The Design and Implementation of a Prototype Exokernel Operating System

by

Dawson R. Engler

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1995

Author . . . .
Department of Electrical Engineering and Computer Science
January 1995

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

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Committee on Graduate Students

The Design and Implementation of a Prototype Exokernel Operating System

by

Dawson R. Engler

Submitted to the Department of Electrical Engineering and Computer Science on January 1995, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

Abstract

Traditional operating systems abstract hardware resources. The cost of providing these abstractions is high: operating systems are typically unreliable, complex, and inflexible. More importantly, applications that are built on these systems are both inefficient and limited in scope: most resource management decisions cannot be made by application designers, leaving them with little recourse when the management of these resources is inappropriate. We believe that the quest to abstract physical resources has sharply limited the flexibility and ambition of applications.

In this thesis, we propose a new operating system structure, *exokernel*, which is built on the premise that an operating system should not abstraction physical resources. Rather, operating systems should concentrate *solely* on multiplexing the raw hardware: from these hardware primitives, application-level libraries and servers can directly implement traditional operating system abstractions, specialized for appropriateness and speed.

This thesis motivates the need for a new operating system structure, provides a set of precepts to guide its design, discusses general issues that exokernels must deal with in multiplexing physical hardware, and describes and measures a prototype exokernel system. In many cases this prototype system is an order of magnitude more efficient than a traditional operating system. Furthermore, it supports a degree of flexibility not attained in any operating system. For example, virtual memory is efficiently implemented entirely at the application level.

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

The Design and Implementation of a Prototype Exokernel Operating System

by
Dawson R. Engler

Submitted to the Department of Electrical Engineering and Computer Science
on January 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Traditional operating systems abstract hardware resources. The cost of providing these abstractions is high: operating systems are typically unreliable, complex, and inflexible. More importantly, applications that are built on these systems are both inefficient and limited in scope: most resource management decisions cannot be made by application designers, leaving them with little recourse when the management of these resources is inappropriate. We believe that the quest to abstract physical resources has sharply limited the flexibility and ambition of applications.

In this thesis, we propose a new operating system structure, *exokernel*, which is built on the premise that an operating system should not abstraction physical resources. Rather, operating systems should concentrate *solely* on multiplexing the raw hardware: from these hardware primitives, application-level libraries and servers can directly implement traditional operating system abstractions, specialized for appropriateness and speed.

This thesis motivates the need for a new operating system structure, provides a set of precepts to guide its design, discusses general issues that exokernels must deal with in multiplexing physical hardware, and describes and measures a prototype exokernel system. In many cases this prototype system is an order of magnitude more efficient than a traditional operating system. Furthermore, it supports a degree of flexibility not attained in any operating system. For example, virtual memory is efficiently implemented entirely at the application level.

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

"A considerable amount of bitter experience in the design of operating systems has been accumulated in the last few years, both by the designers of the systems which are currently in use and by those who have been forced to use them. As a result, many people have been led to the conclusion that some radical changes must be made, both in the way we think about the functions of operating systems and in the way they are implemented." — Butler Lampson, 1971.

The cheapest, fastest and most reliable component of an [operating system] are those that aren't there — Gordon Bell, paraphrase.

Acknowledgments

This thesis and its ideas has been very much a joint project with my advisor Frans Kaashoek: many of the insights and motivations I present resulted from our frequent discussions. James O'Toole was involved in much of the early exokernel meetings and publication [36, 37, 35]; the idea of dynamically loading application code into the kernel was his. Without the involvement of these two men there would be no exokernel.

I give special thanks to my office mate, Deborah Wallach. Her high scientific standards and biting humor are much appreciated.

Special recognition goes to Frans Kaashoek, Massimiliano Poletto, Carl Waldspurger, and Deborah Wallach, who all read the complete thesis (Frans did so multiple times). Their heroic editorial endurance is greatly appreciated. Kevin Lew and Wilson Hsieh also gave very useful feedback. The voice of sanity (played alternatively by Deborah Wallach and Robert Bedichek) removed a few of my more fantastic claims.

I thank Sandeep Gupta for writing the timer interrupt and inverted page-table implementations.

Greg Andrews first introduced me to research. Without his assistance, it is unlikely that I would be doing anything nearly so joyous.

Most especially I thank Kim Danloe, who has tolerated long hours, monosyllabic conversations, and general surliness throughout my research career: without her support, the ivory tower would be lonely and empty.

Contents

List of Figures

Chapter 1

Introduction

The operating system is interposed between applications and the physical hardware. Therefore, its structure has a dramatic impact on the performance and the scope of applications that can be built on it. Since its inception, the field of operating systems has been attempting to identify an appropriate structure: from familiar monolithic [99, 103] to micro-kernel operating systems [2, 43] to more exotic language-based [81] and virtual machine [27, 40] operating systems. This search, spanning the last three decades, has been aggressive and vigorous but, unfortunately, has not been satisfactorily resolved [13, 22, 36, 43, 63, 72].

In this thesis we propose a new operating system structure that dramatically departs from previous work. An *exokernel* eliminates the notion that an operating system should provide abstractions on which applications are built. Instead, it concentrates solely on multiplexing the raw hardware: from these hardware primitives, *application-level* libraries and servers can directly implement traditional operating system abstractions, specialized for appropriateness and speed.

1.1 The Problem

The cost of inappropriate, inefficient operating system abstractions has had a large impact on applications [5, 13, 44, 45, 72, 91, 97]. This situation has persisted for the last three decades, and has survived numerous assaults (e.g., object-oriented operating systems and micro-kernels). Any concept that cannot be realized after such a long period of time should be reexamined.

The standard definition of an operating system is software that securely multiplexes and *abstracts* physical resources. The core assumption of this definition is that it is possible both to define abstractions that are appropriate for all areas and to implement them to perform efficiently in all situations. We believe that this definition, specifically its view of the OS as an abstractor of hardware, sets an unattainable grail for the operating system designer and sharply curtails the ambition of systems that can be built on top of the resultant system. The basic intuition behind our arguments is that no OS abstraction can fit all applications: it is *fundamentally* impossible to abstract resources in a way that is useful to all applications and to implement these abstractions in a way that is efficient across disparate needs. Operating systems that have been built in this manner have one or (usually) more of the following characteristics: they are complex and large, decreasing system reliability and aggressively discouraging change; they are overly general, making their use expensive and their implementation consume significant fractions of machine resources; and they enforce a high-level interface, precluding the efficient implementation of new abstractions outside of the OS.

We believe that the attempt to provide OS abstractions is the root of many operating system problems. We contend that these problems can be solved directly by the elimination of OS abstractions, lowering the interface enforced by the OS to a level close to the raw hardware. It is important to note that we *favor* abstractions, but they should be implemented outside the operating system so that application designers can select among a myriad of implementations or, if necessary, "roll their own."

The structure of this chapter is as follows: Section 1.2 examines the costs of operating system abstractions; Section 1.3 outlines specific features of our target operating system structure; and Section 1.4 explores some of the advantages of this new OS structure. Section 1.5 addresses common concerns expressed about an exokernel structure (e.g., possible portability and size problems). We outline the remainder of the thesis in Section 1.6 and conclude in Section 1.7.

The following definitions are used throughout this thesis.

Operating system: software that applications cannot either change or avoid. User-level device drivers, privileged servers, and kernels are all included by this definition.

Application-level: software that can be changed and/or avoided by any application. Libraries and applications are included in this definition.

User-level: software which can require high privileges to adapt or replace. For example, replacing a device driver often requires supervisor privileges). The key distinction between user- and application-level is that ordinary programs can replace, extend and implement application-level software. The same condition does not necessarily hold for user-level software, and thus dramatically limits its flexibility.

1.2 Motivation

The goal of the OS designer should be to push the interface defined by the OS to the level of the raw hardware. The distinction between application-level and user-level is an important one. Many microkernel proponents do not appear to properly appreciate it: while moving in-kernel implementations of OS services (such as filesystems and device drivers) to user-space can improve the extensibility of the system as seen by the "super-user," it does little for the more common class of applications.

The thesis behind the exokernel structure is that the operating system should not abstract physical resources. The sole role of an operating system should be to ensure protection; it should not either hide hardware functionality or inhibit application-level resource sharing. Abstraction and management of hardware resources should be left to application-level libraries and servers which are in a better position to implement these mechanisms than any general purpose operating system.

Providing operating system abstractions results in: poor reliability, poor adaptability, poor performance and poor flexibility. We discuss each of these points below.

1.2.1 Poor reliability

Abstracting resources (e.g., providing a full-featured virtual memory system with copy-on-write, memory-mapped I/O and other features) requires a large amount of complex, multi-threaded code. These characteristics, along with dynamic storage allocation and management and the paging of kernel data structures and code, greatly decrease the reliability of the system. Large pieces of software have large numbers of bugs, and therefore, the large size of current operating systems has a direct impact on their reliability. As noted by Spier:

> Multiple-user operating systems are especially susceptible to software-quality problems. In an attempt to satisfy the market's ever growing demand for various sophistications, more and more code is being crammed into already exceedingly complex monolithic operating system monitors (i.e., supervisors, executives) to the point of rendering them virtually impossible to maintain at an acceptable level of correctness. The industry is well aware of this fact, and every change — no matter how trivial — is typically cautiously debated and evaluated as to whether or not it really has to be incorporated; its implementation is typically undertaken with great reluctance [89].

The interesting fact about this observation is that it was made in 1973; since then, operating system complexity has *increased*. And for good reason: traditional operating system design forces kernel implementors to implement the virtual machine for radically different applications and requirements. To do this successfully, OS designers have had to include multiple standards and interfaces within the same operating system. In an exokernel design, these standards and implementations could each be realized in an isolated fashion (e.g., in a library), dramatically simplifying both their design and their implementation. Furthermore, unlike a micro-kernel design (which also addresses this problem), modularity does not have to be expensive. Unlike a microkernel server, a library operating system can trust the application that uses it and, therefore, perform most operations in the same address space as the application (eliminating the cost of cross-domain IPC, security checks, copying arguments, etc.). Of course, the danger to such flexibility is pervasive code duplication. We believe this can be reduced in two ways. The first by using dynamic linking. The second is that application-level code is generally simpler than operating system code. For example, it can be written for a restricted set of application domains whereas the operating system implementation must be extremely general purpose. We provide additional examples in Section 1.4.

1.2.2 Poor adaptability

An operating system that provides a virtual machine is large and complicated. Changing large, complicated pieces of software is *hard*. This creates a disincentive to incorporate new features or tune existing ones. Furthermore, since all applications "depend on" the operating system, change is not localized, which is an additional discouragement to operating system alteration. Finally, only the kernel architect can incorporate new changes, further restricting adaptability.

This point must be emphasized: the operating system is used by all applications; consequently, *changes to it cannot be localized to a particular application.* The direct consequence of this structures is that it cannot be changed by non-trusted personnel. Put another way, the implementation of new abstractions, the tuning of old ones, or even simple bug fixes can only be incorporated with great care. Thus, the operating system interface and implementation is, for practical purposes, fixed and unchanging no matter how inappropriate it is. However, if this interface is pushed lower (preferably to the raw hardware), more and more of its implementation can be done in application space and, therefore, localized to particular applications, allowing ambitious application-level experimentation and, furthermore, enabling the results of these experiments to be used on a large scale, far more readily than modifications to an operating system. An application-level library can simply be linked into an application without destroying system integrity. This single characteristic enables pervasive experimentation and customization. (Without discipline such a system would become a Babel of incompatible interfaces: we believe that this rigor should be implemented through standards rather than encoded in system structure.)

The inability of traditional operating systems to support localized change can be easily seen: few of the good ideas in the last 10 years of operating system research have been incorporated (or allowed at application-level) by any operating system other than the one they were developed on. For example, what operating systems support scheduler activations [4], multiple protection domains within a single-address space [20], efficient IPC [67], or efficient and flexible virtual memory primitives [5, 45, 58]? The structure of traditional operating systems ensures that good ideas will, for the most part, remain research curiosities. In an exokernel based system, these mechanisms can be implemented directly at application-level without special privileges, and without compromising system integrity. We expect that this single characteristic will dramatically increase both the degree and the utilization of operating system experimentation.

1.2.3 Poor performance

OS abstractions are often overly general, as they attempt to provide any feature needed by any reasonable application, and all applications must use a given OS abstraction. Applications that do not need a given feature pay unnecessary overhead [4, 72]. Additionally, simply using a given feature is costly, since the operating system must interpret a myriad of system call options [72]. Furthermore, the mere existence of OS abstractions consumes significant amounts of main memory, cache space, TLB space, and cycles, which could be used by applications to perform useful work.

Finally, any OS implementation makes trade-offs: whether to use a hierarchical or inverted page-table, whether to optimize for frequent reads or random writes, whether to have copy-on-write or a large page size, etc. Unfortunately, any trade-off penalizes applications that were neglected or not anticipated by the OS designer. However, this situation is easily avoidable: if the OS does not abstract resources, it does not have to make such trade-offs.

1.2.4 Poor flexibility

Poor reliability, poor adaptability, and poor performance in operating systems could be acceptable if applications could simply ignore the operating system and implement their own abstractions. Unfortunately, the high-level nature of current operating system interfaces makes this approach infeasible. At best, applications can emulate the desired feature on top of existing OS abstractions; unfortunately, such emulation is typically clumsy, complicated, and prohibitively expensive. For example, once the application has no access to the raw disk interface, database records must be emulated on top of files: further examples of conflicts between OS-provided abstractions and what applications require can be found in the literature [5, 13, 44, 45, 72, 91, 97].

The abstractions of modern operating systems hide useful information and capabilities. For instance, even simple operations such as monitoring TLB misses or determining the physical page used to map a virtual one are not possible on *any* operating system, even though the effects of cache conflicts can be great [104]. Finally, the high-level nature of operating systems is self-fulfilling: removing application-level access to hardware resources and operations forces any software that must use these resources to be implemented in the kernel. This, in turn, completes a vicious circle where additional functionality is pulled into the kernel because it needs to directly access resources abstracted by the previous wave of software. For example, a file system requires, at most, a protected directory and name-space structure; instead, modern operating systems implement all operations (read, write, create) in the filesystem itself. This removes the ability of applications to directly access file buffers, decide which information should be cached, or determine the layout and structure of the files themselves. Furthermore, it forces additional policies and trade-offs to be incorporated into the file system: buffer management decisions (which to replace, which to allocate), prefetching (what blocks and how many to prefetch) and what access patterns will be anticipated. The single insight that we wish to offer is that *these policy decisions and implementation trade-offs are unnecessary.* If the operating system tracks ownership, application-level libraries can implement the rest — more reliably, efficiently, and *appropriately* than any general-purpose operating system could hope to do [3].

In closing, there is no "best way" to implement a high-level abstraction; hard-wiring the implementation in the operating system, where it can neither be changed nor avoided is, therefore, a dangerous practice. [1]

1.2.5 Summary

In short, operating systems are complex, fragile, inflexible, and slow, because they attempt to provide a general-purpose virtual machine. The operating system is basically hardware masquerading as software: it cannot be changed, all applications must use it, and the information it hides cannot be recovered. Operating system design should incorporate the lessons learned by hardware designers during the transition from CISC to RISC: hardware should provide primitives, *not* high-level abstractions.

1.3 The Solution: Exokernels

We contend that the solution to all of these difficulties is straightforward: eliminate operating system abstractions. The OS should only export physical resources in a secure manner; it should not present a machine-independent interface to applications.

In this section we give a quick sketch of an OS structure that embodies an "abstraction-free", low-level interface. We call such a structure an *exokernel.* The sole function of an exokernel is to allocate, deallocate, and multiplex physical resources in a secure way. The resources exported by this kernel are those provided by the underlying hardware: physical memory (divided into pages), the CPU (divided into time-slices), disk memory (divided into blocks), DMA channels, I/O devices, translation look-aside buffer, addressing context identifiers, and interrupt/trap events.

Security is enforced by associating every resource usage or binding point with a guard that checks access privileges. For example, as one of the steps in preserving memory integrity, the kernel guards the TLB by checking any virtual-to-physical mappings given by applications before they are inserted into the TLB.

[1] Operating systems are governments. Politically aware readers will recognize in our arguments many used in debates against strong centralized governments (e.g., individual versus state rights, capitalism versus communism, etc.). This is a useful view: many of the intuitions and arguments are similar.

Those few global optimizations that require kernel participation can be implemented by an exokernel's control over control over the allocation and revocation of physical resources (this issue is discussed more thoroughly in Section 2.8). With this control it can enforce proportional sharing, or what resources are allocated to which domains.

To make these examples concrete, we outline what address spaces, time-slices, and IPC might look like under the regime we have described (more thorough descriptions will be provided in subsequent chapters). The details we present are highly machine-specific, but the general outline should be similar across machines; the main goal in each is to answer the question: what is the minimum functionality that the kernel needs to provide in order for this primitive to be implemented in application space?

Address space To allow application-level virtual memory, the OS must support bootstrapping of page-tables, allocation of physical memory, modification of mapping hardware (e.g., TLB), and exception propagation. The simplest bootstrapping mechanism is to provide a small number of "guaranteed mappings" that can be used to map the page-table and exception handling code. Physical memory allocation should support requests for a given page number (enabling such techniques as "page-coloring" for improved caching [14]). Privileged instructions (e.g., flush, probe, and modify instructions) can be wrapped in systems calls, and those that write to privileged state (e.g., TLB write instructions) are associated with access checks. Exception propagation is done in a direct manner by saving a few scratch registers in some agreed-upon location in application-space and then jumping to an application-specified PC-address [97].

All of these operations can be sped up by downloading application code into the kernel [13, 36]. This implementation techniques aside, the full functionality provided by the underlying hardware should be exposed. For instance reference bits, the ability to disable caching on a page-basis, the ability to use different page sizes, etc. should all be available for application-level control.

Process The only state needed by the operating system to define a process is a set of exception program counters that the operating system will jump to on an exception, an associated address space, and both prologue and epilogue code to be called when a time-slice is initiated and expires. Placing context-switching under application control (through the application-defined prologue and epilogue code) enables techniques such as moving the program counter out of critical sections at context-switch time [15].

IPC The basic functionality required by IPC is simply the transfer of a PC from one protection domain to an agreed-upon value in another, with the donation of the current time-slice, installation of the called domain's exception context, and an indication of which process initiated the call (security reasons for this last constraint are given in Lampson [64]). This lightweight cross-domain calling mechanism implements the bare-minimum required by any IPC mechanism, allowing the application to pay for just the functionality that it requires. For instance, a client that trusts a server may allow the server to save and restore the registers it needs, instead of saving the entire register file on every IPC. Since the machine state of current RISC machines is growing larger [76], this can be crucial for good performance.

This is far from a complete enumeration of all system primitives (for example, we have so

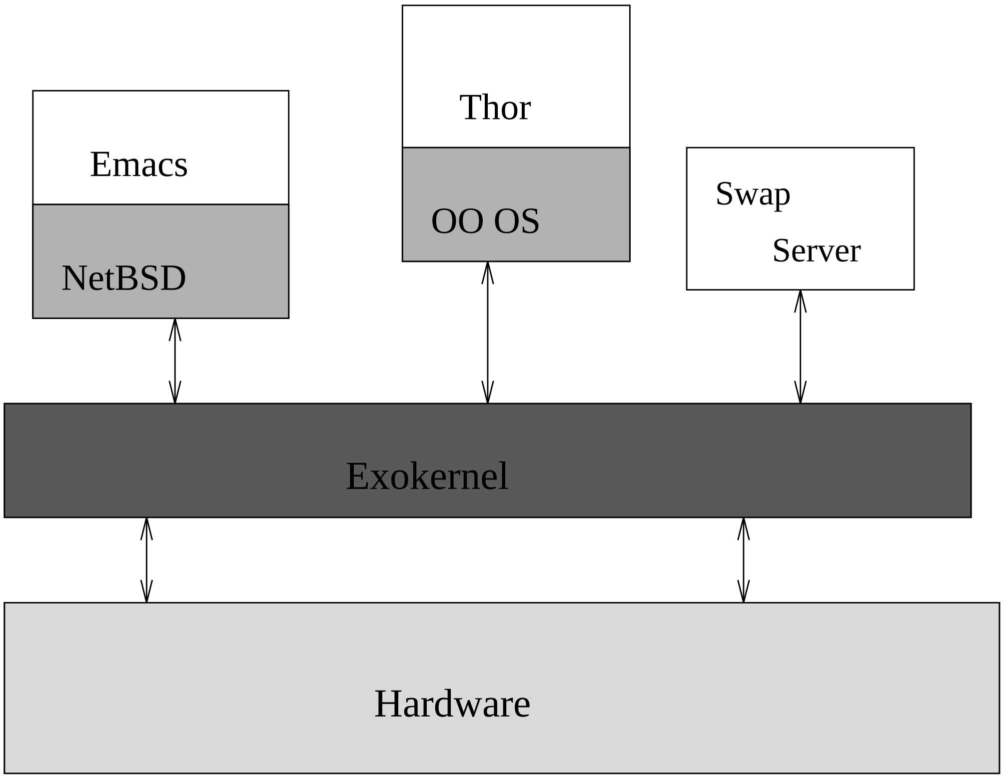

Figure 1-1: An example exokernel-based system

far neglected disks and devices), but should give a feel for what level of functionality the OS is *required* to provide. The bare minimum is much removed from the policy-laden, overly general, and restrictive implementations of today's operating systems.

The structure of a possible exokernel-based system is given in Figure 1-1. This structure consists of a thin exokernel veneer that multiplexes physical resources securely and library- and server-based operating systems that implement system objects and policies. This structure allows the extension, specialization and even replacement of abstractions. For example, page-table structures can vary across different applications. UNIX processes can use a traditional hierarchical page-table; protected objects can use small linear page tables, and parallel operating systems hierarchical page-tables for local data and inverted page-tables for global data. Figure 1-2 illustrates different structures that may be appropriate to different address space sizes; these structures can co-exist on the same machine.

1.4 Impact of an Exokernel Structure

We discuss how our proposed structure solves the traditional problems of reliability, efficiency, and extensibility; these points have at their core the observation that the most efficient, reliable, and extensible OS abstraction is the one that is not there.

