The Exokernel Operating System Architecture

by

Dawson R. Engler

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 1998

Author . . . .
Department of Electrical Engineering and Computer Science
December 11, 1998

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

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

The Exokernel Operating System Architecture

by

Dawson R. Engler

Submitted to the Department of Electrical Engineering and Computer Science
on December 11, 1998, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

On traditional operating systems only trusted software such as privileged servers or the kernel can manage resources. This thesis proposes a new approach, the exokernel architecture, which makes resource management unprivileged but safe by separating management from protection: an exokernel protects resources, while untrusted application-level software manages them. As a result, in an exokernel system, untrusted software (e.g., library operating systems) can implement abstractions such as virtual memory, file systems, and networking.

The main thrusts of this thesis are: (1) how to build an exokernel system; (2) whether it is possible to build a real one; and (3) whether doing so is a good idea. Our results, drawn from two exokernel systems [25, 48], show that the approach yields dramatic benefits. For example, Xok, an exokernel, runs a web server an order of magnitude faster than the closest equivalent on the same hardware, common unaltered Unix applications up to three times faster, and improves global system performance up to a factor of five.

The thesis also discusses some of the new techniques we have used to remove the overhead of protection. The most unusual technique, untrusted deterministic functions, enables an exokernel to verify that applications correctly track the resources they own, eliminating the need for it to do so. Additionally, the thesis reflects on the subtle issues in using downloaded code for extensibility and the sometimes painful lessons learned in building three exokernel-based systems.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

“But I don' t want to go among mad people,” Alice remarked.

“Oh, you can' t help that,” said the Cat: “we' re all mad here. I' m mad, you' re mad.”

“How do you know I' m mad?” said Alice.

“You must be,” said the Cat, “or you wouldn' t have come here.”

- Lewis Carroll (1832-1898), **Alice In Wonderland**

Acknowledgments

The exokernel project has been the work of many people. The basic principles of Chapter 2 and the Aegis exokernel come from a paper [25] written jointly with Frans Kaashoek and James O' Toole (which descended from my master's thesis, done under Kaashoek, with ideas initiated by [26, 27]).

In contrast to Aegis, Xok has been written largely by others. Dave Mazieres implemented the initial Xok kernel. Thomas Pinckney Russell Hunt, Greg Ganger, Frans Kaashoek, and Hector Briceno further developed Xok and made ExOS into a real Unix system. Greg Ganger designed and implemented C-FFS [37] and the Cheetah web server (based in part on [49]), the two linchpins of most of our application performance numbers. Ganger and Kaashoek oversaw countless modifications to the entire system. Eddie Kohler made an enormous contribution in our write up of these results. This work, described in [48], forms the basis for Chapter 5, and as a less primary source for Chapters 2— 4.

Grateful thanks to Butler Lampson and John Guttag for serving as thesis readers when both had much more pressing claims on their time.

Greg Andrews first introduced me to research, fundamentally changing my life. The effects of his intervention still reverberate.

Frans has been an unbelievable advisor. I am extravagently lucky to have worked with him. He guides with an amazing skill, without seeming to do anything at all. His technical excellence constantly pushed me to see and think more deeply. His fundamental *goodness* has made me much better than I was.

Thanks to: Dave for volume, in all its forms; John for rescues and (laptop) wizardry; Greg for an example; Tom and Rusty for going onward; Max for science and sense; aNd EdDie TWo, for bEiNg yOu.

And special thanks to all for enduring years of being on the wrong end of vapid bromides and vacuous optimism.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

And now that the legislators and the do-gooders have so futilely inflicted so many systems upon society, may they end up where they should have begun: may they reject all systems, and try liberty... — Frederic Bastiat

Traditional operating systems (OSes) abstract and protect system resources. For example, they abstract physical memory in terms of virtual memory, disk blocks in terms of files, and exceptions and CPU in terms of processes. This organization has three significant benefits. First, it provides a portable interface to the underlying machine; applications need not care about the details of the underlying hardware. Second, it provides a large default functionality base, removing the need for application programmers to write device drivers or other low-level operating system code. Finally, it provides protection: because the operating system controls all application uses of resources, it can control application access to them, preventing buggy or malicious applications from compromising the system. Empirically, the ability to have multiple applications and users sharing the same machine is useful. Despite this organization's benefits, it has a serious problem: only privileged servers and the kernel can manage system resources. Untrusted applications are restricted to the interfaces and implementations of this privileged software. This organization is flawed because application demands vary widely. An interface designed to accommodate every application must anticipate all possible needs. The implementation of such an interface would need to resolve all tradeoffs and anticipate all ways the interface could be used. Experience suggests that such anticipation is infeasible and that the cost of mistakes is high [3, 9, 16, 25, 43, 79].

The *exokernel architecture* attacks this problem by giving untrusted application code as much safe control over resources as possible, thereby allowing orders of magnitude more programmers to innovate and use innovations, without compromising system integrity. It does so by dividing responsibilities differently from the way conventional systems do. Exokernels separate protection from management: they protect resources, applications manage them. An exokernel strives to move all functionality not required for protection out of privileged kernels and servers into unprivileged applications. For example, in the context of virtual memory,

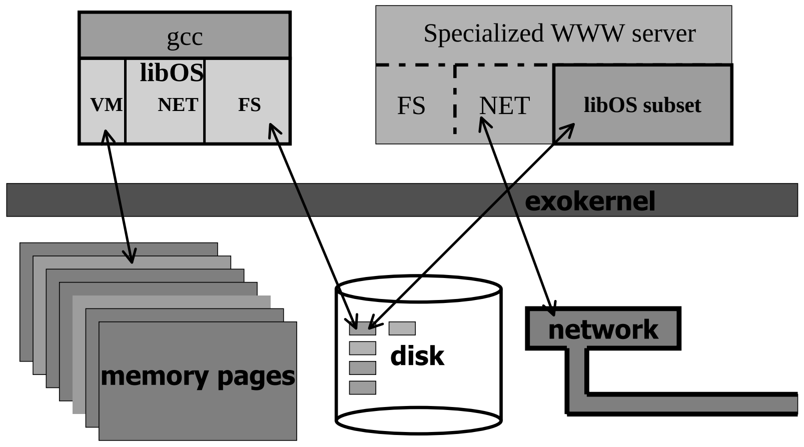

Figure 1-1: Possible exokernel system. Applications link against library operating systems (libOSes), which provide standard operating system abstractions such as virtual memory, files, and network protocols. Because libOSes are unprivileged, applications can also specialize them or write their own, as the web server in the picture has done. Because the exokernel provides protection, completely different libOSes can simultaneously run on the same system and safely share resources such as disk blocks and physical pages.

an exokernel protects physical pages and disk blocks used for paging, but defers the rest of management to applications (i.e., paging, allocation, fault handling, page table layout, etc.). The ideal exokernel makes untrusted software as powerful as a privileged operating system, without sacrificing either protection or efficiency.

Of course, not all applications need customized resource management. Instead of communicating with the exokernel directly, we expect most programs to be linked with libraries that hide low-level resources behind traditional operating system abstractions. However, unlike traditional implementations of these abstractions, library implementations are unprivileged and can therefore be modified or replaced at will. We refer to these unprivileged libraries as *library operating systems,* or libOSes. On the exokernels described in this thesis, libOSes implement virtual memory, file systems, networking, and processes. Applications are written on top of these libraries in a way similar to how they are written on top of current operating systems. As a result, applications can achieve appreciable speedups on an exokernel by simply linking against an optimized library operating system. Figure 1-1 illustrates a generic exokernel system.

An exokernel retains the three benefits of traditional operating systems: default functionality and portability come from writing applications on top of a libOS, while protection comes from the exokernel, which guards all resources. In addition, we hope that the exokernel organization, because it places OS software into an unprivileged library, will dramatically improve system innovation. We have four main reasons for this hope:

1. Fault-isolation: an error in a libOS only affects the applications using it, in contrast to errors in privileged operating systems, which can compromise the entire system. Thus, an exokernel significantly reduces the risk of using operating system innovations.

2. Co-existence: by design, multiple, possibly specialized, library operating systems can co-exist on the same exokernel system, in contrast to traditional systems, which by-and-large prevent more than one operating system running at a time. Thus, an exokernel enables innovation composition.

3. Increased implementor base: there are several orders of magnitude more systems programmers than privileged implementors. We hope the rate of innovation increases proportionally.

4. Increased user base: by making operating system software no different from other runtime libraries, the number of people with discretion to use an innovation increases by an even larger factor. Similarly, using innovations becomes a simple matter of linking in a new library rather than having to replace an entire system-wide operating system (and forcing all other users of the system to use it, and its bugs, in the process).

These four features remove many of the practical barriers facing operating system innovation development and deployment.

We do not assume that application programmers will modify operating system software as a matter of course. Instead, we regard libOS modification as similar to that of compilers: both are large, relatively complex pieces of software, not altered in the normal course of day-to-day programming. However, in the case of compilers, the enormous implementor and user community coupled with, unprivileged, fault-isolated compiler implementations has resulted in thousands of languages and implementations. For example, new languages such as Java, Tcl, Perl, and C++ have swept the implementor base every few years. Observed operating systems revolutions happen both on a more attenuated time scale, and with less dramatic scope. We hope that by making OS software more similar to compilers in the above ways that their evolution will become more similar as well.

An exokernel's success does not depend on a panoply of different operating systems. In our view, similar again to compilers and languages, there will be a few dominant operating systems but, importantly, the fact that they are unprivileged will enable them to evolve more readily than traditional systems.

1.1 Relation to other OS structures

There is a large literature on extensible operating systems, starting with the classic rationales by Lampson and Brinch Hansen [41, 53, 54]. Previous approaches to extensibility can be coarsely classified in to three groups: better microkernels, virtual machines, and downloading untrusted code into the kernel. We discuss each in turn and then relate exokernels to recent work in extensible operating systems.

The exokernel differs from traditional monolithic systems in that it places the bulk of operating system functionality in unprivileged libraries. It similarly differs from a microkernel in that, while both organizations move code out of the kernel, a microkernel pushes code into privileged servers, which applications cannot modify. In some sense, the principal goal of an exokernel—giving applications control—is orthogonal to the question of monolithic versus microkernel organization. If applications are restricted to inadequate interfaces, it makes little difference whether the implementations reside in the kernel or privileged user-level servers [39, 39]; in both cases applications lack control. For example, it is difficult to change the buffer management policy of a shared file server. In many ways, servers can be viewed as fixed kernel subsystems that happen to run in user space. Whether monolithic or microkernel-based, the goal of an exokernel system remains for privileged software to provide interfaces that do not limit the ability of unprivileged applications to manage their own resources.

Some newer microkernels push the kernel interface closer to the hardware [16, 42, 73], obtaining better performance and robustness than previous microkernels and allowing for a greater degree of flexibility, since shared monolithic servers can be broken into several servers. Several go even further and make servers unprivileged [16, 73]. However, because servers encapsulate state in a centralized way, they are “in practice” privileged. For example, on a system with a default file server, applications cannot implement their own file server and use it to share files with others using the existing server. By placing OS code in libraries, an exokernel decentralizes an abstraction's implementation, and thus allows replacement to happen seamlessly.

Techniques to reduce the cost of shared servers by improving IPC performance, moving code from servers into libraries, mapping read-only shared data structures, and batching system calls [8, 39, 58, 61] can also be successfully applied in an exokernel system.

Hydra was the most ambitious early system to have the separation of kernel policy and mechanism as one of its central tenets [94]. An exokernel takes the elimination of policy one step further by removing “mechanism” wherever possible. This process is motivated by the insight that mechanism *is* policy, albeit with one less layer of indirection. For instance, a page-table is a very detailed policy that controls how to translate, store and delete mappings and what actions to take on invalid addresses and accesses. [1]

Virtual machines [12, 34, 38] (VMs) are an OS structure in which a privileged virtual machine monitor (VMM) isolates less privileged software in emulated copies of the underlying hardware. Virtual machines and exokernels differ in two main ways. First, virtual machines emulate, exokernels do not. Emulation hides information. This can lead to ineffective use of hardware resources; for instance, the VMM has no way of

[1]The OS community has no rigorous definition of either “policy” or “mechanism.” In its most general sense, policy refers to the goals of a computer system (e.g., what security threats to resist, what resources to protect). In context of extensible operating systems, policy refers to the algorithm used to make security or resource management decisions, while mechanism refers to the machinery used to implement a particular policy. For example, a virtual memory paging policy is to evict least recently used pages to disk. The data structures used to do so, such as page tables and a sorted page list, would be mechanism. Similar to the concepts of code and data, there is no clear fundamental difference between these two notions.

knowing if a VM no longer needs a particular virtual page. In contrast, an exokernel attempts to expose all information about the system and explicitly communicates with the library operating system rather than presenting a virtual facade and making its own resource management decisions. Second, VMs can only share resources through remote communication protocols. This prevents VMs from sharing many OS abstractions such as processes or file descriptors with each other. Thus, VMMs confine specialized operating systems and associated processes to isolated virtual machines. Rather than partitioning the machine into disjoint pieces, an exokernel allows non-trusting applications to use customized libOSes to share resources (such as physical memory and disk blocks) without sacrificing a single view of the machine,

Downloading code into the kernel is another approach to extensibility. In many systems only trusted users can download code, either through dynamically-loaded kernel extensions or static configuration [33, 43]. In the SPIN and Vino systems, any user can safely download code into the kernel [9, 78]. Safe downloading of code through type-safety [9, 75] and software fault-isolation [78, 89] is complementary to the exokernel approach of separating protection from management. Exokernels use downloading of code to let the kernel leave decisions to untrusted software [25].

In addition to these structural approaches, much work has been done on better OS abstractions that give more control to applications, such as user-level networking [82, 88], lottery scheduling [90], application-controlled virtual memory [44, 55] and file systems [13, 72]. All of this work is directly applicable to libOSes.

1.1.1 Recent extensible operating systems

SPACE is a "submicro-kernel" that provides only low-level kernel abstractions defined by the trap and architecture interface [73]. Its close coupling to the architecture makes it similar in many ways to an exokernel, but we have not been able to make detailed comparisons because its design methodology and performance have not yet been published.

The SPIN project is building a microkernel system that allows applications to make policy decisions [9] by safely downloading *extensions* into the kernel. Unlike SPIN, the focus in the exokernel architecture is to obtain flexibility and performance by securely exposing low-level hardware primitives rather than extending a traditional operating system in a secure way. As a result, exokernel interfaces tend to be lower level and, thus, grant more control to applications.

Anderson [3] makes a clear argument for application-specific library operating systems and proposes that the kernel concentrate solely on the adjudication of hardware resources. The exokernel design addresses how to provide secure multiplexing of physical resources in such a system, and moves the kernel interface to a lower level of abstraction. In addition, our exokernel systems demonstrate that low-level secure multiplexing and library operating systems can offer excellent performance.

Like an exokernel, the Cache Kernel [16] provides a low-level kernel that can support multiple application-level operating systems. To the best of our knowledge the Cache Kernel and ExOS, the libOS used on three generations of exokernels [25, 48], are the first general-purpose library operating systems implemented in a multiprogramming environment. The difference between the Cache Kernel and an exokernel is mainly one of high-level philosophy. The Cache Kernel focuses primarily on reliability, rather than securely exporting hardware resources to applications. As result, it is biased towards a server-based system structure. In our experience, servers become de facto privileged in that their functionality cannot be overridden by applications.

The Nemesis kernel [56, 76] has many similarities to an exokernel, despite a vast difference in goals. Nemesis is designed to improve quality-of-service for multimedia applications. The problem it attacks is that traditional systems make resource accounting difficult in that code running on behalf of an application may be strewn throughout a collection of servers and the kernel. Like an exokernel, their solution is to push operating system functionality into applications. In this way, code running on behalf of the application typically runs within the application itself, which then can trivially be charged for its resource consumption. The primary parallels between Nemesis and an exokernel system are low-level interfaces for resources needed by multimedia applications (events, network, disk, and CPU), and a reliance on library operating systems. However, the difference in goals leads to stark contrasts. While Nemesis is designed to simplify accounting, an exokernel aims to improve innovation. It does so by ceding *all* control not needed for protection to applications. Thus, an exokernel's interfaces grant more pervasive power to applications and also promote sharing of resources (so that different implementations can safely share the same state). For example, applications have only limited control over virtual memory, since Nemesis forces a single-address space on the entire system. Similarly, Nemesis disk multiplexing lacks the power of our disk subsystem, XN (discussed in Chapter 4). We know of no experimental results for Nemesis, which prevents us from performing a more detailed comparison of the relative strengths of the two approaches.

1.2 The focusing questions of this thesis

An exokernel attempts to make unprivileged library operating systems as powerful as privileged operating systems. The main thrusts of this thesis are:

1. How to build an exokernel system.
2. Whether it is possible to build a real one.
3. Whether doing so is a good idea.

The first half of the thesis, Chapters 2 and 4, focuses on how to build an exokernel system, and the later chapters on the approach's efficacy.

The text below provides an overview of the questions each issue raises, along with a sketch of their associated answers.

1.2.1 How to build an exokernel?

Traditionally, operating systems have provided a high-level interface to unprivileged software. Chapter 2 provides a constructive methodology for how to lower this interface to a level sufficient for libOSes to build abstractions such as networking, virtual memory and file systems. The six principles of this methodology are: (1) separate management from protection; (2) expose all hardware to applications; (3) expose resource allocation, placing it under the discretion of applications; (4) expose revocation, thereby letting applications determine what resources to relinquish; (5) protect at a fine-grained level, making sharing and management lightweight; and (6) expose information, such as hardware capabilities, physical names, and global system statistics.

A key aspect of exokernel design is leaving implementation decisions to the client. Rather than focusing design on deciding how to implement a policy or mechanism, focus on constructing an interface that leaves all interesting decisions to applications. An interface that merely checks that an application has performed an operation correctly frequently gives greater freedom for important decisions, as well as being simpler to implement. In some sense, exokernel design focuses on the art of transmuting the imperative (deciding how to implement a policy or mechanism in the kernel) to the declarative (specifying *what* interface an application must implement, and checking that it does so correctly).

Chapter 3 contains a number of examples of how to apply this methodology, and Chapter 4 discusses an extended example of how to multiplex a hardware disk, which has been the most challenging resource for us to handle.

The following four subsections discuss important subproblems of exokernel construction.

How to recapture resource semantics?

By dislodging OS code into libraries, an exokernel also ejects a significant portion of the code that understands resource semantics. For example, in the context of networking, because the exokernel dislocates network protocol code (e.g., TCP/IP and UDP) into untrusted libraries, it no longer understands packet semantics. As a result, it lacks the information necessary to decide which application owns what message: a decision it must make if it is to implement protection.

We have used downloaded code as a powerful mechanism to allow untrusted software to convey these semantics back into the exokernel in a general way. In the context of networking we use packet filters (see Chapter 2), for disk, deterministic meta data interpreters (see Chapter 4).

From a different perspective, an exokernel, by "uploading" code into the application, removes the need to protect many pieces of state, and, thus, trivially need not understand their semantics.

Chapter 6 reflects on our experience using downloading code: when downloaded code was superior (or inferior) to a functional interface, our mistakes, and a number of surprises, visible only in much delayed hindsight. This technique has subtle implications, resulting from, among other things: the asymmetry in trust between the extension and the host; the fact that code for most useful languages is Turing complete, while most procedural interfaces are decidedly not; and that downloaded code, because it can be restricted, can be granted abilities that unrestricted external application code cannot. A consequence of this latter attribute is that most of our uses of downloaded code have little to do with speed but rather with granting applications power otherwise not possible.

How to protect shared state?

Traditional operating systems encapsulate and enforce well-formed updates to state shared amongst processes. For example, a Unix kernel performs modifications on shared file descriptors, the file name cache, the buffer cache, etc. In contrast, an exokernel dislocates such state into unprivileged applications, which then modify it, potentially incorrectly. How then can invariants on shared state be enforced? We have used a plethora of techniques to enforce this.

The most general solution is to apply the exokernel precepts recursively: divide libraries into privileged and unprivileged parts, where the "privileged" part contains all code required for protection, and must be used by all applications using a piece of state, while the unprivileged contains all management code and can be replaced by anyone. Forcing applications to use the privileged portion of a libOS can be done by placing it in a server, using a restricted language and trusted compiler, or downloading code into the kernel. We have used all three.

In the more common case, library operating systems can frequently be designed for localization of state and to use standard fault-isolation techniques (such as type-safe languages and memory protection).

In many cases, the desire to protect shared state with mechanisms more elaborate and read and write access checks makes little technical sense: if the raw data itself can be corrupted, it is unlikely that there is any reason to preserve high-level invariants on the resultant garbage. One of the most common situations where this dynamic arises is in the protection of shared bookkeeping data structures. Consider the case of a shared buffer (say a Unix pipe), that uses a buffer record to track the number of bytes in the buffer. At one level, one would like to guarantee that an application decrements the byte count on data removal, and increments it on insertion. However, while this desire is sensible from a software engineering perspective, it is not a protection issue. Since there are no guarantees on the sensibleness of the inserted data, knowing how many bytes of garbage are in the buffer gains no security. In practice, social, rather than technical, methods guarantee these sort of invariants — e.g., assemblers adhere to standard object code formats rather than going through a set of system enforced methods for laying out debugging and linking information.

How to protect without overhead?

An exokernel adds another layer to the system in that it takes operating system code, which formerly ran on bare hardware, and places it in an unprivileged library, which runs on top of an exokernel which runs on bare hardware. Since performance motivates exokernels, the cost of this extra layer must be negligible.

Fortunately, for most resources, this overhead has been a non-issue. Exokernel implementation of virtual memory, networking, exceptions, and process management have all performed well, without aggressive tuning. In fact, on the first exokernel system we built, Aegis [25], if an exokernel primitive did not perform significantly faster — typically a factor of five to tenfold better — than that of a traditional system from the beginning, we looked for “what was wrong.”

In practice, protection does not impose overhead. As we discuss in Chapter 5, it tends to be off of the critical path, coupled to expensive operations that dwarf its overhead, or recovered by other features of the architecture (e.g., libOSes make many system calls into function calls).

Can applications be trusted to track ownership?

At its most basic level, protection requires a table lookup to map a principal to access rights. While maintaining this table adds little overhead for most resources, for others, such as a disk, it is infeasible. Chapter 4 presents the reasons for this in detail.

This problem's solution comes from noticing that correct applications track what resources they have access to, rendering operating system bookkeeping redundant. For example, a library file system tracks the disk blocks each file maps to. Thus, if the operating system can reuse the application's data structures, it can eliminate this redundancy. In the case of file systems, if the exokernel can reuse the library file system's “meta data” (the data structures it uses to map files to blocks), then it need not perform duplicate bookkeeping. We have invented an online verification technique that lets an exokernel verify that library file systems correctly track what disk blocks they own, without the operating system having to understand how they do so.

1.2.2 Can you build a real exokernel system?

Does the exokernel organization only work for relatively simple, isolated resources such as physical memory, or can it apply to more difficult shared resources such as disk? Can one build an exokernel system? Fortunately for this thesis, one can. There have been three exokernel systems built so far, in increasing verisimilitude: Aegis [25], Glaze [60], and Xok [48]. Most of our performance data and examples come from Xok and ExOS, its default libOS. While ExOS does not handle some Unix corner cases, it is not a toy either. For example, it runs most Unix applications (e.g., perl, gcc, telnet, and csh) without modification. This fact can be seen in that the bulk of our performance data comes from application end-to-end numbers rather than micro-benchmarks.

1.2.3 Are exokernels a good idea?

Chapter 5 (and to a lesser degree, Chapter 3) focuses primarily on evaluating exokernel efficacy. We use a performance driven approach; our experiments address four questions, discussed below.

Do common, unaltered applications benefit?

If an exokernel only improves the performance of strange niche applications or requires that applications be modified to benefit, then its usefulness is severely diminished.

Our experiments show that an exokernel matters, even for common applications. We measure the performance of a software development workload made up of mainstream applications (most are found in "/usr/bin" on a Unix system). When linked against an optimized library file system and run on our exokernel system, these programs run up to three times faster than identical versions executed on competitive monolithic operating systems.

In general, normal applications benefit from an exokernel by simply linking in an optimized libOS that implements a standard interface. Thus, even without modifications, they still profit from an exokernel. [2] While an exokernel allows experimentation with completely different OS interfaces, a more important result may be improving the rate of innovation of implementations of existent interfaces.

Does a libOS run slower than an OS?

An exokernel provides extreme flexibility. Rightfully, any systems builder would expect that this flexibility would have a performance cost. In particular, given the same application code and same OS code (placed in a libOS on an exokernel, in the kernel on a monolithic system), does a traditional system provide superior performance? Or, phrased another way, if an optimization is done on an exokernel and gives a factor of four, would the same optimization done on a traditional OS give even more?

To partially answer this question we have taken the library file system discussed in the previous experiment and re-implemented it in the kernel of a traditional, monolithic operating system. We then run the same workload on it. Our results show that, at least in this case, an exokernel does not pay for its structure. Or, in the alternative way, that an optimization done on an exokernel can give the same performance improvement as done on a traditional system.

Are aggressive applications ten times faster?

In part, the exokernel architecture was motivated by the ability to perform application-specific or, more commonly, domain-specific optimizations. [3] Two natural questions, then, are first, does an exokernel give sufficient power that interesting optimizations can be done? Second, do domain-specific optimizations yield significant improvements or do they only give "noise" level improvements?

Experiments show that an exokernel enables domain-specific optimizations that give order-of-magnitude performance improvements [48]. Our specific results come from an interface designed for fast I/O, which improves the performance of applications such as web servers.

[2] In principle, given sufficient resources a traditional OS can implement any innovation done on an exokernel. However, in practice, we expect an exokernel system to evolve significantly faster than traditional systems.

[3] In fact, the original exokernel paper [25] focuses almost exclusively on application-specific optimizations rather than improving the rate of general-purpose innovations, which we have come to regard as at least as important.

Does local control lead to bad global performance?

An exokernel gives applications significantly more control than traditional operating systems do. A positive aspect of this power transfer is that applications can perform optimizations not previously possible. A negative aspect is that their selfish optimization efforts could destroy system performance. Thus a key question is: can an exokernel reconcile local control with good global performance?

It can, for two main reasons. First, when an exokernel structure enables improved application performance there will be more resources to go around, and thus, global performance will improve. Second, an exokernel mediates allocation and revocation of resources. Therefore it has the power to enforce any global policy that a traditional operating system can. The single new challenge it faces is deriving information lost by dislocating abstractions into application space. For example, traditional operating systems manage both virtual memory and file caching. As a result, they can perform global resource management of pages that takes into account the manner in which a page is being used. In contrast, if an exokernel dislocates virtual memory and file buffer management into library operating systems it no longer can make such distinctions. While such information matters in theory, in practice we have found it either unnecessary or crude enough that no special methods have been necessary to derive it. However, whether this happy situation always holds is an open question.

More concretely, our experiments demonstrate that: (1) given the same workload, an exokernel performs comparably to widely used monolithic systems, and (2) that when local optimizations are performed, that whole system performance improves, and can do so significantly.

In summary, the experiments show that, compared to a traditional system, an exokernel provides comparable to significantly superior performance while simultaneously giving tremendous flexibility.

1.3 Concerns

Below, we discuss five concerns about the exokernel architecture: (1) that an it will compromise portability, (2) that it will lead to a Babel of incompatible libOSes, (3) that libOSes are too costly to specialize, (4) that they consume too much space per application, and (5) that the modification of a “free software” OS provides a “good enough” way to innovate.

