Flexible and Efficient Sharing of Protected Abstractions

by

George M. Candea

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

Bachelor of Science in Computer Science

and

Master Of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1998

Author ....
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by ....
M. Frans Kaashoek
Associate Professor of Computer Science and Engineering
Thesis Supervisor

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

Flexible and Efficient Sharing of Protected Abstractions

by

George M. Candea

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science
and
Master Of Engineering in Electrical Engineering and Computer Science

Abstract

Traditional operating systems are overly restrictive and do not allow user-level applications to modify operating system abstractions. The exokernel operating system architecture safely gives untrusted applications efficient control over hardware and software resources by separating management from protection. Decentralized control, however, makes it very difficult for mutually distrustful applications to share system abstractions.

This thesis presents the design, implementation, and evaluation of the *protected abstraction mechanism* (PAM), a novel way to safely share user-level abstractions in an exokernel. PAM enables unprivileged, untrusted applications to define and securely share generic abstractions at run-time. PAM achieves a good flexibility-performance combination by eliminating the need for context switches and optimizing for the common case, in which the same abstraction is invoked repeatedly. PAM's design emphasizes simplicity and provable correctness, which makes it easy to understand and use: a couple of manual pages are sufficient for the average programmer to start using PAM.

We report measurements of PAM's performance on null method calls. In spite of the fact that such invocations do not take advantage of PAM's context switch-free operation, the PAM version of a simple abstraction outperforms the equivalent LRPC implementation by over 15% on null method calls. It is also considerably easier to write an abstraction using PAM. We therefore believe the protected abstraction mechanism is a viable solution to the problem of safely sharing user-level abstractions in the exokernel.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I would like to thank first and foremost my parents — without their vision, guidance, and help, I would not have been here and would not have achieved all that I did. Mom and Dad, this thesis is dedicated to you, for being the most wonderful parents in the world

Many thanks go to Frans and all of PDOS, especially Rusty, JJ, Dave, Professor Greg, Tom, Hector, Mr. Cates, and Dawson for providing me with insight, advice, and many of the ideas that this thesis is based upon. I learned a lot while hanging around these first-class hackers.

Milyn, thank you for making the past four years some of the nicest and most meaningful years of my life. I am very happy fate made our roads cross.

Contents

Chapter 1

Introduction

Numerous researchers have shown that traditional operating systems are rigid and unable to provide applications with the functionality and performance they need [Ousterhout 90; Anderson 92; Bershad et al. 95; Engler et al. 95]. Extensible operating systems address these problems by giving unprivileged applications increased control over the system they are running on. The exokernel [Engler et al. 95] is one such extensible operating system, built around the idea of giving applications protected and efficient control of the system's hardware and software resources.

An exokernel provides solely resource protection, delegating all management decisions to the applications. Operating system abstractions (e.g., pipes) can be implemented by the applications themselves or by libraries (libOS) which are linked into the applications. Programmers are free to rewrite system abstractions to best fit the purposes of their applications. With such increased flexibility, however, sharing these abstractions among mutually distrustful applications in an efficient way becomes very difficult. The goal of my thesis is to provide a mechanism for safe, flexible, and efficient sharing of user-level abstractions in the exokernel.

1.1 Problem Statement

Standard operating systems enforce system abstractions by placing all relevant state in a privileged address space that user-level programs can access only through a fixed system call interface. This centralized model makes it easy to ensure that the shared state remains consistent and unauthorized access does not occur. However, the rigid interface forces applications to use pre-defined abstractions subject to pre-determined policies, which leads to significant performance loss. For example, the replacement policy for disk blocks in a UNIX buffer cache is "least-recently-used" (LRU), which turns out to be inappropriate for a variety of applications. Such applications could greatly benefit from a different policy, but do not have the ability to replace the existing one.

In our most recent exokernel/libOS implementation, called Xok/ExOS [Kaashoek et al. 97; Briceno 97], most operating system abstractions are implemented in libraries and place their state in shared memory. Those abstractions that cannot compromise on protection use protected memory regions [Kaashoek et al. 97]. Unfortunately, this mechanism does not always provide the required performance (as is the case, for instance, in filesystems).

Kaashoek et al. [1997] introduced the idea of *protected methods* to address this performance issue. Using protected methods, applications could safely share high level abstractions by securing their state behind an interface of access methods. Ideally, these access methods would be defined by unprivileged applications to best suit the use of those abstractions.

Prior to this thesis, a true implementation of protected methods did not exist. The idea has been tested with the exokernel implementation of the CFFS filesystem [Ganger and Kaashoek 97], but all the methods were placed in the kernel. CFFS' main abstractions — the inode, superblock, and directory — can only be updated via the `sys_fsupdate_dinode`, `sys_fsupdate_superblock`, and `sys_fsupdate_directory` system calls / protected methods. The exokernel ensures that all updates made to CFFS' metadata preserve the hard-coded invariants implied by the three system calls. An example of such an invariant is the requirement that every entry in a directory have a non-zero inode value.

Unfortunately, an in-kernel implementation is not viable because it lacks flexibility — defining protected methods requires special privileges and the kernel has to be rebuilt. Additionally, debugging in-kernel protected methods is equivalent to debugging the kernel, with all the associated inconveniences. Both these properties are contrary to the exokernel principles [Engler et al. 95].

1.2 Solution Overview

Building upon the idea of protected methods, this thesis presents the *protected abstraction mechanism* (PAM). It is closely related to the concept of data encapsulation, found in object-oriented programming languages. An abstraction takes the form of an object, whose state resides in a protected area that can only be read or written by the abstraction's access/protected methods. However, both the protected abstraction's interface and its state can be defined at run-time, independently of the defining process' privilege level. The mechanism offers a high level of flexibility and eliminates the need for placing performance-critical abstractions in the kernel.

Protected abstraction state is guarded using Xok's hierarchically named capabilities [Kaashoek et al. 97; Mazières and Kaashoek 97]. The state of an abstraction is associated with a capability, which is granted to user applications whenever they invoke a method in that abstraction's interface. Without this capability, processes cannot access the abstraction's state; protected methods have full rights to the state and can perform their task unencumbered. The transfer of control to the protected methods is guaranteed by the kernel, so no other entity in the system besides these methods can be running while owning the state capability.

Some of the advantages of PAM are:

- Applications can safely share arbitrary state among themselves by defining protected abstractions and exporting the corresponding interfaces. Therefore, applications are free to define not only operating system abstractions, but also application-specific abstractions.

- In order to write an abstraction, a programmer needs to read only a couple of manual pages and then use his/her favorite compiler[1].

- Protected abstractions can provide the exact same functionality that in-kernel abstractions would offer.
- Any process running on an exokernel can define abstractions at run-time, without requiring a recompilation of any system components.
- Debugging protected abstractions is equivalent to debugging user-level programs.
- The kernel has the ability to provide the required protection without being aware of the abstraction's invariants, which is the type of approach advocated by the exokernel philosophy [Engler et al. 95].

1.3 Main Contribution

This thesis' contribution is threefold:

- It presents the design and implementation of the protected abstraction mechanism;
- It provides a step-by-step guide to using this mechanism for writing and debugging new abstractions (Appendix A);
- It discusses some of the protection issues and challenges faced by an extensible operating system such as the exokernel.

1.4 Thesis Structure

The rest of the thesis is structured as follows: Chapter 2 presents the design of the protected abstraction mechanism, followed by Chapter 3, which describes PAM's

[1]The current implementation of the protected abstraction mechanism assumes a C calling convention, so we require a compiler that offers the option of using this convention.

implementation. Chapter 4 evaluates the system in terms of performance, scalability, and ease of use. Chapter 5 provides an overview of other projects that are related to the material described here and then Chapter 6 gives a summary of the thesis and concludes. Finally, Appendix A constitutes a brief programmer's manual for using PAM, and Appendix B shows code snippets for the sample abstraction described in Chapter 4.

Chapter 2

Design

This chapter will describe the design of the protected abstraction mechanism. Section 2.1 gives a presentation of the basic design, followed by section 2.2, which discusses PAM's trust model. Section 2.3 explains more detailed issues regarding design correctness. Finally, Section 2.4 describes a conceptual interface to the protected abstraction mechanism along with a high-level design of each operation in the interface.

2.1 Basic Design

The protected abstraction mechanism constitutes a software protection layer placed on top of the exokernel's hardware protection mechanisms (e.g., XN [Kaashoek et al. 97]). The hardware protection layer guards hardware abstractions, whereas protected methods guard software abstractions.

A *protected software abstraction* consists of two components:

- **State** — Memory pages, disk blocks, etc. that the abstraction's definer wishes to safely share with other processes in the system. Safe sharing means sharing under the kernel-provided guarantee that a certain set of invariants will be preserved with respect to this state. The invariants are implicitly defined by the abstraction's methods.

- **Protected methods** — a set of procedures that can query and update the state. These procedures are responsible for maintaining the invariants that define the abstraction's consistency and must provide a complete interface to the protected abstraction.

The main idea behind PAM is to make the state of an abstraction available only to the abstraction's methods. This ensures that the state remains consistent across updates and access rules are strictly observed. In order to make authorization simple and not require the kernel to understand and keep track of abstractions' state, PAM employs Xok's capability system.

The exokernel protects system resources (such as hardware page tables, network devices, disk block extents, physical memory pages, etc.) with hierarchically-named capabilities [Mazières and Kaashoek 97]. These capabilities are composed of *properties* and *names*. Properties define attributes such as "valid", "modify/delete", "allocate", "write", and capability name length. Names, besides identifying the capabilities, also define a dominance relationship: if cap_1's name is a prefix of cap_2's, then cap_1 dominates cap_2. This relationship defines a hierarchy in which the owner of a capability has all rights associated with any of the dominated capabilities.

Each process owns a list of capabilities ("cap list"); every resource is associated with a set of capabilities that grant their owner certain access rights to the resource. A process can request that it be granted additional capabilities and, if authorized, its cap list is augmented correspondingly. The capabilities are used whenever a process wants to bind a resource to a local name[1]: the process presents a capability to the kernel and, if that capability dominates any of the capabilities associated with the resource, then the process is allowed to effect the binding.

In the protected abstraction mechanism, the entire state associated with an abstraction is guarded by one single capability. A process that does not have this capability in its cap list cannot bind a local name to that state, and thus cannot access it. However, when the user of an abstraction invokes a protected method, the

[1]Since all names manipulated by an exokernel are physical names, such as physical page numbers or physical disk blocks, applications must map these names to logical names in order to use them.

kernel grants that process the state's capability and transfers control to the requested method. The method runs in the caller's context and can update the state as needed. Prior to returning, the protected method destroys the capability and unbinds (unmaps) the abstraction's state. This ensures that only protected methods have the capability to access the abstraction state.

This design appears to be similar to lightweight remote procedure calls (LRPC) [Bershad et al. 90]. However, LRPC invocations are made across peer domains. In the protected abstraction mechanism, invocations are within the same domain; the caller domain's privilege with respect to the accessed state is temporarily elevated, after which it drops back to the initial privilege level. Protected methods can be viewed as "user-level system calls"[2], or code that one application can download into another one's address space with the help of the kernel.