Reliability Exposing hardware resources safely and efficiently requires neither sophisticated algorithms nor many lines of code. As a result, an exokernel can be small and readily

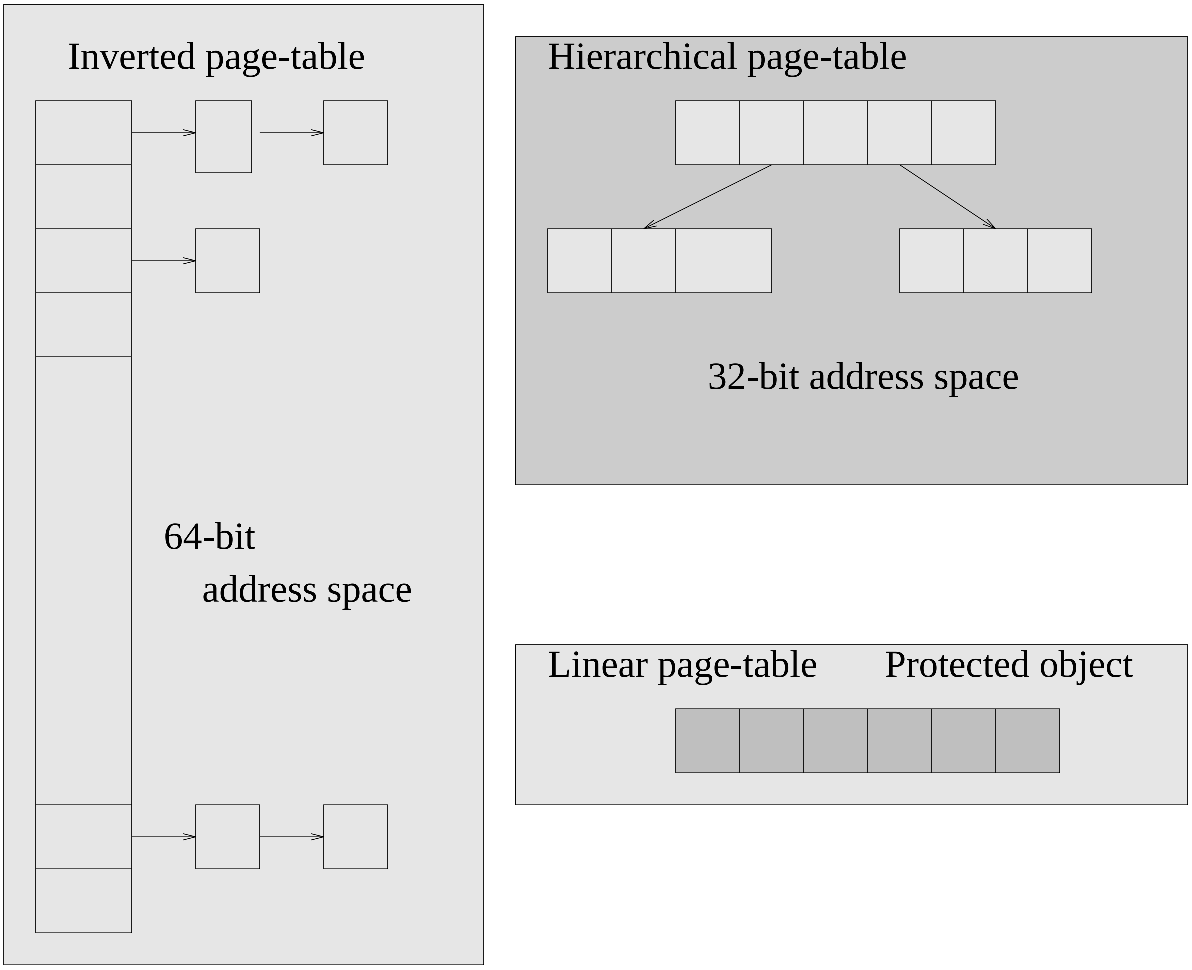

Figure 1-2: A set of example address spaces

understood: both of these properties aid correctness.

Additionally, the application-level implementation of operating system services is likely to be much simpler in structure and smaller in realization than a traditional, general-purpose OS. For example, it does not have to multithread among multiple, potentially malicious entities, nor worry about the peculiar characteristics of supervisor mode (e.g., the particular locking constraints that arise within the kernel to guard against loss of interrupts and deadlock). Finally, since this application operating system "trusts" the application, it can use application state directly and simply; a general-purpose operating system is constantly copying user data, guarding against illegal addresses, and checking for validity. All of these concerns can be ignored in an application-level operating system since, if the application does something wrong, the damage is to itself only.

Adaptability The kernel's simplicity enables easy modification. Furthermore, since most of the operating system code deals with the allocation and protection of system resources, there is not much that needs to be tuned.

By allowing fundamental system primitives (e.g., processes) to be implemented at application-level, we have removed the dependence of the entire system on the correctness of these implementations. In other words, by localizing a change within a single library or server operating system, applications that wish to use a new feature can link it in (albeit with steps such as dynamic linking to reduce memory consumption). Those that do not, do not need to. Traditional operating systems occupy the unfortunate position of having every application depend on their correct and appropriate implementation. This dependence drastically limits the degree to which experimentation can be carried out and the results used.

An important point of this structure is that it is now easy to "overthrow" inappropriate implementations. Over time, any successful piece of software acquires features, generality, bulk. In the case of operating systems, applications are forced to use such a piece of software with no recourse. In an exokernel structure they can avoid inappropriate library implementations by simply linking in a different library. (Of course, we hope that the exokernel itself will not fall victim to this trend: hopefully, the simplicity of its central premise — that the operating system should not abstract hardware resources — will preclude this fate.)

Another advantage of improved adaptability is that traditional OS abstractions can be implemented in application space, with access to application-level development environments (e.g., debuggers and profilers).

Efficiency An exokernel pushes resource management out into application space, allowing implementations to exploit application-specific knowledge in making trade-offs (e.g., optimizing for reads or random writes, sparse address spaces, etc.). Furthermore, since implementations can be highly specialized, they can eliminate the cost of generality present in most OS abstractions. This structure allows a broad pool of non-kernel architects to implement alternative implementations. Since the entire system does not depend on these implementations non-privileged implementors can readily alter and experiment with application-level operating systems. Furthermore, these operating systems will be readily used, since they do not have to be used by the entire system (and hence trusted in a very real sense).

Finally, since most interactions between an application and its companion operating system can occur in the same address space, the current contortions to minimize the cross-domain costs of TLB pollution/misses, system call traps, and context-switches are completely obviated.

Flexibility Applications can now implement system abstractions in ways that are fundamentally impossible on traditional operating systems. Radical page-table structures, process abstractions, address spaces, and file systems can be constructed safely and efficiently on top of this structure. We expect that this freedom will enable a broad class of applications that are not feasible under current operating systems.

The ability of application-level operating systems to support powerful, efficient, and unusual abstractions cannot be overemphasized. By allowing any application writer to implement fundamental system objects, the degree, ease, and pervasiveness of experimentation and *utilization* of the results of this experimentation can dramatically increase.

1.5 Discussion

In this section we discuss some common concerns that have been expressed about an exokernel structure. They are arranged in roughly decreasing order of occurence.

Doesn't an exokernel cause portability problems? There are two levels of portability that must be provided, machine portability and OS interface portability. The first may be achieved in the standard manner: namely, a low-level layer that hides machine dependence. The second can be achieved through application-level implementations of industry standards (e.g., POSIX). The difference is that the implementation of these layers are now in application space and, therefore, can be replaced without special privileges. Such an organization allows efficient and unusual implementations of system objects to be realized in a flexible and localized manner, directly simplifying the addition of new standards and features not anticipated by kernel architects.

Won't trap costs go up, decreasing system performance? The exokernel dispatches most hardware traps far more efficiently than in current operating systems (see Chapter 4). Furthermore, the potentially high cost of exceptions such as TLB faults can be countered either through caching state in the exokernel or by safely importing application code into the kernel. We provide initial measurements of basic system costs in Chapter 4. Finally, trusted servers can be used: flexibility will come at "zero cost" for those applications that don't need it.

However, it is worth noting that this concern is misguided: even if the overhead of basic operations increases, we believe that the cost of these operations is not the most important system overhead. Rather, the cost of *managing* system objects constitutes the main system overhead: exceptions comprise very little of this metric. By pushing the implementation of these managerial mechanisms out into application space, many specialized and more efficient implementations will be able to exist. We expect this will translate into real improved system performance.

What happens to the system structure when any application can define its own interfaces? The vast flexibility available to applications on an exokernel system raises this legitimate question. As language, GUI and standard library implementors can attest, preventing a Babel of incompatible interfaces is quite simple: define standards and conventions. As we argue in this thesis, the effects of hard-wiring interfaces into the system structure are a convincing demonstration that such an approach is not the right way to define a standard.

In closing, while an exokernel allows the possibility of a chaotic system it also allows the creation of a harmonious, elegant one as well: a system whose structure does not have to be anticipated by kernel architects.

OS software is too difficult to change whether it is in application space or not. While it is indeed true that changing large pieces of software is difficult, it is clear that changing such software in the kernel is even more difficult. By moving most code to application level, modifications to it become no more (or less) difficult than changing any other large piece of software. Furthermore, as we argue in Section 1.4, application-level libraries and servers can be significantly simpler and cleaner than traditional operating system implementations.

Finally, this question has an incorrect assumption: it is usually not possible to change traditional operating systems *at all*, regardless of their complexity. Either the OS source is not available or proprietary, or modification requires root privileges. Furthermore, since any OS modification must be used by *all* applications change is actively discouraged. With an exokernel structure, proprietary binaries can always be replaced by application-level software.

Won't executables become large? As a practical matter, libraries that implement traditional abstractions can be sizable. However, shared libraries can be used to combat this problem. They have been used successfully in equivalent situations. For example, the X-window libraries are typically dynamically linked. In the worst case, servers can be used to multiplex code, data and threads of control.

Of course, this argument assumes that a reasonable amount of sharing takes place on the system. It is true that if an application uses its own specialized library then it will have to link in this library's code. But, even without special methods, this should not be a large overhead: applications typically only use a very small subset of the OS interface, and therefore most functionality does not need to be linked in. Furthermore, as argued in Section 1.4, these libraries should be much simpler and smaller because they can be tailored to a particular domain. Finally, since these implementations reside in application space, their text and data can be rearranged for increased locality.

Does an exokernel structure introduce a new class of bugs? An unpleasant possibility of migrating OS abstractions into an application's address space is that bugs can result in very unusual effects. For example, if an application performs wild writes over its page-tables, the effects can be quite surprising. This problem can be addressed in a number of ways. One possibility is to implement abstractions through servers (who are protected by address-space fire-walls) during development: when the code is ready for production use, these services are inlined in libraries. Another is to use multiple protection domains within the same address space to prevent wild writes to sensitive state. A very interesting possibility is to map critical data to arbitrary locations in the address space: in a large address space it is unlikely that a wild write could find such state. In a sense, the virtual address is a capability [110]. Finally, the flippant answer is, of course, that applications should be written in a type-safe language.

In conclusion, there are any number of methods to protect critical state from buggy applications. At a practical level, we do not expect this problem to be a difficult one to solve.

How does an exokernel optimize global system performance? We do not believe that effective global optimizations require centralized OS resource management. Section 2.8 discusses this issue in more detail.

Isn't all this code motion really just a shell game? No. Application-level code can be simpler than operating system code because it can be specialized and because it does not need to deal with the difficulties of supervisor mode. Additionally, as micro-kernel proponents also argue, user-level development tools are typically much more sophisticated and effective than those available for kernel development. Finally, modification of application-level code should not require high privileges or degrees of trust. Therefore, many more maintainers are available than for traditional operating system code.

A closely related question is how multiple specialization of a general purpose library can be easily maintained. As with any large system, modularity is critical; object-oriented techniques such as inheritance, sub-typing and overloading can also be used. We do not believe that this area brings up problems unique to operating system code.

1.6 Thesis Overview

This thesis is laid out as follows: Chapter 2 discusses general design decisions that must be made in an exokernel and the specific decisions we have made in our implementation. Chapter 3 details our prototype exokernel and library operating system. Chapter 4 provides concrete examples of *why* operating system extensibility is worthwhile and demonstrates the efficacy of the exokernel approach through a number of experiments. We discuss related work in Chapter 5 and conclude in Chapter 6.

Before closing, we note that Chapter 2 is not a lightweight chapter: readers who find its level of discussion tedious or who are still ambivalent about the exokernel methodology are encouraged to skip ahead to Chapter 4, where we show the large performance gains that can be obtained by an exokernel-based system.

1.7 Summary

Traditional operating systems have suffered from poor reliability, poor adaptability, poor performance and poor flexibility. Furthermore, operating system complexity (and its attendant problems) will be exacerbated as OS designers attempt to provide increasingly general purpose virtual machines (e.g., through the inclusion of multiple standards within the same OS implementation). More importantly, application performance and flexibility have been directly harmed by by these problems, and the ambition of designers has been sharply curtailed by their limited ability to manage resources. We believe that many operating system problems can be solved by lowering the operating system interface: namely, by exporting physical resources to applications directly, management and abstraction of these resources can then be specialized for simplicity, efficiency, and appropriateness.

The exokernel philosophy has at its core two simple observations:

1. There is no "best way" to implement a high-level abstraction: hard-wiring the implementation in the operating system, where it can neither be changed nor avoided is, therefore, undesirable.

2. The most reliable, extensible, and efficient kernel primitive is the one that is not there. By pushing abstractions out to application space, they can be crafted for specific application domains, which will increase both their efficiency and appropriateness.

Both of these intuitions motivate migrating the implementation and specification of high-level abstractions into library- and server-based operating systems.

The exokernel's central tenet can be viewed in the context of the tension between distributed and centralized control: we believe that distributed control is the most effective mechanism for flexible, efficient systems.

Chapter 2

Exokernel Design Issues

An exokernel attempts to eliminate all abstractions and, as a result, all resource-specific policies from the kernel. However, since it multiplexes the hardware, some policy decisions cannot be avoided: for example, how resources are named, allocated, revoked, and protected. This chapter examines the specific set of policies an exokernel must include, and reconciles them with the overarching goal of exposing the full complement of resources necessary for application-level management. We provide a set of principles that can guide the determination of how to implement/design each policy in such a way as to give the most control to applications.

This methodology can be put in a general framework of distributed control: traditional operating systems have a very strong centralization of control (i.e., the operating system controls most resources), while an exokernel attempts to move most control to applications. This design space is markedly different from the one that has traditionally been explored in both the research literature and in practical implementations; we touch upon its important points. While our discussion of multiplexing assumes an exokernel as context, the issues we examine and the design choices we make are applicable to traditional operating systems as well, and can be incrementally incorporated into a traditional structure.

This chapter has two parts. The first deals with the general principles that an exokernel is built upon. These principles are stated in Section 2.1 and elucidated in the following sections: Section 2.2 examines the tensions between distributed and centralized control; Section 2.3 discusses resource allocation; Section 2.4 discusses resource revocation; Section 2.5 examines naming issues; Section 2.6 some protection considerations. Finally, device multiplexing is considered in Section 2.7. Some of these issues are tightly intertwined (e.g., the representation of resources has a deep impact on how they are revoked, allocated and multiplexed). To aid clarity, however, we discuss these points in isolation. Each of these issues is examined in its own section; these sections have the following structure: the first half deals with the general issues and tradeoffs involved, the second discusses the specific choices made in our exokernel, Aegis.

The second part of the chapter deals with practical considerations to the general principles: when they should be violated, caveats to their use, and challenges they present to system construction. The obvious difficulty with distributed resource management is global optimization. Effective global optimization does not require centralized resource management: our reasoning is presented in Section 2.8. A more subtle difficulty is the complexity that distributed control introduces. We discuss how selective application-level centralization (in the form of servers) can counteract this complexity in Section 2.9. We conclude in

Section 2.10.

The default class of applications discussed in this chapter is competing entities that are each attempting to maximize their performance. For the purposes of this chapter, applications are assumed to be non-malicious and equal in the amount and type of resources they may allocate and manipulate. Extending these conditions is largely orthogonal to an exokernel design; we do not consider it here.

2.1 Design Principles

The fundamental goal of an exokernel is the simple, safe and efficient multiplexing of raw physical resources across applications. The ideal exokernel interface is the actual hardware. Informally, an exokernel attempts to allow applications to control all resources available to traditional operating systems without either additional overhead or restrictions. We have developed a number of simple principles that can be used as guides during this process. This list is not complete and, as our experience grows, will likely be shown to be not wholly correct. However, our initial experience indicates that they form a basis for designing an exokernel system. Adherence to them greatly aids in the methodical elimination of operating system abstractions and, more importantly, in the exportation of all hardware capabilities to applications. Finally, their explicit enumeration allows their violation to be done with clear intent, rather than by accident. These principles are summarized below:

1. Applications should manage all physical resources; exokernels should only manage resources to the extent required by protection (e.g., management of allocation, revocation, and ownership). This is the central principle of an exokernel: all other principles are derived from it. Our motivation is the belief that distributed resource management is the best way to build an efficient, flexible system.

2. Fine-grain allocation of all hardware resources. The more resources that can be allocated, the more effective and ambitious application-level management can become. Therefore, an exokernel attempts to allocate all hardware resources directly to applications. To increase the flexibility of both management and resource sharing, the unit of allocation is fine-grain.

3. Resource interfaces should be low-level. A low-level interface requires the least operating system management and allows the most application-level control. Therefore, an exokernel designer attempts to push the interface that defines hardware resources to the level of the raw hardware.

4. All hardware operations should be exported. Application-level management is aided by access to all operations the hardware provides for a given resource. For example, virtual memory management is aided by access to any DMA capabilities the hardware supports. Therefore, an exokernel attempts to securely export all hardware operations.

5. Resource revocation should be visible. Visible revocation is informative and, more importantly, directly aids lightweight application-level resource management. Therefore, an exokernel should directly query applications during revocation.

6. Exokernel name spaces should be visible. Any name-space that the exokernel manages or creates should be enumerable from application-level. For example, exokernel

"free-lists" should be published so that application-level resource managers can tailor allocation requests to the available resources.

7. Resource names should be physical. The use of physical names exposes the full power of resource names, allows the greatest flexibility in application-level resource management, and the most minimal operating system resource management. Therefore an exokernel should avoid the use of virtual names and the translation and management that it entails.

8. Resources can be requested by name. Physical names encapsulate powerful resource attributes. For example, the distance of a disk block from the current disk head position determines the speed of disk access.

9. Resource virtualization should be used *only* when a resource cannot be multiplexed without it or when efficiency demands it. Resource virtualization consumes resources and requires operating system management: it is anathematic to all exokernel precepts and should be avoided if at all possible. Unfortunately, device multiplexing frequently requires some form of virtualization; this issue is discussed in Section 2.7. In rare cases, efficiency demands that the underlying hardware resources be virtualized. For example, the use of a virtualized TLB to absorb the capacity misses of a hardware TLB; this issue is discussed in Section 3.2.1,

10. Global system optimization should not be done through centralized resource management. Many useful system-wide optimizations that would seem to require a strong, centralized kernel. An exokernel architect must carefully resist the temptation to follow this garden path: as we argue in Section 2.8 many global optimizations can be done directly through an exokernel's control over allocation and device interfaces.

While these precepts are not difficult to understand and may seem quite innocuous, they represent a fundamental change in operating system design. Traditionally, the ability of application-level software to directly manage resources was weak, if it existed at all. An exokernel gives a profound degree of control to application-level managers. We explore the implications of this change in the sections that follow. As a brief setting of context between an exokernel and past systems, the differences are painted with a "broad-brush" in Figure 2-1. While there are many individual exceptions to our characterizations, the general contrasts hold, and can be used to understand how the exokernel structure relates to more traditional architectures.

2.2 Tradeoffs in Distributed and Centralized Control

This section examines the first (and most basic) exokernel principle: that applications should manage all physical resources and that the exokernel should confine itself to solely to what management is required to enforce protection. The premise behind this principle is that *distributed* resource management is the best way to built an efficient, flexible system. This premise (which is partly shared by other "extensible" operating systems [108]) departs from traditional operating systems. Since Chapter 1 has already presented arguments for why high-level operating system abstractions are deleterious, this section concentrates on the tradeoffs between distributed and centralized control. The high-level interface adopted by most operating systems results directly in centralized control, since it *forces* the operating

	Monolithic	**Micro-kernel**	**Virtual Machine**	**Exokernel**
Management	Centralized	Centralized (OS + servers)	Centralized (in emulator)	Distributed
Abstraction-level	High	High to medium	Low (emulated)	Low (real)
Major abstractions	Files, processes, IPC	Files, processes, IPC	Hardware (emulated)	Hardware (real)
Names	None or virtual	None or virtual	Virtual	Physical
Protection	Abstraction & name-based	Abstraction & name-based	Abstraction-based	Explicit access control
Operations	High-level	High-level (medium for trusted servers)	Machine operations (emulated)	Machine operations (real)
Deallocation	Implicit	Mostly implicit	Implicit	Explicit

Figure 2-1: Comparison of exokernels to traditional systems

system to manage resources. The low-level interface adopted by an exokernel results in distributed control, since it pushes resource management into applications.

Centralized control

The tension in the level of resource allocation is the tradeoff between centralized and distributed control: a low-level interface pushes resource management out into application space, where it can be optimized for local conditions while a high-level interface keeps management under centralized control, where it can be optimized for global conditions. Frequently, these two levels are in direct conflict.

Chapter 1 has already detailed many disadvantages of centralized control; in this subsection we look at four potential advantages:

- Less state duplication. Object allocation and control exists through a single centralized entity. This allows book-keeping at one level to be reused by another. With lower-level allocation, the system may have pervasive state duplication, since each entity may not trust the resource tracking implemented by another.

  However, the cost of this space optimization is that there is a single, centralized manager who cannot be circumvented, no matter how inappropriate its decisions. If centralized resource managers always made effective policy decisions then this optimization could be compelling. However, we believe that the inappropriateness and cost of use of operating system abstractions mitigates against such a structure. Besides, in cases where it is possible for a centralized manager to consistently make wise decisions an exokernel certainly allows the use of application-level servers to aggregate code, data and functionality. The important difference is that this aggregation is voluntary rather than by fiat.

- More effective global optimization. The centralized organization forced by high-level abstractions can give the operating system a clear view of the entire system. Optimization requires information; this organization can provide a substantial amount.

  We do not believe that effective global optimization *requires* centralized management.

The information required to optimize resource usage can, typically, be derived from secondary operations (e.g., LRU usage from allocation requests, disk writes, etc.). We provide examples of this in Section 2.8.

- Greater uniformity. Centralized control implies a single and (perhaps) uniform interface. Additionally, applications only have to communicate with a single entity for any allocation request. In a distributed system, many different entities may control resources and their interfaces may be widely disjoint.

  The decision that must be made here is whether interfaces should be encoded in system structure or at a higher level (i.e., in a standard). An exokernel takes the latter view, which allows multiple standards to easily exist on the same system. Furthermore, many of the benefits of single entity systems can be derived from servers; we discuss this in Section 2.9.

- Faster access checks. A subtle point of high-level abstractions is that they contain a large aggregate of *bindings* (e.g., virtual to physical mappings bound in the page-table structure, file pointers, etc.); the presence of these internal binding can be used to speed access checks. For example, in the case of a TLB fault, if the operating system finds the virtual to physical mapping in the page-table it can insert it directly into the mapping hardware: if the page-table is managed by the application, the operating system must check each mapping for validity, adding overhead.

  However, efficient binding checks can be implemented through caches, rather than the heavy-weight mechanism of high-level abstractions: centralized resource management is not a necessary prerequisite for efficient access checks.

In summary, centralization is sometimes effective; in these specific instances an exokernel can derive the same benefits through the use of servers.

Despite the potential benefits, centralization has a cost. As discussed in Chapter 1, the most obvious is the sheer magnitude of code and data to efficiently and correctly implement the book-keeping it requires. Economies of scale do not hold in computer systems: if anything, the opposite is true. For example, a small, tight kernel can used unmapped code and data; as the kernel's role (and size) grows the kernel must map and page itself, which dramatically increases complexity. Because of their generality, centralized mechanisms have been unreliable, inefficient, inflexible and have precluded application-level management. A more subtle problem is that high-level management requires the kernel to pre-emptively make resource management decisions, thereby ensuring that local resource management is not possible and that, as a result, *application-specific* requirements will be met poorly, if at all. For example, both garbage collectors and distributed shared memory systems would benefit from tight integration with virtual memory, trap, and disk hardware.

Distributed control

As argued in Chapter 1, the advantage of distributed control is the flexibility that it allows. This flexibility is due to a single fact: distributed resource management allows multiple resource managers to co-exist and to be replaced and created without special privileges. Many useful abilities fall out from this single property. For example, subsystems can manage resources in application-specific ways, allowing them to be tuned to the "common-case" operations of different domains. Furthermore, resources can be abstracted in fundamentally different ways *on the same system* easily, directly, and without special privileges. This advantage cannot be underestimated: constructing large software systems is difficult; this difficulty can be eased by the application of suitable abstractions. In a centralized implementation abstractions are, for practical purposes, fixed, and thus conflicts between what abstractions are provided and what abstractions are needed cannot be effectively resolved. The operating system is required to multiplex the hardware so that applications can co-exist. If it does this by inflicting painful abstractions, then the resultant applications will not be either very useful or very interesting and thus its central purpose, supporting coexistence, will be unnecessary.

Despite the many advantages of distributed control, there can be very real costs. For example, global optimization can become more difficult, because the operating system does not manage resources. In general, distributed control is costly in situations where the operating system cannot independently make quick decisions about a situation but, because it has seceded control to applications, must instead communicate with them. This communication has an overhead both in the obvious cost in cycles and in more subtle ways (e.g., this communication allows control to escape from the operating system into an application at critical times). We do not believe that this cost outweighs the benefits of flexible, application-level control; the experiments discussed in Chapter 4 tentatively show this assumption to be valid.