First, it is a virtue for an exokernel to expose the details of the system's hardware and software. Our exokernels expose information such as the number of network and scatter/gather DMA buffers available and TLB entry format. Naively handled, this exposure can compromise portability. Fortunately, traditional techniques for retaining portability work just as well on exokernel systems. While applications that use an exokernel interface directly will not be portable, those that use library operating systems that implement standard interfaces (*e.g.*, POSIX) are portable across any system that provides the same interface. Library operating systems themselves can be made portable by designing them to use a low-level machine-independent layer to hide hardware details. This technique has been widely used in the context of more traditional operating systems [21, 74].

As in traditional systems, an exokernel can provide backward compatibility in three ways: one, binary emulation of the operating system and its programs; two, implementing its hardware abstraction layer on top of an exokernel; and three, re-implementing the operating system's abstractions on top of an exokernel.

Second, since library operating systems are unprivileged, anyone can write one. An obvious problem is what happens to system coherence under an onslaught of “hand rolled” operating systems? Similar to the previous challenge, applications already solve the problem of chaos from freedom through the use of standards and good programming practices. The advantage of standards on paper rather than hard wired in the kernel is that they can be re-implemented and specialized, selected amongst, or, in the case of the worst ones, ignored.

Third, does specialization require writing an entire library operating system? Our experience shows that it does not. All of the application performance improvements in this thesis, even the most dramatic, were achieved by inserting the specialized subsystem within the default libOS, rather than rewriting an entire libOS from scratch. Given a modular design, other library operating systems should allow similar specialization. It is possible that object-oriented programming methods, overloading, and inheritance can provide useful operating system service implementations that can be easily specialized and extended, as in the VM++ library [51].

Most libOS code is no more difficult to modify than other systems software such as memory allocators. It tends to be conceptually shallow, and concerned chiefly with managing various mappings (page tables, meta data, file tables, etc.). As such, once operating system software is removed from the harsh environment of the kernel proper, modification does not require extraordinary competence.

Fourth, if each application has its own library operating system linked into it, what about space? As in other domains with large libraries, exokernel systems can use dynamic linking and shared libraries to reduce space overhead. In our experience, these techniques reduce overheads to negligible levels. For example, our global performance experiments place a severe strain on system memory resources, yet an exokernel is comparable to or surpasses the conventional systems we compare against.

Finally, what about Linux, FreeBSD, and the rest of the public-domain operating systems? Do they not evolve as quickly as an exokernel will? We believe that an exokernel structure has important software engineering benefits over these systems. First, libraries are fault-isolated from each other, allowing different operating systems to co-exist on the same system, something not possible with the current crop of operating systems. Second, library development is much much easier than kernel hacking. For example, standard debugging tools work, and errors crash only the application using the library instead of the system, allowing easy post-mortem examination. Third, all users have the power to use an innovation, simply by linking an application against a libOS.

1.4 Summary

The exokernel in a nutshell:

1. What: anyone can safely manage resources.

2. How: separate protection from management. An exokernel protects resources, applications manage them. Everything not required for protection is moved out of privileged kernels and servers into untrusted application libraries. This is the one idea to remember for an exokernel, all other features are details derived from it.

   The exokernel ideal: the libOS made as powerful as privileged OS, without sacrificing performance or protection.

3. Why? Innovation. An exokernel makes operating systems software co-exist, unprivileged, and modifiable by orders of magnitude more programmers.

The following chapter articulates the principles of the exokernel methodology. Chapter 3 draws on examples from two exokernel systems, Aegis and Xok, to show how these principles apply in practice. The most challenging problem for protection, disk multiplexing, is discussed in detail in Chapter 4. Chapter 5 presents application performance results that show that that such separation is profitable. We reflect on using downloaded code for extensibility in Chapter 6. Finally, Chapter 7 discusses future work, reflects on general lessons learned while building exokernel systems, and concludes.

Chapter 2

How to build an exokernel: principles

It is impossible to design a system so perfect that no one needs to be good. — T. S. Eliot:

The goal of an exokernel is to give as much safe, efficient control over system resources as possible. The challenge for an exokernel is to give library operating systems maximum freedom in managing resources while protecting them from each other; a programming error in one library operating system should not affect another library operating system. To achieve this goal, an exokernel separates protection, which must be privileged, from management, which should be unprivileged.

Figure 2-1 shows a simplified exokernel system that is running two applications: an unmodified UNIX application linked against the ExOS libOS and a specialized exokernel application using its own TCP and file system libraries. Applications communicate with the kernel using low-level physical names (e.g., block numbers); the kernel interface is as close to the hardware as possible. LibOSes handle higher-level names (e.g., file descriptors) and supply abstractions. Protection in this figure consists of forcing applications to only read or write (cached) disk blocks that they have access rights to. The kernel does this by guarding all operations on both the disk blocks themselves and on the physical pages that hold cached copies.

In general, resource protection consists of three tasks: (1) tracking ownership of resources, (2) ensuring protection by guarding all resource usage or binding points, and (3) revoking access to resources. To prevent rogue applications from destroying the system, most exokernels protect physical resources such as physical memory, the CPU, and network devices (to prevent theft of received messages and corruption of not-yet-transmitted ones). Additionally, they must guard operations that change privileged state such as the ability to execute privileged instructions, and writes to “special” memory locations that control devices. Finally, they may guard more abstract resources such as bandwidth.

To make this discussion concrete, consider the protection of physical pages. The core requirement is that the exokernel must check that every read and write to any page has sufficient access rights. On modern architectures, encapsulation involves forcing all memory operations to either go through the kernel (which checks permissions manually) or through translation hardware (the TLB), which checks them against a set of

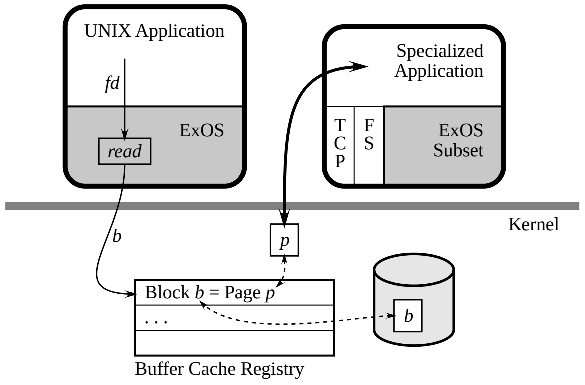

Figure 2-1: A simplified exokernel system with two applications, each linked with its own libOS and sharing pages through a buffer cache registry.

cached pages. To ensure that malicious processes do not corrupt the TLB, the kernel must check modifications of it as well. In practice this means that applications cannot issue privileged instructions directly, but must go through the kernel. The next three chapters give many examples of resources to protect and ways to do so.

Applications cannot override privileged software. This inflexibility makes protection possible. However, if it spills over into non-protection areas, applications will needlessly lose the ability to control important decisions. Thus, an exokernel overrides application decisions only to the extent necessary to build a secure system.

This chapter describes five principles that give an exokernel's declarative goal of separating protection from management operational teeth. These principles illustrate the mechanics of exokernel systems and provide important motivation for many design decisions discussed later in this thesis. To better understand what the architecture is, we then show what it is not. The remainder of the chapter discusses how an exokernel protects the state of higher-level abstractions, and performs resource revocation, and then presents a general technique, *secure bindings*, we have used to make protection efficient.

2.1 Exokernel principles

The goal of an exokernel is to give efficient control of resources to untrusted applications in a secure, multi-user system. We follow these principles to achieve this goal:

Separate protection and management. Exokernels restrict resource management to functions necessary for protection: allocation, revocation, sharing, and the tracking of ownership. Giving applications control over all non-protection mechanisms and policies makes the system “optimally” extensible, in that any further application customization would compromise system integrity.

In general, an exokernel strives to provide applications access to all operations that a privileged OS has. This requires providing ways for libOSes to recapture aspects of the kernel's privileged execution context, such as being tightly coupled to hardware interrupts, which they have lost by running as application software.

(From another perspective, the privileged exokernel adds a layer of indirection between a libOS and hardware events. The exokernel must take steps to ensure that this layer does not preclude flexible application control of hardware.)

Expose hardware. Exokernels give applications protected access to all resources. Applications have access to privileged instructions, hardware DMA capabilities, and machine resources. The resources exported are those provided by the underlying hardware: physical memory, the CPU, disk memory, translation look-aside buffer (TLB), and addressing context identifiers. Applications have full view of resource attributes, such as the number, format, and current set of TLB mappings or the number and size of incoming and outgoing network buffers, as well as the ability to modify them (inserting TLB entries). This principle extends to less tangible machine resources such as interrupts, exceptions, and cross-domain calls.

An exokernel exposes resources and events at the lowest possible level required for protection—ideally, at the level of hardware (disk blocks, physical pages, etc.) rather than encapsulating them in high-level abstractions such as files, or Unix signal or RPC semantics.

The following two principles fall from a general pattern: involve applications in important decisions rather than subjecting them to a hard wired policy.

Expose allocation. Applications allocate resources explicitly. The kernel allows specific resources to be requested during allocation, giving applications control over abstraction semantics and performance decisions, such as when and which disk blocks to allocate.

Expose revocation. Exokernels expose revocation policies to applications. They let applications choose which instance of a resource to give up. Each application has control over its set of physical resources. Exokernels allow applications to revoke resources from processes they control, thereby allowing enforcement of local policies.

Protect fine-grained units. To make resource management and sharing lightweight, exokernel protection is fine-grained (e.g., at the level of disk blocks rather than partitions). Fine-grained protection reduces the overhead of mapping high-level abstractions down to the discrete resource units that an exokernel protects (e.g., mapping files to disk blocks, virtual memory to pages, network messages to message buffers, etc.). Coarse-grained protection wastes space as well as time. For example, in the context of disk, by increasing the length of disk arm sinks after increasing internal fragmentation through coarse-grained allocation. Coarse-grained protection also makes sharing clumsy, since resources cannot be shared at a finer-granularity than the exokernel protects.

Expose names. Exokernels use physical names wherever possible. Physical names capture useful information and do not require potentially costly or race-prone translations from virtual names. For example, physical disk blocks encode position, one of the most important variables for file system performance. Virtual names, in contrast, would not necessarily encode this information and would require on-disk translation tables. Using these tables increases disk accesses (extra writes for persistence, extra reads for name translation) and protecting them across reboots degrades performance (e.g., by ordering writes to disk, and causing multiple writes).

In practice, physical names also provide a uniform mechanism for optimistic synchronization. They allow locks to be replaced with checks that “expected” conditions hold (that a directory name maps to a specific disk block and has not been deleted or moved, etc.).

Expose information. Exokernels expose system information, and collect data that applications cannot easily derive locally. For example, applications can determine how many hardware network buffers there are or which pages cache file blocks. An exokernel might also record an approximate least-recently-used ordering of all physical pages, something individual applications cannot do without global information. Additionally, exokernels export book-keeping data structures such as free lists, disk arm positions, and cached TLB entries so that applications can tailor their allocation requests to available resources.

We have found it useful in practice to provide space for protected application-specific data in kernel data structures. For example, the structures describing an execution environment are improved if application-specific data can be associated with them (e.g., to record the numeric id of an process' parent, its running status, program name, etc.).

These principles apply not just to the kernel, but to any component of an exokernel system. Privileged servers should provide an interface boiled down to just what is required for protection.

2.1.1 Policy

An exokernel hands over resource policy decisions to library operating systems. Using this control over resources, an application or collection of cooperating applications can make decisions about how best to use these resources. However, as in all systems, an exokernel must include policy to arbitrate between competing library operating systems: it must determine the absolute importance of different applications, their share of resources, *etc.* This situation is no different than in traditional kernels. Appropriate mechanisms are determined more by the environment than by the operating system architecture. For instance, while an exokernel cedes management of resources to library operating systems, it controls the allocation and revocation of these resources. By deciding which allocation requests to grant and from which applications to revoke resources, an exokernel can enforce traditional partitioning strategies, such as quotas or reservation schemes. Since policy conflicts boil down to resource allocation decisions (*e.g.,* allocation of seek time, physical memory, or disk blocks), an exokernel handles them in a similar manner.

2.2 Kernel support for protected abstractions

Many of the resources protected by traditional operating systems are themselves high-level abstractions. Files, for instance, consist of meta data, disk blocks, and buffer cache pages, all of which are guarded by access control on high-level file objects. While exokernels allow direct access to low-level resources, exokernel systems must be able to provide UNIX-like protection, including access control on high-level objects where required for security. One of the main challenges in designing exokernels is to find kernel interfaces that allow such higher-level access control without either mandating a particular implementation or hindering application control of hardware resources.

We have met this challenge by providing both protection machinery in the kernel and ways for applications to download code to augment this machinery. In the former category, our exokernels provide provides software abstractions to bind hardware resources together. For example, as shown in Figure 2-1, the Xok buffer cache registry binds disk blocks to the memory pages caching them. Applications have control over physical pages and disk I/O, but because the exokernel protects the binding of disk block to page, can also safely use each other's cached pages. Our exokernel's protection mechanism guarantees that a process can only access a cache page if it has the same level of access to the corresponding disk block.

More generally, by allowing applications to download code, an exokernel gives them a mechanism to encapsulates an abstraction's state and can thereby enforce invariants on it. This strategy is required for abstractions whose protection does not map to hardware abstractions. For example, files may require valid updates to their modification times. Our oldest use of downloaded code is packet filters. Packet filters are application-written code fragments that applications download into the kernel to select what incoming network packets they want.

However, while these software abstractions reside in the kernel, they could also be implemented in trusted user-level servers. This microkernel organization would cost additional (potentially expensive) context switches. Furthermore, partitioning functionality in user-level servers tends to be more complex.

From one perspective, downloaded code provides a conduit for pushing semantics across the user-kernel trust boundary. This activity is more important for exokernels than traditional systems. By dislodging OS code into libraries, an exokernel also removes the code that understands resource semantics and, thus, lose the ability to protect these resources. For networking, because an exokernel ejects the code that understands network protocol semantics it no longer understands how to bind incoming packets to applications. Packet filters provide a way to recapture these semantics by encapsulating them in a black box (the filter), which is then injected into the kernel. The exokernel thus trades an intractable problem — anticipating all possible invariants that must be enforced — for one that is tractable: testing the downloaded code for trustworthiness (in this case, that one filter will not steal another's packets). Chapter 6 discusses this perspective in more detail.

The key to these exokernel software abstractions is that they neither hinder low-level access to hardware resources nor unduly restrict the semantics of the protected abstractions they enable. Given these properties, a kernel software abstraction does not violate the exokernel principles.

Protected sharing

The low-level exokernel interface gives libOSes enough hardware control to implement all traditional operating system abstractions. Library implementations of abstractions have the advantage that they can trust the applications they link with and need not defend against malicious use. The flip side, however, is that a libOS cannot necessarily trust all other libOSes with access to a particular resource. When libOSes guarantee invariants about their abstractions, they must be aware of exactly which resources are involved, what other processes have access to those resources, and what level of trust they place in those other processes.

As an example, consider the semantics of the UNIX fork system call. It spawns a new process initially identical to the currently running one. This involves copying the entire virtual address space of the parent process, a task operating systems typically perform lazily through copy-on-write to avoid unnecessary page copies. While copy-on-write can always be done in a trusted, in-kernel virtual memory system, a libOS must exercise care to avoid compromising the semantics of fork when sharing pages with potentially untrusted processes. This section details some of the approaches we have used to allow a libOS to maintain invariants when sharing resources with other libOSes.

Our latest exokernel, Xok, provides three mechanisms libOSes can use to maintain invariants in shared abstractions. First, *software regions*, areas of memory that can only be read or written through system calls, provide sub-page protection and fault isolation. Second, Xok allows on the-fly-creation of *hierarchically-named capabilities* and requires that these capabilities be specified explicitly on each system call [62]. Thus, a buggy child process accidentally requesting write access to a page or software region of its parent will likely provide the wrong capability and be denied permission. Third, Xok provides robust critical sections: inexpensive critical sections that are implemented by logically disabling software interrupts [10]. Using critical sections instead of locks eliminates the need to trust other processes to follow a correct locking discipline.

Three levels of trust determine what optimizations can be used by the implementation of a shared abstraction.

Optimize for the common case: Mutual trust. It is often the case that applications sharing resources place a considerable amount of trust in each other. For instance, any two UNIX programs run by the same user can arbitrarily modify each others' memory through the debugger system call, ptrace. When two exokernel processes can write each others' memory, their libOSes can clearly trust each other not to be malicious. This reduces the problem of guaranteeing invariants from one of security to one of fault-isolation, and consequently allows libOS code to resemble that of monolithic kernels implementing the same abstraction.

Unidirectional trust. Another common scenario occurs when two processes share resources and one trusts the other, but the trust is not mutual. Network servers often follow this organization: a privileged process accepts network connections, forks, and then drops privileges to perform actions on behalf of a particular user. Many abstractions implemented for mutual trust can also function under unidirectional trust with only slight modification. In the example of copy-on-write, for instance, the trusted parent process must retain exclusive control of shared pages and its own page tables, preventing a child from child making copied pages writable in the parent. While this requires more page faults in the parent, it does not increase the number of page copies or seriously complicate the code.

Mutual distrust. Finally, there are situations where mutually distrustful processes must share high-level abstractions with each other. For instance, two unrelated processes may wish to communicate over a UNIX domain socket, and neither may have any trust in the other. For OS abstractions that can be shared by mutually distrustful processes, the most general solution is to apply the exokernel precepts recursively: divide libraries into privileged and unprivileged parts, where the "privileged" part contains all code required for protection, and must be used by all applications using a piece of state, while the unprivileged contains all management code and can be replaced by anyone. Forcing applications to use the privileged portion of a libOS can be done by placing it in a server, using compiler techniques, or downloading code it into the kernel. We have used all three.

Fortunately, sharing with mutual distrust occurs very infrequently for many abstractions. Many types of sharing occur only between child and parent processes, where mutual or unidirectional trust almost always holds.

2.3 Implementation-defined Decisions

The exokernel architecture defines a family of implementations. To understand it, it helps to understand what an exokernel is not. The architecture leaves the following five decisions to the system designer.

What to protect. While it is hard to imagine a usable system that elides memory and disk protection, the exokernel architecture does not mandate what should be protected. It only says that once this decision has been made, that all functionality not needed for protection should be implemented by untrusted software. Example resources that our exokernel implementations deliberately do not protect are: bandwidth quality of service guarantees, deadline guarantees, or any number of restrictions required by real time systems.

What global system policies to implement. Every system includes global policies enforced on all applications that decide which processes to revoke resources from and which requests to grant or deny (even if this policy is no policy at all). What specific global policy is enforced — whether it be optimized for interactive performance, throughput, real time, etc. — is orthogonal to the architecture. However, given a policy, an exokernel designer strives to involves application decisions in its implementation.

How to protect. While we provide examples of how our exokernel systems have multiplexed memory, disk, network, etc., these ways are not the only ones. Again, what is important in an exokernel is allowing untrusted software to implement everything not needed for protection; the designer has discretion in how he achieves this. This decision includes how to track access rights: our first exokernel used self-authenticating capabilities [15], then access control bitmaps [16], our second hierarchical capabilities [63].

The level of protection. An exokernel interface may contain high-level primitives. The exokernel principles are guides for *how* to give control to applications, and are only applicable after it has been determined *what* level of control is allowable.

In particular, state shared between applications can require fairly high-level semantic guarantees and, hence, any protected interface will be high-level. For example, processes sharing a Unix-flavor file system may require that file modification times be accurate, which involves ensuring disk operations modify file access times. An exokernel is about protection. If such protection is required, providing it in the kernel does not violate exokernel precepts. What the exokernel does say is that non-protection related functionality should be migrated to unprivileged software. In addition, applications not needing these file modification guarantees should not be forced to use them. They should be able to to access the low-level disk in order to implement an alternative file system.

How the kernel is organized. It is easy to read the exokernel principles as an excessive concern for what is in the kernel proper. This is not what is intended. The central question of an exokernel is not about how to organize privileged code. It does not matter whether an exokernel is implemented as a monolithic kernel or as a collection of trusted processes around a micro-kernel. Rather the architecture's central question is how how to make traditionally privileged code unprivileged by limiting the duties of the kernel to just these required for protection.

Acceptable default functionality. An exokernel may provide significant default functionality as long as it can be overridden without compromising either protection or efficiency.

2.4 Visible Resource Revocation

Once resources have been bound to applications, there must be a way to reclaim them and break their bindings. Revocation can either be *visible* or *invisible* to applications. Traditionally, operating systems have performed revocation invisibly, deallocating resources without application involvement. For example, with the exception of some external pagers [2, 77], most operating systems deallocate (and allocate) physical memory without informing applications. This form of revocation has lower latency and is simpler than visible revocation since it requires no application involvement. Its disadvantages are that library operating systems cannot guide deallocation and have no knowledge that resources are scarce.

An exokernel uses visible revocation for most resources. Even the processor is explicitly revoked at the end of a time slice; a library operating system might react by saving only the required processor state. For example, a library operating system could avoid saving the floating point state or other registers that are not live. However, since visible revocation requires interaction with a library operating system, invisible revocation can perform better when revocations occur very frequently. Processor addressing-context identifiers are a stateless resource that may be revoked very frequently and are best handled by invisible revocation.

Revocation and Physical Naming

The use of physical resource names requires that an exokernel reveal each revocation to the relevant library operating system so that it can relocate its physical names. For instance, a library operating system that relinquishes physical page “5” should update any of its page-table entries that refer to this page. This is easy for a library operating system to do when it deallocates a resource in reaction to an exokernel revocation request. An abort protocol (discussed below) allows relocation to be performed when an exokernel forcibly reclaims a resource.

We view the revocation process as a dialogue between an exokernel and a library operating system. Library operating systems should organize resource lists so that resources can be deallocated quickly. For example, a library operating system could have a simple vector of physical pages that it owns: when the kernel indicates that some page should be deallocated, the library operating system selects one of its pages, writes it to disk, and frees it.

The Abort Protocol

An exokernel must also be able to take resources from library operating systems that fail to respond satisfactorily to revocation requests. An exokernel can define a second stage of the revocation protocol in which the revocation request ("please return a memory page") becomes an imperative ("return a page within 50 microseconds"). However, if a library operating system fails to respond quickly, the bindings need to be broken "by force." The actions taken when a library operating system is recalcitrant are defined by the *abort protocol*.

One possible abort protocol is to simply kill any library operating system and its associated application that fails to respond quickly to revocation requests. We rejected this method because we believe that most programmers have great difficulty reasoning about hard real-time bounds. Instead, if a library operating system fails to comply with the revocation protocol, an exokernel simply breaks all existing bindings to the resource and informs the library operating system.

To record the forced loss of a resource, a *repossession vector* can be used. When an exokernel takes a resource from a library operating system, this fact is registered in the vector and the library operating system receives a "repossession" exception so that it can update any mappings that use the resource. For resources with state, an exokernel can write the state into another memory or disk resource. In preparation, the library operating system can pre-load the repossession vector with a list of resources that can be used for this purpose. For example, it could provide names and capabilities for disk blocks that should be used as backing store for physical memory pages.

Another complication is that an exokernel should not arbitrarily choose the resource to repossess. A library operating system may use some physical memory to store vital bootstrap information such as exception handlers and page tables. The simplest way to deal with this is to guarantee each library operating system a small number of resources that will not be repossessed (*e.g.,* five to ten physical memory pages). If even those resources must be repossessed, some emergency exception that tells a library operating system to submit itself to a "swap server" is required.

2.5 Secure Bindings

One of the primary tasks of an exokernel is to multiplex resources *securely,* providing protection for mutually distrustful applications. To implement protection an exokernel must guard each resource. To perform this task efficiently, an exokernel allows library operating systems to bind to resources using *secure bindings*.

A secure binding is a protection mechanism that decouples authorization from the actual use of a resource. Secure bindings improve performance in two ways. First, the protection checks involved in enforcing a secure binding are expressed in terms of simple operations that the kernel (or hardware) can implement quickly. Second, a secure binding performs authorization only at bind time, which allows management to be decoupled from protection. Application-level software is responsible for many resources with complex semantics (*e.g.*, network connections). By isolating the need to understand these semantics to *bind time*, the kernel can efficiently implement access checks at *access time* because it need no longer involve an application. Simply put, a secure binding allows the kernel to protect resources without understanding them.

Operationally, the one requirement needed to support secure bindings is a set of primitives that application-level software can use to express protection checks. The primitives can be implemented either in hardware or software. A simple hardware secure binding is a TLB entry: when a TLB fault occurs the complex mapping of virtual to physical addresses in a library operating system's page table is performed and then loaded into the kernel (bind time) and then used multiple times (access time). Another example is the packet filter [65], which allows predicates to be downloaded into the kernel (bind time) and then run on every incoming packet to determine which application the packet is for (access time). Without a packet filter, the kernel would need to query every application or network server on every packet reception to determine who the packet was for. By separating protection (determining who the packet is for) from authorization and management (setting up connections, sessions, managing retransmissions, *etc.*) very fast network multiplexing is possible while still supporting complete application-level flexibility.

We use three basic techniques to implement secure bindings: hardware mechanisms, software caching, and downloading application code.

Appropriate hardware support allows secure bindings to be couched as low-level protection operations such that later operations can be efficiently checked without recourse to high-level authorization information. For example, a file server can buffer data in memory pages and grant access to authorized applications by providing them with capabilities for the physical pages. An exokernel would enforce capability checking without needing any information about the file system's authorization mechanisms. As another example, some Silicon Graphics frame buffer hardware associates an ownership tag with each pixel. This mechanism can be used by the window manager to set up a binding between a library operating system and a portion of the frame buffer. The application can access the frame buffer hardware directly, because the hardware checks the ownership tag when I/O takes place.

Secure bindings can be cached in an exokernel. For instance, an exokernel can use a large software TLB [7, 46] to cache address translations that do not fit in the hardware TLB. The software TLB can be viewed as a cache of frequently-used secure bindings.

Secure bindings can be implemented by downloading code into the kernel. This code is invoked on every resource access or event to determine ownership and the actions that the kernel should perform. Downloading code into the kernel allows an application thread of control to be immediately executed on kernel events. The advantages of downloading code are that potentially expensive crossings can be avoided and that this code can run without requiring the application itself to be scheduled. Type-safe languages [9, 75], interpretation, and sandboxing [89] can be used to execute untrusted application code safely [26].

2.6 Methodology Discussion

> You can' t learn too soon that the most useful thing about a principle is that it can always be sacrificed to expediency. — W. Somerset Maugham (1874-1965)

Stylistically, exokernel design consists of two different activities: giving applications control of resources, and implementing protection – i.e., making sure they control only their resources. Ideally, a library operating systems has safe, efficient access to anything a privileged OS does.

Exokernel design profits from a deceptively simple shift in goals. Rather than focus on defining the right abstraction, or how to implement it — characteristic of almost all other software design (operating system related or otherwise) — an exokernel architecture concentrates instead deferring these decisions to untrusted software, and then verifying that they are implemented correctly. This requires constructing interfaces that do checking of an operation rather than imperatively deciding how to do it. Thus, algorithmic design becomes a partitioning process of dividing problems into two pieces. The first contains the most "interesting" aspects, which the exokernel leaves to applications. The second part contains the residue that an exokernel must perform to verify correctness. For this partitioning to be practical, checking must be comparatively inexpensive.

