Improved Portability of Shared Libraries

James Donald
Princeton University
Princeton, NJ 08544
jdonald@cs.princeton.edu

January 25, 2003

Abstract

Dynamic linking provides developers and users with increased power and flexibility. However, this also brings with it the power to create unexpected problems and incompatibilities. Although some upgrades or installations involving shared libraries can successfully match up on all original interfaces and continue proper functioning behavior, this is not always this case.

Surprisingly, it is arguable that the shared library problem under Linux is perhaps even worse than the corresponding problems in Microsoft Windows [1]. The proposed protective layer given in this paper seeks to bring better library compatibility to Linux and other flavors of UNIX.

The basic system comprises of (i) a daemon that keeps track of many current and old libraries along with (ii) enhancements to the user's shell or desktop window system that detect a library incompatibility and try to resolve the problem seamlessly.

1. Introduction

The use of dynamic linking to shared libraries--also known as Dynamic Link Libraries (DLLs) on Microsoft Windows--provides many benefits. For example, dynamic linking can shrink application binaries, save disk space, save memory, and provide an elegant path for software upgrades [17].

However, the added flexibility of dynamic linking may damage the robustness and create compatibility problems. Some components, such as DirectX, have been known to break many existing applications with some of their upgraded versions [1]. Rick Anderson coined the term "DLL Hell" to refer collectively to a whole range of compatibility problems caused by DLLs [2]. Although in an ideal world, all original interfaces would match up and implementations could be changed seamlessly, this is not always the case. See Table 1 for a list of some common trouble-makers.

Culprit	Platform	Reason for portability problems
DirectX	Windows	involves hardware-specific code [1] [3]
OpenGL	all	involves hardware-specific code [5]
MFC42.DLL	Windows	used so much, yet was constantly evolving [2] [11]
libc (glibc)	UNIX	used so much, plus very large and complex [1] [18]

Table 1: Some shared libraries for which occasionally something goes wrong.

Even if a new library properly adheres to its original spec and is practically bug-free, it can still be incompatible with a legacy program. This often happens because existing bugs in the original libraries whereby the legacy applications relied on improper side effects. However, once these original side effects are removed to conform to the actual spec, the legacy applications no longer function. Anderson referred to this problem as Type II DLL Hell. Type III is the problem mentioned earlier when a new library actually does have bugs. Type I, the most common on Windows, happens when a rogue program has bugs in its installation so it installs libraries in the incorrect places or overwrites newer libraries with older ones [2]. Furthermore, for example, if program A requires an old version, while program B requires a newer version, it becomes very difficult for the programs to coexist. Some applications use private older copies of shared system libraries to get around this problem, and this works alright under Windows, but can often lead to headaches when attempted under Linux [18].

This project aims to prevent and mitigate the disasters caused by all such shared library problems.

2. Motivation

Microsoft's Windows Application Compatibility Group is dedicated to backwards compatibility of various legacy programs. The primary method by which they create backwards compatibility is "injecting" protective DLL's (at runtime) into old programs that were not previously equipped for dealing with upgraded libraries. The effort is arguably quite a success, as Windows has

maintained arguably much better continuous binary-compatibility for applications than Linux.

One example of how Microsoft shows its confidence in its ability to provide backwards compatibility is shown with DirectX. Despite its problems in the past, no uninstall feature is provided with the most recent versions of DirectX, and a claim is made that it is never necessary to uninstall because there has been sufficient testing for full backwards compatibility [3].

However, for the Linux world, as with many flavors of UNIX, applications have in the past been mostly open-source. When library incompatibilities arose, it was sometimes just expected of the user to download the source and recompile the programs. As the popularity of Linux increases, however, more closed source applications may appear, and so the recompilation option may become less viable [1]. Alternatively, the user can download different versions of binaries made for different library sets, but this is still less convenient than binary compatibility across library versions.

.

The Linux community does not have the equivalent of the Windows Application Compatibility Group, and so it is worrisome that the same laissez-faire attitude about library compatibility may continue indefinitely at Linux users' expense.

And so, the project presented in this paper targets Linux specifically in hopes of finding a solution that is efficient and inexpensive.