2.2 Trust Model

The protected abstraction mechanism is based on the last of the three popular trust models shown below. In this context, we identify two types of principals: the process that *defines* an abstraction (D), and the processes that *use* the abstraction ($U_1, U_2, ...$). The three trust models are:

1. When everyone mostly trusts everyone else, both D and U_i trust that U_j makes well-formed updates to the abstraction's state. However, U_i does not trust D to have access to U_i's address space. Any abstraction that places its state in shared memory, and allows direct access to it, implements this type of trust.

2. When D is the only trusted principal, it may be trusted solely with respect to maintaining the abstraction's invariants. In this case, D does not trust U_i to update abstraction state, but will perform an update on U_i's behalf, if that update preserves the invariants. User-level servers in microkernel operating

[2] Clearly the notion of *system* call looses its meaning if it is implemented at user level, but this term is convenient for explaining the analogy.

systems adopt this trust model, and abstraction users communicate with the servers using inter-process communication (IPC) mechanisms.

3. In PAM's trust model, U_i is not trusted at all, but D is both trusted to maintain the abstraction invariants and to access U_i's address space. This corresponds to moving part of the kernel (D) into user space, in the form of a protected abstraction. This model does not make the unsafe assumption that all users of an abstraction maintain the required invariants, yet it does not incur the performance penalties imposed by the use of IPC for every operation. Programmers understand this model and can easily program to it.

It is possible that the trust model assumed by PAM is not conservative enough for certain classes of applications. Such demands could be easily satisfied by "trusted abstractions," which are defined by programs owned by privileged users (e.g., `root` on UNIX systems). The abstractions defined by such applications would be trusted by all processes running in the system. Of course, unprivileged programs would be unable to modify these abstractions, but privileged users can easily change them, without having to rebuild the kernel.

2.3 Correctness Considerations

In order to ensure correctness of the protected abstraction mechanism, three distinct properties must be preserved: execution atomicity, state integrity, and controlled resource allocation.

2.3.1 The Issues

The execution of a protected method must be atomic with respect to the caller process, i.e. PAM must guarantee *execution atomicity.*

During the execution of a protected method, the calling process' privilege with respect to the abstraction's state is temporarily raised; control must continuously remain within that method up to the point where it retires the process' privilege level to the initial level. However, application specific handlers [Kaashoek et al. 97; Wallach et al. 96], one of the exokernel's flexibility features, make this hard to enforce. A process can register handler procedures with the exokernel and have them invoked whenever the corresponding interrupt, fault, or exception[3] occurs in the owning process. For example, page fault handlers can be provided by applications to tailor paging policy to their own needs. After handlers complete, program execution resumes at the point of interruption.

This type of architecture poses a great risk for PAM: if an interrupt occurs during the execution of a protected method, control would automatically be transferred to one of the caller's interrupt handlers. At that point, all the abstraction's state is accessible to the caller application, which could (maliciously or inadvertently) alter it. It is therefore imperative that PAM either block this type of events or enable the protected abstraction to handle such exceptions itself.

Due to Xok's other flexibility features, abstraction *state integrity* becomes an important concern. For example, user processes have the ability to set up direct memory accesses (DMA) from I/O devices. This provides applications with an asynchronous way of overwriting memory pages they own, including the ones containing their stack. A process could set up a DMA to a stack page, invoke a protected method, and hope the DMA will complete while the method is still executing. If the protected method ran on the same stack as its caller, it may end up performing unwanted actions due to a clobbered stack. Applications would thus have a simple way of corrupting the abstraction's state[4].

Given that Xok/PAM performs certain actions for the protected abstractions and handles their state, it is important to not do anything on an abstraction's behalf that may be inappropriate for that abstraction's privilege level. One particularly important issue is that of *controlled resource allocation*: PAM should never allocate or keep allocated resources on an abstraction's behalf unless the process defining or

[3]The notions of fault, exception, and interrupt will be used interchangeably, although they are not the same. The distinction, though, is not relevant in the context of this thesis.

[4]The protected methods' stack is part of the abstraction's state, because it holds spilled registers and the protected method's local data structures.

using the abstraction has sufficient privilege.

2.3.2 The Solutions

As mentioned above, there are essentially two ways to ensure *execution atomicity*: by blocking the events that may interrupt execution or by giving the abstraction control over the handling of these events.

Giving control to the abstraction in the event of an interrupt is a more flexible solution and has a lesser impact on the rest of the system. In PAM, whenever a process defines an abstraction, it also instructs the kernel to use a different set of fault handlers whenever a protected method is running. These handlers are provided by the abstraction definer. Abstraction-specific fault handlers ensure that execution atomicity can be reliably supervised by the abstraction and control will not accidentally slip to an unauthorized entity.

State integrity can easily be guaranteed through isolation: in PAM, the state of an abstraction is completely disjoint from the space controlled by the caller of a method. Each abstraction has its own separate stack guarded by the same capability that guards the rest of its state. Aside from protection, this design decision has also another benefit, especially during the development phase of the abstractions: using separate stacks provides fault isolation, so accidental stack clobbering done by either caller or protected method is easier to detect and correct.

PAM provides *controlled resource allocation* by ensuring that any resource used by the kernel on an abstraction's behalf is "charged" to either the abstraction or the abstraction's client. The kernel never takes ownership of any of these resources. Two of the more significant examples are state and CPU time allocation.

The protected abstraction mechanism employs a very simple policy: whatever state is associated with an abstraction belongs to the definer process. Consequently, the abstraction can keep its state and be available to the rest of the system only during the lifetime of the definer. This avoids any resource leakage that may occur from the kernel taking ownership of state components and keeping them allocated even after the definer has exited.

One benefit of having protected methods run in user mode is that they are easy to preempt, which avoids many of the CPU utilization problems that arise in other systems. One specific example (discussed extensively by Wahbe et al. [1993] and Deutsch and Grant [1971]) is the need for limiting the execution time of kernel-downloaded code, given that it runs at the highest privilege level. In PAM, the task of dealing with an epilogue (imminent end of time slice) event is completely delegated to the application/abstraction level.

These decisions closely follow the exokernel philosophy of exposing all aspects of resource management to user applications. Specifically, state (memory pages, disk blocks, etc.) management and scheduling events are entirely handled by the protected abstractions.

2.4 Interface

PAM's interface for managing protected abstractions consists of three basic operations: two to be used by definers (`prot_define` and `prot_remove`), and one for the client applications (`prot_call`). This section presents only a conceptual interface; for performance and implementation-specific reasons, the actual interface is slightly different. Chapter 3 describes the implemented interface and Appendix A gives detailed instructions on how to use it.

```
prot_define (  method_entry_addrs, capability,
               metadata_addr, handler_entry_addrs )
```

`prot_define` is used to define an abstraction. It returns an ID that uniquely identifies the just-defined abstraction. The arguments are:

- *method_entry_addrs* provides the address of a data structure containing the entry points (as virtual addresses) of this abstraction's protected methods;
- *capability* indicates which capability protects the shared state;

- *metadata_addr* is the virtual address of the metadata page (see explanation below);
- *handler_entry_addrs* is a structure containing the entry points of the abstraction's fault handlers.

To register an abstraction, the definer process will typically do the following:

1. Allocate the shared state (memory pages, disk blocks, etc.) and protect all of it with a system-wide unique capability. This is done using existing exokernel system calls.
2. Load the code for the protected methods and obtain every exported method's entry point.
3. Load the code for the fault handlers and obtain their addresses.
4. Perform `prot_define` to register the abstraction.

It is easy to see why, in order to register an abstraction, the definer must provide the kernel with the entry points for the abstraction's protected methods, the capability that guards the shared state, and the abstraction-specific handlers.

For bootstrapping purposes, one more piece of data is necessary. Whenever a protected method is invoked, it has no knowledge of where the abstraction's state resides, given the total separation between data and code. Thus, the definer allocates one memory page (called the *metadata*[5] page) and passes it to the kernel. This memory page will then be mapped by the kernel for every protected method, and the method will be able to use the shared information in this memory page in order to locate the rest of the state. The metadata page is very similar to the root inode in a file system: it is a well-known starting point, from where the entire state can be reached.

[5]It is called a metadata page because abstractions will usually keep their administrative data structures in this memory page.

From the kernel's point of view, the four items described above completely define an abstraction. The exokernel needs no knowledge of what the state is composed of, because it is under the complete management of the abstraction code. At this point, the kernel's task consists solely of updating its internal data structures that keep track of currently defined abstractions, and return to the definer an ID that uniquely identifies the abstraction.

There are three issues that deserve consideration here: partitioning the abstraction state, naming, and persistent abstractions.

It may seem that offering the option of protecting sections of the shared state with different capabilities would add flexibility. However, implementing such partitioning of the state is completely up to the abstraction code. One possible implementation would be to use individual capabilities for each partition and have them all be dominated by the abstraction state capability.

A second issue has to do with naming, which is an exokernel-wide problem. In PAM, protected abstractions and methods are specified by numbers, which can lead to cumbersome programming. However, resolving this issue is not up to the in-kernel protected abstraction mechanism. Possible solutions include abstraction definers that can be asked for name to ID mappings and vice versa, using standard IPC mechanisms, or a global name server that keeps track of the mappings between all names and IDs.

One last point is that some programmers may feel the need to define abstractions which persist across reboots. The preliminary design included a provision for such persistent abstractions: PAM was maintaining an on-disk registry of abstractions, similar to the one used by XN [Kaashoek et al. 97]. However, this feature turned out to complicate PAM unnecessarily. If there are abstractions that need persistence, it is easy to modify startup scripts such that the corresponding definer processes are started upon boot-up.

Additionally, persistent abstractions would provide a way for unaccounted resource consumption (the resource being registry space), which was shown in Section 2.3.1 to be undesirable. Moreover, such a registry would require additional resource management, which would need to be performed by the kernel. Such a design would go against the exokernel principle of delegating all management to untrusted user-level code.

```
prot_call ( abstraction_id, method_num, arguments_addr )
```

Whenever a process wants to invoke a protected method, it uses the `prot_call` operation, which returns the result of executing that particular protected method. The arguments are:

- *abstraction_id* specifies which abstraction the desired method belongs to;
- *method_num* indicates which of the abstraction's methods is to be invoked (this is simply a number that indicates 1^{st}, 2^{nd}, ... method);
- *arguments_addr* provides the virtual address at which the protected method can find its arguments.

Given that the abstractions interface is dynamic (i.e., new abstractions get defined and others get removed at run-time), it makes sense to use a dispatcher-type operation, such as `prot_call`, to invoke these methods.

The invocation of a regular procedure consists of setting up the callee's stack with the arguments and transferring control to the procedure's entry point. Analogously, invoking a protected method consists of giving the kernel a pointer to the arguments for the method, along with an identifier for that method. The kernel then performs the following steps:

1. Grant capability to the caller;
2. Install the protected abstraction's fault handlers;
3. Bind the metadata memory page to an address known to the called method;
4. Set up the stacks and switch to the protected method's stack;

5. Transfer control to the method's entry point.

Protected methods run completely in their callers' address spaces. Once control has been transferred to the protected method, it can map any additional state that it needs, because the "host" process holds the necessary capability. After that it can perform its task.