As an example, consider the problem of writing cached disk blocks to stable storage in a way that guarantees consistency across reboots. Rather than an exokernel deciding on a particular write ordering and having to struggle with the associated tradeoffs in scheduling heuristics and caching decisions required, it can instead allow the application to construct schedules, retaining for the much simplified task of merely checking that any application schedule gives appropriate consistency guarantees. Application of this methodology enables an exokernel to leave library operating systems to decide on tradeoffs themselves rather than forcing a particular set, a crucial shift of labor.

While this style of design may seem obvious, in practice it has been depressingly easy to slide into "the old way" of deciding how to implement a particular feature rather than asking the unusual question of how one can get out of doing so, safely. A useful heuristic to catch such slips is to note when one is making many tradeoffs. A plethora of tradeoffs almost invariably signals a decision that could be implemented or optimized in different important ways and, thus, should be left to applications.

Because libOSes understand the higher-level semantics of their abstractions, they “just know” many things that an exokernel does not. For example, that related files will likely be clustered together and if one block is fetched, the succeeding eight blocks should be as well. Thus, a variant of the above methodology is designing interfaces where actions do not require justification. As an example, consider the act of fetching a disk block in core. If a file system must present credentials before the kernel will allow the fetch to be initiated, then it faces serious problems doing the prefetching it needs, since it would first have to read in all file directory entries, show each file's capability to the kernel, and only then initiate the fetches. In contrast, by only performing access control when the actual data in the block or read or written, rather then when the block is fetched, these disk reads can be initiated all at once.

Chapter 3

Practice: Applying exokernel principles

To illustrate the exokernel principles, this chapter discusses how to export low-level primitives such as exceptions and inter-process communication, and multiplex physical resources such as memory, CPU, and the network. To make this discussion concrete, we draw on examples from two exokernel systems: Aegis [25], and Xok [48]. Aegis is the first exokernel we built. It runs on the MIPS [50] DECstation family. Xok is the second. It runs on the x86 architecture and is the more mature of the two. Construction of these systems spanned four years: two for Aegis, two (and counting) for Xok.

Xok and Aegis differ in important ways, helping to show how the application of exokernel principles changes in the face of different constraints. Most of these differences result from the fact that the x86 and MIPS hardware have radically different architectural interfaces (e.g., software page tables for the MIPS, hardware for x86) and implementation performance (an order of magnitude in favor of the x86).

The implementations we discuss are merely examples of how resources *can be* multiplexed, not how they must be. Other implementations are possible.

Stylistically, the discussion of each resource focuses on two questions: (1) how to give applications control over the resource and (2) how to protect it. Some of the resources we discuss have little to do with protection due to the lack of "sharing." For example, besides memory protection issues, exceptions are contained within a single process. Lack of multiplexing makes protection a non-issue. Others, such as networking, focus more heavily on it.

We provide a quick overview of Aegis and Xok below. The remainder of the chapter discusses how they multiplex resources.

3.0.1 Xok overview

Xok multiplexes most standard machine resources: network, CPU, physical memory, and disk. It currently lacks support for display devices. Its default library operating system, ExOS, provides "Unix in a library." ExOS does not handle some Unix corner cases, but it is not a toy. Many sophisticated applications, such as csh, perl, gcc, and telnet, run without modification. There are two main caveats on the ExOS implementation used in this thesis' experiments: (1) lack of virtual memory paging support (the file buffer cache was paged), and (2) some shared data structures (such as the global file descriptor table) reside in shared memory and could be corrupted by a malicious process. Neither is an inherent limitation, but simply the result of lack of time. We have since added paging to ExOS, and we are adding more protection to data structures as time allows. Neither does either improve our performance. If anything, lack of paging hurts our experiments since it means that memory that could be used for the file buffer cache is wasted on backing virtual memory. As discussed in Chapter 5 our experiments compensate for this lack of protection.

3.0.2 Aegis overview

Aegis, while older then Xok, is more primitive, lacking solid support for disk and having a more crude approach to access control (flat access control lists). Its version of ExOS (from which that of Xok descends) also is not as developed. Most of our Aegis measurements use microbenchmarks rather application timings.

The microbenchmarks discussed in this section compare Aegis and ExOS with the performance of Ultrix4.2 (a mature monolithic UNIX operating system) on the same hardware. While Aegis and ExOS do not offer the same level of functionality as Ultrix, we do not expect these additions to cause large increases in our timing measurements.

Ultrix, despite its poor performance relative to Aegis, is *not* a poorly tuned system; it is a mature monolithic system that performs quite well in comparison to other operating systems [69]. For example, it performs two to three times better than Mach 3.0 in a set of I/O benchmarks [67]. Also, its virtual memory performance is approximately twice that of Mach 2.5 and three times that of Mach 3.0 [5].

Our measurements were taken on a DECstation5000/125 (25MHz), with an R3000 processor and a SPECint92 rating of 25.

3.1 Multiplexing Physical Memory

Physical memory is one of the simplest resources to multiplex. When a library operating system allocates a physical memory page, the exokernel records the owner and permissions (e.g., read and write) of the allocating process. The owner of a page has the power to change the capabilities associated with it, to share it, and to deallocate it.

To ensure protection, the exokernel guards every access to a physical memory page by requiring that the capability be presented by the library operating system requesting access. If the capability is insufficient, the request is denied. Typically, the processor contains a TLB, and the exokernel must check memory capabilities when a library operating system attempts to enter a new virtual-to-physical mapping. To improve library operating system performance by reducing the number of times secure bindings must be established, an exokernel may cache virtual-to-physical mappings in a large software TLB.

If the underlying hardware defines a page-table interface, then an exokernel must guard the page table instead of the TLB. Although the details of how to implement secure memory bindings will vary depending on the details of the address translation hardware, the basic principle is straightforward: privileged machine operations such as TLB loads and DMA must be guarded by an exokernel. As dictated by the exokernel principle of exposing kernel book-keeping structures, the page table should be visible (read only) at application level.

To reclaim a page, an exokernel must change the associated capabilities, mark the resource as free, and remove all bindings. In the example of physical memory, bindings include TLB mappings and any queued DMA requests.

3.1.1 Aegis: application virtual memory

The MIPS architecture has software defined page tables. Thus, in accordance with deferring management to applications, Aegis allows applications to define their own page tables. We look at two issues in supporting application-level virtual memory: bootstrapping and efficiency.

To bootstrap the virtual naming system, there must be support for translation exceptions on both application page-tables and exception code. Aegis provides a simple bootstrapping mechanism through the use of a small number of guaranteed mappings. A miss on a guaranteed mapping will be handled automatically by Aegis. This organization frees the application from dealing with the intricacies of boot-strapping TLB miss and exception handlers, which can take TLB misses. To implement guaranteed mappings efficiently, an application's virtual address space is partitioned into two segments. The first segment holds normal application data and code. The second holds exception code and page tables. Virtual addresses in this segment can be “pinned” using guaranteed mappings. Typically libOSes pin exception handling code and page-tables.

On a TLB miss, the following actions occur:

1. Aegis checks which segment the virtual address resides in. If it is in the standard user segment, the exception is dispatched directly to the application. If it is in the second region, Aegis first checks to see if it is a guaranteed mapping. If so, Aegis installs the TLB entry and continues; otherwise, Aegis forwards it to the application.

2. The application looks up the virtual address in its page-table structure and, if the access is not allowed raises the appropriate exception (*e.g.,* “segmentation fault”). If the mapping is valid, the application constructs the appropriate TLB entry and its associated capability and invokes the appropriate Aegis system routine.

3. Aegis checks that the given capability corresponds to the access rights requested by the application. If it does, the mapping is installed in the TLB and control is returned to the application. Otherwise an error is returned.

4. The application performs cleanup and resumes execution.

In order to support application-level virtual memory efficiently, TLB refills must be fast. To this end, Aegis caches TLB entries (a form of secure bindings) in the kernel by overlaying the hardware TLB with a large software TLB (STLB) to absorb capacity misses [7, 46]. On a TLB miss, Aegis first checks to see whether the required mapping is in the STLB. If so, Aegis installs it and resumes execution; otherwise, the miss is forwarded to the application.

The STLB contains 4096 entries of 8 bytes each. It is direct-mapped and resides in unmapped physical memory. An STLB "hit" takes 18 instructions (approximately one to two microseconds). In contrast, performing an upcall to application level on a TLB miss, followed by a system call to install a new mapping is at least three to six microseconds more expensive.

As dictated by the exokernel principle of exposing kernel book-keeping structures, the STLB can be mapped using a well-known capability, which allows applications to efficiently probe for entries.

Using the control given by libOS-defined page tables ExOS has a variety of different page table structures [1], high-performance network paging, and application-specific page coloring (for improved cache performance).

3.1.2 Xok: hardware-defined page tables

Unlike the MIPS architecture, the x86 architecture on which Xok runs defines the page-table structure. Since x86 TLB refills are handled in hardware, this structure cannot be overridden by applications. However, applications are able to control all hardware-defined per-page attributes, including protection levels (read-only, writable, execute-only) and caching. Additionally, each valid page-table entry contains three software-defined bits, which Xok gives over to application control. Invalid entries (those without the "present bit" set) can contain any value the library operating system wishes. Example uses of this location are to store the names of disk blocks used as backing store or even network addresses for remote pages.

Since the hardware does not verify that the physical page of a translation can be mapped by a process, applications are prevented from directly modifying the page table and must instead use system calls. Although these restrictions make Xok less extensible than Aegis, they simplify the implementation of libOSes (see Chapter 7 for more discussion).

Like Aegis, Xok allows efficient and powerful virtual memory abstractions to be built at the application level. It does so by exposing the capabilities of the hardware (e.g., all MMU protection and dirty bits) and exposing many kernel data structures (e.g., free lists, inverse page mappings). Xok's low-level interface means that paging is handled by applications. As such, it can be done from disk, across the network, or by data regeneration. Additionally, applications can readily perform per-page transformations such as compression, verification of contents using digital signatures (to allow untrusted nodes in a network to cache pages), or encryption.

[1] Illustrative of the customizability of an exokernel system, Tom Pinckney while still an undergraduate was able to implement a new page table structure in a week, while taking his final exams. As a testament to the difficulty of modifying current operating systems, the proposers of this page table structure were only able to simulate it [83].

3.2 Multiplexing the Network

This subsection discusses how to give applications control over the network, and how to implement protection. There are two primary requirements for efficient networking. First, elimination of data copies, which comes from both giving applications access to any scatter-gather DMA provided by the hardware and allowing them to direct where messages are placed. Second, tight coupling to interrupt events, primarily, message reception and timer interrupts. In order to initiate low-latency responses to messages, application messaging code must be able to run quickly after message arrival. Aegis performs this by downloading application code into the kernel and running it in the interrupt handler [92]. Since Aegis was implemented, the ratio of processor speed to network latency has increased dramatically (about a factor of five to ten for small messages). Thus, Xok, running on modern hardware, does not need to eliminate boundary crossings, and simply yields to the receiving application. Timer interrupts are needed to build efficient retransmission timers, which are needed to implement fast reliable messaging on unreliable network hardware.

To enable application resource management, an exokernel dislocates operating system code into libraries. However, there are important differences between a library's execution context and that of the kernel. One of the challenges in an exokernel is recapturing these aspects. Tight coupling of libOS code to events can be viewed as an example of this. Compared to Ultrix, Aegis's provision of efficient access to these two abilities gives library operating systems a factor of ten performance improvement in round trip Ethernet times [25].

Protection for networking is answering the question: given a message, who owns it? On a connection-oriented network, answering this question is easy: whichever connection (or flow) it belongs to. Binding of application to flow can happen at connection initiation, removing the need to make decisions based on packet contents. An example of a hardware-based mechanism is the use of the virtual circuit in ATM cells to securely bind streams to applications [23]. However, answering this question is more difficult on a connectionless network such as Ethernet, where message ownership requires understanding header semantics. Since the exokernel has dislocated all network protocol code that understands packet semantics into library operating systems it lacks the information necessary to decide which application owns what message.

To solve this problem, an exokernel requires that networking libraries download packet filters [65] to select the messages they want. Conceptually, every filter is invoked on every arriving packet and returns “accept” (the filter wants the message) or “reject” (the filter does not want the message). The operating system thus need not understand the actual bits in a message in order to bind an arriving packet to its owner.

For protection, the exokernel must ensure that that a filter does not “lie” and accept packets destined to another process. To prevent this theft we intentionally designed our filter language to make “overlap” detection simple. (Alternatively, simple security precautions such as only allowing a trusted server to install filters could be used to address this problem.) Finally, we ensure fault isolation through a combination of language design (to bound runtime) and runtime checks (to protect against wild memory references and unsafe operations).

Both Aegis and Xok use packet filters, because our current network does not provide hardware mechanisms for message demultiplexing. One challenge with a language-based approach is to make filters fast. Traditionally, packet filters have been interpreted, making them less efficient than in-kernel demultiplexing routines. One of the distinguishing features of the packet filter engine used by our prototype exokernel is that it compiles packet filters to machine code at runtime, increasing demultiplexing performance by more than an order of magnitude as compared to other packet filters [28, 31]. [2]

Sharing the network interface for outgoing messages is easier. Messages are simply copied from application space into a transmit buffer. In fact, with appropriate hardware support, transmission buffers can be mapped into application space just as easily as physical memory pages [23].

An exokernel defers message construction to applications. A protection problem this creates is how to prevent applications from "spoofing" other applications by sending bogus messages. Our two exokernel systems do not prevent this attack, since they were designed for an insecure networking environment. However, within the context of a trusted network, an exokernel could use "inverse" packet filters to reject messages that do not fit a specified pattern (or, alternatively, reject messages that do). The techniques that worked for DPF could be applied here as well.

3.3 Multiplexing the CPU

Control over the CPU involves the ability to: (1) allocate and share CPU time-slices (similar to other resources), (2) yield a time-slice to a specific named process and (3) receive notification (e.g., via an upcall) when time-slices begin and end. The attributes of time slices are quantity and "timeliness." Applications should be able to allocate specific time slices to control either of these attributes. Finally, the notion of "process" should be low-level, consisting of the information required by hardware, such as program counters to vector machine exceptions to, and protection-required information such as a binding between the process and a principal.

The following discussion makes these rules concrete by considering Aegis' treatment of the CPU and its process and exception facilities.

3.3.1 Aegis and Xok CPU multiplexing

Both Xok and Aegis multiplex the CPU in the same way. They represent the CPU as a linear vector, where each element corresponds to a time slice. Time slices are partitioned at the clock granularity and can be allocated in a manner similar to physical memory. Scheduling is done "round robin" by cycling through the vector of time slices. A crucial property of this representation is *position*, which encodes an ordering and an approximate upper bound on when the time slice will be run. Position can be used to meet deadlines and to trade off latency for throughput. For example, a long-running scientific application could allocate contiguous

[2] However, recently we have experimented with the use of an aggressive interpreter that, by preprocessing filters and exploiting "super-operator" instructions, runs roughly within a factor of four of hand-tuned code — a perfectly adequate speed given that demultiplexing is coupled to a high-latency I/O event (packet reception).

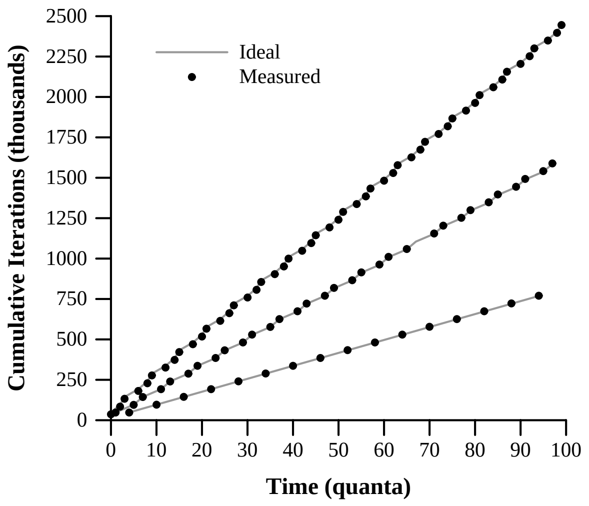

Figure 3-1: Application-level stride scheduler.

time slices in order to minimize the overhead of context switching, while an interactive application could allocate several equidistant time slices to maximize responsiveness.

The kernel provides a yield primitive to donate the remainder of a process' current time slice to another (specific) process. Applications can use this simple mechanism to implement their own scheduling algorithms.

Timer interrupts denote the beginning and end of time slices, and are delivered in a manner similar to exceptions (discussed below): a register is saved in the “interrupt save area,” the exception program counter is loaded, and the kernel jumps to user-specified interrupt handling code with interrupts re-enabled. The application's handlers are responsible for general-purpose context switching: saving and restoring live registers, releasing locks, *etc*. This framework gives applications a large degree of control over context switching. For example, because it notifies applications of clock interrupts, and gives them control over context switching's state saving and restoration, it can be used to implement scheduler activations [4].

Fairness is achieved by bounding the time an application takes to save its context: each subsequent timer interrupt (which demarcates a time slice) is recorded in an excess time counter. Applications pay for each excess time slice consumed by forfeiting a subsequent time slice. If the excess time counter exceeds a predetermined threshold, the environment is destroyed. In a more friendly implementation, the kernel could perform a complete context switch for the application.

This simple scheduler can support a wide range of higher-level scheduling policies. As we demonstrate below, an application can enforce proportional sharing on a collection of sub-processes.

Extensible Schedulers on Aegis

Using the Aegis yield primitive and control over time slices, we have built an application-level scheduler that implements *stride scheduling* [91], a deterministic, proportional-share scheduling mechanism that improves on recent work [90]. The ExOS implementation maintains a list of processes for which it is responsible, along with the proportional share they are to receive of its time slice(s). On every time slice wakeup, the scheduler calculates which process is to be scheduled and yields to it directly.

We measure the effectiveness of this scheduler by creating three processes that increment counters in shared memory. The processes are assigned a 3:2:1 relative allocation of the scheduler's time slice quanta. By plotting the cumulative values of the shared counters, we can determine how closely this scheduling allocation is realized. As can be seen in Figure 3-1, the achieved ratios are very close to idealized ones, showing that applications can effectively control scheduling.

It is important to note that there is nothing special about this scheduler either in terms of privileges (*any* application can perform identical actions) or in its complexity (the entire implementation is less than 100 lines of code). As a result, any application can easily manage processes.

3.3.2 Aegis Processor Environments

An Aegis processor environment is a structure that stores the information needed to deliver events to applications. All resource consumption is associated with an environment because Aegis must deliver events associated with a resource (such as revocation exceptions) to its designated owner.

Four kinds of events are delivered by Aegis: exceptions, interrupts, protected control transfers, and address translations. Processor environments contain the four contexts required to support these events:

Exception context: for each exception, an exception context contains a program counter for where to jump to and a pointer to physical memory for saving registers.

Interrupt context: for each interrupt an interrupt context includes a program counters and register-save region. In the case of timer interrupts, the interrupt context specifies separate program counters for start-time-slice and end-time-slice cases, as well as status register values that control co-processor and interrupt-enable flags.

Protected Entry context: a protected entry context specifies program counters for synchronous and asynchronous protected control transfers from other applications. Aegis allows any processor environment to transfer control into any other; access control is managed by the application itself.

Addressing context: an addressing context consists of a set of *guaranteed mappings*. A TLB miss on a virtual address that is mapped by a guaranteed mapping is handled by Aegis. Library operating systems rely on guaranteed mappings for bootstrapping page-tables, exception handling code, and exception stacks. The addressing context also includes an address space identifier, a status register, and a tag used to hash into the Aegis software TLB. To switch from one environment to another, Aegis must install these three values.

These are the event-handling contexts required to define a process. Each context depends on the others for validity: for example, an addressing context does not make sense without an exception context, since it does not define any action to take when an exception or interrupt occurs.

3.3.3 Implementing Unix Processes on Xok

The *process map* maps UNIX process identifiers to Xok environment numbers using a shared table. The *process table* records the process identifiers of each process, that of its parent, the arguments with which the process was called, its run status, and the identity of its children. The table is partitioned across application-reserved memory of Xok's environment structure, which is mapped readable for all processes and writeable for only the environment's owning process. ExOS uses Xok's IPC to safely update parent and child process state. The UNIX *ps* (process status) program is implemented by reading all the entries of the process table.

UNIX provides the *fork* system call to duplicate the current process and *exec* to overlay it with another. *Exec* is implemented by creating a new address space for the new process, loading on demand the disk image of the process into the new address space, and then discarding the address space that called *exec*. Implementing fork in a library is peculiar since it requires that a process create a replica of its address space and state *while it is executing*. To make fork efficient, ExOS uses copy-on-write to lazily create separate copies of the parent's address space. ExOS scans through its page tables, which are exposed by Xok, marking all pages as copy-on-write except those data segment and stack pages that the *fork* call itself is using. These pages must be duplicated so as not to generate copy-on-write faults while running the *fork* and page fault handling code. Groups of page table entries are updated at once by batching system calls to amortize the system call overhead over many updates.

3.4 Exposing Machine Events

An exokernel gives applications low-level efficient access to machine events such as exceptions and interrupts (which we have already seen in the context of networking and the CPU). It also provides primitives for inter-domain calls, which safely change the program counter in one process to an agreed upon value in another's address space. Below, we discuss Aegis' mechanisms for exceptions and control transfer, along with Xok's facilities for interprocess communication.

3.4.1 Aegis Exceptions

Aegis dispatches all hardware exceptions to applications (save for system calls) using techniques independently developed but similar to those described in Thekkath and Levy [87]. To dispatch an exception, Aegis performs the following actions:

1. It saves three scratch registers into an agreed-upon "save area." (To avoid TLB exceptions, Aegis does this operation using physical addresses.)

2. It loads the exception program counter, the last virtual address that failed to have a valid translation, and the cause of the exception.

3. It uses the cause of the exception to perform an indirect jump to an application-specified program counter value, where execution resumes with the appropriate permissions set (*i.e.*, in user-mode with interrupts re-enabled).

After processing an exception, applications can immediately resume execution without entering the kernel. Ensuring that applications can return from their own exceptions (without kernel intervention) requires that all exception state be available for user reconstruction. This means that all registers that are saved must be in user-accessible memory locations.

Currently, Aegis dispatches exceptions in 18 instructions (1.5 microseconds on our 25MHz DECstation5000/125), roughly a factor of a hundred faster than Ultrix, a monolithic OS running on the same hardware, and over five times faster than the most highly-tuned implementation in the literature. Mainly this improvement comes from the two facts that (1) exokernel primitives are low-level, which means applications do not pay for unnecessary functionality, and (2) the exokernel concentrates on doing a few things well. For example, Aegis is small, and does not need to page its data structures. Because, all kernel addresses are physical they never miss in the TLB, and Aegis does not have to separate kernel TLB misses from the more general class of exceptions in its exception demultiplexing routine.

Fast exceptions enable a number of intriguing applications: efficient page-protection traps can be used by applications such as distributed shared memory systems, persistent object stores, and garbage collectors [5, 87]. Exception times are discussed further in [25].

3.4.2 Aegis Protected Control Transfers

Aegis provides a *protected control transfer* mechanism as a substrate for efficient implementations of inter-process compunction (IPC) abstractions. This mechanism illustrates the principle of not encapsulating primitives within high-level abstractions. It consists of the minimum operations needed to transfer execution from one process to another in a safe way: it changes the program counter to an agreed-upon value in the callee, donates the current time slice to the callee's processor environment, and installs the required elements of the callee's processor context (addressing-context identifier, address-space tag, and processor status word).

Aegis provides two forms of protected control transfers: *synchronous* and *asynchronous*. The difference between the two is what happens to the processor time slice. Asynchronous calls donate only the remainder of the current time slice to the callee. Synchronous calls donate the current time and all future instantiations of it; the callee can return the time slice via a synchronous control transfer call back to the original caller. Both forms of control transfer guarantee two important properties. First, to applications, a protected control transfer is atomic: once initiated it will reach the callee. Second, Aegis will not overwrite any application-visible register. These two properties allow the large register sets of modern processors to be used as a temporary message buffer [17].

Currently, Aegis' synchronous protected control transfer operation takes 30 instructions (1.4 microseconds on a 25MHz DECstation5000/125) [25]. Roughly ten of these instructions are used to distinguish the system call “exception” from other hardware exceptions on the MIPS architecture. Setting the status, co-processor, and address-tag registers consumes the remaining 20 instructions, and could benefit from additional optimizations. Because Aegis implements the minimum functionality required for any control transfer mechanism, applications can efficiently construct their own IPC abstractions. As an example, we have implemented a “trusted” local remote procedure call for use between a client and a server that the client trusts to save and restore callee-saved registers. By eliminating the need to save all registers, trusted LRPC improves performance by roughly a factor of two.

Hardware differences make comparing our numbers to others in the literature difficult. We attempt an extremely crude comparison of our protected control transfer operation to the equivalent operation on L3 [57], the fastest published result by normalizing the IPC on both systems using SPECint92 rating of Aegis's DEC5000 and L3's 486 (16.1 vs. 30.1). Aegis's trusted control transfer mechanism is 6.6 times faster than the scaled time for L3's RPC mechanism.

Given the difference in architectures, this scaling should be treated as a “back of the envelop” computation. The main conclusion to draw is that Aegis' control transfer is fast.

3.4.3 Implementing Unix IPC on Xok

UNIX defines a variety of interprocess communication primitives: signals (software interrupts that can be sent between processes or to a process itself), pipes (producer-consumer untyped message queues), and sockets (differing from pipes in that they can be established between non-related processes, potentially executing on different machines).

Signals are layered on top of Xok IPC. Pipes are implemented using Xok's *software regions*, which provide sub-page memory protection, coupled with a “directed yield” to the other party when it is required to do work (i.e., if the queue is full or empty). Sockets communicating on the same machine are currently implemented using a shared buffer.

Inter-machine sockets are implemented through user-level network libraries for UDP and TCP. The network libraries are implemented using Xok's timers, upcalls, and packet rings, which allow protected buffering of received network packet,

3.5 Discussion

This chapter has discussed how an exokernel can safely export a variety of resources — physical memory, the CPU, the network, and hardware events — to applications. The low-level of the exokernel interface allows library operating systems, working above the exokernel interface, to implement higher-level abstractions and define special-purpose implementations that best meet the performance and functionality goals of applications. This organization allows the redefinition of fundamental operating system abstractions by simply changing application-level libraries. Furthermore, these different versions can co-exist on the same machine and are fully protected by Aegis.

The next chapter presents the resource we have found most challenging, disk, along with our solution to it (and five failed solutions).

Chapter 4

The Hardest Multiplexing Problem: Disk

An exokernel must provide a means to safely multiplex disks among multiple library file systems (libFSes). Each libOS contains one or more libFSes. Multiple libFSes can be used to share the same files with different semantics. In addition to accessing existing files, libFSes can define new on-disk file types with arbitrary meta data formats. An exokernel must give libFSes as much control over file management as possible while still protecting files from unauthorized access. It therefore cannot rely on simple-minded solutions like partitioning to multiplex a disk: each file would require its own partition.

To allow libFSes to perform their own file management, an exokernel stable storage system must satisfy four requirements. First, creating new file formats should be simple and lightweight. It should not require any special privilege. Second, the protection substrate should allow multiple libFSes to safely share files at the raw disk block and meta data level. Third, the storage system must be efficient—as close to raw hardware performance as possible. Fourth, the storage system should facilitate cache sharing among libFSes, and allow them to easily address problems of cache coherence, security, and concurrency.

