Protected Shared Libraries
A New Approach to Modularity and Sharing

Arindam Banerji, John Michael Tracey, David L. Cohn
Hewlett-Packard Laboratories, IBM T.J. Watson Research Center, University of Notre Dame

Protected Shared Libraries - A New Approach to Modularity and Sharing

Arindam Banerji
Hewlett-Packard Laboratories
axb@hpl.hp.com

John Michael Tracey
IBM T.J. Watson Research Center
jtracey@watson.ibm.com

David L. Cohn
University of Notre Dame
dlc@cse.nd.edu

ABSTRACT

Protected Shared Libraries, or PSLs, are a new type of support for modularity that form a basis for building flexible library-based operating system services. PSLs extend the familiar notion of shared libraries with protected state and data sharing across protection boundaries. Protected state information allows PSLs to be used to implement sensitive operating system services. Sharing of data across protection boundaries yields significant performance benefits. These features make PSLs a viable basis on which a complete operating system can be built largely as a set of dynamically loadable libraries without compromising protection or sacrificing performance. PSLs also allow highly flexible implementations of new functionality to be added to current commercial operating systems. A prototype PSL implementation has been built into AIX 3.2.5 and early performance results are encouraging.

1. Introduction[1]

Software flexibility relies on *modularity*, the ability to modify or replace individual software components easily. Modularity in turn relies on not only the software's internal structure, but also on the degree to which modularity is supported by the underlying operating system and the efficacy of that support. It is not surprising, therefore, that traditional monolithic systems which lack comprehensive support for modularity are characteristically inflexible and difficult to develop and maintain. Production of highly adaptable and manageable systems relies on the development of modularity support which is flexible, efficient, and easy to use.

Attempts to produce modular operating systems have generally followed one of two approaches. The first is to separate an existing operating system kernel into a microkernel that provides a basic set of fundamental constructs and one or more user-level server tasks which run on top of the microkernel and provide operating system services [Black et al. 92] [Rosier at al. 92]. This approach has been applied to a number of commercial operating systems [Batlivala et al. 92] [Borgendale at al. 94] [Golub et al. 90] [Golub et al. 93] [Malan et al. 90] [Phelan et al. 93] [Weicek et al. 93]. The second approach is to design an entirely new operating system emphasizing flexibility using object-oriented technology which generally includes language support. The second approach has primarily been relegated to academic and research environments [Bershad et al. 95] [Campbell et al. 93] [Yokote 92].

Neither of these approaches features adequate flexibility, efficiency, and ease of use. By itself the microkernel approach conveys separation of an operating system kernel along only a single line, the kernel-user boundary. Finer-grained decomposition of both the kernel-level and user-level portions remains an issue. Also, decomposition of the user-level portion into multiple user-level server tasks may be inefficient due to overhead associated with task-based protection [Condict et al. 93] [Ford & Lepreau 94] [Lepreau et al. 93] [Maeda & Bershad 93]. The language based object-oriented approach is generally applicable only to new systems.

An alternate approach to modularity which provides sufficient flexibility, efficiency, and is easily applicable to operating systems is needed. Protected Shared Libraries (PSLs) are just such an approach. PSLs extend the familiar notion of shared libraries by adding support for protected state and allowing data to be shared across protection boundaries. Protected Shared Libraries consist of two separate mechanisms: *Protected Libraries* and *Context Specific Libraries*.

Protected Libraries associate access to specific state information with each library entry point. Entrance to a library routine, which can be enacted only via a defined entry point, conveys access to data associated with the routine; access is revoked when the routine is exited. Similarly, access to a process's data segment is revoked upon entry to a Protected Library routine and restored upon returning from it. Thus, library and client data are protected from each other's code.

Context Specific Libraries, or CSLs, share a single copy of code at a single address as traditional shared libraries do, but offer significantly increased flexibility regarding shared data. They may be seen as a mechanism for encapsulating information that needs to be shared between protection domains. CSLs allow data to

1. David Cohn is currently on sabbatical at the IBM Corporation, Sommers, NY. This work was supported by a grant form the IBM Corporation.

be shared between clients and a service in different ways. A CSL may share a single copy of data between multiple clients, such that all clients see the same data at identical locations. Alternatively, a CSL may share data between a client and a service at a single location such that the actual contents of the shared region is associated with the calling (client) or the called (service) region. Later sections present details of the various forms of sharing supported.

The PSL infrastructure is based on two distinct hypotheses. First, given the benefits of shared libraries, it may be easier to compose user-level system services as sets of cooperating shared libraries rather than as separate processes. Associating protection with shared libraries would allow modular library-based system services to be constructed to replace traditional servers and perhaps even some privileged mode components such as loadable virtual file systems. The ease with which shared libraries can be loaded, unloaded and dynamically relocated provides flexibility not easily attained with cooperating processes. Second, sharing between cooperating entities has to be an intrinsic part of the programming model and not retrofitted through facilities such as `mmap`. CSLs allow programmers to share information in libraries and control the exact nature of sharing. This opens up possibilities for easily creating UNIX u_block [Goodheart & Cox 92] implementations, sharing I/O buffers across protection domains [Khalidi & Nelson 93] and sharing closures and objects across protection boundaries [Banerji et al. 94a].

The remainder of this paper proceeds as follows. The next section presents the motivation for Protected Shared Libraries, both to improve modularity and to facilitate sharing. After that, PSL semantics are described. Implementation issues are discussed in Section 4. Performance results from a prototype PSL implementation are presented in Section 5, and finally, Section 6 presents a brief discussion of the contributions made by Protected Shared Libraries.

2. Design Motivation

Protected Shared Libraries are motivated by two factors. First, passive protection domains, particularly shared libraries, provide an excellent basis for software modularity. Second, efficiency requires cross-domain interactions use shared data. Discussion of these observations continues in the next subsection. Following that, the overall PSL design approach is described.

2.1 Shared Libraries as Protection Domains

Enforced protection boundaries have been found to be a very effective software structuring tool especially for large systems [Nelson 91] [Bogle 94] [Khalidi & Nelson 93] [Chase 94]. Protection can be enforced through a variety of means including separate address spaces [Acetta et al. 86], language support [Nelson 91], and post-processing of binary code [Wahbe 93]. Each of these approaches has been used to increase modularity and security and to facilitate debugging of large software systems. Protection has also been used to ease modification or replacement of software components [Pu 95] [Khalidi & Nelson 93] [Orr 92].

Most work regarding enforcement of protection boundaries in current operating system software, both user-level and kernel-level, has been focussed on improving the efficiency of cross-domain invocations. Invocation times have been significantly reduced by handoff scheduling [Black 89] and thread migration [Bershad 90] [Lepreau et al. 94] [Hamilton & Kougiouris 93]. These protection domains, however, have typically been associated with processes. Counter examples exist [Organick 72] [Scott et al. 90] [Wulf et al. 81], but have generally been limited to research efforts encompassing entirely new operating systems. The only system known to use passive protection domains effectively in a commercial operating system is Mach 4.0 [Lepreau et al. 94], but even that implementation is closely tied to the notion of processes. Our focus on using passive abstractions to represent protection domains is based both on the experience of others [Carter et al. 93] as well as our own [Banerji et al. 94a] that clearly demonstrate the advantages of passive modularity.

Use of passive entities, as opposed to active processes, to represent protection domains has usually taken one of two forms, objects as in Clouds and Psyche or shared libraries as in Multics.[2] A strong case for the support of passive objects as the basic structuring mechanism has been made [Ford 93]. The most important advantages of passive protection domains are their ability to better represent the common case of synchronous communication, their documented ability to support optimized implementations [Druschel 92] [Chase 94] [Carter et al. 93], and the ease with which they can be managed in user-level client code. The last advantage is especially important in making shared libraries a good vehicle for passive protection domains.

2.1.1 Shared Library Limitations

Most commercial operating systems support shared libraries in one form or another, but semantics vary from system to system. On most UNIX systems, library code is shared, but each client process accessing a shared

2. In Multics all object modules were effectively shared libraries.

library gets its own copy of library data. This copy gets mapped into the process' private data segment and is, therefore, equally accessible by client and library code. Some systems such as OS/2 [Deitel & Kogan 92] and Windows [King 94] allow shared or *dynamically linked libraries* (DLLs) to contain shared data as well as code, but offer little or no protection. Each client task accessing a DLL has equal access to the DLL's data. Malicious or errant clients can, therefore, corrupt shared data and adversely impact other clients.

