65

Shared Libraries in an Exokernel Operating System

by

Douglas Karl Wyatt

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1997

Author ....
Department of Electrical Engineering and Computer Science
August 28, 1997

Certified by ....
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Shared Libraries in an Exokernel Operating System

by

Douglas Karl Wyatt

Submitted to the Department of Electrical Engineering and Computer Science on August 28, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Exokernel operating systems export much of the raw hardware interface to applications, allowing each application to provide its own operating system abstractions and interfaces. While this removes the burden of dealing with sub-optimal abstractions that traditional operating systems force on applications, there are potential costs of higher memory and disk consumption by applications that are statically linked with large amounts of Library Operating System (LibOS) code. This increase in application size can result in poor cache performance, increased paging to disk, slower process load times and possible upper limits on the number of concurrently running applications. As most applications are expected to use the same LibOS, or large portions thereof, use of LibOS's as shared libraries would alleviate much of this problem. Since traditional mechanisms for loading shared libraries rely on high level operating system abstractions, implementing a LibOS as a shared library presents a difficult bootstrapping problem: How to read the file and virtual memory systems from disk without a file or virtual memory system?

This thesis presents a design and implementation of a solution to this problem. A Shared Library Server (SLS) is implemented, which provides access to basic file I/O and VM routines via a simple inter-process communication (IPC) interface. A small startup library is built that performs the same process as traditional shared library loaders but using the SLS to read files from disk and manipulate the virtual memory of the loading process. As such, it does not rely on the functionality provided by the LibOS. This startup library is statically linked with the application, adding approximately 27 KBytes to the application, as opposed to the entire LibOS code, which is currently more than 680 KBytes. This greatly reduces the size of applications and speeds up the process execution times.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Shared Libraries in an Exokernel Operating System

by

Douglas Karl Wyatt

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Exokernel operating systems export much of the raw hardware interface to applications, allowing each application to provide its own operating system abstractions and interfaces. While this removes the burden of dealing with sub-optimal abstractions that traditional operating systems force on applications, there are potential costs of higher memory and disk consumption by applications that are statically linked with large amounts of Library Operating System (LibOS) code. This increase in application size can result in poor cache performance, increased paging to disk, slower process load times and possible upper limits on the number of concurrently running applications. As most applications are expected to use the same LibOS, or large portions thereof, use of LibOS's as shared libraries would alleviate much of this problem. Since traditional mechanisms for loading shared libraries rely on high level operating system abstractions, implementing a LibOS as a shared library presents a difficult bootstrapping problem: How to read the file and virtual memory systems from disk without a file or virtual memory system?

This thesis presents a design and implementation of a solution to this problem. A Shared Library Server (SLS) is implemented, which provides access to basic file I/O and VM routines via a simple inter-process communication (IPC) interface. A small startup library is built that performs the same process as traditional shared library loaders but using the SLS to read files from disk and manipulate the virtual memory of the loading process. As such, it does not rely on the functionality provided by the LibOS. This startup library is statically linked with the application, adding approximately 27 KBytes to the application, as opposed to the entire LibOS code, which is currently more than 680 KBytes. This greatly reduces the size of applications and speeds up the process execution times.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Acknowledgments

I would first like to thank Prof. Frans Kaashoek for his guidance and support as an advisor. His patience and encouragement proved invaluable in making this thesis a success. Working with him has enabled me to experience a breadth and depth of systems research that will serve me well.

I would also like to thank all those in the Parallel and Distributed Operating Systems Group for their assistance and friendship. Without their help, this would never have been possible. In particular, I would like to thank Tom Pinckney, Hector Briceno, Rusty Hunt, Dave Mazieres, Emmett Witchel and Greg Ganger for answering all of my questions, even the dumb ones. I would like to thank all of the other members of the group as well for making all the hours spent in lab enjoyable, or at least tolerable.

I would also like to thank all of my friends here over the past 5 years for my making my stay at MIT so far a most memorable one. They have made MIT seem a bit less formidable, somewhat more human and a lot more fun.

Finally, and most importantly, I would like to thank my parents for all of their support. In addition to writing large checks to MIT on my behalf on a regular basis, they have provided the encouragement and support, without which I couldn't even have dreamed of getting to MIT, let alone graduating from here.

This work was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Exokernels are operating systems that are designed with a new interface for process/kernel interaction. The driving principle behind exokernels is to give applications as much control over hardware resources as possible. This design leads to the removal of most, if not all, abstractions that are provided by traditional kernels. These abstractions are typically provided in user level library operating systems (LibOS's). Each application is linked with the LibOS that the developer wishes to use, and becomes part of the user-space code. The operating system abstractions and interface can thus be customized on a per-application basis, forcing applications to pay for only the abstractions that they want or need. This ability to provide custom operating system abstractions would be prohibitively expensive if processes using the same LibOS could not share code, though.

The inclusion of large amounts of LibOS code for each application has clear performance and resource implications. As we expect most applications will be linked against a standard LibOS with a UNIX-like interface, much of this code will be the same for many applications. The fact that this significant amount of code will be common amongst many processes presents an even stronger argument for the use of shared libraries than in traditional systems.

Shared libraries allow common collections of code to be shared by multiple applications, thus reducing the amount of redundant disk and memory usage. These libraries are linked into shared object files, and are loaded, if necessary, and mapped into the application address space before the program begins. Typically, applications have a small startup code section that is run at startup and that loads the necessary shared libraries into the address space. This code reads data stored in the application, which contains the name and version number for the shared libraries it was linked against, and uses the file system and virtual memory interface to place an image of each shared library into memory and then patch indirection tables for each symbol that is defined in a shared library so that it contains the correct address.

This process is not trivial, often relying on rather high level file system and memory system operations to read shared libraries from disk and place them in memory. In traditional systems, these abstractions are provided by the kernel or via subsystems that are running and universal for the entire system. In exokernel systems, though, these abstractions and functions are provided by the LibOS. Thus, loading the LibOS as a shared library leads to the bootstrapping problem of how to load the LibOS from disk into memory, when the ability to do so is dependent on code in the LibOS.

In this thesis, this problem is solved by a Shared Library Server (SLS) and startup library that provides enough I/O and memory functionality to perform the operations necessary to load shared libraries into a process's environment. This is accomplished through a limited interface for the required operations which is provided via interprocess communication (IPC) between the process that is attempting to load shared libraries and the SLS. This way, with a small amount of IPC abstraction implemented, and another set of abstraction semantics agreed upon, the process has access to a set of high-level abstraction functionality.

This thesis describes the design and implementation of the SLS and alternative solutions to this problem. The current implementation is for Xok, an exokernel developed on the Intel x86 platform. In Chapter 2, issues surrounding shared libraries are discussed. Then, in Chapter 3, attributes of exokernels, and Xok in particular, that affect the use of shared libraries and the implementation of a shared library mechanism are discussed. In Chapter 4, design principles that were used to guide the development of a solution are described. Chapter 5 describes the solution that was implemented, including a design overview, discussion of the implementation issues encountered and an analysis of this solution compared with the design principle objectives. Several alternative solutions are also discussed and analyzed. Chapter 6 makes a performance analysis of the proposed solution and one of the alternatives that has also been implemented. Chapter 7 draws conclusions from the results and describes future work directions.

