USENIX

1986 Summer

Technical Conference & Exhibition

Atlanta, Georgia

Conference Proceedings

Shared Libraries on UNIX™ System V

James Q. Arnold

AT&T

ABSTRACT

As people move the UNIX[1] System to smaller machines, it becomes increasingly important to use disk space, memory, and computer power efficiently. A shared library can offer savings in all three areas. By consolidating multiple copies of library routines, a shared library can make executable files smaller on the disk, make processes take less memory, and reduce paging or swapping activity.

This paper describes a shared library design that lets existing application source and object code use shared libraries. By extending current mechanisms—instead of inventing new ones—the shared library facility preserves the value of existing application code while offering additional benefits. A shared C library, for example, shrank 115 system programs an average of 11,000 bytes per file.

1. INTRODUCTION

Executable files on the UNIX System traditionally have held all the code and initialized data for process images. One layer of sharing exists in the UNIX System: processes created from the same executable file share one copy of the program text in memory. But, if several programs use the same library routines, each executable file holds its own copy of these routines on the disk, and each process has its own copy in memory. A *shared library* lets multiple processes—from different executable files—share a single copy of library code.

Shared libraries make programs' sizes on disk shrink, because executable files no longer hold the code for library routines. Combining numerous copies of common routines also reduces the entire system's memory requirements. Moreover, a shared library lets one fix library errors without rebuilding executable files. By installing a new shared library, old executable files automatically use the updated library.

Before going any farther, I should summarize the properties of our shared library design.

- *Application source code compatibility.* No application source code changes are needed to use shared libraries.
- *Application object file compatibility.* We did not change the C compiler's code generation either for user code or for shared library code. Thus previously existing relocatable object files can be linked with shared libraries.
- *Multiple shared libraries.* A process can use several shared libraries simultaneously.
- *No special permissions.* Anyone may build and use a shared library, without needing privileged commands.
- *Limited library source code changes.* Although one can write library code that works unchanged in either shared or unshared libraries, some minor source changes may be necessary to transform existing unshared library code into shared library code. We were willing to impose minor changes on library developers to maintain compatibility for applications.

1. UNIX is a trademark of AT&T.

Unfortunately, data have no branch table equivalent. C compilers typically generate code to reference external and static data symbols directly, without implicit indirections. When an executable file references a data symbol, the machine instructions normally hold the object's address. Furthermore, when the compiler is generating code for an executable file, it cannot know where data objects will reside at run time. Some may reside in the executable file, and others may reside in shared libraries; but the final resolution is unknown until the link editor builds the executable file.

Although we could have extended the language to control code generation, we wanted strict source code and object code compatibility for application programs. Thus we decided to permit shared library data in spite of the complications. A few data structuring techniques help library developers keep data addresses from changing between versions, but data changes have more restrictions than text. Practically speaking, these data restrictions affect library developers but not application developers. Because of the potential for change, we strongly encourage library developers to limit or avoid external data symbols.

3. HOST LIBRARIES

Application programmers use host shared libraries just as they do other development libraries. For example,

```
cc program.c -o program -lX
```

builds the output `program` using library `X`. Application programmers do not need to know whether `X` is shared or unshared.

Briefly, a development library is a collection of functions and data that developers can use. A special file, called an *archive*, lets the link editor select the functions and data an executable file needs. The link editor does not copy unreferenced archive members to executable files, keeping executable files smaller than they would otherwise be.

3.1 Shared Member Contents

Regular archive members hold the text and data for the corresponding C code. Each member typically corresponds to one C source file, compiled to relocatable object code. When the link editor extracts a relocatable archive member, it copies the text and data to the output file. This works fine for a single executable file, but one of our main goals for shared libraries was to eliminate the executable files' copies of library routines.

One builds target shared libraries from relocatable object files too. After building the target shared library, all visible library symbols have fixed, known addresses. When one builds an application with a shared library, only the addresses need to be copied to the executable file. We therefore remove the files' text and data before archiving them in the host.

Although a host library's archive has members that correspond to the target's relocatable files, the archive members are "empty." We create the archive members by looking up the target's symbol addresses, creating absolute definitions in the members' symbol tables, and deleting the members' text and data. When the link editor extracts a host member, it sees absolute symbol definitions to resolve references, but the member contributes neither text nor data to the executable file being built.

3.2 Mixing Shared and Unshared Members

Because the host shared library is a standard archive and the host library members are relocatable (albeit empty) object files, one can mix shared and unshared members. We used this technique when we built a shared version of the standard C library. Although we didn't want to include all the regular C library in the target shared C library, we did want the shared and unshared libraries to be compatible. Consequently, the final host shared C library has shared members for the routines in the target and unshared members for the routines not in the target.