What is desired then is shared library support that features both per-client data as in UNIX, and global data as in OS/2 and Windows, but with protection. Specifically, we desire protection of library data, including per-client data, from client code, and client data from library code.

2.2 Cross-Domain Sharing

Increasingly, system software complexity is being addressed through use of modular protection domains. Cross-domain interactions usually take the form of fast RPC mechanisms that circumvent much of the traditional in-kernel RPC code-path. [Bershad 90] [Hamilton & Kougiouris 93] [Condict et al. 93] Efficiency of fast RPC mechanisms has been improved through use of shared message buffers [Bershad 90]. Some optimized implementations, such as the Fbufs approach [Druschel & Peterson 93], improve throughput by two orders of magnitude. Efficiency concerns, therefore, make a compelling case for sharing. Sharing is also indicated by structural considerations.

Cross-domain sharing has been used to improve structure in various parallel programming models [Scott et al. 90], and to support persistent databases [Bogle 94] [Chase 94] and shared object frameworks [Campbell et al. 93] [Banerji et al. 94]. Sharing enables cooperation between domains with limited trust [Chase 94]. Thus, sharing can be used to support a variety of interactions including producer-consumer, non-intrusive monitoring, asynchronous service providers [Bogle 94], shared pipes, stateless servers with client maintained state, and shared objects exported by servers [IBM 93].

There is a third reason for sharing across protection domains. Most implementations of cross-domain object interactions include a fair amount of overhead for the locally distributed case [IBM 93] [Janssen 95]. Thus, sharing object instances [Banerji et al. 94] and passing enclosures between protection domains on the same machine are usually inefficient. Most of these problems may be solved by judicious sharing of data and addresses between interacting protection domains. Such techniques can drastically reduce the cost of object interaction between protection domains on the same machine.

Capturing this rich yet diverse set of structuring possibilities in an uniform abstraction requires careful design. Adequate programming support is needed so the benefits of sharing may be fully and easily exploited.

2.2.1 Programming Support for Sharing

Two attributes make shared information attractive to programmers.

- First, the ability to share pointer-rich data across domains is attractive. This ability has been found to be useful for persistent stores [Chase 94], shared C++ objects [Banerji et al. 94], distributed shared data and system software. The obvious argument against uniform sharing is the need to reserve portions of an address space. This concern is decreasing with the increasing popularity of large effective address spaces such as the 52-bit global address space and 64-bit non-segmented address space in the POWER [Weiss 94] and Alpha architectures respectively. The advantages in efficiency of avoiding pointer transformations in various applications and system software, as well as programmer convenience are significant.
- Second, the ability to treat shared data through symbolic names that maintain meanings across domains is attractive. A good example of this is the ability to call the shared "libc" version of malloc, uniformly from any process that links in the shared libc library. This facility can easily be extended to shared data, by involving the linker or loader in the manipulation of shared information. This approach can be seen in the shared libraries of systems as diverse as Multics, OS/2 and Hemlock [Garrett et al. 93].

Clearly, with a little system support uniform addressing and naming, can be integrated into relocateable object modules, as has been done with shared libraries in Multics, Hemlock and OS/2. In current implementations, however, the available sharing mechanisms provide limited flexibility.

Context Specific Libraries are motivated by the observation that with a few system extensions, different modes of sharing, along with uniform addressing and naming, can easily be integrated with the common notion of shared libraries. Although other efforts have provided improved forms of sharing, few have been integrated with commercial operating systems.

3. Protected Shared Library Semantics

Protected Shared Libraries extend traditional shared libraries in two ways, with protected state data and cross-domain data sharing. Protected Libraries protect library and client data from each other's code. Context Specific Libraries allow global, client-specific and domain-specific data to be shared across protection domains. The semantics of each of these mechanisms is now described in detail.

3.1 The Mechanisms

Protected Libraries improve modularity and security and facilitate debugging of large software systems by enforcing protection domains between client and library code. Previously, other approaches to protection based on active entities such as processes have been used to improve modularity. Protected libraries investigate the alternative of using dynamically loadable passive shared libraries to enforce protection. This idea is based on efforts such as Psyche [Scott et al. 90] and Multics [Organick 72] both of which supported protected dynamically loadable object modules.

Context Specific Libraries represent modules of code and data that may be shared in various forms between different protection domains. They represent a communication channel between protection domains and thus augment traditional RPC mechanisms. CSLs extend the notion of cross-domain data and address sharing as found in Fbufs [Druschel & Peterson 93], the zero-copy I/O framework and most implementation of the UNIX u-block [Leffler et al. 89]. Together, these mechanisms form a coherent set of structuring and communication abstractions which support construction of system software at user-level.

3.2 Protected Libraries

Protection domain semantics are determined primarily by resource management. Traditional protection domains such as processes act as containers of resources such as memory, threads, file handles, and semaphores. With process-based protection, resource management during control transfer from one domain to another is fairly simple. Resources belonging to the currently running process are accessible. Resource management is more complex with library-based protection because resources are not typically associated with libraries.

Resources may be categorized into memory resources, such as client and service data, and non-memory resources, such as file handles and semaphores. A programmer using protected libraries typically need be concerned only with the handling of memory resources. Handling of non-memory resources is more complicated and usually only relevant to advanced programmers and library authors. Prior to describing resource management issues in detail, we define several terms.

3.2.1 Definitions

Two threads executing concurrently within a protection domain always see the same set of memory resources. A UNIX process constitutes a *primary* or *root protection domain* in which execution is initiated. A Protected Library is viewed as a *secondary protection domain* which is always associated with one or more primary domains. Execution in a secondary domain is always initiated by a thread entering it from another primary or secondary domain.

A thread is the primary unit of execution and can traverse protection domains. As in most multi-threaded process models, each thread owns a small set of per-thread resources such as scheduling and accounting information. These resources, usually encapsulated in a *shuttle* [Hamilton & Kougiouris 93], belong to the thread and remain associated with the thread as it traverses protection domains. Most resources a thread accesses belong to the domain in which it is currently executing. All threads within a domain have equal access to the domain's resources. Threads could conceivably be created in either primary or secondary domains, but thread creation is currently restricted to primary domains.

If Protected Library calls were the only form of cross-domain interaction, thread movement would be restricted to one primary domain and its associated secondary domains. Alternate mechanisms such as thread migration would allow a thread to move between multiple primary domains. The following discussion is simplified by limiting cross-domain control transfers to Protected Library invocations and eliminating consideration of other forms of IPC.

3.2.2 Memory Resources

A Protected Library is an enforced unit of modularity which contains code and data. Any stateless or statefull service that requires protection from either multiple users or from other software components can be implemented in a Protected Library. Protected Libraries can be viewed as regular shared libraries that export protected entry points. Data associated with a Protected Library is accessible only once the library has been entered via a defined entry point. To summarize, a Protected Library is a passive protection domain that can be entered only through defined entry points.

Figure 1 depicts a Protected Library that does not use any Context Specific Libraries. A client process cannot access any of the service data until it calls a defined entry point. A thread starts executing in the client-domain where it can access only the process's own private code and data. This includes code and data defined in the client program and in any ordinary libraries linked with it. Upon calling a defined service entry point, the thread enters the service protection domain. In doing so the thread loses access to its own private code and data and gains access to service code and data.

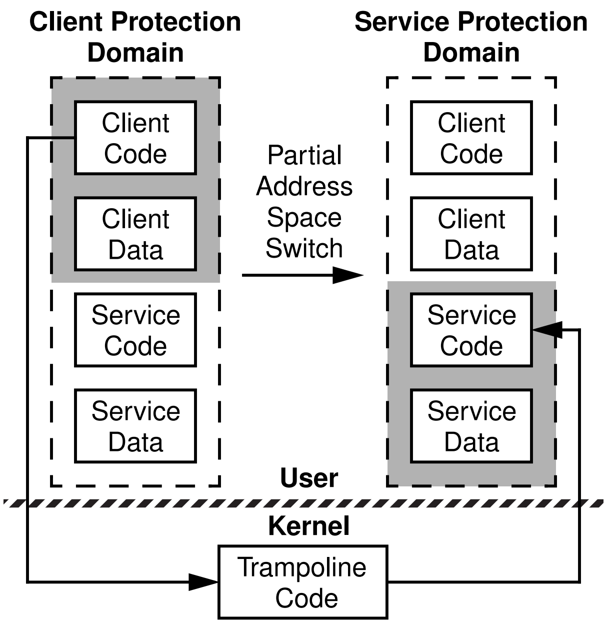


Figure 1 Protected Library

This description illustrates the similarity between Protected Libraries and encapsulated objects. However, Protected Libraries differ from encapsulated objects in three respects. First, Protected Libraries enforce protection boundaries. Admittedly, some object implementations also do this. Second, Protected Libraries allow for direct information sharing between a client and a service, through Context Specific Libraries as described shortly. Finally, because Protected Libraries are actually protection domains, their semantics encompass operating system and user environment resource management.