The goal of disk multiplexing is to make untrusted library file systems (libFSes) as powerful as privileged file systems. As listed above, there are a number of engineering challenges to reaching this goal. The hardest challenge by far, however, is access control: i.e., answering the deceptively simple question "who can use a given disk block?" Inventing a satisfactory solution to this problem took us three years and four systems.

This chapter describes how we multiplex stable storage, both to show how we address these problems and to provide a concrete example of the exokernel principles in practice. We first describe out solution to efficient access control. An interesting aspect of this solution is that it uses the libFSes own data structures to track what the libFS owns. Another is the the code verification technique we invented to do such re-use without impinging on libFS flexibility. We then give an overview XN, Xok's extensible, low-level in-kernel stable storage system. We also describe the general interface between XN and libFSes and present one particular libFS, C-FFS, the co-locating fast file system [37].

4.1 Efficient, fine-grained disk multiplexing

To provide protection, the operating system must force each libFS to only access those disk blocks for which they have access rights. For flexibility and speed an exokernel provides fine-grained protection: i.e., at the level of disk blocks rather than partitions. To do so we must answer the simple question “who can use this block?” Doing so requires constructing a mapping of each disk block to every principal that can use it, which turns out to be quite difficult [48] mainly because it requires building the moral equivalent of a file system. For example, the mapping table will be enormous (since its size is proportional to the size of disk) and, hence, not fit in main memory. As a result, managing it as it is is moved between both memory and disk requires solving all the standard file systems issues — deciding which pieces to cache, prefetch and evict, how to perform allocation of the table on disk, how to reconstruct the table's state after a system crash, etc. Obviously, if an exokernel already implements a file system, libFSes built on top of it will have little freedom. Fortunately, there are a series of insights that can allow us to trust the libFS itself to efficiently track the blocks it owns, thereby eliminating the need for duplicate bookkeeping.

The first insight is to notice that a correct libFS will already track what disk blocks it owns; doing so efficiently is, after all, one of the main purposes of a file system. Therefore, any bookkeeping an exokernel does is redundant. Thus, if it can reuse the bookkeeping data structures of the libFS then we can eliminate this redundancy. [1] For example, a Unix file system tracks what blocks are associated with it (and what principals are allowed to use them) using “meta data” consisting of directory blocks, inodes, as well as single, double, and triple indirect blocks. In other words, everything the kernel needs to track to perform access control. If it can reuse libFS meta data, then, it does not need to perform duplicate tracking of what blocks are associated with which libFS.

Our new problem is that reusing libFS bookkeeping structures requires that we understand them, both so that we can force them to be correct, and so that we can extract ownership information from them. One traditional solution would be to provide a fixed set of components (e.g., pointers to disk blocks, disk block extents, etc.) from which libFSes could build their meta data from. However, creating a set of universal building blocks for file system meta data is so hard as to be infeasible: file system research is still an active area even after three decades. A component set capable of describing all results of this research does not seem possible. Instead, we solve this problem by having the libFS interpret its meta data for us in a way that we can test for correctness.

[1] Note that this insight applies to all all areas that deal with protection, not just file systems: correct applications track what resources they own. Future work will involve exploiting this fact more aggressively.

Determinism + induction = verification

Our struggle has two conflicting pieces: flexibility (in that we want to allow client meta data to have any "syntax" and semantics whatsoever, including those that we have not anticipated) and validity (in that we require that they do so correctly, even when we don' t understand its representation). *Untrusted deterministic functions* (UDFs) let us achieve both goals: clients can use any possible meta data representation, while we still can verify correctness, no matter how mysterious the representation they chose.

Flexibility comes from adding a layer of indirection: rather than have meta data built from components the operating system understands, libFSes provide an "interpreter" function, owns, for each meta data type. Given an instance of that type, the owns UDF produces the set of blocks that that instance controls: owns: (meta) $\rightarrow$ set of blocks controlled by meta. For example, a Unix file system will provide an owns function for each of the types of meta data it uses — inodes, directory blocks, indirect nodes, double indirect nodes, triple indirect nodes, etc. Thus, when the the operating system needs to know what disk blocks a piece of meta data controls, it simply runs that meta data through its associated owns function. Since owns is written in a quasi-Turing complete language (in our implementation a pseudo-assembly language) it can describe any possible file system meta data layout. [2]

At this point, while we have flexibility, guarantees of correctness have become more challenging. owns is written in general purpose code. Verification (i.e., that owns correctly implements its specification) is beyond current formal methods. The UDF techniques discussed in the remainder of the chapter are an online alternative that is both simple and robust.

UDFs start from the fact that if a deterministic Turing machine terminates and produces an output O then, given the same initial state and input, it will always terminate and produce O. The key to verification is that determinism gives persistence: once owns(meta) produces a valid result, it will always do so, until meta is allowed to change. Without persistence past tests can tell us nothing about present or future behavior.

To make UDFs deterministic we must guarantee three conditions:

1. The instructions the UDF executes have deterministic semantics.

2. No information can be allowed to flow into the universe of the UDF and its state. For example, on each invocation of the UDF all registers and stack locations must be initialized to the same values, the UDF cannot use self-modifying code, call non-deterministic functions (such as a "get time of day" routine), have access to cycle counters, etc.

3. All state that persists across a UDF's invocations is visible to the verifier so that it can retest the UDF when this state is modified.

[2] While the language used to write UDFs is more-or-less Turing complete, their execution environment is restricted (since UDFs cannot run "too long") as are UDFs (since they must be practical to verify). Similar restrictions will hold for any code downloaded into the kernel, since it must be prevented from at least corrupting kernel data structures. For linguistic we will still refer to such extensions as Turing complete despite this restricted execution environment.

We assume that UDFs are made of immutable code and protected data and that the execution of this code has been made "safe" to protect the testing evaluator from malice: the code is prevented from corrupting the evaluator's state, looping "too long," etc. The implementation discussed in this section guarantees these conditions using a safe interpreter. A trusted compiler could also be used.

Determinism gives persistence and, as a result, allows us to use induction to verify UDF correctness. Inductive testing has two phases: initialization and modification. Initialization tests that the UDF's initial state is a valid one. Modification tests that when a UDF's state is mutated, the mutation leaves the UDF in a valid state.

Thus, to trust libFS meta data and its associated owns function, the verifier needs to check that owns(meta) produces a valid output when meta is initialized, and then retest it after each modification. Once the verifier can force all libFS meta data to produce valid results, we have accomplished the goal of this section: libFSes can now track their disk blocks without the kernel having to duplicate this bookkeeping.

More specifically, initialization checks that when the libFS allocates a piece of meta data, it should not control any disk blocks:

```
% verify owns(meta) is in a valid initial state:
% i.e., meta should control no disk blocks.
proc initialize(meta)
  if owns(meta) != {}
    error "Bogus initial state!";
end;
```

If owns(meta) does not yield the empty set then the verifier knows that either owns is incorrect or meta is not properly initialized. In either case it rejects meta. (We do not actually care if owns itself is correct — i.e., that it works on all possible inputs — just that it work on the current set of used inputs.) Otherwise, it accepts meta. Because owns is deterministic the verifier is guaranteed that until meta is modified owns(meta) will always produce the empty set.

Next, when a file system wants to modify its meta data (say to allocate a disk block to a file) the verifier checks that meta goes from its current valid state to a new valid state, where, because of determinism, it must remain.

Allocation is thus:

```
% Give meta:type control of disk block b
proc allocate(meta, type, b)
 old_set = type.owns(meta)      % record original state
```

```
    <let libFS scribble on meta as it will>
    new_set = type.owns(meta)     % record new state

    % check that owned set grew by exactly b.
    if new_set != old_set U { b }
      error "Bogus modification!";
  end;
```

Because owns is deterministic the verifier can find out its current state by simply querying it. This is the crucial part of the process that frees us from duplicating any bookkeeping data structures. Using that current state the verifier can ensure that the modification goes to a valid next state with a straightforward process of computing set union and equality.

Deallocation is similar to allocation:

```
  % Deallocate disk block b from meta:type
  proc deallocate(meta, type, b)
    old_set = type.owns(meta);      % record original state
    <let libFS scribble on meta as it will>
    new_set = type.owns(meta);      % record new state
    % check that owned set shrunk by exactly b.
    if set_diff(new_set, b) != old_set
      error "Bogus modification!";
  end;
```

While the verifier must run owns in a safe evaluation context after meta has been modified, any subsequent invocation of owns(meta) can run without safety checks since owns is deterministic and, after testing, the verifier knows that it executes safely on this value of meta. In the pseudo-code above, while new_set = owns(meta) must be run in a safe context, the statement old_set = owns(meta) can run unchecked. One way to look at this fact is that halting problem is trivial for deterministic Turing machines that one knows have halted in the past.

Naively, UDF induction might appear to be nothing more than a simplistic application of testing pre- and post-conditions. The crucial difference is that the pre-condition is supplied by the untrusted UDF implementor, who has been rendered a trustworthy partner through determinism.

Determinism and XN's trusted set implementation (used to check UDF output) prevents UDFs from "cheat" and producing bogus output after testing.

At this point we can incrementally verify that owns implements its specification correctly. As a result, the operating system can now rely on potentially malicious libFSes to track what disk blocks they own, in ways that the operating system does not understand, while still being able to guarantee correctness. Figure 4-1 sketches how the above verification steps are embedded in the context of a block allocation system call.

Because our approach merely verifies *what* a UDF did rather than *how* it did so it is both more robust and simpler than traditional verification approaches. Only a handful of lines of pseudo-code are required to verify the correctness of code that is written in an untyped, general-purpose assembly language that allows pointers (including casts between integers and pointers), aliasing, stores, dynamic memory allocation, arbitrary loops, and unstructured control flow. Automated theorem provers, in contrast, are both complex and unable to verify such code.

4.1.1 Efficiency: State Partitioning

Since an instance of meta data can control a large number of blocks, enumerating all blocks after each modification can be inefficient. (For example, the Unix "indirect block" in Figure 4-2 controls up to 1024 disk blocks.) Fortunately, the computation to produce a given element in a UDF's owns set typically uses only a limited portion of the UDF's state. This skewed usage can be exploited by *state partitioning*, which at a high level allows UDFs to partition their associated state into sets of disjoint ranges, where each set contains all the state needed to compute a subset of owned blocks (much smaller in size than the own set controlled by the meta data). Our inductive steps proceed as before, with the two changes that at initialization we need to verify that the partitions are non-overlapping and at modification that the libFS only modifies state in the indicated partition.

More concretely, when a libFS creates a type, it indicates how many partitions that type has, and supplies a UDF, access which, given partition id, returns the set of bytes in that partition. Access is a "constant" UDF in that it does not look at any data, and thus never undergoes a state change.

On initialization, the verifier checks that no partition overlaps with another:

```
proc initialization(meta, type)
  set = {};
  for id := 0 to type.n 1
    if(set_overlap(set, type.access(id))
      error "Partitions cannot overlap!";
    set = set U type.access(id);
  end;
end;
```

```
/*
 * C pseudo code sketch of how to allocate block 'req' and stuff a pointer to
 * it in the parent.   For simplicity we assume meta is BLOCKSIZE big and that
 * setting a block to all zeros is a valid initialization.
 */
int sys_alloc_blk(blk_t req, blk_t parent, void *new_meta) {
  set_t old_owns, new_owns;
  set_t (*owns)(void *);      /* pointer to owns function */
  void *old_meta, *kid_meta;

  if(!isfree(disk_freemap, req))              /* Is requested block is on the freelist? */
    return NOT_FREE;
  if(!can_read(new_meta, BLOCKSIZE))          /* Is [new_meta,new_meta+BLOCKSIZE) valid memory? */
    return NOT_VALID;
  if(!(old_meta = buffer_cache_lookup(parent))) /* Is parent in core? */
    return NOT_IN_CORE;
  if(!can_write(parent))                      /* Can the current process write to parent? */
    return CANNOT_ACCESS;

  owns = owns_lookup(parent);                 /* get owns function for parent. */
  old_owns = owns(old_meta);                  /* compute current and potential owned sets. */

  /* if owns(new_meta) did an illegal operation, safe_eval returns nil. */
  if((new_owns = safe_eval(owns(new_meta)))
    return ILLEGAL_OP;

  /* compare: old U req = new */
  if(!set_equal(new_owns, set_union(old_owns, req)))
    return BOGUS_UPDATE;

  /* allocate a buffer cache entry to hold kid. */
  if(!(kid_meta = buffer_cache_alloc(req)))
    return CANNOT_ALLOC;

  memcpy(old_meta, new_meta, BLOCKSIZE);      /* overwrite parent with new value. */
  memset(kid_meta, 0, BLOCKSIZE);             /* initialize kid buffere cache entry to all zeros. */
  set_dirty(parent);                          /* ensure parent will be flushed back to disk. */

  return SUCCESS;
}
```

Figure 4-1: Implementation sketch of a disk block allocation system call that uses UDFs to let untrusted file systems track the blocks they control.

```
% Unix file system indirect block.
struct indirect_block {
    unsigned blocks[1024];
};

% UDF that returns the (singleton) set of
% [offset, nbyte) tuples of all bytes in partition i.
set indirect_access(int id) {
   return { [sizeof(unsigned) ∗ id, sizeof(unsigned)] };
}

% return number of partitions in an indirect block.
int indirect_npartitions(void) {
   return 1024;
}
```

Figure 4-2: Unix indirect block and its associated state partitioning UDFs, indirect_access and indirect_npartitions. The UDFs are "constant" in that they do not use the meta data block to compute partitions. Non-constant UDFs can be used as long as they are retested when the state they depend on has been modified.

Allocation checks that the given partition id is valid, and then adds the block to the set:

```
proc allocation(meta, id, type)
   if id < 0 || id ≥ type.n
     error "Bogus partition!";

   old_set = type.owns(id);
   <let libFS modify state in type.access(id)>
   new_set = type.owns(id);

   if old_set U db != new_set
      error "Bogus modification";
end;
```

Partitioning of state is controlled by UDFs, since it is their implementor that knows the natural partitions of the problem. For instance, in the simple case of the Unix indirect block given in Figure 4-2, which is simply a vector of 1024 disk block pointers, a partition corresponds to the 32 bits in which a single block pointer is stored. In most situations we have encountered state partitioning can reduce output sets to singleton entries.

In a sense, we already do coarse-grain state partitioning, since we only rerun a UDF when the disk block it is associated with is modified rather than requiring that a file system have a single UDF that is run on modification to any block on disk. Partitioning simply takes this subdivision to finer levels.

To simplify writing UDFs we can refine the partition scheme to make the enumeration of the bytes in a partition implicit rather than explicit. Instead of requiring the UDF client supply an access function, we instead have the safe UDF evaluator trace the memory locations examined by a particular UDF invocation. Initialization checks that these read sets do not overlap by tracing owns on each partition id rather than calling access. Modification checks that the read set of the modified meta data on a given id is the same as the original read set. One complication of this model is handling the case where the read set grows or shrinks. Due to space limitations we elide a discussion of how to do this gracefully; interested readers can refer to [30].

4.2 Overview of XN

Designing a flexible exokernel stable storage system has proven difficult: XN is our fourth design. This section provides an overview of how we use UDFs, the cornerstone of XN; the following sections describe some earlier approaches (and why they failed), and aspects of XN in greater depth.

XN provides access to stable storage at the level of disk blocks, exporting a buffer cache registry (Section 4.4) as well as free maps and other on-disk structures. The main purpose of XN is to determine the access rights of a given principal to a given disk block as efficiently as possible. XN must prevent a malicious user from claiming another user' s disk blocks as part of her own files. On a conventional OS, this task is easy, since the kernel itself knows the file' s meta data format. On an exokernel, where files have application-defined meta data layouts, the task is more difficult. On XN libFSes provide UDFs that act as meta data translation functions specific to each file type. XN uses UDFs to analyze meta data and translate it into a simple form the kernel understands. A libFS developer can install UDFs to introduce new on-disk meta data formats. UDFs allow the kernel to safely and efficiently handle any meta data layout without understanding the layout itself.

UDFs are stored on disk in structures called *templates.* Each template corresponds to a particular meta data format; for example, a UNIX file system would have templates for data blocks, inode blocks, inodes, indirect blocks, etc. Each template T has two UDFs: *owns-udf*$_T$ and *refcnt-udf*$_T$, and two untrusted and potentially nondeterministic functions: *acl-uf*$_T$ and *size-uf*$_T$. All four functions are specified in the same language but only *owns-udf*$_T$ and *refcnt-udf*$_T$ must be deterministic. The other two can have access to, for example, the time of day. The limited language used to write these functions is a pseudo-RISC assembly language, checked by the kernel to ensure determinacy. Once a template is specified, it cannot be changed.

The *owns-udf* function allows XN to check the correctness of libFS meta data modifications (specified as a list of byte range modifications) using techniques from the previous subsection.

The *refcnt-udf* function allows libFSes to represent reference counts however they wish. For simplicity, reference count are stored in the block that is pointed to (and, thus, they are persistent). Whenever an edge to this block is formed, XN verifies that *refcnt-udf* increases by one. And, when an edge is deleted, that the count is decremented.

The *acl-uf* function implements template-specific access control and semantics; its input is a piece of meta data, a proposed modification to that meta data, and set of credentials (e.g., capabilities). Its output is a Boolean value approving or disapproving of the modification. XN runs the proper *acl-uf* function before any meta data modification. *acl-uf*s can implement access control lists, as well as providing certain other guarantees; for example, an *acl-uf* could ensure that inode modification times are kept current by rejecting any meta data changes that do not update them.

The *size-uf* function simply returns the size of a data structure in bytes.

4.3 XN: Problem and history

The most difficult requirement for XN is efficiently determining the access rights of a given principal to a given disk block. We discuss the successive approaches that we have pursued.

Disk-block-level multiplexing. One approach is to associate with each block or extent a capability (or access control list) that guards it. Unfortunately, if the capability is spatially separated from the disk block (e.g., stored separately in a table), accessing a block can require two disk accesses (one to fetch the capability and one to fetch the block). While caching can mitigate this problem to a degree, we are nervous about its overhead on disk-intensive workloads. An alternative approach is to co-locate capabilities with disk blocks by placing them immediately before a disk block's data [54]. Unfortunately, on common hardware, reserving space for a capability would prevent blocks from being multiples of the page size, adding overhead and complexity to disk operations.

Self-descriptive meta data. Our first serious attempt at efficient disk multiplexing provided a means for each instance of meta data to describe itself. For example, a disk block would start with some number of bytes of application-specific data and then say "the next ten integers are disk block pointers." The complexity of space-efficient self-description caused us to limit what meta data could be described. We discovered that this approach both caused unacceptable amounts of space overhead and required excessive effort to modify existing file system code, because it was difficult to shoehorn existing file system data structures into a universal format.

Template-based description. Self-description and its problems were eliminated by the insight that each file system is built from only a handful of different on-disk data structures, each of which can be considered a type. Since the number of types is small, it is feasible to describe each type only once per file system—rather than once per instance of a type—using a *template*.

Originally, templates were written in a declarative description language (similar to that used in self-descriptive meta data) rather than UDFs. This system was simple and better than self-descriptive meta data, but still exhibited what we have come to appreciate as an indication that applications do not have enough control: the system made too many tradeoffs. We had to make a myriad of decisions about which base types were available and how they were represented (how large disk block pointers could be, how the type layout could change, how extents were specified). Given the variety of on-disk data structures described in the file system literature, it seems unlikely that any fixed set of components will ever be enough to describe all useful meta data.

Our current solution uses templates, but trades the declarative description language for a more expressive, interpreted language—UDFs. This lets libFSes track their own access rights without XN understanding how they do so; XN merely verifies that libFSes track block ownership correctly.

4.4 XN: Design and implementation

We first describe the requirements for XN and then present the design.

Requirements and approach

In our experience so far, the following requirements have been sufficient to reconcile application control with protected sharing.

1. To prevent unauthorized access, every operation on disk data must be guarded. For speed, XN uses *secure bindings* [25] to move access checks to bind time rather than checking at every access. For example, the permission to read a cached disk block is checked when the page is inserted into the page table of the libFS's environment, rather than on every access.

2. XN must be able to determine unambiguously what access rights a principal has to a given disk block. For speed, it uses the UDF mechanism to protect disk blocks using the libFS's own meta data rather than guarding each block individually.

3. XN must guarantee that disk updates are ordered such that a crash will not incorrectly grant a libFS access to data it either has freed or has not allocated. This requirement means that meta data that is persistent across crashes cannot be written when it contains pointers to uninitialized meta data, and that reallocation of a freed block must be delayed until all persistent pointers to it have been removed.

While isolation allows separate libFSes to coexist safely, protected sharing of file system state by mutually distrustful libFSes requires three additional features:

1. Coherent caching of disk blocks. Distributed, per-application disk block caches create a consistency problem: if two applications obliviously cache the same disk block in two different physical pages, then modifications will not be shared. As explained below, XN solves this problem with an in-kernel, system-wide, protected cache registry that maps cached disk blocks to the physical pages holding them. Because XN guards this cache it allows (1) libFSes to trust the mapping of disk block to physical page and (2) cached blocks to persist across process lifetimes.

2. Atomic meta data updates. Many file system updates have multiple steps. To ensure that shared state always ends up in a consistent and correct state, libFSes can lock cache registry entries. (Future work will explore optimistic concurrency control based on versioning.)

3. Well-formed updates. File abstractions above the XN interface may require that meta data modifications satisfy invariants (e.g., that link counts in inodes match the number of associated directory entries). UDFs allow XN to guarantee such invariants in a file-system-specific manner, allowing mutually distrustful applications to safely share meta data.

XN controls only what is necessary to enforce these three protection rules. It attempts to give untrusted libFSes control of all other functionality: I/O initiation, disk block layout and allocation policies, recovery semantics, and consistency guarantees.

The buffer cache registry

The XN buffer cache registry allows protected sharing of disk blocks among libFSes. The registry tracks the mapping of cached disk blocks and their meta data to physical pages (and vice versa). Unlike traditional buffer caches, it only records the mapping, not the disk blocks themselves. Because the registry exists independently of libFSes, it allows cached blocks to persist across process invocations. The disk blocks are stored in application-managed physical-memory pages. The registry tracks both the mapping and its state (dirty, out of core, uninitialized, locked). To allow libFSes to see which disk blocks are cached, the buffer cache registry is mapped read-only into application space.

Access control is performed when a libFS attempts to map a physical page containing a disk block into its address space, rather than when that block is requested from disk. That is, registry entries can be inserted without requiring that the object they describe be in memory. Blocks can also be installed in the registry before their template or parent is known. As a result, libFSes have significant freedom to prefetch.

Registry entries are installed in two ways. First, an application that has write access to a block can directly install a mapping to it into the registry. Second, applications that do not have write access to a block can indirectly install an entry for it by performing a “read and insert,” which tells the kernel to read a disk block, associate it with an application-provided physical page, set the protection of that page page appropriately, and insert this mapping into the registry. This latter mechanism is used to prevent applications that do not have permission to write a block from modifying it by installing a bogus in-core copy.

XN does not replace physical pages from the registry (except for those freed by applications), allowing applications to determine the most appropriate caching policy. Because applications also manage virtual memory paging, the partitioning of disk cache and virtual memory backing store is under application control. To simplify the application's task and because it is inexpensive to provide, XN maintains an LRU list of unused but valid buffers. By default, when LibOSes need pages and none are free, they recycle the oldest buffer on this LRU list.

XN allows any process to write "unowned" dirty blocks to disk (i.e., blocks not associated with a running process), even if that process does not have write permission for the dirty blocks. This allows the construction of daemons that asynchronously write dirty blocks. LibFSes do not have to trust daemons with write access to their files, only to flush the blocks. This ability has three benefits. First, the contents of the registry can be safely retained across process invocations rather than having to be brought in and paged out on creation and exit. Second, this design simplifies the implementations of libFSes, since a libFS can rely on a daemon of its choice to flush dirty blocks even in difficult situations (e.g., if the application containing the libFS is swapped out). Third, this design allows different write-back policies.

Ordered disk writes

XN must guarantee that a system crash cannot cause protection violations — i.e., that losing volatile state will not cause library file systems to incorrectly gain (or lose) access to disk blocks. To this end, XN enforces the rules given by Ganger and Patt [36] achieving strict file system integrity across crashes: First, never reuse an on-disk resource before nullifying all previous pointers to it. Second, never create persistent pointers to structures before they are initialized. Third, when moving an on-disk resource, never reset the old pointer in persistent storage before the new one has been set.

The first two rules are required for global system integrity—and thus must be enforced by XN—while a file system violating the third rule will only affect itself.

The rules are simple but difficult to enforce efficiently: a naive implementation will incur frequent costly synchronous disk writes. XN allows libFSes to address this by enforcing the rules without legislating how to follow them. In particular, libFSes can choose any operation order which satisfies the constraints.

The first rule is implemented by deferring a block's deallocation until all on-disk pointers to that block have been deleted; a reference count performed at crash recovery time helps libFSes implement the third rule.

The second rule is the hardest of the three. To implement it, XN keeps track of *tainted* blocks. Any block is considered tainted if (1) its type allows pointers to other disk blocks and (2) its on disk representation is uninitialized or it points to a tainted block. LibFSes must not be allowed to write a tainted block to disk, since a system crash could then incorrectly grant it access to disk blocks it did not own. However, two exceptions allow XN to enforce the general rule more efficiently.

First, XN allows entire file systems to be marked "temporary" (i.e., not persistent across reboots). Since these file systems are not persistent, they are not required to adhere to any of the integrity rules. This technique allows memory-based file systems to be implemented with no loss of efficiency.

The second exception is based on the observation that unattached subtrees—trees whose root is not reachable from any persistent root—will not be preserved across reboots and thus, like temporary trees, are free of any ordering constraints. Thus, XN does not track tainted blocks in an unreachable tree until it is connected to a persistent root.

4.5 XN usage

To illustrate how XN is used, we sketch how a libFS can implement common file system operations. These two setup operations are used to install a libFS:

Type creation. The libFS describes its types by storing templates, described above in Section 4.2, into a *type catalogue.* Each template is identified by a unique string (e.g., “FFS Inode”). Once installed, types are persistent across reboots.

LibFS persistence. To ensure that libFS data is persistent across reboots, a libFS can register the root of its tree in XN's *root catalogue.* A root entry consists of a disk extent and corresponding template type, identified by a unique string (e.g., “mylibFS”).

After a crash, XN uses these roots to garbage-collect the disk by reconstructing the free map. It does so by logically traversing all roots and all blocks reachable from them: it marks reachable blocks as allocated, non-reachable blocks as free. If this step is too expensive, it could be eliminated by ordering writes to the free map (so that the map always held a conservative picture of what blocks were free) or using a log.

During reconstruction XN also checks for errors in meta data reference counts by counting all pointers to all meta data instances. If this count does not match a meta data's reference count, XN records this violation in an error log. After finding all errors, it then runs libFS-supplied patch programs to fix these and any libFS-specific errors. If errors remain after this process, XN marks the root of any tree that contains a bogus reference count as “tainted.” These errors must be fixed before the tree can be used. We discuss reconstruction further in the next section.

After initialization, the new libFS can use XN. We describe a simplified version of the most common operations.

Startup. To start using XN, a libFS loads its root(s) and any types it needs from the root catalogue into the buffer cache registry. Usually both will already be cached.

