8. SHARED LIBRARIES

Introduction

Efficient use of disk storage space, memory, and computer power is becoming increasingly important. A shared library can offer savings in all three areas. For example, if constructed properly, a shared library can make **a.out** files (executable object files) smaller on disk storage and processes (**a.out** files that are executing) smaller in memory.

The first part of this chapter, "Using a Shared Library," is designed to help you use the shared libraries. It describes what a shared library is and how to use one to build **a.out** files. It also offers advice about when and when not to use a shared library and how to determine whether an **a.out** uses a shared library.

The second part in this chapter, "Building a Shared Library," describes how to build a shared library. You do not need to read this part to use shared libraries. It addresses library developers, advanced programmers who are expected to build their own shared libraries. Specifically, this part describes how to use the operating system tool **mkshlib**(1) (documented in the *Programmer's Reference Manual*) and how to write C code for shared libraries. An example is included. Read this part of the chapter only if you have to build a shared library.

NOTE

> Shared libraries are a new feature of Release 3. An executable object file that needs shared libraries will not run on previous releases of the operating system.

Using a Shared Library

If you are accustomed to using libraries to build your applications programs, shared libraries should blend into your work easily. This part of the chapter explains what shared libraries are and how and when to use them.

What is a Shared Library?

A shared library is a file containing object code that several **a.out** files may use simultaneously while executing. A shared library, like a library that is not shared, is an archive file. For simplicity, however, we refer to an archive file with shared library members as a shared library and one without as an archive library.

When a program is compiled or link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the operating system executes the resulting **a.out** file, it uses the information in this section to bring the required shared library code into the address space of the process.

A shared library offers several benefits by not copying code into **a.out** files. It can:

- save disk storage space

  Because shared library code is not copied into all the **a.out** files that use the code, these files are smaller and use less disk space.

- save memory

  By sharing library code at run time, the dynamic memory needs of processes are reduced.

- make executable files using library code easier to maintain

  As mentioned above, shared library code is brought into a process' address space at run time. Updating a shared library effectively updates all executable files that use the library, because the operating system brings the updated version into new processes. If an error in shared library code is fixed, all processes automatically use the corrected code.

  Archive libraries cannot, of course, offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

"Deciding Whether to Use a Shared Library" in this chapter describes shared libraries in more detail.

Shared Libraries Provided with the Operating System

The C shared library is provided with Release 3 and later releases; the networking library included with the Networking Support Utilities is also a shared library.

These libraries, like all shared libraries, are made up of two files called the host library and the target library. The host library is the file that the link editor searches when linking programs to create the **.lib** sections in **a.out** files; the target library is the file that the operating system uses when running those files. Naturally, the target library must be present for the **a.out** file to run.

Shared Library	Host Library Command Line Option	Target Library Path Name
C Library	**−lc_s**	**/shlib/libc_s**
Networking Library	**−lnsl_s**	**/shlib/libnsl_s**

Notice the **_s** suffix on the library names; we use it to identify both host and target shared libraries. For example, it distinguishes the standard relocatable C library **libc** from the shared C library **libc_s**. The **_s** also indicates that the libraries are statically linked.

The relocatable C library is still available on the operating system; this library is searched by default during the compilation or link editing of C programs. All other archive libraries from previous releases of the system are also available. Just as you use the archive libraries' names, you must use a shared library's name when you want to use it to build your **a.out** files. You tell the link editor its name with the **−l** option, as shown below.

Building an a.out File

You direct the link editor to search a shared library the same way you direct a search of an archive library on the operating system:

cc *file*.**c** **−o** *file* ... **−l***library_file* ...

To direct a search of the networking library, for example, you use the following command line:

cc *file*.**c** **−o** *file* ... **−lnsl_s** ...

And to link all the files in your current directory together with the shared C library, you'd use the following command line:

```
cc *.c -lc_s
```

Normally, you should include the **–lc_s** argument after all other **–l** arguments on a command line. The shared C library will then be treated like the relocatable C library, which is searched by default after all other libraries specified on a command line are searched.

Coding an Application

Application source code in C or assembly language is compatible with both archive libraries and shared libraries. As a result, you should not have to change the code in any applications you already have when you use a shared library with them. When coding a new application for use with a shared library, you should just observe your standard coding conventions.

However, do keep the following two points in mind, which apply when using either an archive or a shared library:

- Don't define symbols in your application with the same names as those in a library.

  Although there are exceptions, you should avoid redefining standard library routines, such as **printf**(3S) and **strcmp**(3C). Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.

- Don't use undocumented archive routines.

  Use only the functions and data mentioned on the manual pages describing the routines in Section 3 of the *Programmer's Reference Manual*. For example, don't try to outsmart the **ctype** design by manipulating the underlying implementation.

Deciding Whether to Use a Shared Library

You should base your decision to use a shared library on whether it saves space in disk storage and memory for your program. A well-designed shared library almost always saves space. So, as a general rule, use a shared library when it is available.

To determine what savings are gained from using a shared library, you might build the same application with both an archive and a shared library, assuming both kinds of library are available. Remember, that you may do this because source code is compatible between shared libraries and archive libraries. (See the above section "Coding an Application.") Then compare the two versions of the application for size and performance.