3.2.3 Non-Memory Resources

Protected Libraries use the “sum of resources” model to specify accessibility to non-memory resources when executing in a secondary or Protected Library domain. A thread’s resource set while executing in a Protected Library is the sum of a set of resources passed from the original primary domain, a set of resources associated with the library domain, and a subset of the per-thread resources associated with a thread no matter what domain it is executing in. The “sum of resources” model only affects Protected Shared Library (secondary) domains and not the root or primary domain which is represented by a regular process context. This follows the Hydra [Wulf et al. 81] model of resource management which allows certain sets of resources to be passed essentially as parameters into a domain along with a call to the domain. The set of resources that must be explicitly passed when entering a secondary domain is part of a Protected Library’s interface specification.

For the most part, programmers building Protected Libraries can safely ignore the notion of passing resources from one domain to another. Default specifications allow this detail to be disregarded, in general, without complications. A programmer might have to choose a particular set of resources to be passed into a domain in the case of exception vectors, such as UNIX signal handlers, for user-level threads. Even in the case of UNIX signals, however, determination of which signals should be handled on a per-domain basis and which must be handled by the primary domain would generally be straightforward.

3.3 Context Specific Libraries

Context Specific Libraries (CSLs) add sharing primitives to the PSL abstraction. CSLs are units of modularity, but not protection, that may be accessible by multiple domains simultaneously. CSLs contain information that must be shared across multiple domains. The CSL is mapped into different protection domains depending on the type of sharing required. CSLs thus extend traditional cross-domain RPC with which information is shared through parameters only by adding communication via shared memory. CSLs are not protection domains but communication channels that may be associated with protection domains to share information. Depending on the type of sharing, as described below, a CSL may be viewed as a fixed piece of information accessible by all domains, mapped information that moves with a thread across domains, or in other ways. A programmer need decide only what information resides in the CSL and what kind of sharing is required. Mapping and sharing of CSLs in multiple domains is handled by the PSL implementation.

3.3.1 Context Specific Library Properties

A CSL is uniformly shared and named in all domains in which it is visible. This implies that symbol names and addresses seen by different domains, are consistent for a given CSL. It does not imply that all domains that use a particular CSL necessarily see the same contents. However, all domains using a particular CSL see the same resolved names at the same addresses; they may or may not see the same contents depending on the type of CSL.

As discussed in the previous section, this approach of sharing information between domains has considerable benefits, such as the ability to exchange pointers between domains and deal with shared data through symbolic names rather than pointers. This advantage results from encapsulating information in shared libraries. Use of shared libraries rather than a facility such as mmap or shared memory IPC implies that shared information is subject to relocation during loading. This allows uniform relocation and address space allocation required for uniform naming and sharing to be implemented relatively easily. CSLs may, therefore, be called and used from PSL based protection domains or the originating root domain with exactly the same effect. From the point of view of the calling protection domain, CSLs look like regular shared libraries with different data sharing characteristics. CSLs appear to execute in the context of the calling domain with one exception.

A CSL module executes in the kernel context of the calling domain. The kernel resources available to CSL code are, therefore, those of the client domain calling the CSL. A CSL also maintains its own user-level context. The reason for this is based on programming experience with CSLs which are frequently used to allocate and manipulate shared data. In such cases, it is extremely useful to depend on a memory allocation or malloc function that is specialized towards the kind of sharing supported by the particular CSL. Thus, a programmer requiring a particular kind of data sharing encapsulates data and its manipulating code in a CSL and uses regular malloc calls. This frees the programmer from explicitly having to deal with multiple versions of malloc.

CSLs support three distinct types of sharing each intended to support a different type of interaction between protection domains. In all three types, each protection domain accessing the CSL views the CSL at the same address. The three types of sharing differ in the contents seen by different domains.

3.3.2 Global Context Specific Libraries

Global CSLs are used to share data and addresses among multiple domains. A Global CSL is depicted in Figure2. Global CSLs feature a single instance of their associated data. This instance is mapped at a single address into the address space of each primary or secondary protection domain accessing the CSL. The global sharing model relies on clients of the shared data using voluntary locking protocols to maintain coherency.

One example use of a Global CSL is for a global memory allocation facility that can be used by multiple processes and PSLs to support globally shared data. In this scenario, both addresses and their associated data must be shared. Nearly the same effect could be achieved by mmaping files between different protection domains. However, the involvement of the linker in the creation of CSLs allows reference to data variables via symbolic names instead of through pointers. Global CSLs thus simplify use of shared data.

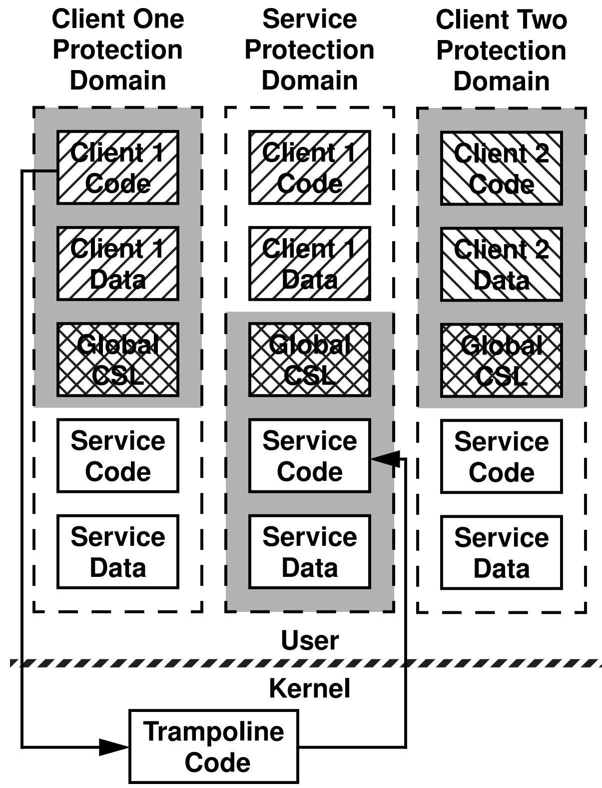


Figure 2 Global Context-Specific Library

3.3.3 Client Context Specific Libraries

Client CSLs facilitate sharing of process-specific information between client and library domains. Shared information, encapsulated in a library, is mapped into each process's address space as shown in Figure3. All processes get their own copy of the data, but the data is located at identical locations in all domains. Upon calling a Protected Library entry point, the CSL data of the current primary (process) domain gets re-mapped into the service domain. Thus, Protected Library code sees the CSL data belonging to the current process. Figure3 shows that when process one calls the service, its CSL data gets mapped into the library domain. With this form of sharing, the client CSL gets mapped and unmapped as a protected call traverses multiple domains. Synchronization mechanisms based on voluntary locks may be used to ensure no two threads of the same parent process simultaneously modify shared data.