3.3 Building Executable Files

Executable files that need shared libraries hold a special section, as I described in §2.2. This section tells the operating system what libraries to attach for the process. Each host shared library contains a special member that defines its own information for the library section. When a program references any symbol from the host archive, the link editor loads the special member into the executable file.

If an executable file directly uses several shared libraries, it clearly will hold all their special information. Having shared libraries reference other shared libraries might seem to complicate things. We resolved the issue by forcing all library requests to reside in the executable file. For example, if a program uses library A, which in turn uses B, then the program is forced to depend on both A and B.

Two important properties arise from this choice. First, it limits the stack growth in the kernel. When the process loader looks at an executable file, it immediately knows how many libraries the file needs. It does not have to trace library chains of A needing B needing C needing . . . . Second, the behavior preserves the way we currently handle regular libraries that need other libraries. Programmers do not have to learn a new set of rules for dealing with shared libraries, and we did not have to change the link editor algorithms for resolving symbols.

4. IMPORTED SYMBOLS

Shared libraries normally are reasonably self-contained. As an absolute executable file, a target shared library can have no unresolved references. Nonetheless, a shared library can *import* symbols, thus giving it a way to reference functions and data outside its immediate control. Import capabilities markedly improve shared libraries' versatility. First, library code can use application services, relying on the user to supply customized facilities such as error handlers. Second, a library can use imported symbols to let an application's routines replace its own versions. Third, imported symbols give library developers the freedom to choose shared and unshared routines, without worrying about secondary dependencies. Selected routines can be excluded from a shared library, even though they are needed by other library routines.

4.1 Imported Symbol Pointers

Shared library developers can reach outside the library by defining imported symbol pointers. For example, the following code uses both an integer object and a function indirectly:

file.c

```
extern int *datum_ptr;
extern void (*print_ptr)( );
...
value = *datum_ptr;
(*print_ptr)(value);
...
```

Figure 2. Library code with indirections

Instead of referencing the imported symbols `print` and `datum` directly, the library code uses explicit indirections through `print_ptr` and `datum_ptr` to reach the desired text or data.

Depending on the number of symbols a library imports, the library code might have to change significantly. The C preprocessor and a convenient property of C allow one to hide the source code changes:

import.h

```
#define datum (*datum_ptr)
#define print (*print_ptr)
```

New *file.c*

```
#include "import.h"

extern int datum;
extern void print( );
...
value = datum;
print(value);
...
```

pointer.c

```
int (*datum_ptr) = 0;
void (*print_ptr)( ) = 0;
```

Figure 3. Library source code

By defining macros for `datum` and `print` in the *import.h* header file, *file.c* appears to be normal C. The compiler sees indirections, but the programmer does not.

4.2 Initialization Code

Finally, the pointers for imported symbols must be set to their proper values at run time. Libraries define their own pointers, those pointers start with a null value, and each process supplies its own initializations. Library data are private for each process; so one process's assignments cannot interfere with another's.

When the utility program builds a shared library, it knows the inter-file dependencies. Thus if an executable file needs *file.c* above, it also will need *pointer.c*. We therefore arrange for the host member for *pointer.c* to hold initialization code:

pointer.o

```
extern int datum, *datum_ptr;
extern void print( ), (*print_ptr)( );

datum_ptr = &datum;
print_ptr = &print;
```

Figure 4. Imported symbol initializations

Strictly speaking, I lied in §3.1. Some host archive members are not empty, because they hold initialization code that goes *into the executable files*. When the process starts running, it executes the initializations, thus setting all the required imported symbol pointers for the shared libraries.

This example shows the convenience of using the archive format for the host shared library. If the executable file references anything in the library that needs `datum_ptr` or `print_ptr`, the executable file will receive the initialization code for *pointer.c*. The initialization code references `datum` and `print`, forcing them to be defined too. The symbols can be avoided just as easily with the same mechanism.

I'll make one more point about imported symbols. The definitions for `datum` and `print` can reside anywhere in the process. They could be in the executable file, in another shared library, or even in the same shared library as `datum_ptr` and `print_ptr`. A library can import its

own symbols, thus supplying standard versions while allowing an application to give its own.

5. BUILDING LIBRARIES

Previous sections described how shared libraries work. This section describes how one builds a library. I mentioned a utility program that builds a shared library. Using this program, called *mkshlib*, is the primary difference between building regular libraries and shared libraries.