Chapter 2

Shared Libraries

2.1 Background

Shared libraries enable multiple applications to share common libraries or collections of functions. The concept of sharing physical memory between processes was originally described and implemented in the Multics Project[2]. Several more robust and abstract implementations have been implemented since then. Most UNIX systems today implement a version based on that which was designed for System V[1]. This design allows shared libraries, which may themselves be reliant on other shared libraries, to be compiled. These shared libraries export a set of text and data symbols that applications may be linked against. Applications that uses shared libraries have a small amount of code that is run during startup, which loads the needed shared libraries into memory, performs any run-time relocations necessary and then jumps to the user-defined `main()` routine. Numerous optimizations on this implementation have been proposed and implemented, most notably by Ho and Olsson[4]. Additionally, many implementations of shared libraries allow for on-the-fly symbol relocation, which allow the process to replace functions during runtime without modifying the text segment.

2.2 Issues

Shared Libraries allow for multiple applications that use a common set of functions to load the code for those functions once and share the memory between those processes that rely on it. This is done by mapping a single physical memory page, or set of pages, into multiple process virtual address page tables, allowing each process to see the physical pages in their address space. This can have a number of benefits, including decreased resource consumption, increased application load speed and development environment improvements. But there are also some issues that need to be resolved in order to enable the sharing of code between processes. These include ensuring that the address references in the shared library text segment valid in each process environment and implementing a useful application development environment interface to use of these libraries.

2.2.1 Enabling Physical Memory Sharing

A large part of the benefits that using shared libraries achieves comes from the fact that physical memory is conserved. In a hierarchical memory system with cache, primary memory and paging to disk, this can greatly improve the performance of applications because references are more likely to serviced by faster memory as there is less physical memory in use for a given workload. In systems with absolute constraints on the amount of memory available, this can greatly increase the number of applications that can be run simultaneously.

In order share code between applications in memory, the code must be exactly the same for each application. Given this, either the environments, or certain portions thereof, of all involved processes must be identical, or there must be a way to mask the differences. For instance, the addresses of locations in the shared library code that are referenced from within the shared library must be the same for all processes, or there must be a mechanism to determine the addresses.

This requirement leads to the most central issue involved in shared library systems, that of symbol resolution. There are several ways to ensure that a symbolic reference to a particular location in a shared library is resolved correctly at run-time. First, and most basic, is to require that a given shared library always be loaded into the virtual address space at the same location. This scheme is depicted on the left of Figure 2-1. This both allows the application to know where symbols in the shared library will be loaded at link-time, but internal references to symbols within the shared library are also known when the shared library is compiled. This proposal is possible, and has even been implemented in some systems. However, more robust solutions are possible.

The more common implementation involves indirection tables for symbols in the shared library. Both the application which use shared libraries and the shared libraries themselves have a data area which is populated with the actual addresses of the symbols when the shared library is loaded. Each process environment has private copies of these tables. This can be seen in the center of Figure2-1. All references to symbols in the shared library, both from the application and the shared library itself, consist of loading the actual address from the appropriate table and then jumping to, loading from, or writing to the correct address. Shared libraries can also reference internal symbols through relative offset instructions. These instructions allow jumps, loads and stores that are addressed as offsets from either the current program counter or some other register. Since the relative offsets of symbols are known when the shared library is compiled, these offsets can be hard-coded into the shared libraries and will be correct independent of the load address of the library. This optimization is depicted on the right side of Figure 2-1. However, relative offset references eliminate the potential of replacing these symbol bindings at run-time, and thus are usually only used for references to local symbols, that is, symbols which are purely internal to a given object file, and thus would not be dynamically replaceable.

While this method involves an extra level of indirection for each external symbol reference, the performance loss tends to be small and the benefits of being able to load the libraries at any address and modify the tables at run-time are large. Modifying the indirection table during run-time would allow, for example, a word processor to be linked against a generic printer driver interface, and load and unload drivers for

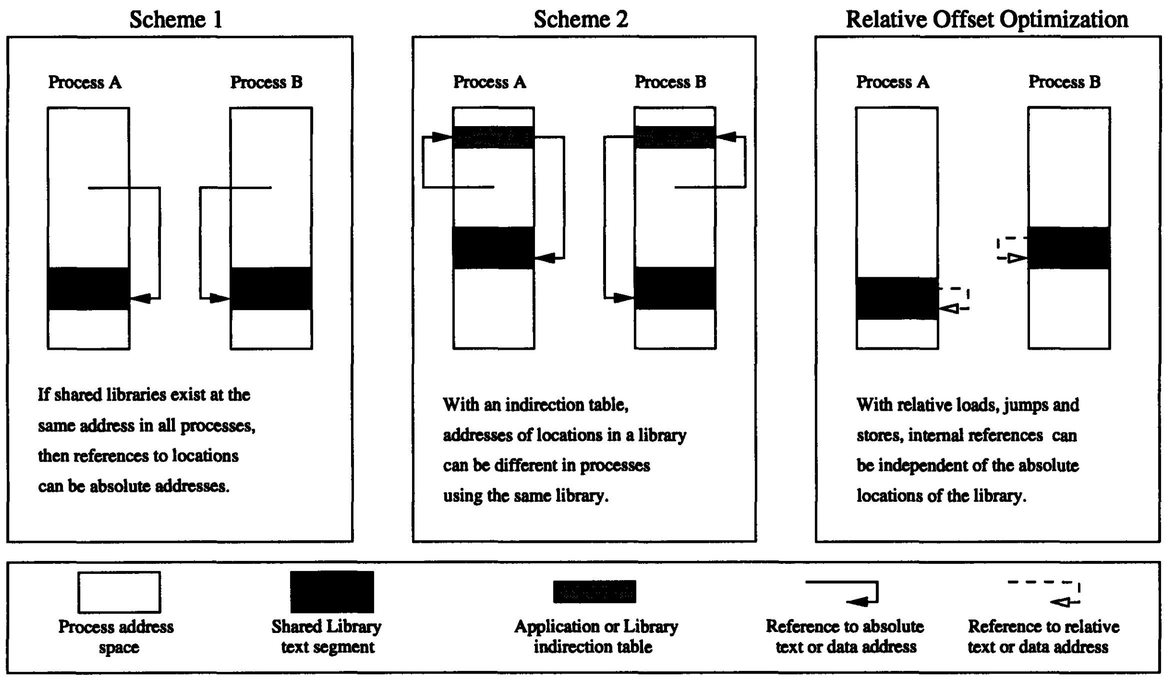
Scheme 1
Process A
Process B
If shared libraries exist at the same address in all processes, then references to locations can be absolute addresses.
Scheme 2
Process A
Process B
With an indirection table, addresses of locations in a library can be different in processes using the same library.
Relative Offset Optimization
Process A
Process B
With relative loads, jumps and stores, internal references can be independent of the absolute locations of the library.
Process address space
Shared Library text segment
Application or Library indirection table
Reference to absolute text or data address
Reference to relative text or data address

Figure 2-1: Three ways of ensuring correct resolution of addresses in shared libraries.

specific printers while running, instead of including every possible printer driver in the application statically.