Once the protected method has completed, it simply returns directly to its caller. In order to ensure that abstraction state remains protected, the method must unmap any sensitive state (which will usually be the entire abstraction state), and destroy its copy of the state capability. Next, it restores the caller's fault handlers and switches to the caller's stack. These actions are easily done using existing Xok system calls. However, measurements indicate that performing individual system calls for each action is expensive, so the actual implementation optimizes by batching these system calls together, as shown in Chapter 3.

It becomes clear that defining and using protected abstractions consists of virtually downloading the code of the protected methods into caller processes' address spaces and invoking the kernel to grant special privileges to the downloaded code whenever necessary. This also imposes an added restriction on the execution of the protected methods: in the event of an epilogue (notification of the imminent end of the process' time slice), the protected method needs to either bring itself to a consistent state, revoke its privileges, and hand off the epilogue event to its calling process, or completely handle the epilogue itself, so it can achieve secure re-entrancy.

```
prot_remove ( abstraction_id, capability_slot )
```

This operation removes a currently defined abstraction by deleting it from the kernel's data structures and returns a status code indicating whether the operation succeeded or not. The arguments are:

- *abstraction_id* specifies which abstraction is to be removed (should be the value returned by the `prot_define` operation that registered the abstraction);
- *capability_slot* denotes the slot in the caller's cap list where the abstraction state's capability can be found. The capability needs to be identical to the one passed to the corresponding `prot_define`.

Chapter 3

Implementation

This chapter presents the implementation of PAM, following the design shown in Chapter 2. Each of the three operations forming the conceptual interface introduced in Section 2.4 naturally correspond to a system call in the actual interface. After a discussion of PAM's data structure, each system call's implementation will be described in detail. We will also introduce a fourth system call, for performance reasons. The chapter will conclude with an analysis of the implementation.

3.1 Major Data Structures

PAM uses a fixed area for protected methods, both to improve performance and for simplicity. By convention, protected method code lies between addresses `PROT_AREA_BOT` and `PROT_AREA_TOP`, which are currently defined as 0x8000000 and 0x10000000, respectively.This represents 128 Megabytes, which should be sufficient for most protected abstractions. This particular area was chosen to be half way between the bottom of the address space (0x0) and the beginning of the area where the shared ExOS library is mapped; it leaves 96 Megabytes for the application.

The implementation of PAM necessitated the addition of new fields to a process' environment. First, the user-accessible part of the environment now has one entry for each user-vectored handler; this is where the kernel saves the old entry points of the caller's handlers, prior to installing the protected abstraction's handlers. Second, the part of the environment that is only accessible to the kernel has one new field which indicates whether the process is currently inside a protected method, and, if yes, which abstraction that method belongs to. This value is used by PAM to locate the abstraction whose handlers need to be restored. It also has potential future uses if the kernel will need to know whether a process is executing a protected method or not.

PAM maintains one abstraction table in which it keeps track of the currently defined abstractions. As mentioned in section 2.4, an abstraction is completely defined by four elements: the methods' code, the capability that guards the abstraction's state, the metadata page used to "bootstrap" protected methods, and the abstraction-defined exception handlers.

As expected, the abstraction table contains these elements; Table 3.1 shows the structure of each entry in this table, using C-like types.

Field Type	**Field Name**
unsigned int	num_methods
unsigned int	entry_points [MAX_NUM_METHODS]
unsigned int	num_code_pages
PTE	PTEs [MAX_NUM_PAGES]
cap	state_capability
PTE	metadata_page_PTE
unsigned int	metadata_virtual_address
handlers_t	handler_entry_points

Table 3.1: Structure of an entry in the kernel's abstraction table.

- For the kernel to be able to manage the methods' code, it must keep track of two items:

    1. The entry point for each method: this information is kept in an array of virtual addresses (*entry_points*), containing *num_methods* elements, up to a maximum of *MAX_NUM_METHODS*. Note that virtual addresses are the size of a machine word (`unsigned int`).

    2. The physical pages containing the code for the methods, as indicated by the corresponding page table entries (PTE): this component consists of

the array *PTEs*, with *num_code_pages* elements, up to a maximum of *MAX_NUM_PAGES*. Note that managing this table becomes somewhat complicated once the exokernel starts supporting page swapping to disk, because PTEs in the table will need to be updated in concert with the swapping code.

- *state_capability* is the capability that guards the state of the abstraction.

- To keep track of the metadata memory page, the kernel needs its PTE (*metadata_page_PTE*) as well as the virtual address at which to map it in the caller's address space (*metadata_virtual_address*). Protected methods expect to find their metadata at *metadata_virtual_address*.

- Finally, the kernel also needs to know the entry points for the various exception handlers (*handler_entry_points*). They are stored in a `handlers_t` structure, defined as an aggregation of virtual addresses, one for each type of handler (epilogue, prologue, page fault, IPCs, and clock).

It may seem that a table structure is inefficient, especially when it comes to removing abstractions. However, this structure offers fast lookups in the common, performance critical case: protected method invocation. Obviously, a fixed size structure such as a table restricts the number of simultaneous abstractions and introduces management complications. In addition, there is space that is allocated yet not used all the time. These disadvantages, however, are minor and constitute a good tradeoff for better performance, especially when the number of abstractions is high.

3.2 sys_prot_define

```
int sys_prot_define (  unsigned int method_tab_addr,
                       unsigned int capability_slot,
                       unsigned int code_size,
                       unsigned int metadata_addr,
                       unsigned int handler_entry_addrs )
```

The system call's arguments are:

method_tab_addr This is the address of a structure that contains the number of methods in the abstraction and an array of entry points (virtual addresses in the caller's address space). There is one entry point for each exported protected method in this abstraction's interface.

capability_slot The index of the slot in the caller's capability list (stored in the process' environment) which contains the capability meant to protect the abstraction's state.

code_size The number of bytes that this abstraction's code occupies. This argument helps the kernel infer which pages of code to map when a method is invoked. *code_size* comprises the total size of the abstraction's protected methods and must include also non-exported procedures. It is assumed that the methods' code is contiguous in memory.

metadata_addr This is the virtual address at which the protected methods will expect the metadata to be mapped. For simplicity, PAM allocates the metadata page in the kernel as opposed to expecting the caller to do it. No loss of flexibility results from this, given that all abstractions will always need to use at least one page of memory for their internal data structures.

handler_entry_addrs The address of a structure containing an entry point for each user-vectored interrupt handler. Specifically, these handlers correspond to `entepilogue`, `entprologue`, `entfault`, `entipc1`, `entipc2`, and `entrtc`.

The kernel implements `sys_prot_call` with the following steps:

1. Validate the arguments. The requirements for the system call to not fail at this point are the following:

   (a) There is sufficient space in the abstraction table for a new abstraction;

   (b) The capability in *capability_slot* is valid;

   (c) The number of methods in the abstraction are within admissible limits;

   (d) All the addresses passed in are valid user addresses and the abstraction is fully contained between `PROT_AREA_BOT` and `PROT_AREA_TOP`.

2. Find a free entry in the abstraction table.

3. Make a copy of the capability indicated by the caller and save it in the abstraction table. Also fill in the other required information: number of methods, method entry points, fault handlers entry points.

4. Allocate the metadata page on the caller's behalf. Place the corresponding PTE in the abstraction table. Note that the design required the abstraction definer

to allocate the metadata page and pass it to the kernel. However, it is easier for the programmer to not have to be concerned with this aspect.

5. Deduce which (and how many) pages contain the code of this abstraction. Update the abstraction table correspondingly.

If all these steps are successful, the system call returns the index of this abstraction in the abstraction table. Otherwise, an error status code is returned.

3.3 sys_prot_call

```
int sys_prot_call (  unsigned int invocation_id,
                     unsigned int argument_addr )
```

The meaning of the arguments is as follows:

invocation_id This number is a 32-bit quantity, with the following fields:

- The highest bit is a flag instructing the kernel whether to map the methods' code pages or not (see explanation below);

- The next 15 bits contain the abstraction identifier and the low 16 bits contain the method index. These two fields uniquely identify the method that is to be invoked. Such an encoding limits the number of simultaneous abstractions to 2^{15}, each with a maximum of 2^{16} methods. It is hard to imagine a situation in which these numbers would constitute any sort of limitation.

The decision to pass in the flag and method identifier encoded this way was primarily for performance: in Xok, if a system call has two[1] arguments or fewer, they are passed in registers instead of on the stack.

argument_addr The address (in the caller's address space) where the protected method's arguments can be found. This is essentially a pointer to an abstraction-specific structure containing all the argument information.

The discussion of this call's implementation will include a presentation of the mechanics of Xok's system call handling, in order to make the implementation easier to understand and also demonstrate its correctness. Given that Xok is an exokernel running on Intel x86 processors, corresponding Intel assembly notation will be used.

The steps involved in executing a protected method are described in detail below:

In the caller (user mode)

In order to make a system call, a process needs to load the `EAX` register with the system call number (`SYS_prot_call` in this particular case) and then generate an interrupt using the INT <T_SYSCALL> machine instruction (currently `T_SYSCALL` is defined to be 0x30). All these actions are performed by a stub in the `xok/sys_ucall.h` system header file.

In the interrupt handler (kernel mode)

Once the `INT` instruction is executed, control is passed through the interrupt gate corresponding to `T_SYSCALL`[Intel 96]. This interrupt gate is found in the interrupt descriptor table, which is set up by the exokernel at boot time. After passing through the interrupt gate, the stacks are switched (from user stack to kernel stack) and control

[1] It is actually three arguments or fewer, but every system call is also passed a system call number, not shown in this interface and not visible to the user-space applications.

is passed to the kernel-defined system call interrupt handler. The two stacks are set up as shown in figure 3-1.

At this point, the handler is running in privileged mode (ring 0), in the context of the calling process (except it is using the kernel stack). The interrupt handler then saves on the stack a number of registers and calls the `sys_prot_call` routine.

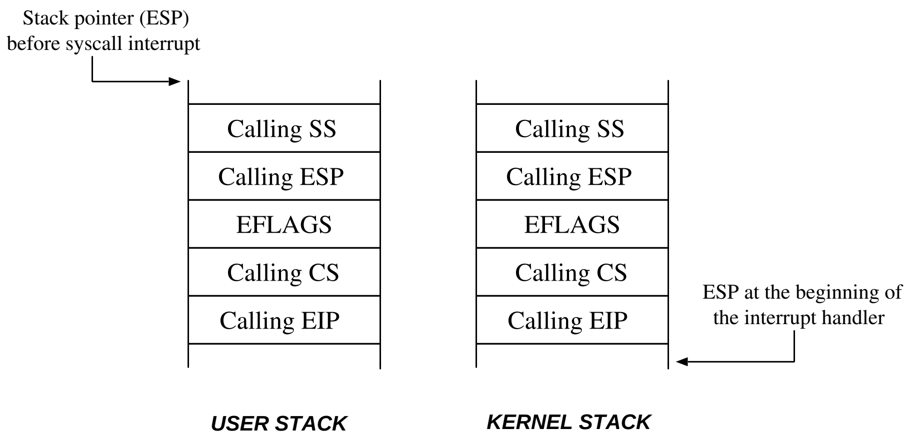

Figure 3-1: Stack configuration after passing through the interrupt gate (stacks grow downward).