The following example models this procedure:

```
$ cat hello.c
main()
{
   printf("Hello\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc_s
$ size unshared shared
unshared: 8680 + 1388 + 2248 = 12316
shared: 300 + 680 + 2248 = 3228
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. The following section gives a more detailed discussion about when a shared library does and does not save space.

When making your decision about using shared libraries, also remember that they are not available on releases prior to Release 3. If your program must run on previous releases, you will need to use archive libraries.

More About Saving Space

This section is designed to help you better understand why your programs will usually benefit from using a shared library. It explains:

- how shared libraries save space that archive libraries cannot
- how shared libraries are implemented on the operating system
- how shared libraries might increase space usage

How Shared Libraries Save Space

To better understand how a shared library saves space, we need to compare it to an archive library.

A host shared library resembles an archive library in three ways. First, as noted earlier, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside and made available outside the library are called exported symbols. Note that the library may also have imported symbols, symbols that it uses but usually does not define. Third, the link editor searches the library for these symbols when linking a program to resolve its external references. Resolving the references produces an executable version of the program, the **a.out** file.

NOTE

> Note that the link editor is a static linking tool; static linking requires that all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both an archive library and a shared library.

Although these similarities exist, a shared library differs significantly from an archive library. The major differences relate to how the libraries are handled to resolve symbolic references, a topic already discussed briefly.

Consider how the operating system handles both types of libraries during link editing. To produce an **a.out** file using an archive library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor does not copy any code from the library into the program's object file. Instead, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values. When the operating system executes the resulting **a.out** file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

Figure 8-1 depicts the **a.out** files produced using a regular archive version and a shared version of the standard C library to compile the following program:

```
main()
{
   ...
   printf( "How do you like this manual?\n" );
   ...
   result = strcmp( "I do.", answer );
   ...
}
```

Notice that the shared version is smaller. Figure 8-2 depicts the process images in memory of these two files when they are executed.

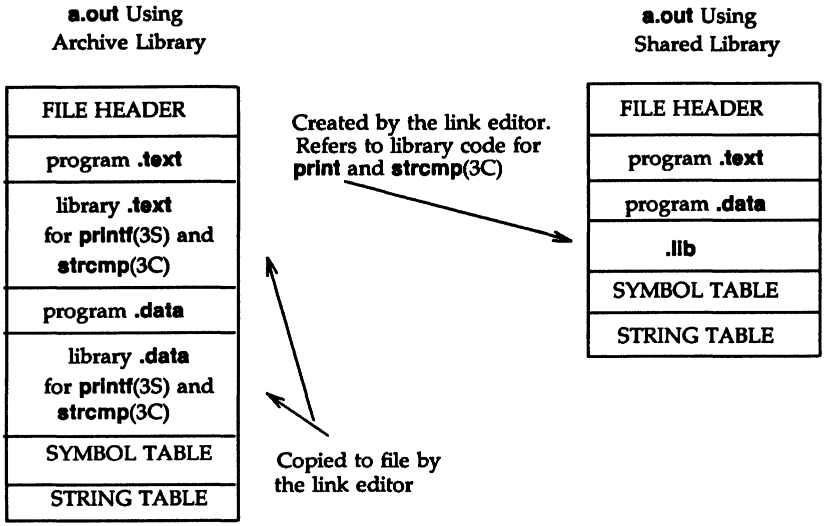

Figure 8-1. a.out Files Created Using an Archive Library and a Shared Library

Now consider what happens when several **a.out** files need the same code from a library. When using an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the **a.out** files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the **a.out** files, as shown in Figure 8-2. This separation enables all processes using the same shared library to reference a single copy of the code.

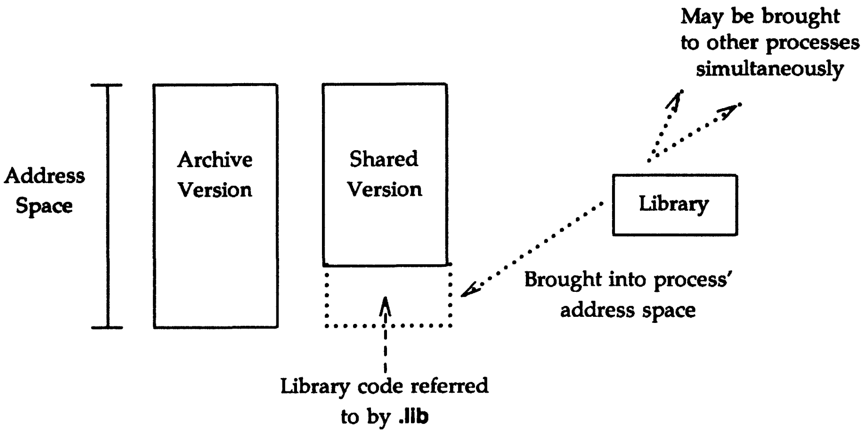

Figure 8-2. Processes Using an Archive and a Shared Library

How Shared Libraries Are Implemented

Now that you have a better understanding of how shared libraries save space, you need to consider their implementation to understand how they might increase space usage (this seldom happens).

The Host Library and Target Library

As previously mentioned, every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which you build an **a.out** file; the target machine is the machine on which you run the file. Of course, the host and target may be the same machine, but they don't have to be.

The host library is just like an archive library. Each of its members (typically a complete object file) defines some text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation or link editing of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, as mentioned earlier, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the **a.out** file mentioned earlier (see the section "What is a Shared Library?") and shown in Figure 8-1 as **.lib**.

