Read-only Sharing in Operating Systems

Ramesh Govindan

Computer Science Division
Department of Computer Science and Engineering
University of California
Berkeley, CA 94720

ABSTRACT

Sharing read-only code and data between address spaces reduces physical memory usage as well as paging I/O and disk space. In this paper, we study mechanisms for read-only sharing in operating systems. Our work consists of three parts. First, we discuss the issues that arise in the design and implementation of such mechanisms. We examine the solutions adopted by different operating systems. Second, we analyze memory and disk space savings obtained by sharing programs in 4.3 BSD Unix. We show that introduction of a more general sharing mechanism in 4.3 BSD has potential for substantial savings. Finally, we analyze the efficiency of different sharing mechanisms. The read-only sharing mechanism in DASH is shown to be as efficient as the Unix mechanism for program code sharing. Moreover, the DASH approach allows for general read-only sharing (such as shared libraries), not just program code.

November 16, 1989

This research has been supported in part by the University of California with a MICRO program grant, Cray Research, Inc., Hitachi Ltd., IBM Corp., Ing. C. Olivetti & C., S.p.A., and the International Computer Science Institute. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing official policies, either expressed or implied, of any of the sponsoring organizations.

Table of Contents

Read-only Sharing in Operating Systems

Ramesh Govindan

Computer Science Division
Department of Computer Science and Engineering
University of California
Berkeley, CA 94720

1. INTRODUCTION

Physical memory is shared between address spaces for one of two reasons:

- *Read-write sharing*: The shared physical memory is used to communicate between processes executing in the address spaces.
- *Read-only sharing*: Sharing read-only code and data reduces physical memory usage as well as paging I/O and disk space (Section 2).

In this paper, we study the latter kind of physical memory sharing.

Our work consists of three parts. First, we list issues that arise in the design and implementation of a mechanism for read-only sharing in operating systems. When and how references from shared code and data are resolved is one example of such an issue. Management of updates to shared code and data is another. We qualitatively examine solutions to these issues. We also show how the following systems have solved some of these issues: 4.3 BSD Unix [7], SunOS 4.0 [5], System V Unix [2], Multics [9], DASH [1] and Cedar [10].

Second, we analyze the memory and disk space savings obtained by sharing programs and libraries in 4.3 BSD Unix. Different processes executing the same program share the read-only code of the program in 4.3 BSD. We show that in a typical 4.3 BSD system, about 30% of the programs are shared by two or more processes at any given instant. About 4 times as much physical memory would be used if read-only program code were not shared. We also show that with the introduction of shared libraries, there is potential for doubling the physical memory savings. Further, at least 40% of the disk space used by programs can be saved. These results show that substantial savings can be obtained with the introduction of a more general sharing mechanism in 4.3 BSD.

Finally, we analyze the efficiency of different read-only sharing mechanisms. We show that a more general sharing mechanism need not necessarily tradeoff efficiency for generality. Our experiments prove that the DASH read-only sharing mechanism is *as efficient as the Unix mechanism for program code sharing*. Moreover, DASH allows for general read-only sharing (such as shared libraries), not just program code.

This research has been supported in part by the University of California with a MICRO program grant, Cray Research, Inc., Hitachi Ltd., IBM Corp., Ing. C. Olivetti & C., S.p.A., and the International Computer Science Institute. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing official policies, either expressed or implied, of any of the sponsoring organizations.

The structure of the paper is a follows. In Section 2, we list some of the benefits that a read-only sharing mechanism can provide. Section 3 lays down some terminology used in the rest of the paper. Section 4 discusses, in some detail, the issues involved in the design and implementation of a read-only sharing mechanism. Section 5 lists the experiments we conducted and draws conclusions from the results obtained. Section 6 shows how different operating systems solve those issues. Finally, we present our conclusions in Section 7.

2. BENEFITS OF READ-ONLY SHARING

The following are some of the benefits of read-only sharing:

- *Physical memory savings*: In the absence of read-only sharing, each address space uses its own physical copy of read-only code or data. Thus, physical memory is saved with read-only sharing (Section 5.1).
- *Reduction in I/O*: For the same reason, read-only sharing is responsible for a reduction in I/O (paging traffic from local disk or network).
- *Disk space savings*: If libraries can be shared between address spaces, the disk images of programs need not contain copies of the shared library code. This results in disk space savings (Section 5.2).

Most existing multi-user systems (e.g. 4.3 BSD Unix) provide mechanisms for sharing read-only code of programs. The benefits of sharing program code decrease in an environment of networked single-user workstations. This is because the probability that two or more processes execute the same program program simultaneously is low. More general read-only sharing mechanisms (e.g. shared library mechanisms) offer potential for greater benefits (Section 5.2).

Trends in software structure also indicate the need for more general read-only sharing mechanisms. Operating systems designers nowadays tend to provide only minimal mechanism at the kernel level. This increases the complexity of application programs. Much of this complexity is hidden from application programmers in toolkits and libraries. Sharing of libraries and toolkits, therefore, offer promise of substantial savings in physical memory, disk space and I/O.

3. TERMINOLOGY

Read-only sharing refers to the sharing of read-only code and data between different address spaces or between different processes in the same address space. We use *code* to refer interchangeably to executable machine instructions and high-level language statements. The exact usage will be clear from the context.

A *segment* is a range of virtual memory representing a logical unit of code or data or both. Associated with every segment is backing store containing the *disk image* of the segment. The term *segment* is used to refer interchangeably to a range of virtual memory as well as its associated disk image. The exact usage will be clear from the context.

A segment has the following attributes: a *name* for the disk image, a set of *entry points* (starting addresses of functions or base addresses of data structures) and embedded references (*inter-segment references*) to other segments of the form <*segment name, entry point*>. Segments may be read-write or read-only, data segments or code segments. *A read-only segment is the unit of sharing between address spaces.*

An example of a segment in 4.3 BSD Unix is program code, which forms a read-only segment (called the *text segment*). The disk image of the text segment is part of the program file (the file also contains some initialized data). The text segment is shared between address spaces.

The creation of the disk image of a segment is called *segment compilation*. The creation of a segment and association with its disk image is called *segment loading*. *Load time* refers to the interval of time during which a segment (and, in some cases, other segments it references) is loaded. *Compile time* refers to the interval of time during which the disk image of a segment is created. *Run time* refers to the interval of time during which a process actually executes a code segment.

