Applying Exokernel Principles to Conventional Operating Systems

by

John Jannotti

S.B., Massachusetts Institute of Technology (1997)

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Feb 1998

Author . . . .
Department of Electrical Engineering and Computer Science
Feb 4, 1998

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

Accepted by . . . .
Arthur P. Smith
Chair, Department Committee on Graduate Students

Applying Exokernel Principles to Conventional Operating Systems

by

John Jannotti

Submitted to the Department of Electrical Engineering and Computer Science
on Feb 4, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The exokernel approach to operating system design has demonstrated the potential for excellent performance gains through enhanced application flexibility in three experimental systems. However, each of these systems was designed as an exokernel from its beginning. Outside of academia, if the developers of a widely used, mature operating system wished to employ exokernel ideas, a complete redesign and rewrite would normally be unthinkable. This thesis explores how the advantages of an exokernel can be brought to a conventional operating system in an evolutionary way.

The focus is on I/O — disk and network — since device access tends to drive the performance of interesting modern applications, such as file and web servers. Two exokernel systems, XN for low level disk access and DPF for raw network access, are integrated into Linux. The thesis demonstrates how these interfaces can be integrated with Unix abstractions so that they are available to applications without forcing the applications to abandon their use of normal Unix abstractions.

It is demonstrated that DPF can be effectively integrated with little engineering effort for worthwhile performance gains. Despite the modest effort, Cheetah, a high performance web server, is able to specialize it's TCP/IP stack to outperform Harvest by 30-100%. XN's integration involves more effort, and it is more difficult to quantify its performance. However, microbenchmarks of primitive operations demonstrate that XN on Linux provides efficient primitive operations slightly *faster* than the raw disk access offered through reading and writing to raw devices.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The exokernel approach to operating system design has demonstrated the potential for excellent performance gains through enhanced flexibility in three experimental systems [4, 7, 6]. However, each of these systems was designed as an exokernel from its beginning. Outside of academia, if the developers of a widely used, mature operating system wished to employ exokernel ideas, a complete redesign and rewrite would normally be unthinkable. This thesis explores how the advantages of an exokernel can be brought to a conventional operating system in an evolutionary way.

The focus is on I/O — disk and network — since device access tends to drive the performance of interesting modern applications, such as file and web servers. Two exokernel systems, XN for disk access and DPF for network access, are integrated into Linux. The thesis demonstrates how these interfaces can be can be integrated with Unix abstractions so that they are available to applications without forcing the applications to abandon their use of normal Unix abstractions. An evaluation of the integration's success is provided; engineering effort involved is quantified, preliminary performance results are presented, and the completeness of integration of the exokernel interface is discussed.

1.1 The Problem

The idea of an exokernel was proposed three years ago [2], and in that time three research operating systems based on the idea have been built. Exokernels strive to expose hardware to applications to the greatest degree possible, while maintaining protection between processes. This affords the application greater flexibility and control than traditionally abstracted OS interfaces provide, leading to more efficient resource usage. Each of these systems has shown increased performance — with the latest, most complete system, Xok, showing improvements in the range of 10-300% for unmodified Unix-like applications and up to 800% for applications written to take maximal advantage of the flexibility offered by its low level interfaces.

Although exokernels offer performance advantages, their novel structure offers some challenges to existing systems. Each of the existing exokernels was designed from conception to be an exokernel. This leaves unanswered the central question of this thesis: How might an existing operating system gain the benefits of an exokernel?

One may view the integration of exokernel principles with an existing system along a spectrum. At one end, doing nothing; at the other, rewriting the system completely. For the purposes of this thesis, it is assumed that the first choice has been eliminated by a desire to reap the benefits offered by exokernels. It is further assumed that the second choice has been eliminated by a desire to minimize changes to an existing system, or, at the very least, to minimize the size of incremental changes between working, reliable systems.

The goal for this thesis will be to modify a conventional operating system (Linux) in an attempt to gain exokernel benefits, while minimizing the engineering effort involved. The set of modifications chosen should increase application flexibility substantially, while avoiding far reaching modifications to the existing operating system. This should provide a large share of exokernels' performance advantages, at a relatively low engineering cost. Design decisions will be an explicit topic of discussion, including exploration of how developers intending to gain even greater benefits might best proceed, as well as looking at the possibility of curtailing the effort expended in a more modest approach.

1.2 Approach

This thesis concentrates on I/O. This decision is based upon two observations. First, I/O plays an increasingly large role in determining the performance of large, modern systems. This is a result both of I/O generally be quite slow compared to the CPU and I/O's increased importance as more computers are networked and more applications handle large, streaming multimedia files. Second, when standard Unix applications were benchmarked on a Unix-like exokernel system (Xok), I/O heavy applications saw the largest gains. That is, exokernels seem particularly adept at improving system performance by enhancing I/O performance. Work is guided by examining the needs of Cheetah, a high-performance webserver that, on a "true" exokernel, provides speedups ranging from a factor of two to a factor of eight.