The target library used for execution resembles an **a.out** file. The operating system reads this file during execution if a process needs a shared library. The special **.lib** section in the **a.out** file tells which shared libraries are needed. When the operating system executes the **a.out** file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them. That is, the operating system uses the same physical code for each process that attaches a shared library's text.

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region (contiguous area of virtual address space that mirrors the **.data** section of the target library). So that they do not interfere with one another, processes that share text do not share data and stack area.

As suggested above, the target library is a lot like an **a.out** file, which can also share its text, but not its data. Also, a process must have execute permission for a target library to execute an **a.out** file that uses the library.

The Branch Table

When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. This is because a static linking loader like **ld** binds symbols to addresses during link editing. In this way, the **a.out** file for the program has an address for each referenced symbol.

What happens if library code is updated and the address of a symbol changes? Nothing happens to an **a.out** file built with an archive library, because that file already has a copy of the code defining the symbol. (Even though it isn't the updated copy, the **a.out** file will still run.) However, the change can adversely affect an **a.out** file built with a shared library. This file has only a symbol telling where the required library code is. If the library code were updated, the location of that code might change. Therefore, if the **a.out** file ran after the change took place, the operating system could bring in the wrong code. To keep the **a.out** file current, you might have to recompile a program that uses a shared library after each library update.

To avoid the need to recompile, a shared library is implemented with a branch table. A branch table associates text symbols with an absolute address that does not change even when library code is changed. Each address labels a jump instruction to the address of the code that defines a symbol. Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 8-3 shows two **a.out** files executing that call **printf**(3S). The process on the left was built using an archive library. It already has a copy of the library code defining the **printf**(3S) symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

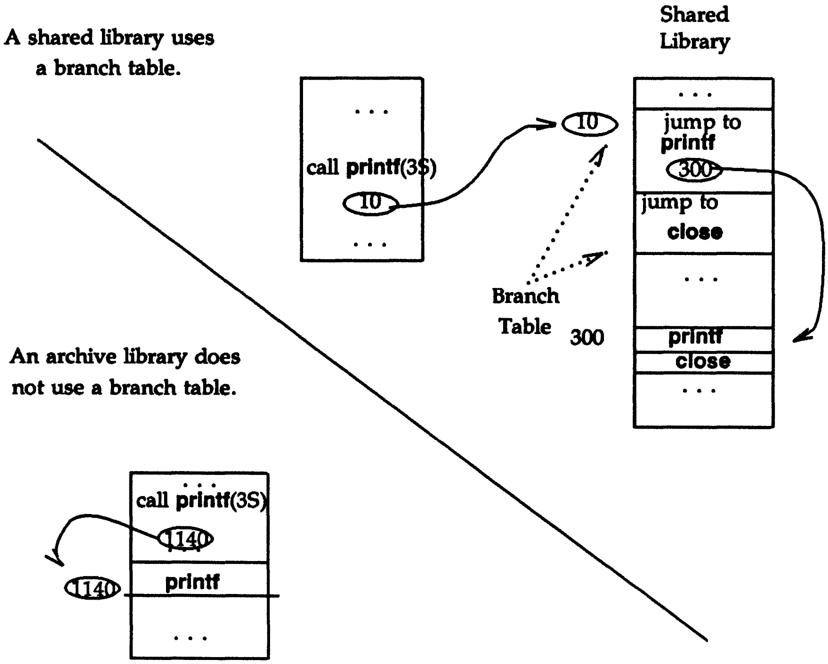

Figure 8-3. A Branch Table in a Shared Library

How Shared Libraries Might Increase Space Usage

A host library might add space to an **a.out** file. Recall that the link editor uses static linking, which requires that all external references in a program be resolved before it is executed. Also recall that a shared library may have imported symbols, which are used but not defined by the library. These symbols might introduce unresolved references during the linking process. To resolve these references, the link editor has to add the **.text** and **.data** sections defining the referenced imported symbols to the **a.out** file. These sections increase the size of the **a.out** file.

A target library might also add space to a process. Again recall from "How Shared Libraries are Implemented" in this chapter that a shared library's target file may have both text and data regions connected to a process. While the text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. Naturally, this region adds to the process's memory requirements. As a result, if an application uses only a small part of a shared library's text and data, executing the application might require more memory with a shared library than without one. For example, it would not be wise to use the shared C library to access only **strcmp**(3C). Although sharing **strcmp**(3C) saves disk storage and memory, the memory cost for sharing all the shared C library's private data region outweighs the savings. The archive version of the library would be more appropriate.

Identifying a.out Files that Use Shared Libraries

Suppose you have an executable file and you want to know whether it uses a shared library. You can use the **dump**(1) command (documented in the *Programmer's Reference Manual*) to look at the section headers for the file:

```
dump –hv a.out
```

If the file has a **.lib** section, a shared library is needed. If the **a.out** does not have a **.lib** section, it does not use shared libraries.

With a little more work, you can even tell what libraries a file uses by looking at the **.lib** section contents:

```
dump –L a.out
```

Debugging a.out Files that Use Shared Libraries

Debugging support for shared libraries is currently limited. Shared library data are not dumped to core files, and **sdb**(1) (documented in the *Programmer's Reference Manual*) does not read shared libraries' symbol tables. If you encounter an error that appears not to be in your application code, you may find debugging easier if you rebuild the application with the archive version of the library used.

Building a Shared Library

This part of the chapter explains how to build a shared library. It covers the major steps to the building process, the use of the **mkshlib**(1) tool to build the host and target libraries, and some guidelines for writing shared library code.

This part assumes that you are an advanced C programmer faced with the task of building a shared library. It also assumes you are familiar with the archive library building process. You do not need to read this part of the chapter if you only plan to use the standard distributed shared libraries or other shared libraries that have already been built.

The Building Process

To build a shared library on the operating system, you have to complete six major tasks:

- Choose region addresses.
- Choose the path name for the shared library target file.
- Select the library contents.
- Rewrite existing library code to be included in the shared library.
- Write the library specification file.
- Use the **mkshlib** tool to build the host and target libraries.

Each of these tasks is discussed here.

Step 1: Choosing Region Addresses

The first thing you need to do is choose region addresses for your shared library.

Shared library regions correspond to memory management unit (MMU) segment size, each of which is 128 KB. The following table gives a list of the segment assignments (as of the copyright date for this guide) and shows what virtual addresses are available for libraries you might build.

Start Address	**Contents**	**Target Path Name**
0x80000000 ... 0x803E0000	Reserved for Motorola Shared C Library Networking Library	**/shlib/libc_s** **/shlib/libnsl_s**
0x80400000 0x80420000	Generic Database Library	Unassigned
0x80440000 0x80460000	Generic Statistical Library	Unassigned
0x80480000 0x804A0000	Generic User Interface Library	Unassigned
0x804C0000 0x804E0000	Generic Screen Handling Library	Unassigned
0x80500000 0x80520000	Generic Graphics Library	Unassigned
0x80540000 0x80560000	Generic Networking Library	Unassigned
0x80580000 ... 0x80660000	Generic – to be defined	Unassigned
0x80680000 ... 0x807E0000	For private use	Unassigned

What does this table tell you? First, the shared C library and the networking library reside at the path names given above and use addresses in the range reserved for the operating system. If you build a shared library that uses reserved addresses, you run the risk of conflicting with future Motorola products.

Second, a number of segments are allocated for shared libraries that provide various services such as graphics, database access, and so on. These categories are intended to reduce the chance of address conflicts among commercially available libraries. Although two libraries of the same type may conflict, that doesn't matter. A single process should not usually need to use two shared libraries of the same type. If the need arises, a program can use one shared library and one archive library.

NOTE

> Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single **a.out** file doesn't need to use both libraries.

Third, several segments are reserved for private use. If you are building a large system with many **a.out** files and processes, shared libraries might improve its performance. As long as you don't intend to release the shared libraries as separate products, you should use the private region addresses. You can put your shared libraries into these segments and avoid conflicting with commercial shared libraries. You should also use the private areas when you will own all the **a.out** files that access your shared library. Don't risk address conflicts.

NOTE

> If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well. Some of your customers might not find the shared library appropriate for their applications. Others might want their applications to run on versions of the operating system without shared library support.

Step 2: Choosing the Target Library Path Name

After you choose the region addresses for your shared library, you should choose the path name for the target library. We chose **/shlib/libc_s** for the shared C library and **/shlib/libnsl_s** for the networking library. (As mentioned earlier, we use the **_s** suffix in the path names of all statically linked shared libraries.) To choose a path name for your shared library, consult the established list of names for your computer or see your system administrator. Also keep in mind that shared libraries needed to boot the operating system should normally be located in **/shlib**; other application libraries normally reside in **/usr/lib** or in private application directories. Of course, if your shared library is for personal use, you can choose any convenient path name for the target library.

Step 3: Selecting Library Contents

Selecting the contents for your shared library is the most important task in the building process. Some routines are prime candidates for sharing; others are not. For example, it's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. What you include will depend on the individual needs of the programmers and other users for whom you are building the library. There are some general guidelines you should follow, however. They are discussed in the section "Choosing Library Members" in this chapter. Also see the guidelines in the "Importing Symbols" and "Tuning the Shared Library Code" sections.

Step 4: Rewriting Existing Library Code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See the "Changing Existing Code for the Shared Library" section in this chapter.

Step 5: Writing the Library Specification File

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. An example specification file is shown in the next section, "An Example." The contents and format of the specification file are given by the following directives (see also the **mkshlib**(1) manual page):

#address *sectname address*

Specifies the start address, *address*, of section *sectname* for the target file. This directive is typically used to specify the start addresses of the **.text** and **.data** sections.

#target *pathname*

Specifies the path name, *pathname*, of the target shared library on the target machine. This is the location where the operating system looks for the shared library during execution. Normally, *pathname* will be an absolute path name, but it does not have to be.

This directive can be specified only once per shared library specification file.

#branch Starts the branch table specifications. The lines following this directive are taken to be branch table specification lines.

Branch table specification lines have the following format:

funcname <white space> *position*

The *funcname* is the name of the symbol given a branch table entry and *position* specifies the position of *funcname*'s branch table entry. The *position* may be a single integer or a range of integers of the form *position1-position2*. Each *position* must be greater than or equal to one. The same position cannot be specified more than once, and every position from one to the highest given position must be accounted for.

If a symbol is given more than one branch table entry by associating a range of positions with the symbol or by specifying the same symbol on more than one branch table specification line, then the symbol is defined to have the address of the highest associated branch table entry. All other branch table entries for the symbol can be thought of as empty slots and can be replaced by new entries in future versions of the shared library.

Finally, only functions should be given branch table entries, and those functions must be external.

This directive can be specified only once per shared library specification file.

#objects Specifies the names of the object files constituting the target shared library. The lines following this directive are taken to be the list of input object files in the order they are to be loaded into the target. The list simply consists of each filename followed by white space. This list of objects will be used to build the shared library.

This directive can be specified only once per shared library specification file.

#init *object* Specifies that the object file, *object*, requires initialization code. The lines following this directive are taken to be initialization specification lines.

Initialization specification lines have the following format:

pimport <white space> *import*

pimport is a pointer to the associated imported symbol, *import*, and must be defined in the current specified object file, *object*. The initialization code generated for each line resembles the C assignment statement:

pimport = **&***import* **;**

The assignments set the pointers to default values. All initializations for a particular object file must be given at once and multiple specifications of the same object file are not allowed.

#ident "*string*" Specifies a string, *string*, to be included in the **.comment** section of the target shared library and the **.comment** sections of every member of the host shared library. Only one **#ident** directive is permitted per shared library specification file.

## Specifies a comment. The rest of the line is ignored.

All directives that are followed by multi-line specifications are valid until the next directive or the end of file.

Step 6: Using mkshlib to Build the Host and Target

The operating system command **mkshlib**(1) builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as**(1), and link editor, **ld**(1). Tools are invoked using **execvp** (see **exec**(2)) which searches directories in a user's $PATH environment variable. Also, prefixes to **mkshlib** are parsed in much the same way as prefixes to the **cc**(1) command and invoked tools are given the prefix, where appropriate. For example, **3b2mkshlib** invokes **3b2ld**. These commands all are documented in the *Programmer's Reference Manual*.

The user input to **mkshlib** consists of the library specification file and command line options. We just discussed the specification file; let's take a look at the options now. The shared library build tool has the following syntax:

mkshlib –s *specfil* **–t** *target* [**–h** *host*] [**–n**] [**–q**] .DE

–s *specfil* Specifies the shared library specification file, *specfil*. This file contains all the information necessary to build a shared library, as described in Step 5. Its contents include the branch table specifications for the target, the path name in which the target should be installed, the start addresses of text and data for the target, the initialization specifications for the host, and the list of object files to be included in the shared library.

–t *target* Specifies the name, *target,* of the target shared library produced on the host machine. When *target* is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section "Writing the Library Specification File"). If the **–n** option is given, then a new target shared library will not be generated.

–h *host* Specifies the name of the host shared library, *host*. If this option is not given, then the host shared library will not be produced.

–n Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **–t** option must still be supplied since a version of the target shared library is needed to build the host shared library.

–q Suppresses the printing of certain warning messages.

An Example

Follow each of the steps in the library building process to build a small example shared library. While building this library, appropriate guidelines will be displayed amidst text. Note that the example code is contrived to show samples of problem areas, not to do anything useful.

The name of our library will be **libexam**. Assume the original code was a single source file, as shown below.

