Libraries

Every modern linker handles libraries, collections of object files that are included as needed in a linked program. In this chapter we cover traditional statically linked libraries, leaving the more complex shared libraries to Chapters 9 and 10.

Purpose of libraries

In the 1940s and early 1950s, programming shops had actual code libraries containing reels of tape or later decks of cards that a programmer would visit and select routines to load with his program. Once loaders and linkers started to resolve symbolic references, it became possible to automate the process by selecting routines from the library that resolve otherwise undefined symbols.

A library file is fundamentally no more than a collection of object files, usually with some added directory information to make it faster to search. As always, the details are more complicated than the basic idea, so we work them out in this chapter.

In this chapter, we use the term file to refer to a separate object file, and module to refer to an object file included in a library.

Library formats

The simplest library formats are just sequences of object modules. On sequential media like magnetic or paper tape, there's little point in adding a directory since the linker has to read through the whole library anyway, and skipping over library members is no slower than reading them in. On disks, though, a directory can speed up library searching considerably and is now a standard facility.

Using the operating system

OS/360 and its descendants including MVS provide *partitioned data sets*(PDS), that contain named members, each of which can be treated as a sequential file. The system provides features for giving multiple aliases to a single member, for treating multiple PDS as a single logical PDS for the duration of a program, for enumerating the names in a logical PDS, and of

course for reading or writing the members. Member names are eight characters which probably not coincidentally is the length of an external symbol in a linker. (MVS introduces an extended PDS or PDSE which has some support for names up to 1024 characters, for the benefit of C, C++, and Cobol programmers.)

A linker library is merely a PDS where each member is an object file named by its entry point. Object files that define multiple global symbols have an alias for each global symbol manually created when the library is built. The linker searches the logical PDS specified as the library for members whose names match undefined symbols. An advantage of this scheme is that there's no object library update program needed, since the standard file maintenance utilities for PDS suffice.

Although I've never seen a linker do so, a linker on a Unix-like system could handle libraries the same way; the library would be a directory, the members object files within the directory, with each file name being a global symbol defined in the file. (UNIX permits multiple names for a single file.)

Unix and Windows Archive files

UNIX linker libraries use an ''archive'' format which can actually be used for collections of any types of files, although in practice it's rarely used for anything else. Libraries consist of an archive header, followed by alternating file headers and object files. The earliest archives had no symbol directories, just a set of object files, but later versions had various sorts of directories, settling down to one used for about a decade in BSD versions (text archive headers and a directory called `__.SYMDEF`) and the current version used with COFF or ELF libraries (text archive headers with an extension for long file names, directory called `/`) in System V.4, later versions of BSD, and Linux. Windows ECOFF libraries use the same archive format as COFF libraries, but the directory, although also called `/`, has a different format.

Unix archives

All modern Unix systems use minor variations of the same archive format, Figure 1. The format uses only text characters in the archive headers,

which means that an archive of text files is itself a text file (a quality that has turned out in practice to be useless.) Each archive starts with the "magic" eight character string `!<arch>\n`, where `\n` is a new line. Each archive member is preceded by a 60 byte header containing:

- The name of the member, padded to 16 characters as described below.
- The modification date, as a decimal number of seconds since the beginning of 1970.
- The user and group IDs as decimal numbers.
- The UNIX file mode as an octal number.
- The size of the file in bytes as a decimal number. If the file size is odd, the file's contents are padded with a newline character to make the total length even, although the pad character isn't counted in the size field.
- The two characters reverse quote and newline, to make the header a line of text and provide a simple check that the header is indeed a header. Each member header contains the modification time, user and group IDs and file mode, although linkers ignore them.