A high premium is also placed on the ease with which a single application may use both exokernel and Posix abstractions. Access to both interfaces is important, since the most common path to highly optimized applications that make use of exokernel abstractions will likely be a gradual migration from applications written for Posix compliant systems.

1.2.1 Cheetah

Cheetah makes use of low level access to the network and to the disk. For network access, Cheetah uses DPF [3]. To access the disk, Cheetah currently uses a very low level interface that XN has made obsolete. XN provides an equally flexible interface to the disk, but solves protection difficulties of the older interface. Since experiments indicate that XN's overhead compared to the older interface is negligible, XN was deemed more appropriate for this migration effort.

DPF allows Cheetah to efficiently implement its own TCP/IP stack at user level, while low level disk access allows it to implement its own filesystem. Cheetah uses it's network specialization to decrease total network usage. Cheetah's filesystem is a derivative of C-FFS [5], which provides excellent small file performance by colocating data blocks of files from the same directory and embedding inodes in directory blocks.

1.2.2 Engineering Effort

It would be inappropriate to report that this thesis's integration effort was successful without also giving an indication of the work involved. Quite simply, such a result result is almost vacuous without this qualification. Clearly, if developers are so inclined, any level of integration is *possible*. This thesis will help determine what level of integration is *worth it*.

One concern that developers may have is that the extraction of a single subsystem from Xok may be difficult. Indeed, many exokernel subsystems, at least ostensibly, require other subsystems. An application that uses DPF or XN also uses wakeup predicates [8] to avoid busy waiting. DPF and wakeup predicates require vcode. XN uses Xok's application accessible buffer cache. Many subsystems benefit from Xok's simple memory management that allows the kernel to access the memory of all processes, at all times, easily. A developer's worst fear is that these requirements will cascade, forcing a small project to add, for instance, DPF to also add nearly all of Xok, while rewriting much of the original operating system. For this reason special care will be taken to break this chain of dependencies early. The goal is to add *only* DPF and XN. A concession is made to add vcode, since DPF would be impossibly slow for it's purpose without dynamically generated code, but no other exokernel subsystem will be added. In fact, most of vcode could have been eliminated as well, only a small part of vcode is actually used. It was not separated however, since it required less effort to treat vcode as a black box.

A second concern is that the incorporation of a subsystem might require thorough understanding of the subsystem's internals. DPF, for instance, uses dynamically generated code. Many developers would be hesitant to modify dynamic code generation modules due to unfamiliarity with the process. In short, if current exokernel implementations make assumptions throughout their code that proves to be wrong for the target OS, it may be difficult for unfamiliar developers to modify a large body of code that they had no hand in constructing.

A final concern comes from the notorious difficulty of modifying operating systems. Developers may believe that adding an exokernel interface is likely to have wide ranging effects on apparently unrelated systems in the target OS. Although this is a valid concern that will be explored, it is worthwhile noting that this concern must be considered the least important of the three. If it proves difficult to untangle exokernel interfaces, or if exokernel implementations make assumptions that are unwarranted for the target OS, that seems to be an important failing for the general technique of migrating gracefully from conventional to exokernel structure. If, on the other hand, it is difficult to make even minor changes to the target OS to accommodate a new interface, this may indicate only a failing on the part of the target OS.

1.3 Contributions

This thesis serves as a data point in the space of implementing a portion of the ideas proposed by exokernel research. Currently, little is known about which optimizations in exokernel systems yield the highest gain, only that in combination, the ideas seem effective. Therefore, in a project of limited resources, it it is difficult to determine which ideas to tackle first. This thesis describes the results of one possible division of effort, thereby helping to illuminate the design-space.

Beyond providing data to help decompose the performance benefits of various ideas, this thesis serves as a demonstration of the types of problems that are likely to be encountered in such an effort. In earlier systems, every abstraction was designed with exokernels in mind — integration problems did not appear. In contrast, for this thesis to succeed, considerable thought must be given to integrating exokernel abstractions rationally with existing interfaces. The presentation of these problems, and suggested solutions, such as channels, may prove directly useful to other attempts to use exokernel ideas.

This thesis also serves as an existence proof for bringing exokernel subsystems to a conventional operating system. DPF and XN have been added to Linux, their performance is reasonable, and the code is understandable.

Some contributions of this thesis are in the form of negative advice. That is, some of the approaches taken seem regrettable in hindsight, but seemed reasonable at the outset. The experiences described here may help guide similar projects away from these pitfalls. In particular, the thesis concludes that in a project intended to guide an OS toward a completely exokernel design, a bottom-up, rather than goal-directed, approach should be preferred. The experience with XN suggests that simplified memory management and a low-level interface to the buffer cache should be tackled before approaching the higher level interface of XN.