In some systems, when segments are loaded into an address space, an initialization routine needs to be executed. *Segment inclusion* is the process of loading a segment and execution of its initialization routine. The term is sometimes used interchangeably with loading.

4. ISSUES IN READ-ONLY SHARING

The following are the issues that arise in the design and implementation of a read-only sharing mechanism:

- *Resolution of inter-segment references*: What is the latest instant when intersegment references can be resolved — at compile, load or run time?
- *Address assignment to segments*: Do segments have to have fixed load addresses or can they be arbitrarily relocated? In particular, can shared segments be loaded at different addresses in different address spaces?
- *Segment naming and referencing*: How are segments named? Once intersegment references are resolved, how are functions within segments referred to?
- *Version management*: How are updates to segments dealt with? Are versions of segments maintained, and, if so, how are versions named and specified?
- *Private data of code segments*: How is read-write data referenced by read-only code (*private data*) handled?
- *Dynamic inclusion of segments*: Can segments be included dynamically in address spaces or must the entire collection of segments be loaded before program execution can begin?
- *Sharing boundary*: Does the operating system allow multiple processes per address space so that segments can be shared within the same address space by different processes (*intra-address space sharing*)? Can segments be shared only between address spaces (*inter-address space sharing*)?

In this section, we discuss these issues in detail. Solutions to the issues are outlined and the tradeoffs between the different solutions examined. A similar list of issues may be found in [6].

4.1. Resolution of Inter-Segment References

Resolution is the process of replacing a symbolic inter-segment reference (Section 3) by a memory address. Resolution can be performed at the latest at compile time, load time or run time. The choice of when and how references are resolved impacts the nature and extent of read-only sharing possible in systems. In this section, we discuss mechanisms for resolution of references and list their merits and demerits. The discussion specifically refers to shared code segments. It is also valid for shared data segments which may contain inter-segment references.

Compile Time Reference Resolution

In this scheme, all references are resolved during program compilation. Compile time reference resolution can be done in one of two ways. One solution ensures that there are no inter-segment references left unresolved when the disk image of a segment is created. If functions or data are referenced by a segment, those functions are included in the disk image. The program loader merely includes this image into the address space. The sharing mechanism is simplified by the absence of inter-segment references. No run time reference resolution costs are incurred. However, code that is common to two segments cannot be shared since each segment gets its own copy of the code.

Another solution assigns a fixed range of addresses to segments (Section 4.2). Inter-segment references can be resolved at compile time. Sharing is at the level of individual segments. The loader includes all the segments referenced by the program into the address space. Since segments are assigned a fixed range of addresses in all address spaces, no code relocation is necessary. The only overhead involved is that of including (at load time or run time) the segments in the address space. This is relatively high for the first inclusion, but, since the segment is shared, lower for subsequent inclusions.

Load Time Reference Resolution

In this scheme, references may remain unresolved after program compilation. At load time, all referenced segments are included in the address space and references resolved. Symbolic references are reduced to absolute addresses by searching the included segments.

If segment addresses are fixed, references can be resolved at compile time. If references are resolved at load time, segments can be assigned different addresses in different address spaces. Code cannot be altered to reflect the binding of the inter-segment reference to a memory address (the *purity* constraint for shared code).

A *linkage table* (Figure 1) is used to resolve references at load time. The linkage table is a read-write segment. Inter-segment references are indirected through an entry in the linkage table. All entries in the table are initially null. At load time, those table entries are filled in with the actual address corresponding to the reference. The linkage table itself is referenced via a *linkage base register*.

This mechanism incurs the overhead of reference resolution at load-time. A search is involved to resolve an inter-segment reference. The cost of including the segment in the address space (finding a suitable address range and performing the mapping) is also incurred. Locating a segment in the name space (Section 4.3) is, however, a one time cost.

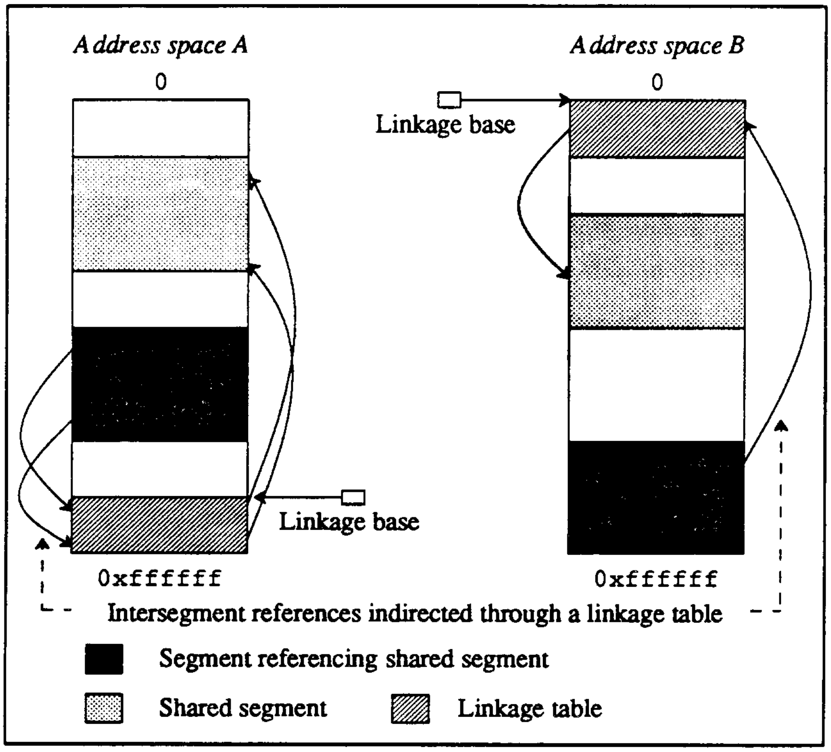

Figure 1
The linkage table
The linkage table is a read-write per address space segment through which all inter-segment references are indirected. In the figure, only the code of the shared segment is shared between address space A and address space B.

Run Time Reference Resolution

Mechanisms for run time reference resolution are similar to those for load time resolution. The linkage table mechanism (Figure 1) is also used here to ensure sharability of segments. At run time, an unresolved inter-segment reference (recognized by a special bit in the linkage table entry) generates a trap. The operating system then locates the segment and includes it in the address space. The inter-segment reference is resolved by filling the linkage table with the appropriate value.