In `sys_prot_call` (kernel mode)

The system call proceeds through the following steps:

1. Validate the arguments. This consists of making sure that *invocation_id* denotes a valid (i.e., existing) method and abstraction.

2. Insert the capability associated with the abstraction's state in a well-known slot (`CAP_PROT`) in the caller's cap list.

3. Enable the metadata page in the caller's address space, at the "agreed upon" virtual address. The reason it is being "enabled" and not "mapped" is for performance reasons.

Mapping and unmapping a page consists of a number of costly operations. The only purpose for unmapping a page at the end of a protected method would be to make that page unavailable to the caller application (as opposed to, for instance, the purpose of installing a new mapping in place of the old one). Therefore, it is sufficient to make this page inaccessible, without actually evicting its PTE from the page table. Upon exiting a protected method (see section 3.4), PAM changes the protection (`PG_U`) bit in the PTE from "user accessible" to "kernel restricted".

Hence, in `sys_prot_call`, PAM verifies whether the current mapping of the metadata virtual address corresponds to the physical page containing the metadata, and whether the protection bit is set to kernel mode. If yes, it simply changes the bit back to user mode, without having to perform a costly mapping operation. Otherwise, the mapping is performed as needed. It is important to note that this is a powerful optimization for the common case, in which, if the metadata page were not purposely unmapped by the exiting protected method, its mapping would persist across multiple protected method calls.

The protected method's stack starts at the end of the metadata page and grows toward its beginning. Clearly this limits the size of the stack, but if a protected method expects to need a bigger stack, it can easily allocate additional memory pages and relocate the stack.

4. Install the abstraction's fault handlers. The previous handler entry points are saved in the caller's environment and will be restored when the protected method completes.

5. Depending on whether the user requested it or not (via the flag in *invocation_id*), map the methods' code pages.

   The reason behind this flag is again performance. Given that protected methods have a specific memory area set aside for their code, most applications will not unmap the code pages in between method calls, so there is no need for them to be remapped in `sys_prot_call`. However, if the application did unmap those pages, it can instruct the kernel to remap them. If, for some reason, the caller does not pass in the flag when necessary, a page fault will occur.

It may seem that leaving it completely up to the application to map the code pages would be an interesting feature. However, this would not increase flexibility in any significant way. More importantly, however, it would completely invalidate PAM's correctness: the caller process would have the ability to map any sort of code, not necessarily the one belonging to the protected abstraction. This would violate the trust model described in Section 2.2.

6. Prepare the protected method's stack. This consists of setting up part of a stack frame such that it looks as if the method was called directly from the application. Push the caller's stack pointer and *argument_addr* onto the protected method's stack.

7. Upon completion of its work, the protected method will need to switch from its own stack to the caller's stack, after which it will return. In order for this to be possible, the kernel must prepare the caller's stack at this point in time by pushing the return address onto the caller's stack.

8. Set the field in the caller's environment to reflect the abstraction ID it is currently using.

9. Modify the stack pointer (`ESP`) in the trap frame to point to the protected method's stack (i.e., the end of the metadata page).

10. Modify the instruction pointer (`EIP`) in the trap frame to point to the beginning of the protected method. The eventual `IRET` instruction will transfer control directly to the protected method.

In the interrupt handler (kernel mode)

When `sys_prot_call` returns, control is transfered back to the system call interrupt handler. Some of the registers are restored from the stack, after which the `IRET` instruction is executed.

The effect is that the `CS` (code segment) and `EIP` (instruction pointer) registers are loaded from the trap frame. When the interrupt occurred, the caller's `CS:EIP` were written to the trap frame, but then `sys_prot_call` modified it to point to the protected method's entry point. In Xok, the caller's `CS` equals the protected method's `CS`.

Next, the `EFLAGS` are popped and the `SS` (stack segment) and `ESP` (stack pointer) registers are restored from the trap frame. Remember that, when the interrupt occurred, the caller's `SS:ESP` were written to the trap frame, but `sys_prot_call` modified them to point to the protected method's stack. In Xok, the caller's `SS` is the same with the protected method's `SS`.

Next, control is transferred to the protected method, in ring 3 (lowest privilege level).

In the protected method (user mode)

At the beginning of the protected method, the top of the stack contains *argument_addr*, followed by the caller's `ESP`. The method can now pop *argument_addr* and use it to get the arguments that the caller wanted to pass it. It also has full access to the abstraction's state, because it has the appropriate capability in slot `CAP_PROT`. The method can map the necessary state, perform its work, then unmap the state, and destroy the capability (details on how to write a protected method are given in appendix A).

After that, the protected method needs to pop the stack pointer and return. Popping `ESP` will cause the stack to be switched to the caller's stack and the `RET` instruction will use the `EIP` pushed by the kernel prior to passing control to the protected method.

Back in the caller (user mode)

Control has returned to the caller, just as if it had made a normal procedure call. Its stack is in the expected state and the capability to access the abstraction's state has been destroyed.

3.4 sys_prot_exit

```
void sys_prot_exit ( void )
```

This system call is invoked by a protected method whenever it wishes to relinquish its relatively privileged status. In most cases, the protected method would return to the caller after invoking `sys_prot_exit`, however this is in no way required or enforced by the kernel. The method may need to perform, prior to returning, some actions that do not require access to the abstraction's shared state (see an example in Section A.1.1).

`sys_prot_exit` performs four simple tasks:

- Change the protection bit (`PG_U`) in the metadata page's PTE from "user accessible" to "kernel restricted"; invalidate the TLB entry for the corresponding virtual address. This ensures that, upon return, the process cannot access the metadata page.
- Invalidate (in the calling process' cap list) the capability that protects the abstraction's state.
- Restore the fault handlers to the values they had prior to the invocation of the protected method. These values were saved in the caller's environment by `sys_prot_call`.
- Set the field in the process' environment indicating that it is not in the sensitive region of a protected method (i.e. set the "executing abstraction" ID to -1).

3.5 sys_prot_remove

```
int sys_prot_remove (  unsigned int abstraction_id,
                       unsigned int capability_slot )
```

The system call's arguments are:

abstraction_id The identifier of the abstraction to be removed. This should be the return result of the `sys_prot_define` call that defined the abstraction.

capability_slot The index of the slot containing the capability that guards this abstraction's state. Normally, only the abstraction's definer would have this capability, or any other process to whom the definer gave the capability.

Provided the capability in *capability_slot* dominates the abstraction's capability, all the fields in the corresponding line in the abstraction table are zeroed out. The return value indicates whether the operation succeeded or not.

3.6 Discussion

The implementation of the protected abstraction mechanism has occasionally deviated from the design presented in Chapter 2, primarily for performance reasons. Some of the more important optimizations are:

- The metadata page is typically not unmapped upon exiting a protected method, and not mapped again upon entry. Instead, the PTE corresponding to this page is modified in such a way that the process cannot access the page anymore. Clearing/setting a bit in the PTE is much faster than unmapping/mapping the page.

- The protected methods' code is mapped "on demand," whenever the caller of a protected method wishes so. Both this optimization and the previous one make the common case very efficient, by eliminating the need for updating page table contents.

- In the design we stated that the protected abstraction would be responsible for unmapping all its state. For performance reasons, the system calls that are expected to be invoked most often when exiting a protected method were batched together into a single call, `sys_prot_exit`.

It is important to understand what the costs of this implementation are in terms of space. The abstraction itself is associated with its state (consisting in most cases of memory pages and disk blocks) and the space taken by the protected methods. In addition to that, every caller's environment has two sets of fields used by PAM: one where the original fault handler entry points are saved and one where the currently executing abstraction's ID is kept.

Some of the benefits of PAM's implementation are:

- It provides a simple, effective, and efficient way to force applications to access abstraction solely through the protected methods. These methods can therefore enforce all necessary invariants.

- It is clear that the kernel simply provides the necessary *mechanism* and enforces absolutely no management policy.

- Different library operating systems can easily communicate and share objects using this scheme, given that it is not dependent on any specific operating system personality (e.g., a program linked with a Windows NT libOS could easily share the same filesystem with a UNIX process).

- The abstractions and their methods are very easy to debug, since they run in user space.

Chapter 4

Evaluation

This chapter will evaluate three different aspects of the protected abstraction mechanism: performance, scalability, and usability. To analyze performance, we define a sample abstraction and implement it both as a user-level server and as a protected abstraction. We then measure and compare performance of the three implementations. Section 4.4 analyzes to what degree PAM scales with the number of defined abstractions as well as number of abstraction users. Finally, Section 4.5 provides an assessment of PAM's ease of programming.

4.1 Pseudo-Stack: A Sample Abstraction

For the purpose of measuring performance, we will use a very simple abstraction: a pseudo-stack. This abstraction is a repository into which bytes can be pushed and from which bytes can be popped. Ordering is "last in first out" (LIFO) with respect to the independent push and pop operations (i.e., a pop will see the data most recently pushed). The abstraction user can push or pop multiple bytes at a time, and the ordering among these bytes is "first in first out" (FIFO) (i.e., popping two bytes from the pseudo-stack yields the first-before-last followed by the last byte of the last push operation).

There is no particular reason to this strange asymmetry except that it significantly simplifies the implementation of the abstraction. This particular ordering is in no way relevant to the results reported below, because the measurements are not concerned with the exact semantics of the abstraction.

The pseudo-stack's interface consists of the following operations:

- `S_INIT ()` initializes the abstraction; this method must be called prior to any other operation.
- `S_PUSH (push_buf, push_nbytes)` pushes onto the stack *push_nbytes* bytes from the *push_buf* buffer.
- `S_POP (pop_nbytes, pop_buf)` pops *pop_nbytes* bytes from the stack into *pop_buf*.
- `S_EMPTY ()` is a null method used for testing null call performance. It is not necessarily part of the interface to a pseudo-stack, but it is convenient for the sake of experimentation.

The state of a pseudo-stack contains two elements:

- A counter, `nbytes`, which keeps track of the number of bytes currently on the pseudo-stack;
- A buffer, `buf`, containing the pseudo-stack data; the bottom of the pseudo-stack is at the beginning of `buf`, the `nbytes`$^{\mathrm{th}}$ element of `buf` represents the byte at the top of the pseudo-stack.

As is the case with any abstraction, there is a set of invariants and update rules that define consistency of the pseudo-stack's state. Here are some of the less trivial ones:

- The number of bytes on the stack does not exceed a fixed maximum number (`MAX_SIZE`);
- The number of bytes in `buf` equals `nbytes`;
- In a `S_PUSH` operation, if `push_nbytes` + `nbytes` exceeds `MAX_SIZE`, then the operation fails;

- In a `S_POP` operation, if `nbytes - pop_nbytes` is less than zero, then the operation fails.