Additionally, distributed control requires that ownership and privileges need to be tracked and to be checked more frequently. This overhead can be substantial, however, we show techniques that can effectively reduce it. For example, checking the access rights to a given physical page on a TLB miss can add significant overhead to virtual memory management but can be directly countered through the use of a software TLB to handle hardware TLB capacity misses [9, 49]. We believe that the global effect of the flexibility and efficiency of application-specific resource management will outweigh the micro-costs of its use. Again, the experiments in Chapter 4 provide provisional assurance that this assumption is correct.

Finally, distributed control is inherently more complicated than a single centralized point of control. In Section 2.9 we discuss how the judicious use of application-level servers can capture the useful functionality provided by centralized methods without their heavy-weight cost.

2.3 Resource Allocation

In this section, we examine *which* resources are allocable, the *interface* to allocatable resources, and the *operations* that can be performed on them. More than any other issue, the answers to these questions determine the character of a system, the ambition of the applications and subsystems that can be built from it, and the power available to them.

2.3.1 What to allocate

The premise of an exokernel is that application management is crucial for efficient, flexible systems. Management requires allocation, therefore an exokernel attempts to allow all hardware resources to be allocated: time-slices, context-identifiers, physical pages, disk blocks, etc. Traditionally, application management has been minimal: operating systems have encapsulated resources within abstractions (e.g., disk blocks have been encapsulated within files and physical memory within address-spaces that, in turn, are encapsulated within processes). Even the low-level abstractions of virtual machines have precluded much application level management: since the main goal of the virtual machine is to be *invisible* to an application, it is should not be surprising that application management (which requires communication between application and the operating system) should be weak. Previous operating systems have placed a preponderance of emphasis on abstraction-based resource management and therefore, have avoided constraints in protection, naming, deallocation, and sharing that an exokernel structure must address.

Closely related to what can be allocated is the unit (or *grain*) of allocation. Coarse-grain allocation allows amortization of protection and book-keeping information, but impedes sharing and flexible management. An exokernel allocates space-shared resources at as fine a grain as possible.

2.3.2 Interface level

The level of resource allocation is the dual of "what" can be allocated. Possible levels span a large range, from the high-level abstraction of a UNIX process (which encapsulates the allocation of CPU, physical memory, and access rights in a single entity) to the low-level, fine-grain allocation of a single physical register. The level at which resources are allocated forces a wide variety of design choices, and is a strong determinant of the operating system structure.

Resource allocation should be at the level detailed by the chip specification. If the allocated unit is portable across machines, it is probably too high. Low-level allocation should not be achieved through emulation. The emphasis on a low-level interface is motivated by the fact that low-level allocation is essential for application resource management and directly eliminates many high-level policy decisions and mechanisms. Operating systems have traditionally provided a high-level virtual machine that is portable across architectures (indeed, many successful operating system interfaces have outlived the machines that birthed them [99]). Those kernels that do allow low-level allocation typically only allocate those resources to a server, which in turn manages them (e.g., window managers are typically granted exclusive access to a frame-buffer).

2.3.3 Export all hardware operations

Flexible management requires access to the full capabilities of a system object. Therefore, positing access rights, all operations of the underlying hardware are allowed on an owned resource: all privileged instructions are supported, and all memory operations allowed (e.g., DMA). Traditional operating systems do not export the full set of hardware resources and certainly do not allow the full complement of hardware operations on them.

2.3.4 Virtualization

Somewhat orthogonal to the level of the interface is the extent to which this interface is a *virtualized* facsimile. While virtualized interfaces can indeed be low-level, they have most of the disadvantages of high-level interfaces: the emulation they provide is both complex and expensive to implement and, in all probability, provides applications with no more real control than a high-level kernel interface. For instance, VM/370 provided each user with a virtual machine detailed to the level of devices and privileged instructions [27], consumed significant resources in the process and, as discussed previously, did not allow much more *actual* distributed control than a traditional operating system.

However, there are particular cases where virtualization is either unavoidable or can dramatically improve performance. As discussed in Section 2.7, virtualization must be used to multiplex devices, since ownership of device input/output can typically only be derived with detailed knowledge of data the device manipulates. For example, to determine which applications own which network packets requires knowledge of network protocols such as TCP, UDP, etc. Hard-coding this information in the kernel would decrease simplicity, flexibility and reliability. Virtualization can be used to avoid such a situation.

A specific place in an exokernel where virtualization may improve performance (possibly by a substantial amount) is in increasing the size of the hardware TLB through the use of a software TLB [49]: by virtualizing the TLB to a larger size, capacity misses can be reduced, and with them, the cost of inserting a mapping into the TLB (e.g., system calls and access right checks). We measure the overhead of application-level virtual memory in Section 4.6.7.

In closing, virtualization is used *only* when resource multiplexing could not occur without it or, in extremely rare cases, when efficiency demands it.

2.3.5 Aegis

We examine how Aegis satisfies each of the principles we have enumerated. Aegis allows applications to allocate all resources that it understands: pages, CPU slices, and context-identifiers. In the full system this set of resources will be extended to include network buffers, disk blocks, frame-buffers and the full complement of I/O devices (sound cards, etc.). The issues that arise in device allocation and multiplexing are explored in Section 2.7. Space-shared resources are allocated at as fine a granularity as possible: physical memory is divided into pages, CPU slices are partitioned at the system clock granularity, and context-identifiers are allocated singly.

The CPU representation is unique and deserves a brief discussion. The CPU is viewed as a space-multiplexed device: the CPU is represented as a 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 guarantee of 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 time-slices in order to minimize the overhead of context-switching, while an interactive application could allocate several equidistant time-slices in order to maximize responsiveness.

Some resources are either time-shared, or not associated with owners. Registers are allocated "whole cloth" to each process: on the current architecture allocating specific registers to a given process is more trouble then it is worth. However, on a different architecture this might not be such an obvious decision. For instance, on a SPARC processor [51] allocating individual register windows may be useful. TLB entries and cache blocks do not have explicit owners: they can be written and read by each individual process.

Aegis allows applications to issue all privileged instructions and load/store operations to memory mapped I/O devices. Protection is guaranteed by associating these operations with guards. The ability to issue privileged instructions allows applications to fully exploit the hardware resources. For example, applications can probe the TLB for specific entries or flush it entirely; these abilities allows detailed tracking of "working set" and reference information. Exposing DMA hardware (or emulating it in software) allows applications to perform bulk memory transfers or initializations without polluting either the cache or the TLB. As described in Chapter 4, this detailed level of control is very useful when implementing flexible application-level virtual memory.

There are two places where Aegis virtualizes a resource. The first is the TLB: because access checks would add a large overhead to each TLB miss, Aegis overlays the hardware TLB with a larger software TLB [9, 49] to absorb capacity misses. The second area is the network interface, which requires virtualization in order to multiplex; the specifics of this virtualization are discussed in Section 2.7. We do not anticipate virtualizing other non-device resources.

2.4 Resource Revocation

Once resources have been allocated to processes there must be a way to reclaim them. This section concentrates on the tradeoffs of visible and invisible revocation schemes.

Revocation can either be *visible* or *invisible* to processes. Systems with external pagers [73] use a form of visible revocation for physical memory. Traditionally, operating systems have performed revocation invisibly. For example, with the exception of some external pagers, most operating systems deallocate (and allocate) physical memory without informing applications.

Visible revocation is informative and directly aids lightweight application management (i.e., physical names can be used). Pervasive use of visible revocation is a marked departure from traditional systems, and will dramatically increase applications' ability to manage resources.

2.4.1 Tradeoffs

Visible revocation: there are three good characteristics of visible revocation. First, it allows applications to adjust to changing conditions by providing explicit cues when a resource is in short supply. Secondly, it potentially enables applications to exert fine-grain control over what specific instance of a given resource is deallocated. For example, user-level pagers allow applications to select which particular page to deallocate. Finally, visible revocation allows the operating system to place more responsibility on applications to manage resources and, correspondingly, to eliminate the same management within itself.

There are many possible variants of visible revocation. Each form must make a decision about how binding the revocation interaction is. For instance, if an application does not return a resource promptly, does the operating system kill it or is a more graceful scheme used. This contract (or *abort protocol*) is the main determinant of their character. The reason for this is that the main cost of visible revocation is that it requires, at some point, explicit communication between the operating system, and application, adding both overhead and latency to revocation. A more subtle consideration is the degree to which the operating system's control over the system is reduced: visible revocation can allow control to escape to the application during a resource shortage.

Invisible deallocation: the operating system deallocates resources as it sees fit, without application involvement. Invisible deallocation has lower latency and is simpler than visible deallocation: applications do not have to be queried at revocation time; therefore, there does not need to be an "abort protocol" in place to deal with applications that do not return resources promptly. The disadvantage directly results from this strength: applications have no control over deallocation and, more subtly, no overt cues that resources are scarce.

The three variables that influence which deallocation mechanism to use are *resource activity* (the rate of allocation and deallocation), the amount of state associated with each resource, and the potential impact of application-level knowledge. Visible deallocation works best with static resources that have a large amount of state associated with them. For example, pages are large, infrequently deallocated entities that can profit from visible revocation mechanisms. The performance advantage of reducing page faults easily allows the overhead of explicit communication between application and operating system to be recouped. Invisible deallocation works well for highly active, stateless resources. For example, context-identifiers can be deallocated frequently, and have little state associated with them: reclaiming CIDs simply requires flushing TLB mappings that contain that CID from the TLB; no state needs to be saved. Furthermore, the latency of reclamation is important: blocking a runnable process until another has returned a context-identifier is unacceptable. In this setting, invisible deallocation is likely the correct choice.

Finally, the frequency of revocation directly influences the profitability and implementation of bindings (e.g., the binding of a virtual to physical page number in the TLB). Bindings must be explicitly removed at deallocation time. If revocations are frequent or must happen efficiently, all bindings must be enumerated quickly or system performance will suffer. If there are a large number of bindings, the state required to enumerate them can itself consume significant resources: bindings are not a panacea for system performance.

The abort protocol

Visible revocation protocols interact with applications. If the system needs the resource, the time of this interaction must be bounded. The system's *abort protocol* is used to determine what action to take if this bound has been exceeded.

The strongest constraint on abort protocols is that they encode a notion of time. Since programmers do not reason effectively about time bounds, some care must be taken in the design of an appropriate protocol. This constraint aside, it is possible to implement simple, effective abort strategies. We discuss some possibilities below.

Hint based. Revocation is viewed as dialogue between application and operating system in order for the application to suggest specific resources to deallocate. If the application does not give a hint in a timely manner, the operating system makes its own determination. It then gives the application an exception (e.g., "revocation time exceeded") with the name of the resource that was deallocated. This exception allows applications to relocate any mappings that use the physical name associated with the resource. Other than the obvious need to allot enough time for applications to "usually" make a deallocation decision, there is the additional complication of ensuring that in the case where the kernel has to deallocate a resource, a critical instance of that resource is not deallocated (e.g., the page holding TLB exception code). A partial solution is to allow applications to mark resource instances as non-deallocatable. For example, this is done by the Cache Kernel to pin virtual memory mappings [22]. The problem then becomes how these pinned resources can be deallocated. A possible solution is to associate with every application a "swap server" that acts as a guardian. If the operating system can find no unpinned resource to deallocate it notifies the swap server, which then swaps out the application and frees some or all of its associated resources.

Fascist. From the kernel's point of view, a simpler mechanism than hints is to simply bound the time required to deallocate a resource. Applications are then responsible for organizing resource lists in such a way that resources can be deallocated quickly. For example, an application could have a simple vector of physical pages that it owns (pinned in main memory): when the kernel indicates that a page should be deallocated, the application simply selects one of these pages, writes it to disk, and frees it. Such an organization is not difficult to construct and, if the kernel is made aware of certain "good faith" operations (e.g., the writing of a page to disk in preparation for deallocation) it should not be an unreasonable alternative. This method should be implemented with at least two stages of revocation notifications, in order to allow the low-level allocation mechanisms to interact with higher-level application software. For example, if revocation occurs in two stages ("please return a page" and "return a page in 50 micro-seconds") then the low-level paging software can do an upcall to higher-levels on the first notification. If this higher-level does not respond by the second notification, the page-manager simply deallocates the resource directly.

2.4.2 Aegis

Aegis uses visible deallocation for CPU slices, pages and the system objects that it controls. Context-identifiers are reclaimed "invisibly" from applications.

Revocation is done when resource usage exceeds a specified threshold. Which instance of the resource to revoke and who to revoke it from are highly system dependent. For example, on a system with proportional sharing [105], this determination will be made differently than on a "fair-share" system [46, 55]. As discussed later, Aegis will allow this to be parameterized on a system-wide basis. We discuss the default mechanism below.

As devices are added, revocation will be done at two levels: the first is a request for the given object, the second level is a demand. The abort protocol follows the "hint" protocol described above. Resources are forcibly flushed to a backing store (similar to the Caching Kernel [22]), and the application is given an exception that its revocation time was exceeded. The location of the backing store will be included as an argument to the exception.

Initially, random selection will be used to select which application to revoke a resource from. Each active resource will be associated with its owner; to revoke a resource, Aegis will randomly select the owner from this vector, and send the application an exception to indicate that it should relinquish a resource of the indicated type. A critical point is that it is the application decides *which* instance of the resource to deallocate. Randomized revocation has very good properties for equal sharing. Introducing a proportional-share structure on top of it is a matter of scaling this selection so that some entities are more likely to be selected than others [105]. We do not consider these issues in this thesis.

2.5 Resource Naming

The representation of names determines both how heavyweight the OS management must be and the effectiveness of application-level resource management. An exokernel attempts to use physical names wherever possible. This is a fundamental difference between it and previous operating systems. With few rare exceptions (e.g., external pagers), physical names have not been used in any other multiprogrammed operating system structure; certainly no such operating system has placed the importance on them that an exokernel does.

We discuss the tradeoffs between virtual and physical names. For a more extensive and general discussion of naming see Saltzer [85]. The three issues we look at are the impact of the naming scheme on: (1) revocation (how "relocation" meshes with a sensible revocation mechanism), (2) sharing (how names can be translated across contexts), and (3) allocation (what can be named, and if the full information encoded in a name can be exploited).

2.5.1 Name spaces

Name-spaces that are managed or created by the exokernel should have an application-level "enumerate" operator. Allocation of specific resources is aided by an accurate picture of what resources are in use. Therefore, publishing this book-keeping information (e.g., free-list, estimation of disk head position, etc.) allows more effective allocation to be provided. Furthermore, exokernel data structures can provide crucial information as to the resource utilization of the system. For example, the mappings in a software TLB can give an approximation of the current active set of physical pages. This principle can be viewed as simply exposing all bookkeeping primitives to applications.

Traditional operating systems are markedly reticent about supplying such information, which makes it difficult to get a clear picture of what resources are in use at any point in time. Part of this fact is simply because applications on these systems cannot typically manage resources, and so would not be able to make much use of such information. The other part is that information hiding is a very effective way to decouple the OS implementation from its specification. We believe that the cost of decreased application-level management outweighs the benefit of such information hiding.

2.5.2 Physical versus virtual names

The naming issues we discuss here have been made in the specific context of names managed by the operating system and seen by the application. Naming can be done using the actual name of an object (physical naming) or via a layer of indirection (virtual naming). Efficient, reliable translation of virtual names is difficult, and can consume significant resources (e.g., virtual memory systems). Most resources cannot be named in traditional operating systems; those that can do so through virtual naming (e.g., virtual addresses, "logical" disk blocks, etc.). The challenge in the use of physical names is relocation and, as a direct consequence, revocation.

Physical naming

Physical naming is WYSIWYG: page 5 is page 5, disk block 137 is disk block 137. In such a scheme, processes access and locate objects using their actual names. The chief advantage of physical naming is that it requires no translation, and therefore is lightweight and easy to implement. Its main disadvantage is the constraints it places on revocation: at either use or revocation time, the process itself must be notified of revocation, so that it can *relocate* its names. For example, a process using physical page "5" in virtual mappings will have this information scattered throughout its address space, which removes any possibility of the kernel relocating it. This issue can be viewed as who controls the "context" in which names are resolved: control over the context determines who can sensibly alter it.

If revocation happens visibly, then relocation does not pose additional problems: the operating system will explicitly communicate with applications at revocation time anyway, so relocation adds no new constraints. However, there exist situations where physical naming should be used but deallocation is not explicit. For instance, a graceful abort protocol may deallocate a resource "by force" if a process exceeds the time-limit it had to deallocate a resource. Unfortunately, in the presence of physical names, this resource is still named throughout the application's context: nothing good can come of this. To mesh physical names with invisible deallocation an auxiliary data structure, a *relocation vector*, must be introduced. When a resource is deallocated by the operating system, this fact is registered in the vector and the application receives a "relocation" exception (to allow it to update any mappings that uses the resource). Since this situation will not arise frequently, the vector can be small; furthermore, since lookup does not have to be efficient, its structure can be simple as well.

Virtual naming

A virtual naming scheme adds a level of indirection between a name used by an application and the real, physical name of the resource. A familiar example of this is virtual memory. This level of indirection (called a *context* [85]) can be per-process, per-system, or any range in between. The primary advantage of virtual naming springs from the transparent relocation it allows. For example, if an operating system controls the mappings of virtual to physical names, it does not have to explicitly communicate with applications on revocation.

Virtual naming's main costs are the resources consumed by the mapping structure and the overhead of translating virtual to physical names. If translation does not have to be efficient, it can be made quite simple (e.g., a vector of tuples). Typically, however, translation must be efficient, which leads to complex mapping structures (e.g., page-table-based virtual memory systems are notoriously complicated). Furthermore, the state required to perform this mapping can be quite large. Some resources, on the other hand, are easily associated with virtual names. For instance, context-identifiers are small in number, and thus require few resources to map efficiently.

The use of virtual naming allows some operating system optimizations that are otherwise difficult. For instance, disk compaction is very easy with the use of virtual names, since applications and higher-level filesystems do not need to be involved in the process [30]. However, the use of such a "logical disk" to map disk blocks can require over 4 megabytes of physical memory for disks of even moderate size [30].

Summary

The cost of virtual naming is the associated loss of information and the computation required to map the virtual name to the corresponding physical name. Furthermore, its strength, transparent revocation, is often irrelevant: application-level resource managers will *want* to be notified when revocation happens (e.g., the deallocation of a physical page).

An exokernel attempts to use physical names whenever possible. Because the use of physical names at application-level eliminates the need for kernel name management (e.g., translation and relocation), we believe that it is the cornerstone of light-weight application resource management.

2.5.3 Allocation by name

Names encapsulate useful resource attributes. For example, the position of a disk block in relation to the current position of the disk arm determines how quickly it can be written. Allocating a a disk block by name allows a more efficient file-system to be constructed. Furthermore, names have power. For instance, in a system with direct mapped caches, the name of the physical page (i.e., the page-number) determines where it maps in the cache. Two pages with names that fall in the same "page-color" equivalence class will contend for the same regions of the cache [83]. If applications can request specific physical pages, they can minimize their conflict between already owned pages and between the pages allocated elsewhere in the system. Traditionally, operating systems have not provided any support for direct request of resources by name (e.g., physical page "42").

At first glance, useful name-based allocation appears to require substantial machinery. For example, to return to our disk arm example, applications may want to phrase questions such as "allocate a block close to sector 10 or 20, whichever is less fragmented." Obviously, such queries can get arbitrarily complicated and baroque. Fortunately, a complex query engine is completely avoidable: if the kernel publishes the data it would use to satisfy such requests, applications can satisfy extremely detailed and volatile predicates without any operating system interaction. For instance, if the operating system publishes the disk block "free list" and estimation of disk arm position, then applications decide which disk block fits their requirements. This functionality is extremely simple to provide and will likely exceed the sophistication of query mechanisms provided by the kernel. The one drawback of such a method is that it is not atomic: between the time an application-level OS scans a freelist and requests a resource, the resource could have been allocated. However, such a situation will probably be very infrequent.

2.5.4 Aegis

Aegis attempts to use physical names whenever possible. Physical pages are named explicitly, as are time-slices and TLB entries. Aegis allows resources to be requested by name; as suggested above, this facility is made more useful by exposing the current free-list to applications, which gives them a high probability of success in their allocation requests. Resources that are already allocated will not be revoked to satisfy such requests.

As discussed above, physical names and name-driven allocation enable a large number of optimizations. For instance, applications may want to construct runs of contiguous physical pages to allow larger page sizes to be mapped in the TLB or to allow larger DMA transfers.

Context-identifier names are virtual. Because our current machines only support a small number of identifiers (64), frequent revocations can occur. Additionally, context identifiers must be returned promptly on revocation if the system is to make progress or if a "graceful" deallocation policy is to occur with any regularity.

2.6 Protection

Because an exokernel explicitly allocates physical resources and attempts to move all management of these resources to the application, protection assumes a more explicit and central role than in a traditional high-level operating system. However, the issues we discuss here are hardly unique to exokernels: we discuss them because they are so central, not because of unique tradeoffs. Protection is a well-known area: our discussion is based mainly on the literature. We elide many details; for a general discussion, consult Saltzer [86].

2.6.1 Tradeoffs

At a high-level, there are two protection schemes we consider: access control lists and capabilities [86]. Both associate guards with resources: when principals attempt to use a resource (or encode a binding involving it) these guards check access rights. ACLs are simply lists of principals allowed to use a given resource. ACLs cannot be circumvented: the kernel has absolute control over access rights. Furthermore, the exact list of principals that can access a resource can be controlled completely. This strength is also a weakness: sharing must involve the kernel. In contrast, capabilities are a ticket-based approach, where possession of a ticket is taken as *prima facie* evidence that the principal has the right to use the resource. There are numerous capability schemes. We only consider *self-authenticating* capabilities, which are large numbers that are unlikely to be guessed [21]. In other words, security is achieved through obscurity. Self-authenticating capabilities allow ease of sharing and requires only simple protection machinery. Sharing is easy, since capabilities can be passed as data between principals without kernel intervention. Protection is simple to implement since the kernel does not need to track who has access to what resources: the kernel simply has to check that each use of a resource is by a principal with the appropriate ticket (i.e., with an if-statement).

2.6.2 Aegis

Aegis uses capabilities for protection. Pages, time-slices and environments are associated with capabilities. Every point where a resource can be used is associated with a guard. When a principal wishes to use a resource, it presents the capability to the guard. The guard checks the access rights given by the capability. If the use is allowed, the operation is allowed to proceed. Otherwise an exception is given to the client. Each resource is associated with a read and a write capability; additionally, each allocated resource has an owner that may free it (or change its capabilities). Environments are defined hierarchically (i.e., an environment that allocates another owns it and all of its associated resources). Capabilities are selected by the applications themselves, allowing space-efficient encodings (e.g., all pages owned by a particular process may be associated with the same capability value) and a "public encryption" style of sharing (e.g., two resources may agree a priori what capabilities to use for resources; once these resources are allocated, they "publish" the names and use them).

At a practical level, the merits of capabilities over ACLs for general-purpose protection are unclear. There were two advantages of capabilities that led us to use them over ACLs. The first advantage capabilities have over ACLs is that with capabilities the kernel's storage of access rights does not increase proportional to sharing: with ACLs, every additional principal requires more bits to track. However, this advantage may be illusory: principals certainly have to track ownership and so, in practice, total space requirements will still grow with sharing. While this space requirement can be countered by having one capability control more than one instance of a resource, such an approach is clumsy. The second advantage capabilities have over ACLs is the ability to share resources without kernel intervention. However, on a real system with non-trusting applications, an application will not want to "invisibly" share a resource with another, since it could not prevent the owner of the resource from freeing it "unexpectedly." To prevent such occurrences requires some form of reference counting, which will amount to an ACL. Finally, capabilities have a practical disadvantage in terms of bookkeeping and interfaces: tracking large numbers and passing them to system routines can be clumsy.