3. Sample System

Because the upgrading of libraries is such a long-term and user-specific effect, it is difficult to measure the cumulative damage or how often certain users are affected. For starters though, it can help to sample how often a typical system has its libraries updated. Figure 2 shows a histogram of the times since modification for a sample of 1000 shared libraries (*.so files) taken from the main /usr/lib directory on Princeton's computer science Linux servers.

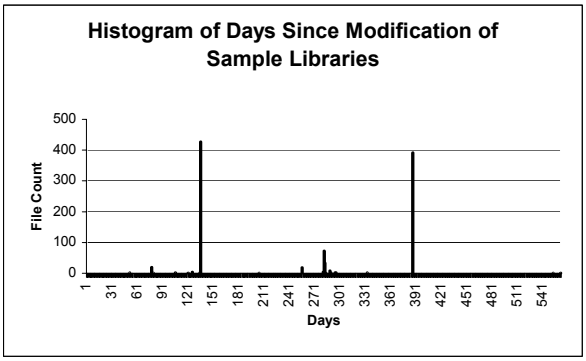

Figure 1: Histogram of days-since-modification on a sample of 1000 shared library (*.so) files in /usr/lib on penguins.cs.princeton.edu.

For this single case, it appears that some major library system changes are done approximately every six months. Other servers or personal computers may vary greatly though, depending on the decisions of their system administrators.

Also, it is uncertain whether library upgrades done more often can lead to increased or reduced problems. For library modifications done very often, there is a larger quantity of opportunities for incompatibilities to sneak in. On the other hand, if the modifications are done less often, they probably involve more drastic--so perhaps more dangerous--upgrades each time.

4. Design

The library portability improvement system consists of two main components: the daemon responsible for backing up libraries and the shell component responsible for detecting a library compatibility malfunction.

4.1 Library Protection Daemon

The job of the library protection daemon is to maintain compressed backup copies--along with checksums--of the libraries in case they get modified later. Certainly, it was desired to have a more elegant solution than one that involves backing up every library. Unfortunately though, there does not seem to be an easy way around this because since libraries are so extremely intricate (especially in their *bugs*) that any loss of any library would be a loss of needed code. Now, the concept of simply backing up files before catastrophically modifying your system isn't very groundbreaking to the average PC user. However, the purpose of this portability protection scheme is to do it in such a way that doesn't resemble the typical crash, burn, and start-over cycle.

Although shared libraries can contain version information, as is standard in Windows DLL's, another safe way we are tagging these files is to record the sizes and appropriate checksums based on the libraries' binary contents. These checksums can be compared fairly quickly later on to identify whether each system library still matches to certain backup copies.

Upon storage by the library protection daemon, the libraries are compressed. We could use whatever various compression algorithm, but for now gzip is chosen. It seems that Linux libraries compress about threefold under gzip, so the space required by this daemon is approximately 35% times the space used normally by system libraries. This is very little compared to the amount of space that would be used by statically linking all applications, which is proportional to the system library space multiplied by the number of applications. A backed up library will need to be decompressed only when it is found to be probably necessary (as confirmed by various tests with the checksums) for our main purposes. And so this is probably a rare enough event that the space saved by compression outweighs the decompression overhead cost when the time comes. Figure 2 below summarizes the basics of the process.

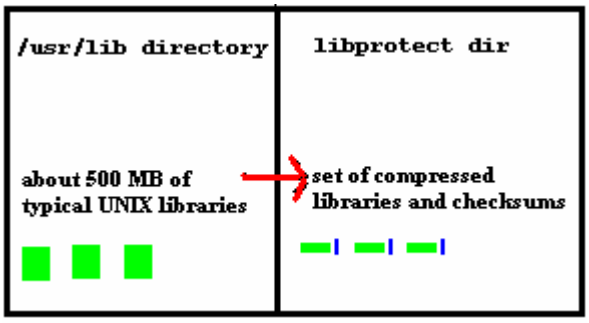

Figure 2: What the library protection daemon starts off doing.

Shown below are sample contents of a libprotect directory.

```
tux% ls libprotect
82687       kcalc.so.gz
128         kcalc.so.gz.checksum
11111       kchart.so.gz
128         kchart.so.gz.checksum
20390       kcminit.so.gz
128         kcminit.so.gz.checksum
```

When it comes time to make use of a backed up library, perhaps because an incompatibility with a new library was detected, the backup copy will be decompressed then ran against. If it turns out that backup copy will be needed in use for a while, it may remain uncompressed so it can be commonly used.