2.2.2 Application Development Environment

Another issue that must be resolved when designing a shared library implementation is that of how they will be used or referenced during the application development process. Typically, shared libraries are compiled into `.so` files which are linked against when building applications. The application, instead of including the shared library in the actual application, merely contains a reference to the shared library its version number. This process decouples the shared library and application development process. This allows the shared library to be updated, to some degree, without requiring applications that use it to be recompiled.

```
Compile_Application
   Determine which symbols are defined in shared libraries
   Allocate Indirection Table for shared library symbols
   Convert references to these symbols to indirections through table
   Create DYNAMIC data structure which contains, for each library :
      Names of shared libraries
      Major and minor revision numbers

Begin_Application
   Load process text and data segments into memory
   FOREACH shared library in DYNAMIC structure
      Load_Shared_Lib(lib_name, major_rev, minor_rev)
   Jump to main

Load_Shared_Lib(lib_name, major_rev, minor_rev)
   Search paths for shared lib_name
   If found and (major revision number == major_rev)
      If (minor revision number < minor_rev)
         Warn user that library may be out of date.
      Map shared library text and data segment into address space
      Patch indirection table contents for application and library
      If library_name was linked against shared libraries itself
         FOREACH shared library that (lib_name) uses,
            Load_Shared_Lib(new_lib_name, major_rev, minor_rev)
   Else, print error and die
```

Figure 2-2: Pseudocode for shared library compilation and loading in System V UNIX

2.3 Implementation

Different operating systems implement shared libraries with a variety of the previously described concepts. The method that was chosen for this thesis is based on the System V UNIX implementation[1]. The source code for this method is derived from the implementation in the OpenBSD UNIX distribution and modified to work in the Xok environment. Figure 2-2 shows the process by which applications that used shared libraries are compiled and executed.

2.3.1 Compilation, Linking and Loading Applications

This design uses a combination of indirection tables and relative references. Each process (or shared library) which is dependent on one or more shared libraries has an indirection table for all symbols that are defined by the shared libraries.[1] References to these symbols are indirected through this table. When the process is initiated, a small amount of startup code is run that loads the shared library somewhere into the process's address space. Based on the location of where the shared library was loaded, the startup library fills in the indirection tables with the correct locations of the symbols. If any of the shared libraries loaded are themselves dependent on other shared libraries, these secondary shared libraries are loaded as well and the process is continued recursively until all needed libraries have been loaded and the relevant tables have been populated.

In loading the shared libraries, typically there are built-in routines for mapping a disk file or set of disk blocks into a region of virtual memory. These calls will load the disk blocks into physical memory and then insert the appropriate mappings for those physical pages into the process's virtual page table. In cases where those disk blocks are already resident in physical memory, all that needs to be done is to insert the mappings for those pages into the new process's page table. This is how physical memory is shared. Using this technique, an arbitrary number of processes can see the same shared library in their address space at different virtual addresses while only one copy of it is actually resident in physical memory.

2.3.2 Optimizations

Once this has been done, the user defined code can be executed since the indirections will be correct in the current environment. In reality, many implementations actually perform some of these symbol address calculations lazily. In particular, references to text segment symbols are left as the address to a special routine. When a process

[1]Actually, many linkers calculate all references to external symbols through an indirection table when an application is linked against shared libraries, not just symbols in the shared libraries. Addresses of application symbols are contained in an initialized data area and read from disk.

attempts to jump to a function address, it actually jumps to this handler routine which then determines where the text address really is, replaces the entry in the table with this value and then jumps to this address. This is especially useful for applications that are linked against large shared libraries but which use only a few procedures. Thus, only the portions of the indirection table which are actually used are filled in.

2.3.3 Compiling Shared Libraries

Shared libraries themselves are compiled as Position Independent Code (PIC) in this implementation. This involves making the code of the library independent of the address it is loaded at. There are several ways this could be done. One is to implement all loads and stores as program counter relative instructions. As such, the 'address' of a symbol is how far it is from the instruction referencing it. Typically, however, this is performed through an indirection table similar to that with which the application references shared library symbols. This table, actually two tables: one for for data symbols and one for text symbols, is generated by the linker and it is at a known relative offset from the code. Without some mechanism of allowing a shared library to exist unmodified at different virtual addresses, all shared libraries would have to be loaded at a predetermined address which would limit the number shared libraries one would be able to load.

2.3.4 Benefits

This design offers a number of benefits. First, overall system resources consumption is reduced. When an application attempts to load a shared library that is being used by another application, the memory consumption for that shared library is amortized over all processes that use it. This can be quite a significant saving, as the sizes of typical shared libraries as seen in Table 2.1 can be quite large compared with the actual processes. For instance, `libc.so` is used by almost every application written in C, so on a typical multi-user system, the number of processes sharing this

Shared Library	**Size in KBytes**
libc.so.15.0	464
libg++.so.27.1	305
libcurses.so.3.1	104
libtermlib.so.2.0	83
libtermcap.so.0.0	9

Table 2.1: Typical Shared Library Sizes on x86 OpenBSD

library can be in the hundreds. While each process, were it linked statically would admittedly not necessarily need all of the code in `libc.so`, the memory saving are still quite significant. Additionally, since the processes that rely on a shared library don't include the code in their own binaries, only one copy of the library needs to exist on disk. While disk space is inexpensive, it is not free, and this can provide significant reductions in disk usage. More importantly, with the reduction in the physical disk space used to store applications, both the disk buffer cache and the main memory cache performance improves.

Additionally, this process can improve performance. For short running applications, the time needed to load the process from disk can be significant. The use of shared libraries allows this to be reduced by eliminating the need to load much of the code when a process is initiated. Additionally, and probably more importantly, with the reduction of overall memory consumption, page faults are less frequent so applications run faster in the general case. The only drawback is that references into shared libraries pay an indirection cost of looking up the address in a table.

Finally, one side benefit of this implementation of shared libraries is that system evolution is greatly aided. Shared libraries can be compiled and maintained independently of the applications which use them. Thus, if an implementation of a routine in a shared library proves to be inefficient or incorrect the library can be modified and recompiled. Then, every process which is dependent on that library will be able to use the new version without having to be recompiled. Only when major changes to the shared library, such as the interface to existing functions are changed or new functions are added, do the applications that use them need to be recompiled. Typically, older versions of the library are left available so that programs that have not been recompiled still function correctly; they merely do not take advantage of the new library. The determination of which version of a shared library is made via the major and minor revision numbers of a library, which are appended to the end of the file names. For instance, `libc.so.14.0` and `libc.so.15.0` represent different major revisions of `libc.so` while `libc.so.14.0` and `libc.so.14.1` represent different minor revisions which are interchangeable.

Chapter 3

Problem: Shared Libraries for Exokernels

Exokernels can provide significant performance improvements as compared to traditional operating systems. However, due to their design, sharing text segments of LibOS's is important. Unfortunately, traditional mechanisms for doing so do not work.

3.1 Background

Exokernels are operating systems that are designed with the principle of exporting as much information and control of hardware resources as possible to the application. As Engler, Kaashoek and O'Toole state, "Traditional operating systems limit the performance, flexibility and functionality of applications by fixing the interface and implementation of operating system abstractions..."[3] While giving the application the ability to customize such interfaces and implementations allows them to achieve maximum performance and flexibility, requiring the user to program at such a low level would be impractical and burdensome for most applications. The solution is to implement useful sets of operating system abstractions in one or more Library Operating Systems (LibOS's), which applications can include to provide a useful high-level interface to the hardware resources.

