Relatório Técnico

Núcleo de Computação Eletrônica

Shared Libraries in Tropix: A Simple and Efficient Implementation

Pedro Salenbauch

NCE - 21/2000

Universidade Federal do Rio de Janeiro

SHARED LIBRARIES IN TROPIX: A SIMPLE AND EFFICIENT IMPLEMENTATION

Pedro Salenbauch
pedrosal@nce.ufrj.br

NCE/UFRJ
Núcleo de Computação Eletrônica da
Universidade Federal do Rio de Janeiro

Abstract

During many years, the TROPIX Operating System had no shared library facility. As the time was mature to develop this feature, several other Operating Systems were studied, and we readily verified that the various used methods worked correctly, but were excessively complex. We wondered, then, if it wasn't possible to develop a simpler technique, that also provided the good sides of all theses methods, with minimal disadvantages.

Now we know that this is possible. In this paper we present a shared library method that reduces the disk storage and main memory spaces, reduces the load/execution time for a program, and the program runs at full speed. It also doesn't need special compilers/assemblers, doesn't use linkage tables nor needs dynamic link-editions.

The only caveat is that if, during an upgrade of the library, one or more functions of the shared library grow over a certain (tunable) threshold, the executable files that reference the shared library must be relink-edited.

I. The TROPIX Operating System

The TROPIX Operating System was born in 1982, as several members of the NCE/UFRJ returned from their Ph.D. studies in Universities in the USA and England. They used the UNIX * Operating System in their researches, learned to like it, and on their return home, were surprised with the absence of it at the Brasilian Universities.

At that time, UNIX was distributed for free by AT&T, including the source code, for American, European and Australian Universities. During one whole year, the NCE tried to get a licence of UNIX from AT&T, but this was stubbornly refused based on the argument that Brasil (and most the other countries of Latin-America) are "software pirates", and thus not trustworthy.

* UNIX is a trademark of Unix System Laboratories

After this bad experience, the NCE decided to develop its own UNIX-like Operating System, with a team leaded by Newton Faller. The necessary hardware, a "supermicrocomputer" with 2 MOTOROLA 68020 processors was also built, and in 1985, TROPIX (at that time called PLURIX [1]), began to run, already multiprocessor.

We will skip addicional details of the TROPIX biography (which will be subject for a future paper), to give the present status of the Operation System. It has already been ported to the PC (Intel Pentium platform), and besides the basic UNIX commands, it has the Internet TCP/IP interface with several client/servers and the X-Window Graphical System, with a small set of clients.

II. Shared Libraries

As we know, shared libraries offer 4 advantages over regular (non-shared) libraries. They are:

1. Save disk storage space, because the executable files are smaller (they don't include the code of the functions of the library).
2. Save main memory space, by the same reason.
3. Save program execution setup time, since smaller executable files can be read faster from the disk. However, if dynamic link-edition is used, the savings are not so significant (see below).
4. Make the maintenance of the executable files easier, because if some functions of the shared library are updated, there is no need to recompile and/or relink-edit all the executable files. Only the shared library has to be replaced (it is not always so simple, see below).

On the other side, shared libraries can also introduce some caveats. Depending on the implemention method, there may occur the following problems:

1. The need of a different set of <include> files: one set to use with the shared library, and another set with the regular library.
2. Necessity of a special compiler and/or assembler. In certain methods, the compiler must know if the function being compiled is for a shared or regular library and generate a PIC (position independent code) for the shared library.
3. The execution of a PIC (see above) is slower than regular code.
4. The necessity of a linkage table for the external references of the program, which also makes the execution slower.
5. Time wasted with a dynamic link-edition each time the program is to be executed. In this case, the link-editor has to invoked at "exec" time.
6. Time wasted with the traps that occur on the first reference to an unresolved symbol. In this case, the kernel has to complete the link-edition at the first reference to each unresolved symbol, during execution.

The basic methods for implementing shared libraries are:

1. Static link-edition with fixed addresses: during link-edition, all the addresses of the entry points and global variables of the shared libraries are fixed and known in advance through a symbol table.
2. Static link-edition with a linkage-table: during link-edition, all the calls to the entry points of the functions and the reference to global variables are done indirectly,

through a linkage-table. This technique is used by the UNIX System V Release 3 [2].

3. Dynamic link-edition at "exec" time: As the executable file is being prepared to be executed, there is made a dynamic link-edition with the shared library. This method is used by SunOS [3], and also by FreeBSD and LINUX.
4. Dynamic link-edition during the execution: All the references to the shared library receive an invalid address to force a "trap" during the first occurence, which will cause the dynamic link-edition of each symbol.