In the future, we will likely use an ACL scheme modeled on Lampson's "bitstrings" [62] to protect most resources (a likely exception will be disk blocks). This approach has been recently used both in SPACE [79] and in the Cache Kernel [22].

2.7 Device Multiplexing

Devices are difficult to multiplex: their interfaces are complex and their state extremely volatile. For example, device buffer usage can fluctuate dramatically within short amounts of time. Both of these characteristics make time-sharing impractical (context-switching a large amount of device state is expensive) and space-sharing challenging (the volatility of the device's state makes the static divisions required by space-sharing difficult and expensive to implement). However, these implementation issues aside, the main difficulty in device multiplexing is that associating the input/output of a device with an owner (the basic requirement for multiplexing) requires a large amount of application-specific knowledge. For example, if the kernel is to divide the frame-buffer in a reasonable manner it must have a fairly precise notion of "windows," "exposure events," etc. Another example is the network: demultiplexing a string of bits coming in off of the wire requires a detailed knowledge of network protocols such as TCP, UDP, etc.

In general, the demultiplexing problem can be solved a number of ways. The first is to simply encode the required knowledge in a server or the kernel itself. In some cases this can be perfectly acceptable; usually, however, this encoding significantly reduces the flexibility of the system. A more flexible mechanism uses *secure bindings* to associate resources with owners.

A secure bindings separates resource protection from management. A key property is that domain-specific knowledge is only required to setup the binding, not to maintain it. The ability to implement protection without an understanding of the underlying resource semantics is extremely powerful. For our purposes, the most interesting feature is that application-level resource management can be implemented with great freedom. To clarify this admittedly vague discussion, consider virtual memory management. Traditional virtual memory implementations serve the two functions of protection and management. An OS can directly cleave these functions by allowing applications to generate address translations and then simply ensuring that a given translation is allowed. By restricting its attention to the secure binding of address mappings, an operating system allows applications to implement their own page-tables and have complete control over their address space structures. For more complicated resources (i.e., devices) a common implementation grants a set of servers the right to bind applications to a particular resource; once this binding is in place, the kernel (or hardware) can check access rights. We look at exactly how this can be realized below. The important point here is that access control is the only function of the server: the application has complete discretion in management and usage of the resource, once allocated. Separating protection from management in this manner allows applications to have a large degree of freedom in resource management. The utility of this idea is large.

2.7.1 Frame-buffer

Encoding detailed knowledge of a windowing system in the kernel is not desirable. There are a number of methods used to avoid this problem. The most common solution is to memory-map the device buffer into user-space where a server can manipulate it directly. Unfortunately, this still requires that applications go through a centralized server to access the frame-buffer. An intriguing solution taken by Silicon Graphics can be viewed as an instance of secure bindings. The hardware associates each pixel with a context identifier; a process that has the same context-identifier as the pixel can write directly to its memory. This process is both flexible and efficient. The only system involvement is at binding setup, when a server sets the context identifier of the required pixels to that of the process [1]. This same technique is also used for the sound port.

The disadvantage to these approaches is that their implementation consumes a large amount of silicon. A more "low-tech" approach is to allow a window server process to download pieces of code into the kernel; these pieces can be called by applications to perform operations that they require. Protection can be ensured by bounding memory operations and jumps [104] and by limiting resource consumption [31] (e.g., CPU time and memory). The window server can use this capability to allow the applications themselves to write code fragments (say to implement a particular clipping algorithm) that it can check for safety and then download into the kernel. This functionality can be used to give applications a wide degree of freedom in what operations and optimizations they can perform.

2.7.2 Network

There are two problems in multiplexing a networking device: sending and receiving. Sending is simpler; we consider it first.

Multiplexing the send capabilities of the network device is simply a matter of multiplexing send buffers. With simple hardware support, this is fairly easy to achieve. For example, one solution (suggested by Druschel [34]) is to map a send buffer into the address space of each process. These buffers can be allocated/shared/revoked in a manner analogous to physical memory.

Demultiplexing messages is a more difficult problem, since it requires associating a string of bits with protocol-specific meaning. If backwards compatibility is not a constraint, this is one area where the complete knowledge of the domain can be encoded easily in the kernel: each message can be tagged with a process identifier; the kernel could directly vector incoming messages to the associated process. This approach is used in [53]. Unfortunately, networking has tremendous backwards compatibility constraints. An exuberant alternative is to treat every process as a connection point and give it its own Ethernet address [61]. This allows simple vectoring and requires no protocol changes. Unfortunately there are reasons why this cannot, in general, be done with impunity.

The traditional compromise between flexibility and backwards compatibility has been to allow applications to demultiplex the messages themselves through packet filters [74]. Packet filters are predicates written in a small type-safe language. Logically, packet filters examine all incoming network packets; those messages that satisfy their predicate are delivered to the filter's associated application. The obvious problem with this structure is ensuring that a filter does not "lie" and accept packets destined for another process. Traditionally, this problem is eliminated in two ways. First, each filter is associated with a priority, such that filters with lower-priority cannot intercept messages destined for higher-priority filters. Second, only trusted servers can install a packet-filter (this is another example of a secure binding). Unfortunately, since applications are precluded from installing packet filters directly, this structure limits flexibility. For instance, applications are limited to those protocols known to the server, which removes their ability to experiment with novel protocols.

Fortunately, we have a solution to this situation that is built upon the simple observation that most applications do not care if other applications on the same node can view their

[1] In practice, there are more contexts than context identifiers, so auxiliary techniques are used to enforce ownership.

messages (either because the messages are encrypted or because they are uninteresting). Therefore, as long as other applications cannot consume a message intended for another, it does not matter if a message is read. We split packet filters into those that care about secrecy (i.e., those that consume their messages) and those that do not (i.e., those that do not consume their messages). The first grouping has side-effects, the second is referentially transparent. A filter that consumes its message can only be installed through trusted servers; any packet that it accepts cannot be claimed by any other filter. However, if no such filter accepts the packet, then the packet filter engine checks with each non-consuming filter to see if it desires the packet: all those that do are given it.

This simple distinction allows applications that do not rely on packet filter measures for security to directly install packet filters for *any* protocol they desire. We believe this will allow nearly all applications to experiment at will: it is very rare for applications to rely on packet filter measures, as opposed to encryption, for security. Furthermore, given the ability of other nodes to "snoop" the wire, these applications likely have faulty security assumptions in any event. In the subsection we discuss some practical concerns on making this process efficient.

2.7.3 Aegis specifics

We only look at packet-filter issues here. Since each filter must be applied to each message, computation rises linearly with the number of packet-filters installed. Fortunately, there are very effective optimizations that can counter the cost of using packet-filters. We look at both intra-filter [39] and inter-filter [109] optimizations.

The most effective intra-filter optimization is to dynamically generate the machine code corresponding to the filter. Our prototype implementation shows that compiled packet filters execute an order of magnitude more efficiently than when interpreted [39]. Thekkath suggests this optimization in [98], but does not implement it.

Inter-filter optimizations revolve around the recognition of commonalities between filters: many filters look for similar packets (e.g., TCP, UDP, etc.) and therefore have large overlap [109]. These commonalities can be used to collapse similar predicates, allowing many filters to be pruned for each failed condition.

Unfortunately, recognition of overlaps has been hampered by the low-level nature of current packet filter languages, which are typically pseudo-assembly languages. This low-level representation allows a rudimentary interpreter to run with acceptable efficiency, but destroys high-level information that could be used by a sophisticated compiler. This can be seen in the current crop of inter-filter optimizers, which requires that filters be textually equal from their initial statement onward in order to be grouped in equivalence classes with other, similar packet filters (indeed, filters reversed in even a *single* instruction are not recognized as equivalent [16]). Such a constraint is severely limiting on an exokernel system that allows multiple applications to write and install their own packets. In effect, it would require each filter writer to be aware of all other packet filters on the system in order to write the filter in some "canonical" form. We have solved this problem by recognizing that the over-specification of current filters is directly due to their low-level imperative nature. Our solution is to use a medium-level declarative language (`dpf`) that does not impose an ordering on its statements. Lack of statement ordering enables aggressive inter-filter optimizations, since functional equivalence is no longer obscured by structural artifacts. `dpf` is not the first declarative language for packet-filters. However, the other declarative packet filter language that we know of (described in Bailey et al. [8]) does not utilize the power of declarative specifications: predicates are not re-arranged to increase commonalities and similar inter-filter "suffixes" are not utilized [77].

Once a destination process is found for a packet, the problem then becomes where to put the message. We will allow packet filters to determine destination by adding support for computation within the packet filter itself (this portion of the filter can be viewed as an *active message* [102]). This ability directly eliminates the copying costs associated with current filter implementations. (Concurrently with our work, the SPIN project is adding active message support to the packet filter language MPF [16].)

2.8 Global Optimization

One of the most common concerns about an exokernel structure is that it disallows effective global optimization. We address this concern by showing how the requirements of global optimization can be separated from resource management for three of the most common global optimizations done by traditional operating systems: tracking initialized pages, LRU page replacement, and disk block allocation.

We believe that global system optimization does not require centralized resource management. Of all the exokernel principles, this is the most controversial. Given the lack of a mature system it is difficult to verify; we briefly discuss three situations where effective global optimization can be realized without resorting to centralized resource management.

2.8.1 Simple global optimizations

Many effective global optimizations are quite simple, and can be added directly to an exokernel. An example is the tracking of zeroed pages. On systems with non-trusting applications, it is important that no other application can read the "garbage" of another. Additionally, in many situations, applications desire zero-initialized pages. Without operating system support these two constraints will result in double initializations: the first process will clear its pages of sensitive information and the second with pessimistically set its pages to zero. However, an operating system can easily track which pages are already initialized and allocate these to specific requests. An exokernel can track information about which pages are zeroed as easily as a traditional operating system, and so does not make this optimization any more difficult.

2.8.2 Working set derivation

A more challenging problem is how to derive the working set of various applications so that page allocation can be spread equitably among them. This is a straight-forward operation on a traditional operating system: it has access to all reference bits and can determine whether page deallocation causes an increase in paging activity.

An exokernel has no intrinsic knowledge about the requirements and uses of any application. Fortunately, derivations of working sets do not require such knowledge. Reference information can be garnered by monitoring TLB insertions. The fact that page-revocation induces paging can be derived by monitoring for increased physical-page allocation requests. The latter action captures the information given by monitoring paging in a traditional operating system. Indiscriminate resource requests can be actively discouraged by suspending applications that request a resource that is in short supply.

2.8.3 Disk utilization

Disk arm latency is the crucial bottleneck in a disk system. An effective global policy, then, attempts to ensure that the disk arm does not need to move frequently. This can be done by allocating (or migrating) frequently accessed blocks along a narrow band (which lowers seek time to requested blocks) and through the aggressive use of file-caching.

The difference in file-caching under an exokernel system is that the applications themselves manage these caches: whether this is done through proxy servers or directly is of little concern — the characteristics and effectiveness of this file-buffer caching should not suffer.

The only question, then, is how to ensure that the disk can form "hot sectors" that minimize disk arm seeks. The first step in this has already been discussed: the disk block free list and an approximation of the disk arm position is published, which allows disk managers to explicitly allocate blocks close to the current disk arm position. The problem then becomes how to multiplex this scarce resource in the face of heated contention. This can be done in a direct manner by bounding the number of disk blocks applications can allocate in hot regions.

However, we expect that in the common case application-level disk managers will approach good global utilization because of their sophistication. In the case of the limitation of "good" disk blocks, disk managers will know that these blocks are very critical. Furthermore, they will realize that if they allocate too many of them the disk arm will have to seek to another sector and therefore, the disk blocks that were so valuable before will become less so. We believe all of these factors will encourage disk managers to wisely manage their consumption of blocks in this very attractive territory.

Finally, the growing sophistication of disks will make them relatively impervious to any form of disk management (or mis-management) [57]; as such, many of the mechanisms we discuss will become irrelevant for effective disk performance.

2.9 Some Practical Considerations

As discussed in Section 2.2 centralization can have a simplifying effect on the system as a whole. This section examines two areas where this simplicity can be recaptured without forfeiting the flexibility of a distributed structure. The first is structuring an exokernel-based system so that application-level operating system implementations are decoupled from each other. The second is the very strong role servers can play in implementing operating system abstractions.

2.9.1 Decoupled implementation

Operationally, traditional operating systems can be viewed as shared, dynamically linked libraries: as long as agreed upon integers are used for system calls, applications are effectively decoupled from the actual operating system implementation. This decoupling is crucial in terms of space: each application does not require its own copy of operating system text. Furthermore, this decoupling is critical for backwards compatibility since it allows the operating system to be developed in a totally separate environment from applications: to use a new operating system implementation, applications do not have to be re-linked (e.g., binary compatibility can be achieved by radically different operating system implementations [24]).

There are a number of ways that exokernel-based systems can realize these same benefits.

The first is that a library operating system can use system call forwarding to communicate with an application strictly through a system call interface. Servers can also be used as a primitive form of dynamic linking: each server represents aggregates of code and data shared among multiple processes, and can be accessed through well-known IPC ports. Finally, a more customized approach is to use real dynamic linking, similar to what some modern object-oriented systems implement.

While the methods we have listed entail more sophistication than the traditional interaction between operating systems and applications, we believe that this complication can be minimized.

2.9.2 The role of servers

In this subsection we look at three important functions that have been performed by operating systems: the breaking of self-referential loops in applications (e.g., an application cannot "swap itself back in"), atomic actions and the management of shared, fault-isolated caches. An exokernel-based system no longer has a single, centralized operating system that knows the requirements and abstractions of the applications that utilize it; instead these functions will have to be performed by application-level servers. Because these servers will be critical, we briefly discuss the general requirements of these functions.

Self-referential loops

Self-referential loops occur when a process requires that an operation be performed on it "from the outside": these are, typically, boot-strapping requirements. For example, for a process to be "swapped in" requires the intercession of a third party that can set up page-tables, etc. Similarly, an application requires help to "wake up" after suspension. Servers can be used to perform these functions.

Atomicity

Implementation of some operations is aided by atomicity guarantees. For example, it can be complicated for a running application to directly restructure its page-table, since pages can be swapped out, mappings can change, etc. during the process. A simpler methodology is for the application to be suspended by a third party who restructures the page-table, and then unsuspend it. Again, this action can be done by servers.

Cache and binding management

Many of the issues discussed in this subsection have arisen previously in micro-kernel systems. We review some of the more subtle issues.

While applications can manage shared state directly, such an approach has poor fault-isolation characteristics and, more to the point, is not viable between applications that do not trust one another. A solution is to use a shared, fault-isolated cache that is managed by a trusted (or at least highly accountable) entity. The uses of servers to manage caches (and the bindings to them) are significantly more subtle than the simple use of boot-strapping. There are roughly five properties that a server can ensure: that all accesses to it have the appropriate access rights, that the cache is fault-isolated, that operations on the shared state are well-formed, that shared state is kept coherent, and that appropriate bindings are formed and managed. We consider each in turn.

The most obvious role of a server in cache management is that of guardian: the server ensures that all operations to the shared state are by applications with appropriate credentials.

The shared cache can be viewed as an object whose interface is guaranteed by its encapsulation in the server. This fault-isolation can take two forms. The first is simply the use of address-space protection to prevent wild application writes. The second is the use of the server's interface to ensure that all operations on this state are well-formed: the server performs all operations and, therefore, applications do not have to be trusted to acquire appropriate locks, update book-keeping data structures, etc. A nice property of this model is that it maps directly onto the operating system's traditional role as a trusted manager [71]. Typically, servers are used to guarantee the correctness of operations to shared state when a logical structure has been imposed on physical memory; this abstract structure cannot be described (and hence, enforced) through the virtual memory system and so must be synthesized at a "language level." For example, database managers ensure that all operations to the database are well formed.

The role of coherence manager occurs when multiple applications cache certain objects and require a trusted third party to manage the updates between them. For example, if two applications concurrently read, modify and write-back disk block 4, one of their changes will be overwritten. This calamity can be prevented by either merging the writes or by granting exclusive access to the disk block. Either method can be safely performed by a server, which can manage a cache of file-buffers, ensuring coherence, protection and fault-isolation [70]. The important property of this configuration is that applications do not have to trust each other to manage objects correctly. Furthermore, boot-strapping is simplified: for example, with a centralized disk block cache, a reader of a disk block does not have to ferret out all writers to that disk block.

Binding and the management of bindings is also an important server role. Bindings have to do with the granting of access rights to shared state: given appropriate credentials, a server will form a *binding* between the application and a resource (e.g., perhaps through memory mappings). Management of bindings involve tracking access rights and the changing conditions of the cache. For example, a binding formed to a resource with only a single reader may need to be changed when a writer is added; or a binding may change because the underlying physical resource involved changes. For instance, a binding of a logical to a physical disk block may need to be updated if the information in the physical block is moved during disk compaction. There are many examples of servers as managers of bindings. For example, Maeda discusses a situation where a server manages a cache of file-buffers [70]. If access rights are allowed, this server binds the file-buffer to the application (i.e., maps the buffer into the processes address space); as conditions change, the server modifies this binding: if multiple readers and writers are manipulating the file the coherency requirements are different than in the case of a single reader. A more simple-minded binding server could be a text cache manager that dynamically links processes with the object code that they require.

In closing, the reader should note that, paradoxically, the addition of servers to a system can dramatically decrease the complexity of that system. Decreasing complexity through the *addition* of entities is very unusual, and deserves some thought. In this case, it results from the centralization servers can have on a system: servers can condense diffuse distributed control into a single, centralized mechanism, with the expected simplification of interaction. The judicious use of servers will allow an exokernel system to approach the simplicity of a centralized system without sacrificing application-level resource management.

2.10 Summary

Exporting hardware to applications involves making tradeoffs in a design space that is radically different from that explored in traditional operating systems. This chapter has discussed a variety of design points in this space and has articulated a set of explicit principles that can guide the design and implementation of an exokernel.

Chapter 3

An Exokernel System

This chapter describes two novel and interesting software systems whose organization is built upon the simple principle of exposing all hardware functionality: *Aegis*, a prototype exokernel, and *ExOS*, a prototype library operating system. Aegis is fundamentally different from *any* operating system, past or present; as such, the implementation details we give should provide the reader with a firm grasp of what an exokernel actually looks like. Furthermore, many of the issues that arise in an exokernel are intriguing in their own right. The most unusual aspect of ExOS is that it manages fundamental operating system abstractions (e.g., virtual memory and process mechanisms) *at application-level*, completely within the address space of the application that is using it. To the best of our knowlege ExOS is one of the first general-purpose library operating system implemented in a multiprogramming environment. (The Cache Kernel is investigating this approach concurrently with our work [22].) For this reason, the implementation techniques we describe should warrent the attention of the reader. Finally, this chapter describes some implementation considerations that, while well known to some designers in the community, have not been recorded: the specific details of user-level traps, design tradeoffs in software TLBs, and exception handling conventions.

This chapter is a "snapshot" of the current implementation, provided to make the ongoing discussion more concrete, not as a a final implementation: the prototype currently makes little provisions for device interrupts and has only rudimentary IPC binding and system call interposition facilities. Furthermore, this implementation is primarily an exploratory one: we have not attempted to make the various pieces fit in a cohesive, unified whole. Rather, we have concentrated on exposing the functionality of the hardware, and providing the basic primitives that are required.

Section 3.1 gives a brief overview of our platform; Section 3.2 describes event handling (both in general and in Aegis), specifically: exception forwarding, interrupt handling, and upcalls. Section 3.3 discusses the internals of the current implementation; it explains the data structures that define each object. Section 3.4 gives the current (rudimentary) system call interface and Section 3.5 the full complement of privileged operations that can be performed on the machine state. Section 3.6 highlights features of our library operating system, ExOS. We conclude in Section 3.7.

3.1 Platform

An exokernel avoids portability: machine independent interfaces hide power, and require resources to implement. As such the architecture for which an exokernel is designed for has a profound impact on its structure and interface.

We base our exokernel on the MIPS architecture [54]; Aegis runs on the DECstation/MIPS family [54]. These machines have separate instruction and data caches, software handled TLB faults, 64 context identifiers and a math coprocessor that controls floating-point operations. The machine configurations that we use are uni-processor machines with physically indexed caches, 31 general-purpose and 32 floating-point application-level registers.

TLB misses and exceptions are handled in software. The MIPS architecture specifies two exception handlers: the first for user-TLB misses, the second for "everything else" (e.g., kernel TLB misses, interrupts, integer overflow, system calls, etc.). The MIPS ABI reserves two registers for kernel use: `k0` and `k1`; it (roughly) divides general-purpose and floating point register sets between caller and callee saved registers. Register zero holds the constant 0.

All privileged instructions are performed by co-processor 0, and most use the instructions `mtc0` (move to co-processor) and `mfc0` (move from co-processor). Branches, loads and co-processor operations have delay slots; the instruction set is a typical RISC.

The MIPS TLB hardware holds 64 entries. Each entry consists of two words: the first contains a virtual page number and address space tag, the second the physical page that it maps and the attributes of the mapping. Mapping attributes control whether the mapping is cached, valid, dirty or "global". Global mappings do not check the context identifier on TLB lookup; because the global attributes allow a TLB entry to match any application's virtual address, it is the only capability Aegis does not export to applications. (This restriction could be lifted if Aegis flushed the TLB when context-switching from processes that used these bits.)

The following special registers will be referenced to in subsequent discussion (all information taken from [54]):

status : the current process status word: supervisor and user mode bits and the status of interrupts and co-processors (enabled or disabled).

tlbhi : contains the context identifier used to match the virtual address with a TLB entry when virtual addresses are presented for translation.

tlblo : holds the low order 32 bits of a TLB entry when performing TLB read and write operations.

tlbcxt : contains a pointer to a kernel virtual page table entry array (i.e., page table pointer); it is used in the TLB refill handler. On all addressing exceptions (except bus errors), this register holds the virtual page number (VPN) from the most recent virtual address for which the translation was invalid.

tlbbad : contains the virtual address that most recently failed to have a valid translation.

epc : holds the exception program counter, i.e., the address where processing resumes after an exception has been serviced.

cause : holds the cause of the last exception or interrupt. General purpose exceptions are vectored through a common exception handler, this register is used to disambiguate which particular exception occured.

To reclaim a context identifier or physical page, all mappings that involve it must be flushed from the TLB. The MIPS hardware does not provide primitives to do these operations, so they must be performed by software. Furthermore, the non-coherent instruction and data caches of the MIPS architecture require that special care be taken when pages used for program text are reused. For example, as is common on machines with split instruction and data caches, initialization of these pages will not affect any instruction cache lines that were derived from them. Therefore, if stale values for these pages still reside in the instruction cache, a cache flush is required to ensure that the current instructions are read.

3.2 Events

In this section we define the actions that occur in Aegis during exception and interrupt processing and what is required to perform an upcall. Exceptions are synchronous events corresponding to the current computation (e.g., a TLB miss). Interrupts, in contrast, are asynchronous, and caused by external events: they may or may not be associated with the current process. Upcalls transfer the program counter from one domain into another. They are the building blocks of IPC.

The exokernel forwards all hardware exceptions except system calls and interrupts. Generally, each exception or interrupt must have a *save area* to hold state. We look at two methods for managing these save areas. The first method is to treat exceptional events as function calls and dynamically allocate an activation record as each occurs. Unfortunately, dynamic management adds overhead to the base case (e.g., it requires checks for stack overflow, etc.). The alternative is to disallow "recursion" in these events and to statically allocate an activation record for each possible event that can occur simultaneously. [1] Recursion in this context means that handlers cannot cause or incur an exceptional event of the type they were processing. For example, an exception handler dealing with integer overflow must not itself incur an integer overflow exception.

The exokernel statically allocates event activation records. These activation records are specified as physical memory locations at process creation (they can be subsequently changed); they are authorized through capabilities, are 36 bytes in size, and cannot straddle page boundaries. In the case of exceptions, applications are expected to refrain from incurring additional exceptions during exception handling. The caveat to this constraint is user-TLB exceptions. It would be overly restrictive to force applications to avoid TLB misses while processing general exceptions, therefore Aegis allows the definition of a separate TLB miss area. Interrupt handlers are either called with the given interrupt disabled or must arrange that subsequent interrupts of the same type will not overwrite the current interrupt state. For example, in the case of a network interrupt handler, Aegis will not vector subsequent network interrupts to the application until the application is finished processing the previous one.