Later on it may be found that multiple versions of a library will have to be stored (perhaps because of repeated, incompatible upgrades by the administrator). For this case, it might be possible that if similar library versions do not differ too much in their binary contents, the binary comparisons (bdiff) can be used to save space. However, the amount of used disk space may simply be insignificant for these cases.

4.2 Incompatibility Detection

The second main necessary component is the mechanism for detecting library compatibility problems. The detector is to be placed in the shell, perhaps coded into the shell or as a plug-in. Since the shell is strongly tied to paths and user properties (for example, the default shared library paths), it is in many ways a good choice of point to detect and attempt to fix dynamic linking problems. In place of the shell we may also use the user's desktop window manager, but regardless, the detection point should be a user-based (not kernel-based) program that spawns most of the user's utilities directly.

Now, a surprising claim here will drastically simplify the detection process. It is the author's experience that most dynamic linking problems manifest themselves in obvious ways. For example, you may try to start up a program and simply see the message "The ordinal __xx_func could not be found in libyy.so". Assuming that most (but definitely not all) dynamic linking problems manifest themselves in such an obvious way, the detection for most cases becomes very simple. The shell can simply watch the text output of the user's directly-spawned programs and look out for any giveaway error code or giveaway error message.

In some slightly more complicated cases, this doesn't happen right at the start of execution, but rather halfway through the program. However, the error message might still be clear, and although it won't be the most user-friendly repair, at least the problem will be detected properly.

Lastly, an even smaller portion of programs may crash much more obscurely (segmentation fault) when due to a library problem, but if program crashes are briefly checked (upon spotting an unusual error code), then library comparisons can still be done. For example, if there's an amazing coincidence such as records of no crashes prior but all crashes immediately after a library modification, then here is another clue that can be acted on.

Upon detecting the problem, the next step is to analyze the executable to see which shared libraries it makes use of. Plain linked programs should contain information in their binary format that can be read from the linkage header to quickly identify the shared libraries that the application makes use of. This is assuming the user has read access to the executable, but it is usually the case that if the user has execute-access to the file he probably has read-access.

From here, it goes on to communication with the library protection daemon to pull out libraries and see if this can be immediately explained by a change in a certain library. If so, the proper information is recorded, and the next user-friendly step is to automatically immediately try and run the program with some old set of libraries that are likely to work. If all goes well and the original error message is hidden, then this can be done in a user-friendly way such that the user doesn't notice anything went wrong in the first place.

The overall step-by-step process is diagrammed in Figure 3.

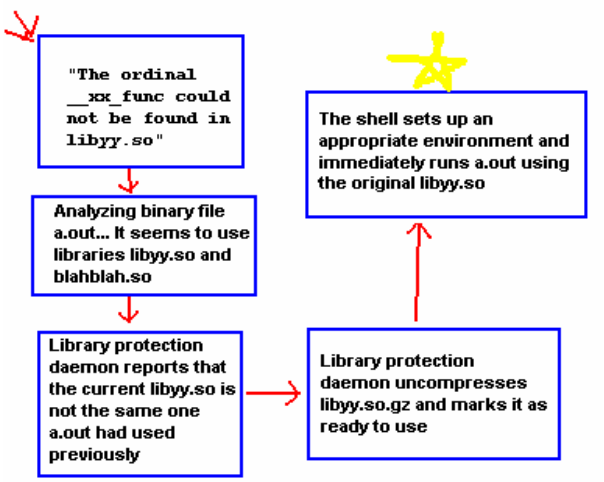

Figure 3: Step-by-step example of the incompatibility detection and resolution.

The best time to install this foundation would be immediately preceding a significant library upgrade. For example, this would be right before you are upgrading from libc 5 to glibc 2.0.

5. Current Status

At the present time, the fundamentals of the model have not yet been written or tested. It would take a fair amount of work to get this design up and running to confirm many of the conjectures put forth in this paper.

6. Other Issues

In the design of the incompatibility detector, it was assumed that Linux binaries can easily be read to determine which shared libraries they directly depend on. If the applications are linked to shared libraries in the standard simple way, this is probably valid. However, some Linux programs open up shared libraries with explicit calls to dynamic linking functions like dlsym() and dlopen(). In this case, it may be much more challenging to detect the library dependency, but it is uncertain whether this will actually constitute a significant percentage of cases. Furthermore, such programs often supply their own routines, and sometimes elegant error recovery, for dealing with failed libraries.