1.4 Thesis Structure

Chapter 2 will discuss general issues involved in integrating exokernel interfaces with the traditional abstractions of a Posix system. Next, Chapter 3 will focus on the specific experience of implementing interfaces to DPF and XN following the guidelines laid out in Chapter 2. In Chapter 4 results are examined. An attempt to quantify both the success of integration and the work involved will be made. To further measure success, it will be determined whether the Linux interface is as flexible as Xok's, and how closely it approaches Xok's performance.

Chapter 2

Exokernel Interfaces in a Posix World

This chapter will describe some of the issues encountered in bringing exokernel interfaces to a Posix operating system. Posix is a standardized Unix interface, so the lessons learned here are applicable to many operating systems. The first section will describe, in general terms, some of the issues involved in reconciling exokernel and Posix interfaces. It will identify the varying use of communication as a central difference between these interfaces. The second section will explore some of the details of Xok's communication between user and kernel space. The final section of the chapter will describe how these communication needs can be addressed on a Posix system. It describes techniques to easily address a common assumption of exokernels: that the kernel, after validation, may treat pointers into user space as it does normal kernel pointers which may be dereferenced. Finally the *channel* is introduced as way of providing kernel to user notifications. A channel is a simple abstraction that allows a Posix system to notify applications of an important class of events efficiently.

2.1 Issues

An exokernel interface seeks to expose as much of the resource it is abstracting as possible. In short, not to "abstract" it at all. An exokernel interface is as close to the interface that the hardware provides as is possible without compromising protection. In order to do this, exokernels tend to use more communication paths between kernel and user space than more traditionally structured systems. As an example of how the desire to expose hardware creates more points of communication, Xok's interface to multitasking and page faults will be examined.

A reading of the Exokernel papers will further elaborate on the reasons behind these interfaces, but for the purposes of this thesis, it is most important to consider what they actually look like. What mechanisms does an operating system need in order to provide these interfaces? Further, how can these mechanisms be reconciled with the mechanisms behind the operating system's existing interfaces?

On exokernels, communication is as likely to be initiated by the kernel as the user and takes place in a number of ways, such as upcalls and shared memory mappings. On Posix systems, kernel initiated communication is uncommon, and the support is correspondingly weak. Enriching these interfaces is one of the keys to successfully moving an exokernel interface to a Posix system.

On Xok, all I/O interfaces are asynchronous. After an asynchronous call to the kernel to perform I/O, the kernel later communicates with the application to indicate that the I/O has completed. In both cases, this notification is by simply writing to the applications memory, updating a value or data structure that the application can inspect to learn of the event.

In contrast, on Posix systems, most information comes to the application as a direct result of making a system call. The kernel communicates either by return values, or by synchronous manipulation of buffers passed into the call. Beyond this, a signal may be delivered to an application asynchronously. A signal is a limited form of kernel upcall which transfers control to one of an application's signal handlers — functions that the application has previously registered with the kernel to handle particular signals.

Although signals, at least in name, seem to offer an excellent way to "signal" a process asynchronously, they are not always well suited for this task. Signals provide a mechanism for asynchronous communication from kernel to application for a few important events, but they are generally slow, awkward, and inflexible. First, signals provide no data. Thus, a signal can not provide enough information to support any real richness of information exchange. Second, signals offer an awkward programming interface. Signal handlers are called asynchronously, and present applications with a limited programming environment in which only certain system calls are guaranteed to work and only variables of a particular type (`atomic_t`) may be modified. As such, signal handlers generally must interact with the rest of the program in simplified ways, often by setting a flag that the program checks periodically and takes action upon after the signal handler has returned. Finally, signals do not queue. That is, while an application is servicing a given signal, any further attempts to deliver that signal to the application with fail.

2.2 Exokernel Interfaces

2.2.1 User Pointers

Consider the simplest example of system/user communication, the system call. This is generally the fundamental way for applications to communicate their wishes to the kernel, and Xok is no exception. Arguments, be they scalar or pointer values, are placed in designated locations (generally the stack), and a trap to the system is executed. The kernel receives the trap, and begins executing in privileged mode. After determining which system call is being executed, the kernel's first priority is to insulate itself from malicious or buggy user programs. Beyond the simple argument checking that any robust software must do (such as determining that an integer argument given as an index is within the range of the array it will index), kernel code must be careful not to reference memory through unchecked pointers. This can lead both to security breaks, if the application supplies a pointer that it is forbidden to read, but the kernel can; or to spectacular crashes as the kernel references unmapped memory with no means to catch the fault.