Segments are only loaded when they are actually referenced. The scheme is efficient when the number of segments actually referenced is small compared to the number of inter-segment references.

However, an individual reference resolution is more expensive than if the reference were resolved at load time. Every inter-segment reference incurs the additional overhead of a trap.

4.2. Address Assignment to Segments

The problem of assigning address ranges to segments reduces to deciding whether a segment is loaded in the same address range in all address spaces or not. Assigning a fixed address range to a segment is called the *uniform addressing solution* [3]. Non-uniform addressing implies that segments can be loaded at arbitrary and, in particular, different address ranges in different address spaces. In this section, we qualitatively discuss the two approaches to this problem. This discussion specifically refers to shared code segments. It is also valid for shared data segments which may contain inter-segment references.

Non-uniform Address Assignment

Under this solution, segments can have arbitrarily different addresses in different address spaces. Address space management is not a problem. This leads to better utilization of the address space resource than is possible with the uniform address space solution.

However, the solution is expensive in performance terms. Since segments are loaded at different addresses, inter-segment references cannot be embedded into segment code. A separate segment called the *linkage table* (Section 4.1), is used for indirecting inter-segment references. Sometimes, the linkage table forms part of the data segment of every shared code segment. Thus every inter-segment reference incurs the extra cost of an indirection.

The code of shared segments has to be position independent (PIC). Normally, position independent references are relative to the program counter. This solution imposes a performance penalty on data and code accesses.

Moreover, under the non-uniform addressing scheme, shared segments are included in the address space at load time or run time. Since the addresses of segments are not known at compile time, inter-segment references have to be resolved when segments are included. Section 4.1 discusses the overhead incurred by load and run time reference resolution.

Uniform Addressing

Under uniform addressing, all segments are assigned a fixed address range. Segments are loaded at the same address in all address spaces. All entry points to a segment can be assigned fixed addresses. Thus, intersegment references can be resolved at compile time. References need not be indirected through a linkage table. Intra-segment references can be in terms of absolute virtual addresses and need not be position independent. The private data of code segments is also assigned a fixed range of addresses. Hence, the code sharing mechanism is extremely efficient.

The complexity of code sharing appears in a different guise, namely address space management. It now becomes necessary to assign a fixed range of addresses to every segment that is sharable. However, since this is not done dynamically, solutions need not be efficient.

Fixed allocation of address space ranges to segments is an inefficient use of address space resources. If two segments can possibly be co-resident in memory, they must be assigned disjoint ranges of addresses. The two may not actually be co-resident at any given instant of time.

A simple solution is possible in systems where code sharing is restricted to specific segments (such as libraries). An administrator statically assigns a range of addresses in every address space for different kinds of libraries — the standard C library, window system libraries, database libraries etc. It is unlikely that two different libraries performing the same function will be referenced by a segment at the same time (if necessary, one of them could be compiled along with the program). Thus, all libraries that perform the same function are assigned the same range of addresses.

In systems where sharing can be more general, each segment has to be assigned a fixed range of addresses. The problem of assigning address ranges to all segments system-wide is a formidable one, since the interdependencies between segments may be arbitrarily complicated. One way to simplify the problem is to split up segments into two classes — segments which may be shared system wide (standard libraries and utility programs) and those which may be shared among smaller groups of users. The two classes can then be handled separately.

Address assignment for the system-wide shared segments may be done by a system administrator. For those segments which may be used by a small group of people (working on the same project, for instance), the administration of the address space is left to the group. User level tools may be provided to manage the address range allocation.

Thus, uniform and non-uniform address assignment represent a trade-off between performance and flexibility. However, under the assumption that virtual address space as a resource is abundant, the uniform addressing solution is attractive from an efficiency viewpoint.

4.3. Segment Naming and Referencing

A segment name is a user-visible entity used to locate and identify disk images of segments. A segment reference is a machine-level entity used to access code within segments. For instance, a segment can be named by a path name of the form `a/b/c/d` while it can be referenced in program code by an absolute memory address such as `0x2f000`. In this section, we consider the design choices in naming and referencing segments.

Naming

Segments usually form part of the file system name space. In distributed file systems, different parts of the name space may be implemented on different file servers. Hence, segment location (file name resolution) in these systems can be quite expensive. However, if a segment is already in use, another reference to the segment does not incur the cost of name resolution. Thus, for actively shared segments, the cost of name resolution may be amortized over the number of address spaces sharing the segment.

Where the sharing facility is restricted to libraries alone, they may form a separate name space that can be more efficiently managed. Libraries are *mostly-read* and rarely updated. Use of this property may be made by keeping separate copies of libraries on local disks to speed up access.

Referencing

In systems where segments have fixed addresses, segment members are referred to by their virtual addresses. A separate mechanism ensures that segments are included in the address space before the reference is made. One solution lists all libraries referenced by the program in the file header. The loader ensures that referenced libraries are mapped into the address space before execution commences. Another solution is to have explicit system calls to include referenced segments (see Section 4.6). This solution is more expensive, since a system call is incurred for every segment inclusion.

In systems where segments can be assigned arbitrary addresses, intersegment references are indirected through an entry in the linkage table (Section 4.1). The table entry may contain the name of the segment before the reference is resolved. At run time, when the reference is made, a trap is incurred and the segment is included in the address space. Alternately, the table entry is marked unresolved. The file header contains the list of segments referenced. The loader includes the segments and fills up the linkage table.

4.4. Version Management

One of the benefits of sharing code among different address spaces is convenient update propagation (Section 2). Updates to shared code and data are available to users without the need to recompile their programs. Code segment updates can involve changes and extensions to the segment interface and improvements to the implementation of the interface. Each update to a segment creates a new *version* of the segment. Two versions are *compatible* if all programs that execute correctly with one version execute correctly with the other and vice versa.

Versions of a segment can, in general, form a tree. The root of the tree represents the original segment. Nodes at each level represent updates made to their ancestors in the version tree. Figure 2 shows a version tree for a standard C library.

Design of a mechanism to manage version trees of segments (*version management*) involves two issues. The first issue is how versions are designated. The second issue is at what level (user- or kernel-) the versioning mechanism is implemented.