```
/* Original exam.c */
#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

int     count   = 0;
char    *Error;

char *
excopy(s)
        char    *s;
{
        char    *new;

        ++count;
        if ( (new = malloc(strlen(s)+1)) == 0 )
        {
                        Error = "no memory";
                                                         return 0;
        }
        return strcpy(new, s);
}

excount()
{
        fprintf(stderr, "excount %d\n", count);
        return count;
}
```

To begin, let's choose the region address spaces for the library's **.text** and **.data** sections from the segments reserved for private use on VME-based computers; note that the region addresses must be on a segment boundary (128K):

```
.text     0x80680000
.data     0x806a0000
```

Also choose the path name for our target library:

```
/my/directory/libexam_s
```

Now you need to identify the imported symbols in the library code. (See the guidelines in the section about "Importing Symbols": **malloc**, **strcpy**, **strlen**, **fprintf**, and **_iob**.) A header file defines C preprocessor macros for these symbols; note that you don't use **_iob** directly except through the macro **stderr** from <**stdio.h**>. Also notice the **_libexam_** prefixes for the symbols. The pointers for imported symbols are exported and, therefore, might conflict with other symbols. Using the library name as a prefix reduces the chance of a conflict occurring.

```
/* New file import.h */

#define malloc          (*_libexam_malloc)
#define strcpy          (*_libexam_strcpy)
#define strlen          (*_libexam_strlen)
#define fprintf         (*_libexam_fprintf)
#define _iob            (*_libexam__iob)

extern char             *malloc();
extern char             *strcpy();
extern int              strlen();
extern int              fprintf();
```

NOTE

The file **import.h** does not declare **_iob** as **extern**; it relies on the header file <**stdio.h**> for this information.

You will also need a new source file to hold definitions of the imported symbol pointers. Remember that all global data need to be initialized:

```
/* New file import.c */

#include <stdio.h>

char    *(*_libexam_malloc)()    = 0;
char    *(*_libexam_strcpy)()    = 0;
int     (*_libexam_strlen)()     = 0;
int     (*_libexam_fprintf)()    = 0;
FILE    (*_libexam__iob)[]       = 0;
```

Next, look at the library's global data to see what needs to be visible externally. (See the guideline "Minimize Global Data.") The variable **count** does not need to be external, because it is accessed through **excount()**. Make it static. (This should have been done for the relocatable version.)

Now the library's global data need to be moved into separate source files. (See the guideline "Define Text and Global Data in Separate Source Files.") The only global datum left is **Error**, and it needs to be initialized. (See the guideline "Initialize Global Data.")

Error must remain global, because it passes information back to the calling routine:

```
/* New file global.c */

char   *Error   = 0;
```

Integrating these changes into the original source file, we get the following (notice that the symbol names must be declared as **externs**):

```
/* Modified exam.c */
#include "import.h"

#include <stdio.h>

extern int      strlen();
extern char     *malloc(), *strcpy();

static int      count = 0;
extern char     *Error;

char *
excopy(e)
        char    *e;
{
        char    *new;

        ++count;
        if ( (new = malloc(strlen(e)+1)) == 0 )
        {
               Error = "no memory";
               return 0;
        }
        return strcpy(new, e);
}

excount()
{
        fprintf(stderr, "excount %d\n", count);
        return count;
}
```

NOTE

> The new header file **import.h** must be included before <**stdio.h**>.

Next, we must write the shared library specification file for **mkshlib**:

```
         /* New file libexam.sl */
 1         #target /my/directory/libexam_s
 2         #address .text 0x80680000
 3         #address .data 0x806a0000

 4         #branch
 5                 excopy          1
 6                 excount         2

 7         #objects
 8                 import.o
 9                 global.o
10                 exam.o

11         #init import.o
12                 _libexam_malloc  malloc
13                 _libexam_strcpy  strcpy
14                 _libexam_strlen  strlen
15                 _libexam_fprintf fprintf
16                 _libexam__iob    _iob
```

Briefly, here is what the specification file does. Line 1 gives the path name of the shared library on the target machine. The target shared library must be installed there for **a.out** files that use it to work correctly. Lines 2 and 3 give the virtual addresses for the shared library text and data regions, respectively. Line 4 through 6 specify the branch table. Lines 5 and 6 assign the functions **excopy()** and **excount()** to branch table entries 1 and 2. Only external text symbols, such as C functions, should be placed in the branch table.

Lines 7 through 10 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, a change in the size of static data in **exam.o** would move external data symbols and break compatibility.

Lines 11 through 16 give imported symbol information for the object file **import.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libexam_malloc** will hold a pointer to **malloc**, and so on.

Now, we have to compile the **.o** files as we would for any other library:

```
cc –c import.c global.c exam.c
```

Finally, we need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib –s libexam.sl –t libexam_s –h libexam_s.a
```

Presuming all the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libexam_s.a,** and the target library, **libexam_s**.

Guidelines for Writing Shared Library Code

Because the main advantage of a shared library over an archive library is sharing and the space it saves, these guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility. When appropriate, we describe our experience with building the shared C library to illustrate how following a particular guideline helped us.