4.2 Three Implementations of Pseudo-Stack

There are a number of different ways in which abstractions can be implemented and used safely. One option is the one advocated by the microkernel architecture: the abstraction is implemented as a privileged process and user processes can use IPC to communicate with the server and have it perform actions on their behalf. Another option would be to download abstraction code into the kernel (as described in [Deutsch and Grant 71], [Wahbe et al. 93], [Necula and Lee 96], and others). Protection is ensured by either sandboxing the code or verifying its conformance to a given set of correctness rules. A more exokernel-specific approach would be that of using application-specific handlers, which are described in [Wallach et al. 96]. Yet another possibility is to use a LRPC-like mechanism, as shown in [Bershad et al. 90].

We chose for comparison two different approaches: a simple server using pipes to communicate with its clients, and another server using synchronous IPC. Both approaches achieve a degree of flexibility similar to PAM's and provide a satisfactory level of performance. The following section describes the pipe-based implementation and Section 4.2.2 presents the LRPC-like approach.

4.2.1 Client/Server with Pipes

The pseudo-stack was implemented as a server process, which forks off its clients and establishes pipes between itself and its children. The children/clients issue commands to the parent/server by sending specific byte values through the pipe. These values are interpreted by the server as requests for invoking specific methods in the abstraction's interface.

Following the command, clients send arguments through the pipe. On the receiving end, the method corresponding to the initial command reads its arguments from the pipe, performs its task, and then sends its return value through the pipe back to the client. Appendix B shows some of the code for this implementation of the pseudo-stack.

4.2.2 Client/Server with LRPC

We also implemented the pseudo-stack as a server process that can get requests from clients to execute procedures (the abstraction's methods) in its own address space. Arguments are passed between client and server via shared memory and the requests/responses are mediated by standard IPC mechanisms, in effect implementing a form of LRPC.

As shown in Appendix B, the test program forks off a new process that constitutes the client; the parent process plays the role of the server. The parent registers an IPC handler, which acts as a dispatcher: on an incoming request, it decides which of its internal procedures to invoke in order to perform the desired operation.

Communication between the client and the server occurs via two different mechanisms: IPC and shared memory. The client process allocates a page of memory which gets mapped by both the client and the server. Arguments and results are passed around in this page. The method calls done by the client get translated into IPC requests that get sent to the server; every time an IPC is received by the server process, its IPC handler is invoked and it passes control to the requested procedure. The procedure is then responsible for extracting arguments from the shared memory page, perform its task, and then return the result in that same memory page. The implementation is optimized in that some of the arguments are passed as the IPC's arguments.

4.2.3 Pseudo-Stack as a Protected Abstraction

The protected method implementation of the pseudo-stack follows closely the methodology described in Appendix A. We define a set of protected methods along with a page fault handler. This handler is not particularly useful, but it is provided here to illustrate the concept and also serve as an example.

The astute reader will notice that the pseudo-stack abstraction is not fully protected, given that only the page faults are caught by the abstraction code. The other handlers were not written for reasons of simplicity and clarity. Given that neither of the handler-specific events occur during the tests, the lack of handlers will not influence in any way the results.

One interesting feature is the definition of a non-exported function `local_memcpy`. Given that a protected abstraction cannot transfer control to untrusted code (such as the ExOS library), it cannot invoke `memcpy`, the way the abstraction server does. It must provide its own. The distinction between the two trust instances is fairly subtle: the server must trust its version of `memcpy`, because it belongs to the library with which the server is linked; however, if the protected abstraction needed to trust `memcpy`, then it would in effect have to trust its users' versions of `memcpy`.

4.3 Null Method Call Performance

All measurements reported in this chapter were performed on a personal computer based on a 200 MHz Pentium Pro processor, with 64 Megabytes of RAM. All other aspects of the PC's configuration are irrelevant to the tests.

This section presents the performance of null method calls. These null calls are invocations of methods that do not perform any action, and so they are a good indication of the overhead (fixed cost) associated with PAM and the two server versions of the abstraction.

4.3.1 Average Null Call Cost

In this experiment we performed 10,000 null calls consecutively and measured the time it took for them to complete. The average cost per call was then computed. In addition to measuring the performance of PAM and the servers, we also measured, for reference purposes, the cost of a null system call. The null system call is an indication of the cost involved in crossing the user/kernel boundary. Table 4.1 shows the results, in microseconds.

Null call	Total time (μsec)	Average time (μsec/call)
Pipe-based server	206,675	20.67
LRPC server	68,030	6.80
PAM implementation	54,823	5.48
System call	19,607	1.96

Table 4.1: Null call costs for the pipe-based server, LRPC server, PAM, and kernel.

As seen in Chapter 3, each protected method requires the invocation of two system calls. Hence, assuming invocation times can be added linearly, 2 x 1.96 = 3.92 microseconds are consumed by the two system calls, and the remainder of $5.75 - 3.92 = 1.83$ microseconds represents PAM's overhead in managing data structures, hiding/revealing the shared state, granting/revoking capabilities, installing/restoring fault handlers, switching stacks, etc.

4.3.2 Null Method Call Breakdown

The measurements presented in this section were used primarily during PAM's optimization. Figure 4-1 gives an approximate breakdown of the time spent during a null protected method invocation. Measurements are given in number of machine cycles and represent averages over 10,000 invocations. This figure, in conjunction with the information provided in Sections 3.3 and 3.4, provides a fine grain picture of how workload is distributed among the different sections of code. The tests were performed on a 200 MHz processor, so dividing the number of cycles by 200 will yield time in microseconds.

As can be seen in Figure 4-1, the main costs are associated with crossing the user/kernel boundary.

4.3.3 Batched Null Calls

This experiment looks at the behavior of the PAM implementation and the LRPC-based implementation under different types of loads: we start the test process, have

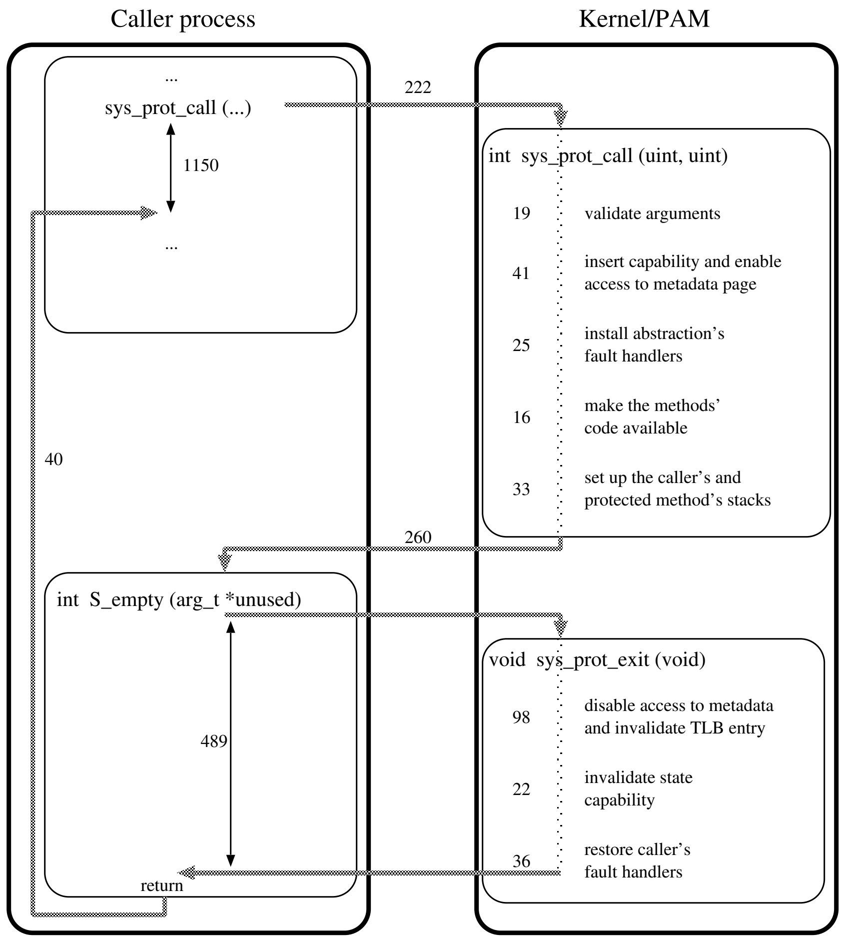

Figure 4-1: Breakdown of time spent in a null method call.

it make a sequence of null calls, and then measure the time it took to complete this sequence. We repeat the test for different sequence lengths. Between every two consecutive method invocations, the test process uses the `bzero` function to touch an entire Megabyte of memory. This action ensures that both the level 1 cache (8 Kb) and level 2 cache (256 Kb) are completely emptied between successive calls. The pipe-based server's performance is several times poorer than both the LRPC-based server and the PAM implementation of the pseudo-stack. Hence, the pipe-based results are not shown here.

Figure 4-2 shows a plot of the 400 data points we collected during the test. This graph reveals the type of behavior we should expect from the two systems we are comparing. It is a somewhat pleasant surprise to see that, over the range that was measured, both PAM and the server exhibit a linear increase in duration. However, the rate at which the server-based abstraction's durations increase is higher than the rate at which PAM durations increase.

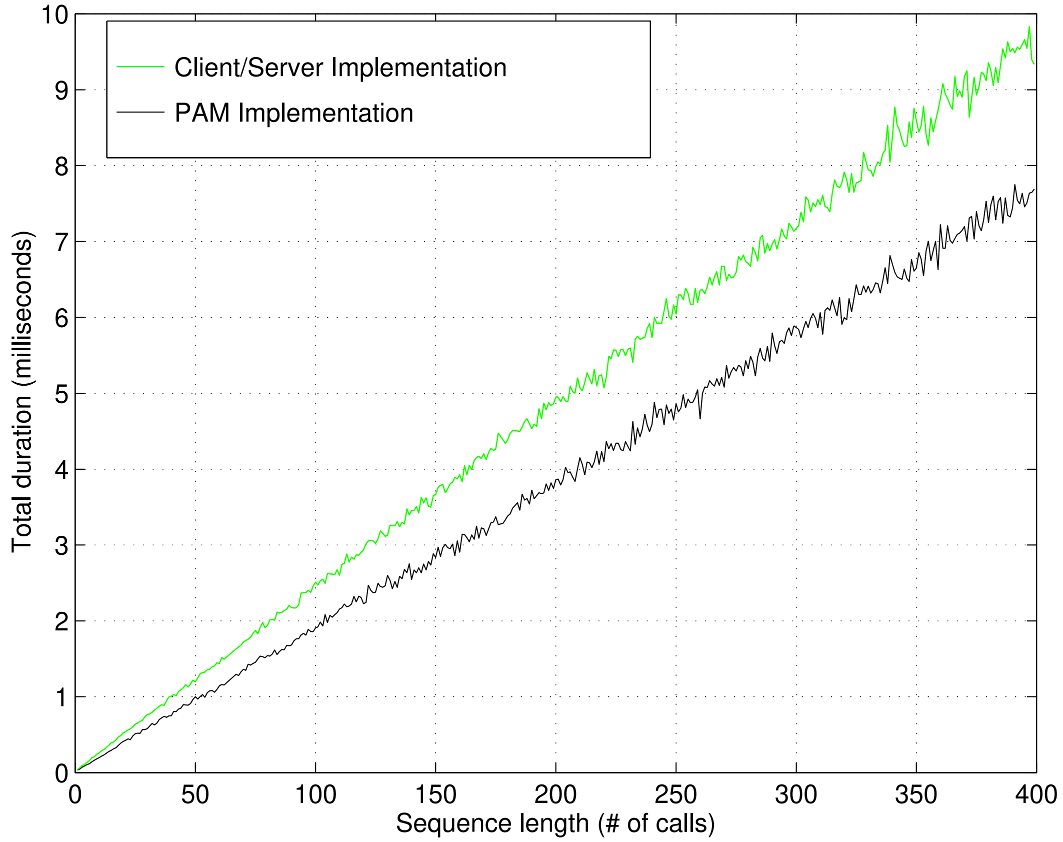

Figure 4-2: Performance of batched null calls.