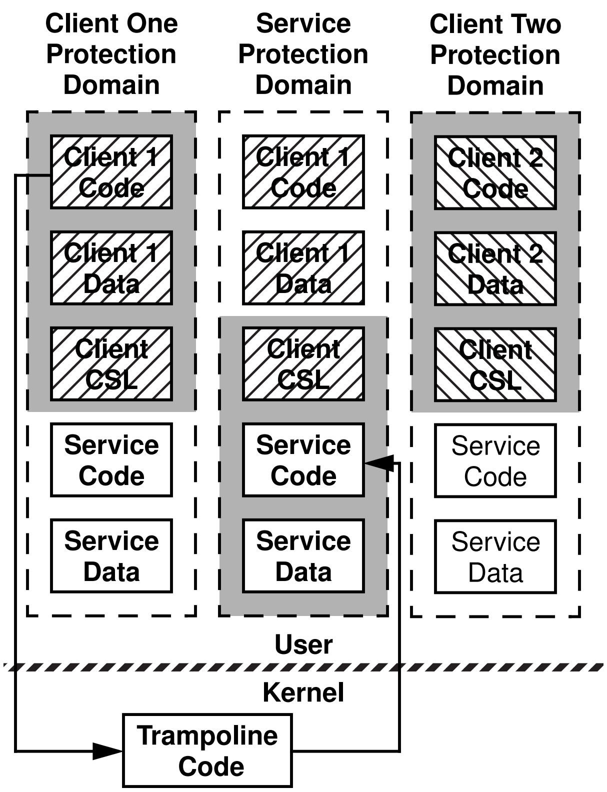


Figure 3 Client Context-Specific Library

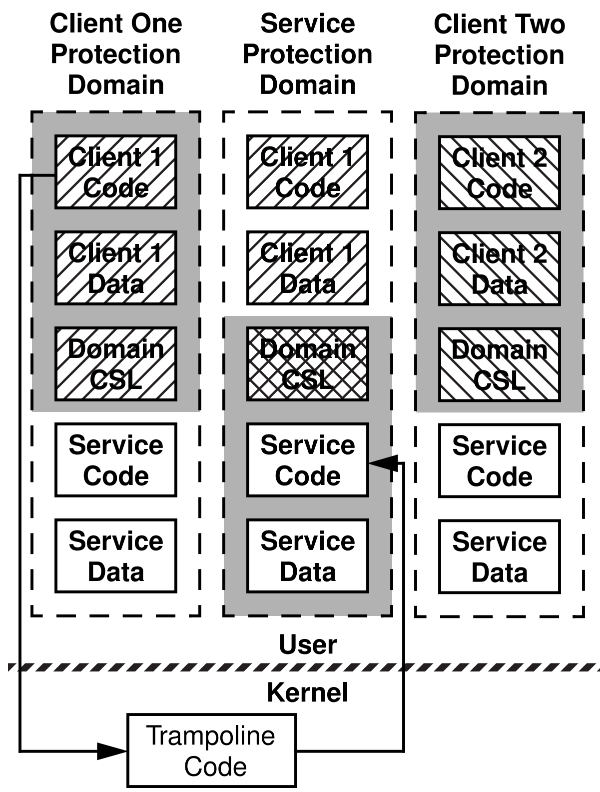


Figure 4 Domain Context-Specific Library

Client CSLs can be used to implement client-specific meta-data. A Protected Library which implements a file system, for example, might maintain process specific meta-data at a certain fixed address. Whenever a thread of a particular process invokes the file system library, the library code automatically refers to the client-specific meta-data. A similar principle is used to implement u-blocks in most commercial UNIX implementations. Per-client meta-data may also be used to maintain per-client method/function tables. This allows client-process specific behavior to be automatically encapsulated within Client CSLs. Any changes to a client-specific behavior is thus easily limited to particular process.

3.3.4 Domain Context Specific Libraries

With a Domain Context Specific Library, depicted in Figure 4, addresses are shared, but data is not. With a Domain CSL, a distinct copy of the library data is maintained for each protection domain. The data is mapped at the same address in each domain. This form of address sharing includes both static and dynamic data. Dynamic allocation of memory in a Domain CSL may be viewed as acquisition of a shared resource, specifically the addresses shared by all clients of the Domain CSL. If a particular domain allocates memory in a Domain CSL, the allocated addresses may not be reused by any other domain. Consequently, in Domain CSLs locking is supported during address allocation but no locks are necessary when data is accessed because every domain has its own data.

One possible use of Domain CSLs is for sharing of C++ objects that contain pointers to virtual function tables (vtbls). In this case, the vtbl must be located at the same address in every domain, but its contents must be unique per domain [Banerji et al. 94]. Thus, the address of a function table can be shared while ensuring the contents of the table are unique per domain.

3.4 Potential Uses of PSLs

Protected Shared LIbraries are a valuable tool for structuring systems. PSLs provide obvious benefits including the efficiency associated with the use of shared data and passive as opposed to active protection domains. PSLs also provide more subtle benefits as described below.

3.4.1 Scope Management

Protected Libraries and CSLs may be used to efficiently implement scope management in systems that use meta-object protocols for extensibility. Meta-object protocol based implementations offer two sets of interfaces; one provides access to normal functionality, the other optionally allows manipulation of the service implementation. The two interfaces associated with meta-object protocols are depicted in Figure 5. A critical issue in meta-object protocol implementations is ensuring that changes made to a service implementation by a client affect only the client making the changes. This issue is referred to as scope management.

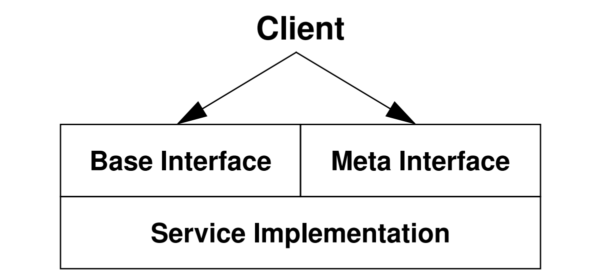


Figure 5 Meta-Object Protocol

Scope management is often implemented by maintaining per-client function or method tables. Creating these per-client method tables in client CSLs as shown in Figure 6 ensures that the service protected library always "sees" the method tables of the current calling root domain. Most UNIX implementations use a similar facility to implement `u_blocks`. However, this requires that `u_block` addresses be hard-coded and thus the functionality is limited to in-kernel co-location of `u_blocks`. Client CSLs eliminate the need for hard-coding addresses and open up the functionality to any protected library client.

3.4.2 Locally Distributed Objects

Distributed object implementations that cross machine boundaries need marshalling/unmarshalling and method table pointer initializations because shared memory facilities often do not extend across machines. Most marshalling/unmarshalling and method table pointer manipulations are unnecessary in distributed object implementations that do not cross machine boundaries. However, most implementations do not use shared memory to implement distributed objects efficiently in the local case [Radia 95]. Client CSLs and domain CSLs can be used to avoid marshalling/unmarshalling or method table pointer initialization int the local case.

Figure 7 shows instance data created in a client CSL with a method table in a domain CSL. The instance data gets mapped into the called domain when a locally distributed object is invoked. Use of domain CSLs

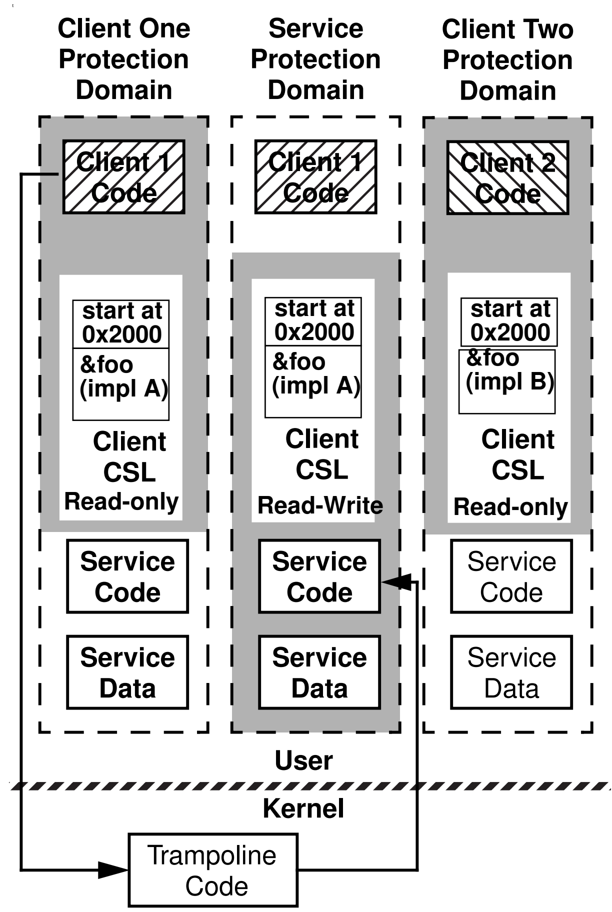


Figure 6 Scope-Management with PSLs

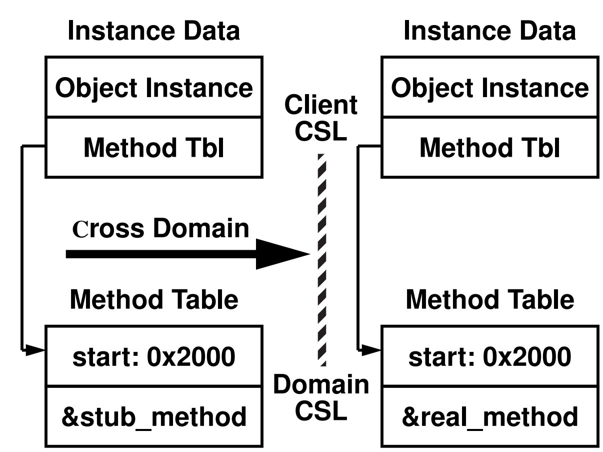


Figure 7 Locally Distributed Objects

ensures the appropriate method tables are found in each domain at identical addresses. As Figure 7 indicates the caller and callee method tables are co-located since they are created in a domain CSL, but their contents are different. In the caller, the method table contains a pointer to a stub-method, whereas in the callee the method table contains a pointer to the actual method. Thus, judicious use of client CSLs and domain CSLs can eliminate extraneous overhead in locally distributed object implementations.