In this section, we discuss version management with specific reference to shared libraries. The discussion is equally valid for code and data sharing in general.

Version Designation

The most generic designation is the Dewey decimal format, `x.y.z.w...`, where `x`, `y`, `z` and `w` are integers (Figure 2). In the degenerate case, a single integer can be used to designate versions. This corresponds to a two level version tree with the leaves of the tree representing updates to the original segment.

A common approach is a two component version designation of the form `x.y`. This corresponds to a three level version tree. Versions of the form `x.y` and `x.z` (those that are children of the same version of the original) are compatible. Versions that differ in the first component (those versions that do not have the same parent) are incompatible.

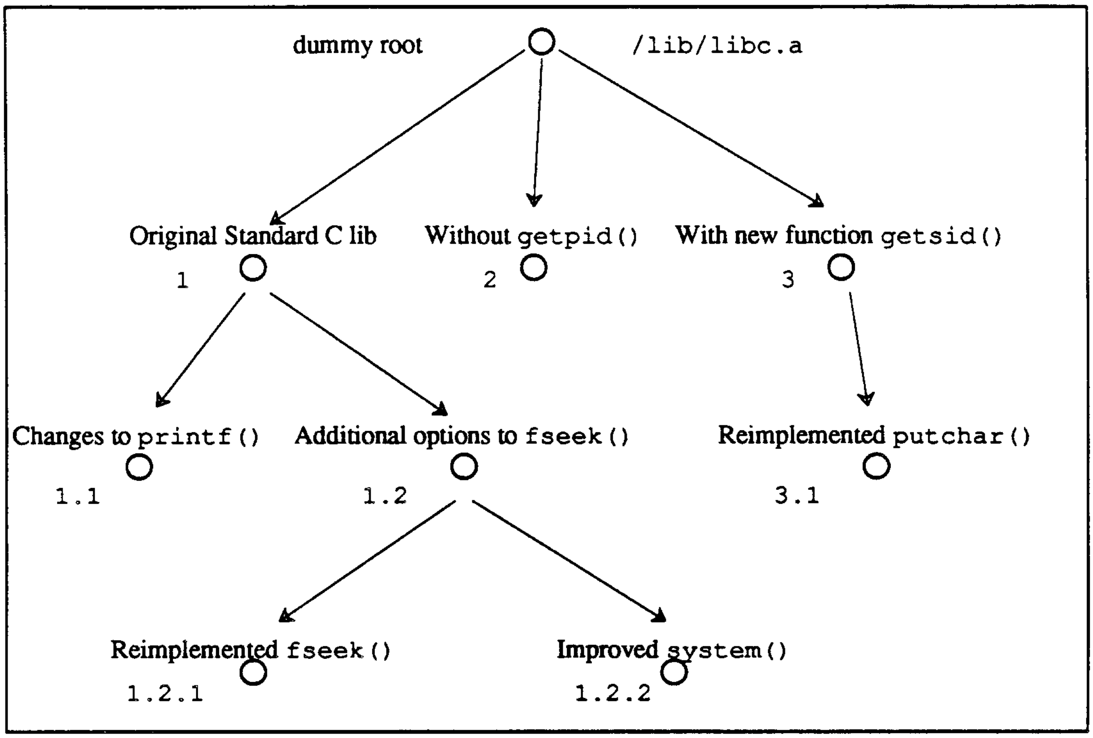

Figure 2

Version tree for standard C library

Each path in the version tree specifies a sequence of changes made to the original segment. Each circle represents a version and the numbers represent the Dewey decimal version numbers.

User or Kernel Level Versioning Mechanisms

Most operating systems provide no explicit support for versions of shared code. User-level tools are provided for version management [11]. Such tools provide support for versioning large software systems, where many developers may be simultaneously altering files.

However, it is difficult to extend them to provide support for versions of shared code. Libraries can potentially be shared by every program in the system. The version management system cannot keep track of the dependencies between programs and versions of the libraries they reference.

The fundamental problem in versioning libraries lies in the naming mechanism. In most cases, libraries form part of the name space of the file system (Section 4.3). Programs are embedded with the name of the file representing the shared library segment. Suppose that a program references `/lib/libc.a`, the standard C library. A new version of the C library would also have the same name, otherwise all programs would have to be changed. Thus, programs cannot specify access to a particular versions. All programs would have to use the same (e.g. the latest) version.

One solution to the problem embeds knowledge of a versioning mechanism within the operating system. The program loader knows about versions of libraries. The loader implements the semantics of the versioning mechanism when including versions of the libraries.

This solution lacks flexibility. The trend in modern day operating system design is to provide minimal mechanism within the kernel, so that different policies may be implemented at the user level.

User-level library versioning mechanisms are difficult to build in traditional systems because a name in the name space implies a single unique entity. Some systems have extended the concept of a name space to include *services* as nodes. Whenever a path traversal reaches a service node, a server implementing the service completes the traversal using the remaining components of the name.

Such a mechanism can be used to design a user-level file system with versioning. The file system is implemented as a service. The service can implement the semantics of the versioning mechanism without kernel intervention. For instance, a service implementing the two-component version designation (Section 4.4), when asked for version `x` can access version `x.y` such that `y` is the largest numbered child of `x` in the version tree (i.e. the most recent compatible version).

4.5. Private Data of Code Segments

While code can be shared, read-write data accessed by the code must necessarily be private to the address space. Usually, private data of a code segment forms a separate segment. Reference to data constitutes an inter-segment reference. If segment addresses are fixed, inter-segment references are absolute addresses. If load or run time reference resolution is employed, references to data segments are bound dynamically. All data references must be indirected through a linkage table (Section 4.1).

In systems that have single address space per machine, there is no notion of data private to an address space. Segment code as well as data is shared between processes, executing in the single address space. Access to shared data is regulated by some mutual exclusion mechanism.

4.6. Dynamic Loading of Segments

Some systems permit segments to be loaded at run time. *Dynamic loading* (also called *dynamic inclusion*) is independent of when inter-segment references are resolved. For instance, it is possible to resolve all references at compile-time and yet dynamically load the segment.