Library dependencies can, and very often do, occur as a complicated graph. For example, library A depends on B, while B depends on C, while C depends on a long chain eventually pointing back to A. Although a straightforward method to scan for the complete required set can be achieved, this might add significant overhead to detection of library modifications.

It was decided that the detection scheme would be placed within the shell, desktop manager, or other user-based program that saw over a significant number of program executions. This was chosen instead of a kernel-based approach because it might be simpler and more secure. However, this would not have omnipotent watch over all spawned programs. If an overseen program spawned another program for private use, that child would not be overseen. Furthermore, a shared library failure message from the child process may be misinterpreted as a shared library problem for the parent process. And so if a shell-based watcher fails to much for this reason, then perhaps a kernel-based approach ought to be attempted.

Library upgrades are often done for reasons related to security. For instance, many old libraries contain buffer overruns that compromise the system. This model presented here does not account for this and so it may be a security hazard.

Shared libraries typically come in sets of well-organized installations [1], and unfortunately the hack-restore job of a few libraries as would be done by this system may not be sufficient in many cases. It remains to be determined what kinds of surprising problems may arise from the intricacy of libraries being designed only for their standard sets.

7. Alternatives

One of the most obvious alternatives to dynamic linking is static linking. Even though, compared to in the past, static linking is used much less proportion-wise nowadays than dynamic linking, it is still in common usage even for standardized system libraries. For example, Microsoft Visual Studio still provides programmers the option to link MFC42.DLL statically, because developers often still desire to do this for obvious reasons. Perhaps even there are more creative options to use besides simply giving choice to the developers. For example, automation can be brought into a program's decision to use static linking on less stable systems but dynamic linking on the more stable ones.

It is thought to be possible for clever users to configure their Debian Linux environment in order to use multiple versions of the libc library simultaneously [18]. However, this is difficult, is probably cannot be done with most libraries, is not yet simple to do, nor is it guaranteed to provide good results.

Another possible alternative is simply hoping that program writers and library writers can write more compatible interfaces to better withstand upgrades and changes. Although this is wishful thinking, it is part of a holy grail that is always sought, and even if never fully achieved, progress toward it may continue.

Sergey Ayukov suggests other desperate options in addition to using statically linked executables. These ideas include switching to finer library version numbering, expanding the Linux kernel definition to libraries, and creating an oversight committee for Linux. Unfortunately, these solutions may not be politically feasible [1].

As a solution to some common library problems, Windows 2000 (and as is inherited by Windows XP) implemented the feature of Windows File Protection (WFP) for DLL's. This managed to solve all of the "Type I" problems as described by Anderson. It protects system libraries directly to avoid rogue programs trying to modify them. Only signed Microsoft updates can perform the upgrades to these files [2]. Although this could theoretically applied to other operating systems, for Linux it would require some more "standardization" on the common libraries, which does not seem very possible given the nature and visions of Linux at this time.

8. Possible Enhancements

Within the time constraints, no actual library-application-breaking testing was done to confirm the claims of occasional incompatibility. This sort of breaking is a very long-term and user-specific problem. However, controlled experiments can be done to locate points in version history of libraries when such breaks have occurred. For example, Microsoft has made good use of its "send error report" feature to record data on application failures in the real world. A similar approach can be used to get realistic data on failure of libraries.

It was argued that most library incompatibilities under Linux can be detected easily because they appear as a descriptive error code and error message sometimes even immediately at the start of execution. However, this has not been proven, and it still does not account for all other kinds of library failures. One promising method to help detect and test the proper behavior of new libraries is multi-version execution as shown by Cook and Vedagiri in 2002 [4]. Even if multi-version execution is only shown to be beneficial in some cases, these special cases can be chosen and selected to contribute to the overall efficacy of an incompatibility-detector.

The user-program (shell) approach for library incompatibility detection was chosen over a kernel-based approach. However, as mentioned earlier it is possible that a kernel-based detector may have distinct advantages. Another advantage, in addition to overseeing a wider range of program executions, is that it can actively monitor indirect shared library calls such as dlopen() and dlsym(). Although these are set up in different ways on different flavors of UNIX, the implementation of the functions is typically tied to the operating system.