The current exokernel operating system that is under development runs on x86 architectures and is called Xok[5]. The previous version, Aegis, ran on MIPS machines. Currently, a LibOS, ExOS, has been implemented which provides abstractions and semantics roughly equivalent to BSD4.4.

The unique design of exokernels has several implications on the use of shared libraries. The design makes the use of shared libraries much more significant in system performance. While traditional operating systems share such code implicitly in the kernel, sharing LibOS code in an exokernel system would significantly reduce the physical memory usage. Unfortunately, the design also makes a shared library system much more difficult to implement.

3.2 Increased Benefits for Shared Library Usage

Since exokernels move much of the functionality of traditional operating systems into user level library operating systems (LibOS's), a typical statically linked application is much larger than the same application on a UNIX-like operating system (in the ExOS, the increase is more than 500 KBytes). Most applications in an exokernel environment will be linked against one or a few LibOS libraries which provide a common set of OS interfaces. The ability to pull the LibOS code out of the application and put it in a shared library would have enormous effects on the disk space required by the typical suite of application available as well as the physical memory required to keep multiple applications resident. It would also drastically reduce the time needed to load applications, eliminating the need to read more than 500 KBytes of ExOS information every time an application starts up when the application code can be as small as several kilobytes. Additionally, with large amounts of LibOS code being shared between processes, buffer and main memory cache performance will improve.

3.3 Difficulties using Shared Libraries

Unfortunately, exokernels make it rather difficult to implement shared library systems. The traditional method, as implemented in the startup library and the run time linker (`common.c` and `ld.so` in most UNIX systems) does not work for the fundamental reason that it relies on the existence of high level OS system calls (such as `open`, `read`, `mmap`, etc.) that are provided by the LibOS in an exokernel system, and thus are not available to load the LibOS. This problem is harder than it may seem, since many of these calls require significant portions of the LibOS to operate properly, so duplicating the functionality in the startup code is not a satisfactory option.

Additionally, ExOS code often relies on data structures that are used by the startup code. Shared libraries cannot have any undefined external references including references to data or text symbols in the application that loaded (except through explicit pointer passing) Since the startup code must have these symbols defined, and the space allocated, in order to run, and the LibOS must be able to reference the same data, ExOS needs to learn what the locations of these symbols are somehow. For instance, data structures which contain information about the process environment and file descriptor tables are initialized before ExOS is loaded, but used quite heavily in ExOS. ExOS has no way of knowing when it is compiled where those data structures will be, so some mechanism for informing it once it is loaded must be implemented.

Thus, in order to implement a shared library system, the following obstacles must be overcome. First, processes must be able to access the file system without the LibOS. Ideally, this includes both local disks as well as any remote file systems, such as NFS or AFS, that are available. Additionally, it must also be able to manipulate its virtual address space in a way that allows the mapping of the shared library to take place in cases when the shared library is already in memory. Both of these tasks must also be accomplished without 'reimplementing' another LibOS in the startup code. Finally, a mechanism needs to be in place that allows the startup code to update the data structures in the LibOS once it is loaded.

Chapter 4

Design Principles

In attempting to solve these problems, several design principles were formulated. These principles enumerate what the design goals of the solution are with their relation to the performance and robustness of the system.

4.1 Maximize Code Sharing

The first principle, maximizing code sharing between processes, come directly from the need for shared libraries in the first place. A shared library system that required a significant portion of of common code to be linked statically into each application would not solve our problem. While supporting generic user-defined shared libraries is one of the end goals of the system, the initial need arose from the fact that each application was being statically linked with ExOS, bloating even a typical `HelloWorld` program to almost 500 KBytes in size. A solution which did not allow ExOS code to be used as a shared library would not be sufficient.

In order for a piece of code the be able to be shared between processes, it must be exactly the same in each process. This can be difficult because the virtual memory layouts of different processes are quite unique. One possible technique for this would be to require that a given shared library always be loaded at the same virtual address in every process that loads it. If this were true, then all internal references could be hard coded and would still work. This is possible and has been used in several systems.

With the use of PIC code, however, the same piece of text segment can exist at different virtual addresses and not require any text segment relocation. This is a big win for shared libraries because this eliminates the need to determine, at compile time, where a given shared library must be located, and no central record keeping or address space allocation policy needs to exist.

4.2 Minimize Indirection Costs

Additionally, if it is possible to reduce the traditional indirection costs of shared libraries, this would be desirable. The standard mechanism for implementing shared libraries is to include an indirection table in the data segment of the application. Shared libraries themselves also contain a data area which is used for symbol addresses. All references to symbols defined in shared libraries are then converted to indirections through these tables. These indirections are expensive compared to a standard jump or load instruction. A shared library mechanism that did not require such run-time indirections would have better performance than one which used the standard strategy.

Additionally, a significant load time performance cost is undesirable. While spending significant time performing calculations and modifying the data or text segment might provide for improvements in run time performance, one must weight these improvements with the additional time necessary to perform these optimizations. There are cases where the time necessary to perform such operations dwarfs the possible improvements in post-load run-time performance.

4.3 Robustness

The final principle is to achieve similar robustness, as a development tool as well as a run-time tool, as that of traditional shared library mechanisms. The ability to compile shared libraries and applications independently is an important benefit of shared library systems. This reduces the compile time of an application greatly because all of the shared library code need not be compiled. More importantly, when a shared library changes, not having to recompile every application that uses that library is a big win from the development perspective.

Additionally, a high level naming scheme for shared libraries is useful. This, as opposed to sharing files based on inodes or disk blocks, allows for shared libraries to be moved from location to location and from machine to machine easily. It is easier for a developer to think about sharing particular files which encapsulate functionality than to worry about which disk blocks contain particular basic blocks which might be called from other code.

While most applications will be linked against a standard LibOS, the solution should work all applications. A solution which only works for applications linked against ExOS would not be as useful as a general case implementation. Additionally, if LibOS's could be broken down into small modules, such as a separate file system, virtual memory management system, network stack, etc., it would be desirable that these could be shared in a fine grained manner. Requiring a specific set of these sub-systems would limit the usability of such a system.

The ability to change symbol bindings at run-time is a powerful tool available for traditional systems. New modules can be loaded in place of old one while the application is running, providing different capabilities without bloating the application or library.

Finally, the ability to use widely available and supported development tools is important as well. Many tools, such as `gcc` and `ld` are widely available and well supported. The ability to use these tools, as opposed to necessitating the development and maintenance of separate ones, is highly desirable. In designing the shared library mechanism, a strong effort was made to satisfy all of these goals.

Chapter 5

Solution: Shared Library Server and Startup Library

Implementing a shared library mechanism with a statically linked LibOS is not difficult. The standard run-time linker (`ld.so`) works with minor modifications to the startup code used for Xok. The difficult task is to implement a solution which enables the use of ExOS as a shared library as well. Such an implementation is the focus of this thesis.

The solution that was developed is centered around a Shared Library Server (SLS). The SLS is a process that is initiated when an exokernel is booted. Applications are statically linked with a small startup library which performs the necessary work of loading the shared libraries. The startup code makes proxy LibOS calls through the SLS to perform all of the needed I/O and virtual memory work necessary to load the shared libraries. Then, once the shared libraries for the application have been loaded, calls are rerouted to the LibOS of the actual application. The proxy LibOS calls are performed via IPC calls to the SLS. The following section will describe the design and implementation of the SLS, modifications to the startup code and conclude with an analysis of the solution both numerically and as compared with the design principles enumerated in Chapter 4.