4.4 Scalability

There are two operating characteristics of PAM whose magnitudes can presumably affect its performance: the number of abstractions and methods currently defined in the system, and the number of simultaneous users of an abstraction.

As the number of abstractions increases, PAM would not lose its performance because of the table data structure described in Chapter 3. Lookup operations in a table can be effected in constant time, regardless of the size of the table. The same argument holds for increases in the number of methods.

However, there exist some second order effects which need to be taken into consideration. For example, the higher the number and size of methods in an abstraction, the more code pages are required to run either of these methods. Consequently, the cost of the first protected method call for each abstraction may be considerably high. However, the cost of all subsequent calls will not be affected by the size of the abstraction in most cases, due to the optimization described in Section 3.3.

The case in which the cost of a protected method invocation would be always high is when the application uses multiple abstractions in sequence. The alternating calls would cause the abstractions to evict each other's code pages, so the pages would need to be remapped every time a method would be invoked. The performance loss in this case could be significant.

PAM, however, scales quite graciously with the number of simultaneous clients of an abstraction. Protected methods, being "downloaded" into every abstraction user's address space, become distributed throughout the system, which avoids the formation of a bottleneck. For instance, in the case of a user-level server, if many clients were to IPC to the server, each client would experience an increased delay in response time. Under Xok in particular, it would be the client with the first outstanding IPC that would suffer the most. This is not the case in PAM, assuming that synchronization of access to the shared state is not done at a level that is too coarse.

In the same context, one other PAM feature that improves scalability is the fact that time spent in a protected method is "charged" to the caller, along with any other additional resources the method uses during that particular invocation. This would clearly not be the case when using a client/server implementation of abstractions. PAM's self-enforcing model allows the cost of using the abstraction to be independently offloaded to every single client.

4.5 Ease of Use

One of this thesis' claims is that the paradigm of "downloading part of the kernel into user space" is easier to understand than many of the other mechanisms available for sharing abstractions. Protected methods play the role of system/library calls, dynamically augmenting the existing system/library functionality.

Programmers use the system/library call interface in every day programming, so learning to use an extension of such an interface is trivial. This is particularly true when we contrast using PAM with writing and using server-based abstractions, which requires doing IPCs, writing IPC handlers, setting up and mapping shared memory pages, etc. Preferably programmers should be shielded from such low-level details.

Another aspect of PAM that makes it easy to use is the fact that abstractions have full access to their callers' address space. Hence, a programmer does not need to decide a priori which data structures will be shared with which abstractions. He/she can easily pass as arguments pointers to these structures, in the same way that local procedure calls are given pointers to large data structures in order to avoid passing them on the stack. This type of flexible data structure sharing would not be possible in a client/server implementation of abstractions.

One last point is that writing concurrent abstractions is easy with PAM. The distributed nature of protected methods makes it clear that there are effectively multiple copies of the access methods using the same shared state in parallel, and thus requires simple, rigorous synchronization. Every set of methods downloaded into an address space effectively constitutes a thread of control with respect to the abstraction, yet there is no need for a sophisticated threads package.

Chapter 5

Related Work

Microkernels [Rozier et al. 88; Golub et al. 90; Hildebrand 92] allow for servers to define abstractions. This is an easy and elegant method for providing protection. Usually there are high overheads associated with the communication between applications and the server, but recent results have shown that "second-generation" microkernels can reduce the communication overheads and provide performance that is closer to that of monolithic kernels [Härtig et al. 97].

A number of operating systems use language features to provide protection: Pilot [Redell et al. 80], Fox [Cooper et al. 91], Oberon [Mössenböck 94], and, more recently, SPIN [Bershad et al. 95]. SPIN provides a mechanism by which applications can extend and specialize the operating system. The SPIN approach is essentially that of downloading safe object code into the kernel via dynamic linking. The downloadable extensions are mostly written in Modula-3, a type-safe language that requires explicit specification of interfaces between modules and provides automatic storage allocation/deallocation. The interface specification requirement makes it easy to control memory accesses across module boundaries and thus makes for a good fault isolation model and an elegant implementation of logical protection domains. However, SPIN cannot enforce invariants via these extensions. Also, using extensions written in Modula-3 requires that SPIN contain the Modula-3 run-time system in the kernel, which represents over 20% of the kernel size (adds slightly over 280 KB). The use of a specific language restricts extension writers unnecessarily.

Another language-based approach is that of extending kernels through small interpreted languages [Mogul et al. 87; Lee et al. 94; Yuhara et al. 94]. Some of the problems with this approach, as evidenced for instance by the packet filter [Mogul et al. 87], are the restricted possibility of expression, narrow interfaces, and overhead due to interpretation.

One of the primary goals of PAM was to make it very convenient for programmers to write protected abstractions, which means they should be allowed to use the compiler of their choice. Unfortunately, the most widely used languages are not type-safe, so a language-based approach to protection would not have met the usability goals set forth by this thesis.

Heidemann and Popek [1994], Rozier et al. [1988], Candea and Jones [1998] describe different ways of extending the kernel, however all require special privilege and provide no fault isolation, which means that the installable extensions can crash the entire system. Such a system does not allow user-level applications to define and safely export abstractions.

Wahbe et al. [1993] describe two basic methods for achieving software-based fault isolation: segment matching and address sandboxing. This protection mechanism provides very powerful features but unfortunately comes at a significant cost, because every load and store instruction has to be enclosed in a "protection envelope". However, many systems have used mechanisms similar to this one and it seems to be valuable [Lucco 94; Small and Seltzer 94; Engler et al. 95].

Deutsch and Grant [1971] show how to use code inspection to dynamically add components (e.g., for profiling) to a kernel at run-time. The basic approach is to identify the load, store, and branch instructions that would potentially violate the safety assumptions.

[Bershad et al. 90] describes an approach that could be used as an alternative to PAM: lightweight remote procedure call (LRPC). Bershad et al. [1990] present a facility designed to optimize regular RPC for intra-machine communication. LRPC uses four techniques to improve the performance of RPC systems: simplified control transfer, simplified data transfer using sharing techniques, optimized stubs, and optimizations for multiprocessors.

Yarvin et al. [1993] take an interesting approach to protection by exploiting the large size of 64-bit address spaces. The paper shows how to map pages at random addresses in a process' address space and thus provide protection by secrecy. The probability that a process can discover the location of the mapped page is low. However, this approach is not practical for 32-bit address spaces, which is what our current exokernel platforms have.

An approach that is quite similar to the one shown in this thesis is "protected shared libraries" (PSL) described by Banerji et al. [1997]. PSLs are a type of support for modularity that forms a basis for building library-based operating system services. PSLs extend the familiar notion of shared libraries with protected state and data sharing across protection boundaries. However, the trust model assumed by PSLs is different from the one we used for PAM. For this reason, PSLs incur additional overhead that would not have been tolerable for PAM (e.g., large amounts of memory may be mapped/unmapped on every call).

One other interesting approach is the use of proof-carrying code, described by Necula and Lee [1996]. Using this technique, the code that is about to be downloaded can provide a proof that it obeys a safety policy set by the kernel. While this technique may not be practical at this point in time, it is very promising and may prove to be a powerful tool for extending kernels.

Chapter 6

Conclusions

This thesis has presented motivation for building a mechanism that provides safe sharing of user-level abstractions via protected methods and the design, implementation, and evaluation of the *protected abstraction mechanism* (PAM).

The primary motivation for this work has been the need for securely sharing system abstractions while still maintaining all the flexibility that an exokernel architecture offers. Conceptualizing an abstraction as an object of the type used in object-oriented languages helps understand how PAM can guard the invariants associated with shared state: if access to the abstraction's state is restricted to a set of trusted access methods, then the state's consistency can be guaranteed.

Central to PAM is the concept of downloading abstraction code into applications, or moving part of the kernel into user space — the access methods that enable use of an abstraction are mapped into every user process' address space. Access to the abstraction's state is controlled with Xok's system of hierarchically named capabilities. A given abstraction's state is hidden whenever a process is not executing a protected method belonging to that abstraction, and revealed as soon as control is passed to one of the access methods. This way, the kernel can guarantee that the invariants implicitly encoded in the access methods are preserved.

As evidenced in Chapters 3 and 4, PAM exhibits a number of desirable features:

- It is flexible, in that unprivileged applications have the ability to define and use protected abstractions.

- PAM exhibits a total separation between mechanism and policy, an approach advocated by the exokernel philosophy. This increases the amount of control that abstractions and their users have over the management of abstraction state.

- PAM is easy to use and understand, as it preserves the well-understood concept of a system call interface to the protected abstractions. Reading Appendix A of this thesis should be sufficient for the average programmer to successfully start writing and debugging abstractions.

- While certainly not as fast as kernel implementations of the equivalent abstractions, PAM outperforms server-based solutions.

Appendix A

Programmer's Manual

This appendix presents two important aspects of programming with PAM: writing abstractions and debugging them. It is meant as a quick guide to writing functional abstractions in a short period of time.

A.1 Writing Abstractions

All the PAM-related types are defined in `xok/protmeth.h`; this is the only file that needs to be included in order to make use of PAM. There are two steps involved in defining an abstraction: writing the methods and declaring the abstraction.

A.1.1 Writing Protected Methods

This section shows how to write a simple abstraction. The abstraction's state consists of one integer value, stored at the beginning of the metadata page, which starts at address `META_VA`. The abstraction's invariant states that the value stored in the state is always between 0 and 100. The abstraction has one single protected method, called `set_value`, which sets the value of the abstraction state and returns the value it had prior to the update. If the value passed in as an argument is invalid, the method prints out an error message and returns -1.

```
int set_value (int *argptr)
{
  if (*argptr>0  &&  *argptr<100) {
    int prev = *META_VA;
    *META_VA = *argptr;
    sys_prot_exit();
    return prev;
  }

  sys_prot_exit();
  printf ("set_value: Argument out of range.\n");
  return (-1);
}
```

Upon entry to the method, the metadata page is available read/write at address `META_VA`. This address is passed in at definition time (see Section A.1.2) and stays constant throughout the lifetime of the abstraction. If the requested update satisfies the invariant, the method performs it, calls `sys_prot_exit` to drop privilege, and returns. If the argument is not valid, the method drops privilege, prints an error message, and returns.