[1] This organization is analogous to early versions of FORTRAN that, because they did not support recursion, pre-allocated activations records at compile time.

We note that applications that wish to extend these semantics can do so through "prologue" code that manages a stack of event save areas: when an exception or interrupt occurs, its state is copied to a new event activation record; after this point, additional events can occur without overwriting the previous state.

3.2.1 Exceptions

To forward an exception, Aegis:

- Saves three scratch registers into an agreed-upon save area. To avoid TLB exceptions, Aegis does this operation using physical addresses.
- Loads the exception program counter, the last virtual address that failed to have a valid translation, and the cause of the exception (included in this is information about whether the exception occurred in a branch-delay slot and, on co-processor faults, which co-processor was responsible).
- Uses the exception cause to perform an indirect jump to a user-specified program counter value; execution resumes with the appropriate permissions set (e.g., interrupts are re-enabled).

The application is then responsible for handling the exception and restoring any state that was saved during exception forwarding. After processing the exception, applications resume execution directly or initiate a system call that will restore saved registers and set the program counter to a specified address. Allowing applications to return from their own exceptions (without kernel intervention) requires that Aegis ensure that all state it saves be available for user reconstruction. The means that all registers that are saved must be in user-accessible memory locations, etc.

Currently, Aegis dispatches exceptions in 18 instructions. In the absence of cache misses these 18 instructions consume 18 clock cycles. Figure 3-1 shows the MIPS assembly required to do this (the data-structure fields it accesses are defined in Section 3.3). The low-level nature of Aegis allows an extremely efficient implementation: exception forwarding requires almost *four times* fewer instructions than the most highly-tuned implementation in the literature [97]. Part of the reason for this improvement is that Aegis does not used mapped data structures, and so does not have to carefully seperate out kernel TLB misses from the more general class of exceptions in its exception demultiplexing routine. Another reason for the improvement is the way in which the ExOS ABI allows application-level code to manage the saving and restoring of registers as needed.

3.2.2 TLB exceptions

On the MIPS, the critical bottleneck in Aegis is the handling of user TLB misses: the entire system will be unusable if these exceptions are inefficient. We discuss how applications can effectively handle their own TLB misses and how the cost of this handling has been (largely) eliminated.

Supporting application-level virtual memory

An application's virtual address space is partitioned into two segments. The first holds normal application data and code. The second segment is used to hold exception handling

```
# Forward an exception to the current environment.
#

# 1. Separate interrupts and system calls from user−exceptions.
mfc0   k0, c0_cause             # get cause of exception
nop                             # delay slot
and    k1, k0, ~(8<<2) & 255    # mask out syscalls and interrupts
beq    k1, zero, sys_or_int     # jump to separate exception handler

# 2. Load pointer to current process
lui    k1, HI(PSW_ENV)               # load upper bits
lw     k1, LO(PSW_ENV)(k1)           # load pointer
# 3. Index into user−exception vector to find exception pc and save area.
and    k0, k0, 254                   # remove upper bits of cause
add    k0, k1, k0                    # index to exception handlers
lw     k1, GENX_SAVE_AREA(k1)        # load a pointer to the exception save area
lw     k0, ENV_XH(k0)                # load pointer to exception handler

# 4. Save 3 scratch registers.
sw     a0, A0_SAVE(k1)
sw     a1, A1_SAVE(k1)
sw     a2, A2_SAVE(k1)

# 5. Load exception cause, exception pc and exception virtual address.
mfc0   a0, c0_epc           # load exception program counter
mfc0   a1, c0_tlbbad        # load virtual address of faulting instruction
mfc0   a2, c0_cause         # load the cause register again

# 6. Return from exception (jump to user code and reset interrupts).
j      k0                   # jump to exception handler
rfe                         # return from exception
```

Figure 3-1: Assembly code to perform exception forwarding (18 instructions) code, page-tables, etc. The exokernel allows mappings in the second segment to be "pinned" through *guaranteed* mappings. A miss on a guaranteed mapping will be handled by Aegis, invisibly to the application. This frees the application from dealing with the intricacies of boot-strapping the TLB and exception handlers that can take TLB misses. On a TLB miss, the following actions occur:

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

3. The exokernel checks that the given capability corresponds to the access rights requested by the application. If so, the mapping is installed in the TLB and software TLB (STLB) [9, 49]; control is then returned to the application. Otherwise an error is returned.

4. The application performs cleanup and resumes execution.

The obvious challenge in supporting application-level virtual memory is making it *fast*. We accomplish this by overlaying the hardware TLB with a large software TLB (STLB) to absorb capacity misses [9, 49]. 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 efficiency gains the STLB enables are belied by the modest expenditure of memory it requires.

We briefly discuss STLB structuring issues; these insights, while not new, are not discussed elsewhere in the literature: we explore them to remove the current necessity of re-inventing the STLB design space.

At a high-level there are two competing STLB designs: a unified STLB, multiplexed among all applications and a private, per-process STLB. The important tradeoffs are discussed below.

Unified STLB: this structure shares space well (in a manner similar unified instruction and data caches), and its size and location are easily "fixed" at compile time, allowing these values to be encoded as constants. It has three main disadvantages: (1) multiple, possibly similar, address streams must be merged, increasing the probability of pathological "worst-case" behavior; (2) a more complicated hash function (since, again, it must merge multiple address streams); and (3) its large size can make flushing invalid mappings expensive.

Private STLB: this structure allows an efficient hash function and eliminates the need to merge multiple address streams. Additionally, since the STLB only maps a single address space, the context identifier is implicit in the structure, removing the requirement that the STLB be flushed when the context identifier is reused. There are two disadvantages: (1) large private STLBs waste space and small private STLBs waste time, and (2) for efficient hashing either there can only be a single STLB size, fixed at compile time, or the STLB hash function must be generated "on the fly" at runtime. The former approach is inflexible; the latter can increase context-switching costs (the MIPS TLB miss handler resides at a fixed address, forcing specialized TLB handlers to be copied to this location on every context-switch).

Less importantly, the simpler hash fucntion that can be used in a private STLB allows ranges of virtual addresses to be flushed more efficiently than in a unified STLB. It is not clear if this difference is important outside the context of micro-benchmarks.

Currently we use a unified STLB. To improve hashing coverage and to decrease the number of TLB flushes that occur when context identifiers are recycled, each process is associated with an 11 bit tag field: this field is constant over the process' life-time, and is recycled infrequently. To decrease the likelihood of "worst-case" hashing collisions, the tag is selected randomly from a collection of $2^{11} - 1$ tags. The STLB contains 4096 entries of 8 bytes each; it is a direct-mapped, resides in unmapped physical memory, and on an STLB "hit", replaces the desired mapping in 18 instructions — the additional load it adds (to check the virtual address space tag) resides in the same cache line as the mapping, and so does not add additional cache miss overhead.

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

The structure of our STLB is given in Figure 3-2; the assembly code required to index into it is given in Figure 3-3. To avoid the worst-case behavior of a direct mapped STLB, we will likely move to a two-way set-associative structure as the implementation matures (as is used in Rialto [33] and PA-RISC operating systems [49]). Another alternative is the use of a "victim cache" [52] to absorb STLB conflicts.

As noted in Huck [49], avoiding cache misses is crucial for efficient TLB replacement. This constraint can have a noticeable impact on TLB lookup. For example, currently we expend one cycle aligning the `tlbctx` register to 8-bytes; this instruction could be eliminated by separating the STLB tag from the entry it maps. However, doing so would put them on different cache lines and could thus result in an additional cache miss.

As a philosophical tie-in, the reader should carefully note the large number of tradeoffs involved in realizing an abstraction as simple as an STLB: the tradeoffs made in implementing traditional, high-level operating system abstractions should be frightening!

The STLB is the largest "non-essential" abstraction in Aegis. Unfortunately, despite the tradeoffs in its design, it appears to be required. To put a sharp point on it: we eat dinner with the devil, but hope the spoon is long.

3.2.3 Interrupts

Interrupts are caused by external events (e.g., timers, I/O devices). For our purposes, the only difference between interrupts and exceptions is that interrupts involve scheduling decisions: either the destination process is "woken up" or the interrupt is enqueued for later delivery. Currently, Aegis handles only timer interrupts. As it is extended to include disk and network drivers, time-slices will be accorded a modicum of control over interrupt delivery: a sequence of bits will be associated with each time-slice indicating whether it can be pre-empted by a particular interrupt.

Timer interrupts denote the beginning and end of time-slices, and are delivered in a manner similar to exceptions: a register is saved in the "interrupt save area", the excep-

```
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 3-2: STLB structure

tion program counter is loaded, and Aegis 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. The time the application has to save its state is bounded: if this limit is exceeded, Aegis destroys the application. In a more mature implementation the kernel will simply context-switch the application "by hand."

3.2.4 Upcalls

Aegis provides an *upcall* [25] mechanism as a substrate for implementing efficient IPC mechanisms (e.g., [12, 48, 67]). An upcall is a light-weight, cross-domain calling mechanism. Operationally, an upcall changes the value of the PC in the caller to an agreed-upon value in the callee (the "gate" [62]), donates the current time-slice, and installs the destination domain's context (context identifier, address space tag, and process status word).

Two flavors of upcalls are provided: *synchronous* and *asynchronous*. Synchronous upcalls transfer ownership of the current time-slice to the callee; the callee can return the time-slice via a synchronous upcall back into the original caller. Asynchronous upcalls simply donate the remainder of the current time-slice to the callee. Both implementations guarantee two important properties: (1) to applications, an upcall is atomic: once begun, it will arrive at the callee's gate; (2) the exokernel will not overwrite any application-visible register. Among their advantages, these properties allow the vast register state of modern processors to be used as a temporary message buffer [23].

Currently, our synchronous upcall implementation costs 30 instructions. Because Aegis implements the bare minimum required for any IPC mechanism, applications can pay for just the functionality they require. This is a very important property, since it allows IPC to be optimized for application-specific conditions. For example, upcalls between clients and trusted servers can be optimized by allowing the server to save and restore any registers that it uses, rather than requiring that the client save and restore its entire register state on every IPC. To the best of our knowlege, Aegis is the first operating system whose IPC semantics allow such conditions to be exploited.

Approximately a third of the IPC cost is due to the fact that the MIPS architecture requires that the system call "exception" be disambiguated from other hardware excep-

```
# 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 ^ 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 3-3: Assembly code used by Aegis to lookup mapping in STLB (18 instructions).

tions. The remaining twenty instructions can be slightly reduced with straight forward optimizations. The assembly code for synchronous upcalls is given in Figure 3-4.

3.3 Objects

Currently, Aegis supports five objects: capabilities, time-slices, pages, context identifiers, and environments. This section introduces the data structures and operations that can be performed on these objects. In general the design has emphasized simplicity. For example, static allocation is used throughout the implementation, in a manner similar to [22, 78]. Also, the Aegis implementation is completely unmapped (both code and data), eliminating the complication (and inefficiency) of virtual memory.

3.3.1 Capabilities

Capabilities consist of the resource name and a random sequence of bits; they are untyped and may be passed freely as data. Capabilities are *application-specified*: at allocation, the application presents the capabilities to which it wants the object to be bound. This allows "well-known" capabilities to be used to access resources, enabling a distributed notion of sharing. The owner of the resource can change any of the capabilities (or delete the object); there are no provisions (yet) to have more than one owner or to transfer ownership. The current implementation uses 32-bit capabilities; *this small size has been chosen strictly for convenience in writing the assembly language routines that use capabilities.* The structures involved are given in Figure 3-5.

Capability checks occur at resource-usage points. Because these checks can add overhead to certain operations, Aegis allows certain uses to be *bound*. For example, applications can insert bindings of virtual to physical pages in the TLB rather than have every memory access require a capability check. Less obvious areas include the exception save areas and guaranteed mappings. Logically, all bindings are flushed when the resource is deallocated.

3.3.2 Time-Slices

The time-slice structure is shown in Figure 3-6. The CPU is represented as sequence of time-slices; the number of time-slices is a compile-time parameter. Each time-slice has the length of the configured clock granularity (in the current implementation, 15.625 milliseconds). Usable time-slices are associated with an environment: a null environment value causes the time-slice to be skipped. Possessors of the write capability (`wc`) can change the environment which will be resumed when the time-slice is initiated.

The important fields in the time-slice structure are `ticks` and `int`. `ticks` is used to provide fairness by bounding the time an application takes to save its context: each subsequent timer interrupt is recorded, and when a threshold is exceeded, the environment is destroyed. When a time-slice is selected to run, `ticks` is checked: if its value is non-zero, `ticks` is decremented and the time-slice is skipped; if its value is zero, the time-slice is initiated. `int` indicates whether the time-slice can be pre-empted due to a device interrupt (currently `int` is unused).

The representation of the CPU that Aegis uses (i.e., as a vector of time-slices) is unique; an important property (discussed previously) is the notion of *position* that it encodes. The exokernel cycles through the time-slices in a round-robin sequence. This simple scheduler can support a wide range of higher-level scheduling policies. For example, a server could

```
        # Perform a synchronous upcall between environments.
        # Get rid of at least 6 instructions; bias towards IPC rather than
        # exceptions (can get rid of status register?)
        #
        # 1. Separate interrupts and system calls from user-exceptions (5 instructions).
        mfc0  k0, c0_cause               # get cause of exception
        nop                              # delay slot
        and   k1, k0, ~(8<<2) & 255      # mask out syscalls and interrupts
        beq   k1, zero, sys_or_int       # jump to separate exception handler
        lui   k1, HI(PSW_ENV)            # load upper bits to process status word

        ... << exception handling code >> ...

sys_or_intx:
        # 2. Separate interrupts from system calls.
        and   k0, k0, 254
        beq   k0, zero, interrupt        # interrupt has code = 0
        andi  k0,a0,0x007f               # force legal IPC index
        beq   t0, AE_IPC, ipc            # special case syscall: check to see if IPC
        sll   k0,k0,ENV_SHIFT_BITS       # shift index
        j     sysx                       # some other system call
        nop
        # 3. Perform IPC (heavily scheduled code)
ipc:
        # 4. Load and install destination Context (20 instructions)
        la    t0, env                    # load environment pointer
        addu  k0, k0,t0                  # index into it
        lb    t0, ENV_CID(k0)            # get destination context id
        lw    a0, LO(CUR_ENV)(k1)        # load pointer to current environ
        bltz  t0, cid_deallocated        # make sure it has a CID
        sw    k0, LO(CUR_ENV)(k1)        # psw_env = e
        sll   t0, TLB_HI_PID_SHIFT       # line up pid bits
        mtc0  t0, c0_tlbhi               # assert new pid
        lh    k1, ENV_TAG(k0)            # load address tag
        lb    a0, ENV_ENVN(a0)           # load environ identifier
        sll   k1, k1, PTEBase_SHIFT      # move tag into correct location
        mtc0  k1, c0_tlbcxt              # set new context
        lw    k1, ENV_GATE(k0)           # load gate
        lw    k0, ENV_STATUS(k0)         # load status register
        mtc0  k0, c0_status              # set status register
        j     k1                         # gate(psw)
        rfe
```

Figure 3-4: Assembly code to perform a synchronous upcall

```
struct auth {
    unsigned    n:12,           /* name */
                prot:2,         /* page-protection */
                cap:AUTHSZ;     /* capability */
};
```

Figure 3-5: Capability representations

```
struct slice {
    struct auth     wc,     /* write capability */
                    rc;     /* read capability */
    struct env      *e;     /* associated environment (null if no one) */
    unsigned short  next,   /* next slice */
                    prev,   /* previous slice */
                    ticks,  /* ticks consumed in interrupts */
                    int:1;  /* whether it can be pre-empted on interrupts */
};
```

Figure 3-6: Slice data structure

enforce proportional sharing (perhaps through lottery scheduling [105]) on a collection of sub-processes by allocating a number of time-slices; as each time-slice is initiated the server first determines which of its sub-process should run and then enables it by performing a `yield` system call to the chosen process. The exokernel's efficient implementation of `yield` allows high-level schedulers to perform their operations with minimal overhead (this overhead is measured in Section 4.3).

3.3.3 Page

The page structure is given in Figure 3-7. Each physical page has an associated `page` structure; the finest granularity of allocation is a single page. The minimum state for each page is quite simple: capabilities for read and write accesses. Applications that have the capabilities for a given page can install TLB bindings, issue DMA operations, and use it for exception save areas.

To support two 64-bit capabilities per page requires 16 bytes per page; assuming 4K pages, this translates into an overhead of $16/4096 = 0.4\%$.

```
/* page structure */
struct page {
    struct auth     wc,     /* write capability */
                    rc;     /* read capability */
};
```

Figure 3-7: Page data structure

3.3.4 Context identifier

Context identifiers are not associated with capabilities but are instead numbered by small integers. Each environment can allocate a small number of CIDs; they are installed via a system call. The ability to explicitly switch context identifiers allows an application to cheaply switch between multiple protection domains within itself.

Attentive readers will recognize that the representation of context identifiers violates a number of exokernel principles (e.g., that revocation is visible and naming is physical). We have made these violations with full awareness: they are necessary to allow efficient, fluid reclamation. These reasons are discussed more thoroughly in Section 2.4.

3.3.5 Environment

An environment is the unit of accountability in Aegis: all resource consumption is associated with an environment.

Environments contain the contexts required to define an address space and execution environment; they are the most complicated entities Aegis supports. Four distinct events can happen during program execution: an exception, an interrupt, an upcall, and an address translation. Environments must contain the four contexts required to support these events. Therefore, at a high-level, an environment exactly represents the binding of the following contexts:

- An *exception context* that includes values that the program counter should be set to when a given exception occurs (`xh`) and pointers to physical memory where registers can be saved when an exception occurs (`tlbx_save_area` and `genx_save_area`).

- An *execution context* that specifies PC values to both initiate (`pro`) and complete (`epi`) time-slices, a register save area (`intx_save_area`) for timer interrupts, and a status register (controlling what co-processors and interrupts are enabled). As devices are included into Aegis, this context will be expanded.

- An *upcall context* that specifies legal PC values for synchronous (`sync_gate`) and asynchronous (`async_gate`) upcalls. Currently, Aegis does not manage IPC bindings. This implies that any environment can call into any other — access control is managed by the application itself. Experience will indicate whether this task is too difficult.

- An *addressing context* that consists of a set of guaranteed mappings (`xl`, used to map the environment's page-table, exception handling code), exception stacks, an address space identifier (`cid`) and a tag used to hash into the system STLB (`tag`). To switch from one environment to another, Aegis must install these values (and that of the status register).

These are the base set of contexts required to define a process. All contexts require the others for validity: for example, an addressing context does not make sense without an execution or exception context. Currently, it is not possible to selectively substitute contexts. For example, it is not possible to have multiple exception contexts within the same addressing context: to achieve this structure, the application must either emulate it, or create a new environment. We have not found it useful to separate these contexts.

The read and write capabilities (`rc` and `wc`) allow an environment to alter the fields of another. This enables a server process to control an environment's exception context, base

```
/*
 * exception handler type:
 *      epc is the exception pc;
 *      cause is the exception cause;
 *      badaddr is the last bad virtual address.
 */
typedef void (*exh_t)(unsigned epc, unsigned cause, addr_t badaddr);

/* interrupt handler: epc holds the exception program counter. */
typedef void (*inth_t)(unsigned epc);

/* upcall: caller is the environment id of calling environment. */
typedef void (*upcall_t)(unsigned caller);

struct env {
      struct auth_t   wc,     /* Write capability. */
                      rc;     /* Read capability. */
      exh_t   xh[NEX];        /* Hardware exception handlers. */
      inth_t  pro,     /* Prologue code. */
              epi,     /* Epilogue code. */
              init;    /* Called to initiate environment and hold continuation. */
      addr_t  tlbx_save_area,        /* Save area for TLB refill exceptions. */
              genx_save_area,        /* Save area for general hardware exceptions. */
              intx_save_area;        /* Save area for timer-interrupts. */
      /*
       * The following four fields are clustered to be on the same cache line(s); they
       * completely characterize the context of the environment. These fields are what
       * is installed on context-switching by Aegis.
       */
      signed char     cid;            /* Address space identifier. */
      unsigned char   envn;           /* Environment number. */
      unsigned        tag:11;         /* 11 bit tag. */
      unsigned        status;         /* Status register. */
      upcall_t        sync_gate,      /* Synchronous upcall entry point. */
                      async_gate;     /* Asynchronous upcall entry point. */
      struct tlb      xl[MAXXL];      /* Guaranteed translations. */
};
```

Figure 3-8: Environment data structure

mappings, etc. More powerfully, it allows environments to allocate resources and charge them to another environment. The `rc` capability allows the named resource to be read by a given environment.

The `envn` field stores the current environment number: this is simply a time optimization for upcalls.

3.4 System Calls

The following are a subset of the system calls provided by Aegis:

`int yield(int who)`

> Yield the current time-slice to environment `who`; the flag `AE_RANDOM` causes the destination to be chosen randomly. When the yielding application's next time-slice is initiated, its execution resumes on the instruction after the system call, without going through prologue code.

`int sync_upcall(int who)`

> Perform a synchronous upcall into environment `who`. The current time-slice is donated until the callee re-donates the time-slice (through either a synchronous upcall back or explicit donation). The upcall fails if `who` does not exist (e.g., it has been deallocated). The exokernel does not alter any application-visible registers on this call, allowing all registers to be used to pass messages. Once initiated, the call completes in the destination address space.

`int async_upcall(int who)`

> Peform an asynchronous upcall into environment `who`. The remainder of the current time-slice is yielded to the callee; execution resumes at the instruction after the system call, without going through the prologue code. The upcall fails if `who` does not exist (e.g., it has been deallocated). The exokernel does not touch any user-visible registers on this call, allowing all registers to be used to pass messages. Once initiated, the upcall will complete in the other address space.

`int alloc(int type, auth c, void *obj)`

> Allocate a resource of type `type` to owner `c` initialized to the values in `obj`. The following resources can be allocated: pages, environments, time-slices, and context identifiers. `type` gives the type of resource to allocate: `AE_PAGE`, `AE_ENV`, `AE_SLICE` and `AE_CID`. `c` gives the write capability for the environment that will be charged for the resource. If `c` is set to -1, then the object is allocated to the current environment. `obj` holds the address of a page, time-slice or environment structure (it is `NULL` when allocating context identifiers). Objects can be requested by name (e.g., page 5). The exokernel publishes the "free list" associated with each object, allowing applications to derive an accurate picture of which resources are available. On success, the object name is returned.

`int dealloc(int type, struct auth c)`

> Deallocate object `c` of type `type`.

`int dma_cpy(auth dst, auth src, int flags)`

> Page-sized DMA (possibly software-emulated): `dst` is the write capability for the destination page, `src` the read capability for the source page. `flags` indicates whether the source or destination should be cached or not.

`int dma_set(auth dst, unsigned val, int flags)`

> Page-sized DMA (possibly software-emulated): `dst` holds the destination page's write capability, `val` the value to be written, and `flags` whether the write will be cached.

`int getrate(void)`

> Returns the current clock rate in milliseconds.

`struct {unsigned hi, lo; }  gettick(void)`

> Returns the current clock tick.

Errors

The following errors can be produced exokernel system calls:

`AE_PERM_DENIED`: The appropriate capability was not provided.

`AE_BOGUS_NAME`: The given resource does not exist.

`AE_BOGUS_TYPE`: The requested type does not exist.

`AE_DEAD_ENV`: The called environment has been terminated.

`AE_NO_FREE`: There are no unallocated instances of the resource.