Now, we analyse of the advantages/disadvantages of each method:

1. Static link-edition with fixed addresses:

   Advantages: we can use the regular <include> files, the regular compiler/assembler, no time wasted with dynamic link-edition nor "traps".

   Disadvantages: any change in any of the functions of the shared library will force the recompilation and/or relink-edition of all the executable files.
2. Static link-edition with a linkage-table:

   Advantages: we can use the the regular compiler/assembler; no time wasted with dynamic link-edition nor "traps"; no need to recompile and/or relink-edit the executable files if there are made changes in functions of the shared library (the linkage-table will be modified accordingly).

   Disadvantages: we need special <include> files; the execution is slower because of the indirect references.
3. Dynamic link-edition at "exec" time:

   Advantages: no need to recompile and/or relink-edit the executable files if there are made changes in functions of the shared library.

   Disadvantages: the need of the dynamic link-edition at the beginning of the execution, though special link-edition functions embedded in the kernel of the operating system or spawning a new process. This will take a considerable time whenever the program is to be executed, and this inspired devising a kind of "cache" (see reference [4]).
4. Dynamic link-edition during the execution:

   Advantages: no need to recompile and/or relink-edit the executable files if there are made changes in functions of the shared library.

   Disadvantages: needs special services of the trap functions of the kernel which must use special link-edition functions embedded in the kernel. This will take a considerable time for the link-edition of all the unresolved symbols at their first reference.

III. An efficient method

After the many methods presented above, each with its own advantages/caveats, we come to a crucial question: what would be an ideal method for implementing shared libraries?

The question is not difficult to answer; it would be a method that:

1. Uses the regular set of <include> files, compiler and assembler, that are also used with the regular libraries.
2. Doesn't use linkage-tables, which causes overheads.
3. Doesn't need dynamic link-edition at run time, neither at the beginning of the execution, nor with the "traps" during execution.

4. The functions of the shared library can be changed without forcing a recompilation and/or relink-edition of the executable files.

Is the "ideal method" feasible? We present below the TROPIX method, which is "almost ideal":

This method uses a static link-edition of the shared library with fixed addresses, but reserves spaces between each of the sections (TEXT, initialized data, simply called DATA and uninitialized data, which we will call BSS) of each consecutive pair of the functions of the shared library. Theses spaces are reserved for future growths of the functions.

A space is also reserved between the end of the DATA of the library and the beginning of the BSS (the space after the end of the TEXT is normally implicit, since the DATA/BSS regions use other virtual addresses). This space is reserved for future addition of new functions in the shared library.

All these reserved spaces permit (in most of the cases) future changes in the functions of the shared library without having to recompile and/or relink-edit all the executable files.

A diagram of the structure of a shared library is given on Table 1.

IV. The building of the Shared Library

The building of the shared library ("mkshlib" utility) goes as follows:

1. Each shared library receives a unique number (beginning with 0), called the "index". For each index are assigned fixed virtual addresses for TEXT and DATA/BSS, which are alocated on others regions than that of the regular regions of the executable files.
2. The TEXTs of the functions are processed: they are relocated to their final addresses, and all external references must be known at the end of the building of this shared library. After each TEXT is left a reserved space in the virtual addresses. At present our "default" value is set at 16 % of the size of the corresponding section of the function (it can be changed through an option of "mkshlib").
3. The DATAs of the functions are processed: they are relocated to their final addresses, and all external references must be known at the end of the building of this shared library. After each DATA is left a reserved space in the virtual addresses. Notice that even in the case that a function has no DATA, it is important to reserve a space, since in a future modification, it may have one.
4. The BSSs of the functions are processed; after each BSS is left a reserved space in the virtual addresses. Notice that even in the case that a function has no BSS, it is important to reserve a space, since in a future modification, it may have one.
5. The shared library is written on disk, with the following sections: header, TEXT, DATA, symbol table and module table.
   a. The header has the information on the sizes of each section, the index number of this library and its version number (among other informations).
   b. The TEXT and DATA sections have the final relocated code for theses sections, including the "reserved" space, which is coded as zeros.
   c. The symbol table contains the final relocated addresses for all symbols of this shared library, which will be used directly by the link-editor.
   d. The module table contains the name of each function file that was used during the construction of the library, with the respective sizes allocated for each section. Also included is the percentage used for the reserved spaces. This table is a necessary information for the creation of a compatible new version of the shared library, and is used only by "mkshlib".