Read. Reading a block from disk is a two-stage process, where the stages can be combined or separated. First, the libFS creates entries in the registry by passing block addresses for the requested disk blocks and the meta data blocks controlling them (their *parents*). The parents must already exist in the registry—libFSes are responsible for loading them. XN uses *owns-udf* to determine if the requested blocks are controlled by the supplied meta data blocks and, if so, installs registry entries.

In the second stage, the libFS initiates a read request, optionally supplying pages to place the data in. Access control through *acl-uf* is performed at the parent (e.g., if the data loaded is a bare disk block), at the child (e.g., if the data is an inode), or both.

A libFS can load any block in its tree by traversing from its root entry, or optionally by starting from any intermediate node cached in the registry. Note that XN specifically disallows meta data blocks from being mapped read/write.

To speculatively read a block before its parent is known, a libFS can issue a raw read command. If the block is not in the registry, it will be marked as “unknown type” and a disk request initiated. The block cannot be used until after it is bound to a parent by the first stage of the read process, which will determine its type and allow access control to be performed.

Allocate. A libFS selects blocks to allocate by reading XN's map of free blocks, allowing libFSes to control file layout and grouping. Free blocks are allocated to a given meta data node by calling XN with the meta data node, the blocks to allocate, and the proposed modification to the meta data node. XN checks that the requested blocks are free, runs the appropriate *acl-uf* to see if the libFS has permission to allocate, and runs *owns-udf*, as described in Section 4.2, to see that the correct block is being allocated. If these checks all succeed, the meta data is changed, the allocated blocks are removed from the free list, and any allocated meta data blocks are marked tainted (see Section 4.4).

Write. A libFS writes dirty blocks to disk by passing the blocks to write to XN. If the blocks are not in memory, or they have been pinned in memory by some other application, the write is prevented. The write also fails if any of the blocks are tainted and reachable from a persistent root. Otherwise, the write succeeds. If the block was previously tainted and now is not (either by eliminating pointers to uninitialized meta data or by becoming initialized itself), XN modifies its state and removes it from the tainted list.

Since applications control what is fetched and what is paged out when (and in what order), they can control many disk management policies and can enforce strong stability guarantees.

Deallocate. XN uses UDFs to check deallocate operations analogously to allocate operations. If there are no on-disk pointers to a deallocated disk block, XN places it on the free list. Otherwise, XN enqueues the block on a "will free" list until the block's reference count is zero. Reference counts are decremented when a parent that had an on-disk pointer to the block deletes that pointer via a write.

4.6 Crash Recovery Issues

This section discusses two issues of reconstruction: the difficulty of resolving file system errors due to XN's lack of understanding of libFS semantics, and providing more flexible mechanisms for guaranteeing invariants in the presence of crashes.

As we discussed above, XN marks file system roots that contain meta data with erroneous reference counts as "tainted." An apparently simpler solution would be to fix the counts directly. There are two reasons XN does not do so. First, it does not understand libFS meta data syntax. While it could use libFS "helper" functions to fix reference counts, it cannot rely on being able to do so, since they could contain errors. Second, while XN could perhaps use UDF-style techniques to verify helper functions, the action to take in the face of a violation depends on libFS semantics. For example, consider the case of a erroneously high reference count. It is not clear how to correct this count, since it could have arisen in multiple ways. The meta data could have been being moved from one directory to another by a libFS that conservatively deletes the old "source" edge only after the new "destination" edge has been written to disk. (In which case the old edge would have to be removed.) Or an edge could have been removed before the reference count was persistently decremented. (In which case the reference count should be decremented.) XN takes the view that resolving these ambiguities is best left to libFSes.

Guaranteeing that file system invariants hold across system crashes has been an active area of file system research. However, other than control over write orders, XN provides little flexibility in this area. We discuss improvements below. A simple way to improve XN would be to incorporate the notion of logging into it. Violations and (possibly) libFS annotations could be written into a "write-ahead" log that was written to disk before persistent state was updated. In this case, any on disk violations can be fixed by simply performing the log actions. To increase flexibility, the format of log records could be controlled by a UDF.

This approach works, and does not appear too hard to implement. Unfortunately, it has serious problems. First, it requires that XN pre-empt many design decisions about how to implement logging. Second, and more seriously, it is not robust: it requires that we anticipate logging and put support for it in XN rather than having logging "fall out" of a general recovery mechanism in XN. The main game of exokernels is extensibility — that all uses of a system can be implemented using its interface — having to anticipate a particular extension (logging) shows its interface is weak.

Fortunately, it appears that we can use UDF techniques to allow libFSes to implement recovery in a completely general way. Consider what XN really needs for valid crash recovery: all it must guarantee is that if a libFS violates an invariant that XN cares about, that on reboot either (1) the violation goes away (because it only affected volatile state) or (2) that the libFS tells XN about the violation. A log is one way to get the latter guarantee, there are others. One way is to have the libFS signal violations: rather than forcing libFSes to write blocks out in certain order to prevent violations or using a log to track them persistently, XN has libFSes provide a "reboot UDF" (reboot) that, when given a file system tree walks down the tree emitting any violations. How it detects violations is not XN's concern, it need only verify that reboot will. Conceptually, XN performs this verification by first recording all violations in volatile memory and then, when a libFS wants to do a write that would violate some invariant (e.g., a stably writing a pointer to an uninitialized piece of meta data), XN checks that the associated libFS' reboot UDF would inform it about this violation. XN does so in a way similar to owns verification by (conceptually) running the reboot UDF on the pre-write stable state, performing the disk write, then running the reboot UDF on the post-write stable state. The obvious problem with this approach is efficiency. It is obviously not practical to examine the entire disk on every disk write. Fortunately, it appears that by using both a variation of state partitioning (based on continuation passing [29]) and checkable hints from the file system, we can checkpoint the UDF precisely, to the point that verification need only examine a few bytes in one or two cached disk blocks. We are currently designing this scheme.

4.7 C-FFS: a library file system

This subsection briefly describes C-FFS (co-locating fast file system [37])—a UNIX-like library file system we built—with special reference to additional protection guarantees it provides.

XN provides the basic protection guarantees needed for file system integrity — that both meta data block pointers and reference counts are correct, and that block pointers are correctly typed. But real-world file systems often require other, file-system-specific invariants. For instance, UNIX file systems must ensure the uniqueness of file names within a directory. This type of guarantee can be provided in any number of ways: in the kernel, in a server, or, in some cases, by simple defensive programming. C-FFS currently downloads methods into the kernel to check its invariants. We are currently developing a system similar to UDFs that can be used to enforce type-specific invariants in an efficient, extensible way.

Our experience with C-FFS shows that, even with the strongest desired guarantees, a protected interface can still provide significant flexibility to unprivileged software, and that the exokernel approach can deal as readily with high-level protection requirements as it can with those closer to hardware.

C-FFS makes four main additions to XN's protection mechanisms:

1. Access control: it maps the UNIX representation and semantics of access control (uids and gids, etc.) to those of exokernel capabilities.

2. Well-formed updates: C-FFS guarantees UNIX-specific file semantics: for example, that directories contain legal, aligned file names.

3. Atomicity: C-FFS performs locking to ensure that its data is always recoverable and disk writes only occur when meta data is internally consistent.

4. Implicit updates: C-FFS ensures that certain state transitions are implicit on certain actions. Some examples are that modification times are updated when file data are changed, and that renaming or deleting a file updates the name cache.

It is not difficult to implement UNIX protection without significantly degrading application power. C-FFS protection is implemented mainly by a small number of if-statements rather than by procedures that limit flexibility. The most intricate operation—ensuring that files in a directory have unique names—is less than 100 lines of code that scans through a linked list of cached directory blocks to ensure name uniqueness.

4.8 Discussion

Similar to how XN uses UDFs to interpret libFS meta data, libFSes can use UDFs to traverse other file system's meta, even when they do not understand its syntax. (Of course, modification of this meta data typically requires such understanding.)

UDFs add a negligible cost to XN. As the next Chapter discusses, we have run file system benchmarks with and without XN enabled and its overhead (and, thus, the overhead of UDFs) is lost in experimental noise. There are two reasons for this. First, protection tends to be off of the critical path. Second, XN operations tend to be embedded in heavy-weight disk operations. For example, most block operations on cached blocks can simply interact with the buffer cache registry, rather than using to UDFs. Even if neither of these two conditions held, UDFs would not add significant overhead: most UDFs tend to be fairly simple, and if they were directly executed rather than interpreted, cost a few tens of instructions.

XN allows "nestable" extension of file systems. Because XN verifies the correctness of reference counts, pointers, and meta data interpreters, it allows untrusted implementors to extend an existing file system without compromising its integrity. Thus, it is possible to add an entirely new directory type to a file system and have it point to old types, perform access control on them, etc. without the existing implementation open to malice. We do not know of any other way to achieve this same result.

Stable storage is the most challenging resource we have multiplexed. Future work will focus on two areas. First, we plan to implement a range of file systems (log-structured file systems, RAID, and memory-based file systems), thus testing if the XN interface is powerful enough to support concurrent use by radically different file systems. Second we will investigate using lightweight protected methods like UDFs to implement the simple protection checks required by higher-level abstractions.

Chapter 5

Performance of exokernel systems

The fundamental principle of science, the definition almost, is this: the sole test of the validity of any idea is experiment. – Richard P. Feynman

This chapter tests whether the exokernel architecture delivers on its promises. While we do so by measuring a specific exokernel system, we believe that our results generalize: other exokernels will share the same characteristic — untrusted libOSes — that give this one good performance. Four questions drive our experiments:

1. Do common, unaltered applications benefit from the exokernel architecture? To answer this question we present performance results of Unix applications on Xok that have been linked against an optimized libFS. Applications on Xok run comparably or significantly faster compared to both FreeBSD and OpenBSD.

2. Is exokernel flexibility costly? To partially answer this we present performance numbers of the previous experiment running on top of a re-implementation of the libFS, the C-FFS file system, within OpenBSD. The exokernel and OpenBSD perform roughly comparably.

3. Are aggressive applications significantly times faster? Among other experiments, we compare the performance of an optimized webserver, which is eight times faster than on traditional systems.

4. Does local control lead to bad global performance? An exokernel gives applications significantly more control than traditional operating systems do. Can it reconcile strong local control with good global performance?

   To answer this question we measure aggressive workloads on Xok and FreeBSD: (1) given the same workload, an exokernel performs comparably to widely used monolithic systems, and (2) when local optimizations are performed, that whole system performance improves, sometimes significantly.

We describe our experimental environment below. The remainder of the chapter addresses each question in turn.

Benchmark	Description (application)
Copy small file	copy the compressed archived source tree (cp)
Uncompress	uncompress the archive (gunzip)
Copy large file	copy the uncompressed archive (cp)
Unpack file	unpack archive (pax)
Copy large tree	recursively copy the created directories (cp).
Diff large tree	compute the difference between the trees (diff)
Compile	compile source code (gcc)
Delete files	delete binary files (rm)
Pack tree	archive the tree (pax)
Compress	compress the archive tree (gzip)
Delete	delete the created source tree (rm)

Table 5.1: The I/O-intensive workload installs a large application (the lcc compiler). The size of the compressed archive file for lcc is 1.1 MByte.

5.1 Xok Experimental Environment

We compare Xok/ExOS to both FreeBSD 2.2.2 and OpenBSD 2.1 on the same hardware. Xok uses device drivers that are derived from those of OpenBSD. ExOS also shares a large source code base with OpenBSD, including most applications and most of libc. Compared to OpenBSD and FreeBSD, ExOS has not had much time to mature; we built the system in less than two years and moved to the x86 platform only a year ago.

All experiments are performed on 200-MHz Intel Pentium Pro processors with a 256-KByte on-chip L2 cache and 64-MByte of main memory. The disk system consists of an NCR 815 SCSI controller connecting a fast SCSI chain with one or more Quantum Atlas XP32150 disk drives on the PCI bus (vs440fx PCI chip set). Reported times are the minimum time of ten trials (the standard deviations of the total run times are less than three percent).

As discussed in Chapter 3, some ExOS data structures do not have full protection. To compensate for this lack, we have artificially inserted three extra system calls before every write to shared tables. This gives a pessimal feel for the cost of protection. All measurements reported in this thesis include these extra calls. In practice, protection is off the critical path, and these overheads are lost in experimental noise.

It is important to note that a sufficiently motivated kernel programmer can implement any optimization that is implemented in an extensible system. In fact, a member of our research group, Costa Sapuntzakis, has implemented a version of C-FFS within OpenBSD. Extensible systems (and we believe exokernels in particular) make these optimizations significantly easier to implement than centralized systems do. For example, porting C-FFS to OpenBSD took more effort than designing C-FFS and implementing it as a library file system. The experiments below demonstrate that by using unprivileged application-level resource management, any skilled programmer can implement useful OS optimizations. The extra layer of protection required to make this application-level management safe costs little.

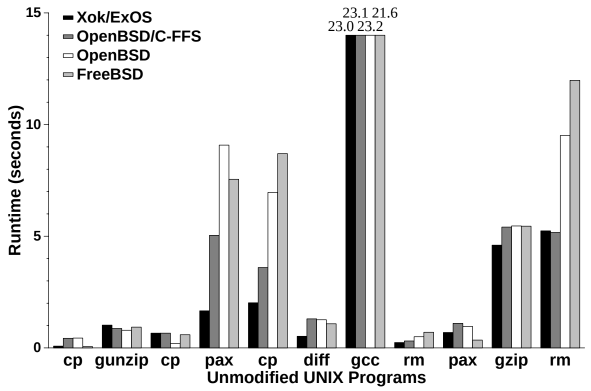

Figure 5-1: Performance of unmodified UNIX applications. Xok/ExOS and OpenBSD/C-FFS use a C-FFS file system while Free/OpenBSD use their native FFS file systems. Times are in seconds.

5.2 Performance of common, unaltered applications

If an exokernel only improves the performance of strange, niche applications or, requires that applications be modified to benefit, then its usefulness is severely diminished.

Our experiments show that an exokernel matters, even for common applications. We measure the performance of an I/O-intensive software development workload made up of the mainstream applications listed in Table 5.1 (most are found in "/usr/bin" on a Unix system). As Figure 5-1 shows, by linking against an optimized library file system (C-FFS), some unaltered UNIX applications run significantly faster on top of Xok/ExOS than identical versions executed on traditional systems. Xok/ExOS completes all benchmarks in 41 seconds, 19 seconds faster than FreeBSD and OpenBSD. On eight of the eleven benchmarks Xok/ExOS performs better than Free/OpenBSD (in one case by over a factor of four). ExOS's performance improvements are due to its C-FFS file system.

In general, normal applications benefit from an exokernel by being linked against an optimized libOS. Thus, even without modifications, they still profit from an exokernel. While an exokernel allows experimentation with completely different OS interfaces, a more important result may be improving the rate of innovation of implementations of existent interfaces.

We also ran the Modified Andrew Benchmark (MAB) [69]. On this benchmark, Xok/ExOS takes 11.5 seconds, OpenBSD/C-FFS takes 12.5 seconds, OpenBSD takes 14.2 seconds, and FreeBSD takes 11.5 seconds. The difference in performance on MAB is less profound than on the I/O-intensive benchmark, because MAB stresses fork, an expensive function in Xok/ExOS. ExOS's fork performance suffers because Xok does not yet allow environments to share page tables. Fork takes six milliseconds on ExOS, compared to less than one millisecond on OpenBSD.

5.3 The cost of exokernel flexibility

An exokernel provides extreme flexibility. Rightfully, any systems builder would expect that this flexibility would have a performance cost. This section looks at two possible costs: (1) the opportunity cost of OS abstractions at library level, and (2) the brute cost of protection,

5.3.1 The cost of OS abstractions in libraries

Given the same application code and same OS code (placed in a libOS on an exokernel, in the kernel on a monolithic system), does a traditional system would provide superior performance? Or, phrased another way, if an optimization is done on an exokernel and gives a factor of four, would the same optimization done on a traditional OS give even more?

To partially answer this question we measure the performance of an OpenBSD-based implementation of C-FFS (done by Costa Sapuntzakis) on the workload in the previous experiment.

Figure 5-1 shows the performance of these applications over Xok/ExOS and OpenBSD/C-FFS. The total running time for Xok/ExOS is 41 seconds and for OpenBSD/C-FFS is 51 seconds. Since ExOS and OpenBSD/C-FFS use the same type of file system, one would expect that ExOS and OpenBSD perform equally well. As can be seen in Figure 5-1, Xok/ExOS performance is indeed comparable to OpenBSD/C-FFS on eight of the 11 applications. On three applications (pax, cp, diff), Xok/ExOS runs considerably faster (though we do not yet have a good explanation for this).

From these measurements we conclude that, even though ExOS implements the bulk of the operating system at the application level, common software development operations on Xok/ExOS perform comparably to OpenBSD/C-FFS. They demonstrate that—at least for this common domain of applications—an exokernel's flexibility can be provided for free: even without aggressive optimizations ExOS's performance is comparable to that of mature monolithic systems. The cost of low-level multiplexing is negligible: an optimization done on an exokernel can give the same performance improvement as done on a traditional system.

5.3.2 The cost of protection

While the above experiment by no means “proves” that the exokernel structure will never penalize applications, it agrees with our experiences that exokernel flexibility does not impose overheads. The main reason for this is that protection tends to be off the critical path. For example, when the benchmarks in in Figure 5-1 are run without XN or any of the extra system calls, the performance difference is “noise”: 39.7 seconds to 41.1 seconds, despite reducing the overall number of Xok system calls from 300,000 to 81,000. Real workloads are dominated by costs other than protection and system call overhead.

Some secondary, more specific reasons for the lack of impact of an extra protection layer are that: in many cases exokernel protection is not a duplication, since it allows library operating systems to remove their corresponding checks. Furthermore, the cost of checking protection (usually a table lookup to map principals to access rights) tends to be dwarfed by the cost of the operations it is coupled to. For example, system calls are roughly an order of magnitude more expensive while I/O operations such as disk or network accesses can be over a 1000 times more costly. In those rare cases where protection checks are more elaborate, they tend to be naturally cachable. In the case of file blocks, for instance, access rights can be stored in the buffer cache along with the block they correspond too.

While an exokernel can impose extra protection checks, it also makes some operations cheaper: using a library operating system means that many operations that formerly required system calls now can be performed with function calls. Thus, for some uses an exokernel is intrinsically faster than a more traditional system. [1] For example, an OpenBSD emulator written on our most recent exokernel, Xok, sometimes runs OpenBSD application binaries faster than their non-emulated performance on OpenBSD for the simple reason that that many system calls (e.g., to read data structures) become function calls into the emulator's libOS.

5.4 Aggressive application performance

In part, the exokernel architecture was motivated by the ability to perform application-specific or, more commonly, domain-specific optimizations. [2] Two natural questions, then, are first, does an exokernel give sufficient power that interesting optimizations can be done? Second, do domain-specific optimizations yield significant improvements or do they only give “noise” level improvements?

Experiments show that an exokernel enables domain-specific optimizations that give order-of-magnitude performance improvements [48]. Our specific results come from an interface designed for fast I/O, which improves the performance of applications such as web servers.

5.4.1 XCP: a “zero-touch” file copying program

XCP is an efficient file copy program. It exploits the low-level disk interface by removing artificial ordering constraints, by improving disk scheduling through large schedules, by eliminating data touching by the CPU, and by performing all disk operations asynchronously.

Given a list of files, XCP works as follows. First, it enumerates and sorts the disk blocks of all files and issues large, asynchronous disk reads using this schedule. (If multiple instances of XCP run concurrently, the disk driver will merge the schedules.) Second, it creates new files of the correct size, overlapping inode and disk block allocation with the disk reads. Finally, as the disk reads complete, it constructs large writes to the new disk blocks using the buffer cache entries. This strategy eliminates all copies; the file is DMAed into and out of the buffer cache by the disk controller—the CPU never touches the data.

XCP is a factor of three faster than the copy program (CP) on Xok/ExOS that uses UNIX interfaces,

[1] Admittedly, in our experience, even high system call overhead has little impact on real application performance. But this point argues even further that having operating system code in a library is not a limiting performance issue.

[2] In fact, the original exokernel paper [25] focuses almost exclusively on application-specific optimizations rather improving the rate of whole-system innovations, which we have come to regard as a more significant benefit.

irrespective of whether all files are in core (because XCP does not touch the data) or on disk (because XCP issues disk schedules with a minimum number of seeks and the largest contiguous ranges of disk blocks).

The fact that the file system is an application library allows us both to have integration when appropriate and to craft new abstractions as needed. This latter ability is especially profitable for the disk both because of the high cost of disk operations and because of the demonstrated reluctance of operating systems vendors to provide useful, simple improvements to their interfaces (e.g., prefetching, asynchronous reads and writes, fine-grained disk restructuring and “sync” operations).

5.4.2 The Cheetah HTTP/1.0 Server

The exokernel architecture is well suited to building fast servers (e.g., for NFS servers or web servers). Server performance is crucial to client/server applications [45], and the I/O-centric nature of servers makes operating system-based optimizations profitable.

Greg Ganger has developed an extensible I/O library (XIO) for fast servers and a sample application that uses it, the Cheetah HTTP server. This library is designed to allow application writers to exploit domain-specific knowledge and to simplify the construction of high-performance servers by removing the need to “trick” the operating system into doing what the application requires (e.g., Harvest [14] stores cached pages in multiple directories to achieve fast name lookup).

An HTTP server's task is simple: given a client request, it finds the appropriate document and sends it. The Cheetah Web server performs the following set of optimizations as well as others not listed here.

Merged File Cache and Retransmission Pool. Cheetah avoids all in-memory data touching (by the CPU) and the need for a distinct TCP retransmission pool by transmitting file data directly from the file cache using precomputed file checksums (which are stored with each file). Data are transmitted (and retransmitted, if necessary) to the client directly from the file cache without CPU copy operations. (Cao et al. have also used this technique [70].)

Knowledge-based Packet Merging. Cheetah exploits knowledge of its per-request state transitions to reduce the number of I/O actions it initiates. For example, it avoids sending redundant control packets by delaying ACKs on client HTTP requests, since it knows it will be able to piggy-back them on the response. This optimization is particularly valuable for small document sizes, where the reduction represents a substantial fraction (e.g., 20%) of the total number of packets.

HTML-based File Grouping. Cheetah co-locates files included in an HTML document by allocating them in disk blocks adjacent to that file when possible. When the file cache does not capture the majority of client requests, this extension can improve HTTP throughput by up to a factor of two.

Figure 5-2 shows HTTP request throughput as a function of the requested document size for five servers: the NCSA 1.4.2 server [68] running on OpenBSD 2.0, the Harvest cache [14] running on OpenBSD 2.0, the base socket-based server running on OpenBSD 2.0 (i.e., our HTTP server without any optimizations), the base socket-based server running on the Xok exokernel system (i.e., our HTTP server without any optimizations

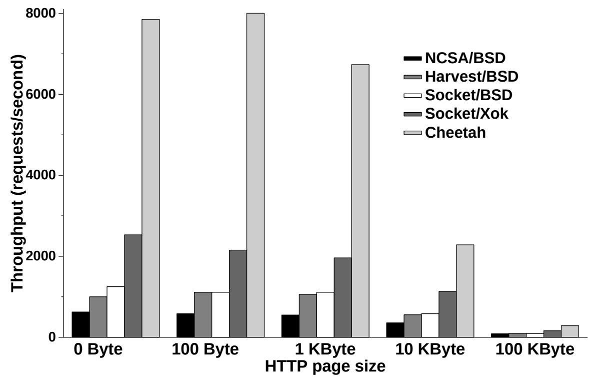

Figure 5-2: HTTP document throughput as a function of the document size for several HTTP/1.0 servers. **NCSA/BSD** represents the NCSA/1.4.2 server running on OpenBSD. **Harvest/BSD** represents the Harvest proxy cache running on OpenBSD. **Socket/BSD** represents our HTTP server using TCP sockets on OpenBSD. **Socket/Xok** represents our HTTP server using the TCP socket interface built on our extensible TCP/IP implementation on the Xok exokernel. **Cheetah/Xok** represents the Cheetah HTTP server, which exploits the TCP and file system implementations for speed.

with vanilla socket and file descriptor implementations layered over XIO), and the Cheetah server running on the Xok exokernel (i.e., our HTTP server with all optimizations enabled).

Figure 5-2 provides several important pieces of information. First, our base HTTP server performs roughly as well as the Harvest cache, which has been shown to outperform many other HTTP server implementations on general-purpose operating systems. Both outperform the NCSA server. This gives us a reasonable starting point for evaluating extensions that improve performance. Second, the default socket and file system implementations built on top of XIO perform significantly better than the OpenBSD implementations of the same interfaces (by 80–100%). The improvement comes mainly from simple (though generally valuable) extensions, such as packet merging, application-level caching of pointers to file cache blocks, and protocol control block reuse.

Third, and most importantly, Cheetah significantly outperforms the servers that use traditional interfaces. By exploiting Xok's extensibility, Cheetah gains a four times performance improvement for small documents (1 KByte and smaller), making it eight times faster than the best performance we could achieve on OpenBSD. Furthermore, the large document performance for Cheetah is limited by the available network bandwidth (three 100Mbit/s Ethernets) rather than by the server hardware. While the socket-based implementation is limited to only 16.5 MByte/s with 100% CPU utilization, Cheetah delivers over 29.3 MByte/s with the CPU idle over 30% of the time. The extensibility of ExOS's default unprivileged TCP/IP and file system implementations made it possible to achieve these performance improvements incrementally and with low complexity.

The optimizations performed by Cheetah are architecture independent. In Aegis, Cheetah obtained similar performance improvements over Ultrix web servers [49].

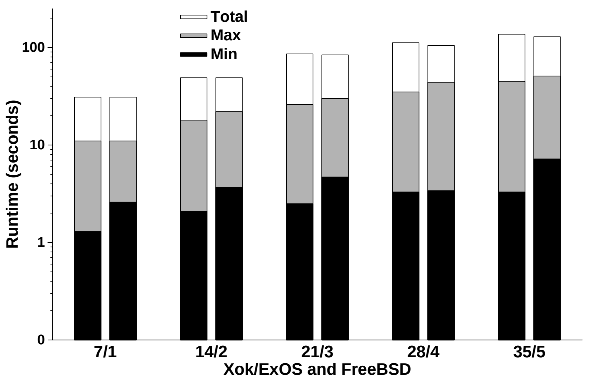

Figure 5-3: Measured global performance of Xok/ExOS (the first bar) and FreeBSD (the second bar), using the first application pool. Times are in seconds and on a log scale. *number/number* refers to the the total number of applications run by the script and the maximum number of jobs run concurrently. **Total** is the total running time of each experiment, **Max** is the longest runtime of any process in a given run (giving the worst latency). **Min** is the minimum.

5.5 Global performance

An exokernel gives applications significantly more control than traditional operating systems do. However, it must also guarantee good global performance when running multiple applications concurrently. The experiments in this section measure the situation where the exokernel architecture seems potentially weak: under substantial load where selfish applications are consuming large resources and utilizing I/O devices heavily. The results indicate that an exokernel can successfully reconcile local control with global performance: (1) given the same workload, an exokernel performs comparably to widely used monolithic systems, and (2) that when local optimizations are performed, that whole system performance improves, and can do so significantly.