5.1 Design Overview

The overall design of the Shared Library Server solution of enabling application use of LibOS's as shared libraries was divided into two parts; work to be done by the actual process and work to be done by the SLS. The decisions about this were made primarily based on the amount of code necessary to perform these operations and how integral to the standard operation of ExOS. Functions that do not require much code to re-implement, or whose implementation for the purposes of loading shared libraries could be quite minimal could be implemented by the actual process. Other functions which were complex or relied heavily on much ExOS to operate would be implemented remotely by the SLS.

After examining these issues, it seemed logical to put the file system, screen and console I/O and virtual memory functionality in the SLS, and the smaller, more fine grained routines, such as a number of the string operations and the malloc family operations, in the startup code itself.

5.2 SLS Implementation

The Shared Library Server was implemented as a IPC server. On startup, it registers itself with an IPC nameserver running at a well known location. The name server allows other programs to discover where the SLS is located in order to be able to connect to it. It also allows for the possibility of multiple SLS-type programs to be running and for applications to have the ability to chose among them for a particular interface or functionality. Additionally, with modifications to the nameserver, it could allow for multiple SLS processes to be round-robin served to applications to reduce the load on particular SLS processes, thus improving throughput, as the SLS currently blocks on I/O requests.

Once running, the SLS listens for requests from applications that are starting up and attempting to load shared libraries. It provides the ability to open, read, write and mmap files from disk, as well as open and read directories and perform some

Table 5.1: Interfaces Implemented in the Shared Library Server

Function Name	**Description**
`open`	Opens a file for reading
`read`	Reads a specified number of bytes from a file
`lseek`	Sets the file position of a file
`close`	Closes a file
`mmap`	Maps a region of a file into memory
`munmap`	Unmaps a regions of memory from a file
`mprotect`	Sets the protection bits on a region of memory
`dup`	Duplicates a file descriptor
`dup2`	Duplicates a file descriptor to a specific descriptor
`opendir`	Opens a directory for reading
`readdir`	Reads the next directory entry from a directory
`closedir`	Closes a directory
`status`	Displays to console the status of the SLS
`printf`	Prints to the terminal or console a string
`printd`	Prints to the terminal or console an integer

basic screen and console I/O such as printf and kprintf. Table 5.1 describes those functions which are provided via IPC by the SLS.

File operations are based on 'proxy file descriptors' which the SLS uses to index into an internal structure containing information about local file descriptors. Periodically, the SLS cleans up its internal data structures, removing entries for file descriptors that were left open for processes that no longer exist.

Memory operations are performed through a LibOS interface which allows processes to read and modify page tables of other processes. With the use of explicit capabilities for these regions of the client applications memory, the SLS would have access to only the area needed and only for the time needed, thus reducing the amount of trust necessary between the client and server.

Screen I/O is provided to a minimal extent because it is useful to be able to print out messages both to debug the code during development as well as tell the user what happens if an error occurs before the shared libraries have been successfully loaded. There are interfaces for printing out both strings and numbers (in both decimal and hexadecimal) to both the console and to the tty.

The SLS is 630 lines of C code, which translates to 6 KBytes of object code prior to linking with the startup code and LibOS. As a comparison, the resulting application is slightly larger than cp. If this process is linked statically with ExOS, the total size is approximately 500 KBytes. The nameserver which was implemented to allow applications to find the SLS, as well as any other service-providing servers, is less than 100 lines of C code.

5.3 Startup Library

Another large portion of the changes to the system involved modifying the startup code used by applications to prepare the environment before calling the user defined main procedure. For applications using ExOS, this includes code to set up the environment variables, process information, map in shared segments of memory and initialize the file descriptor table. For applications that use shared libraries, this also includes code to map the shared libraries into the process virtual memory address space and perform the necessary run-time linking for the symbols defined in the shared libraries.

Traditional applications are linked against a small amount of startup code which contains process initialization routines. In this implementation, this has been augmented to include such initialization routines, as well as the code to perform run-time relocations and IPC stub routines to interface with the SLS. The initialization routines are necessary to allow IPC to function properly, and to prepare certain data structures.

This code relied on a number of LibOS routines which in most systems were available from the OS. These routines are not available until ExOS is loaded, so some other mechanism must be provided. For the file, VM and other I/O routines that the SLS provides, IPC wrapper stubs were implemented with the same syntax and semantics as the real calls. For other calls, bare minimum implementations were included statically in the startup code.

A crude memory allocation system was implemented to replace malloc, calloc,

Table 5.2: Interfaces Implemented in the Startup Code

Function Name	**Description**
`getuid`	stub user id functions
`getgid`	stub group id functions
`getenv`	returns the environment data structure
`strerror`	Returns an error string
`strsep`	Separates a string based on a delimiter
`strdup`	Duplicates a string
`strcmp`	Compares two strings
`strncmp`	Compares n bytes of two strings
`strcat`	Concatenates two strings
`strchr`	Locates a character in a string
`memcpy`	Copies a set of bytes from one location to another
`ui2s`	Converts an unsigned integer to a string
`strtol`	Converts a string to a base 10 integer
`malloc`	Allocates an n byte chunk of memory
`free`	Null, as opposed to the standard implementation
`calloc`	Allocates an n byte chunk of memory and zero's it out
`realloc`	Enlarges a previously allocated chunk of memory
`bzero`	Fills a region of memory with zero's

realloc, free and bzero. This was done using an incremental pointer into a region of well known VM. All new allocations are done at the next point in this location, with the size of the allocation being stored first. When an attempt to realloc is made, if the new size is larger than the original size, the contents are copied to fresh space at the end of the incrementing pointer. This process does not allow for reuse of memory that is deallocated via free or realloc, but during tests, less than a 4k page of memory was used during the entire process so this is not much of an issue. Additionally, several string manipulation and evaluation routines were also implemented statically in the startup code. Fortunately, this only required about 150 lines of C code to implement, thus not bloating the startup code too much. Much of this is used in ld.so to deal with file names and path names of the search paths for loading in the shared libraries. Table 5.2 describes the routines which were re-implemented in the startup code of processes.

Incidentally, to reduce the complexity, the code to the run-time linker which is typically a shared library itself, `ld.so`, was statically compiled in with the startup code. There is no inherent reason why this has to be true, and to reduce the size of the statically compiled portion of the startup code this could be removed and used as a shared library. However, with the addition of a technique described later in this thesis, the benefits of this would be negligible.

5.4 Modifications to ExOS

The final step, which was more difficult than first imagined, was to convert ExOS into a shared library. First, the routines which are only used during startup were removed as vestigial. In several places, interfaces were added to allow the startup code to set values of shared library data structures so that the process state which the startup code initialized could be passed on to ExOS once it was loaded.

Then, in numerous places in ExOS, `asm` statements which manually inserted sequences of assembly instructions had to be modified to be PIC compliant. When procedures are compiled as PIC code with OpenBSD gcc and ld, the `ebx` register is reserved to hold offset information for the Global Offset Table, used in the calculation of addresses of symbols in PIC code. In some of the manual insertions of assembly code, this register was used, destroying the state that the compiler depends on. This was common in system calls which had 3 or more arguments because the system call interface passed the third argument in this register. Instead of modifying the kernel system call interface, a `pushl ebx` and `popl ebx` were placed surrounding instances of assembly which used this register, thus preserving the visible consistency of this register.