TEXT of function "a"
Reserved for the growth of TEXT of function "a"
TEXT of function "b"
Reserved for the growth of TEXT of function "b"
..... (TEXTs of other functions) .....

.....

(Gap in the virtual address)

.....

DATA of function "a"
Reserved for the growth of DATA of function "a"
DATA of function "b"
Reserved for the growth of DATA of function "b"
..... (DATAs of other functions) .....
Reserved for the DATAs of new functions
BSS of function "a"
Reserved for the growth of BSS of function "a"
BSS of function "b"
Reserved for the growth of BSS of function "b"
..... (BSSs of other functions) .....

Table 1

The vertical axis depicts virtual addresses, growing downwards.

The re-building of the shared library (modification) is similar to the original building. Instead of using a percentage for the reserved spaces between the functions, "mkshlib" will try to use the original sizes, after checking if the new sizes aren't bigger than the original sizes allocated.

If the reserved spaces are sufficient to accomodate all the size increases of the old functions, and the space reserved between the end of the DATA and the beginning of the BSS is sufficient for the DATA of the new functions, the new version of the library is accomplished with success, and it can replace the old version, without the recompilation and/or link-edition of the executable files.

Obviously, size decreases (which may also occur) present no difficulties.

If only one of the reserved spaces isn't sufficient, the new version will not be compatible with the old (i.e. will have other virtual addresses), and in this case (unfortunately) all the executable files have to be compiled/link-edited again.

Of course, the bigger the reserved space percentage is, the lower will be the probability that this is the case. At present, we are working with the value of 16 %, but it can be tuned in the light of the necessity for future modifications of the functions. As the functions of the library attain their final versions, the library becomes more and more stable and the percentage may be dropped.

V. Examples

The TROPIX Operation System, at present, has 3 shared libraries called "libt.o", "libx.o" and "liby.o":

1. The "libt.o" encompasses the "libc" (regular "C" language library), the "libm" (mathematical functions), "libcurses" (terminal emulation functions) and "libxti" (networking functions). All executable files link to this library (including "init", "login" and "sh").
2. The "libx.o" and "liby.o" include all the graphic X-Window functions. These functions have been distributed into 2 separate libraries, where the most used are in "libx.o", so that almost all of the X-Window clients need to link only to one shared library (besides "libt.o").

The "libt.o" (including over 400 functions) has 104 KB of TEXT, and 936 bytes of DATA (including the 16 % space reserve) It is interesting to notice that the DATA needs less than 4 BK, which is the page size of the Pentium memory management unit.

As a comparison, "libx.o" has 435 KB of TEXT and 13172 bytes (4 pages) of DATA, "liby.o" has 566 KB of TEXT and 30476 bytes (8 pages) of DATA (all with the 16 %).

The image of the a shared library in memory includes not only its TEXT (which is shared by all the programs), but also an original version of its DATA, which needs to be copied at the beginning of the execution of a program.

Maintaining the DATA at a minimum is important to avoid execution overheads. Many functions that had static allocated areas were modified to use dynamic allocated areas through a call to "malloc".

The size of the BSS is not so critical, because it doesn't have to be copied from anywhere.

VI. The link-edition with Shared Libraries

The link-edition with a shared library is simple: its symbol table can be used directly (the addresses have already their final values); no code has to be inserted in the executable file. Only a flag has to be set in a bitmask in its header, to indicate that later, at execution time, the library has to be already resident in memory.

As we see, the link-edition with a shared library is easier and faster than with a regular library.

VII. Kernel support for the Shared Libraries

Some (small) additions to the TROPIX kernel were necessary to support shared libraries. They are:

1. A new system call: "shlib" was introduced. It loads/unloads the shared libraries from main memory. Normally it is called by the "ldshlib" utility, which manages the resident shared libraries, but during the initialization of the kernel, it is automaticatelly called to load "libt.o", which is used by all programs, including "init", "login" and "sh".
2. The "exec" system call had to be extended: if the incoming program uses shared libraries (what is indicated in the bit mask of the header of the executable file), additional regions must be attached to the process, containing the TEXT and DATA+BSS sections of the shared libraries.

   First, the kernel checks if all needed shared libraries are already loaded in memory (if not, the execution will be aborted). After this, the regions are created and private copies of the DATA sections are made, since the read-only TEXTs can be shared. If the set of shared libraries used by the new program happens to be different from the set used by the program being replaced, the "exec" system call must also release the regions that will no longer be necessary.
3. The "fork" and "exit" system calls were also extended: it is necessary to duplicate ("fork") and release ("exit") the regions corresponding to shared libraries.