We recommend that you read these guidelines once from beginning to end to get a perspective of the things you need to consider when building a shared library. Then use it as a checklist to guide your planning and decision-making.

Before we consider these guidelines, let's consider the restrictions to building a shared library common to all the guidelines. These restrictions involve static linking. Here's a summary of them, some of which are discussed in more detail later. Keep them in mind when reading the guidelines in this section:

- Exported symbols have fixed addresses.

  If an exported symbol moves, you have to re-link all **a.out** files that use the library. This restriction applies both to text and data symbols.

- If the library's text changes for one process at run time, it changes for all processes.

  Therefore, any library changes that apply only to a single process must occur in data, not in text, because only the data region is private. (Besides, the text region is read-only.)

- If the library uses a symbol directly, that symbol's run time value (address) must be known when the library is built.

- Imported symbols cannot be referenced directly.

  Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

Choosing Library Members

Include Large, Frequently Used Routines

These routines are prime candidates for sharing. Placing them in a shared library saves code space for individual **a.out** files and saves memory, too, when several concurrent processes need the same code. **printf**(3S) and related C library routines (which are documented in the *Programmer's Reference Manual*) are good examples.

When we built the shared C library...

The **printf**(3S) family of routines is used frequently. Consequently, we included **printf**(3S) and related routines in the shared C library.

Exclude Infrequently Used Routines

Putting these routines in a shared library can degrade performance, particularly on paging systems. Traditional **a.out** files contain all code they need at run time. By definition, the code in an **a.out** file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single **a.out** file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance. See also "Organize to Improve Locality."

When we built the shared C library...

Our original shared C library had about 44 KB of text. After profiling the code in the library, we removed small routines that were not often used. The current library has under 29 KB of text. The point is that functions used only by a few **a.out** files do not save much disk space by being in a shared library, and their inclusion can cause more paging and decrease performance.

Exclude Routines that Use Much Static Data

These modules increase the size of processes. As "How Shared Libraries are Implemented" and "Deciding Whether to Use a Shared Library" explain, every process that uses a shared library gets its own private copy of the library's data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set.

For example, **getgrent**(3C), which is documented in the *Programmer's Reference Manual,* is not used by many standard operating system commands. Some versions of the module define over 1400 bytes of unshared, static data. It probably should not be included in a shared library. You can import global data, if necessary, but not local, static data.

Exclude Routines that Complicate Maintenance

All exported symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols don't have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of exported data may affect symbol addresses and break compatibility.

Include Routines the Library Itself Needs

It usually pays to make the library self-contained. For example, **printf**(3S) requires much of the standard I/O library. A shared library containing **printf**(3S) should contain the rest of the standard I/O routines, too.

NOTE

> This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Changing Existing Code for the Shared Library

All C code that works in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for archive libraries (although it may be slightly bigger and slower). The guidelines mostly explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data from a shared library to an archive library.

Minimize Global Data

In the current shared library implementation, all external data symbols are global; they are visible to applications. This can make maintenance difficult. You should try to reduce global data, as described below.

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant.

Third, allocate buffers at run time instead of defining them at compile time. This does two important things. It reduces the size of the library's data region for all processes and, therefore, saves memory; only the processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Define Text and Global Data in Separate Source Files

Separating text from global data makes it easier to prevent data symbols from moving. If new exported variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Archive libraries let the link editor extract individual members. This sometimes encourages programmers to define related variables and text in the same source file. This works fine for relocatable files, but shared libraries have a different set of restrictions. Suppose exported variables were scattered throughout the library modules. Then visible and hidden data would be intermixed. Changing hidden data such as a string, like **hello** in the following example, moves subsequent data symbols (even the exported symbols).

```
Before                    Broken Successor

int head = 0;             int head = 0;
func()                    func()
{                         {
   . . .                     . . .
   p = "hello";              p = "hello, world";
   . . .                     . . .
}                         }
int tail = 0;             int tail = 0;
```

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus will be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

Adding new exported variables to a shared library may change the addresses of static symbols, but this doesn't affect compatibility. An **a.out** file has no way to reference static library symbols directly, so it cannot depend on their values. Thus it pays to group all exported data symbols and place them at lower addresses than the static (hidden) data. You can write the specification file to control this. In the list of object files, make the global data files first.

```
#objects
   data1.o
   . . .
   lastdata.o
   text1.o
   text2.o
   . . .
```

If the data modules are not first, a seemingly harmless change (such as a new string literal) can break existing **a.out** files.

Shared library users get all library data at run time, regardless of the source file organization. Consequently, you can put all exported variables' definitions in a single source file without a penalty. You can also use several source files for data definitions.

Initialize Global Data

Initialize exported variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. Using initialized variables is another way to prevent address changes.

The C compilation system on the SYSTEM V/68 operating system puts uninitialized variables in a common area, and the link editor assigns addresses to them in an unpredictable way. In other words, the order of uninitialized symbols may change from one link editor run to the next. So that library developers can preserve compatibility, however, the link editor will not change the order of initialized variables.

Preserve Branch Table Order

You should add new functions only at the end of the branch table. After you have a specification file for the library, try to maintain compatibility with previous versions. You may add new functions without breaking old **a.out** files as long as previous assignments are not changed. This lets you distribute a new library without having to relink all the **a.out** files that used a previous version of the library.

Importing Symbols

Shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 8-4, the symbols **_libc.ptr1** and **_libc.ptr2** are imported from user's code and the symbol **_libc_malloc** from the library itself.