Figure 1: Unix archive format

File header:
`!<arch>\n`
Member header:

```
char name[16]; /* member name */
char modtime[12]; /* modification time */
char uid[6]; /* user ID */
char gid[6]; /* group ID */
char mode[8];/* octal file mode */
char size[10]; /* member size */
char eol[2]; /* reverese quote, newline */
```

Member names that are 15 characters or less are followed by enough spaces to pad the name to 16 characters, or in COFF or ELF archives, a slash followed by enough spaces to pad the total to 16 characters. (Unix and Windows both use slashes to separate components in filenames.) The a.out archives didn't support member names longer than 16 characters, reflecting pre-BSD Unix file system that limited file names to 14 characters per component. (Some BSD archives actually did have a provision for longer file names, but since linkers didn't handle the longer names correctly, nobody used them.) COFF, ELF and Windows archives store names longer than 16 characters in an archive member called `//`. This member contains the long names separated by a slash, newline pair on Unix or a null character on Windows. The name field of the header for member with a long name contains a slash followed by the decimal offset in the `//` member of the name string. In Windows archives, the `//` member must be the third member of the archive. In Unix archives the member need not exist if there are no long names, but follows the symbol directory if it does.

Although the symbol directory formats have varied somewhat, they are all functionally the same, mapping names to member positions so linkers can directly move to and read the members they need to use.

The a.out archives store the directory in a member called `__.SYMDEF` which has to be the first member in the archive, Figure 2. (The member's The member starts with a word containing the size in bytes of the symbol table that follows it, so the number of entries in the table is 1/8 of the value in that word. Following the symbol table is a word containing the size of the string table, and the string table, each string followed by a null byte. Each symbol table entry contains a zero-based offset into the string table of the symbol's name, and the file position of the header of the member that defines the symbol. The symbols table entries are conventionally in the order of the members in the file.

Figure 2: SYMDEF directory format

```
int tablesize; /* size in bytes of following table */
struct symtable {
```

```
  int symbol; /* offset in string table */
  int member; /* member pointer */
} symtable [];
int stringsize;    /* size of string table */
char strings[];    /* null terminated strings */
```

COFF and ELF archives use the otherwise impossible name / for the symbol directory rather than `__.SYMDEF` and use a somewhat simpler format, Figure 3. The first four byte value is the number of symbols. Following that is an array of file offsets of archive members, and a set of null terminated strings. The first offset points to the member that defines the symbol named by the first string, and so forth. COFF archives usually use a big-endian byte order for the symbol table regardless of the native byte order of the architecture.

Figure 3: COFF / ELF directory format

```
int nsymbols; /* number of symbols */
int member[]; /* member offsets */
char strings[];  /* null terminated strings */
```

Microsoft ECOFF archives add a second symbol directory member, Figure 4, confusingly also called / that follows the first one.

Figure ECOFF second symbol directory:

```
int nmembers; /* count of member offsets */
int members[]; /* member offsets */
int nsymbols; /* number of symbols */
ushort symndx[]; /* pointers to member offsets */
char strings[];  /* symbol names, in alphabetical order */
```

The ECOFF directory consists of a count of member entries followed by an array of member offsets, one per archive member. Following that is a count of symbols, an array of two-byte member offset pointers, followed by the null terminated symbols in alphabetical order. The member offset pointers contain the one-based index in the member offset table of the member that defines the corresponding symbol. For example, to locate the member corresponding to the fifth symbol, consult the fifth entry in the pointer array which contains the index in the members array of the offset of the defining member. In theory the sorted symbols allow faster searching, but in practice the speedup is not likely to be large, since linkers typically scan the entire table looking for symbols to load, anyway.

Extension to 64 bits

Even if an archive contains objects for a 64 bit architecture, there's no need to change the archive format for ELF or ECOFF unless the archive grows greater than 4GB. Nonetheless some 64 bit architectures have a different symbol directory format with a different member name such as `/SYM64/`.

Intel OMF libraries

The final library format we look at is that used for Intel OMF libraries. Again, a library is a set of object files with a directory of symbols. Unlike the Unix libraries, the directory is at the end of the file, Figure 5.