Mechanisms for dynamic loading of segments can either be *explicit* or *implicit*. Implicit dynamic loading occurs when an inter-segment reference to an unloaded segment generates a trap. This mechanism requires hardware support. Explicit dynamic loading mechanisms usually provide a system call to the loader for including the segment.

Dynamic loading incurs the extra cost of a system call or a trap. If the segment is not shared, the cost of segment inclusion is largely in locating the segment and loading it into the address space. When code is shared, the cost of the second and subsequent inclusion is small; it is just the overhead of the system call or trap. However, only segments that are actually referenced are included.

4.7. Sharing Boundary

A peripheral issue is whether segments can be shared between processes in an address space as well as between address spaces. In this section, we examine how the sharing boundary affects the code sharing mechanism.

Intra-address Space Sharing

Multiple processes in an address space can share the code and data of a segment. No extra mechanism is required except the ability to start up multiple processes in an address space.

No kernel overhead is involved in this type of sharing. However, processes must use some synchronization mechanism for accesses to shared data. Hence, this type of sharing is not transparent to the programmer. Moreover, sharing is on an all-or-nothing basis. All processes have the same privileges since the address space defines a protection boundary.

Inter-address Space Sharing

In this type of sharing, segments are shared across address spaces. If code segments are shared, each address space gets a private copy of the read-write data associated with the code. It is possible for a segment to be both intra- and inter-address space shared.

Additional mechanism is needed within the kernel to accommodate inter-address space sharing. This takes the form of data structures for keeping track of segments currently loaded in an address space. The copy of the segment in physical memory is reused after the first time it is included in the address space. However, that code is being shared is largely transparent to the programmer.

5. MEASUREMENTS

We conducted a set of experiments designed to answer the following questions:

- *Extent of sharing*: What is the extent of read-only sharing in an operating system such as 4.3 BSD Unix? How much reduction in physical memory does this level of sharing provide? We chose 4.3 BSD since it provides a very simple form of read-only sharing (see Section 6.1).
- *Potential savings from sharing*: What additional savings can be obtained by allowing a greater degree of sharing in 4.3 BSD Unix? For example, if library sharing were introduced in 4.3 BSD, how much reduction in physical memory and disk space can be expected?
- *Work done by different sharing mechanisms*: How much work is done by different sharing mechanisms at compile time and how much at load and run time? We use this as a yardstick for the efficiency of read-only sharing mechanisms.

The first two sets of questions are motivational in that they seek to establish whether a sharing mechanism is needed at all. The last set tries to quantify the efficiency of different sharing mechanisms.

This section discusses the design and outcome of these experiments. In designing the experiments, we strove for simplicity and ease of implementation. The numbers obtained give a feel for the quantities they represent. In some cases they provide lower or upper bounds but are not necessarily very accurate.

5.1. Extent of Sharing

The first experiment estimates the extent of read-only sharing in a "typical" multiuser 4.3 BSD Unix system. We loosely define a typical system to be one which is moderately to heavily utilized. The advantages of sharing manifest themselves under these load conditions.

Methodology

Text segments of programs may be shared between processes in 4.3 BSD Unix (see Section 6.1). The *extent of sharing* is defined as the ratio of programs shared to the total number of programs in memory at any given instant.

To compute this ratio, we sampled a typical 4.3 BSD Unix system (*ernie*, a VAX-11/785 at the Computer Science Division of the University of California at Berkeley) at regular intervals. The samples were taken during the busiest periods of the day (defined as being from 9 am to 12 noon and from 1pm to 4pm on working days). The experiment is described in greater detail below.

System statistics in 4.3 BSD Unix may be obtained using `pstat`. `pstat -x` lists the *text table*, which keeps track of the text segments of programs in use by processes. For each text table entry, `pstat -x` lists the number of processes using the text segment, number of physical pages occupied by the segment, and whether the text segment is currently in physical memory. If more than one process is using a text segment, the text segment is read-only shared between those processes.

Using the above information, we can compute the number of "active" segments (the text segments currently in physical memory) and the number of "shared" segments. The ratio of these quantities gives the extent of sharing in the system. We can also compute the ratio of the physical memory that would have been occupied by the text segments without sharing to that with sharing. This ratio (the *memory savings ratio*) indicates the physical memory savings that this mechanism provides.

Results and Discussion

Table 1 shows the result of this experiment. The experiment was conducted with different sampling rates and for different periods of time. The average extent of sharing over all samples is about 30%. Multics was found to have a much smaller extent of sharing [8]. Only 12% of the segments are in shared in Multics.

The memory savings ratio is found to be about 4. This ratio also indicates how many processes, on an average, share a text segment. For instance, under some assumptions made in Section 5.2, we can show that the memory savings ratio is 1.3 when each "shared" segment is shared between exactly 2 processes. With 3 processes sharing a "shared" segment, the figure rises to 1.6.

The above results show that a significant fraction of text segments are shared in 4.3 BSD Unix. Moreover, each text segment is shared by more than 2 processes, on the average.

Thus, even with a very simple sharing mechanism, the extent of sharing and the benefits thereof are significant.

Extent of sharing	0.298
Memory savings ratio	4.12

Table 1
Sharing in a typical 4.3 BSD system
The extent of sharing is defined as the ratio of the number of shared segments to the number of active segments. The memory savings ratio is defined as the ratio of physical memory without text segment sharing to that with text segment sharing.

5.2. Potential Savings from Sharing

The second experiment was designed to estimate the fraction of library code in different sets of 4.3 BSD Unix programs. The programs in `/bin`, `/usr/ucb`, and `/usr/bin` on a 4.3 BSD system were examined for C library code. The X10 and X11 client application programs were examined for X library code.

For each set of programs, two fractions called the *code fraction* and the *disk space fraction* were computed. The *code fraction* is the ratio of library code to the total code in all programs in the set. The *disk space fraction* is the ratio of library code to the total disk space occupied by all programs in the set. These fractions indicate how much physical memory and disk space savings could be obtained by sharing libraries in 4.3 BSD.

Methodology

The list of all symbols in a library may be obtained from its name list (`nm -n`). The name list of a binary can be scanned to find out which of its symbols is from the library. The scan also yields the total memory occupied by the library code. This can be used to compute the disk space fraction and the code fraction.