There are two intuitions behind these results. First, most local optimizations, because they make applications run faster, lead to more resources globally. For example, if an application, after being linked against an optimized libOS cuts its runtime from ten seconds to one second, then there are nine seconds of freed resources to go around the entire system. Second, an exokernel mediates allocation and revocation of resources. Therefore it has the power to enforce any global policy that a traditional operating system can. Thus, all else equal, it has no problem achieving similar global performance. The single new challenge an exokernel faces is deriving information lost by dislocating abstractions into application space. For example, traditional operating systems manage both virtual memory and file caching. As a result, they can perform global resource management of pages that takes into account the manner in which a page is being used. In contrast, if an exokernel dislocates virtual memory and file buffer management into library operating systems it no longer can make such distinctions. While such information matters in theory, in practice we have found it either unnecessary or crude enough that no special methods have been necessary to derive it. However, whether this happy situation always holds is an open question.

5.5.1 Experiments

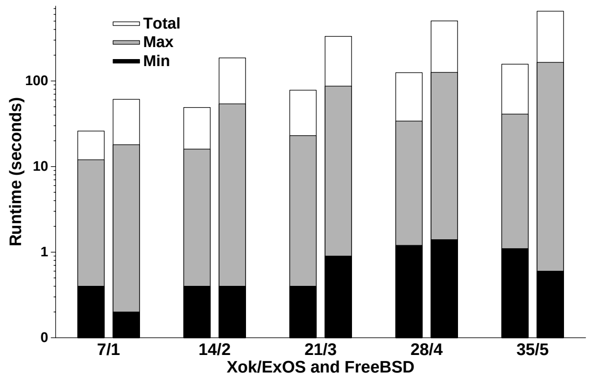

Figure 5-4: Measured global performance of Xok/ExOS (the first bar) and FreeBSD (the second bar), using the second application pool. Methodology and presentation are as described for Figure 5-3.

Global performance has not been extensively studied. We use the total time to complete a set of concurrent tasks as a measure of system throughput, and the minimum and the maximum latency of individual applications as a measure of interactive performance. For simplicity we compare Xok/ExOS's performance under high load to that of FreeBSD; in these experiments, FreeBSD always performs better than OpenBSD, because of OpenBSD's small, non-unified buffer cache. While this methodology does not guarantee that an exokernel can compare to any centralized system, it does offer a useful relative metric.

The space of possible combinations of applications to run is large. The experiments use randomization to ensure we get a reasonable sample of this space. The inputs are a set of applications to pick from, the total number to run, and the maximum number that can be running concurrently. Each experiment maintains the number of concurrent processes at the specified maximum. The outputs are the total running time, giving throughput, and the time to run each application. Poor interactive performance will show up as a high minimum latency.

The first application pool includes a mix of I/O-intensive and CPU-intensive programs: pack archive (pax -w), search for a word in a large file (grep), compute a checksum many times over a small set of files (cksum), solve a traveling salesman problem (tsp), solve iteratively a large discrete Laplace equation using successive overrelaxation (sor), count words (wc), compile (gcc), compress (gzip), and uncompress (gunzip). For this experiment, we chose applications on which both Xok/ExOS and FreeBSD run roughly equivalently. Each application runs for at least several seconds and is run in a separate directory from the others (to avoid cooperative buffer cache reuse). The pseudo-random number generators are identical and start with the same seed, thus producing identical schedules. The applications we chose compete for the CPU, memory, and the disk.

Figure 5-3 shows on a log scale the results for five different experiments: seven jobs with a maximum concurrency of one job through 35 jobs with a maximum concurrency of five jobs. The results show that an exokernel system can achieve performance roughly comparable to UNIX, despite being mostly untuned for global performance.

With a second application pool, we examine global performance when specialized applications (emulated by applications that benefit from C-FFS's performance advantages) compete with each other and non-specialized applications. This pool includes tsp and sor from above, unpack archive (pax -r) from Section 5.2, recursive copy (cp -r) from Section 5.2, and comparison (diff) of two identical 5 MB files. The pax and cp applications represent the specialized applications.

Figure 5-4 shows on a log scale the results for five experiments: seven jobs with a maximum concurrency of one job through 35 jobs with a maximum concurrency of 5 jobs. The results show that global performance on an exokernel system does not degrade even when some applications use resources aggressively. In fact, the relative performance difference between FreeBSD and Xok/ExOS increases with job concurrency.

5.5.2 Discussion

The central challenge in an exokernel system is not *enforcing* a global system policy but, rather, *deriving* the information needed to decide what enforcement involves and doing so in such a way that application flexibility is minimally curtailed. Since an exokernel controls resource allocation and revocation, it has the power to enforce global policies. Quota-based schemes, for instance, can be trivially enforced using only allocation denial and revocation. Fortunately, the crudeness of successful global optimizations allows global schemes to be readily implemented by an exokernel. For example, Xok currently tracks global LRU information that applications can use when deallocating resources.

We believe that an exokernel can provide global performance *superior* to current systems. First, effective local optimization can mean there are more resources for the entire system. Second, an exokernel gives application writers machinery to orchestrate inter-application resource management, allowing them to perform domain-specific global optimizations not possible on current centralized systems (e.g., the UNIX “make” program could be modified to orchestrate the complete build process). Third, an exokernel can unify the many space-partitioned caches in current systems (e.g., the buffer cache, network buffers, etc.). Fourth, since applications can know when resources are scarce, they can make better use of resources when layering abstractions. For example, a web server that caches documents in virtual memory could stop caching documents when its cache does not fit in main memory. Future research will pursue these issues.

5.5.3 Summary

> Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that! Charles Lutwidge Dodgson (Lewis Carroll) (1832-1898) “Through the Looking Glass”

Our experiments show that even common, unaltered applications can benefit on exokernels, simply by being linked against an optimized library operating system. Importantly, libOS optimizations appear just as effective as their equivalent in-kernel implementation. Aggressive applications that want to manage their resources show even greater improvements. The improvement is especially dramatic for I/O-centric applications, such as our web server, which runs up to a factor of 8 faster than its closest equivalent. Finally, the power that an exokernel gives to applications does not lead to poor global performance. In fact, when this control is used to improve application speed, an exokernel system can have dramatically improved global performance, since there are more resources to go around. Based on these experiments, the exokernel architecture appears to be a promising alternative to traditional systems.

Chapter 6

Reflections on Downloading Code

The individual's whole experience is built upon the plan of his language. — Henri Delacroix

Extensibility refers roughly to how easily a system's functionality can be augmented. A strong thread in computer science has been developing techniques to enhance extensibility, ranging from programming methodologies such as structured programming to assist program modification, to dynamic linking of device drivers to add new functions to an operating system kernel. Using language to build extensible systems has had a venerable tradition: the widely-used text editor emacs has done so since its inception [11, 81], database systems exploit it to enrich queries and extend data types, and more recently web browsers and servers have used it to extend their base functionality.

A variety of operating systems have allowed applications to download untrusted code into them as a way to extend their functionality [9, 22, 25, 32, 48, 71, 79, 80, 92]. This chapter documents experiences drawn from the exokernel systems described in this thesis. These experiences cover a period of four years, and span numerous rethinkings of the role of downloaded code, and, as well, much belated realization of its implications and misuses.

The ability to download code has subtle implications. This chapter's central contribution is its perspective on the abilities downloaded code grants and removes, as well as its concrete examples of how these gained and lost abilities matter in practice. Some specific insights include:

1. "Infinite" extensibility requires Turing completeness, Turing completeness gives infinite extensibility.

   Solving the negative problem of extensibility requires supporting all unanticipated uses. A guaranteed solution is to let applications inject general-purpose code into the system, thereby granting them the ability to implement any computable policy or mechanism. (An alternative code motion, uploading operating system code into the application, provides the same guarantee.)

   Conversely, an interface striving for infinite generality is implicitly attempting to provide Turing completeness. Explicitly realizing this fact leads to the obvious solution of having clients pass in general-purpose code.

The following point provides an example:

2. Correct applications track what resources they have access to, rendering the operating system's bookkeeping redundant. Thus, if the operating system can reuse the application's data structures, it can eliminate this redundancy.

   To ensure that the operating system can understand these structures without restricting their implementation, we use the previous point: clients provide a data structure interpreter, written in a Turing complete language, that the operating system uses to extract the bookkeeping information it needs.

   To force these interpreters to be correct, we use the following technique:

3. Inductive incremental testing provides a practical way to verify the correctness (not just mere safety) of deterministic code. We call functions amenable to this approach *untrusted deterministic functions* (UDFs).

   Using UDFs, operating systems can avoid pre-determining implementation tradeoffs by leaving implementation decisions to client UDFs. Our most interesting use of UDFs, verifying the resource interpreters described above, lets untrusted file systems track what disk blocks they own without the operating system understanding how, yet without being vulnerable to malice.

4. There are practical nuances between using code or data to orchestrate actions between entities. Two examples follow.

   Data is not a Turing machine: compared to downloaded code, data is transparent, and its operations (read and write) trivially bounded in cost and guaranteed to terminate. Code is not, necessarily, any of these things. Injecting potentially non-terminating black boxes into an operating system does not help simplicity. We only use downloaded code in a few specific instances, and have removed it more than once.

   Code requires indirection. Imposing code between the operating system and its data forces it to go through a layer of potentially non-terminating indirection. For example, since our disk subsystem relies on client UDFs to interpret file system structures, it cannot modify them directly, but requires assistance for operations as simple as changing one disk block pointer to another in order to compact the disk.

5. The main benefit of downloaded code is *not* execution speed, but rather trust and consequently power.

   While we started with the view that downloading code was useful for speed (e.g., to eliminate the cost of kernel/user boundary crossings) [25] it has turned out to be far more crucial for power: because downloaded code can be controlled, it can be safely granted abilities that external, unrestricted application code cannot.

   For example, since downloaded file system UDFs can be verified, they can be trusted to track that file system's disk blocks. Obviously, unrestricted applications cannot similarly be trusted.

The chapter is organized as follows. The next four sections discuss different language-based subsystems, which form the spine for our experiences and lessons: DPF [31], our packet filter engine; *application specific message handlers* (ASHs) [25, 92, 93], a networking system which invokes downloaded code on message arrival; XN [48], the disk protection subsystem described in Chapter 4, which contains our most interesting use of downloaded code; and finally, *protected methods*, which applications use to enforce invariants on shared state. Section 6.5 explores some of the slippery implications of using computational building blocks in lieu of passive procedural interfaces. Section 6.6 links our experiences to the existent literature on extensible systems.

6.1 DPF: dynamic packet filters

Packet filters are one of the most successful examples of downloaded code: most modern operating systems provide support for them. This section discusses our packet filter system, DPF ("dynamic packet filters") [25, 31], which was briefly discussed in Chapter 3. DPF uses a combination of language and compilation techniques to get speed and protection. We focus on what abilities downloading code granted, along with some lessons, including our in-hindsight naive mistakes applying dynamic compilation to packet filters.

6.1.1 Language design

The main challenge in DPF is effective detection of filter overlap: i.e., knowing when two filters are comparing the same message offsets to the same values. Protection requires this feature, since otherwise an application could easily steal another' s messages. Additionally, overlap detection aids efficiency, since it enables merging of overlapping filter segments [6, 95]. Such merging is crucial for scalability. Typical systems have many filters simultaneously active, one for each network connection.

We made the DPF language declarative, unlike previous packet filter languages [65, 64, 95]. A lower-level imperative language artificially complicates overlap detection due to the presence of operationally different yet functionally equivalent instruction sequences. A declarative language allowed us to avoid puzzling out such artifacts. [1]

A declarative language also assists a sophisticated optimizer, since it makes programmer intent clearer. The small simplicity of the DPF language (e.g., the lack of loops, aliasing, variables) made constructing a sophisticated in-kernel compiler tractable. Given more primitive compilation technology, an imperative language would have been more appropriate, since it would allow a clever programmer to implement optimizations the compiler would not otherwise do.

DPF uses dynamic compilation to compile filters, which makes such semantic incorporation easy. Its optimizations include encoding filter constants in the instruction stream rather than being loaded from a

[1] Independently of our work, the PATHFINDER packet filter language was also designed declaratively, though here this form made it easier to put into hardware [6].

data structure, coalescing comparisons of adjacent message values, estimating alignment of message loads to eliminate the pessimal use of unaligned memory loads and, finally, eliminating bounds checks.

The most unusual optimization DPF does is *hash table compilation*. When filters compare the same message offset to the same value we merge them. However, when they compare to different values we create a hash table holding the constant each filter compares to. DPF uses dynamic code generation to compile this hash table to executable code. For example, if the hash table has no collisions, the lookup can elide collision checks. If there are a small number of keys (say 8 or less) it instead generates a specialized binary search that hard codes each key as an immediate value in the instruction stream. Otherwise it creates a jump table. Additionally, since the number and value of keys are known at runtime, DPF can select among several hash functions to obtain the best distribution, and then encode the chosen function in the instruction stream.

6.1.2 Downloaded code provides power

Because downloaded code can be controlled, it can be safely granted abilities that external, unrestricted application code cannot.

Because packet filters can be restricted (to terminate quickly, and to not overlap) they can be trusted to correctly claim incoming packets. The alternative, asking applications if they want a particular packet, is clearly untenable since there is nothing preventing an application from claiming all packets. The each of the remaining three subsystems we describe illustrate this point.

To restrict code, download it. In general, code on the other side of a trust boundary cannot be prevented from doing arbitrary computations, nor from communicating. Embedding downloaded code in a trusted execution context allows the host to completely control the code, preventing actions otherwise impossible to restrict. DPF overlap detection is one example of restricting computation. Another is letting filters see packets they do not own, since the they cannot leak packet contents to their associated applications. An example outside the context of this chapter is Myers' information flow control work [66], which uses a trusted compiler and runtime environment to prevent applications from leaking sensitive information.

6.1.3 Some lessons

The cost of extensibility. Compared to “hard-wired” systems, extensibility can add noticeable complexity. A static packet demultiplexer is roughly an order of magnitude smaller than DPF's three thousand lines of code. It is simpler as well, consisting of a sequence of conditions and a hash table implementation. The benefits of enabled functionality can make introduction of this layer of indirection worthwhile, but they rarely come without cost.

Code is data, but data is not code. In a theoretical sense, all code is data. However, in a practical sense, some code is more data than others. Data changes frequently, code does not. As a result, it is significantly more difficult to dynamically compile data than code. Data's rate of change typically requires that the implementor use self-modifying code and, worse, understand how to make it fast on modern architectures that penalize the operations it needs.

While a single packet filter code fragment is constant during its life in the kernel, the trie created by merging filters changes incessantly, potentially on each insertion. In a sense, by merging filters we have transmuted them from code (changing rarely) to data (changing frequently). Our first real implementation of DPF missed the implications of this transformation. We naively treated the trie as slowly changing code in that we potentially regenerated the entire structure on insertion of a new filter [31]. This regeneration did not scale at all with large number of branches (which correspond to protocols) in the trie. A second implementation made filter insertion cheaper by incrementally patching the previously generated code. Conceptually, this change was simple: instead of representing each filter operation of “load message value and compare” as a basic block, it became a code fragment linked to the next fragment by an indirect jump. This design isolated the effects of adding a new filter to patching the jumps threading the code together using either self-modifying code or, on machines where instruction cache flushing is expensive, indirect jumps.

While this modification is conceptually trivial, it conflicts with modern architecture trends. RISC processors require programmers manually implement instruction and data cache coherence via explicit cache flushes. In conditions of frequent insertion (e.g., once every six message matches) this operation adds significant overhead, both in the cost of the flush itself, and in the opportunity cost of subsequent cache misses. [2] There are ways to mitigate this cost, but they complicate the code generator. (For example, generating code into memory guaranteed to not be in the instruction cache.) Another problem, banal but real, is that threading requires that pointers be loaded as immediates, which on 64-bit machines can be expensive. Again, fixes are not intellectually deep, but can require tedious bookkeeping.

In contrast, dynamically compiling code is much simpler. The code changes infrequently, if ever, and tends to be used many times before being discarded. As a result, compilation is a “one off” affair and its cost easily recouped. While current dynamic compilation techniques works reasonably well for compiling languages, they must mature further before they can be readily used to compile data structures.

6.2 Application-specific message handlers

This section describes lessons learned in the context of the ASH networking subsystem. [25, 92, 93]. ASHs (application-specific message handlers) are application code downloaded into the operating system, made safe using a variant of software fault isolation [89], and invoked upon message arrival.

An exokernel attempts to place most privileged operating system code in the more forgiving and innovative environment of untrusted software. For example, it uses DPF to enable library-based networking. In a sense, ASHs allow the reverse movement in that they allow applications to exchange a more general environment, where they can run unrestricted in time and in operation, for a more restrictive environment that grants two

[2]For all the drawbacks of its instruction set, the x86 family of machines makes self-modifying code simple, since the prevalence in running applications requires that it work, and not cost outrageously.

specific abilities: tight coupling to clock and network events, and a way incorporate application semantics into OS actions. These two abilities enable libOSes to efficiently handle incoming messages. Tight coupling to interrupts allows low-latency message replies. Control over over message placement eliminates intermediate buffering and its associated copies. The challenge in this motion comes from forcing ASHs to remain unprivileged (otherwise dynamic linking would suffice).

This section focuses on these two ASH abilities and then closes with experiences, including some mistakes.

6.2.1 Pulling application semantics into event handling

ASHs are cheap to invoke (a few tens of instructions), and their runtime bounded. Thus, they can be run in situations where heavy-weight application scheduling is unacceptable. ASHs thus represent a way for applications to decouple actions from their own execution. Frequently, decoupling happens in order to tightly couple the extension to an event which cannot tolerate the overhead of context switching to an application. By decoupling ASHs they can be run whenever a message arrives, unlike applications. An more venerable example is the monitoring system of Deutsch and Grant [22], which allows code fragments to log various kernel events.

More generally, the decoupled nature of ASHs allow them to be invoked *reactively*, when events happen, rather than when an application executes. This fact forms the basis of a common pattern: reactive incorporation of application knowlege into event handling.

Applications have information that OSes can use to make better decisions. Unfortunately, extracting this information can be difficult. Events must be serviced frequently and quickly. Application context-switching is relatively costly and their runtime difficult to bound at a fine-granularity. However, by decoupling code that can make this decision from the application, it can be tamed and run reactively. Both DPF filters and ASHs fit this pattern. Other examples include the page replacement extensions of Cao et al.' [13], and the messaging systems of Edwards et al. [24] and Fiuczynski and Bershad [32].

One way to view the benefit of this pattern is that decisions profit from more information: the ability to make reactive rather than *a priori* decisions in a dynamic environment like networking can be quite useful. Another way is that that event handling is best done at occurrence, if for no other reason than it removes the overhead of buffering and reduces latency. A final way is that, in general, the more semantics that are known, the faster an operation can be performed, in part because it can be specialized (made more appropriate) based on semantic constraints.

For example, inspired by the observations of Clark and Tennenhouse [18], ASHs can integrate operations such as checksumming, byte swapping, or encryption into the copy from network buffer to application space. The ASH copy engine allows ASHs to provide a code snippet, which it integrates into a specialized copy loop using dynamic code generation. This facility allows ASHs to eliminate duplicate data touching steps.

Unlike packet filters, this ASH ability is entirely motivated by speed, not power. The reason for this difference is that the data copy loop does not involve protection — since all operations are done on the ASH's data — and, therefore, need not be restricted in any way. Packet filters, on the other hand, allow untrusted code to compute protection-critical functions, making restrictions essential.

To summarize: (1) tamed code can be made lightweight and, thus, it can be run in situations where application scheduling is infeasible, and (2) this decoupling allows application semantics to be reactively incorporated into event processing.

6.2.2 Discussion

Source level versus object code level sandboxing. Software fault isolation (SFI) can be done either at the source or at the object code level [59, 89]. ASHs were built using object code SFI, which has the theoretical advantage that it works across languages and compilers, and with pre-compiled code. In retrospect, a source level implementation would have been better. Object-level modification is difficult and extremely non-portable, obviously varying across different architectures and object code formats. Even worse, it also varies across different compiler releases due to the practice of commercial vendors of deliberately changing object code formats in undocumented ways in order to stifle third-party competitors [59].

Source-level SFI, because it is tightly integrated within a compiler, is much simpler to develop. It requires the addition of modest, mostly portable operations done at the level of a compiler's intermediate representation. A secondary benefit of integration is that SFI operations are optimized by the host compiler, unlike object SFI implementations. In contrast, object SFI constantly must fight against the fact that it has lost much lost semantic information. For example, object code modification requires that compiler-generated jump tables be relocated, which can be challenging, since simply finding these tables is difficult, typically requiring compiler-specific heuristics.

An obvious disadvantage of using source-level SFI is that it is specific to one compiler back end and whatever languages its front end(s) consume. In theory, this is important. In practice, operating system software is written in the C programming language. On those rare systems where other languages are used, special support would be required even with object code SFI, since these languages typically use a runtime system, which must be adapted to run in an operating system's restrictive context.

A more serious problem is that a source SFI system typically increases the size of the trusted computing base more than object code SFI system does. A specious counter to this problem is the belief that if the trusted compiler is the same as that used to compile the operating system, then the trusted computing base has not really increased since the correctness of that compiler must already be trusted. However, there is a large difference between compiling a kernel correctly and resisting the malice of clever hackers trying to find a hole in 10-100K lines of compiler, assembler, and dynamic linker code.

The lack of a widely available object code SFI system forced us to “roll our own.” Realistically, the reliability of a trusted compiler is most likely better than an object code SFI module we have implemented ourselves.

We no longer use ASHs. In theory, coupling packet arrival to application semantics is profitable. Separate from our work, Edwards et al. [24] provide a way use simple application-provided scripts to direct message placement while Fiuczynski and Bershad [32] provide a fully general messaging system. However, practical technology tradeoffs have made us eliminate ASHs. The three main benefits of ASHs are (1) elimination of kernel crossings, (2) integrated data copying, and (3) fast upcalls to unscheduled processes, thereby reducing processing latency (e.g., of send-response style network messages). On current generation chips, however, the latency of I/O devices is large compared to the overhead of kernel crossings, making the first benefit negligible. The second does not require downloading code, only an upcall mechanism [19]. In practice, it is the latter ability that gives speed. Finally, the presence of DMA hardware makes the data integration ASHs provide irrelevant, since there is no way to add user extensions to the hardware's brute copy.

6.3 XN: efficient disk multiplexing

Language shapes the way we think, and determines what we can think about. — B. L. Whorf

This section explores language issues in Xok's disk multiplexing system, XN, its most “language heavy” subsystem, which contains the most interesting use of downloaded code.

6.3.1 Language Evolution

As discussed in Chapter 4, the languages we use to describe client meta data have gone from through four iterations, becoming increasingly general, lower-level, and abstract. We went from an approach that did not use a language at all to one with an expressive declarative description language (which reading file system literature showed as not expressive enough) to our quasi-Turing complete. One view of the above evolution is as a struggle to define a universal data layout language. A universal language for most domains requires Turing completeness. Once this fact is realized, it becomes obvious that one needs to provide general-purpose computational primitives. Unfortunately, we only made this connection in hindsight after several years of struggles with disk multiplexing.

6.3.2 Insights

Infinite generality requires Turing completeness. A designer attempting to build an infinitely (or even very general) interface or component set is implicitly striving for Turing completeness. Explicit articulation of this fact makes it clear a solution is to allow clients to customize policies and interface implementations using a Turing complete language rather than, say, a “jumble of procedure flags.”

The author belatedly had this insight after struggling with the problem of how to define a completely general set of meta data building blocks and, in fact, only months after coming up with the solution (UDFs) did *why* they solve the problem become clear.

Turing completeness guarantees infinite extensibility. “Solving” extensibility requires showing that any unanticipated use of a system can be implemented. Proving a negative property is hard. A key realization of this chapter is that, when appropriate, Turing completeness provides a way to guarantee that the extensibility problem is solved. For example, the fact that UDFs are (roughly) Turing complete guarantees that that they can describe any computable data layout, anticipated or not.

Transmuting the imperative to the declarative. System implemented functionality is imperative. It determines how to resolve tradeoffs in interface construction: i.e., whether to optimize for latency, throughput, or space. Such predetermination can cause problems when many tradeoffs exist. Downloaded code can be used by a system designer to defer such tradeoffs to clients. By allowing client code to implement functions, the system builder can switch from an imperatively deciding how to implement this function, to declarative testing that client code did so correctly. For example, rather than imperatively deciding how to represent meta data XN declaratively tests that a UDF produces the correct output. This approach has been noticeably easier than the previous struggle to construct a universal data layout language.

The cost of this approach is that testing can be more complex than implementing the functionality (it can also be simpler) and more expensive, though this can be a net win if the algorithm is not on the critical path or grants sufficient power or speed.

Code enables semantic compression. Data representation is important. In a sense, UDFs can be viewed as semantics-exploiting meta data compressors. One could, after all, define a space-inefficient and inflexible but fully general meta data layout. However, UDFs allow representation to be more succinct. For example, much of the meaning of a libFS's meta data is encode in its code, eliminating the need to duplicate this information in the meta data itself (e.g., a libFS “just knows” that certain types of block pointers point to four contigous blocks rather than one). As a more sophisticated example, consider an algebraic relation between blocks such as a file system that allocates blocks at the beginning of every cylinder group. While a predefined data structure would have to list every block, a UDF can encode this knowledge in a function that reads the base block from a instance of meta data and constructs its set:

```
proc owns(meta)
    base = meta >base_block;
    set = {};
    for i = 0 to number_of_partitions
        set = set U { i ∗ blocks_in_cylinder_group + base };
    return set;
```

UDFs can make code transparent. A strength of downloaded code is that it can compute its results however it wishes, in ways the underlying system did not anticipate. However, mysterious result computation can also be a liability: users of the code may want to know when it computes a certain output. For example, consider a library file system function, access, that given a principal and inode, indicates whether that principal is allowed to use the inode (access(inode, pid) $\rightarrow$ bool). Given this function, it is not obvious what values of pid will cause it to return true for a piece of meta data. Thus, an application creating a file controlled using access cannot determine if there exists a special "back door" value of pid that would give others access to its files. UDFs can eliminate this problem. First, we transform this function into one that given an inode produces the set of principles allowed to use it (access(inode) $\rightarrow$ { set of principles }). (In a sense, we transform access into a function that returns the set of values for which the original access returned true.) Second, we ensure that access is deterministic. Now, at each modification of an inode we can use online testing to ensure that the set of principles associated with it grows or shrinks exactly as it should.

Program verification enables "nestable" extensibility. Because XN verifies the correctness of reference counts, pointers, and meta data interpreters, it allows untrusted implementors to extend an existing file system without compromising its integrity. Thus, it is possible to add an entirely new directory type to a file system and have it point to old types, perform access control on them, etc. without the existing implementation open to malice. We do not know of any other way to achieve this same result.

6.3.3 Lessons

Provide reasonable defaults. UDFs are written in a pseudo-assembly language. A simple virtual machine can be both easy to implement (ours took roughly a day) and small (ours was a few hundred lines of code). The cost, of course, is the unpleasantness of writing assembly-level code. Fortunately, for limited domains, such as packet filters or meta data interpreters, this drawback can be eliminated by hiding such code behind higher-level procedural interfaces, which clients use instead.

Unfortunately, we repeatedly neglected to construct good default libraries after building the base extensible system. As a result, clients typically wrote their code in terms of raw (albeit portable) assembly language, leading to impenetrable code scattered throughout programs. This cycle was self-reinforcing: new programmers that wanted to use the system would look at existing clients, typically not understand what they did, and so cut-and-paste the original code with ad hoc modifications.

The observation that an extensible system must provide good default libraries is neither deep nor unique [11]. Nonetheless it was frequently violated: we did so with packet filters, then with wake up predicates, then with UDFs.