3.5 Primitive Operations

The exokernel supplies a number of *primitive operations* that should be viewed as pseudo-instructions (similar to the Alpha's use of PALcode [88]). Each operation is guaranteed to not alter the value of any application register: as such it can be used to read and write machine state without requiring the saving and restoring of appropriate registers, etc. Some of these instructions are aggregates of underlying hardware instructions: they are provided to meet atomicity requirements (e.g., writing to the TLB is a single operation). With the ability to download application code into the kernel, these operations would be unnecessary. Interestingly, the ability to download code into the kernel [13, 31, 36, 104] would *simplify* the interface: Aegis would not have to provide all "reasonable" machine instructions and their aggregates but, instead, would just make the underlying hardware securely visible to applications.

An important implementation note is that the MIPS ABI splits the register set into callee- and caller-saved registers. To efficiently implement primitive operations requires either that the operation be coded in assembly or that the calling conventions be changed (i.e., so that caller-saved registers are not overwritten). Fortunately, the GNU C compiler [90] allows command line alterations of the register convention. This allows primitive operations (and other routines that must assume all registers are "live") to be coded in ANSI C and still have guarantees that they will not trample caller-saved registers. As a final implementation note, Aegis will likely remove the need to check for error codes in these operations: those that complete successfully will resume execution with the PC set to three instructions beyond their call site; those that fail will resume execution immediately after the call site. This convention will allow very light-weight error detection.

The provided operations are in relation to the current process; in the future, a layer of indirection may be added to allow server processes to more easily control their dependents.

`int mfc0(int reg)`

Coprocessor 0 is the conduit of all privileged instructions. Its registers hold many useful values (e.g., the process status word, etc.). Applications can read any of these registers using this primitive operation. The specific register to be read is specified by `reg`, which is a flag formed by capitalizing the register name as given in [54] and prepending `AE_` (e.g., `AE_CONTEXT`, `AE_ERROR`, `AE_STATUS`, `AE_BADVADDR`, etc.).

`int tlbwr(unsigned hi, unsigned lo, auth c)`

Insert the given virtual to physical translation in a pseudo-random TLB entry. The exokernel probes the TLB and deletes the mapping if it is already present.

`int tlbwi(int index, unsigned hi, unsigned lo, auth c)`

Insert the given virtual to physical translation at the hardware TLB line given by `index`. The exokernel probes the TLB and deletes the mapping if it is already present.

`int tlb_vadelete(unsigned hi)`

Delete the indicated virtual mapping; returns the mapping or -1.

`int tlb_padelete(unsigned pfn)`

Delete the indicated physical page.

`int tlb_probe(unsigned hi)`

Probe for a particular mapping; returns the low part of the mapping or -1.

`struct { unsigned hi, lo; } tlbr(int index)`

Read indexed hardware TLB entry; it is returned in indicated structure. (The STLB can be mapped by applications, removing the need to provide a similar instruction for it.)

`int tlb_dirty(unsigned hi)`

Delete dirty bit in virtual mapping `hi`; returns previous value or -1 if there was no match.

`int tlb_cache(unsigned hi, int cache)`

Change cache bit in virtual mapping `hi`; returns previous value or -1 if there was no match.

`int tlb_hwflush(unsigned lo, unsigned hi)`

Flush all hardware TLB mappings from `lo` to `hi` (inclusive).

`tlb_swflush(unsigned lo, unsigned hi)`

Flush software TLB mappings from `lo` to `hi` (inclusive).

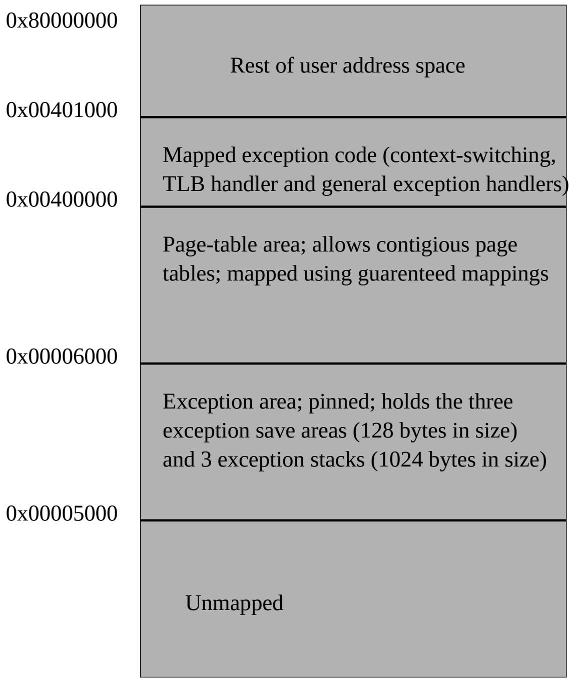

Figure 3-9: The address space structure of ExOS-based applications

`int cid_switch(unsigned cid)`

Install the context identifier `cid`.

`int fpu_mod(int flag)`

Enable or disable the floating-point co-processor. Returns the previous value.

`void int_rfe(unsigned epc)`

Return from interrupt. Restores the system call index register and the argument zero register; this primitive is used if the application is not able to simultaneously restore interrupt state while jumping to the interrupt site. (Similar functions are provided for general exceptions and TLB exceptions.)

3.6 ExOS: A Simple Library Operating System

This section gives a few details of our prototype library operating system: ExOS. The current system supports time-sharing, exceptions, virtual memory and process creation. We briefly discuss these issues below.

3.6.1 ExOS ABI conventions

The virtual address space structure is given in Figure 3-9. An important implementation point is that the exception and interrupt "save areas" are located in the lower 15 bits of the virtual address space. This allows loads and stores to be done directly using the "zero" register (instead of loading an offset register with the save area pointer and somehow restoring it). This allows efficient exception handling and easy state restoration after an exception has been processed.

The MIPS ABI reserves the registers `k0` and `k1` for the operating system. Since the application-level operating system has to perform most of the functions that motivated these registers, ExOS extends the MIPS ABI to include two application-level operating system registers: `u0` and `u1` (these were formerly the callee-saved `s6` and `s7` registers). In our experience, compilers on the MIPS do not make effective use of all callee-saved registers anyway, so this is not a traumatic modification (and has the small added benefit of making context-switching faster).

3.6.2 The live-register problem

This section discusses how an application can return from an exception or interrupt without kernel intervention. In the following discussion we use the term exception to refer to both interrupts and exceptions.

As a general rule, exceptions and interrupts can occur when all registers are live. To resume execution after processing an exception, the application needs to jump to the exception address using a register while simultaneously restoring this register. This requirement raises two challenges: the first is how to restore the register without requiring *another* register (which in turn would have to be restored, which requires another register, ...). The second is how to simultaneously restore a register and perform a jump using it. The first problem can be solved directly: because ExOS defines save areas low in the address space, registers can be restored using register zero as the offset register; this register always holds the constant zero and so does not need to be restored. For example, to restore register `r1`:

```
lw    r1, R1_SAVE_AREA(zero)  # restore r1 at constant offset R1_SAVE_AREA
                              # (this constant must be ≤ 2^15)
```

The second problem appears to have an easy solution as well: on the MIPS, jumps have delay slots. Therefore, the code sequence to jump and restore a register "simultaneously" could look like (again, we use `r1`):

```
j     r1                      # jump using r1
lw    r1, R1_SAVE_AREA(zero)  # restore r1 in the delay slot
```

Unfortunately, loads have a delay slot as well. This implies that the behavior of the jump's destination will be undefined if it uses `r1`. Therefore, we need to guarantee the invariant that the destination instruction cannot use the register restored in the delay slot. This invariant is guaranteed using two mechanisms: the first relates to application code, the second to library operating system code. The ExOS ABI prohibits application code from using `u0`. Therefore, if we return from an exception using this register, we are guaranteed that if application code initiated the exception, there will be no conflict. The other case we must handle is when the library operating system itself causes an exception. We handle this case by simply ensuring that a piece of exception handling code that uses `u0` cannot take an exception. This code is expected to avoid the general hardware exceptions (e.g., integer-overflow, unaligned pointer accesses, etc). To prevent TLB exceptions, all code that uses `u0` must be mapped using a "guaranteed mapping" (ensuring that it will not take

```
# exception handler to deal with unaligned pointers
unaligned-handler:
        ... process exception ...
        j       rfe-trampoline          # call trampoline to restore exception state
        nop

# a0 holds exception pc: restore this and jump to the value it holds
rfe-trampoline:
        move    u0, a0                  # move a0 to u0
        lw      a0, A0_SAVE(zero)       # restore a0
        j       u0                      # jump to exception pc
        nop
```

Figure 3-10: Return from exception using a trampoline function

an application-visible TLB exception). Obviously mapping all the application and ExOS exception handling code using guaranteed mappings would consume significant resources. To prevent this situation, a small "trampoline" function is used to restore exception state: general exception handlers do not use `u0` or restore their own state but, instead, jump to this function to restore the execution context (this code is given in Figure 3-10). In this manner, very little code must be mapped and exceptions are handled extremely quickly.

Variants of the above method has been used by Bedichek [11] and Thekkath [96]. An alternative solution is to return control using the kernel. Given the exokernel's efficient system call dispatching, this method may not be unacceptable for real applications.

3.6.3 Time-sharing

As discussed previously, Aegis calls application-supplied prologue and epilogue code on time-slice initiation and completion, respectively. ExOS does not stack interrupts, so saving and restoring state is more complicated than necessary. The problem that must be solved is that a timer-interrupt can occur while the application context is being restored. In practice this means that some of the registers will have been restored to their correct value, while the others will have garbage left over from the previous application. If the epilogue code blindly saved all registers it would overwrite correct values with these garbage values. To solve this problem, the epilogue code simply checks to see if the program counter was in the prologue. If it was, then the epilogue does not save the application's state and returns control to the operating system. A final problem that must be solved is that an interrupt can occur when processing exceptions, in which case the two registers `u0` and `u1` could be live. This situation is a variant of the live-register problem discussed in the previous subsection. It is solved by requiring that all code that uses `u0` or `u1` be placed at the beginning of the text segment. This organization allows ExOS to perform a simple conditional check on the program counter to see if it lies in this region or not. If the PC points to an instruction in this segment, ExOS returns control to the application using the exokernel's `rfe` routine. Otherwise, it restores the application's state using `u0` and `u1` and returns directly.

It should be noted that this complication is introduced as an optimization: ExOS could always return using the `rfe` system call. As we identify the bottlenecks in the system, this optimization may well be dropped.

3.6.4 Exceptions

With the exception of system calls, applications receive all exceptions directly. The exokernel demultiplexes the exception cause and transfers control to an application-specified exception handler. Applications are then responsible for handling the cause of the exception and resuming execution. In the future, to allow simple decoupling between applications and their library operating systems, the exokernel will allow system calls to be vectored directly to an application-level handler.

Exceptions are forwarded very efficiently; we measure and discuss uses for fast exceptions in Section 3.2.1.

3.6.5 Virtual memory

ExOS provides a rudimentary virtual memory system (its size is approximately 400 lines of code). Its two main limitations are that it does not handle swapping and that page-tables are implemented as a linear vector (address tranlations are looked up in this structure using binary search). Barring these two implementation constraints, its interface is richer than any other virtual memory system we know of: it provides flexible support for aliasing, sharing, disabling and enabling of caching on a per-page basis, specific page-allocation, DMA, etc.

The cost of application-level virtual memory is measured in Section 4.6; we also detail a myriad of uses for flexible application-level virtual memory.

3.6.6 Process creation

ExOS provides several `fork` variants (similar in spirit to the Plan9 `rfork` mechanisms [100]). Fork requires a few hundred lines of code, most deal with duplication of the page-table. Fork is interesting since it requires that the executing process create a clone of itself while running and do so without directly manipulating physical memory. The latter constraint can be dealt with effectively using Aegis' DMA support. DMA is not only efficient but is also a simplifying primitive since temporary virtual address mappings do not need to be installed for a single copy and then deleted.

3.7 Summary

We have described two novel software systems, Aegis and ExOS, and some pragmatic discussion of their realization. Aegis is an existence proof that an exokernel is not difficult to implement; ExOS is an existence proof that an exokernel is not difficult to use. The most interesting feature of these systems are that virtual-memory, process and trap management is located entirely in application-space, allowing an unprecedented degree of application-level control over how fundamental system abstractions are implemented.

Chapter 4

Application-level Implementations of Basic OS Abstractions

The exokernel exports hardware interfaces directly to application level, allowing efficient and flexible implementation and redefinition of traditionally kernel-level services. In this chapter we motivate the need for more flexible and efficient implementations of basic operating system services: IPC, virtual memory, process creation, exception handling and protection domains. For each of the examples we measure the key operations that they depend on; these results are compared against those obtained on a mature monolithic operating system.

While all of the examples show can be implemented directly in application-space on top of the exokernel, the majority cannot be implemented efficiently (or at all) on any other current operating system. There are four basic reasons for this inability. First, traditional OS-enforced abstractions of hardware resources hide valuable information, giving applications very little control over hardware resources. Second, operating systems necessarily make trade-offs in implementing abstraction. Since many trade-offs are application-dependent, this leads to poor performance in applications that were neglected or not anticipated by the kernel architects (e.g., LRU and MRU page replacement policies). Third, OS abstractions have to be more general and thus more expensive than specialized, tuned implementations. Fourth, extensions to traditional OS abstractions can only be implemented by kernel architects. In the real world this makes change very slow: creating sufficient incentive for kernel architects to incorporate a new feature into an already bloated, complex and fragile operating system is difficult.

This chapter tests three hypotheses. First, low-level multiplexing is not expensive: i.e., the cost of flexibility in an exokernel system is either negligible or easily recouped. Second, basic system abstractions can be implemented at application level in a direct manner. Finally, specialization and extensibility of these abstractions can cause substantial performance improvements. The following sections test each hypothesis for a particular operating system abstraction: Section 4.2 discusses processes, Section 4.3 IPC, Section 4.4 protection domains, Section 4.5 exceptions, and Section 4.6 virtual memory. Each section is divided into three parts. The first provides a brief discussion of exokernel capabilities in implementing each abstraction. The second presents a number of motivating examples that are enabled by an exokernel structure. The third gives a set of measurements demonstrating that the examples we discuss can be implemented both efficiently and well on top of an exokernel.

It is very important to note that the exokernel we measure is currently a *toy implemen-*

Machine	OS	SpecInt89	~MIPS	Memory
DECstation 2100 (12.5 MHz)	Ultrix 4.2	8.7	11	12 MB
DECstation 3100 (16.67 MHz)	Ultrix 4.2	11.8	15	24 MB

Figure 4-1: Experimental platforms

tation: it has no disk support, and only rudimentary software. While we do not expect these additions to cause large fluctuations in our measurements, we emphasize that ours is *not* a widely-used, robust implementation: the entire user community is, at the moment, three people.

4.1 Experimental environment

We run experiments within the DECstation/MIPS family; the machine configurations we use are shown in Figure 4-1. The two machine configurations are used to get a tentative measure of the scalability of the exokernel. All times are measured using the “wall-clock.” We used `clock` on the Unix implementations and a micro-second counter on the exokernel. The exokernel's time-quantum was set at 15.625 milliseconds. All benchmarks were compiled using an identical compiler and flags: `gcc` version 2.6 with optimization flags “-O2”. None of the benchmarks use floating-point instructions; we do not, therefore, save floating-point state. Both systems were run in “single-user” mode.

All experiments were measured by performing a large number of trials and dividing by this number to get the base cost of a single operation. None of the experiments used loop unrolling to minimize looping overhead. Since this overhead is equivalent on both systems, it understates the performance gains that our system obtains over Ultrix. Because such measurements do not consider cold start or capacity misses in the cache or TLB misses, they represent a “best case”. However, Ultrix has a much larger cache and virtual memory footprint than Aegis, again making this form of measurement more favorable to it than to the exokernel. We therefore believe that the difference in performance between the two systems is a conservative one. Finally, we note that 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 research operating systems. For example, it performs two to three times better than Mach 3.0 in a set of I/O benchmarks [75]. Also, its virtual memory performance (measured in Section 4.6) is approximately twice that of Mach 2.5 and three times that of Mach 3.0.

A few of the Aegis benchmarks were extremely sensitive to instruction cache conflicts. In some cases the effects amounted to a factor of three performance penalty. A minor rearrangement of object code during linking was sufficient to remove most conflicts. A happy side-effect of using application-level libraries is that object code rearrangement is extremely straight forward (i.e., a “makefile” edit). Furthermore, with instruction cache tools, all conflicts between application and library operating system code could be removed automatically — an option not available to applications using traditional operating systems! We believe that the large impact of instruction cache conflicts is due to the fact that most Aegis operations are performed at near hardware speed. As such, even minor conflicts are noticeable.

The base cost for a null procedure and system calls are given in Figure 4-2. The null procedure call is presented as a sanity check: given its minimal operating system requirements,

Machine	OS	procedure call	syscall (getpid)
DEC2100	Ultrix4.2	.57	32.2
DEC2100	Aegis (path1: stack)	.56	4.7
DEC2100	Aegis (path2: no stack)	.56	3.2
DEC3100	Ultrix4.2	.42	33.7
DEC3100	Aegis (path1: stack)	.42	3.5
DEC3100	Aegis (path2: no stack)	.42	2.9

Figure 4-2: Null procedure and system call; times are in micro-seconds

it should be (and is) the same on both operating systems. It obliquely shows that for Aegis' flexibility does not add overhead to base operations. Aegis has two system call paths. Both perform the same initial demultiplexing actions required by the MIPS architecture: a series of branches to determine the cause of the exception and whether the system call index is too large, and an indirect jump through a jump table to vector the PC to the appropriate value. The difference between them is that path1 uses a stack, while path2 does not. Path1 performs an additional jump through a jump table and has the additional overhead of saving and restoring a few registers and loading the exokernel stack. With the exception of upcalls, which are special-cased for efficiency, all Aegis system calls are vectored along one of these paths. It is not clear whether Ultrix's `getpid` routine uses a stack or not; regardless, it is approximately an order of magnitude slower than the either of the two system call paths on the exokernel — this suggests that the base cost of demultiplexing system calls will be noticeably higher on Ultrix. Part of the reason Ultrix is so much less efficient on this basic operation is that it is required to perform a more expensive demultiplexing operation. For example, page-table misses during TLB refills are vectored through this exception handler and so Ultrix must take great care not to disturb any registers that will be required to "patch up" an interrupted TLB miss. Because Aegis does not map its data-structures (and has no page-tables) it can avoid such intricacies. We expect that this will be the common case with all exokernels, since they should be quite small and therefore not require paging of kernel text and code.

4.2 Processes

This section discusses process scheduling, process creation, and context switching on an exokernel.

4.2.1 Process scheduling

The exokernel represents the CPU as a linear vector of time-slices. A crucial property of this representation is *position*, which encodes an ordering (and an approximate guarantee of when the time-slice will be run). Position can be used to meet deadlines, to trade latency for throughput, and to order process initiations. As discussed in Section 3.3.2, this simple scheduler can efficiently support a broad range of higher-level schedulers.

4.2.2 Process Creation

Traditionally, operating system support for process creation has been inefficient. This fact results from the fact that processes have typically been created using a single primitive, `fork()`, intended mainly for multiprogramming. The upshot of this inefficiency is that creating a new address space or protection domain has been very expensive.

Since the exokernel exports the bare minimum required to construct threads (i.e., time-slices) and address spaces (i.e., environments), applications have an enormous amount of freedom in how high-level process abstractions are implemented. Some examples are given below:

Efficient fault-isolation. Applications can isolate untrusted code (e.g., an RPC coming in off of the wire) by executing it in its own address-space/protection-domain. This flavor of fault-isolation is directly enabled by efficient process creation (i.e., the time to create a new address space is low).

Expressive process creation. Application-level process creation allows a great deal of flexibility in implementation; furthermore, as we discussed previously, since the implementation of these abstractions does not require special privileges, good ideas can be incorporated in systems other than those that they originated on (e.g., Plan9's `rfork` [100]).

Specialized `fork` implementations. Many micro-kernels omit effective optimizations that would complicate their virtual memory system. For example, QNX does not support copy-on-write, even though that could speed up process creation by an order of magnitude. Since applications completely control these implementations on top of the exokernel, they can incorporate (or neglect) optimizations as they see fit.

4.2.3 Efficient timer interrupts

The exokernel forwards all exceptions and interrupts to applications. A specific case we consider here is timer interrupts; when a time slice expires (i.e., a process' time-slice is complete) the kernel forwards a timer-interrupt to application-space. This interrupt handler is responsible for saving application state and performing any necessary clean-up actions before the process is suspended. This flexibility allows a number of simple optimizations:

Fast process context switching. Context-switching costs can be reduced by not saving floating-point registers if they have not been used. This well-known technique can substantially reduce the cost of context switches [72]. Information about the use of floating-point can either be deduced statically (e.g., through application-specific knowledge) or dynamically. Dynamic detection can be done in a number of ways. The most common technique marks the floating-point hardware as unusable: subsequent use will generate an exception, which is used to indicate that context switching should save floating point state. Applications can use this technique in an analogous manner.

Fast user-level thread context switching. The methods used to dynamically detect floating-point usage have not traditionally been made accessible to applications. When floating-point usage cannot be determined statically, thread context switching at user level has therefore been inefficient. If the dynamic detection of floating-point usage were possible at application level, thread packages could improve efficiency through the use of this simple technique.

Scheduler activations. The inefficiency of the process abstraction has led to a proliferation of user-level lightweight processes or threads. Unfortunately, since user-level threads are invisible to the operating system, their use can result in both performance and correctness problems [4]. For example, consider the case where two threads are multiplexed on top of a heavy-weight process: if one thread causes a page-fault, the OS will block the entire process, rather than letting the other thread run. Handling this problem is made possible through the exokernel structure, which exposes the physical resources and their associated exceptions (TLB misses, CPU time-outs, etc.) to the running environment.

Fast mutual exclusion. The explicit forwarding of timer-interrupts to application-level allows the exception program counters value at interrupt to be used to implement a variety of simple optimizations. For example, it allows efficient uni-processor synchronization through critical-section aware context-switching code that can "pc-luser" the program counter out of critical sections at context-switching time [15].

Fine-grained PC-sampling. Efficient timer interrupts allow fast (and thus more fine-grained) PC-sampling, a profiling technique based on the simple observation that, since the program counter will most frequently have values on the critical path, recording (or sampling) the program counter values at each timer-interrupt can closely approximate actual execution time [41]. This technique is much more efficient than a deterministic method such as described by Larus [10]; furthermore, it does not require compiler support.

4.2.4 Experiments

We verify that the use of prologue and epilogue code does not introduce noticeable overhead. This can be seen in Figure 4-2, which measured the time to perform a function call ten million calls. This measurement includes a large number of application-level context switches, and is slightly faster than the same operation under Ultrix, which performs "in-kernel" context switches. The lack of difference should be attributed to the fact that the time-quanta (15.625 milliseconds) is fairly large relative to the extra overhead induced by allowing applications themselves to context switch.

4.3 IPC

Fast IPC is crucial for a efficient, decoupled system [12, 48, 67]. As described in Chapter 3, the Aegis upcall mechanism is an efficient substrate for implementing fast IPC mechanisms. We enumerate some IPC mechanisms that can be constructed on top of it.

Trusted LRPC. IPC to a trusted domain can be optimized. For example, a client that trusts a server may, instead of a saving and then restoring its entire register state, allow the server to save and restore the registers it needs [48]. Because the machine state of current RISC machines is growing larger [76], this optimization is crucial for good performance.

Efficient LRPC. As we show by experiment, the efficiency of the upcall mechanism can be used to implement a very efficient RPC with traditional semantics [12].