5.1 Library Description File

Besides the regular code needed to build a library, a library developer must control several other items: target library path name, library addresses, branch table entries, and initialization code. Developers create a separate file that specifies how to build the library. A sample specification file appears in Figure 5 for the code we've seen so far.

```
1  #target /my/directory/libX_s
2  #address .text 0x80680000
3  #address .data 0x806a0000
4  #branch
5              a           1
6              b           2
7              c           3
8  #objects
9              pointer.o
10             file.o
11 #init pointer.o
12             datum_ptr   datum
13             print_ptr   print
```

Figure 5. Library specification file

Mkshlib reads this information to build the host and target libraries. Lines 1, 2, and 3 describe where the target library should "live." The path name must be `/my/directory/libX_s`; the text and data will reside at virtual addresses `0x80680000` and `0x806a0000`, respectively.

Lines 4 through 7 define the branch table by assigning each library function to a specific branch table entry. This order can—and should—remain constant. One can add new entries at the bottom without disturbing previous assignments. This lets a library developer expand a library without forcing application developers to rebuild old executable files. Given these lines, *mkshlib* can create the appropriate jump instructions for the branch table.

Lines 8, 9, and 10 tell what relocatable object files *mkshlib* should use to build the host and target library. The two object files listed will be linked with the branch table to form the target file, and "empty" versions of them will be collected into the host archive.

Finally, the last three lines tell *mkshlib* what initialization code to create. To the object file `pointer.o`, *mkshlib* will add the assignments shown in Figure 4.

5.2 Creating the Host and Target Libraries

With the object files and the specification file, *mkshlib* creates both the host and target library files automatically. A library developer issues one command that builds both files.

6. A CASE STUDY: THE SHARED C LIBRARY

One can build compatible versions of shared and unshared libraries. Given this possibility, one obvious candidate for conversion was the standard C library. This section describes that work.

6.1 Choosing Contents

Deciding what to include in the shared library proved to be more difficult than one might have imagined. We wanted the shared and unshared libraries to be compatible, but we did not know how much of the original library to share. We eventually based our decisions on performance and size studies.

We built a series of libraries and studied their effects on existing programs. This let us identify routines that "paid their way" and discard the "deadbeats." For example, many existing programs use the Standard I/O package; sharing them helps many programs. Some other infrequently used C library routines define large data buffers, and we decided not to share them. Because every process gets a private copy of its libraries' data regions, we excluded them to keep the memory requirements small.

Our resulting shared C library includes shared copies of the Standard I/O functions and other commonly used routines. Relocatable (unshared) copies of the other routines exist in the host archive—but not in the target library—for compatibility.

6.2 Redefining Library Routines

After deciding what we wanted to include in the shared library, we decided what routines we wanted to let applications replace. Although we could have allowed redefinition of every library function, that didn't seem to be the best choice for the C library. Application writers rarely redefine most C library functions, and we did not want to add unnecessary indirections. Moreover, many redefinitions are unintentional and can cause mysterious program behavior. The draft proposed ANSI standard for C recognizes this fact by reserving library identifiers, making program behavior implementation-defined when redefinitions are present.[5] After considering these issues, we looked at existing libraries and commands, finding what functions had previously been redefined. Fewer than 10 functions qualified.

6.3 Program Size

Existing programs proved to be a valuable test for the shared C library. We could easily measure file and memory size, adjusting the shared library contents as appropriate. We used the final shared C library and the regular C library to build many of the standard programs and achieved the following results:

TABLE 1. Shared C library results, 115 programs[6]

Measurement	*With Regular C Library (KB)*	*With Shared C Library (KB)*	*Savings (KB)*
File text size (avg)	19.5	8.9	10.6
File data size (avg)	5.2	4.4	.8
File bss size (avg)	4.4	4.4	0
Library text size		28.1	
Library data size		1.4	
Library bss size		0	
Memory text size (avg)	19.5	$8.9+x$	$10.6-x$
Memory data size (avg)	9.6	10.2	$-.6$
Disk file size (avg)	24.8	13.6	11.2
Disk file size (total)	2,850.5	1,559.1	1,291.4

5. ANSI, *Draft Proposed American National Standard for Information Systems—Programming Language C*, X3J11/86-017 (February 14, 1986), 73.

6. "Bss" means uninitialized data. Because the data are uninitialized they occupy no space in the executable file, even though they have memory allocated at run time.

When using shared libraries, all attached processes share the cost of library text. Thus the real memory consumption depends on system load, the number of active processes, library size, and so on. If one process is active, it must "pay" for all the memory holding the shared library text region; if 100 processes are active, the cost per process is much lower. (The "$+x$" and "$-x$" in the table account for this.)