3.4.3 Per-Client Protection Schemes

Use of passive libraries implies per-client bindings thus allowing for flexibility in composing protected library modules. Shared libraries with or without protection require binding of client relocations to service symbols. The nature of these bindings is such that they are always maintained on a per-client basis. Thus, when shared libraries are used as protection domains, the binding between a client and a protected service may be adjusted on a per-client basis. While most processes transfer to some trampoline code to access a service entry point, a trusted client may be linked to have direct access. This allows for highly efficient trusted processes. Similarly, well-tested and debugged clients may be linked to directly access a service, bypassing protection boundaries. Finally, using facilities for dynamic binding of symbols, protection boundaries may be removed and inserted at run-time, if necessary. All these facilities, result from a single design choice, the use of shared libraries or more specifically the use of passive modules subject to relocation.

4. PSL Implementation

One important aspect of the PSL research to date has been the construction of a prototype implementation. The main objectives of the prototype were to clarify the PSL semantics and provide an experimental testbed for a quantitative performance analysis. The prototype was built on AIX 3.2.5, and consists of a modified AIX kernel, C runtime libraries and a new linker. This prototype is only one possible implementation of PSL semantics. Because other operating systems, executable file formats and hardware architectures may imply different implementations, only the main implementation issues are described here.

We first present an overview that draws the connection between different semantic features and aspects of the implementation. The following subsection describes the components themselves in detail. Prior to delving into implementation issues, a brief description of the RS/6000 memory architecture is presented.

4.1 RS/6000 Memory Architecture

A given address space on an RS/6000 is defined by a set of sixteen segment registers, each of which contains a 24-bit segment ID. The RS/6000 uses 32-bit virtual addresses, four bits of which identify a segment which effectively extends the 32-bit virtual address to 52 bits. Of the remaining 28 bits from the original 32-bit virtual address, 16 bits identify a virtual page within the segment, and the remaining 12 bits identify a byte within the page.

By definition, addresses are not generally valid across address spaces. Regions can, however, be shared among multiple address spaces if each space loads a given segment register with the same 24-bit segment ID. As with most architectures, loading of a segment register is a privileged operation on the RS/6000.

4.2 Implementation Overview

PSL semantic features map rather directly to aspects of the PSL implementation. In each of the following paragraphs the relationship between a specific PSL semantic feature and the relevant aspects of the prototype implementation is described. The relationships between semantic features and implementation aspects are summarized in Table 1.

TABLE 1 SEMANTIC FEATURES AND IMPLEMENTATION ASPECTS

PSL Semantics	Implementation Aspects
Shared Libraries as protection domains	Protection implemented via address space manipulation; system loader maps libraries into separate address space regions; passive domains simplify stack management
Partial address space switch on library call	Partial address space switch performed by trampoline code upon entering and exiting PSL routine; PSL linker replaces library calls with traps to trampoline-code
Sharing between protection domains	Shareable address space regions encapsulate shared information; trampoline-code uses architecture specific tricks to share pages between domains
Uniform addressing and naming	Modifications to system linker (and other components) ensure addresses used to map shared data in one domain are reserved in all domains

4.2.1 Shared Libraries as Protection Domains

The PSL implementation divides a process's address space into different regions. PSLs are mapped into these regions depending on the type of sharing and protection desired. Protection is provided by making different regions visible at any given instance. The implementation utilizes various hardware capabilities, such as page tables, segment registers and supervisor calls to control address space visibility. Division of the process-private address space into regions and mapping of libraries into these regions is the responsibility of the system loader.

4.2.2 Address Space Switch on Library Call

A partial address space switch is performed each time a thread enters a PSL via a protected entry point. The switch is performed by a small piece of privileged mode code called the *trampoline* code. The PSL linker ensures that threads trap to the trampoline code when calling a protected library entry point and upon returning from the PSL routine.

4.2.3 Sharing between Protection Domains

Sharing information between protection domains requires that certain address space regions be deemed shareable. CSLs are mapped into these shareable regions which in turn are then made visible to the appropriate domains. As a thread traverses domains, shared address space regions are mapped and unmapped as needed depending on the type of sharing. This mapping and unmapping is performed by the trampoline code.

4.2.4 Uniform Addressing and Naming

In order for shared data to appear at the same address in different domains, addresses which map shared data in one domain must be reserved in all domains. This reservation is ensured primarily by the system loader. However, because address allocation in UNIX is not encapsulated within the loader or any other single component, several other pieces of the kernel had to be modified to ensure reservation.

4.3 Implementation Components

Implementing PSLs on AIX involved modifying the AIX kernel, C run-time libraries and programming tools. We now describe the primary aspects of the PSL implementation.

4.3.1 Address Space Reservation

Uniform sharing requires a portion of each domain's address space be reserved. Unfortunately, in AIX as with most UNIX implementations, address ranges are allocated independently by a number of kernel subsystems. The situation is further complicated in AIX by hard-coded starting addresses of code and data segments. To ensure address space reservation, almost all responsibility for address allocation was extracted from the various kernel subsystems and relocated to the system loader. In certain low-level assembly routines where this was not possible, address allocation logic was modified in place.

4.3.2 Partial Address Space Switches

The overhead associated with PSL protection is largely determined by the efficiency of the partial address space switches performed by the trampoline code. Typically an address space switch involves setting up page tables and flushing caches and translation lookaside buffers (TLBs). This process can be very costly depending on the number of pages involved and the size of the cache. Partial address space switches were implemented quite carefully in the prototype to minimize overhead and maximize performance.

Page table entries that must be switched during a domain transition are preallocated. These entries are maintained in software using a sparse representation technique [Acetta et al. 86], so a large number of pages can be represented using a small number of entries. The trampoline code only switches a couple of pointers to incorporate these new entries into the software maintained page tables. As pages are referenced in the new domain, the hardware page tables are lazily evaluated by updating them from the software tables.

Certain sets of addresses are invalidated during a partial address space switch using architecture-specific techniques. Typically, cache and TLB contents are invalidated during an address space switch to prevent cached data and translations from being used erroneously. To avoid the high cost of such invalidations, the PSL implementation implements a partial address space by changing segment register contents instead of modifying page tables. This prevents user-level threads from generating illegal addresses, while allowing for verify fast switches. Variations of this architecture-specific technique has been used on other architectures as well [Liedtke 95].

4.3.3 Protected Shared Library Linker

The Protected Shared Library linker subsumes the functionality of `/bin/ld`, the normal AIX linker. For binaries that are not and do not use PSLs, the PSL linker simply calls `/bin/ld`. For binaries that are PSLs or use PSLs, the PSL linker has three main responsibilities. First, wherever it detects calls to a protected library entry point, the linker patches in a trap to trampoline code. Second, when creating a PSL, the linker adds descriptive information into unused field of the resulting binary. This information describes the kind of PSL, and type of sharing. It is used by the system loader to map the library into an appropriate address space region. Finally, the linker ensures PSL initialization and termination routines are called as needed. Like many other shared library implementations [Dietel & Kogan 92], PSLs support sub-system and per-client initialization and termination routines.

4.3.4 Protected Shared Library Loader

The PSL loader replaces the AIX 3.2.5 system loader. In short, the loader implements most static aspects of PSL semantics. It creates multiple address space regions within private address spaces, maps libraries to these address space regions and generates per-domain information made available to the trampoline code. The PSL loader differs from the original AIX loader in two main ways. First, data mapping, symbol resolution and relocation were modified to ensure PSL semantics. Second, functionality was added to set up virtual memory data structures for each address space region a library is mapped into.

Typical UNIX loaders map object module data sections into a single address region, the process data segment. In contrast, the PSL loader may map object module data sections into multiple address space regions. The exact address space regions that a data section is loaded into depends on the type of PSL. This mapping and subsequent resolution and relocation creates multiple address space regions; one of which is the traditional data segment, the rest map PSL libraries as shown in Figure 2, Figure 3, and Figure 4.