4.3.1 IPC Experiments

Measurements section IPC is measured in four ways:

upcall: measures the "bare-bones" cost of the exokernel's upcall mechanism. Times are given in micro-seconds, and were derived by dividing the time to perform a call and reply by 2 (i.e., we measure the time to perform a uni-directional upcall). Untrusted upcalls save callee-saved registers accross calls; trusted upcalls trust the called server to save and restore any registers that it uses. These times are conservative in that they include the overhead to increment a counter and perform a branch. These times are one to two orders

OS	Machine	~MIPS	untrusted	trusted
Aegis	DEC2100/12.5MHz	11	5.9	2.89
L3	486/50MHz	10	10	n/a
L3	486 50MHz(normalized)	10	9.1	n/a
Aegis	DEC3100/16.67MHz	15	4.4	2.2
L3	486/50MHz	10	10	n/a
L3	486/50MHz (normalized)	10	6.67	n/a

Figure 4-3: `upcall` Benchmark; times are in micro-seconds

Machine	OS	Unoptimized (usec)	Optimized (usec)
DEC2100	Ultrix4.2	334	n/a
DEC2100	Aegis	30.9	24.8
DEC3100	Ultrix4.2	231	n/a
DEC3100	Aegis	22.6	18.6

Figure 4-4: `pipe` Benchmark; times are in micro-seconds

of magnitude faster than any similar operation available under Ultrix (in fact, they are an order of magnitude more efficient than `getpid`!).

We do an approximate comparison of our upcall to L3's RPC mechanism — which is the fastest in the world [67] — by scaling L3's times to the MIPS of our DECstation. Aegis's upcall mechanism performs well: the untrusted upcall is approximately 30% faster than L3, and the trusted upcalls approximately *2 times* faster. We have not tuned the Aegis upcall aggressively: architectural characteristics of the MIPS are one of the main determinants of our better performance relative to L3. For example, L3 pays a heavy penalty to enter and leave the kernel (71 and 36 cycles, respectively). While our base cost is not so high, much of the Aegis code does deal with required operations: demultiplexing the system call exception and setting the status, co-processor and address tag registers. Furthermore, for the trusted measurements, application-level code must save 8 general-purpose callee-saved registers.

pipe: measures the time needed to send a word-sized message from one process to another using pipes. It was measured by "ping-ponging" a counter between two processes. This experiment is based on Ousterhout's experiments [76]. The Ultrix `pipe` implementation uses the standard UNIX pipe implementation. The ExOS `pipe` implementation uses a shared-memory circular buffer. Writes to full buffers and reads from empty ones cause the current time slice to be yielded by the current process to the reader or writer of the buffer, respectively. The `pipe` implementation is an application-level library; the only kernel primitives used are the the `yield` system call and those primitives required to construct the application-level virtual memory. We use two `pipe` implementations: the first is a naive implementation, while the second exploits the fact that this library exists in application space by simply inlining the read and write calls. A further optimization (that will be performed in future experiments) is to alias the buffer, eliminating checks for messages that straddle buffer boundaries. Both exokernel `pipe` implementations are less than 200 lines of code. Times are given in Figure 4-4. Aegis' unoptimized `pipe` implementation is an order of magnitude more efficient than the equivalent operation under Ultrix. Much of this

Machine	OS	Time (usec)
DEC2100	Ultrix4.2	334
DEC2100	Aegis	12.4
DEC3100	Ultrix4.2	231
DEC3100	Aegis	9.3

Figure 4-5: `shmem` Benchmark; times are in micro-seconds

Machine	OS	untrusted	trusted
DEC2100	Ultrix4.2	680.	n/a
DEC2100	Aegis	13.9	8.6
DEC3100	Ultrix4.2	457.	n/a
DEC3100	Aegis	10.4	6.4

Figure 4-6: `lrpc` Benchmark; times are in micro-seconds

performance is due to the efficient implementation of `yield` on Aegis.

shmem: this experiment measures the time for two processes to "ping-pong" using a shared counter. The exokernel implementation uses the `yield` system call to yield the current time-slice between partners. Because Ultrix does not provide a yield-like primitive, acceptable efficiency can only be achieved by using pipes to emulate the required functionality. Times are given in Figure 4-5; as in the other IPC tests, the difference between Aegis and Ultrix is large: in this test Aegis almost a factor of thirty faster than Ultrix.

lrpc: RPC into another address space, increment a counter and return its value. ExOS's `lrpc` is built on top of the exokernel's `upcall` mechanism. There are two implementations: `trusted` and `untrusted`. `trusted` only saves and restores the stack pointer: the called domain is assumed to be a trusted server that will restore any registers that it uses. `untrusted` saves the general-purpose callee-saved registers. Ultrix does not have an RPC mechanism, we emulated RPC functionality through a server process that waited on a well known pipe: a client sends an index, the server calls the appropriate function, and returns the result through the pipe.

Both implementations assume that only a singe function is of interest (i.e., neither uses the RPC number to index into a table, etc.) and do not check permissions. Both implementations are also single-threaded. Times are given in Figure 4-6. ExOS's untrusted `lrpc` ranges between 49 and 44 fold faster than Ultrix, while the trusted version ranges between 79 and 71 fold faster: almost a two order of magnitude differential. The most important factor for this difference is the efficiency of the upcall mechanism.

4.4 Protection

Many applications would benefit from the ability to use protection domains as fire-walls for enhanced modularity or to safely execute untrusted code via quarantine (e.g., for *fault-isolation* [104]). The ability to decouple naming from protection is also important in a single address space operating system [20]. The exokernel allows applications to define their own page-tables and to directly manipulate context identifiers. This enables both lightweight protection and the separation of protection and naming.

4.4.1 Embedded protection domains

One method to allow untrusted code to be safely imported into an address space is to use *sandboxing* [104]. Sandboxing is a software fault-isolation technique that makes code safe by bounding memory operations and jumps. Unfortunately, the average cost of general protection (i.e., sandboxing loads, stores and jumps) is quite high. The SPEC92 benchmarks, running on DEC-MIPS and DEC-ALPHA machines, degrade in average performance by approximately 17-20% [104]. Additional costs of sandboxing include a pair of dedicated registers and, since sandboxing relies on the fact that the upper bits in a protection domain are all the same (i.e., there is only one segment), an increase in the difficulty of sharing across segments. Finally, sandboxing requires a trusted compiler; while this requirement is not great if such a compiler already exists, constructing one from scratch is tedious and error-prone.

The motivation of sandboxing is to simulate address-space protection domains when cross-domain calls are expensive. However, if application code can own and install several different context identifiers, then switching address spaces may require only a handful of cycles (e.g., to enter supervisor mode and change the processor context identifier). This use of *embedded protection domains* can speed applications up, eliminate the need to post-process object code (which can be quite challenging [106]), and allow multiple segments to be accessed simultaneously. We look at a few examples of where embedded protection domains would be useful.

Database systems. To eliminate the cost of cross-domain calls, database programs such as POSTGRES [92, 104] allow clients to download code into their address space. Embedded protection domains would allow them to efficiently isolate client code.

Client/server systems. Many client/server configurations (e.g., the Thor system [68]) would also benefit from the ability to ship client code to the server. Furthermore, the tighter coupling between client and server allows the cost of marshaling and copying (required if IPC is used for cross-domain calls) to be eliminated.

Light-weight application fire-walls. Large applications can use different addressing contexts to construct very lightweight protection domains. In effect, this allows "monolithic" applications (such as large servers) to be broken up into a more "micro-kernel" like design without the performance penalty typically associated with modularity.

Fine-grained thread access rights. Cheap, plentiful addressing contexts allow applications to decide on a thread-by-thread basis the access rights that will be granted to a given thread.

Single-address space subsystems. The exokernel exports all the primitives required to construct an address space with a separate protection domain, directly enabling the application-level construction of single-address space subsystems [20, 85].

4.5 Exceptions

Fast user-level traps enable a number of intriguing applications. The exokernel structure is uniquely suited to efficiently dispatch exceptions to user space. Its lack of management allows exceptions to be dispatched in 16 MIPS instructions, four times fewer instructions than the most efficient implementation in the literature (an implementation that improved the documented state-of-the-art by an order of magnitude [97]).

We look at a list of applications and operations that benefit from efficient traps. Many of these examples are drawn from Thekkath [97].

Machine	OS	unaligned	overflow	co-proc	`prot`
DEC2100	Ultrix4.2	n/a	272	n/a	294.
DEC2100	Aegis	2.8	2.8	2.8	3.0
DEC3100	Ultrix4.2	n/a	200.	n/a	242.
DEC3100	Aegis	2.1	2.1	2.1	2.3

Figure 4-7: `Trap` benchmarks; times are in micro-seconds

Emulation of sub-page protection [97]. Many applications would benefit from a page size smaller than that provided by current machines [5].

Efficient page-protection traps. These exceptions are used by applications such as distributed shared memory systems, persistent object stores and garbage collectors [97, 5].

Instruction emulation. The efficient emulation of privileged instructions is crucial for application-level OS emulation via direct execution.

Unbounded data-structures. Efficient unaligned pointer traps can be used to implement conditional signals of various flavors. For example, futures [29] can be implemented efficiently using unaligned pointer traps: the pointer of an unresolved future to its value is made to be unaligned. References to an outstanding future will trap, and the accessing thread can then be suspended. When the value is available, the pointer is set to it and all accesses to this "resolved" future then proceed normally.

In general, most of these operations could be done by inserting explicit checks in code (e.g., futures can be implemented by checking every load and store to determine if it is to an unresolved future). The obvious advantage to using exceptions instead of explicit checks is efficiency. A more subtle advantage is that the use of explicit checks requires compiler support. Writing a well-tuned, *correct* compiler that is portable and generates efficient code is a difficult problem; eliminating this requirement aids the efficient implementation of many operations. Furthermore, explicit checks take up space in the program text, possibly polluting the cache and causing an increase in paging.

4.5.1 Exception experiments

`unaligned`: Time to take an unaligned pointer access trap. This exception cannot be usefully exploited under Ultrix: the kernel attempts to "fix up" the unaligned access and writes an error message to standard error.

`overflow`: Time to take an overflow trap.

`co-proc`: Time to take a floating-point co-processor unusable trap. This exception cannot be caught under Ultrix, since applications cannot mark the floating point co-processor as unusable.

`prot`: Time to take a page protection fault.

Times are given in Figure 4-7. Careful tuning of the exception path (aided by the minimal kernel functionality Aegis provides) allows all traps to be dispatched approximately *two orders of magnitude* faster than Ultrix.

4.6 Virtual Memory

The exokernel is unique among operating systems in that it defers the whole of address space management to applications. To test the usefulness of this ability we examine a number of applications that would benefit from a tighter integration with the virtual memory system. To the best of our knowledge, the bulk of these examples cannot be done on any other operating system. The costs of the in-kernel primitives discussed in the paper are contrasted with the costs of exokernel-based, user-level implementations of these constructs. We break the discussion into five subsections: the effects of exposing all hardware capabilities are discussed in Subsection 4.6.1; optimizations that can utilize specific physical page allocation are examined in Subsection 4.6.2; uses for flexible address space structure are given in Subsection 4.6.3; the effects of tighter integration between the virtual memory system and application-level software are elucidated in Subsection 4.6.4. Page size trade offs are examined in Subsection 4.6.5 and page-table structures in Subsection 4.6.6. Finally, Subsection 4.6.7 provides a number of micro-benchmark measurements of VM system and the cost of application-level virtual memory.

Additional examples of inadequate operating system support for application-level virtual memory primitives can be found in [5].

Applications in the following subsections make use of the following six properties:

- NOCACHE: the ability to disable caching on a per-page basis;
- DMA: the ability to bypass the memory hierarchy and mapping hardware in order to write from one physical address to another;
- NAME: the ability to name which physical pages should be allocated;
- USERPT: the ability to access virtual memory mappings at application-level;
- UTLB: the ability to control the exceptions associated with the TLB (this is closely related to USERPT);
- PAGESIZE: the ability to chose the size of individual pages;

4.6.1 Exposing Hardware Capabilities

The exokernel gives very precise information of and control over the mapping hardware to applications (e.g., the TLB). For example, it exports all privileged instructions and page-attributes (such as caching) and allows applications to securely manipulate the TLB. Exposing the full set of hardware capabilities enables operations that are not possible on current operating systems:

Fine-grain monitoring. The tight coupling between applications and the virtual memory system allows efficient address tracing [101]. Furthermore, *any* application can do this at any time: no special dedicated hardware is needed.

Exposure of memory attributes. For example, reference counting and dirty bits can be simulated in software, once applications have access to TLB exceptions [7].

Monitoring translation hardware. TLB access patterns can be used to derive working sets, or to deduce which pages take frequent TLB misses; this enables techniques such as page-migration [19]. Knowledge about TLB misses is also useful because they approximate cache misses [19]. Application-level operating systems can use this heuristic to decide which pages to mark as uncachable, potentially improving global system performance. Additionally, with a deep memory hierarchy, local performance may improve as well (since applications will by pass the cache hierarchy instead of looking for data that is not there).

DMA operations. The exokernel allows applications to initiate DMA operations allowing them to avoid the memory subsystem during bulk data transfers, thus eliminating cache and TLB pollution. This optimization can aid many operations such as networking and garbage collection, and more common operations such as `fork()` and `memcpy()`.

Fine-grained control over virtual memory attributes. Giving applications access to the full complement of hardware facilities allows precise control of page information (e.g., reference bits, page-size, caching attributes, etc.). Reference bits can be used to track writes to memory pages (useful for garbage-collectors [5]). Application-controlled caching can be used to reduce cache pollution by disabling caching for memory that exhibits poor locality. For example, a log that absorbs many writes before being flushed to disk should not be cached, since it needlessly evicts cache entries (not only is the ability to disable caching a performance optimization, it also aids non-intrusive monitoring). I/O intensive programs and compression algorithms could also benefit from this method of reducing cache pollution.

4.6.2 Specific Page Allocation

An application's ability to request specific physical pages enables a large number of optimizations.

Cache conscious data layout. Control over the physical page numbers allows cache-conscious layout of data on systems with physically mapped caches [14, 18, 45]. For example, "page-recoloring" can be used to reduce cache conflicts [17]. This method is further assisted by the ability of applications to approximate the working set using TLB snapshots [83].

Contiguous physical page sequences. Applications can use this exokernel-bestowed ability to request specific physical pages. This functionality can be used to construct contiguous runs of physical pages. These blocks can be used to construct super-pages [94] and to improve the efficiency of DMA operations [34].

Cache multiplexing. Cache multiplexing is useful in two areas: (1) to guarantee that a process has sole access to a piece of the cache (e.g., for real-time programs perhaps) and (2) to limit the cache pollution of applications that exhibit poor locality (e.g., by forcing them to only use a small section). Cache multiplexing can be done directly via page-coloring: to allocate a specific piece of the cache to a specific process, the OS gives that process pages of the corresponding color; if guarentees have to be made that no other process can access this region, the OS does not allocate other applications pages of this particular color. An obvious use of this technique would be to restrict the impact of poor locality: by allocating the data and code that exhibits poor locality on pages of a particular color, the damage the they can do to the cache is restricted. An important advantage of this technique over just marking pages as uncachable is that applications still get the advantages of cache line prefetching.

"No cache" guarantees. The correctness of some operations requires *guarantees* that certain pieces of data do not reside in the cache. For example, DMA operations by-pass the cache, and thus cause a coherency problem between the data in the cache and the value in main memory: to ensure correctness applications must initiate a cache flush. A more efficient mechanism is to use *intentional cache conflicts* to ensure that the cache does not hold a piece of data (i.e., the "flush" becomes implicit).

Performance debugging. Increasingly, performance anomalies are claimed to be instruction cache related [104]. The control the exokernel provides over page selection allows such propositions to be verified.

4.6.3 Address space structure

Application control over the structure of its address space allows a number of interesting techniques.

Better static fault-isolation techniques. With control of their address space layout, applications place sensitive state in arbitrary locations. This control can be used for improved fault isolation, by reducing the chance that a write or read will access this state; in a sense, the virtual address is a capability [110]. This technique can be used to allow applications to safely import untrusted code (or to guard against their own buggy algorithms).

Fine-grain control of naming. For example, parallel applications benefit from the ability to choose which data will be shared and which will be private: without control over naming, private data must be simulated by indexing into data-structures using their process identification number. This issue comes up in single-address space operating systems: without private naming, "private data" must be relocated to global addresses at load-time [20]. With private names, this naming can be done at compile time.

Control over layout. The ability to partition up virtual memory in an application specific way allows dynamically typed languages to partition their object space (type-segmentation) at different virtual memory segments, in order to have types be self-identifying [42]. An application level virtual memory library allows this operation to be done much more readily.

Aliasing. Many operations such as garbage collection benefit from aliasing [5]. For example, aliasing allows a garbage collector to map the same physical page using different virtual memory attributes: the mapping in the mutator's space can be read-protected, while the mapping in the collector's space has normal protections. A more prosaic example is the simplification of circular buffer algorithms by the ability to alias memory after the "tail" of the buffer to the "head": this ability can eliminate the need to special-case memory copies that straddle the circular buffer.

4.6.4 Tighter integration

Many applications would benefit from a tighter integration with the virtual memory system. We give some examples here.

TLB prefetching. If TLB penalties increase, TLB prefetching will become important [9]. Exposing the TLB to applications allows them to do this optimization simply and directly.

Appropriate page replacement policy. Since the exokernel allows applications to control paging, they can can implement an application-specific paging policy (e.g., LRU, MRU, by priorities, etc.). Furthermore, they can decide what to do with pages that are reclaimed. For example, garbage collectors do not need to page out scanned pages, since they only contain garbage. Unfortunately, current OS implementations do not support this optimization, and therefore garbage-collection results in a "flurry" of I/O activity, because the OS VM system does not realize that while the page has indeed been modified, it contains garbage and so does not need to be stored to disk [26].

Page-prefetching. A scientific application may follow well-defined access patterns, allowing it to predict which pages will be needed [45].

Accurate "in core" information. Accurate knowledge of which pages are resident in memory can be useful to many types of applications. For example, a scientific program manipulating large matrices could work on those pieces that are "in core", while prefetching others. Garbage collectors can use this information as well, by reclaiming only the storage which is resident in main memory. Finally, this information is critical to real-time applications, which must meet deadlines and cannot be subject to the vagaries of current VM systems.

Pinning pages. Predicating that certain pages are pinned in memory before proceeding can allow real-time applications to calculate and meet tighter deadlines [45]. Furthermore, control over which pages are paged-out can allow applications to exploit application-specific knowledge over what pages should reside in main memory. For instance, a database management system could ensure that critical pages, such as those containing directories, reside in physical memory [45].

4.6.5 Page size Trade offs

Increasing or decreasing the page size that is being mapped affects the amount of memory needed to map a virtual address space, the base unit of protection available, the working-set size [75, 93], and the frequency of TLB faults [75, 93, 94]. Generally, large-page sizes allow fewer TLB misses and (possibly) smaller page-tables, but at a cost of larger working-sets and a coarse grain of protection. These trade-offs are highly application dependent, and have thus far been poorly served by existing operating systems [94].

Many machines support multiple page-sizes [54, 51, 94]. There are a number of specific ways that page size fluctuations may be useful:

Reduced page-table overhead. The use of large pages directly reduces the size of page-tables needed to map an address space. Given that address space usage has been growing at a rate of a bit and a half per year [20], this is an important optimization. An object-based virtual memory system may find variable page size useful both in the reduction in fragmentation it can bring and in the space savings that can be had for objects larger than the base page size. E.g., a large object that is eight pages long may be mapped with a single page-table entry if 8 contiguous physical pages may be associated with it.

Reduced false sharing in parallel systems. Smaller pages are a space-efficient way of combating false sharing in parallel systems. Unfortunately, applications currently have no control over either page size or the mapping structure, which removes them from having any useful mechanism to realize this (other than having one object per page, which can get expensive). Small pages can be implemented through either hardware or emulation [97]. An additional area where false sharing may have overhead is on conventional shared memory machines where TLB invalidates of other processors must be done on protection changes, etc.

4.6.6 Page-table Structures

Page-tables are used to construct a discrete function from virtual to physical addresses. The characteristics of this function (its sparsity, the range of inputs and outputs it has to produce and the number of points it may be expected to match) have a deep impact on the appropriateness of a given structure [49]. Below we examine some specific arenas where the page-table structure may be stressed.

Different address space sizes. With the advent of 64-bit machines, the consideration of what address space size is appropriate for a given application should become increasingly important. The page-table mechanisms and structure used to map a sparse 64-bit address space (i.e., inverted page-tables) are different from those appropriate to a dense 32-bit address space (e.g., hierarchical page-tables). Current operating systems fix a general-purpose page-table implementation for all applications. This penalizes those applications that do not fit within the parameters used to derive the systems page-table structure. For example, an operating system on a 64-bit machine will likely chose a page-table structure that trades space in mapping large address spaces for speed in mapping small ones. This tradeoff is a needless one, since, if page-table implementations can be isolated in libraries, applications can select (or have selected for them) any one of an array of page-table structures. For instance, a sparse 64-bit address space could use inverted page-tables while a 32-bit address space on the same system could use hierarchical page-tables, and a protected object (that may be comprised solely of two or three pages) can use a simple linear vector to map its address space. A single-address space OS subsystem may use a completely radical structure; so too might a persistent object store.

Efficient support for sparsity. Sparse address spaces present mapping challenges in that the function of virtual to physical mappings must be able to cope with large holes between valid mappings and still remain efficient both in space and time. The sparsity of the address space will influence the computation/space tradeoffs made in a given page-table structure. Again, this is an example of a trade-off that, if done in application space, allows virtual memory implementations to be tuned to a specific application domain, rather than forcing all applications to use a general-purpose implementation.

4.6.7 Virtual memory experiments

We compare Aegis and ExOS to Ultrix on seven virtual memory experiments, based upon those listed in [5]:

`dirty`: Measures the time to query whether a page is "dirty" or not. Since it does not require examination of the TLB, this measurement is used to test the base cost of looking up a virtual address in ExOS's page-table structure. This operation is not provided by Ultrix.

`(un)prot1`: Measures the time required to change the page protection of a single page. On the exokernel system, this operation involves modifying the page-table, TLB, and STLB entries.

`prot100`: Measures the time required to "read-protect" 100 pages. On the exokernel system, this operation modifies the page-table, TLB and STLB entries.

`unprot100`: Measures the time required to remove read-protections on 100 pages. Involves updating the page-table and modifying the TLB and STLB entries.

`trap`: Time to take a page-protection trap.

`appel1`: Time to access a random protected page and in the fault-handler, protect some other page and unprotect the faulting page (this benchmark is "prot1+trap+unprot" in Appel et al. [5]).

Machine	OS	dirty	(un)prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Ultrix4.2	n/a	51.6	175.	175.	297.	438.	392.
DEC2100	Aegis	17.5	32.5	213.	275.	13.9	74.4	45.9
DEC3100	Ultrix4.2	n/a	47.8	140.	140.	240.	370.	325.
DEC3100	Aegis	13.1	24.4	156.	206.	10.1	55.	34.

Figure 4-8: Virtual memory benchmarks; times are in micro-seconds

`appel2`: Time to protect 100 pages, access each page in a random sequence and, in the fault-handler, unprotect the faulting page (this benchmark is "protN+trap+unprot" in Appel et al. [5]).

The exokernel based virtual memory system resides entirely in application-space. The current page-table structure we use is a simple linear vector: entries are found using binary search. Obviously, these times could be greatly improved.

`dirty` measures the time to parse the page-table for a random entry. If we compare the time required for `dirty` to the time required to perform `(un)prot1`, we can see that over half the time of in `(un)prot1` is due to the overhead of parsing the page-table. This overhead can be directly eliminated through the use of a data structure more tuned to efficient lookup (e.g., a hash-table). Even with this penalty, our system can perform these operations close to two times more efficiently than Ultrix. The likely reason for this difference is that, as shown in Figure 4-2, Aegis dispatches system calls an order of magnitude more efficiently than Ultrix.

In general, our exokernel-based system performs well on this set of benchmarks. The sole exceptions are `prot100` and `unprot100`. Ultrix is extremely efficient in protecting and unprotecting contiguous ranges of virtual addresses: it performs 20% to 60% more efficiently than Aegis in these operations. Part of this difference is a direct result of our immature implementation; another appears to be due to the fact that, on Aegis, changing page-protections requires access to two data structures (Aegis' STLB and ExOS's page-table) to change protection information. We anticipate these times improving as we tune the system. Fortunately, even with poor performance on these two operations, the benchmark that depends on this operation, `appel2`, is close to an order of magnitude more efficient on Aegis than on Ultrix.