Finally, there were a number of symbols used in ExOS which referred to kernel exposed data structures. Several data structures which the kernel used were placed in specific locations in the process address space, and thus were not part of the application data segment. The addresses of these symbols had traditionally been set by using a `DEF_SYM(symbol,address)` macro that inserted assembly directives to create symbol table entries at specific addresses as absolute references. The linker would not relocate these references or allocate storage for these symbols as the storage was allocated by the kernel and the address did not change as a function of the load address. However, the compiler could not handle these references correctly when compiled as PIC code since it did not know it was an absolute reference until it had already generated the assembly instructions to perform the indirection.[1] Additionally, the linker was unable to handle these references in application code that used shared libraries as well. It was unclear if this was a bug in `ld` or a deficiency in its specification. This was solved by replacing all such references with constant addresses `cast`'s to the appropriate types in header files. This has the unfortunate consequence of requiring all code using these symbols to be recompiled when one changes, as opposed to only requiring a re-link, but no other solution to the problem was evident.

5.4.1 Implementation Difficulties

In addition to previously stated problems, there were a number of difficulties encountered while implementing this solution. First, and most frustrating, was the lack of documentation on both specifications of standards and implementations of tools such as `ld`. It was quite difficult to find sources which accurately described the mechanism by which shared libraries were implemented and loaded, run time relocations were performed, requirements of `ld` for correct operation and the like. Going into this project, the concept of linking seemed rather straightforward. There were many unexpected complexities which made it far from that, though.

Reliance on standard tools proved periodically problematic as well. Several bugs in standard development tools were discovered during this project, most of which exercise non-standard code paths[2]. Because of this, even knowledgeable people enlisted for help had a hard time discovering the causes of certain behaviors.

Finally, implementing this process while the implementation of the static LibOS and kernel were evolving was challenging as well. Changes were made in the sys-

[1]This could conceivably be solved by implementing a `__attribute__ absolute` option for data declarations in `gcc`.

[2]For instance, `ld` writes the RRS DATA section incorrectly when linking a program with both the `-T text_offset` and `-Bdynaminc` options.

tem and its interfaces which required reworking previously working portions of this implementation.

5.5 Design Criterion Analysis

This solution satisfies the stated design criterion well. This is in large part due to the fact that it maintains much of the interface and mechanism as traditional shared library systems, which were designed with many of the same goals in mind.

5.5.1 SLS Maximizes Code Sharing

The SLS solution maximizes the code sharing between applications. A shared library implementation that did not allow the library operating system to be implemented as a shared library would not achieve similar benefits on code reduction, either on disk or in memory, as one that did. Since the current solution allows even the LibOS to be used as a shared library, it is reasonable to expect that any other library would be implementable as a shared library using this mechanism.

The one possible shortcoming of this solution, however, is that the startup code has expanded from about 3 KBytes to almost 27 KBytes (when stripped) of code that is statically compiled with the application, and thus not shared. While this is not very large in comparison with many applications being written, it is still an issue. There is a solution to this problem, though. A hybrid of this and one of the alternative solutions discussed in the next section would eliminate this code sharing problem for the startup code without losing the other benefits of the SLS solution.

5.5.2 Minimize Indirection Costs

The SLS solution does not add any indirection costs to shared library references compared to traditional solutions. While a single level of indirection is still present, this cost has shown to be with reasonable bounds and any improvements to general case shared library loading techniques should be applicable to the solution implemented as well[4]. The startup costs, including runtime relocation, are the same as with traditional shared library implementations as well.

5.5.3 Robustness

This solution is quite robust, as well. It allows for shared library file relocation and recompilation, as long as the external symbol interface remains the same, just as traditional implementations do. Additionally, the interface to modifying symbol binding at runtime remains available because the indirection is made through a table in the data segment. This allows for actual dynamic linking at run time.

5.6 Alternative Solutions

During the design of the Shared Library Server solution to loading shared libraries, several other alternative solutions were proposed, and one was implemented by another member of the group. Here several of these solutions are described and analyzed them with respect to the design criterion.

5.6.1 Absolute Mapped LibOS image

One proposal to the problem of sharing LibOS code was for `exec()` to map an image of a 'ghost process' which included the LibOS code into each applications address space, so that each application did not have to include the LibOS code but could instead rely on it being present when it was executed. Applications are linked against an assembly stub file which sets the addresses of all the LibOS symbols to absolute addresses. Then, the image of the process that actually contained all of the LibOS code is mapped into each process when loaded so that the absolute symbol addresses are correct.This method was implemented in the Xok environment for ExOS by Thomas Pinckney.

Implementation

This method was implemented by first compiling a program with an empty main procedure with all of the ExOS code. Similar to traditional shared libraries, every object file that could be needed by an application must be present. From this executable, an assembly file is generated which has symbol definitions for all of the externally defined symbols in ExOS with the locations defined as where they would be if the ghost process were loaded at a particular address (in this case it was 0x10000000).

Using this assembly stub file, programs are compiled without ExOS code and all references to ExOS symbols are relocated at link time to where they are defined in the stub file. Then, when a program is executed, the `exec`'ing process actually loads the application and the ghost process. The application is loaded at the traditional address while the ghost process is mapped into the address space at the predefined location in such a way that the absolute addresses as defined in the assembly stub file are correct.

A few subtleties exist in the technique, most regarding back references from the LibOS code into the application. The LibOS needs to reference the program's main procedure, for which it does not know the address at the ghost process compile time, and which may be different for each process using the ghost process LibOS code. This is solved by having the entry point to the process place the address of the user defined main procedure in a data segment location for the LibOS so that it can call that procedure after the process startup code in the LibOS has completed.

Design Criterion Analysis

This solution achieves some of the goals as defined by our design criterion in Chapter 4.

First, it achieves a very high rate of code sharing for the LibOS, and conceivably other libraries as well. There is no overhead for loading in shared libraries in terms of code size, unlike traditional methods of using shared libraries and the Shared Library Server method. The entire text segment of the shared libraries can be shared, which seems to be the best of both worlds from a code sharing standpoint.

Additionally, there are no indirection costs. Since programs know the address of all LibOS (and shared library, in general) addresses at link time, there is no need for an indirection table. All references, including text segment and data segment symbols, are direct jumps or loads. This avoids the table lookup and run time relocation costs of traditional systems.

Unfortunately, this solution has some severe deficiencies in the robustness domain. First, maintaining a ghost process image is a bit clumsy. Additionally, because the virtual address region of each library must be known at link time, every possible set shared libraries that could be used together must not have overlapping addresses. This is not a scalable solution, as it limits the number of shared libraries that could be run together and would require some sort of central authority to allocate virtual address regions for shared libraries. Finally, whenever even a minor modification to ExOS occurs, all applications must be recompiled because the symbol locations will have changed. While an version of this method could be conceived which worked for multiple LibOS's, it would be significantly more complicated and is not currently implemented.