The PSL loader employs a number of data structures to ensure visibility of address space regions in each protection domain. These structures include both hardware and operating system dependencies and are designed to allow path lengths through the critical trampoline code to be minimized.

4.3.5 Stack Management

Passive protection domains simplify stack management during domain transitions. As in thread migration implementations [Bershad 90], there are two types of stacks. A system maintained activation stack ensures protected library calls can be nested. As a thread enters a new domain, state information for the calling domain is pushed onto the activation stack. Upon returning to the caller, the state information is restored from the activation stack and the stack is popped. The second stack is the execution stack used by almost all run-time environments. Because all secondary protection domains are passive, the execution stack of the calling thread moves with it across domains. Specifically, a thread's execution stack gets unmapped from the calling domain and mapped into the called domain during a protection domain switch. This eliminates the complexity of dynamic stack allocation associated with most thread migration implementations [Bershad 90].

4.3.6 Resource Management

The current PSL prototype transfers almost all resources from the calling to the called domain during a protection domain switch. The only exceptions are signal related resources that are handled on the basis of signal type to allow for error recovery after exceptions. A complete implementation of the PSL resource handling semantics would require significant modifications of the UNIX kernel. This is a reflection on resource handling in UNIX kernels, not on PSL semantics.

Over the last decade or so, kernel code in most UNIX implementations has become quite structured. The vnode [Kleiman 86], HAT layer [Goodheart & Cox 93] and the emerging UDI interfaces [UDI 96] ensure the file-system, low-level virtual memory management services and I/O system are accessed through well-defined interfaces. This provides some degree of encapsulation and isolates clients of these interfaces from implementation changes. Furthermore, indirections such as those postulated by the stackable file systems standard [Heidemann 95] can be easily implemented. This tends to make subsystem implementations more flexible and easier to maintain.

Unfortunately the same cannot be said for process and resource management. Typically this is done through the u_block and proc structures and UNIX kernels are typically littered with direct access to these structures. Such direct access prevents any degree of encapsulation and makes it very difficult to build indirections or change UNIX resource handling. The need for encapsulation of resource handling in UNIX kernels has been previously recognized [Zajcew et al. 93]. Implementation of PSL based protection domains and process migration clearly require better encapsulation of resource handling and standardization in this area is strongly encouraged.

4.3.7 Trampoline Code

The trampoline code performs the partial address space switches required when a client calls and subsequently returns from a protected shared library routine. The trampoline code performs six functions.

- Stack management
- Changing of address space visibility

- Handling of shared address space regions
- Modification of non-memory resource accessibility
- Passing of caller’s resources to called domain
- Transfer of control to target entry point

The trampoline code is responsible for almost all dynamic aspects of PSL semantics. It is by far the most performance critical part of the PSL implementation. Because of this, the code is written in assembly language and pinned in memory at run time. Furthermore, in the prototype AIX implementation, the trampoline code is accessed via a special trap handler that avoids much of the overhead of a typical UNIX system call.

5. Performance

This section sheds light on various aspects of PSL performance. In general, Protected Shared Libraries have been found to perform better than other popular forms of cross-domain cooperation. The next subsection begins with a comparison of null-RPC times. This is followed by a comparison of PSLs with other competitive protection schemes. Finally, the section ends with a breakdown of PSL call costs. A more thorough analysis of PSL performance can be found in [Banerji 96].

5.1 Null-RPC Benchmark

Table 2 shows the null RPC times for several RPC implementations in AIX 3.2.5 on an RS/6000 Model 530 with a relatively slow 25 Mhz POWER processor. The first two numbers are for classic user-level IPC; the third is for hand-off scheduling [Black 89]. The fourth number indicates thread migration is a bit slower than a PSL call due to resource handling overhead. The final two lines indicate the PSL null-call time is comparable to the time required for a null system call.

TABLE 2 NULL RPC TIMES

Implementation	Null RPC Time (ms)
User-level server task - UDP	2000
User-level server task - System V IPC	900
User-level task - Handoff scheduling	190
User-level task - Thread migration	40
Protected shared library	34.5
System call trap	22

5.2 Benchmarks

This section evaluates five different protection

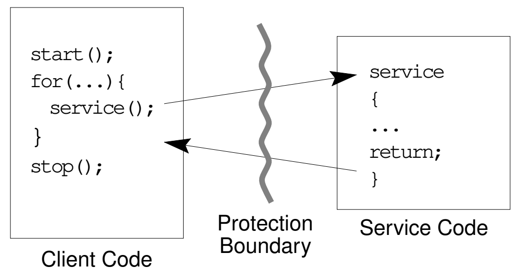


Figure 8 Client/Service Relationship

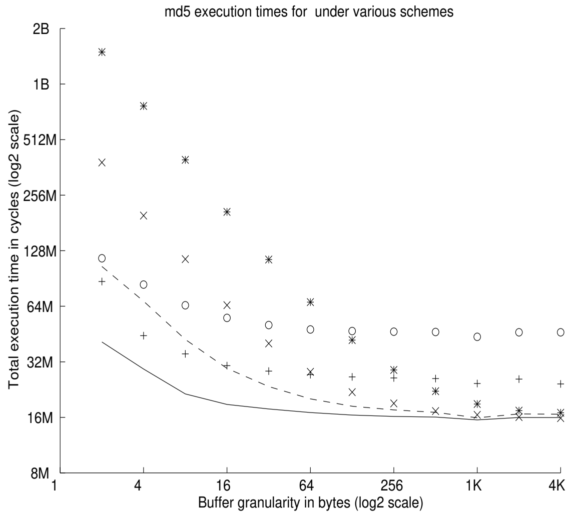
schemes for six different benchmark tests. In each case, the benchmark code is built as a service which is invoked by a client. The goal is to evaluate the cost of protecting the service from the client. Figure 8 shows how each invocation has to cross from client code to service code, and indicates where we start and stop our data gathering. The benchmarks used were:

- MD5, a secure one-way hash function developed to reliably identify long byte strings [Rivest 92]. The implementation used is based on code made available by RSA [RSA 93]. The input byte string is partitioned into fixed-length substrings, and the algorithm operates on the substrings in succession.
- Nsieve, a well-known benchmark that computes prime numbers. Problem size for Nsieve is the total number of primes to calculate; granularity is the number of numbers searched. The iterative portion of the Nsieve code was built as the service, so the number of invocations depends on the density of the primes.
- tdbm_i, tdbm_f, and tdbm_d, three benchmarks involving the tdbm database, a small in-memory database based on the Berkeley UNIX ndbm library. This is a slight modification of the sdbm library released by Ozan Yigit [Yigit 92], and is based on the 1978 dynamic hashing algorithm by Paul Larson [Enbody 88]. Changes were made to avoid unnecessary copying and remove file dependencies. The tests involve insertion of N words from an extended version of /usr/dict/words, random fetch of N/2 words, and deletion of N/2 words.
- Nullc, a custom benchmark that is essentially a null call. The service is passed a block of data. It touches every data page and returns the data to the client. This test measures the base cost of transferring variable-sized parameters between protection

domains.

5.2.1 Protection Schemes

The protection schemes include both classic and new approaches. Three are hardware based and depend upon the kernel protection boundary. The other two are software-based approaches. The protection schemes are null-protection which is used as a baseline, traditional kernel-based system calls, process-based protection with thread migration, library-based PSLs, a language-based safe-subset of Modula 3 and software-fault-isolation [Wahbe 93].

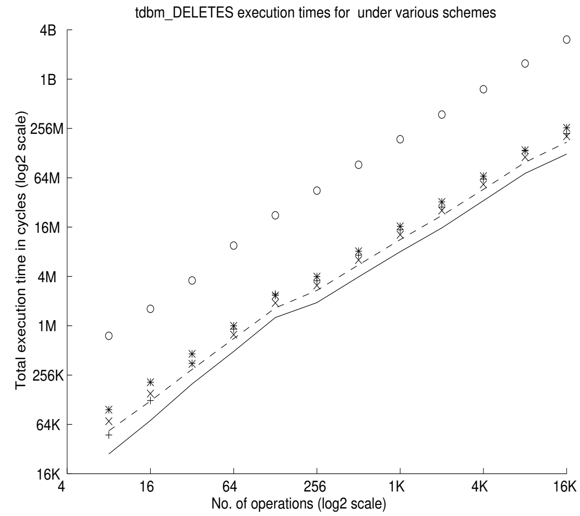


__ No Protection   oo Language-based
xx Library-based   ** Process-based
-- Kernel-based   ++ SFI

Figure 9 Execution Times for MD5

5.2.2 Methodology

Measurements were taken on an IBM RS/6000 Model 390 with a single 66-MHz POWER2 processor. [Weiss 94]. For each benchmark, the granularity of the protection domain was varied, and the number of machine cycles needed to perform the service was recorded. Only plots for md5 and tdbm_d are shown here. For the md5 tests, the total message size was kept constant at 512 KBytes, and the number of bytes passed to the service with each invocation was varied. Figure 9 shows the resulting performance as a function of problem granularity. Increasing granularity increases results in fewer service invocations which causes the execution time of all schemes to decrease. For tdbm, the number of elements in the database was varied. Each invocation deals with one entry, but for larger databases the service does more work. The results for tdbm, shown in Figure 10, differ significantly from the md5 curves in Figure 9.