`trap` is another area where the exokernel system performs extremely well (i.e., 21 to 24 times faster than Ultrix). This performance differential is achieved even though the `trap` benchmark on Aegis is implemented with standard signal semantics: for example, all caller-saved registers are saved. If these semantics were violated by ExOS, the performance difference would become even larger.

Finally, the higher-level benchmarks, `appel1` and `appel2`, also show impressive speedup: up to an order of magnitude in some cases and never less than a factor of five.

These numbers were achieved even though (or rather, *because*) all virtual memory management was performed at application-level. At first blush the micro-cost of application-level virtual memory would seem to add a large overhead to basic memory operations; these benchmarks show that this assumption is wrong.

Machine	OS	matrix
DEC2100	Ultrix4.2	7.1
DEC2100	Aegis	7.0
DEC3100	Ultrix4.2	5.2
DEC3100	Aegis	5.2

Figure 4-9: 150x150 matrix multiplication (time in seconds)

The overhead of application-level virtual memory

The overhead of application-level memory is measured by performing a 150 by 150 matrix multiplication. Because this naive version of matrix multiply does not use any of the special abilities of ExOS or Aegis (e.g., page-coloring to reduce cache conflicts), we expect that it will perform equivalently on both operating systems. The times in Figure 4-9 give a tentative indication that application-level virtual memory does not add a noticeable overhead to operations that have large virtual memory footprints. Of course, this is hardly a conclusive proof: we will investigate this issue more thoroughly in future work.

4.7 Conclusions

This chapter has demonstrated that the following hypotheses hold for an exokernel system:

- The overhead of low-level multiplexing is either negligible or can be directly counter acted. One of the most interesting results from this chapter is that application-level virtual memory can be implemented *without appreciable overhead.*
- Fundamental operating system abstractions can be implemented simply and well at application level. Furthermore, these implementations do not require much effort: the system was implemented by the author in a few months.
- The exokernel's structure allows many basic primitives and abstractions to be implemented up to two orders of magnitude more efficiently than in a traditional operating system.

Future work will involve a more thorough testing of our exokernel system. Two of the most glaring areas are in process creation and protection domain switching.

The single most important property of an exokernel is that it allows non-privileged applications to re-implement (and redefine) basic system objects. This allows systems built upon an exokernel to be *fundamentally* more flexible than those built on traditional operating systems.

Chapter 5

Related Work

5.1 Microkernels: The Garden Path of Simplicity

Micro-kernels were originally intended to solve many of the problems we have listed (poor reliability, poor performance and poor flexibility). Unfortunately, they have fallen short for a number of reasons. First, while they allow replacement of device drivers and high-level servers, such operations typically can only be done by trusted applications. Second, they are still in the business of providing a virtual machine to applications. The high-level interface that they enforce precludes much of the experimentation that applications desire (the reader is invited to compare the primitives described in Section 1.3 to current micro-kernels). Third, their rigid interface tends to be rudimentary or overly general when compared to their monolithic counterparts; they often achieve simplicity by implementing only a small set of high-level OS abstractions. For example, a micro-kernel may have achieved simplicity by dropping support for `mmap`, memory-mapped I/O, and full-featured virtual memory, but not given alternative mechanisms to implement the functionality these features provided: micro-kernels can give applications even less control over hardware resources than a monolithic system does.

Microkernel architects have realized that an operating system should be minimal. However, the manner in which they have attained this goal has not provided applications with much more flexibility than under monolithic systems. This thesis offers a better alternative method to construct minimal operating systems. An OS should become minimal not by enforcing a limited set of operations, but instead, through the systematic elimination of all operating system abstractions in order to expose the hardware to application-level software; from this primal mud, applications can craft their own abstractions, chosen for appropriateness and efficiency, rather than make do with general-purpose, inefficient and inappropriate abstractions.

5.2 Pico-kernels

Some "pico-kernel" architectures have pushed their interface very close to the hardware. However, they are typically biased to server architectures, restricting the ability of *application-level* software to locally manage resources. For example, usage of the low-level kernel interface is typically restricted to highly privileged servers, which are expected to be cooperative during resource allocation and deallocation.

5.2.1 The pico kernel

The pico kernel attempts to provide a minimal set of abstractions: virtual processors and protection domains [6]. As with the other minimal, abstraction-based kernels, the main difference between it and an exokernel is that it is abstraction-based: instead of concentrating solely on multiplexing and exporting the whole of hardware functionality, it attempts to provide a small set of abstractions. Unfortunately, abstractions necessarily hide hardware details and conflict, at least in some ways, with application requirements.

As in the Raven Kernel (discussed below) the pico-kernel designers have noticed that application-level services cause less communication between the application and its operating system.

5.2.2 SPACE

SPACE is an interesting "submicro-kernel" that provides only low-level kernel abstractions defined by the trap and architecture interface [79]. Its close coupling to the architecture has many similarities to the philosophy we have espoused. However, it is difficult to evaluate the SPACE approach, since the authors provide neither an explicit methodology nor performance results.

5.2.3 The Raven Kernel

The Raven kernel dislocates many traditional abstractions to user-space [82]: tasks, exception handling and virtual memory are all that is provided by the kernel. However, it does not share our belief that the operating system has no business abstracting the hardware. Rather, it views the operating systems main task as providing a reliable and convenient work environment: the "raw physical hardware [is] an environment far to exacting for higher level users to deal with" [82]. This view manifests in a simple virtual memory and task interface that hides many of the features of the underlying hardware.

An interesting result of the Raven work is that they have found that simply lowering the kernel interface increases efficiency, since applications do not have to explicitly communicate with the kernel for many operations.

5.3 The Cache Kernel

The Cache Kernel [22] is a low-level kernel that adheres to some of our precepts for a model operating system. The difference between the Cache Kernel and Aegis is mainly one of high-level philosophy: the Cache Kernel focuses primarily on reliability, rather than securely exporting hardware resources to applications. For example, the Cache Kernel attempts to eliminate all dynamic memory allocation (similarly to Popek and Klines' Data Secure Unix [78]). Unsurprisingly, this single constraint lowers the kernel interface as compared to traditional operating systems. However, the de-emphasis on application flexibility and extensibility is telling; the cache kernel is biased towards a server-based system structure (for example, it supports only 16 "application-level" kernels concurrently).

These quibbles aside, we believe that with several straight-forward changes the Cache Kernel would fit firmly and well within our definition of an exokernel.

5.4 Library Operating Systems

In a two-page position paper, Anderson argued for application-level library operating systems. To the best of our knowledge, he was the first to do so. He also proposed that the kernel concentrate solely on adjudication of hardware resources. However, he did not provide a design framework for such a system. For example, he did not address how secure multiplexing of physical resources would be accomplished. In addition, he does not report on an implementation of his ideas. Nevertheless, this paper can be considered a first step in the direction of an exokernel architecture.

5.5 SPIN

The SPIN project investigates adaptable kernels that allow applications to make policy decisions efficiently [13]. The SPIN system encapsulates policies in *spindles* that can be dynamically loaded into the kernel. To ensure safety, spindles will be written in a pointer-safe language and will be translated by a trusted compiler. An interesting result of their implementation is the pervasive role naming can play in protection: control of what applications can name (and hence manipulate) is, effectively, protection [16].

The SPIN project is attempting to solve a different from our work. Namely, they investigate the difficult problem of how large software systems be meaningfully and reliably parameterized by untrusted applications. This problem has at its core efficient, fine-grained control over access rights; as such it is a very general problem and can have an impact far beyond the narrow realm of operating systems.

We view the SPIN project complementary to our exokernel research, and we hope to directly use their results to optimize application-level operating systems.

5.6 VM/370

The interface provided by the VM/370 operating system [27] is very similar to what would be provided by our ideal OS: namely, the raw hardware. However, the important difference is that VM/370 provides this interface by *virtualizing* the entire base-machine. Since this machine can be quite complicated and expensive to emulate faithfully, virtualization can result in a complex and inefficient OS. In contrast, our approach *exports* hardware resources rather than emulates them, allowing an efficient and fast implementation.

The central tenet of the virtual machine movement (and VM/370 in particular) is that an application should not be able to detect that it is not executing on the native hardware. Supporting this illusion aggressively precludes application management: since the application is not supposed to see VM/70 it is unable to communicate with it over such issues as explicit allocation, revocation, naming and sharing (sharing is particularly difficult across virtual machines [59]).

5.7 Synthesis

Synthesis is an innovative operating system that investigated the effects of a tight coupling between operating system and compiler. In particular, Synthesis used dynamic code generation to generate extremely efficient code sequences for a fixed, high-level UNIX interface [72]. While their performance improvements were impressive, the fixed interface they provided precluded much application-level customization. Furthermore, an exokernel's low-level interface directly eliminates the need to partially evaluate complex operating system functions: all functions are simple and, therefore, quite efficient.

We expect to use dynamic code generation pervasively in the optimization of application-level operating systems; the main difference will be our reliance on portable, high-level methods for generating code on the fly (e.g., as described in [39, 38]).

5.8 HAL

Many operating systems have a hardware abstraction layer. If the exokernel ideas work in practice, these layers could aid the transition from traditional operating system structures to more flexible, efficient and simple exokernels. In the future, we hope to perform such a transformation on the codified HAL of Windows/NT [28]

5.9 Extensible Operating Systems

Many early operating systems papers discussed the need for extendible, flexible kernels [50, 65, 81, 107, 108]. Lampson's description of the CAL-TSS [60, 63] and Brinch Hansen's microkernel paper [43] are two classic rationales. Hydra was the most ambitious system to have the separation of kernel policy and mechanism as one of its central tenets [108]. Modern revisitations of microkernels have also argued for kernel extensibility [1, 80, 84, 95, 28]. Anderson and Kiczales et al. also recently argued for minimalism and customizable [3, 56].

The most important difference between our work and previous approaches is the explicit view that the operating system should not provide abstractions; as a result the interface in these systems is a much higher one (e.g., page-tables are implemented by the kernel).

Current attempts include Scout [44], Bridge [69], and Vino [87]. Some of the techniques used in these systems, such as type-safe languages [13, 36, 74, 81] and software fault-isolation [31, 104], are also applicable to exokernels. These systems are just beginning to be constructed, so it is difficult to determine their relationship to exokernels in general and Aegis in particular.

5.10 Dynamically loaded device drivers

The commercial world has long looked at extensibility in the form of dynamically loaded device drivers: the QNX operating system allows user-level handlers for device I/O [47]; Chorus servers are loaded into the kernel [84]; Mach 3.0 has migrated some of the AFS cache manager back into the kernel [75]; Lepreau *et. al.* has provided a tool to reintroduce servers into Mach 3.0 [66]. Draves has argued for selecting among several implementations of a specific kernel abstraction to allow customization [32]. However, all of these customizations require a very high privilege and none guarantee the safety of the code once introduced.

Chapter 6

Conclusions

In this chapter we summarize the exokernel approach, detail the contributions of the thesis, discuss future work and conclude.

6.1 Summary

Traditional operating systems encapsulate physical resources in high-level abstractions. Thus, applications have little recourse in the face of inappropriate or inflexible operating system abstractions. The exokernel structure proposed in this thesis attempts to eliminate the conflict between application requirements and operating system provisions by multiplexing the raw physical resources across applications. Informally, an exokernel strives to allow applications to control all resources available to traditional operating systems without either additional overhead or restrictions. The central principle behind an exokernel is that applications should manage all physical resources. Exokernels manage resources only to the extent required by protection (e.g., allocation, revocation, and ownership). This principle is based upon the belief that distributed resource management is the best way to build an efficient, flexible system. This belief, in turn, is built upon two simple insights. The first is that, fundamentally, there is no "best way" to implement a high-level abstraction. Therefore, hard-wiring the implementation in the operating system, where it can neither be changed nor avoided is a dangerous practice. On the other hand, enabling applications to implement the abstractions they require at application-level allows very efficient and flexible implementations. The second insight is that if a kernel abstraction is eliminated, it can never break, consume cycles, nor conflict with application requirements. In short, an exokernel eschews OS abstractions. Both of these intuitions motivate the exposure of all hardware capabilities to applications. From these hardware primitives, application-level libraries and servers can directly implement traditional operating system abstractions, specialized for appropriateness and speed.

6.2 Contributions

This thesis has attacked three important problems in operating systems: flexibility, performance and reliability. Its specific contributions are listed below:

1. The insight that the reason traditional operating systems are inflexible, inefficient and unreliable is because they are designed to provide abstractions of the underlying hardware.

2. The development of an explicit methodology to directly solve the problems of traditional operating systems. Specifically, the definition of a new kernel structure, the exokernel, that has as its central tenet the secure exportation of raw hardware resources to applications (i.e., the full range of privileged instructions, TLB hardware, DMA, etc.). The principles of an exokernel structure have been enumerated and many of the issues surrounding its design have been fleshed out.

3. The development of a prototype exokernel, Aegis, that provides a preliminary test of the exokernel approach.

4. The development of a prototype library operating system, ExOS, that is used as an existence proof that traditional operating system objects (processes, virtual memory, IPC) can be implemented effectively and *flexibly* in application space. One of the interesting results we show is that virtual memory can be efficiently implemented at application-level.

5. An initial quantitative evaluation of the exokernel ideas. Our initial results are quite promising: many of the benchmarks either run an order of magnitude more efficiently on the exokernel system than on a mature monolithic OS or, more compellingly, cannot even be done on any other operating system.

6.3 Future work

The existing exokernel precepts must be refined and expanded. For example, a more satisfactory framework for devices is needed.

At a practical level, a substantial amount of work needs to be accomplished before we will have a mature exokernel-based system. In the short term, the first two priorities are the implementation of disk and network device drivers and the higher-level software to use them. In the longer term, we intend to implement a complete operating system and services, so that the exokernel ideas can be tested in a production environment. Very high emphasis will be placed on making the implementation as simple as possible. For example, much of the code for the libraries and servers will be taken from the more stable public domain operating systems.

The fundamental system restructuring that an exokernel allows will present a number of interesting problems and insights. We intend to aggressively investigate how different libraries and servers can be specialized and tuned to different application domains.

Additionally, we will be constructing several distributed systems based upon the exokernel. The raw performance and close integration with the hardware should go a long way towards providing fast, high-bandwidth network communication. This raw power can be used as a strong building block for distributed applications; for example, a very efficient distributed shared memory system. Furthermore, it will allow us to build a completely location-transparent distributed system.

At an implementation level, there are a number of techniques we will utilize to further increase the speed of the current exokernel system. The first will be the use of dynamic code generation to tune TLB handling, exception forwarding and the Aegis upcall. The second technique is the use of downloaded application code to improve performance. Downloaded application code can be used to lower the cost of STLB misses (and, with careful attention eliminate the need for it all together), batch system calls and, importantly, simplify the exokernel interface: many of the system calls we require to expose the underlying hardware (e.g., privileged instructions) can be directly eliminated, since applications can download code to perform these operations directly.

6.4 Close

Traditional operating systems have been abstraction-based. These abstractions have hindered reliability and efficiency and conflicted with application requirements. We have proposed a new operating system structure to solve these problems. The central tenet on which this structure is built is that all hardware functionality should be exposed *directly* to applications without either operating system abstractions or management. This exokernel architecture represents a fundamental shift in how operating systems are structured.

We have documented a small step in the realization of a mature exokernel system. While initial results are quite promising (frequently one to two orders of magnitude better than a mature monolithic system on basic operations), further evaluation must be done to determine whether the exokernel architecture can be a successful guide for the next decades of operating system design.

Bibliography

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

[2] W.B. Ackerman and W.W. Plummer. An implementaton of a multiprocessing computer system. *Proceedings of the First ACM Symposium on Operating Systems Principles*, October 1967.

[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 *Proc. Thirteenth Symposium on Operating System Principles*, pages 95–109, October 1991.

[5] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Santa Clara, CA, April 1991.

[6] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, H. Peine, and R. Schwarz. Meeting the application in user space. In *Proceedings of the Sixth SIGOPS European Workshop*, pages 82–87, September 1994.

[7] O. Babaoglu and W. Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 78–86, Pacific Grove, CA, December 1981.

[8] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, 1994.

[9] K. Bala, M.F. Kaashoek, and W.E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on OSDI*, pages 243–253, June 1994.

[10] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 59–70, Albequerque, New Mexico, January 1992.

[11] R. Bedichek. Private Communication, December 1994.

[12] B. N. Bershad. High performance cross-address space communication. Technical Report 90-06-02 (PhD Thesis), University of Washington, June 1990.

[13] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - an extensible microkernel for application-specific operating system services. TR 94-03-03, Univ. of Washington, February 1994.

[14] B.N. Bershad, D. Lee, T.H. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[15] 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.

[16] Brian N. Bershad. Private Communication, December 1994.

[17] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct mapped caches. In *Proceedings of the Sixth International Conference on ASPLOS*, pages 158–170, 1994.

[18] Brian K. Bray, William L. Lynch, and M. J. Flynn. Page allocation to reduce access time of physical pages. Technical Report CSL-TR-90-454, Stanford University, 1990.

[19] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[20] Jeffrey S. Chase, Henry M. Levy, Michel Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report TR 92-03-02, University of Washington, 1992.

[21] 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.

[22] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *Proceedings of the Sixth SIGOPS European Workshop*, September 1994.

[23] D. R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating Systems Review*, 18:12–20, [10] 1984.

[24] D. R. Cheriton, G. R. Whitehead, and E. W. Sznyter. Binary emulation of unix using the v kernel. *Proceedings of the Summer 1990 USENIX Conference*, pages 73–85, June 1990.

[25] 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.

[26] Eric Cooper, Robert Harper, and Peter Lee. The Fox project: Advanced development of systems software. Technical Report CMU-CS-91-178, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

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

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

[29] R.H. Halstead D.A. Kranz and E. Mohr. Mul-t: A high-performance parallel lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[30] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1993.

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

[32] R. Draves. The case for run-time replaceable kernel modules. In *Fourth Workshop on Workstation Operating Systems*, pages 160–165, October 1993.

[33] Richard Draves. Private Communication, December 1994.

[34] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM'94*, pages 2–13, 1994.

[35] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The exokernel approach to extensibility (abstract). In *Proceedings of the First Symposium on OSDI*, November 1994.

[36] 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*, September 1994.

[37] 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.

[38] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. Submitted for publication.

[39] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.

[40] R. Goldberg. Architecture of virtual machines. *1973 NCC AFIPS Conf. Proc.*, 42:309–318, 1973.

[41] Susan L. Graham, Peter B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, June 1982.

[42] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, October 1993.

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

[44] J.H. Hartman, A.B. Montz, David 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.

[45] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on ASPLOS*, pages 187–199, October 1992.

[46] G. J. Henry. The fair share scheduler. *AT&T Bell Laboratories Technical Journal*, October 1984.

[47] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

[48] W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Fourth Workshop on Workstation Operating Systems*, pages 186–190, October 1993.

[49] 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*, 1992.

[50] D.H.R. Huxtable and M.T. Warwick. Dynamic supervisors — their design and construction. *Proceedings of the First ACM Symposium on Operating Systems Principles*, 1967.

[51] SPARC International. *The SPARC Architecture Manual Verson 8*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[52] 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.

[53] M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum. FLIP: an internetwork protocol for supporting distributed systems. *ACM Trans. Comp. Syst.*, 11(1):73–106, Feb. 1993.

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

[55] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, January 1988.

[56] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The need for customizable operating systems. In *Fourth Workshop on Workstation Operating Systems*, pages 165–170, October 1993.

[57] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS-TR94-220, Dartmouth, 1994.

[58] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for development of application-specific virtual memory management. In *Proceedings of OOPSLA*, pages 48–64, October 1993.

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

[60] Butler W. Lampson. *Scheduling and Protection in Interactive Multi-Processing Systems*. PhD thesis, Berkeley, 1967.

[61] Butler W. Lampson. Private Communication, November 1994.

[62] B.W. Lampson. Dynamic protection structures. *AFIPS Conf. Proc. 1969 FJCC*, 35:27–28, 1969.

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

[64] B.W. Lampson. Protection. In *Proc. 5th Princeton Conf. on Inform. Sci. and Syst.*, pages 437–443, March 1971.

[65] B.W. Lampson and H.E. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, May 1976.

[66] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeff Law. In-kernel servers in Mach 3.0: implementation and performance. *Proc. of the Third Usenix Mach Symposium*, 1993.

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

[68] Barbara Liskov, Mark Day, and Liuba Shrira. *Distributed Object Management*, chapter Distributed Object Management in Thor, pages 79–91. Morgan Kaufman, 1993.

[69] Steven Lucco. High-performance microkernel systems (abstract). In *Proc. of the first Symp. on OSDI*, November 1994.

[70] C. Maeda. Thesis proposal, January 1994.

[71] 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, 1993.

[72] H. Massalin. *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, 1992.

[73] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Mach Workshop Conference Proceedings*, pages 17–30, Burlington, VT, October 4-5 1990. USENIX.

[74] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th SOSP*, pages 39–51, Austin, TX, November 1987.

[75] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. *20th Annual International Symposium on Computer Architecture*, pages 27–38, 1993.

[76] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer Usenix*, pages 247–256, June 1990.

[77] M. Pagels. Private Communication, October 1994.

[78] G.J. Popek et al. UCLA data secure UNIX. In *Proc. of the 1979 National Computer Conference*, pages 355–364, 1979.

[79] D. Probert, J.L. Bruno, and M. Karzaorman. SPACE: A new approach to operating system abstraction. In *IWOOS*, 1991.

[80] R.F. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[81] 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.

[82] Duncan Stuart Ritchie. The Raven kernel: a microkernel for shared memory multiprocessors. Technical Report TR 93-36, University of British Columbia, Vancouver, B.C., Canada V6T 1Z2, April 1993.

[83] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First Symposium on OSDI*, pages 255–266, June 1994.

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

[85] J. Saltzer. *Naming and Binding of Objects*, chapter 3.A., pages 99–208. Lecture Notes in Computer Science. Springer-Verlag, 1978.

[86] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.

[87] Margo Seltzer et al. An introduction to the architecture of the VINO kernel, November 1994.

[88] R.L. Sites. Alpha axp architecture. *Comm. of the ACM*, 36(2), February 1993.

[89] M.J. Spier, Thomas N. Hastings, and David N. Cutler. An experimental implementation of the kernel/domain architecture. In *Fourth Symposium on Operating Systems Principles*, October 1973.

[90] Richard Stallman. Using and porting GCC.

[91] M. Stonebraker. Operating system support for database management. *CACM*, 24(7):412–418, July 1981.

[92] Michael Stonebraker. *Readings in Database Systems*, chapter Inclusion of new types in relational data base systems, pages 480–487. Morgan Kaufmann Publishers Inc., 1988.

[93] M. Talluri, S. Kong, M.D. Hill, and D.A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 415–424, May 1992.

[94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on ASPLOS*, 1994.

[95] 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.

[96] C. Thekkath. Private Communication, November 1994.

[97] C. A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[98] C.A. Thekkath, T.D. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *SIGCOMM '93*, pages 64–73, 1993.

[99] K. Thompson. UNIX implementation. *Bell Systems Technical Journal*, 57(6):1931–1946, July 1978.

[100] K. Thompson, R. Pike, et al. Plan9 Manual Page, 1991.

[101] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Schrest. Trap-driven simulation with tapeworm II. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 132–144, October 1994.

[102] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–267, May 1992.

[103] V.A. Vyssotsky, F.J. Corbato', and R.M. Graham. Structure of the multics supervisor. *AFIPS FJCC 1965*, pages 203–212, 1965.

[104] 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, 1993.

[105] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[106] D.W. Wall. Systems for late code modification. *CODE 91 Workshop on Code Generation*, 1991.

[107] B.A. Wichmann. A modular operating system. *Proc. IFIP Cong. 1968*, 1968.

[108] 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.

[109] Yahara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter USENIX 94*, 1994.

[110] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *Proceedings of the Summer 1993 USENIX Conference*, June 1993.