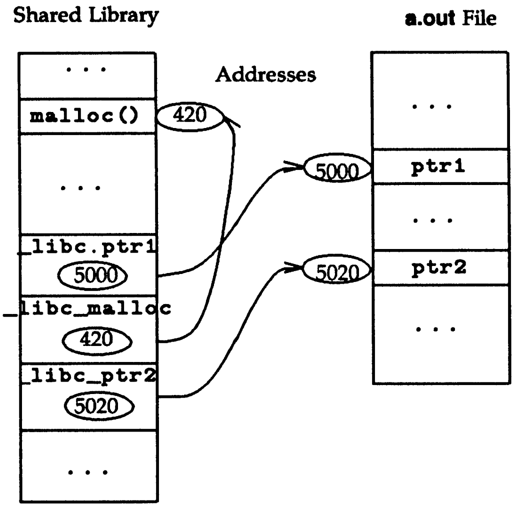

Figure 8-4. Imported Symbols in a Shared Library

The following guidelines describe when and how to use imported symbols.

Imported Symbols that the Library Does Not Define

Archive libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an **a.out** file. Neither target shared libraries nor **a.out** files can have unresolved symbols.

Consequently, shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing archive libraries. For the reasons stated above, it may not be appropriate to include all the archive's modules in the target shared library. If you leave something out that the library calls, you have to make an imported symbol pointer for it.

Imported Symbols that Users Must Be Able to Redefine

Optionally, shared libraries can import their own symbols. At first this might appear to be an unnecessary complication, but consider the following. Two standard libraries, **libc** and **libmalloc**, provide a **malloc** family. Even though most operating system commands use the **malloc** from the C library, they can choose either library or define their own.

When we built the shared C library...

Three possible strategies existed for the shared C library. First, we could have excluded **malloc**(3X). Other library members would have needed it, and so it would have been an imported symbol. This would have worked, but it would have meant less savings.

Second, we could have included the **malloc**(3X) family and not imported it. This would have given us more savings for typical commands, but it had a price. Other library routines call **malloc**(3X) directly, and those calls could not have been overridden. If an application tried to redefine **malloc**(3X), the library calls would not have used the alternate version. Furthermore, the link editor would have found multiple definitions of **malloc**(3X) while building the application. To resolve this the library developer would have to change source code to remove the custom **malloc**(3X), or the developer would have to refrain from using the shared library.

Finally, we could have included **malloc**(3X) in the shared library, treating it as an imported symbol. This is what we did. Even though **malloc**(3X) is in the library, nothing else there refers to it directly. If the application does not redefine **malloc**(3X), both application and library calls are routed to the library version. All calls are mapped to the alternate, if present.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and time overhead to the library, the technique allows a shared library to be one hundred percent compatible with an existing archive at link time and run time.

Mechanics of Importing Symbols

Let's assume a shared library wants to import the symbol **malloc**. The original archive code and the shared library code appear below.

```
Archive Code                          Shared Library Code
                                         /* See pointers.c on next page */

extern char *malloc();           extern char *(*_libc_malloc)();

export()                         export()
{                                {
   ...                              ...
   p = malloc(n);                   p = (*_libc_malloc)(n);
   ...                              ...
}                                }
```

Making this transformation is straightforward, but two sets of source code would be necessary to support both an archive and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The **–I** flag to **cpp**(1) would direct the C preprocessor to look in the appropriate directory to find the desired file.

```
Archive import.h                    Shared import.h
/* empty */                      /*
                                  *  Macros for importing
                                  *  symbols.  One #define
                                  *  per symbol.
                                  */

                                  ...
                                  #define malloc   (*_libc_malloc)
                                  ...
                                  extern char *malloc();
                                  ...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirections for imported symbols. The declaration of **malloc** in **import.h** actually declares the pointer **_libc_malloc**.

```
Common Source

#include "import.h"

extern char *malloc();

export()
{
   ...
   p = malloc(n);
   ...
}
```

Alternatively, one can hide the `#include` with `#ifdef`:

Common Source

```
#ifdef SHLIB
#      include "import.h"
#endif

extern char *malloc();

export()
{
   ...
   p = malloc(n);
   ...
}
```

8

Of course the transformation is not complete. You must define the pointer **_libc_malloc**:

File **pointers.c**

```
char *(*_libc_malloc)() = 0;
```

Note that **_libc_malloc** is initialized to zero, because it is an exported data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of **malloc** must be assigned to **_libc_malloc**. Tools that build the shared library generate the initialization code according to the library specification file.

Pointer Initialization Fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the **a.out** file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches **main**. If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

NOTE

> Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into **a.out** files to set imported pointers to their correct values.

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set **_libc_malloc** to the address of **malloc**:

```
#init pmalloc.o
_libc_malloc    malloc
```

When **mkshlib** builds the host library, it modifies the file **pmalloc.o**, adding relocatable code to perform the following assignment statement:

```
_libc_malloc = &malloc;
```

When the link editor extracts **pmalloc.o** from the host library, the relocatable code goes into the **a.out** file. As the link editor builds the final **a.out** file, it resolves the unresolved references and collects all initialization fragments. When the **a.out** file is executed, the run time startup (**crt1**) executes the initialization fragments to set the library pointers.

Selectively Loading Imported Symbols

Defining fewer pointers in each archive member increases the granularity of symbol selection and can prevent unnecessary objects from being linked into the **a.out** file. For example, if an archive member defines three pointers to imported symbols, the link editor will resolve all three, even though only one might be needed.

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the symbols.

Let's look at some examples. In the coarse method, a single source file might define all pointers to imported symbols:

Old **pointers.c**