__ No protection   oo Language-based
xx Library-based   ** Process-based
-- Kernel-based   ++ SFI

Figure 10 Execution times for tdbm_d

5.2.3 Analysis

A few points about the results shown in Figure 9 and Figure 10 bear mention. First, PSLs outperformed thread migration. This is primarily due to cross-domain sharing, simplified stack management and improved resource management. Second, Figure 9 indicates that at higher problem granularities PSLs outperformed kernel based protection. With md5 this happens when the extra kernel trap and return of PSL interactions is outweighed by data copying costs of kernel interactions.[3] Finally, in certain cases the PSL implementation may actually outperform the unprotected case as shown in Figure 9. This is due to the PSL implementation of shared which allows data to remain in the cache between runs of different processes, whereas ordinary shared library data is always faulted into the caches on a per-process basis (at least in UNIX).

5.3 PSL Cost Breakdown

Figure 11 shows a breakdown of the costs associated with PSL-based protection compared to the case of no protection. Most of the overhead is due to *aliasing*,

3. With kernel trap and return times steadily decreasing (11 cycles in UltraSparc), and cache access latencies decreasing slowly, the break-even byte granularity is expected to decrease.

resolution of multiple virtual addresses to the same physical address. The POWER2 architecture provides hardware support to resolve aliases, but the support is not exploited by AIX 3.2.5.

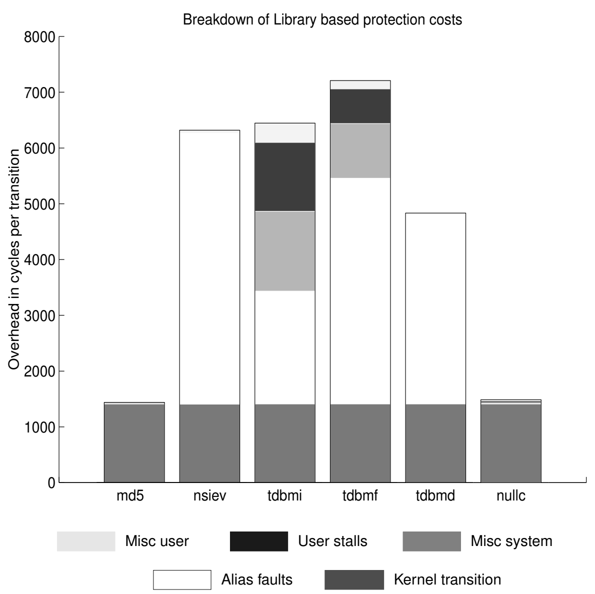


Figure 11 PSL Cost Breakdown

5.3.1 Aliasing Cost

Aliasing arises with hardware-based protection because client and service domains are different virtual address spaces. Consequently, virtual memory data structures must be updated when control is transferred between client and server. Also, references to shared data can cause TLB misses. The resulting *alias faults* dramatically impact transition overhead. These two costs, the in-kernel transition cost of updating aliasing data structures and extra faults due to aliasing, significantly impact PSL performance.

Figure 11 indicates the in-kernel transition cost, which includes adjustments to aliasing data structure, is essentially the same for all benchmarks. For nullc and md5, only one domain actually touches the shared data, so there are no alias faults. For the other tests, however, both client and library code access the data which results in considerable time spent handling aliasing faults. Excluding time for alias faults, for five of the six tests, the PSL overhead lies between 1300 and 2500 cycles.

To assess the cost of updating aliasing data structures, the number of instructions in the transition code was counted for the simplest service, nullc, with a four byte transfer. This code calls AIX routines, written in C, which adjust the virtual memory data structures. Thus, the difference between the transition code instruction count (210) and the total instructions used to make the transition (1250) provides an indication of the cost of aliasing (1040 instructions approx). Hence, if we ignore the costs due to aliasing, which can be safely done for most modern hardware, the cost of a PSL call is actually the cost of executing 210 instructions and the kernel trap/return times. This cost is 275 cycles for the 66 MHz IBM POWER2.

5.3.2 Trap costs

Given that the aliasing problem can be solved in hardware, it is important to look at the other costs. These are approximately 210 instructions per transition, or about 275 cycles including kernel trap and return. This cost approaches those for highly optimized cross-domain transfer mechanisms [Hamilton & Kougiouris 93].

Without aliasing, the hardware cost of the kernel trap and return is significant, 57 cycles for the POWER2. Library-based protection traps twice for every service invocation, thus 114 of the remaining 275 overhead cycles are due to the trap and return. Some modern processors, such as the UltraSparc, have reduced trap overhead to as little as 11 cycles (that is, 22 cycles for PSL-like double trap and returns) and trap costs are expected to continue decreasing.

6. Discussion

Shared libraries form an excellent basis of modularity for structuring large systems. Protected Shared Libraries enhance the popular notion of shared libraries in two ways, by adding protection and allowing data to be shared across protection boundaries. This enables PSLs to be used to securely implement sensitive services. Sharing reduces many of the costs of cross-domain interactions, thus making it a viable alternative to language-based or process-based protection schemes. A prototype PSL implementation has demonstrated the efficiency of the PSL approach.

PSLs are well-suited for use in large user-level applications and for implementation of operating system services at user level as is common in microkernels. There are also less-obvious uses for PSLs. First, dynamically loadable kernel extensions suffer from their ability to corrupt kernel data. Extensions could be prevented from corrupting data to which they do not need write access by building them as dynamically loadable privileged mode PSLs. Thus, PSLs may provide safe ways of extending existing operating system kernels. Second, one important research area in operating systems is the design of low-level nanokernels or exokernels [Engler 95]. These kernels provide low-level protected interfaces to the hardware with most operating system functionality implemented in library routines. PSLs are an attractive approach to protecting such operating systems from application code.

References

[Acetta et al. 86] Acetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for UNIX Development." In *Proceedings of the Summer 1986 USENIX Conference* (Atlanta, GA. July). The USENIX Association, Berkeley, CA, 1986, pp. 93-112.

[Banerji et al. 94] Banerji, A. et al. "Shared Objects and vtbl Placement - Revisited." *Journal of C Language Translation, September 1994, pp. 31-46..*

[Banerji et al. 94a] Banerji, A., Kulkarni, D. and Cohn, D. "A Framework for Building Extensible Class Libraries.", In *Proceedings of the 1994 USENIX C++ Conference*, 1994, pp. 26-41.

[Banerji 96] Banerji, A. et al. "Quantitative Analysis of Protection Options." Technical Report (unnumbered), University of Notre Dame, Notre Dame, IN, 1996.

[Batlivala et al. 92] Batlivala, N., Gleeson, B., Hamrick, J., Lurndal, S., Price, D., Soddy, J. and Abrossimov, V. "Experience with SVR4 Over CHORUS." In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA. April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 223-241.

[Bershad, 90] Bershad, B. "Lightweight Remote Procedure Call." ACM Transactions on Computer Systems, *(1), February, 1990.

[Bershad et al. 95] Bershad, B. N., Saveag, A., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C. and Eggers, S. "Extensibility, Safety and Performance in the *SPIN* Operating System." In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO. Dec. 3-6. ACM Press, NY, 1995, pp. 267-284. (http://www.cs.washington.edu/research/projects/spin)

[Black 89] Black, D. "Scheduling support for Concurrency and Parallelism in the Mach Operating System." Unpublished.

[Black et al. 92] Black, D. et al. "Microkernel Operating System Architecture and Mach." In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA. April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 11-30.

[Bogle 94] Bogle, P., and Liskov, B. "Reducing Cross Domain Call Overhead Using Batched Futures." In *Proceedings of OOPSLA 94*, ACM, 1994.