Low-level type systems are useful. From one perspective XN can be viewed as a dynamic type system placed below a file system to catch errors. It served this role well, catching several errors in the C-FFS file system that had escaped the notice of an experienced implementor. In this sense, XN has benefits similar to a low-level typing system such as the Til assembly language developed to catch low-level compiler bugs [85]. One possible use of exokernel technology is to place them below existent operating systems as efficient runtime type checkers.

Fast languages are unnecessary. Writing downloaded code using an efficient language does not hurt. But, at least in the context of an exokernel, it does not seem to help overly much either. UDFs, for example, are written in an interpreted assembly code and wakeup predicates have numerous excess manual address translations. The reason for this is that code used for protection is usually off the critical path, while non-protection code can be placed in the application itself at (perhaps) the cost of an extra system call.

6.4 Protected Methods

This section briefly discusses our most recent use of downloaded code, *protected methods*. Similar to [9], we provides them as an extensible means for applications to implement safe decentralized sharing of state in those cases where the kernel's low-level access control is insufficient. For example, a file system whose directories are mapped to exokernel-protected disk blocks may also require that names within a directory be unique, an invariant inexpressible solely in terms of hardware protection. Using protected methods, directory blocks could be associated with a unique_name method that untrusted library file systems would have to use to allocate names.

This use of downloaded code has little to do with speed. Rather it is intended to solve the problem that distrusting application cannot force one another to obey execution invariants. For example, consider a model where unique_name is provided in a library, with the admonishment to use it when modifying a directory. Nothing prevents a malicious application from jumping into the middle of the procedure to skip over any invariant checks or, more simply, just writing to the directory block directly.

Mutually mistrusting applications can safely share state by agreeing on code to use, downloading it in the kernel, and accessing the state through this code. Method invocation happens via a system call, forcing execution to begin at a well-defined program counter value. This prevents applications from jumping over guards. Method execution cannot be “hi jacked” by the application manipulating its state via debugging system calls, signals, or page mapping tricks. State exists only in the method's address space. Applications cannot modify it by forging pointers or aliasing virtual addresses. A benefit of this separation is that non-page protection can be readily implemented, as opposed to page granularity memory protection of unrestricted application code. The method cannot write outside its address space. This protects the application from buggy or malicious method code. (Though, the areas where one would trust a method's output, but not trust it to not corrupt the application's state are rare.)

Methods can be used to force the coupling of state modifications, such as forcing the invalidation of a “negative name cache” when a directory entry is allocated. They also help the modification of data structures that span trust boundaries. For example, they can be used to repair file system data structures after a file system crash. Finally, they can provide an easy way to get atomicity.

Protected methods are only one of many ways to provide extensible protection. An alternative is to force all applications to be written in a restricted language and compiled with a trusted compiler. With the advent of languages such as Java, such alternatives may become more palatable.

Another, more traditional way, is to use servers to encapsulate sensitive state. However, because server code is not controlled by the trusted kernel it cannot enforce invariants on it. Thus, server functionality must be trusted completely, and cannot be nested in the same way that methods can be. For example, the kernel can use testing to verify that methods only touches a specific range of bytes in its guarded state, that its modifications preserve pre- and post-conditions, that it is correct, etc.

6.5 Discussion

Data is not a Turing machine. Data is inflexible, but transparent, and its operations (read and write) trivially bounded in cost and guaranteed to terminate. Code is not necessarily any of these three things. The benign characteristics of data can be a relief compared to the uncertainty induced by injecting potentially non-terminating black boxes into a complex operating system. Two specific examples follow.

Information can be communicated by memory (e.g, a flag set when the operating system is allowed to write a disk block) or by code (e.g., a routine called with the disk block asking if it can be written). XN explicitly uses pointers rather than code to track block write orders, despite the lack of flexibility. Code made dependency cycles more difficult to check, and thus, the cost of sharing higher.

It is relatively simple and well understood how to decouple application actions from application execution using the stylized method of buffering. An application that wishes to send a large message on the network can give the buffer for the message to the OS. The OS (or DMA engine) can in turn send it across the network at whatever rate the network supports, irrespective of whether the application that provided the data is currently running. As a result, this pattern of using buffering rather than downloaded code to decouple application actions from scheduling can be seen in all operating systems the author is aware of.

The alternative, having the application explicitly cooperate with the OS via tamed downloaded code, can be accomplished [32, 93] but requires far more machinery than the simple buffer management routines above.

Data has visible transitions. Data, because it is passive, can only be changed by an active entity. Given the right framework (e.g., if applications can only write to data via system calls), this characteristic makes transitions clear and easily coupled to actions or checks. For example, to allow applications to control the order of disk block writes, XN allows them to create dependency chains. Since the pointers used to form chains can only be added using the OS, it is simple to check for cycles. If code was used instead (e.g., a boolean procedure can_write?) that was associated with each block), if the code is in any way opaque, such checking becomes more difficult.

Data is passive, control active. From the application's perspective, the passivity of data can be a significant problem: it makes state transitions invisible, requiring polling to track them. Our exokernel provides *wakeup predicates* [3] as a way to conceptually (if not actually) transmute passive data into active events. Wakeup predicates are application code snippets downloaded into the kernel, bound to useful memory locations (block I/O flags, timer counters, etc.) and evaluated on various interrupts. When they evaluate to true, the process is awakened.

[3] The exokernel's system for this was conceived, designed and implemented by Thomas Pinckney

Code imposes indirection. Placing code between the operating system and its data forces the OS to go through a layer of potentially non-terminating indirection. For example, rather than meta data traversal routines that simply walk down a vector of block pointers, XN requires the use of untrusted iterators and must make provisions to ensure that they do not run too long. Additionally, it cannot simply modify client meta data anymore — since it does not understand their semantics — and, as a result, cannot necessarily do an operation as simple as changing one disk block pointer to another in order to compact the disk.

Further, the indirecting code forces the OS to plan for failure. It must have a contingency plan for when the application code does not update data appropriately or even terminate. Having to rely on a non-trustworthy opponent to do crucial operations can be a practical irritant.

Code hides information. Information can be exchanged from operating system to application using either mapped OS data structures or system calls. The latter interface shields applications from implementation details. However, if the OS does not anticipate the need for a piece of information and encapsulate it within a system call, the client cannot recover it. However, by ripping away this procedural layer layer and exporting data structures (read only) to applications, they can obtain all information, anticipated by the kernel implementor or not. (The potential cost is being tied to a specific implementation.)

Our library operating system's reliance on “wakeup predicates” has driven home the advantages of exposing kernel data structures. Frequently, we have required unusual information about the system. In all cases, this information was already provided by the kernel data structures.

Understanding. A practical problem in using downloaded code is that historically there has been a schism between the compiler and operating system communities. As a result, OS implementors frequently do not understand compilers. They have no equivalent difficulty with data structures.

Alternatives to downloading code for semantics. DPF, ASHs and XN can be viewed as systems to pull application semantics into resource management decisions. However, this is not the only way to get the same effect. The easiest is to upload operating system code into the application. It can then decide how to implement whatever decision it desires. Additionally, it can do so in a Turing complete way, in an unrestricted environment, and with much concern in the operating system about termination and opaqueness issues.

In non-protection situations, the semantics of resources need never be imported into the operating system. The application can instead determine what actions to do, using whatever domain-specific knowledge is important, and then just tell the operating system what to do. This requires constructing interfaces that do declarative checking of an operation rather than imperatively deciding how to do it. For example, consider the problem of writing cached disk blocks to stable storage in a way that guarantees consistency across reboots. Rather than an exokernel deciding on a particular write ordering itself and thus having to struggle with the tradeoffs in scheduling heuristics and caching decisions required to do so well, it can instead allow the application construct schedules, retaining for itself the much simplified task of merely checking that any application schedule gives appropriate consistency guarantees. Application of this methodology enables an exokernel to leave library operating systems to decide on tradeoffs themselves rather than forcing a particular set, a crucial shift of labor.

One of the games we have played frequently in an exokernel is determining how to construct interfaces where an application “just knowing” what is appropriate can be expressed as a kernel action.

6.6 Related Work

There have been a number of papers that evaluate different downloading code mechanisms. Small and Seltzer compare several extension techniques [80], Bershad et al. [9] and Pardyak and Bershad describe [71] different aspects of the SPIN operating system's extensibility framework. This chapter complements this prior work. Stallman [81] and Borenstein and Gosling [11] discuss language issues in the context of the EMACS text editor. The discussion is largely complementary, since extensions in this context are trusted, but there are overlapping lessons (the most painful to rediscover that reasonable defaults must be provided).

Both SPIN [9, 32, 71] and Vino [79, 80] are two other extensible operating systems that use downloaded code.

Code motion has been a recurrent theme in operating systems since inception [41, 22]. Micro-kernels are an attempt to move operating system code out of the harsh environment of the kernel into the more genteel context of processes [41, 53, 2, 77, 84]. Virtual machines [20] similarly move operating system code to application-level. Most modern operating systems provide ways to dynamically download load device drivers.

Several hints for when to download code can be found in Lampson [52]. A useful insight is that downloading is simply an example of higher-order function programing (or in systems languages, the use of function pointers rather than flags as parameters). The main difference is that code is being shipped across trust boundaries rather than, for instance, library interfaces. Thus, it appears possible to take some of the ideas from these mature areas “whole cloth.” Sussman and Abelson [1] is a classic text.

The parallel community has long considered the idea of “function shipping” for speed — e.g., to bring computation closer to data, to be able to migrate computations for load-balancing, etc. Some of the insights from this use can be applied to operating systems. Similarly, the distributed systems community has shipped code as well. Java applets are a topical example [40]. Tennenhouse and Weatherall have proposed to use mobile code to build Active Networks [86]; in an active network, protocols are replaced by programs, which are safely executed in the operating system on message arrival. Curiously, in contrast to our experience, most uses of mobile code in an Active Network seem to be to improve efficiency, rather increase power.

Chapter 7

Conclusion

This chapter discusses possible ways that an exokernel approach could fail, lessons learned in building our exokernel systems, and conclusions.

7.1 Possible Failures of the Architecture

Doubt 'til thou canst doubt no more...doubt is thought and thought is life. Systems which end doubt are devices for drugging thought. —- Albert Guerard

In our mind, the remaining serious questions about the exokernel architecture are sociological ones rather than technical. We list five possible failures of the architecture once it moves from our coddling laboratory into the “real world:”

1. Application writers do not deal well with the freedom they have and become tied too closely to a particular exokernel implementation, preventing upgrades and slowing, rather than improving, system evolution. While adherence to standard interfaces and good programming practices *should* prevent this type of failure (given that these techniques have a venerable track record of doing so in other domains), it remains to be demonstrated if they suffice for exokernels.

2. Commoditization of operating system software makes operating system research irrelevant. Commoditization has already severely restricted the viability of new OS interfaces. The dominance of a few OSes, and the increasing cost of implementing them, may restrict the viability of new implementations of these interfaces as well. If so, then much of the innovation potential of an exokernel will be lost.

3. The technical ability to innovate does not lead to any more innovation than on traditional systems. The exokernel architecture is based on a partially-sociological assumption: that making OS innovation easier and less costly will lead to a vast improvement in innovation. This may well be a false assumption. For instance, innovation may already be “easy enough” for those who care to do it.

4. An exokernel, by migrating most OS code to libraries, removes the “single point of upgrade” characteristic of current systems. This can be an advantage, since applications do not have to wait for the central OS to upgrade but can instead do so themselves. However, it can also impede progress by making improvements harder to disseminate.

5. Users will not switch operating systems. An OS forms the primal mud on which systems are built. Changes to it have far reaching impact. Computer system users have thus demonstrated an understandable reluctance to alter it. It may be that the advantages of an exokernel system do not proof sufficient to lead to such a switch.

   Fortunately, an exokernel does not require “whole cloth” adoption for success. It appears that many exokernel interfaces, especially those related to I/O, can be grafted on to existing systems, with little loss in performance. Normal applications would the existing OS interfaces as a default, while more aggressive applications would have the power to control important decisions.

While we have confidence that the exokernel's technical advantages will allow it to transcend these potential pitfalls, it must still demonstrate that it does.

7.2 Experience

Over the past three years, we have built three exokernel systems. We distill our experience by discussing the clear advantages, the costs, and lessons learned from building exokernel systems.

7.2.1 Clear advantages

Exposing kernel data structures. Allowing libOSes to map kernel and hardware data structures into their address spaces is a powerful extensibility mechanism. (Of course, these structures must not contain sensitive information to which the application lacks privileges.) The benefits of mapping data structures are two-fold. First, exposed data structures can be accessed without system call overhead. More importantly, however, mapping the data structures directly allows libOSes to make use of information the exokernel did not anticipate exporting.

Because exposed data structures do not constitute a well-defined API, software that directly relies on them (e.g., the hardware abstraction layer in a libOS) may need to be recompiled or modified if the kernel changes. This can be seen as a disadvantage. On the other hand, code affected by changes in exposed data structures will typically reside in dynamically-linked libOSes, so that applications need not concern themselves with these changes. Moreover, most improvements that would require kernel modification on a traditional operating systems need only effect libOSes on exokernels. This is one of the main advantages of the exokernel, as libOSes can be modified and debugged considerably more easily than kernels. Finally, we expect most changes to the exokernel proper to be along the lines of new device drivers or hardware-oriented functionality, which expose new structures rather than modify existing ones.

In the end, some aggressive applications may not work across all versions of the exokernel, even if they are dynamically linked. This problem is nothing new, however. A number of UNIX programs such as top, gated, lsof, and netstat already make use of private kernel data structures through the kernel memory device `/dev/kmem`. Administrators have simply learned to reinstall these programs whenever major kernel data structures change.

The use of "wakeup predicates" has forcefully driven home the advantages of exposing kernel data structures. Frequently, we have required unusual information about the system. In all cases, this information was already provided by the kernel data structures.

The CPU interface. The combination of time slices, initiation/termination upcalls, and directed yields has proven its value repeatedly. (Subsequent to our work, others have found these primitives useful [35].) We have used the primitives for inter-process communication optimization (e.g., two applications communicating through a shared message queue can yield to each other), global gang-scheduling, and robust critical sections (see below).

Libraries are simpler than kernels. The "edit, compile, debug" cycle of applications is considerably faster than the "edit, compile, reboot, debug" cycle of kernels. A practical benefit of placing OS functionality in libraries is that the "reboot" is replaced by "relink." Accumulated over many iterations, this replacement reduces development time substantially. Additionally, the fact that the library is isolated from the rest of the system allows easy debugging of basic abstractions. Untrusted user-level servers in microkernel-based systems also have this benefit.

7.2.2 Costs

Exokernels are not a panacea. This section lists some of the costs we have encountered.

Exokernel interface design is not simple. The goal of an exokernel system is for privileged software to export interfaces that let unprivileged applications manage their own resources. At the same time, these interfaces must offer rich enough protection that libOSes can assure themselves of invariants on high-level abstractions. It generally takes several iterations to obtain a satisfactory interface, as the designer struggles to increase power and remove unnecessary functionality while still providing the necessary level of protection. Most of our major exokernel interfaces have gone through multiple designs over several years.

Information loss. Valuable information can be lost by implementing OS abstractions at application level. For instance, if virtual memory and the file system are completely at application level, the exokernel may be unable to distinguish pages used to cache disk blocks and pages used for virtual memory. Glaze, the Fugu exokernel, has the additional complication that it cannot distinguish such uses from the physical pages used for buffering messages [60]. Frequently-used information can often be derived with little effort. For example, if page tables are managed by the application, the exokernel can approximate LRU page ordering by tracking the insertion of translations into the TLB. However, at the very least, this inference requires thought.

Self-paging libOSes. Self-paging is difficult (only a few commercial operating systems page their kernel). Self-paging libOSes are even more difficult because paging can be caused by external entities (e.g., the kernel touching a paged-out buffer that a libOS provided). Careful planning is necessary to ensure that libOSes can quickly select and return a page to the exokernel, and that there is a facility to swap in processes without knowledge of their internals (otherwise virtual memory customization will be infeasible).

7.2.3 Lessons

Provide space for application data in kernel structures. LibOSes are often easier to develop if they can store shared state in kernel data structures. In particular, this ability can simplify the task of locating shared state and often avoids awkward (and complex) replication of indexing structures at the application level. For example, Xok lets libOSes use the software-only bits of page tables, greatly simplifying the implementation of copy on write.

Fast applications do not require good microbenchmark performance. The main benefit of an exokernel is not that it makes primitive operations efficient, but that it gives applications control over expensive operations such as I/O. It is this control that gives order of magnitude performance improvements to applications, not fast system calls. We heavily tuned Aegis to achieve excellent microbenchmark performance. Xok, on the other hand, is completely untuned. Nevertheless, applications perform well.

Inexpensive critical sections are useful for LibOSes. In traditional OSes, inexpensive critical sections can be implemented by disabling interrupts [10]. ExOS implements such critical sections by disabling software interrupts (e.g., time slice termination upcalls). Using critical sections instead of locks removes the need to communicate to manage a lock, to trust software to acquire and release locks correctly, and to use complex algorithms to reclaim a lock when a process dies while still holding it. This approach has proven to be similarly useful on the Fugu multiprocessor; it is the basis of Fugu's fast message passing.

User-level page tables are complex. If page tables are migrated to user level (as on Aegis), a concerted effort must be made to ensure that the user's TLB refill handler can run in unusual situations. The reason is not performance, but that the naming context provided by virtual memory mappings is a requirement for most useful operations. For example, in the case of downloaded code run in an interrupt handler, if the kernel is not willing to allow application code to service TLB misses then there are many situations where the code will be unable to make progress. User-level page tables made the implementation of libOSes tricky on Aegis; since the x86 has hardware page tables, this issue disappeared on Xok/ExOS.

7.3 Conclusion

Our inventions mirror our secret wishes. Lawrence Durrell (1912-1990) “Mountolive” (1959)

This thesis proposes and evaluates the exokernel operating system architecture. An exokernel gives untrusted application code as much safe control over resources as possible. It does so by separating management from protection. All functionality necessary for protection resides in the exokernel, control over all other aspects is given to applications. Ideally, applications can safely and efficiently perform any operation that a privileged operating system can. Thus, unlike traditional systems, on an exokernel system, OS software becomes: (1) unprivileged, (2) able to co-exist with other implementations, (3) modifiable and deployable by orders of magnitude more programmers. We hope that this organization significantly improves operating system innovation.

This thesis has discussed both the exokernel architecture, and how to apply its principles in practice, drawing upon examples from two exokernel systems. These systems give significant performance advantages to aggressively-specialized applications while maintaining competitive performance on unmodified UNIX applications, even under heavily multi-tasked workloads. Exokernels also simplify the job of operating system development by allowing one library operating system to be developed and debugged from another one running on the same machine. The advantages of rapid operating system development extend beyond specialized niche applications. Thus, while some questions about the full implications of the exokernel architecture remain to be answered, it is a viable approach that offers many advantages over conventional systems.

Appendix A

XN's Interface

This appendix describes the public and privileged XN system call interface.

A number of routines expect a device number (an integer of type `dev_t`), which names an active XN-controlled disk. This name is implicit in the disk address type, `da_t`, a 64-bit integer that encodes the device name, disk block, and byte offset within the disk block. A device corresponds to some range of disk blocks, a freemap that tracks these blocks, and a "root catalogue," which is a persistent table that libFSes install types and file system roots into.

In general, any XN system call fails if: (1) a libFS-supplied capability is insufficient, (2) a libFS-supplied pointer is bogus (i.e., not readable or, for modifications, not writeable), or (3) a libFS-supplied name is bogus (e.g., an invalid disk block name, an invalid root catalogue entry character string, etc.). To save space, we do not mention these errors further.

We elide the details of mapping XN data structures and buffer cache entries, since they are specific to the hosting OS's virtual memory interface rather than to XN itself.

A.1 Privileged system calls

Only the kernel or privileged applications can use the following system calls.

A.1.1 XN initialization and shutdown

The following three routines are used to initialize and cleanly shutdown an XN-controlled disk.

```
xn_err_t sys_xn_init(dev_t dev);
```

Initialize XN disk `dev`. Fails if the disk was not cleanly shutdown, in which case the XN disk reconstruction program must be run in order to find all XN invariant violations.

```
xn_err_t sys_xn_shutdown(dev_t dev);
```

Cleanly shutdown disk `dev`. Fails if any blocks in the buffer cache contain violations. Before it can proceed the invoker must iteratively flush the buffer cache to remove these violations.

dev_t sys_xn_format(void);

Prepare a new disk for XN; returns the device number of the disk. Currently, it always succeeds.

A.1.2 Reconstruction

The following two functions are used by privileged reconstruction programs:

db_t sys_db_alloc(dev_t dev, db_t db, size_t n);

Forces the extent [db, db+n) on disk dev to be taken off of the freelist.

xn_err_t sys_xn_mod_refcnt(da_t da, int delta);

Alters da's reference count by delta.

Currently, the error log routines are in flux. We elide them here.

A.2 Public system calls

The following system calls are intended for use by any libFS.

A.2.1 Creating types

The following three system calls are used to create the types for a new new libFS.

xn_err_t sys_install_mount(dev_t dev, char *name, db_t *db, size_t nelem, xn_elem_t t, cap_t c);

Allocate the extent [db, db+nelem*sizeof t) on disk dev and associate it with name in the root catalogue. If db's value is 0, then the kernel decides what extent to allocate and writes the allocated block into db. LibFSes use this system call to install both install file system roots and types. On reboot, the XN reconstructor loads the catalogue and traverses file systems from these roots, garbage collecting and performing consistency checks. The entry can be modified for applications that possess cap. The call fails if any block in the extent has already been allocated, if type is invalid, name or db are not readable.

xn_err_t sys_type_import(dev_t dev, char *type_name);

Converts the root catalogue entry name in dev's root catalogue from a disk block extent to an actual type. It fails if name does not exist, the blocks are not “raw” disk blocks (i.e., of type XN_DB), or the contents of the extent form an invalid type.

xn_err_t sys_reserve_type(dev_t dev, char *name);

Reserve a slot for an as-yet-unspecified type in dev's root catalogue. This call is used to construct mutually recursive types.

To create a new type, the libFS performs the following four steps:

1. Allocates space for it on disk and installs it in the root catalogue using sys_install_mount. The value of type in this installation is XN_DB, indicating the blocks are simple disk blocks. If the type is involved part of a mutually recursive specification (which can happen when the type is a composite type), the libFS can use sys_reserve_type to reserve the type names for these other types, as well as getting their type id (an integer assigned by the kernel), which is needed by the type's owns UDF to compute the typed block set that it outputs.

2. Initializes these blocks using sys_xn_writeb (discussed below) to the contents in this type's "type structure," which holds its owns UDF, its refcnt UDF, and other information.

3. Writes these blocks back to disk (the following step fails if they are dirty).

4. Finally, it uses sys_type_import to convert the blocks from generic blocks to an XN recognized type. sys_type_import performs consistency checks on the type data structure (e.g., that the contained UDFs are deterministic, that the partitions are sensible) and, if the checks succeed, changes the type cataglogue entry to be of type XN_TYPE rather than XN_DB.

At this point, the libFS can create and point to blocks of the new type.

A.2.2 Creating and deleting file system trees

To create a new file system, the libFS installs the root of the tree using sys_install_mount. To load an already existing one from disk into the buffer cache it can use the following two functions:

```
xn_err_t sys_xn_mount(dev_t dev, struct root_entry *r, char *name, cap_t c);
```

Bootstrap the file system tree by inserting the type name's block extent into the buffer cache: getting the types for the rest of the file system tree can be done by recursively applying the associated owns UDF to the root and its children. The call writes the root catalogue entry into r, and then annotates the associated entries in the buffer cache with the given type. Fails if name name does not exist, or the extent pointed to by the root catalogue entry is not in the buffer cache registry.

```
xn_err_t sys_type_mount(dev_t dev, char *type_name);
```

Similar to sys_xn_mount, except that it annotates a a type_name's cached blocks as holding an XN type. It fails if the given name does not exist in the root catalogue, is not a type, or if the blocks are not in the buffer cache.

To use a file system tree, the file system root and any types it needs must first be loaded into the buffer cache using sys_xn_read_and_insert (discussed below). Then, sys_xn_mount annotates these disk blocks with the type given by their root catalogue entry (for the root, its synthetic libFS type, for types themselves XN_TYPE).

Types and roots can be removed from the root catalogue using:

```
xn_err_t sys_uninstall_mount(dev_t dev, char *name, cap_t c);
```

Delete root catalogue entry name on dev. Fails if name is a file system root that contains child pointers, or is a type with cached blocks. If a type is deleted that is used by some non-cached meta data instance, then that meta data's owns function will fail (as will the system call using owns). A better solution would be to count how many blocks of a given type exist and only allow the type to be deleted when there are no more blocks of its type.

A.2.3 Buffer cache operations

The structure xn_op is used for many of the remaining system calls, usually to specify extents, owns partition id's, and (for modification) the byte range(s) to modify. Figure A-1 gives its ANSI C representation.

```
/∗ specify xn operations that involve UDFs. ∗/
struct xn_op {
        /∗ Specify what extent will be allocated/freed/read. ∗/
        struct xn_update {
                size_t own_id;          /∗ id of the udf to run. ∗/
                cap_t   cap;            /∗ capability to use for access. ∗/

                db_t db;          /∗ base   all objects are sector aligned. ∗/
                size_t nelem;     /∗ number of elements of type. ∗/
                xn_elem_t type; /∗ type ∗/
        } u;

        /∗
         ∗ Specify update to a piece of meta data.  Semantics:
         ∗       memcpy((char ∗)meta + offset, addr, nbytes);
         ∗ Ignored for reads.
         ∗/
        struct xn_m_vec {
                size_t   offset;  /∗ what offset in the type ∗/
                void    ∗addr;  /∗ ptr to value to copy there ∗/
                size_t   nbytes;
        } ∗mv;
        size_t n;              /∗ number of elements in mv ∗/
};
```

Figure A-1: Metadata operation structure.

```
struct xn_iov {
        xn_cnt_t ∗cnt;                  /∗ Decremented on every successul io∗/
        struct xn_ioe {
                db_t db;
                size_t nblocks;
                void ∗addr;             /∗ mem to write from/to. ∗/
        } iov[1];
        size_t n_io;                  /∗ number of entries. ∗/
};
```

Figure A-2: I/O vector structure.

The following two routines are used to bring blocks into the buffer cache and prepare them for use and delete them:

xn_err_t sys_xn_read_and_insert(dev_t dev, db_t db, size_t nblocks, xn_cnt_t *cnt);

> Read [db, db+nblocks) from disk dev into the buffer cache. cnt is incremented each time a block is brought in. Fails if the extent does not fit in the buffer cache.

xn_err_t sys_xn_insert_attr(da_t parent, struct xn_op *op);

> Install the type of the buffer cache entry using the parent block named by parent. Allocation, deletion and reading and writing all perform this call implicitly since they require registry entries. Fails if the extent specified in op is not guarded by the -partition specified in op.

xn_err_t sys_xn_delete_attr(dev_t dev, db_t db, size_t nblocks, cap_t cap);

> Remove the buffer cache entries for [db, db+nblocks) associated with disk dev. Fails if removing the entry would lead to an invariant violation or if the entry is in use by some other application.

Importantly, libFSes can fetch any block extent into the buffer cache. They only need to locate the blocks parent (and thus its type and access control mechanism) before actually reading or writing the cached blocks.