The simple model of the library protection daemon is designed to keep safe the public system libraries. However, private libraries for specific applications can possibly benefit from portability improvement as well. The model may be able to be changed to account for these, but it will need some method to locate private libraries that are in use.

Although Linux is the prime target of this study, Solaris and FreeBSD have very similar, perhaps even worse, problems with dynamic linking compatibility, and so the ideas presented in this paper may be well applicable to them as well. Originally, this study was to take Mac OS X into account, but so far the author finds it unclear how serious the shared library problems are on Macintosh systems.

9. Related Work

As mentioned earlier, Cook and Vedagiri have been studying multi-version execution to improve reliability of upgrading. Although at the present time this has only been tested on simple library cases, there is hope that it can be extended to C++ libraries or other more complex scenarios [4].

To aide in fixing common DLL problems on Windows, Anderson wrote a tool known as the DLL Universal Problem Solver (DUPS). By analyzing the system for common problems and automatically solving them in much the same manner as a Microsoft technical support worker would, DUPS has already proven very useful in solving shared library problems [2].

Up until this point, the possibility of bringing online features to aide in portability has not been discussed. However, it is quite clear that a centralized server could be of great benefit, because many of the problems are caused by *not knowing* where the incompatibilties lie. However, bringing online information into the model would unfortunately make this a much more complex project and it would no longer be a general-purpose solution because it would require a centralized server. Keep in mind that this project is not intended to be anything like the expensive yet successful "Windows Update."

10. Conclusion

Because the shared library effect is such a long-term and user-specific problem, it is very difficult to reliably test the benefits of this proposed portability protection. Still, the ideas presented here look promising in that they may be sufficient to solve some simple test cases of real-world problems caused by shared libraries. These fixes are targeted primarily at Linux operating systems. The basic protection system structure consists of the library protection daemon combined with the shell-based library incompatibility detector. Plus, there are many ways in which the foundation can be extended to deal better with various kinds of libraries and programs.

11. References

[1] Ayukov, Sergey. "Shared libraries in Linux: growing pains or fundamental problem?". ayukov.com. 1999.

[2] Anderson, Rick. "The End of DLL Hell". MSDN. Microsoft Corp. 2000.

[3] "Frequently Asked Questions: Microsoft DirectX". www.microsoft.com. Microsoft Corporation. 2003.

[4] Cook, Jonathan, and Navin Vedagiri. "Reliable Upgrading of Unix Shared Libraries through Multi-Version Execution". *New Mexico State University Department of Computer Science Technical Report*. Las Creuses, NM. 2002.

[5] "Linux Quake HOWTO: Quake II", linuxquake.com. 2003.

[6] Daley, R, et al. "Virtual Memory, Processes, Sharing in MULTICS". *Communications in the ACM*. 1968.

[7] Duggan, D. "Type-Safe Linking with Recursive DLLs and Shared Libraries". to appear in *ACM Transactions on Programming Languages and Systems*. 2003.

[8] Orr, Douglas, et al. "Fast and Flexible Shared Libraries". *Proceedings of Summer 1993 Usenix Conference*. 1993.

[9] Bershad, Brian, et al. "Safe Dynamic Linking in an Extensible Operating System". Seattle, WA. 1995.

[10] Nelson, Michael, et al. "High Performance Dynamic Linking Through Caching". *Proceedings of the Summer 1993 Usenix Conference*. 1993.

[11] "SkyMap Pro Support Issues". Thompson Partnership SMP Support Information Page. 2003.

[12] Desitter, Arnaud. "Using static and shared libraries across platforms", fortran-2000.com. 2003.

[13] "[Talk] State of dynamic linking in various platforms". BBS. www.auug.org.au. 2003.

[14] "Debian Policy Manual. Chapter 9 – Shared Libraries". debian.org. 2003.

[15] Phoenix, Chris. "Windows vs. Unix: Linking dynamic load modules". 2003.

[16] "Deploying the shared (dynamic) library". C++ Portable Types Library. melikyan.com. 2003.

[17] Cockroft, Adrian. "Which is better, static or dynamic linking?". SunWorld. February 1996.

[18] Menke, Gregory. "Multiple libc versions under Debian". BBS. linux.umbc.edu. 2003.