Results and Discussion

The results of this experiment are tabulated in Table 2. The largest code fraction is for the set of X11 programs (82%). In fact, some individual programs in this set have code fractions of up to 96%. X10 programs have the smallest code fraction. Commonly used 4.3 BSD Unix system programs in `/bin`, `/usr/ucb` and `/usr/bin` have code fractions between 60-75%.

X11 programs also have the highest disk space fraction (55%). The disk space fraction for 4.3 BSD Unix programs ranges from 39% to 48%. X10 programs would gain least by the introduction of shared libraries. The disks space savings computed above actually give a lower bound. This is because memory occupied by library data is difficult to estimate.

Almost half the disk space used by programs would be saved with shared libraries in 4.3 BSD. The savings in physical memory are equally dramatic as illustrated by the simple calculation below.

Suppose that at any given instant there are n active text segments. 30% of these are shared among processes (using numbers from Section 5.1). Suppose also that each text segment occupies k pages. Hence the total memory occupied when texts are shared is nk. Suppose further that each shared segment is shared by exactly 2 processes. If texts were not shared, the memory occupied would be

$$nk+0.3nk.$$

Assume that 60% of the n text segments is library code. If library code is also shared, the physical memory occupied by the text segments is

$$0.6k+0.4nk.$$

The ratio of physical memory without sharing to that with simple text segment sharing is 1.3. The ratio of physical memory without sharing to that with library sharing is

$$1.3nk/(0.6k+0.4nk).$$

This fraction is about 3.25 for even moderate values of n. Thus, shared libraries can more than double the physical memory savings obtained from simple text sharing.

In the computation, we assume that exactly 2 processes share a segment. This gives a pessimistic estimate of the memory savings with shared libraries.

From the above results, we conclude that shared libraries can provide significantly more savings than sharing text segments as a whole. However, the mechanism for sharing libraries must be efficient. We examine this problem in the next section.

Program set	Code fraction	Disk space fraction	Number of programs
x10progs	0.51	0.27	22
x11progs	0.82	0.55	69
/bin	0.76	0.48	45
/usr/ucb	0.60	0.40	64
/usr/bin	0.60	0.39	87

Table 2

Library code in different sets of programs

The code fraction is the ratio of library code to total code in a program.

The disk space fraction is the ratio of library code to disk space.

5.3. Work Done at Compile Time

Different sharing mechanisms employ different reference resolution schemes (see Section 4.1). These schemes influence the efficiency of code sharing mechanisms. We compared compile time reference resolution in SunOS 4.0, load/run time reference resolution in SunOS 4.0 (with shared libraries) and the fixed virtual address scheme in

DASH. The overhead involved in resolving references can be used as yardsticks for the efficiency of sharing mechanisms.

Methodology

In order to compare resolution of library references, we created a dummy library with 1000 null functions. The members in the DASH library had fixed addresses. For SunOS, both static and dynamic (shared) versions of the library were created.

Our benchmark consisted of a suite of 20 programs which made 50, 100, 150, ..., 1000 library function calls. We computed the time taken to resolve the library references for each benchmark program on the different systems. The experiments were all carried out on a Sun-4. (Although DASH runs on Sun-3s, programs for DASH, at the moment of writing, are compiled on 4.3 BSD Unix machines. We linked the DASH benchmarks on a Sun-4 for ease of comparison).

The times obtained were for a *warm start* (disk accesses were factored out as much as possible). The experiment was repeated a sufficient number of times to give a 95% confidence interval for an error less than 5% [4].

Results and Discussion

Figure 3 shows the time taken to link a program as a function of the number of library functions it references. The time taken to link a program statically is more than the time taken to link the program dynamically. The DASH benchmark was the fastest since all references have fixed addresses and there is no overhead for reference resolution. Clearly, the DASH approach is the most efficient.

The curves are nearly identical upto 200 references because the overhead of reading the object file is greater than the reference resolution overhead till that point. In our case, the dummy library contained null functions. With a real library, the curves would diverge earlier.

It is not unusual to have programs with 1000 library references. Similar numbers of library references are found in some client applications of the X11 window system.

5.4. Work Done at Run Time

Some systems (SunOS, Multics) complete reference resolution during load time and run time. The execution time of a program depends on the reference resolution scheme. It is likely to be higher for systems that resolve references at run-time than for those that do not. In this experiment, we compared the execution times of the benchmarks described in Section 5.3.

Methodology

The methodology used is similar to that in Section 5.3. Each benchmark program was compiled with the SunOS static and dynamic libraries and the DASH library. The resulting binaries were executed on the respective systems; the SunOS binaries on a Sun-4 and the DASH binaries on a Sun-3/50.

As in Section 5.3, the numbers obtained here are for a *warm start*. The experiment was repeated a sufficient number of times to give a 95% confidence interval for an error less than 5%.

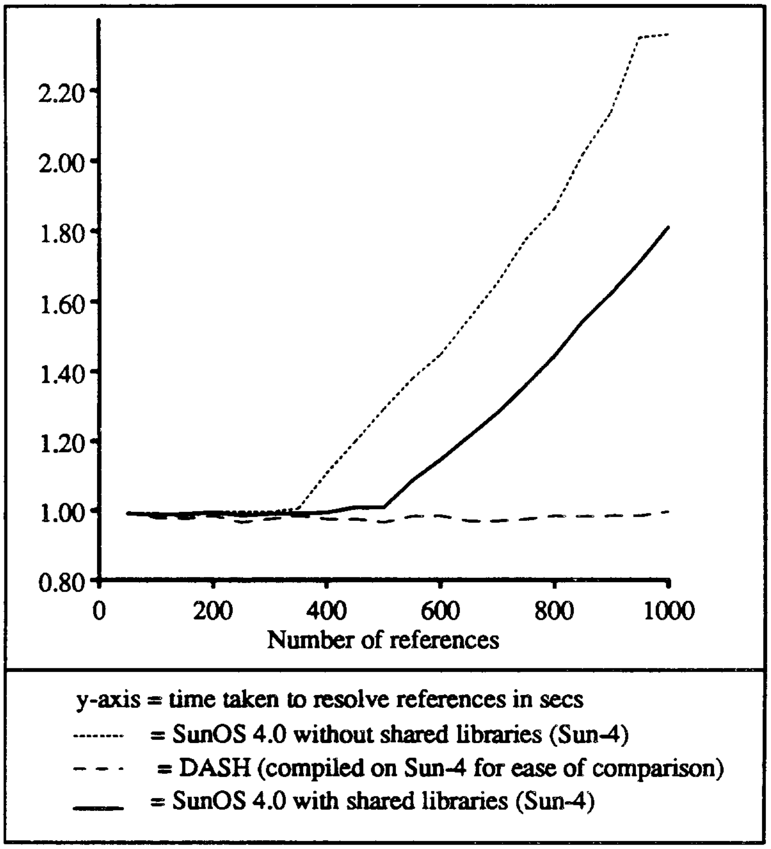

Figure 3
Reference resolution time as a function of number of references
Static reference resolution is costlier than dynamic reference resolution. The DASH approach of fixed addresses for entry points of segments is the most efficient.