[Borgendale et al. 94] Borgendale, K., Bramnick, A. and Holland, I. M. "Workplace OS: What is the OS/2 Personality?" March 24, 1994.

[Campbell et al. 93] Campbell, R. et al. "Designing and Implementing Choices: An Object-Oriented System in C++." *Communications of the ACM*, 36(9), 1993, pp. 117-126.

[Carter et al. 93] Carter, J. et al. "FLEX: A Tool for Building Efficient and Flexible Systems." In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (Napa, CA. Oct. 14, 15). IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 198-202.

[Chase 94] Chase, J., et al. "Sharing and Protection in a Single Address Space Operating System." ACM Transactions on Computer Systems, Vol. 12(4), November 1994, pp. 271-307.

[Condict et al. 93] Condict, M., Mitchell, D. and Reynolds, F. "Optimizing Performance of Mach-based Systems By Server Co-Location: A Detailed Design." August 10, 1993.

[Deitel & Kogan 92] Deitel, H. M. and Kogan, M. S. *The Design of OS/*2. New York: Addison Wesley, 1992.

[Druschel 92] P. Druschel et. al. "Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto." In *Proceedings of the 12th International Conf. on Distributed Computing Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 512-520.

[Druschel & Peterson 93] Druschel, P. and Peterson, L. L. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, NC. December 5-8). ACM Press, New York, NY, 1993, pp. 189-202.

[Enbody 88] Enbody, R. and Du, H. "Dynamic Hashing Schemes." *ACM Computing Surveys,* Vol. 20, No. 2, 1988, pp. 85-113.

[Engler 95] Engler D., et al. "Exokernel: An Operating System Architecture for Application-Level Resource Management" In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO. Dec. 3-6), December 1995.

[Ford & Lepreau 94] Ford, B. and Lepreau, J. “Evolving Mach 3.0 to a Migrating Thread Model.” In *Proceedings of the Winter 1994 USENIX Technical Conference* (San Francisco, CA. Jan. 17-21). USENIX Association, Berkeley, CA, 1994, pp. 97-114.

[Garrett et al. 93] Garrett, W. E., et al. “Linking Shared Segments.” In Proceddings of the Winter 1993 USENIX Conference (San Diego, CA, January 25-29), The USENIX Association, Berkeley, CA, 1993, pp. 13-27.

[Golub et al. 90] Golub, D., Dean, R., Florin, A. and Rashid, R. “Unix as an Application Program.” In *Proceedings of the Summer 1990 USENIX Conference* (Anaheim, CA. June 11-15). The USENIX Association, Berkeley, CA, 1990, pp. 87-95.

[Golub et al. 93] Golub, D. B., Manikundalam, R. and Rawson, F. L. III. “MVM - An Environment for Running Multiple Dos, Windows and DPMI Programs on the Microkernel.” In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM. April 19-21). USENIX Association, Berkeley, CA, 1993, pp. 173-190.

[Goodheart & Cox 93] Goodheart, B. and Cox, J. *The Magic Garden Explained*. New York: Prentice Hall, 1993. (ISBN 0-13-098138-9)

[Hamilton & Kougiouris 93] Hamilton, G. and Kougiouris, P. “The Spring Nucleus: A Microkernel for Objects.” In *Proceedings of the Summer 1993 USENIX Conference* (Cincinnati, OH, June). The USENIX Association, Berkeley, CA, 1993.

[Heidemann 95] Heidemann, J. S. *Stackable Design of File Systems*. Ph.D. Dissertation, University of California, Los Angeles, 1995.

[IBM 93] SOMObjects Developer Toolkit User’s Guide, Version 2.0, June 1993, IBM, Austin, TX.

[Janssen 95] Janssen, B., et al., ILU 1.7 Reference Manual, Xerox Corporation, January 1995.

[Khalidi & Nelson 93] Khalidi, Y. A., and Nelson, M. N. “An Implementation of Unix on an Object-Oriented Operating System.” In *Proceedings of the Winter 1993 USENIX Conference*. The USENIX Association, Berkeley, CA, 1993, pp. 469-479.

[King 94] King, A. *Inside Windows 95*. Redmond, WA: Microsoft Press, 1994.

[Kleiman 86] Kleiman S. “Vnodes: An Architecture for Multiple File System Types in Sun UNIX.” In *Proceedings of the Summer 1986 USENIX Conference,* June 1986, pp. 238-247.

[Leffler at al. 89] Leffler, S., McKusick, M. K., Karels, M. J. and Quarterman, J. S. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. New York: Addison-Wesley Publishing Company, 1989. (ISBN 0-201-06196-1)

[Lepreau et al. 93] Lepreau, J. et al. “In_Kernel Servers on Mach 3.0: Implementation and Performance.” In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM. April 19-21). USENIX Association, Berkeley, CA, 1993, pp. 39-55.

[Lepreau et al. 94] Lepreau, J., et. al. “The Flux Operating System Project.” http://www.cs.utah.edu/projects/flexmach.

[Liedtke 95] Liedtke, J. “On µ-Kernel Construction.” In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO. Dec. 3-6). ACM Press, New York, NY, 1995, pp. 237-250.

[Maeda & Bershad 93] Maeda, C. and Bershad, B. N. “Services without Servers.” In Proceedings of the Fourth Workshop on Workstation Operating Systems (Napa, CA. Oct. 14, 15). IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 170-176.

[Malan et al. 90] Malan, G., Rashid, R., Golub, D., and Baron, R. “DOS as a Mach 3.0 Application.” In *Proceedings of the USENIX Mach Workshop* (Burlington, VT. Oct.). The USENIX Association, Berkeley, CA, 1990, pp. 27-40.

[Nelson, 91] Nelson, G., *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[Orr 92] Orr, D. and Mecklenburg, R. W. “OMOS - An Object Server for Program Execution.” In *Proceedings of the International Workshop on Object Oriented Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 200-209.

[Organick 72] Organick E., *The Multics System: An Examination of its Structure*, Cambridge: The MIT Press, 1972.

[Phelan et al. 93] Phelan, J. M., Arendt, J. W., and Ormsby, G. R. “An OS/2 Personality on Mach.” In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM. April 19-21). The USENIX Association, Berkeley, CA, 1993, pp. 191-201.

[Pu 95] Pu, C., et al. “Optimistic Incremental Specialization: Streamlining a Commercial Operating System.” In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO. Dec. 3-6). ACM Press, New York, NY, 1995.

[Radia 95] Radia S., et al, The Spring Object Model, *Proceedings of the Conference on Object Technologies and Systems*, July 1995.

[Rivest 92] Rivest, R. The MD5 Message-Digest Algorithm, *Network Working Group RCF 1321*, 1992.

[RSA 93] http://www.rsa.com/pub/md5.txt

[Rosier et al. 92] Rosier, M., Abrossimov, F., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W. "Overview of the Chorus Distributed Operating System." In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle, WA. April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 39-69.

[Scott et al. 90] Scott, M. L., LeBlanc, T. J., and Marsh, B. D. "Multi-Model Parallel Programming in Psyche" In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming* (Seattle, WA, March 14-16), 1990, pp. 70-78.

[UDI 96] Uniform Driver Interface, ftp://telford.nsa.hp.com/pub/hp_stds/udi/home.html

[Wahbe 93] Wahbe, R., et. al. "Efficient Software-based Fault Isolation." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 203-216.

[Weiss 94] Weiss, S., Smith, J., *POWER and PowerPC*, San Francisco: Morgan Kauffman Publishers, Inc., 1994.

[Wiecek et al. 93] Wiecek, C. A., Kaler, C. G., Fiorelli, S., Davenport, W. C. Jr., and Chen, R. C. "A Model and Prototype of VMS Using the Mach 3.0 Kernel." In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures* (Seattle, WA. April 27, 28). The USENIX Association, Berkeley, CA, 1992, pp. 187-203.

[Wulf et al. 81] Wulf, W. A., Levin, R. and Harbison, S. P. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.

[Yigit 92] Yigit, O. ftp://ftp.x.org/contrib/util/sdbm

[Yokote 92] Yokote, Y. "The Apertos Reflective Operating System: The Concept and its Implementation." In *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA `92*), ACM Press, NY, 1992, pp. 414-434.

[Zajcew et al. 93] Zajcew, R. et al. "An OSF/1 UNIX for Massively Parallel Multicomputers." In *Proceedings of the 1993 Winter USENIX Conference*, The USENIX Association, Berkeley, CA, 1993, pp. 449-468.