The following two functions write buffer cache entries back to disk:

xn_err_t sys_xn_writeback(dev_t dev, db_t db, size_t nblocks, xn_cnt_t *cnt);

> Writes [db, db + nblocks) back to disk dev. Each write decrements cnt.

xn_err_t sys_xn_writebackv(dev_t dev, struct xn_iov *iov);

> Writes a list of these extents back to disk dev. On some hardware disks, this "gather" mechanism enables more sophisticated scheduling. Figure A-2 gives the ANSI C representation for xn_iov. Note that neither of these two system calls require access checks: They only fail if the extent is invalid or contains blocks tainted with some violation.

The following two routines read and write the bytes in buffer cache entries. No byte range read by a UDF can be modified using these routines.

xn_err_t sys_xn_readb(void * dst, da_t da, size_t nbytes, cap_t cap);

> Reads [da, da + nbytes) into [dst, dst + nbytes). Fails if any block of the extent is not in core.

xn_err_t sys_xn_writeb(da_t da, void * src, size_t nbytes, cap_t cap);

> Writes [src, src + nbytes) into the cached blocks for [da, da + nbytes). All of the bytes must be in memory. Fails for reasons identical to above, with the additional restriction that [dst, dst + nbytes) must be writable.

A.2.4 Metadata operations

The following routines are used to allocate blocks, form edges to existing blocks, delete edges to existing blocks, and change a block's type:

xn_err_t sys_xn_alloc(da_t da, struct xn_op *op, unsigned alloc);

> Allocates the extent [db, db + (sizeof type * nelem) given by op and writes the modifications contained in op into the parent metadata, da. alloc indicates whether the block should be zero filled. The call fails if: (1) the extent cannot be allocated; (2) da is not in core; (3) op contains bogus data (its partition id is invalid or the modification vector has bogus byte ranges); or (4) the UDF returns the wrong data.

xn_err_t sys_xn_free(da_t da, struct xn_op *op);

Frees the extent [db, db + (sizeof type * nelem) given by op. If the extent's has a reference count greater than one, then the reference count is decremented. Otherwise it is placed on the freelist. Fails for similar reasons to sys_xn_alloc.

xn_err_t sys_xn_add_edge(da_t src, struct xn_op *src_op, da_t dst);

Forms an edge from src to dst and increments dst's reference count. Fails if: (1) neither node is in core (src must be modified to add a pointer, dst modified to increment its refernce count); (2) src's owns UDF fails, or dst's refcnt UDF fails; or (3) src_op specifies bogus modifications.

xn_err_t sys_xn_set_type(da_t da, int ty, void *src, size_t nbytes, cap_t cap);

Change the type of a type union instance: XN metadata types can be "unions," which means they can be dynamically converted among a series of listed types. Currently, the type field must be "nil" (i.e., the type has just been initialized). Extending the system call to convert between existant types would not be difficult (it only requires retesting that the new type's owns function emits the same blocks as the old one).

A.2.5 Reading XN data structures

The following six functions allow applications to read various XN bookkeeping data structures. All fail if the given memory is not writable. If the used data structures are mapped into the libFS's addresses space, these calls are simple library calls.

xn_err_t sys_xn_read_attr(dev_t dev, void * dst, db_t db, cap_t cap);

Reads the registry attribute for db on disk dev into dst. Fails if db is not in core.

xn_err_t sys_xn_read_catalogue(dev_t dev, struct root_catalogue *c);

Read dev's root catalogue into c.

xn_err_t sys_dev_list(dev_t *devl, int ndevs);

Stores the enumerated list of active devices into devl, which is ndevs big. Fails if the destination is not large enough.

xn_err_t sys_xn_info(dev_t dev, db_t* r_db, db_t* f_db, size_t* f_nbytes);

Reads the block address that holds the root catalogue into r_db, the block address of the freemap into f_db and the size of the freemap into nbytes.

db_t sys_xn_findfree(dev_t dev, db_t hint, size_t nblocks);

Returns the first free extent (starting at hint, hint+nblocks)) found on dev. If hint is 0, then XN makes its own decision about where to start searching.

xn_err_t sys_xn_list_writeable(dev_t dev, db_t *dbv, size_t *n);

Reads a the list of dirty but writable entries for device dev in the buffer cache into dbv (whose size is given by n). Typically this is used by two programs. A "syncer" deamon that flushes back dirty blocks, and by any shutdown application that needs to flush all blocks back to disk. The latter locks the buffer cache and iteratively flushes blocks until the empty list is returned.

db_t sys_root(dev_t dev);

Return the block address of dev's XN-created "superblock" (i.e., the block that holds the pointers to XN's per-device bookkeeping data structures).

A.2.6 File system-independent navigation calls

The following routines allow program to navigate any libFS meta data. They are used, for example, by our disk reconstruction program.

`xn_err_t sys_xn_get_refcnt(da_t da, cap_t cap);`

Returns `da`'s reference count.

`xn_err_t sys_xn_get_nowns(da_t da, cap_t cap);`

Returns the number of `owns` partitions in `da`'s type.

`xn_err_t sys_xn_udf_enum(da_t da, size_t owns_b, size_t owns_e, struct xn_update *ups, size_t n, cap_t cap);`

Computes a list of of all blocks controlled by `da`'s partitions `owns_b` through `owns_e`. These are written into the vector `ups` (whose size is given by `n`). To ensure that the list does not get "too big" and the system call does not run "too long" (both of which are OS dependent) the enumeration may be broken up by XN into multiple passes.

Appendix B

Aegis' Interface

This appendix gives a large subset of Aegis' system calls. In general, any system call fails if: (1) a libOS lacks permissions for the operation, (2) a libOS-supplied pointer is bogus (i.e., not readable or, for modifications, not writeable), or (3) a libOS-supplied name is bogus (e.g., an invalid page name, packet filter id, etc.). To save space, we do not mention these errors further.

B.1 CPU interface

Figure B.1 presents the ANSI C representation of an Aegis time slice. The routines to allocate, yield, and free time slices are given below. Time slices can be donated temporarily (via ae_yield), permenantly (via ae_donate) or by implicitly by asynchronous IPC. Ownership of the time-slice can be changed by any process that has the owning environment's capability.

Time-slices are initiated via an upcall to application space and revoked in the same manner; this interaction allows applications to control important context-switching operations (e.g., this functionality is sufficient to implement scheduler activations). The price of this functionality is that an application must be prevented from ignoring revocation interrupts. The current method is to record the number of timer interrupts a process has

```
/* Time slice; controls an ordered scheduling quanta. */
struct slice {
        char pad[12];
        struct env        *e;    /* associated environment (null if no one) */
        unsigned short  next,prev;     /* forward and back pointers */
        int ticks;         /* ticks consumed in interrupts */
        int epc;
};
```

Figure B-1: Aegis time-slice representation ignored in "ticks" and when this value exceeds a predefined threshold killing the process. In a more mature implementation we would simply context switch the application by hand.

int ae_s_alloc(int pid, int n);

> Allocate time slice n and give environment pid access to it. Fails if n is not free, or the current process lacks write access to pid.

int ae_s_free(int slice);

> Free time slice slice. Fails if the current process lacks write access to the owning environment.

int ae_yield(int pid);

> Yield the remainder of the current time quantum to pid.

int ae_donate(int pid);

> Permenantly donate current time slice to process pid.

B.2 Environments

Figure B.2 presents the ANSI C representation of an Aegis environment, which is used to store the hardware information needed to execute a thread of control in a virtual address space, along with resource accounting information.

int ae_e_alloc(int n, struct env *e);

> Allocate environment n. Application fills in the guarenteed mappings, exception handlers in the environment structure (given in e). Aegis checks that any given physical addresses are allowed an sensible.

int ae_e_ref(int pid);

> Add a refrence from the current process to environment pid. Fails if the current process lacks permissions.

int ae_e_unref(int pid);

> Unreference environment pid. If there are no other outstanding references, the env is freed. Fails if the current process does not have read permission to the environment.

int ae_e_add(int pid, int n);

> Give pid access to environment n. Only gives write access at the moment. Fails if the current process does not have access to pid.

int ae_e_free(int pid);

> Free all resources consumed by environment pid.

int ae_e_write(int n, size_t offset, void *p, size_t nbytes);

> Modify the byte range [offset, offset+nbytes) in environment n's application-specific data region.

```
/* Environment structure */
struct env {
        /* pointers to exception "save areas" where Aegis stores
         * active registers. */
        addr_t          tlbx_save_area,
                        genx_save_area,
                        intx_save_area;

        /* exception handlers. */
        handler_t       xh[NEX], /* sync exception handlers */
                        epi,    /* epilogue code */
                        pro,    /* prologue code */
                        init;   /* initial code that is jumped to */
        /*
         * The following four fields are clustered to be on
         * the same cache line.
         */
        signed char     cid;    /* address space identifier */
        unsigned char   envn;   /* environment number */
        short           tag;    /* 11 bit tag */
        handler_t       gate;   /* ipc entry point */
        unsigned        status; /* status register */
        struct tlb      xl[MAXXL];      /* guarenteed translations */
        struct ae_intr_queue *iq;       /* interrupt queue */
};
```

Figure B-2: Environment structure. An environment is the most complicated entity in Aegis. It is basically a process: it defines the program counters to vector events to, and serves as a resource accounting point.

```
struct ae_interrupt {
        handler_t  h;           /* handler to jump too. */

        /* Up to three int specific arguments.  Used to parameterize interrupts.*/
        void *arg1;
        void *arg2;
        void *arg3;

        /* Saved values to give application scratch registers
           epc is set to zero if no return point.  */
        unsigned epc;   /* holds value that we were interrupted at. */
        unsigned a0;    /* holds a0 register */
        unsigned a1;    /* holds a1 register */
        unsigned a2;    /* holds a1 register */
};
```

Figure B-3: Structure used to hold an interrupt's state.

B.3 Physical memory

The following three routines allocate, deallocate, and share physical pages.

`int ae_p_alloc(int n);`

Allocate page n. Fails if page is already allocated.

`int ae_p_unref(int n);`

Remove reference to page n. If no other process has a reference to the page it is deallocated. Fails if the current process lacks permissions.

`int ae_p_add(int prot, int pid, int n);`

Give process pid access to page n with protections prot. Fails if the current process lacks appropriate permissions.

B.4 Interrupts

To make interrupt handling efficient, Aegis places interrupt notifications in a user-space interrupt queue that the libOS can read and modify directly. The structures used for this are given in Figure B-3 and Figure B-4.

B.5 Miscellaneous

`int ae_read_exposed_info (struct exposed_info *i);`

Read out offsets and lengths of exposed kernel data structures in preperation for mapping their associated pages. Figure B-5 presents the data structures that can be mapped.

`int ae_getrate(void);`

Get the system's clock rate.

```
/*
 * circular interrupt queue: is provided at user level so that applications
 * can control interrupt handling efficiently.
 *
 * Useful facts:
 *      1. pending == 0   > q is empty
 *      2. tail points to the first entry to dequeue.
 *      3. full: pending == sz
 *      4. head   points to first empty entry.
 *
 * To consume an interrupt:
 *      1. increment tail % AE_INTQ_SZ
 *      2. decrement pending.
 *      3. renable the interrupt type.
 */
struct ae_intr_queue {
        /* counts the number of interrupts pending while the
         * process is running with interrupts disabled.  */
        unsigned short pending;

        /* Number of overflow ints for each type.  (simple way to allow
         * resource specific recovery.) */
        unsigned short overflow[AE_NINTS];
        unsigned short overflow_p;      /* set to 1 if there is overflow. */

        unsigned        mask;           /* 0  > disabled, 1  > enabled. */

        /* Handlers, two for each interrupt.  We make application
         * explicitly check whether it was running or not.  Could
         * have two handlers, where  handler 0 is set when process
         * was not executing, handler 1 is set when it was.  */
        handler_t       h[AE_NINTS];

         /* Circular queue; when it runs out, we write an overflow interrupt.  */
         unsigned char head;
         unsigned char tail;
#        define AE_INTQ_SZ       (1 << 4)
         struct ae_interrupt e[AE_INTQ_SZ];
};
```

Figure B-4: Circular interrupt queue. The kernel stores interrupt events in this queue, and the application consumes them. The queue lives in shared memory, so modifications do not require a system call.

int ae_gettick(void);

Get the current clock tick.

int ae_cacheflush (void *ptr, int sz, int flags);

Flush the address range [ptr, ptr+sz) out of the cache.

int ae_memcpy(int dst_pfn, int src_pfn, int sz, int cache);

Copy pages src_pfn, src_pfn+sz) to the contigious page range starting at dst_pfn, bypassing the TLB. cache indicates whether the copy should also bypass the cache.

int ae_memset(int dst_pfn, int offset, char b, int sz);

Set memory [dst_pfn+offset, dst_pfn+offset+sz) to b. Bypasses the TLB.

int ae_fpu(int flag);

Enable/disable floating-point unit.

int ae_pid(void);

Get the current pid.

B.6 Networking

Aegis provides calls to insert and delete packet filters. It also provides support for Ethernet and AN2 OTTO chips. For simplicity, we only provide the former's interface.

The following two functions install and delete packet filters. Filters can be bound to an ASH (libOS messaging code downloaded into the kernel), which is run when the filter matches. Otherwise the packet will be handled by the libOS.

int ae_dpf_install(void *filter, int sz, int ash_id);

Installs filter (of sz bytes) and binds it to the ASH ash_id (if any). Fails if the size is too large, or the filter overlaps with another filter and the current process has insufficient permissions to override it.

int ae_dpf_delete(int id);

Delete filter id. Fails if the process lacks permission to do so.

The following four functions are used to send and receive messages.

int ae_eth_poll(int fid, struct ae_recv *recv, volatile int *addr);

Install a receive structure, recv, to handle packets arriving for filter fid; receive notification via polling. addr points to a counter that is incremented by the received message size. Figure B-6 shows the ANSI C representation of the receive structure. Fails if there are already too many enqueued receive structures.

int ae_eth_send(void *msg, int sz);

Send msg (of sz bytes) on an Ethernet.

int ae_eth_sendv(struct ae_recv *recv);

“Gather” send of the message specified by recv: Aegis copies the message into a contigous outgoing buffer (the hardware it runs on lacks support for DMA).

int ae_eth_info(addr_t addr);

Get Ethernet address.

```
struct exposed_info {
    unsigned int start_page;   /* which phys page the info starts on */
    int num;                   /* how many pages */

    /* timing related structures */

    char *clock_tick_start;
    int clock_tick_size;
    char *clock_rate_start;
    int clock_rate_size;

    /* page info */
    char *p_refcnt_start, *p_acl_start, *p_map_start;
    int p_refcnt_size, p_acl_size, p_map_size;

    /* env info */
    char *e_refcnt_start, *e_acl_start, *e_map_start, *env_start;
    int e_refcnt_size, e_acl_size, e_map_size, env_size;

    /* time slice info */
    char *s_refcnt_start, *s_acl_start, *slice_start, *s_map_start;
    int s_refcnt_size, s_acl_size, slice_size, s_map_size;

    /* context ids info */
    char *cmap_start, *c_map_start;
    int cmap_size, c_map_size;

    /* stlb */
    char *stlb_start;
    int stlb_size;
};
```

Figure B-5: Structure used to hold where each exposed kernel data structure begins and its size. By default, each environment has read access to the pages containing these structures.

```
/* Structure to hold recv messages. */
struct ae_recv {
        int n;  /* number of entries */
        struct rec {
                int sz;
                void *data;
        } r[MAXPKTS];
};
```

Figure B-6: Network packet receive structure

```
struct stlb {
        /* STLB tag: the contents of the TLB context register (c0_tlbctx) */
        unsigned    :2,
                    vpn:19, /* bad virtual page number */
                    tag:11; /* 11 bit tag associated (pseudo randomly) with each process */
        /* TLB entry */
        unsigned    :8,     /* reserved */
                    g:1,    /* Global: TLB ignores the PID match req */
                    v:1,    /* Valid: if not set, TLBL or TLBS miss occurs*/
                    d:1,    /* Dirty */
                    n:1,    /* Non cacheable. */
                    pfn:20; /* Page frame number */
};
```

Figure B-7: STLB structure

B.7 TLB manipulation

Aegis uses a software TLB [7] to increase the effective hardware TLB size. The STLB has 8192 entries and is a direct mapped hash table; it also has an 8-entry fully-associative overflow buffer (similar to a "victim cache" [47]). Figure B-7 gives the ANSI C representation of an STLB entry. Figure B-8 gives the MIPS assembly code to load an entry from the STLB into the hardware TLB within the TLB handler. LibOSes can map the STLB read-only into their address spaces. LibOS modifications of the hardware TLB are propogated to the STLB. The libOS program counter that Aegis vectors TLB misses to is given in the libOSes environment structure.

The following routines read, insert, and modify TLB entries.

`int ae_tlbwr(addr_t va, struct lo lo);`

Insert TLB entry for virtual address va. lo holds the physical page number, protection information, and various software tags. Figure B-9 gives its ANSI C representation.

`void ae_tlbrprotn(addr_t va, int len);`

Read-protect the region [va, va+len*PAGESIZ). Always succeeds.

`void ae_unprotn(addr_t va, int len);`

Make the region [va, va+len*PAGESIZ) writable. Fails if the current process does not have write access to any page.

`void ae_tlbdeleten(addr_t va, int len);`

Delete [va, va+len*PAGESIZ) from the TLB. Always succeeds.

`void ae_tlbflush(void);`

Flush all entries from STLB and TLB. Always succeeds.

```
# Software refill of TLB: uses an STLB cache.

# 1. Compute hash function
mfc0    k0, c0_tlbcxt           # get virtual page number and 11 bit process tag
mfc0    k1, c0_tlbcxt           # twice

#   Our hash function combines process tag with the lower bits of the virtual
#   page number (VPN) that missed:
#       (((c0_tlbcxt << 17 ˆ c0_tlbctx) >> 16 ) & STLB_MASK&˜7)
sll     k0, k0, 17                  # move VPN up
xor     k1, k0, k1                  # combine with process tag
srl     k1, k1, 16                  # move down (8 byte align)
andi    k0, k1, STLB_MASK & ˜7  # remove upper and lower bits.

# 2. Index into STLB
lui     k1, HI(stlb)            # load STLB (at known location)
add     k1, k0, k1              # index into STLB

# 3. Load the physical page entry and STLB tag
lw      k0, LO(stlb)+4(k1)      # TLB entry
lw      k1, LO(stlb)+0(k1)      # STLB tag

# 4. Load TLB: we first load the fetched TLB entry into tlblo (but do not write
#    this register into the TLB); we then re fetch tlbcxt in preparation to its
#    comparison to the STLB tag.
mtc0    k0, c0_tlblo            # (optimistically) load TLB entry
mfc0    k0, c0_tlbcxt           # get context again
nop                             # delay slot

# 5. Check tag (does not match  > jump to miss handler)
bne     k0, k1, stlb_miss       # compare tags to see if we got a hit
mfc0    k1, c0_epc              # get exception program counter

# 6. Tags matched: install entry into the TLB
tlbwr                       # write tlblo to TLB

# 7. Return from exception
j       k1                  # jump to resumption address
rfe                         # return from exception
```

Figure B-8: Assembly code used by Aegis to lookup mapping in STLB (18 instructions).

```
/* LO portion of TLB mapping. */
struct lo {
        unsigned
                        w:1, /* write perm? */
                        :7,   /* reserved for software (libOS) */
                        g:1,  /* Global: TLB ignores the PID match req */
                        v:1, /* Valid: If not set, TLBL or TLBS miss occurs*/
                        d:1, /* Dirty */
                        n:1, /* Non cacheable. */
                        pfn:20;  /* Page Frame Number: 31..12 of the pa */
};
```

Figure B-9: Hardware defined “low” portion of a TLB entry (i.e., the part bound to a virtual page number).

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, July 1986.

[3] T.E. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, pages 92–94, 1992.

[4] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[5] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991.

[6] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, USA, November 1994.

[7] K. Bala, M.F. Kaashoek, and W.E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 243–253, November 1994.

[8] J. Barrera. Invocation chaining: manipulating light-weight objects across heavy-weight boundaries. In *Proc. of 4th IEEE Workshop on Workstation Operating Systems*, pages 191–193, October 1993.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, USA, December 1995.

[10] B.N. Bershad, D.D. Redell, and J.R. Ellis. Fast mutual exclusion for uniprocessors. In *Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 223–237, October 1992.

[11] Nathaniel Borenstein and James Gosling. Unix emacs: A retrospective. In *ACM SIGGRAPH Symposium on User Interface Software*, October 1988.

[12] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[13] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[14] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Sc hwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of 1996 Usenix Technical Conference*, pages 153–163, January 1996.

[15] D. L. Chaum and R. S. Fabry. Implementing capability-based protection using encryption. Technical Report UCB/ERL M78/46, University of California at Berkeley, July 1978.

[16] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, November 1994.

[17] D. R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating Systems Review*, 18:12–20, October 1984.

[18] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, pages 200–208, Philadelphia, PA, USA, September 1990.

[19] D.D. Clark. On the structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[20] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM J. Research and Development*, 25(5):483–490, September 1981.

[21] H. Custer. *Inside Windows/NT*. Microsoft Press, Redmond, WA, 1993.

[22] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. *Information Processing 71*, 1971.

[23] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, pages 2–13, London, UK, August 1994.

[24] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, pages 14–24, London, UK, August 1994.

[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[26] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the Sixth SIGOPS European Workshop*, pages 62–67, September 1994.

[27] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Operating systems review*, January 1995.

[28] D. R. Engler, D. Wallach, and M. F. Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT, March 1995.

[29] Dawson R. Engler. Simple, robust online verification of program correctness. Submitted for publication.

[30] Dawson R. Engler. Efficient verification of demonically-implemented integer functions (or, demonic determinism, trusted results). available on request, December 1997.

[31] D.R. Engler and M.F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1996*, pages 53–59, Stanford, CA, USA, August 1996.

[32] Marc Fiuczynski and Brian Bershad. An extensible protocol architecture for application-specific networking. In *Proceedings of the 1996 Winter USENIX Conference*, pages 55–64, January 1996.

[33] B. Ford, K. Van Maren, J. Lepreau, S. Clawson, B. Robinson, and Jeff Turner. The FLUX OS toolkit: Reusable components for OS implementation. In *Proc. of Sixth Workshop on Hot Topics in Operating Systems*, pages 14–19, May 1997.

[34] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI 1996)*, October 1996.

[35] Bryan Ford and Sai R. Susarla. CPU inheritance scheduling. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI 1996)*, October 1996.

[36] G. Ganger and Y. Patt. Metadata update performance in file systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 49–60, November 1994.

[37] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Technical Conference*, 1997.

[38] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer*, pages 34–45, June 1974.

[39] D. Golub, R. Dean, A. Forin, and R. Rashid. UNIX as an application program. In *USENIX 1990 Summer Conference*, pages 87–95, June 1990.

[40] J. Gosling. Java intermediate bytecodes. In *Proc. of ACM SIGPLAN workshop on Intermediate Representations*, pages 111–118, march 1995.

[41] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[42] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg andJ. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[43] J.H. Hartman, A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A communication-oriented operating system. Technical Report TR 94-20, University of Arizona, Tucson, AZ, June 1994.

[44] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 187–199, October 1992.

[45] D. Hitz. An NFS file server appliance. Technical Report 3001, Network Applicance Corporation, March 1995.

[46] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 39–51, May 1992.

[47] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[48] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[49] M.F. Kaashoek, D.R. Engler, D.H. Wallach, and G. Ganger. Server operating systems. In *SIGOPS European Workshop*, pages 141–148, September 1996.

[50] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[51] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for development of application-specific virtual memory management. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1993*, pages 48–64, October 1993.

[52] B. W. Lampson. Hints for computer system design. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 33–48, December 1983.

[53] B.W. Lampson. On reliable and extendable operating systems. *State of the Art Report, Infotech*, 1, 1971.

[54] B.W. Lampson and R.F. Sproull. An open operating system for a single-user machine. *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 98–105, December 1979.

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

[56] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, , and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on selected areas in communication*, 14(7):1280–1297, September 1996.

[57] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–188, December 1993.

[58] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[59] Steve Lucco. Personal communication. Use of undocumented proprietary formats as a technique to impede third-party additions, August 1997.

[60] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Victor Lee, Anant Agarwal, and M. Frans Kaashoek. UDM: User Direct Messaging for General-Purpose Multiprocessing. Technical Memo MIT/LCS/TM-556, March 1996.

[61] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, Asheville, NC, USA, 1993.

[62] David Mazieres and M. Frans Kaashoek. Secure applications need flexibile operating systems. In *HotOS-VI*, 1997.

[63] David Mazieres and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the6th Workshop on Hot Topics in Operating Systems*, May 1997.

[64] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, San Diego, CA, Winter 1993. USENIX.

[65] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, USA, November 1987.

[66] A. C. Myers and B. Liskov. Decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[67] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software-managed TLBs. In *20th Annual International Symposium on Computer Architecture*, pages 27–38, May 1993.

[68] NCSA, University of Illinois, Urbana-Champaign. NCSA HTTPd. http://hoohoo.ncsa.uiuc.edu/index.html.

[69] J. K. Ousterhout. Why aren' t operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[70] V. Pai, P. Druschel, and W. Zwaenepoel. I/O-lite: a unified I/O buffering and caching system. Technical Report *http://www.cs.rice.edu/ vivek/IO-lite.html*, Rice University, 1997.

[71] Przemyslaw Pardyak and Brian Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–212, October 1996.

[72] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995.

[73] D. Probert, J.L. Bruno, and M. Karzaorman. SPACE: A new approach to operating system abstraction. In *International Workshop on Object Orientation in Operating Systems*, pages 133–137, October 1991.

[74] J.S. Quarterman, A. Silberschatz, and J.L. Peterson. 4.2BSD and 4.3BSD as examples of the UNIX system. *Computing Surveys*, 17(4):379–418, December 1985.

[75] D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[76] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. Phd Thesis, Technical Report 376, Cambridge, 1995.

[77] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[78] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 213–228, October 1996.

[79] C. Small and M. Seltzer. Vino: an integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard, 1994.

[80] Christopher Small and Margo Seltzer. A comparison of os extension technologies. In *Proceedings of the 1996 USENIX Conference*, 1996.

[81] Richard Stallman. Emacs, the extensible, customizable self-documenting display editor. In *ACM SIGPLAN SIGOA Symposium on Text Manipulation*, June 1981.

[82] V. Buch T. von Eicken, A. Basu and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, USA, 1995.

[83] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[84] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S.J. Mullender, A. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[85] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til: A type-directed optimizing compiler for ml. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[86] D.L. Tennenhouse and David J. Wetherall. Towards an active network architecture. In *Proc. Multimedia, Computing, and Networking 96*, January 1996.

[87] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 110–121, October 1994.

[88] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating data and control transfer in distributed operating systems. In *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 2–11, San Francisco, California, October 1994.

[89] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993.

[90] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, November 1994.

[91] C. A. Waldspurger and W. E. Weihl. Stride scheduling: deterministic proportional-share resource management. Technical Memorandum MIT/LCS/TM528, MIT, June 1995.

[92] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM '96)*, Stanford, California, August 1996.

[93] Deborah A. Wallach. *Supporting application-specific libraries for communication*. PhD thesis, M.I.T., 1996.

[94] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessing operating system. *Communications of the ACM*, 17(6):337–345, July 1974.

[95] M. Yuhara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, pages 153–165, San Francisco, CA, USA, January 1994.