COFF libraries, there's also a little initialization code placed in the stub library to be linked into each executable.

Creating the jump table

The easiest way to create the jump table is to write an assembler source file full of jump instructions, Figure =n+F, and assemble it. Each jump instruction needs to be labelled in a systematic way so that the addresses can later be extracted for the stub library.

A minor complication occurs on architectures like the x86 that have different sizes of jump instructions. For libraries containing less than 64K of code, short 3 byte jumps are adequate. For libraries larger than that, longer 5 byte jumps are necessary. Mixed sizes of jumps aren't very satisfactory, both because it makes the table addresses harder to compute and because it makes it far harder to make the jump table compatible in future builds of the library. The simplest solution is to make all of the jumps the largest size. Alternatively, make all of the jumps short, and for routines that are too far away for short jumps, generate anonymous long jump instructions at the end of the table to which short instructions can jump. (That's usually more trouble than it's worth, since jump tables are rarely more than a few hundred entries in the first place.)

Figure 2: Jump table

```
 ... start on a page boundary
JUMP_read: jmp _read
JUMP_write: jmp _write
 ...
_read: ... code for read()
 ...
_write: ... code for write()
```

Creating the shared library

Once the jump table and, if needed, the loader routine are created, creating the shared library is easy. Just run the linker with suitable switches to make the code and data start at the right places, and link together the bootstrap, the jump tables, and all of the routines from the input library. This both assigns addresses to everything in the library and creates the shared library file.

One minor complication involves interlibrary references. If you're creating, say, a shared math library that uses routines from the shared C library, the references have to be made correctly. Assuming that the library whose routines are neede has already been built, the when the linker builds the new library, it needs only to search the old library's stub library, just as like any normal executable that refers to the old library. This will get all of the references correct. The only remaining issue is that there needs to be some way to ensure that any programs that use the new library also link to the old library. Suitable design of the new stub library can ensure that.

Creating the stub library

Creating the stub library is one of the trickier parts of the shared library process. For each routine in the real library, the stub library needs to contain a corresponding entry that defines both the exported and imported global symbols.

The data global symbols are wherever the linker put them in the shared library image, and the most reasonable way to get their values is to create the shared library with a symbol table and extract the symbols from that symbol table. For code global symbols, the entry points are all in the jump table, so it's equally easy to extract the symbols from the shared library or compute the addresses from the base address of the jump table and each symbol's position in the table.

Unlike a normal library module, a module in the stub library contains no code nor data, but just has symbol definitions. The symbols have to be defined as absolute numbers rather than relocatable, since the shared library has already had all of its relocation done. The library creation program extracts each routine from the input library, and from that routine gets the defined and undefined globals, as well as the type (text or data) of each global. It then writes the stub routine, usually as a little assembler program, defining each text global as the address of the jump table entry, each data or bss global as the actual address in the shared library, and each undefined global as undefined. When it has a complete set of stub sources, it assembles them all and combines them into a normal library archive.

COFF stub libraries use a different, more primitive design. They're single object files with two named sections. The `.lib` section contains all of the relocation information pointing at the shared library, and the `.init` section contains initialization code that is linked into each client program, typically to initialize variables in the shared library.

Linux shared libraries are simpler still, an a.out file containing the symbol definitions with "set vector" symbol described in more detail below for use at program link time.

Shared libraries have names assigned that are mechanically derived from the original library, adding a version number. If the original library was called `/lib/libc.a`, the usual name for the C library, and the current library version is 4.0, the stub library might be `/lib/libc_s.4.0.0.a` and the shared library image `/shlib/libc_s.4.0.0`. (The extra zero allows for minor version updates.) Once the libraries are moved into the appropriate directories they're ready to use.

Linking with shared libraries

Linking with static shared libraries is far simpler than creating the libraries, because the process of creating the stub libraries has already done nearly all the hard work to fake out the linker and make it resolve program addresses to the appropriate places in the libraries. The only hard part is arranging for the necessary shared libraries to be mapped in when the program starts.

Each format provides a trick to let the loader create a list of libraries that startup code can use to map in the libraries. COFF libraries use a brute force approach; ad hoc code in the linker creates a section in the COFF file with the names of the libraries. The Linux linker had a somewhat less brute force approach that created a special symbol type called a "set vector". Set vectors are treated like normal global symbols, except that if there are multiple definitions, the definitions are all put in an array named by the symbol. Each shared library stub defines a set vector symbol `____SHARED_LIBRARIES__` that is the address of a structure containing the name, version, and load address of the library. The linker creates an array of pointers to each of those structures and calls it `____SHARED_LI-`

`BRARIES__` so the runtime startup code can use it. **The BSD/OS shared library scheme uses no linker tricks at all. Rather, the shell script wrapper used to create a shared executable runs down the list of libraries passed as arguments to the command or used implicitly (the C library), extracts the file names and load addresses for those libraries from a list in a system file, writes a little assembler source file an array of structures containing library names and load addresses, assembles that file, and includes the object file in the list of arguments to the linker.**

In each case, the references from the program code to the library addresses are resolved automatically from the addresses in the stub library.