It is important to realize that protected methods may not make calls to library functions while being in the privileged state, because these functions cannot be trusted. The kernel cannot provide any sort of guarantees[1] once the method voluntarily yields control to code that is outside the abstraction. For printing out messages while in privileged mode, a protected method can safely use the trusted `sys_cputs` system call. Although the protected method shown above is invoking untrusted code (i.e., the `printf` library function), it does so after having called `sys_prot_exit`. At this point the process does not have access to the abstraction's state anymore.

The entry point to the method will not be the beginning of `set_value`, rather the beginning of the stub shown below. The stub is the one that calls `set_value` and is in charge of switching the stacks once `set_value` completes. The stacks cannot be switched in a trivial way inside the method itself, because the compiler may spill some of the registers onto the protected method's stack and restore them only at the end of the method. If these registers are callee-saved, then the method would violate

[1]Future work may address this issue by using *callout functions*, which would allow temporary exits from the protected area into unprotected, untrusted code.

the calling convention. This stub's execution time is absolutely negligible (4 cycles on Pentium processors).

```
asm(".align 2              ");
asm(".globl _set_value_stub ");
asm("_set_value_stub       ");
asm("   call _set_value    ");
asm("   movl 4(%esp), %esp  ");
asm("   ret                ");
```

A.1.2 Declaring Protected Abstractions

This section will show how to define an abstraction, once the methods have been written. Defining an abstraction consists of making the `sys_prot_define` system call and passing it the arguments shown in Section 3.2. In the code below, we assume functions for the fault handlers have been written (see Appendix B for a page fault handler example) and are available to the module defining the abstraction as functions *epilogue_handler*, *prologue_handler*, etc.

```
void main (void)
{
  mtab_t     abstraction;
  metadata_t handlers = { epilogue_handler, prologue_handler, fault_handler,
                          ipc1_handler, ipc2_handler, rtc_handler };

  /* Allocate additional state here and protect it with CAP_ENV */

  abstraction.nmethods = 1;
  abstraction.start[0] = set_value_stub;

  if (sys_prot_define (&abstraction, CAP_ENV, 4096, META_VA, &handlers)) {
    printf ("main: Error defining abstraction.\n");
    exit(-1);
  }

  while (1)
    sleep (1000);
}
```

When defining the abstraction, the programmer must specify the capability that protects the shared state. In the code above we used the ExOS-specific slot `CAP_ENV`, which contains a capability of the form `1.uid.uid.pid.pid`. Thus, for the lifetime of this process, this capability is unique system-wide. In the case of the abstraction defined here, however, the capability is really not necessary, since there is no state in addition to the metadata page.

At the end of the program there is an infinite loop which yields the CPU. We cannot allow the defining process to exit, because it must keep ownership of the memory pages containing the code of the abstraction, or else the abstraction becomes unavailable. This loop is simply a way for the process to stay alive without consuming CPU resources.

A.2 Debugging Abstractions

Protected methods are downloaded into their callers' address spaces, so abstractions are debugged by debugging their caller process. Debugging an abstraction is equivalent to debugging any other program, with one minor exception: the protected methods' symbols are not part of the debugged program. The abstraction's symbols must be loaded from the object file corresponding to the protected methods in addition to the caller's symbols. For instance, in the GNU debugger (gdb), the `add-symbol-file methods.o 0x8000000` command could be used.

Appendix B

Pseudo-Stack Implementation

This appendix provides code snippets from the implementation of the pseudo-stack, as described in Chapter 4.

B.1 The Pseudo-Stack as a Protected Abstraction

```
#define STUB(stub_meth,real_meth)         \
  void stub_meth (void);                  \
  asm(".align 2                        "); \
  asm(".globl _" #stub_meth             ); \
  asm(".type _" #stub_meth ",@function"); \
  asm("_" #stub_meth ":                "); \
  asm("   call _" #real_meth            ); \
  asm("   movl 4(%esp), %esp           "); \
  asm("   ret                          ")

/* Initialize the abstraction */

STUB(S_init_stub, S_init);
int S_init (arg_t *unused)
{
  p_state->nbytes = 0;
  p_state->buf    = (byte *) ((uint) p_init + sizeof(int));
  sys_prot_exit ();
  return 0;
}
```

```
/* Push s_state->nbytes bytes from s_state->buf onto the stack. */

STUB(S_push_stub, S_push);
int S_push (arg_t *s_state)
{
  int i;

  if (p_state->nbytes + s_state->nbytes > MAX_LEN) {
    sys_prot_exit();
    return (-1);
  }
  local_memcpy (s_state->buf, p_state->buf + p_state->nbytes,
                s_state->nbytes);
  p_state->nbytes += s_state->nbytes;
  sys_prot_exit();
  return 0;
}

/* Pop s_state->nbytes from stack into s_state->buf */

STUB(S_pop_stub, S_pop);
int S_pop (arg_t *s_state)
{
  int i;

  if (p_state->nbytes < s_state->nbytes) {
    sys_prot_exit();
    return (-1);
  }
  local_memcpy (p_state->buf + p_state->nbytes - s_state->nbytes, s_state->buf,
                s_state->nbytes);
  p_state->nbytes -= s_state->nbytes;
  sys_prot_exit();
  return 0;
}


/* Don't do anything */

STUB(S_empty_stub, S_empty);
int S_empty (arg_t *unused)
{
  sys_prot_exit ();
  return 0;
}
```

```
/* A silly, suicidal page fault handler */

asm(".align 2");
asm(".globl _handler");
asm("_handler:");
asm(" movl %eax,32(%esp)");
asm(" movl %edx,28(%esp)");
asm(" movl %ecx,24(%esp)");
asm(" call _real_handler");

int
real_handler (u_int va, u_int errcode, u_int eflags,
              u_int eip, u_int esp)
{
  sys_cputs ("PAGE FAULT: ");
  if (errcode & FEC_PR)  sys_cputs ("<Protection violation> ");
  if (errcode & FEC_WR)  sys_cputs ("<Fault on write> ");
  if (errcode & FEC_U)   sys_cputs ("<Fault in user mode>");
  asm("hlt");  /* Privileged instruction causes process to be killed */
  return 0;
}

/* Cannot use library implementation of memcpy, so we write our own. */

void local_memcpy (uint src, uint dst, uint nbytes)
{
  __asm __volatile
    ("\n"
     "movl  %0,%%esi   \n"
     "movl  %1,%%edi   \n"
     "movl  %2,%%ecx   \n"
     "cld              \n"
     "shrl  $2,%%ecx   \n"
     "rep             \n"
     "movsl           \n"
     "movl  %2,%%ecx   \n"
     "andl  $3,%%ecx   \n"
     "rep             \n"
     "movsb             " :
     : "m" (src), "m" (dst), "m" (nbytes)
     : "esi", "edi", "ecx", "memory");
}
```

B.2 The LRPC-Like Pseudo-Stack

This section shows code for the implementation of the pseudo-stack as a LRPC type server. Clients invoke server procedures by sending requests over IPC and placing arguments in shared memory (or sending them as part of the IPC).

```
void main (void)
{
  pid_t    pid;
  uint     pte, envid;
  char     buf[MAX_LEN];

  pid = fork();

  /*------------- the server -----------------------*/
  if (pid == 0) {
    ipc_register(IPC_USER, dispatch);
    sleep(5);
    return;
  }

  /*------------- the client -----------------------*/

  envid = pid2envid(pid);
  yield(envid);

  /* Shared memory used to communicate with the server */

  if (! (shared_state = (arg_t *) malloc (sizeof(arg_t))))
    exit(-1);

  /* Figure out the pte corresponding to the shared page */

  shared_state->nbytes = 123; /* touch memory (it's copy-on-write) */
  pte = sys_read_pte ((uint) shared_state, 0, sys_geteid());

  S_INIT(envid,pte);
#include "test-sequence.c" /* this part does the testing */
}

/*----------------------------------------------------------------------------
      The client's methods start here.
  --------------------------------------------------------------------------*/

/* Send request to initialize the abstraction */

int S_INIT (int envid, int pte)
{
  return (sipcout(envid, IPC_USER, INIT, pte, 0));
}
```

```
/* Send request to push num_bytes from buf onto the stack */

int S_PUSH (int envid, byte *buf, int num_bytes)
{
  if (num_bytes > MAX_LEN)
    return -1;
  shared_state->nbytes = num_bytes;
  memcpy (shared_state->buf, buf, num_bytes);
  return (sipcout(envid, IPC_USER, PUSH, 0, 0));
}

/* Send request to pop num_bytes from stack into buf */

int S_POP (int envid, int num_bytes, byte *buf)
{
  if (num_bytes > MAX_LEN)
    return -1;
  shared_state->nbytes = num_bytes;
  if (sipcout(envid, IPC_USER, POP, 0, 0) != 0)
    return -1;
  if (shared_state->nbytes != num_bytes)
    return -1;
  memcpy (buf, shared_state->buf, num_bytes);
  return 0;
}

/* Send request for the empty method */

int S_EMPTY (int envid, int num_bytes, byte *buf)
{
  if (sipcout(envid, IPC_USER, EMPTY, 0, 0) != 0)
    return -1;
  return 0;
}

/*---------------------------------------------------------------------------
     The server's methods start here.
  -------------------------------------------------------------------------*/

arg_t         *s_state;                /* the shared state  */
arg_t         private_area;            /* the private state */
arg_t         *p_state = &private_area;

/* The dispatching IPC handler */

int dispatch (int unused0, int meth_num, int optional, int unused, uint caller)
{
  switch (meth_num) {
  case INIT:  return (S_init (optional));
  case PUSH:  return (S_push ());
  case POP:   return (S_pop ());
  case EMPTY: return (S_empty());
  }
  return -1;
}
```

```
/* Initialize the abstraction */

int S_init (int pte)
{
  s_state = (arg_t *) malloc (sizeof(arg_t));
  if (s_state == NULL || sys_self_insert_pte (0, pte, (uint) s_state) != 0)
    return -1;
  p_state->nbytes=0;
  return 0;
}

/* Push s_state->nbytes bytes from s_state->buf onto stack */

int S_push (void)
{
  if (p_state->nbytes + s_state->nbytes > MAX_LEN)
    return -1;
  memcpy (p_state->buf + p_state->nbytes, s_state->buf, s_state->nbytes);
  p_state->nbytes += s_state->nbytes;
  return 0;
}

/* Pop s_state->nbytes bytes from the stack into s_state->buf */

int S_pop (void)
{
  int i;

  if (p_state->nbytes - s_state->nbytes < 0)
    return -1;
  memcpy (p_state->buf + p_state->nbytes - 1, s_state->buf, s_state->nbytes);
  p_state->nbytes -= s_state->nbytes;
  return 0;
}

/* Do nothing */

int S_empty (void)
{
  return 0;
}
```