5.6.2 Disk Block Fingerprinting

Another solution proposed by Dawson Engler was to share libraries through the sharing of identical disk blocks. This would be implemented by extending the inode structure to include a 'fingerprint', for instance a 32 or 64 bit CRC of the disk block contents. In the file system, when an attempt to read a file is made, the finger print of the disk blocks are read from the inodes and if any disk blocks that currently reside in the buffer cache have an identical disk block fingerprint, the block is not read, but mapped from the buffer cache instead. Thus, if the LibOS is linked in its entirety and placed at the beginning of every application, all symbol references in it would be relocated to the same address, except for several references into the application specific text or data segment, and thus most disk blocks would be identical between applications in their LibOS code. When the application is read in by `exec()`, the LibOS code would be shared automatically.

Design Criterion Analysis

This proposal performs well against the design criterion in several aspects but has some severe shortcomings.

It will indeed allow for a significant amount of code sharing. If `exec()` were implemented such that the few references to the application text and data segment were patched after they were read, the entire LibOS would be sharable. Additionally, this idea could potentially lead to faster reading of disk blocks that happen to be identical but are not known to be by the system. For instance, and programs that happen to have been statically with the same .o files which are offset in the resulting application at the same point off a disk block would be shared by this method. The odds of this happening could be increased by ensuring object files filled to the next disk block size. This does not achieve the benefits of reduced disk storage needs since the duplicate blocks still exist on disk. This is not as important as the memory costs and startup time benefits, though, since disk storage is cheap and growing so quickly.

It would perform better than the SLS solution in terms of indirection costs. Since applications are linked statically, there are no indirections into the shared library code. All jumps are direct since the location of the symbols is known at link time. All 'libraries' must exist at the same location in applications to achieve this benefit though. It would be reasonable to expect this for LibOS code, but extra care would have to be taken to achieve this for other libraries since the number of applications that use specific combination of shared libraries is smaller than the number that use a specific LibOS.

This method, however, has serious shortfalls in robustness. First, the entire application must be compiled and linked and written to disk, unlike traditional systems where the shared libraries are not relocated or written to disk. Additionally, any changes to the shared library code necessitates a recompilation of all applications that use that library for the changes to be effected. Additionally, this solution requires low level modification of the file system which would have to be present on all file systems mounted, including any network file systems. It also prevents the use of any dynamic linking as there is no indirection table to be modified at run-time.

5.7 Summary

The Shared Library Sever implementation seems to provide the best combination of attributes compared with the design goals stated in Chapter 4. It provides a high degree of physical memory sharing and conservation in addition to being highly flexible and robust. While the alternatives would provide the ability to implement direct references to symbols which might improve performance slightly, the cost in flexibility and robustness is very high.

Chapter 6

Performance Analysis

Another important measure of the success of this design is the effects it has on the performance of applications in the Xok environment. To gain more insight into how well this implementation performs, both microbenchmarks and application performance tests were run.

6.1 Experimental Setup and Methodology

These tests were performed on a Intel Pentium 166 MHz computer with 64 megabytes of RAM. It was running Xok and applications were linked against the current version of ExOS. All executables, libraries and data files used in these tests are loaded from a remote NFS file server over standard 10 MBit/sec Ethernet.

These measurements attempt to discern the performance comparisons between statically linked programs, dynamically linked programs loaded via the SLS, and programs linked against the ghost process image. Microbenchmarks such as the speed of specific operations via the SLS compared with those operations as implemented with the process are used to determine the overhead incurred by using IPC to make these requests. Additionally, tests of specific applications were run to determine both process execution time comparisons, and holistic run-time comparisons. Results are averages of 10 trials, and the standard deviation is less than 3% of the average in all cases.

Table 6.1: Time (in milliseconds) per `read()` call : SLS Vs. Native ExOS Operations

Size of read	Native ExOS	SLS	Difference (% increase)
128 Bytes	0.289	0.335	.046 (15.9%)
256 Bytes	0.561	0.599	.038 (6.8%)
512 Bytes	1.090	1.140	.050 (4.6%)
1024 Bytes	2.160	2.210	.050 (2.3%)

6.2 Microbenchmarks

In order to determine the overhead of using the SLS to perform I/O and other functions, tests were run using the SLS and native ExOS calls to read from files. These tests consisted of reading 10,000 contiguous segments from a file ranging from 128 to 1024 bytes per segment using both the native ExOS `open()` and `read()` as well as via the SLS. Both methods were identically linked and used identical code, with the exception of the file operations. The results in Table 6.1 show that the additional costs for making proxy calls through the SLS server amount to between 38 and 50 microseconds for per call. The overhead ranges between a 16% increase for small (128 byte) reads to a 2% increase for larger (1024 bytes) reads. Loading the current version of ExOS as a shared library results in 78 calls made through the SLS, so the run time is increased by less than 4 milliseconds compared with a system which made these calls through ExOS directly.

Then, using a combination of the cycle counter in the Pentium processor and process run times, measurements were taken to determine the costs of `exec`'ing a process. Cycle counts up until the `main()` procedure was called were taken for statically linked programs, dynamically linked programs and programs linked with the absolute mapped ExOS when they were resident in the buffer cache. The overall run time of the process was used when they were not in the buffer cache, since the startup costs were large enough that the body of the program was negligible. Thus the runtime was a reasonable approximation for the startup cost when the startup cost was that high. This was necessary because startup cost was a combination of the time necessary to load it from disk as well as the time between the process was

Table 6.2: Process `exec()` Times (in milliseconds)

App State	Static App	Dynamic App via SLS	Absolute Mapped Apps
On Disk	1196	152	23
In Cache	1.2	20	4.4

initiated and when it reached the `main()` procedure.

Table 6.2 shows the results of these tests. For applications in the buffer cache, the static application outperforms the dynamic application by 18.7 milliseconds. This is consistent with separate measurements of the individual cycle-counts for the call to `rtld()`. This procedure, which performs the run-time relocations, accounted for approximately 18 milliseconds for applications which were linked dynamically with ExOS. The applications which were linked with the absolute mapped ExOS are slower than the static applications due to a small amount of additional code in the startup sequence which copies the `ARGV` array and manipulates a small amount of the page table.

For applications that are not in the buffer cache, however, the dynamic application provide a large performance improvement over static applications. With the shared ExOS resident in memory, the `exec()` time for dynamic applications was only 13% of that of static applications. As expected, applications linked with the absolute mapped ExOS were faster still, as no run-time relocations are necessary, and the application itself is smaller because it does not contain the startup library designed for dynamic applications.

6.3 Process Run-Time Performance

To further measure the startup costs of applications, Table 6.3 shows these cost per perl `exec()` call as measured during a test of a perl program which recursively executes itself 1000 times. The tests include all three types of linked programs, as well as under OpenBSD for comparison. The difference between the dynamic application

Table 6.3: `Exec()` Call Cost (in milliseconds)

App State	**Static App**	**Dynamic App via SLS**	**Absolute Mapped Apps**	**OpenBSD**
Time	39.7	69.7	22.9	35.5

and the version linked against the absolute mapped ExOS show a 46.8 millisecond increase in the perl `exec()` times. This is larger than that accounted for in the raw `exec()` costs shown in Table 6.2. This is expected, though, since there is overhead beyond merely `exec`'ing the process when a perl process is initiated.