Running with shared libraries

Starting a program that uses shared libraries involves three steps: loading the executable, mapping the libraries, and doing library-specific initialization. In each case, the program executable is loaded into memory by the system in the usual way. After that, the different schemes diverge. The System V.3 kernel had extensions to handle COFF shared library executables and the kernel internally looked at the list of libraries and mapped them in before starting the program. The disadvantages of this scheme were ‘‘kernel bloat’’, adding more code to the nonpagable kernel, and inflexibility, since it didn’t permit any flexibility or upgradability in future versions. (System V.4 scrapped the whole scheme and went to ELF dynamic shared libraries which we address in the next chapter.)

Linux added a single uselib() system call that took the file name and address of a library and mapped it into the program address space. The startup routine bound into the executable ran down the list of libraries, doing a uselib() on each.

The BSD/OS scheme uses the standard mmap() system call that maps pages of a file into the address space and a bootstrap routine that is linked into each shared library as the first thing in the library. The startup routine in the executable runs down the table of shared libraries, and for each one opens the file, maps the first page of the file to the load address, and then calls the bootstrap routine. The bootstrap routine then maps the rest of the text segment, the data segment, and maps fresh address space for the bss segment, then returns.

Once the segments are all mapped, there's often some library-specific initialization to do, for example, putting a pointer to the system environment strings in the global `environ` specified by standard C. The COFF implementation collects the initialization code from the `.init` segments in the program file, and runs it from the program startup code. Depending on the library it may or may not call routines in the shared library. The Linux implemention doesn't do any library initialization and documents the problem that variables defined in both the program and the library don't work very well.

In the BSD/OS implementation, the bootstrap routine for the C library receives a pointer to the shared library table and maps in all of the other libraries, minimizing the amount of code that has to be linked into individual executables. Recent versions of BSD use ELF format executables. The ELF header has a `interp` section containing the name of an "interpreter" program to use when running the file. BSD uses the shared C library as the interpreter, which means that the kernel maps in the shared C library before the program starts, saving the overhead of some system calls. The library bootstrap routine does the same initializations, maps the rest of the libraries, and, via a pointer, calls the main routine in the program.

The malloc hack and global data conflicts

Although static shared libraries have excellent performance, their long-term maintenance is difficult and error-prone, as this anecdote illustrates.

In a static library, all intra-library calls are permanently bound, and it's not possible to substitute a private version of a routine by redefining the routine in a program that uses the library. For the most part, that's not a problem since few programs redefine standard library routines like `read()` or `strcmp()`, or even if they do it's not a major problem if the program uses a private version of `strcmp()` while routines in the library call the standard version.

But a lot of programs define their own versions of `malloc()` and `free()`, the routines that allocate heap storage, and multiple versions of those routines in a program don't work. The standard `strdup` routine, for example, returns a pointer to a string allocated by malloc, which the application can free when no longer needed. If the library allocated the string one version of malloc, but the application freed that string with a different version of free, chaos would ensue.

To permit applications to provide their own versions of malloc and free, the System V.3 shared C library uses an ugly hack, Figure 3. The system's maintainers redefined malloc and free as indirect calls through pointers bound into the data part of the shared library that we'll call `malloc_ptr` and `free_ptr`.

```
extern void *(*malloc_ptr)(size_t);
extern void (*free_ptr)(void *);
#define malloc(s) (*malloc_ptr)(s)
#define free(s) (*free_ptr)(s)
```

Figure 3: The malloc hack

picture of program, shared C library.
malloc pointer and init code
indirect calls from library code

Then they recompiled the entire C library, and added these lines (or the assembler equivalent) to the `.init` section of the stub library, so they are included in every program that uses the shared library.

```
#undef malloc
#undef free

malloc_ptr = &malloc;
free_ptr = &free;
```

Since the stub library is bound into the application, not the shared library, its references to malloc and free are resolved at the time each program is linked. If there's a private version of malloc and free, it puts pointers to them in the pointers, otherwise it will use the standard library version. Either way, the library and the application use the same version of malloc and free.

Although the implementation of this trick was a maintenance nightmare, the idea that intra-library calls can be made through pointers that are resolved at program runtime is a good one, so long as it's automated and doesn't require fragile manual source code tweaks. We'll find out how the automated version works in the next chapter.

Name conflicts in global data remain a problem with static shared libraries. Consider the small program in Figure 4. If you compile and link it with any of the shared libraries we described in this chapter, it will print a status code of zero rather than the correct error code. That's because

```
int errno;
```

defines a new instance of errno which isn't bound to the one in the shared library. If you comment out the `extern`, the program works, because now it's an undefined global reference which the linker binds to the errno in the shared library. As we'll see, dynamic linking solves this problem as well, at some cost in performance.

Figure 4: Address conflict example

```
#include <stdio.h>

/* extern */ int errno;

main() {
  unlink("/non-existent-file");
  printf("Status was %d\n", errno); }
```

Exercises

If you look in a /shlib directory on a Unix system with shared libraries, you'll usually see three or four versions of each library with names like `libc_s.2.0.1` and `libc_s.3.0.0`. Why not just have the most recent one?

In a stub library, why is it important to include all of the undefined globals for each routine, even if the undefined global refers to another routine in the shared library?

What difference does it make if a stub library is a single large executable with all of the library's symbols as in COFF or Linux, or an actual library with separate modules?