Figure 5: OMF libraries

LIBHED record
first object module (file)
second object module (file) ...
LIBNAM module names record
LIBLOC module locations record
LIBDIC symbol directory

The library starts with a LIBDIC record that contains the file offset of the LIBNAM record in a (block,offset) format used by Intel's ISIS operating system. The LIBNAM simply contains a list of module names, each name preceded by a count byte indicating the length of the name. The LIBLOC record contains a parallel list of (block,offset) file locations where each module starts. The LIBDIC contains a list of groups of counted strings with the names defined in each module, each group followed by a null byte to separate it from the subsequent group.

Although this format is a little clunky, it contains the necessary information and does the job.

Creating libraries

Each archive format has its own technique for creating libraries. Depending on how much support the operating system provides for the archive format, library creation can involve anything from standard system file management programs to library-specific tools.

At one end of the spectrum, IBM MVS libraries are created by the standard IEBCOPY utility that creates partitioned data sets. In the middle, Unix libraries are created by the ''ar'' command that combines files into archives. For a.out archives, a separate program called ranlib added the symbol directory, reading the symbols from each member, creating the `__.SYMDEF` member and splicing it into the file. In principle ranlib could have created the symbol directory as a real file, then called ar to insert it in the archive, but in practice ranlib manipulated the archive directly. For COFF and ELF archives, the function of ranlib has moved into ar, which creates the sybol directory if any of the members appear to be object modules, although ar still can create archives of non-objects.

At the other end of the spectrum, OMF archives and Windows ECOFF archives are created by specialized librarian programs, since those formats have never been used for anything other than object code libraries.

One minor issue for library creation is the order of object files, particularly for the ancient formats that didn't have a symbol directory. Pre-ranlib Unix systems contained a pair of programs called lorder and tsort to help create archives. Lorder took as its input a set of object files (not libraries),

and produced a dependency list of what files refered to symbols in what other files. (This is not hard to do; lorder was and still is typically implemented as a shell script that extracts the symbols using a symbol listing utility, does a little text processing on the symbols, then uses standard sort and join utilities to create its output.) Tsort did a topological sort on the output of lorder, producing a sorted list of files so each symbol is defined after all the references to it, allowing a single sequential pass over the files to resolve all undefined references. The output of lorder was used to control ar.

Although the symbol directories in modern libraries allow the linking process to work regardless of the order of the objects within a library, most libraries are still created with lorder and tsort to speed up the linking process.

Searching libraries

After a library is created, the linker has to be able to search it. Library search generally happens during the first linker pass, after all of the individual input files have been read. If the library or libraries have symbol directories, the linker reads in the directory, and checks each symbol in turn against the linker's symbol table. If the symbol is used but undefined, the linker includes that symbol's file from the library. It's not enough to mark the file for later loading; the linker has to process the symbols in the segments in the library file just like those in an explicitly linked file. The segments go in the segment table, and the symbols, both defined and undefined are entered into the global symbol table. It's quite common for one library routine to refer to symbols in another library routine, for example, a higher level I/O routine like `printf` might refer to a lower level `getc` or `read` routine.

Library symbol resolution is an interative process. After the linker has made a pass over the symbols in the directory, if it included any files from the library during that pass, it should make another pass to resolve any symbols required by the included files, until it makes a complete pass over the directory and finds nothing else to include. Not all linkers do this; many just make a single sequential pass over the directory and miss any backwards dependencies from a file to another file earlier in the library.

Tools like tsort and lorder can minimize the difficulty due to single-pass linkers, but it's not uncommon for programmers to explcitly list the same library several times on the linker command line to force multiple passes and resolve all the symbols.

Unix linkers and many Windows linkers take an intermixed list of object files and libraries on the command line or in a control file, and process each in order, so that the programmer can control the order in which objects are loaded and libraries are searched. Although in principle this offers a great deal of flexibility and the ability to interpose private versions of library routines by listing the private versions before the library versions, in practice the ordered search provides little extra utility. Programmers invariably list all of their object files, then any application-specific libraries, then system libraries for math functions, network facilities and the like, and finally the standard system libraries. There are rarely any duplicated symbols among the libraries, and if the linker simply searched them all as a group, the programmers would be well served.