Results and Discussion

Figure 4 shows the execution times of the benchmarks as a function of the number of references. Shared libraries on the Sun-4 impose substantial execution overhead. Each functional library reference is resolved at run-time (see Section 6.3). On the other hand, no reference resolution is involved if the library functions are compiled into the program and the runtimes are approximately constant. The time taken to actually execute the functions is negligible since they are all null functions.

In DASH all library references have fixed virtual addresses. Hence the time taken to execute all programs in the benchmark suite is almost constant. Actually, the DASH execution times are comparable to the SunOS static resolution benchmarks. The DASH benchmarks were executed on a Sun 3/50 and the SunOS 4.0 benchmarks on a Sun-4. A simple experiment shows that the two systems have a MIPS ratio of approximately 2.5-3 (the Sun-4 being faster).

Thus, the DASH sharing mechanism is as efficient as the Unix sharing mechanism, while being more general.

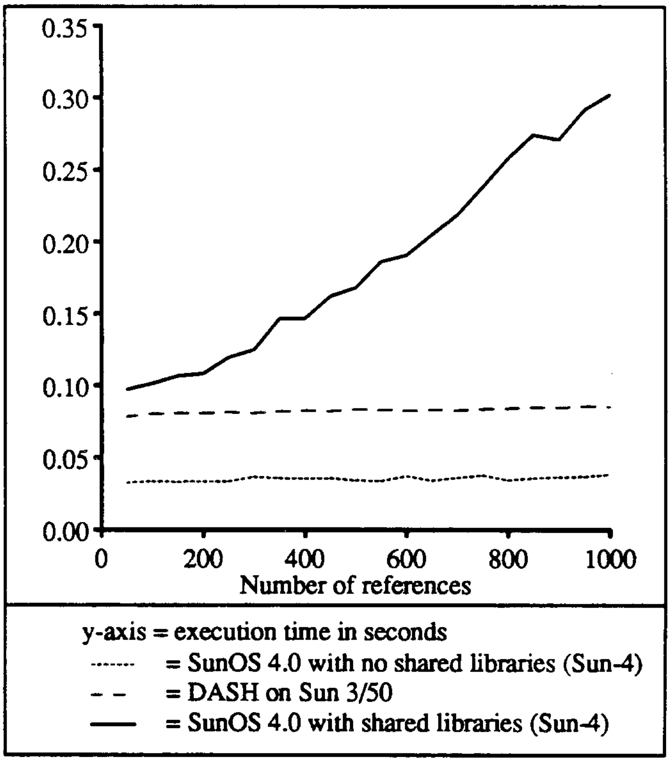

Figure 4

Runtime as a function of the number of references

Dynamic reference resolution causes a dramatic increase in the execution time of programs. The execution times for DASH are actually comparable to those for the statically resolved benchmarks, once the numbers are normalized to account for hardware differences.

5.5. Summary

Our experiments lead us to the following conclusions:

- Sharing is significant in 4.3 BSD Unix, which has a relatively restrictive sharing mechanism (only text segments of programs may be shared).
- Introduction of a more general sharing mechanism (such as shared libraries) hold the promise of greater benefits.
- A more general sharing mechanism need not trade off efficiency for generality. In particular, the approach taken in DASH performs as well as the sharing mechanism in Unix, but is more general.

6. CODE SHARING IN DIFFERENT SYSTEMS

Different operating systems tackle the issues discussed in Section 4 differently. In this section, we tabulate (Table 3, Table 4) the solutions adopted by the systems for each of the issues discussed above. We also discuss, in some detail, the more interesting aspects of some solutions.

6.1. 4.3 BSD Unix

In 4.3 BSD Unix, there is a one-to-one mapping between a process and an address space. The *exec* system call is used to load a segment into a new address space, overwriting the previous contents of the space. The exec call is usually preceded by a *fork* system call which initializes a new address space and starts up a *child* process in the space.

The mechanism for sharing segments in 4.3 BSD Unix is quite simple. Every program in 4.3 BSD Unix consists of three segments - the text segment, the data segment and the stack segment. The virtual memory resources allocated to the text segment of a program are maintained in a *text structure* [7]. When a process *exec*'s a program whose text segment is already in use, the kernel reuses the same physical copy of the text segment for the new process.

6.2. System V Unix

System V Unix uses the same mechanism for read-only sharing as in Section 6.1. Additionally, it has a shared library capability [2].

Shared libraries in System V have fixed addresses determined at the time the libraries are created. Therefore, resolution of inter-segment references can be completed at compile time. The disk image of a segment contains a list of libraries that it recursively references. Library segments are loaded using this list when a segment is loaded.

Only the latest version of libraries is maintained. Any changes to a library while it is in active use by other processes is prevented by the file system.

6.3. SunOS 4.0

In addition to the read-only sharing capability of Unix, SunOS has a shared library capability built into the operating system [5].

Resolution of inter-segment (shared library) references is completed at load time. The disk image of a segment contains a list of library segments referenced. When the main segment is loaded, the system also loads the library segments and resolves all data references. Procedural references are dynamically resolved. This is done by having the procedure linkage table access the dynamic loader when the reference is first made. The linkage table entry is then filled up with the absolute virtual address once the reference has been resolved.

Library segments are loaded at arbitrary locations in the address space of the process. In general, therefore, they may appear at different locations in different address spaces.

The SunOS system supports versioning of libraries. Versions of libraries are designated by two version numbers — a *major* version number and a *minor* version number. Changes in the major version number reflect changes in the interface — versions of libraries with different major version numbers are incompatible. Changes in the minor version number represent compatible changes to the library. Library file names are suffixed by their version numbers.