6.4 Program Performance

Because the shared and unshared versions of the C library were compatible, we built two versions of performance benchmark programs. Our results showed about equal performance for the two systems.

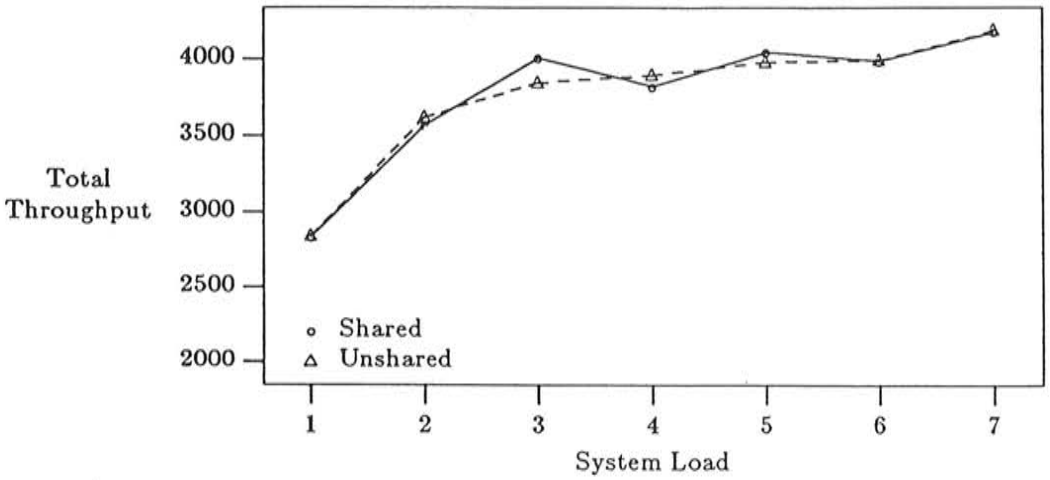

Figure 6. System performance

In the performance graph the vertical axis, "Total Throughput," shows how much work the system accomplishes in a given time period. Higher numbers mean the system did more work, which is good. The horizontal axis, "System Load," shows how much work the benchmarks try to do simultaneously. The higher the number, the heavier the load.

Throughput values for this benchmark frequently vary by 100 points from one run to the next. Although I've plotted averages from several runs and the graphs differ a little, the shared and unshared results are similar. Consequently, one can say the shared C library neither penalizes nor helps system performance for this benchmark, under the work loads tested. Although we believe this benchmark and its work load represent a fair performance test for the shared C library, they do not necessarily apply to other shared libraries or work loads.

Libraries' static and dynamic properties can differ markedly. Some libraries (or parts of libraries) may be needed statically by many applications but executed infrequently. Error routines, for example, might reside in every executable file of a system, but they might never run if the hardware and software are reliable. On the other hand a library might have some code that only a few processes use, but the code might account for much of the total application run time.

With the shared C library, we have seen both properties. Some parts of the library execute frequently, while other routines execute hardly at all. Moreover, the C library frequently shows random execution patterns, because many different kinds of applications use its diverse routines. Libraries dedicated to a single purpose, such as database or graphics services, show much different dynamic behavior than the shared C library.

Paging systems particularly show the value of grouping closely related library functions. A library's text region serves all processes that use the library. In the shared C library case, those processes are largely unrelated, and they touch library pages in unpredictable ways. Analyzing

the paging behavior of a shared library text region can therefore be difficult. The job becomes easier for more controlled environments. If a set of known processes uses a library of coherent functions, the working set can shrink, thus decreasing total paging activity and helping system performance.

7. FUTURE WORK

Some aspects of this effort need further work. First, statically allocating virtual addresses does not work well on machines with restricted virtual address spaces. Second, some applications want to select libraries at run time, instead of link edit time. Third, we want to add more flexibility to target path name specification. And fourth, we'll continue to enhance debugging and make the facilities available for languages other than C.

8. ACKNOWLEDGMENTS

Several people helped design and implement shared libraries. Don Milos worked on the operating system; Deborah Bach and Ken Stein wrote the shared library support tools; Iraida Torres-Irizarry built the shared C library.

9. SUMMARY

Shared libraries often help reduce program size and improve performance, especially for smaller systems. Our main goal for the design was to give applications a way to reduce executable file size. The shared C library shrank our standard UNIX System commands over 11,000 bytes per file on average. Shared libraries can help shrink process size and improve performance too, but these dynamic effects depend on system activity and paging strategies. File size, as a static property, is easier to measure and control. Although the shared C library gives an example of the possibilities, it represents a "worst case" in many ways. Applications with many cooperating processes and heavily used libraries can achieve even better results.