VIII. Version Validation

As an executable file is to be executed, there must be a check to ensure that the file and the shared libraries it references are compatible. This is done as follows:

1. When the shared library is created, it receives the version number 1.
2. If the shared library is modified and the new release is compatible with the previous one, its version number is kept.
3. If the new release is NOT compatible, it receives a new version number, which is the maximum of the version numbers of all the shared libraries, plus one. This guarantees that all current executable files that reference this shared library (and only these) are out of date, cannot be executed and must therefore be relink-edited.
4. During link-edition, the executable file receives a version number that is the maximum of the version numbers of the referenced shared libraries.
5. At execution time, the kernel certifies that the version number of the executable file is equal or greater than all those of the referenced shared libraries.

IX. Creating a new executable file set

If the generation of a new release of "libx.o" and/or "liby.o" (the X-Window shared libraries) produces an incompatible version, this is not so critical, because we can link-edit the new versions of the X-Window executable files using the text mode.

This is not the case with "libt.o", because all executable files reference this shared library (including "init", "login" and "sh"). If the "libt.o" is changed, these utilities won't work any more, and we can't "boot" the operating system.

How can we circumvent this problem? There are at least 2 "safe" solutions: with the first, we build the new executable file set during an intermediate phase, in which we use the critical utilities ("init", "login", "sh", compilers, assembler, link-editor, ...) in a version without shared libraries.

The second, which we prefer and use, is to have an alternative partition containing also a root file system, in which we build the new versions of the shared libraries and executable files. We can test this new alternative partition, and if it doesn't work, we can always "boot" again the old (safe) partition.

X. Semantic side effects

The TROPIX method for shared libraries has a subtle semantic side effect: as the external references of the shared libraries are computed during the library creation, they receive a fixed value, known at the creation time. This value is "frozen" and not recomputed later.

As a concrete example, "libt.o" includes the "stdio" package and the standard memory allocation function "malloc". If a user creates his own version of "malloc", the functions of his program will call his own "malloc", but the "stdio" will continue to call the standard "malloc".

However, this is not serious problem, because the standard "malloc" can coexist with others "malloc"s (and similarly for the others functions of "libt.o). If (in a rare case) a user wants explicitly that the "stdio" uses his "malloc", he can always link-edit his program with the regular (not shared) "libc".

This side effect has its beneficial sides, too. Many times, a novice in the "C" language includes a "write" function in his program, which normally will produce impredictable fatal effects when "printf" calls his "write" instead of the standard "write" system call. With the TROPIX method for shared libraries this will not occur.

XI. Conclusions

We presented a shared library method that has all the advantages of both worlds, and almost no disadvantages. The disk storage space and main memory space are reduced, the load/execution time for a program is reduced, and the program runs at full speed.

The only caveat is that increases of the sizes of the functions of the shared library above a certain (tunable) limit will force the recompilation and/or relink-edition of the executable files.

As examples, we can present 3 classes of programs:

1. A simple program that consists almost only of library function calls: "pwd". Its size was reduced from 10488 to 440 bytes.
2. A bigger program that has a significant size besides the called functions: "vi". Its size was reduced from 71 KB to 38 KB.
3. A program that uses the X-Window libraries: "xedit". Its size was reduced from 599 KB to 3 KB bytes.

From this last example we can deduce that the use of shared libraries with programs using a graphical interface is a "must".

XII. Future Work

We have made some measurements of the size reduction of the method, but not of the speed. We plan to propose a M.Sc. thesis at the NCE/UFRJ to make several comparisons analysis between the following operations with shared and regular libraries:

1. The speed comparison analysis between the link-edition with a shared library and the corresponding regular library.
2. Comparison of the setup time for the execution of a program link-edited with a shared library and the corresponding regular library.
3. Implement a dynamic link-editor and make a global execution time speed comparison between this method and the proposed one.

Bibliographic References

[1] Newton Faller & Pedro Salenbauch: "PLURIX: A Multiprocessing UNIX-like Operating System", *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, IEEE Computer Society Press. Washington, DC, EUA 29-36, Pacific Grove, California, Set 1989.

[2] AT&T UNIX System V Release 3 Programers Guide, Chapter 8 (1986).

[3] Gingell R.A. et alli, "Shared Libraries in SunOS", Sun Microsystems, Inc.. *Proceedings of the USENIX 1987 Summer Conference*, pages 131-145. USENIX Association, June 1987.

[4] Orr, D.B. et alli, "Fast and Flexible Shared Libraries", University of Utah. *Proceedings of the USENIX 1993 Summer Conference*, pages 237-251. USENIX Association, June 1993.