```
int (*_libc_ptr1)() = 0;
char *(*_libc_malloc)() = 0;
int (*_libc_ptr2)() = 0;
```

Being able to use them individually requires multiple source files and archive members. Each of the new files defines a single pointer or a small group of related pointers:

File	Contents
ptr1.c	`int (*_libc_ptr1)() = 0;`
pmalloc.c	`char *(*_libc_malloc)() = 0;`
ptr2.c	`int (*_libc_ptr2)() = 0;`

Originally, a single object file, **pointers.o**, defines all pointers. Extracting it requires definitions for **ptr1, malloc,** and **ptr2**. The modified example lets one extract each pointer individually, thus avoiding the unresolved reference for unnecessary symbols.

Providing Archive Library Compatibility

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes. Perhaps the best way to explain this guideline is by example.

When we built the shared C library...

We had an existing archive library to use as the base. This obviously gave us code for individual routines, and the archive library also gave us a model to use for the shared library itself.

We wanted the host library archive file to be compatible with the relocatable archive C library. However, we did not want the shared library target file to include all routines from the archive: including them all would have hurt performance.

Reaching these goals was, perhaps, easier than you might think. We did it by building the host library in two steps. First, we used the available shared library tools to create the host library to match exactly the target. The resulting archive file was not compatible with the archive C library at this point. Second, we added to the host library the set of relocatable objects residing in the archive C library that were missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared C libraries compatible.

Tuning the Shared Library Code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems, such as Release 3. The suggestions come from the experience of building the shared C library.

The archive C library contains several diverse groups of functions. Many processes use different combinations of these groups, making the paging behavior of any shared C library difficult to predict. A shared library should offer greater benefits for more homogeneous collections of code. For example, a data base library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies were more predictable.

Profile the Code

To begin, profile the code that might go into the shared library.

Choose Library Contents

Based on profiling information, make some decisions about what to include in the shared library. **a.out** file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See "Choosing Library Members" in this chapter for more information.

Organize to Improve Locality

When a function is in **a.out** files, it probably resides in a page with other code that is used more often (see "Exclude Infrequently Used Routines"). Try to improve locality of reference by grouping dynamically related functions. If every call of **funcA** generates calls to **funcB** and **funcC**, try to put them in the same page. **cflow**(1) (documented in the *Programmer's Reference Manual*) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for Paging

The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Once again, an example might best explain this guideline.

When we built the shared C library...

We used a VME-based computer to build the library; the architecture of the computer uses 1-KByte pages. Using name lists and disassemblies of the shared library target file, we determined where the page boundaries fell.

After grouping related functions, we broke them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Then we used the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreased.

After determining the branch table, we rearranged the library's object files without breaking compatibility. We put frequently used, unrelated functions together, because we figured they would be called randomly enough to keep the pages in memory. System calls went into another page as a group, and so on. The following example shows how to change the order of the library's object files:

```
Before          After

#objects        #objects
   ...             ...
   printf.o        strcmp.o
   fopen.o         malloc.o
   malloc.o        printf.o
   strcmp.o        fopen.o
   ...             ...
```

Avoid Hardware Thrashing

Finally, you may have to consider the hardware you're using to obtain better performance. You need to consider its memory management. Part of the memory management hardware is an 8-entry cache for translating virtual to physical addresses. Each segment (128 KB) is mapped to one of the eight entries. Consequently, segments 0, 8, 16, ... use entry 0; segments 1, 9, 17, ... use entry 1; and so on.

You get better performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segment mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment. Of course, the next instruction would reference text and flush the cache entry again.

When we built the shared C library...

We avoided the cache entry conflicts. A library's text and data segment numbers (at least with the VME-based computer architecture) should differ by something other than eight.

Making A Shared Library Upward Compatible

The following guidelines explain how to build upward-compatible shared libraries. Note, however, that upward compatibility may not always be an issue. Consider the case in which a shared library is one piece of a larger system and is not delivered as a separate product. In this restricted case, you can identify all **a.out** files that use a particular library. As long as you rebuild all the **a.out** files every time the library changes, versions of the library may be incompatible with each other. This may complicate development, but it is possible.

Comparing Previous Versions of the Library

Shared library developers normally want newer versions of a library to be compatible with previous ones. As mentioned before, **a.out** files will not execute properly otherwise.

The following procedures let you check libraries for compatibility. In these tests, two libraries are said to be compatible if their exported symbols have the same addresses. Although this criterion usually works, it is not foolproof. For example, if a library developer changes the number of arguments a function requires, the new function may not be compatible with the old. This kind of change may not alter symbol addresses, but it will break old **a.out** files.

Let's assume we want to compare two target shared libraries: **new.libx_s** and **old.libx_s**. We use the **nm**(1) command to look at their symbols and **sed**(1) to delete everything except external symbols. A small **sed** program simplifies the job.

New file **cmplib.sed**

```
sed     ´/|extern|.*/!d
        s///
        /^.bt/d
        /^etext /d
        /^edata /d
        /^end /d´
```

The first line of the **sed** script deletes all lines except those for external symbols. The second line leaves only symbol names and values in the output. The last four lines delete special symbols that have no bearing on library compatibility; they are not visible to application programs. You will have to create your own file to hold the **sed** script.

Now we are ready to create lists of symbol names and values for the new and old libraries:

```
nm old.libx_s | sed –f cmplib.sed >old.nm
nm new.libx_s | sed –f cmplib.sed >new.nm
```

Next, we compare the symbol values to identify differences:

```
diff old.nm new.nm
```

If all symbols in the two libraries have the same values, the **diff**(1) command will produce no output, and the libraries are compatible. Otherwise, some symbols are different and the two libraries may be incompatible. **diff**(1), **nm**(1), and **sed**(1) are documented in the *User's Reference Manual.*

Dealing with Incompatible Libraries

When you determine that two libraries are incompatible, you have to deal with the incompatibility. You can deal with it in one of two ways. First, you can rebuild all the **a.out** files that use your library. If feasible, this is probably the best choice. Unfortunately, you might not be able to find those **a.out** files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target path name to the new version of the library. The host and target path names are independent; so you don't have to change the host library path name. New **a.out** files will use your new target library, but old **a.out** files will continue to access the old library.

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library path name for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, **a.out** files that use your library will not work properly.

NOTE

You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

Summary

This chapter described shared libraries and explained how to use them. It also explained how to build your own shared libraries. Using any shared library almost always saves disk storage space, memory, and computer power; and running the operating system on smaller machines makes the efficient use of these resources increasingly important. Therefore, you should normally use a shared library whenever it's available.