Figure 6-1 shows the results of run time tests of larger programs to measure overall process performance. These involved `cat`'ing and `diff`'ing files that were not in the buffer cache. Figure 6-1 shows the resulting execution times. These were averaged over 10 executions, with the application resident in memory. These tests measure the overall performance effects of run-time relocations and converting ExOS to PIC. In all cases except for `sed`, the execution time of the dynamically linked program was within 2% of that of the statically linked program, and within 3% of that of the program linked with the absolute mapped ExOS. In the case of performing a simple `sed` pattern-replace on a 1 MByte file, the dynamically linked version was 10% slower than the static application and 5% slower than the application with an absolute mapped ExOS.

Disk space consumption was likewise improved. As is evident in Figure 6-2, dynamically linked programs are on average between 450 KBytes and 550 KBytes smaller than their statically linked counterparts. Applications linked with the absolute mapped ExOS were 30-50 KBytes smaller still. This benefit is due to the fact that these processes do not include the startup code necessary for loading shared libraries.

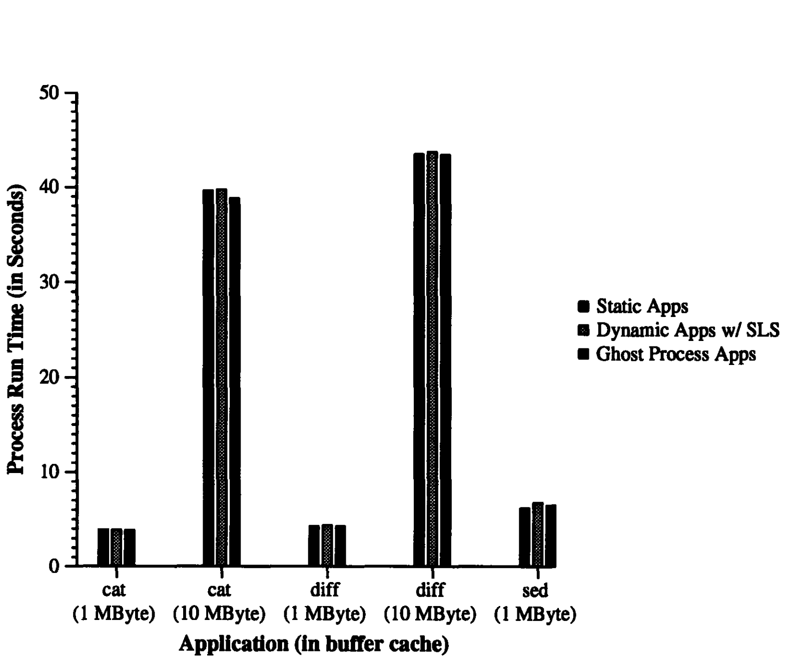

Figure 6-1: Normal Process Run Times

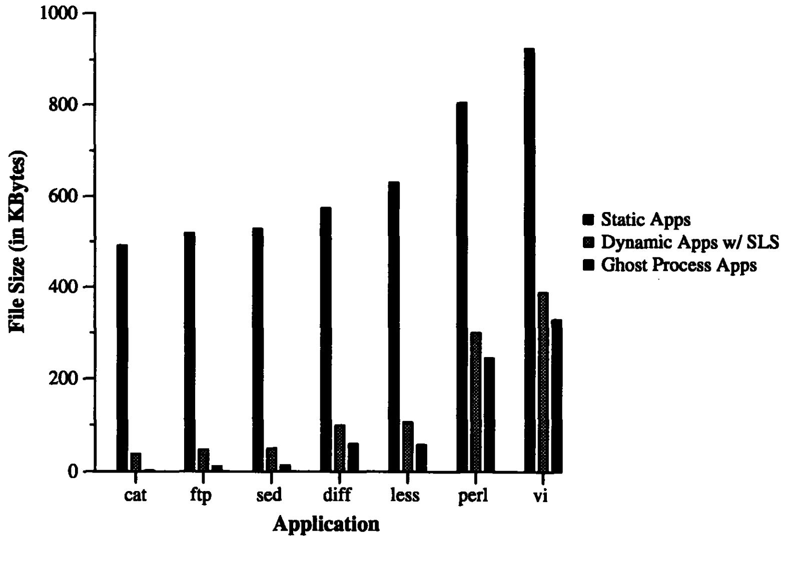

Figure 6-2: Disk Space Consumption for Applications.

Chapter 7

Summary

7.1 Conclusions

The ability of processes to bootstrap their LibOS's via IPC has proved quite successful. In addition to gaining all of the memory and disk space consumption benefits, as well as significant development environment advantages, the process execution time for applications not in the buffer cache drops by almost an order of magnitude. This is an absolute necessity for an operating system that intends to have reasonable global performance in an environment where numerous applications are run simultaneously. While either implemented solution satisfies the necessary performance and resource consumption requirements, the robustness of the SLS solution adds a large improvement in the development environment, as well as enabling true dynamic linking, that the other does not. In all, this project succeeded in meeting the design goals specified at its initiation as well as providing significant performance improvements.

7.2 Future Work

There are several optimizations to the current solution which could be made. First would be to merge the two implemented methods such that the startup code is linked in the same way that the absolute mapped Exos image is used. This would eliminate the extra disk consumption as well as shorted execution times by eliminating the disk I/O necessary to read this in. Additionally, optimizations suggested by Ho and Olsson suggest even better performance possibilities[4]. For processes that need to be executed before the SLS can be started, separate versions should be compiled which use shared libraries. Once the SLS is loaded, these new versions would be spawned to replace the older ones which have their LibOS statically compiled with them, thus allowing them to benefit from the memory requirement reductions using shared libraries. This could also be done for the SLS itself. The shared library version of the SLS could be brought up and replace the mapping in the nameserver for the environment id in which the new SLS exists. The old version could then terminate itself once it determined it was not needed anymore.

The power that exokernels give to processes comes bundled with a requirement that the processes implement the abstractions that they want to rely on. This can migrate bootstrapping problems up from the kernel to the user processes. Loading shared libraries is a prime example of this problem. Processes would like to take advantage of memory and disk abstractions to load shared libraries, but are also required to implement these abstractions. This problem is encountered regularly in the boot process of operating systems, but traditional systems provide these abstractions to, and enforce them on, the processes so that the processes have these abstractions available at execution time.

The solution proposed to this problem could be a model for other similar situations. IPC could be used to provide access to abstractions or functionality that the process did not want to implement. In this way, processes could chose among a possible variety of servers implementing services with different semantics or qualities. With explicit access control transfer via capabilities, the server could be given access to whatever resources were necessary to provide such a service. This might allow implementation of services where there is a high degree of complex shared state between processes that use a particular service or shared resource abstraction. Allowing a server to maintain such shared state through an explicit interface could enable the enforcement of stronger semantic checks on the integrity of such shared state than is available through sharing of low level resources such as memory pages.

Bibliography

[1] J. Q. Arnold. Shared libraries on UNIX^{TM} system v. In *USENIX Conference Proceedings*, pages 395–404, Atlanta, Summer 1986.

[2] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the Fall Joint Computer Conference, 1965*, pages 185–196, Las Vegas, November 1965.

[3] D. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountian, CO, December 1995.

[4] W. Ho and R. Olsson. An approach to genuine dynamic linking. *Software - Practice and Experience*, 21(4):375–389, April 1991.

[5] M. F. Kaashoek, D. Engler, G. Ganger, et al. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating Systems Principles*, Santo Malo, France, 1997.