The primary exception to this rule is that applications sometimes define private versions of a few routines, notably `malloc` and `free`, for heap storage management, and want to use them rather than the standard system versions. For that case, a linker flag specifically saying ‘‘don't look for these symbols in the library’’ would in most cases be preferable to getting the effect by putting the private malloc in the search order in front of the public one.

Performance issues

The primary performance issue related to libraries used to be the time spent scanning libraries sequentially. Once symbol directories became standard, reading an input file from a library became insignificantly slower than reading a separate input file, and so long as libraries are topologically sorted, the linker rarely needs to make more than one pass over the symbol directory.

Library searches can still be slow if a library has a lot of tiny members. A typical Unix system library has over 600 members. Particularly in the now-common case that all of the library members are combined at runtime into a single shared library anyway, it'd probably be faster to create a sin-

gle object file that defines all of the symbols in the library and link using that rather than searching a library. We examine this in more detail in Chapter 9.

Weak external symbols

The simple definition-reference model used for symbol resolution and library member selection turns out to be insufficiently flexible for many applications. For example, most C programs call routines in the `printf` family to format data for output. Printf can format all sorts of data, including floating point, which means that any program that uses printf will get the floating point libraries linked in even if the program doesn't actually use floating point.

For many years, PDP-11 Unix programs had to trick the linker to avoid linking the floating libraries in integer-only programs. The C compiler emitted a reference to the special symbol `fltused` in any routine that used floating point code. The C library was arranged as in Figure 6, taking advantage of the fact that the linker searched the library sequentially. If the program used floating point, the reference to fltused would cause the real floating point routines to be linked, including the real version of fcvt, the floating output routine. Then when the I/O module was linked to define printf, there was already a version of fcvt that satisfyed the reference in the I/O module. In programs that didn't use floating point, the real floating point routines wouldn't be loaded, since there wouldn't be any undefined symbols they resolved, and the reference to fcvt in the I/O module would be resolved by the stub floating routines that follow the I/O routines in the library.

Figure 6: Unix classic C library

...
Real floating point module, define fltused and fcvt
I/O module, defines printf, refers to fcvt
Stub floating routines, define stub fcvt
...

While this trick works, using it for more than one or two symbols would rapidly become unwieldy, and its correct operation critically depends on the order of the modules in the library, something that's easy to get wrong when the library's rebuilt.

The solution to this dilemma is weak external symbols, external symbols that do not cause library members to be loaded. If a definition for the symbol is available, either in an explicitly linked file or due to a normal external causing a library member to be linked, a weak external is resolved like a normal external reference. But if no definition is available, the weak external is left undefined and in effect resolved to zero, which is not considered to be an error. In the case above, the I/O module would make a weak reference to fcvt, the real floating point module would follow the I/O module in the library, and no stub routines would be necessary. Now if there's a reference to fltused, the floating point routines are linked and define fcvt. If not, the reference to fcvt remains unresolved. This no longer is dependent on library order, and will work even if the library makes multiple resolution passes over the library.

ELF adds yet another kind of weak symbol, a weak definition as well as a weak reference. A weak definition defines a global symbol if no normal definition is available. If a normal definition is available, the weak definition is ignored.

Exercises

What should a linker do if two modules in different libraries define the same symbol? Is it an error?

Library symbol directories generally include only defined global symbols. Would it be useful to include undefined global symbols as well?

When sorting object files using lorder and tsort, it's possible that tsort won't be able to come up with a total order for the files. When will this happen, and is it a problem?

Some library formats put the directory at the front of the library while others put it at the end. What practical difference does it make?

Describe some other situations where weak externals and weak definitions are useful.