B.3 The Pseudo-Stack Server With Pipes

This section shows code for the implementation of the pseudo-stack as a server that communicates with its clients over pipes.

```
/* The file descriptors for the ends of the pipes */
static int pfd1[2], pfd2[2];
#define CLI_IN  (pfd1[0])
#define CLI_OUT (pfd2[1])
#define SRV_IN  (pfd2[0])
#define SRV_OUT (pfd1[1])

/*-----------------------------------------------------------------------------
 * The SERVER's state
 *----------------------------------------------------------------------------*/

unsigned char Buf[MAX_LEN];
unsigned int  NBytes;

/*=============================================================================
 * The SERVER's functions
 *============================================================================*/

int srv_empty (void)
{
  return 0;
}

/*--------------------------------------------------------------
 * srv_init initializes the abstraction.
 * actual ARGS: none
 *-------------------------------------------------------------*/

int srv_init (void)
{
  NBytes = 0;
  return 0;
}
```

```
/*----------------------------------------------------------
 * srv_push pushes bytes from the pipe onto the pseudo-stack.
 * actual ARGS: number of bytes
 *              data to be pushed
 *----------------------------------------------------------*/

int srv_push (void)
{
  int nbytes;

  if (read(SRV_IN, &nbytes, sizeof(nbytes)) != sizeof(nbytes)) {
    perror ("srv_push (reading nbytes)");
    return (-1);
  }
  if (NBytes + nbytes > MAX_LEN) {
    printf ("srv_push: Too many bytes to push.\n");
    return (-1);
  }
  if (read(SRV_IN, &Buf[NBytes], nbytes) != nbytes) {
    perror ("srv_push (reading data)");
    return (-1);
  }
  NBytes += nbytes;
  return 0;
}

/*----------------------------------------------------------
 * srv_pop pops bytes off the pseudo-stack and writes them to
 * the pipe.
 * actual ARGS: number of bytes
 *----------------------------------------------------------*/

int srv_pop (void)
{
  int nbytes;

  if (read(SRV_IN, &nbytes, sizeof(nbytes)) != sizeof(nbytes)) {
    perror ("srv_pop (reading nbytes)");
    return (-1);
  }
  if (NBytes - nbytes < 0) {
    printf ("srv_pop: Too many bytes to pop.\n");
    return (-1);
  }
  if (write(SRV_OUT, &Buf[NBytes-nbytes], nbytes) != nbytes) {
    printf ("srv_pop (writing data): ERROR");
    return (-1);
  }
  NBytes -= nbytes;
  return 0;
}
```

```c
/*============================================================================
 * The SERVER dispatcher. This function waits for commands to come
 * down the pipe and then dispatches them. It also sends down the pipe
 * the result of the operation.
 *============================================================================*/

int dispatch (void)
{
  unsigned char command, result;

  while (1) {
    if (read(SRV_IN, &command, sizeof(command)) != sizeof(command)) {
      perror ("dispatch (reading command)");
      return (-1);
    }
    switch (command) {
    case INIT:  result = srv_init(); break;
    case PUSH:  result = srv_push(); break;
    case POP:   result = srv_pop();  break;
    case EMPTY: result = srv_empty(); break;
    default:    printf ("dispatch: unknown command\n"); result = -1;
    }
    if (write(SRV_OUT, &result, sizeof(result)) != sizeof(result)) {
      perror ("dispatch (writing result)");
      return (-1);
    }
  }
}

/*============================================================================
 * The client's functions.
 *============================================================================*/

#define SEND_COMMAND(command)                              \
do {                                                       \
  unsigned char val = command;                             \
  if (write (CLI_OUT, &val, sizeof(val)) != sizeof(val)) { \
    perror("Requesting " # command);                       \
    return (-1);                                           \
  }                                                        \
} while (0)

#define SEND_VAR(var,func)                                 \
do {                                                       \
  if (write (CLI_OUT, &var, sizeof(var)) != sizeof(var)) { \
    perror (#func ## "(sending var)");                     \
    return (-1);                                           \
  }                                                        \
} while (0)
```

```
#define SEND_DATA(buf,nbytes,func)                            \
do {                                                          \
  if (write (CLI_OUT, buf, nbytes) != nbytes) {               \
    perror (#func ## "(sending data)");                       \
    return (-1);                                              \
  }                                                           \
} while (0)

#define CHECK(command)                                        \
do {                                                          \
  unsigned char val;                                          \
  if (read (CLI_IN, &val, sizeof(val)) != sizeof(val)) {      \
    perror("Completing " # command);                          \
    return (-1);                                              \
  }                                                           \
  return 0;                                                   \
} while (0)


int cli_empty (void)
{
  SEND_COMMAND(EMPTY);
  CHECK(EMPTY);
}

/*-------------------------------------------------------------
 * Initialize the abstraction.
 *-------------------------------------------------------------*/

int cli_init (void)
{
  SEND_COMMAND(INIT);
  CHECK(INIT);
}

/*-------------------------------------------------------------
 * Sends data to be pushed. Number of bytes followed by the
 * data itself.
 *-------------------------------------------------------------*/

int cli_push (unsigned char *buf, int nbytes)
{
  SEND_COMMAND(PUSH);
  SEND_VAR(nbytes,cli_push);
  SEND_DATA(buf,nbytes,cli_push);
  CHECK(PUSH);
}
```

```
/*----------------------------------------------------------
 * Requests data to be popped. Sends the number of bytes and
 * then reads from the pipe the data into 'buf'.
 *----------------------------------------------------------*/

int cli_pop (unsigned char *buf, int nbytes)
{
  SEND_COMMAND(POP);
  SEND_VAR(nbytes,cli_pop);
  if (read (CLI_IN, buf, nbytes) != nbytes) {
    perror ("cli_pop (reading data)");
    return (-1);
  }
  CHECK(POP);
}

/*=========================================================================*/

void main (int argc, char **argv)
{
  pid_t    pid;

  if (pipe(pfd1) < 0  ||  pipe(pfd2) < 0) {
    perror ("test-server.c: (creating pipes)");
    exit (1);
  }

  pid = fork();

  //==================== SERVER ====================//
  if (pid == 0) {
    dispatch();
    printf ("SERVER: dispatch returned\n");
    return;
  }

  //==================== CLIENT ====================//

  cli_init();

  /* Perform all the tests here */
}
```

Bibliography

Anderson T., "The Case for Application-Specific Operating Systems." In *Third Workshop on Workstation Operating Systems*, 1992, pp. 92–94.

Banerji A., Tracey J.M., and Cohn D.L., "Protected Shared Libraries – A New Approach to Modularity and Sharing." In *Proceedings of the 1997 USENIX Technical Conference*, 1997.

Bershad B., Anderson T., Lazowska E., and Levy H., "Lightweight Remote Procedure Call." In *ACM Transactions on Computer Systems*, vol. 8, 1990, pp. 37–55.

Bershad B., Savage S., Pardyak P., Sirer E., Fiuczynski M., Becker D., Eggers S., and Chambers C., "Extensibility, Safety and Performance in the SPIN Operating System." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 267–284.

Briceno H.M., 1997. *Decentralizing UNIX Abstractions in the Exokernel Architecture*. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Candea G.M. and Jones M.B., "Vassal: Loadable Scheduler Support for Multi-Policy Scheduling." In *Proceedings of the 2nd USENIX Windows NT Symposium*, to appear, 1998.

Cooper E., Harper R., and Lee P., "The Fox Project: Advanced Development of Systems Software." Tech. Rep. CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

Deutsch P. and Grant C.A., 1971. "A Flexible Measurement Tool For Software Systems." *Information Processing 71* .

Engler D., Kaashoek M., and O'Toole Jr. J., "Exokernel: an operating system architecture for application-specific resource management." In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 251–266.

Ganger G.R. and Kaashoek M.F., "Embedded inodes and explicit grouping: exploiting disk bandwidth for small files." In *Proceedings of the 1997 USENIX Technical Conference*, 1997, pp. 1–18.

Golub D., Dean R., Forin A., and Rashid R., "Unix as an application program." In *Proceedings of the USENIX 1990 Summer Conference*, 1990, pp. 87–95.

Härtig H., Hohmuth M., Liedtke J., Schönberg S., and Wolter J., "The Performance of μ-Kernel-Based Systems." In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997, pp. 66–77.

Heidemann J.S. and Popek G.J., 1994. "File-System Development with Stackable Layers." *ACM Transactions on Computer Systems* 12 (1): 58–89.

Hildebrand D., "An architectural overview of QNX." In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, 1992.

Intel, 1996. *Pentium Pro Family Developer's Manual*, vol. 3: Operating System Writer's Guide. Mt. Prospect, IL: Intel Corporation.

Kaashoek M., Engler D., Ganger G., Briceno H., Hunt R., Mazières D., Pinckney T., Grimm R., Jannotti J., and Mackenzie K., "Application performance and flexibility on exokernel systems." In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997, pp. 52–65.

Lee C., Chen M., and Chang R., "HiPEC: high performance external virtual memory caching." In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994, pp. 153–164.

Lucco S., “High-Performance Microkernel Systems.” In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994, p. 199.

Mazières D. and Kaashoek F., “Secure Applications Need Flexible Operating Systems.” In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, 1997.

Mogul J., Rashid R., and Accetta M., “The Packet Filter: An Efficient Mechanism for User-Level Network Code.” In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987, pp. 39–51.

Mössenböck H., 1994. “Extensibility in the Oberon System.” *Nordic Journal of Computing* 1 (1): 77–93.

Necula G.C. and Lee P., “Safe Kernel Extensions Without Run-Time Checking.” In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996, pp. 229–243.

Ousterhout J.K., “Why aren't Operating Systems Getting Faster as Fast as Hardware?” In *Proceedings of the Summer 1990 USENIX Conference*, 1990, pp. 247–256.

Redell D.D., Dalal Y.K., Horsley T.R., Lauer H.C., Lynch W.C., McJones P.R., Murray H.G., and Purcell S.C., 1980. “Pilot: An Operating System for a Personal Computer.” *Communications of the ACM* 23 (2): 81–92.

Rozier M., Abrossimov V., Armand F., Boule I., Gien M., Guillemont M., Herrmann F., Kaiser C., Langlois S., Léonard P., and Neuhauser W., 1988. “The CHORUS Distributed Operating System.” *Computing Systems* 1 (4): 305–370.

Small C. and Seltzer M., “VINO: an Integrated Platform for Operating Systems and Database Research.” Tech. Rep. TR-30-94, Harvard, 1994.

Wahbe R., Lucco S., Anderson T., and Graham S., “Efficient Software-Based Fault Isolation.” In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, 1993, pp. 203–216.

Wallach D.A., Engler D.R., and Kaashoek M.F., "ASHs: Application-specific handlers for high-performance messaging." In *Proceedings of ACM Communication Architectures, Protocols, and Applications (SIGCOMM '96)*, 1996, pp. 40–52.

Yarvin C., Bukowski R., and Anderson T., "Anonymous RPC: Low Latency Protection in a 64-Bit Address Space." In *Proceedings of the 1993 Summer USENIX Conference*, 1993, p. 0.

Yuhara M., Bershad B., Maeda C., and Moss E., "Efficient packet demultiplexing for multiple endpoints and large messages." In *Proceedings of the Winter USENIX Conference*, 1994, pp. 153–165.