Library segments are compiled into position independent code (PIC) so that no relocation need be done when they are loaded.

6.4. Multics

In Multics, a process represents a thread of execution and an address space is a collection of segments [9]. The address space is organized as a table of segment descriptors.

Multics segments can be dynamically loaded into the address space of a process — at run time, segments are loaded into the address space on first reference.

Resolution of references can be delayed until run-time. This feature is supported by the *unknown-segment trap*. When a segment refers to another segment which has not been included (loaded) in the address space (i.e. is not "known" to the process), this trap is invoked. The trap handler searches for the referenced segment and "makes it known" to the process. This involves loading the segment into the process' address space by finding an empty descriptor word in the segment descriptor table for the process.

Segments may be included in any available descriptor word, that is, a segment may be assigned arbitrary addresses. In particular, a segment may appear in different addresses in different address spaces. In order for sharing of segments to be possible, a linkage table is associated with each process. All references to segments are made indirectly through an entry in the linkage table. This entry is filled up when the segment is made known to the process.

6.5. Cedar

In Cedar, each machine contains a single virtual address space. All segments are loaded and executed within this single address space [10]. Multiple processes can be initiated within an address space.

Each Cedar module *imports* a number of interfaces and *implements* another interface, part of which it may then *export*. A module is equivalent to a segment (Section 3) and import of a segment is equivalent to an inter-segment reference. Modules in Cedar are shared by different processes within the same address space.

Binding of modules to the modules which they import can be done before loading or when the program is being loaded. Each compiled module has a list of modules whose interfaces it imports. When a module is being loaded, this list is examined. If the referenced module has already been loaded, then the image of the module is used. Otherwise, the required module is mapped into the machine's address space.

Modules are loaded into arbitrary addresses. The compiler for Cedar produces position independent code so that the loader need not perform any static relocation. References to modules which are loaded dynamically are indirected through a linkage table.

It is possible to include modules dynamically in the address space of the machine.

6.6. DASH

A virtual address space in DASH [1] is statically divided into three regions each of which performs one function. The *shared segment region* is a read-only region of the virtual address space and is used for loading read-only code of programs and libraries. The *general* region is used to map the data portion of programs and libraries. The shared-segment region is further subdivided into the *homogeneous* subregion and the *heterogeneous* subregion.

Multiple processes may execute within a virtual address space. Hence it is possible to share a segment among different processes within a virtual address space as well as among different address spaces.

All segments are allocated a fixed range of addresses in either the homogeneous subregion or the heterogeneous subregion. All programs in the homogeneous subregion are allocated disjoint address ranges. This is not true of the heterogeneous subregion. Two programs in the heterogeneous subregion may be assigned overlapping address ranges if they can never be co-resident in memory. Binding of an intersegment reference takes place at compile time since the target addresses are fixed.

Segments can be dynamically loaded into a virtual address space. This process is called *segment inclusion*. When a segment is included in an address space, an *initialization routine* [1] is executed, during which the segment may include any other segments it references. A system call interface is also provided for dynamically loading segments into an address space.

Each virtual address space gets a private copy of the data of a shared segment. Within a virtual address space, all processes share the same copy of the data.

Operating System	Reference Resolution	Naming and Referencing	Version Management	Address Assignment
4.3 BSD Unix	compile time	file system absolute address	none	arbitrary
System V	compile time	file system absolute address, segment list in header	none	fixed
SunOS	load/run time	file system linkage table	kernel-level	arbitrary
Multics	run time	file system linkage table	none	arbitrary
Cedar	load time	file system linkage table	none	arbitrary
DASH	compile time	file service absolute address	user-level	fixed

Table 3
Issues in read-only sharing mechanism design

Operating System	Private data	Dynamic loading	Sharing boundary
4.3 BSD Unix	separate	no	inter-address space
System V	separate	no	inter-address space
SunOS	separate	no	inter-address space
Multics	separate	yes	inter-address space
Cedar	not separate	yes	intra-address space
DASH	separate	yes	intra- and intra-address space

Table 4
Issues in read-only sharing mechanism design (contd.)

7. CONCLUSIONS

A general read-only sharing mechanism holds out the promise of significant physical memory, disk space and I/O reductions. The design of a read-only sharing mechanism need not sacrifice efficiency for generality. The DASH Project has designed an efficient yet general read-only sharing mechanism that does not perform worse than the Unix program code sharing mechanism.

There are many issues in the design of a read-only sharing mechanism. These range from reference resolution to version management. Each of these issues may be tackled in a variety of ways — the approaches range from the extremely flexible to the restrictive. If the solution to each of these issues is carefully selected, it is possible to design a general *and* efficient read-only sharing mechanism.

REFERENCES

1. D. P. Anderson, S. Tzou and G. S. Graham, The DASH Virtual Memory System, *Technical Report, UCB,* Sept 1988.
2. J. Q. Arnold, Shared Libraries on UNIX System V, *USENIX Summer Conference Proceedings, 1986,* Atlanta, GA, 1986, 395-404.
3. R. S. Fabry, Capability-Based Addressing, *Communications of the ACM 17,7* (Oct 1973), 613-625.
4. D. Ferrari, *Computer Systems Performance Evaluation,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

5. R. A. Gingell, M. Lee, X. T. Dang and M. S. Weeks, Shared Libraries in SunOS, *Proc. of the USENIX 1987 Summer Conference*, Phoenix, AZ, June 1987, 131-145.
6. R. A. Gingell, Shared Libraries, *Unix Review 7,8* (Aug 1989), 56-66.
7. S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, Reading, MA, May 1989.
8. W. A. Montgomery, Measurements of sharing in Multics, *Proc. 6th Symposium on Operating Systems Principles 11,5* (Nov 1977), 85-90.
9. E. I. Organick, *The Multics System: An Examination of Its Structure*, The MIT Press, Cambridge, MA, 1972.
10. D. C. Swinehart, P. T. Zellweger and R. B. Hagmann, The Structure of Cedar, *ACM SIGPLAN Notices Notices 20,7* (July 1985), 230-244.
11. W. F. Tichy, Design, Implementation, and Evaluation of a Revision Control System, *Proc. 6th International Conference on Software Engineering*, Tokyo, Sep 1982.