Manipulating pointers is the first place where one may run into trouble with an exokernel interface. In Xok's kernel, after checking that a user supplied pointer is valid, the pointer can be dereferenced at will throughout the kernel code, just as any other kernel pointer. However, this convenient memory model is not shared by all kernels. On some systems, special user access routines must be used to access user memory. When porting a Xok subsystem to another kernel, as long as static analysis can determine which pointers actually are kernel pointers and which are user pointers, this will not be a problem. Wherever the kernel dereferences a pointer that is actually a user pointer, special user access routines are substituted.

This becomes trickier if kernel code itself calls system calls, as is sometimes the case in Xok. This can only happen in a system with the simplified memory model and it makes it impossible tell, from the beginning of a system call, whether the call originated in user or kernel space. Thus, statically choosing whether to treat the pointer as a user or kernel space pointer is impossible. Techniques to disambiguate this confusion will be presented in the following section.

2.2.2 Notification

Moving beyond the relatively simple system call interface, the need for notification is common in exokernel interface. Yet there are two distinct classes of application notification. Quantum (time slice) expiration and page fault notification are high priority events. If they are ignored, the application is incorrect. This notification may be referred to as *mandatory*. I/O completion however, has somewhat less importance. The notification should be reliable, and, if the application is waiting for the event, it should be fast. But if the application is busy with other duties, the event should simply queue for later processing. For this reason, this need may be called *voluntary* notification.

Multitasking in Xok is exposed to the application. An application is notified when it's time quantum is about to expire, and it is expected to save it's state and register an entry point with the kernel from which it may be restarted. Xok does not know or care how to save a process context, and an application is free to do it however it pleases. The only protection concern is that the application does, in fact, give up the processor. Therefore, Xok's only concern is timing the application. If Xok notifies an application of it's impending quantum loss, but it fails to yield the processor (with the `sys_yield` system call) in it's allotted time, the application is simply killed.

Similarly, page faults are of no interest to Xok. If a process accesses unmapped memory, Xok essentially forwards the page fault up to the application. It is up to the application to determine if the access indicates a need to map a new stack page, allocate a data page, read in more of it's executable, or was, in fact, an error. Of course, Xok will not allow an application to map in another process's memory (without appropriate privilege), but beyond this protection, an application's memory management is it's own business.

These interfaces demonstrate Xok's increased need for kernel to user communications, but, for the purposes of this thesis's focus on I/O, Xok's voluntary notification system is more important. This interface revolves around kernel writes to interested applications' memories, with wakeup predicates used to determine when this has occurred.

Reproducing Xok's interface on a conventional Posix system may be difficult. Since Posix abstractions never require it, it can be somewhat tricky to write to the memory of a process that is not currently scheduled. This means that at the arbitrary time that an I/O request completes, it may be difficult to write to the requesting process's memory. On many such systems, doing so is *possible*, since physical memory is available through kernel accessible page tables. To actually do so requires that any user memory that the kernel might need to write in order to process an I/O request be pinned into memory at the start of the request in order to to avoid paging. Next, pointers would be translated to point to the memory through the kernel's window onto physical memory, and the request would continue as normal. When the request completed, the pages would be unpinned. There are difficulties however. First, not all systems map all of physical memory into the kernel for easy access. There are tradeoffs in doing so, and some systems simply choose not to do it. More important, however, is that even if such an interface were reproduced exactly, it would be inefficient for a Posix-like application to take advantage of it. Posix applications can not wait for arbitrary changes in their memory without wasting CPU time.

On Xok, the need for applications to block while waiting for memory to change (because the kernel serviced a request and wrote to it), is addressed with wakeup predicates. A wakeup predicate is a predicate written in a small, safe language. The predicate may examine application and kernel memory, compare values, and on that basis return true or false. It is downloaded to the kernel when an application is ready to block, waiting for an interesting event. The application is put to sleep, and the kernel runs the wakeup predicate each time the program may be scheduled. Only if the predicate returns true will the application actually receive CPU time. Wakeup predicates are avoided on principle (see Section 1.2), because their inclusion would be somewhat difficult without a prior simplification of memory management, and because they would require that `select()` be rewritten in terms of wakeup predicates in order to unify the blocking mechanisms.

Using the exokernel interface without wakeup predicates, an application would need to busy wait, constantly checking it's own memory to determine when the kernel had completed a request. In contrast, well behaved Unix applications wait for events using `select()`, which allows applications to wait, without using the CPU, for any of a set of file descriptors to become ready for reading or writing. It is imperative to allow applications to avoid busy waiting on a multitasking system, so some method of blocking on events is necessary. Furthermore, since applications should be able to use newly available exokernel interfaces alongside other Posix subsystems that require the use of `select()` for blocking on events, an exokernel-only blocking mechanism is inadequate.

2.3 Solutions

2.3.1 Copying

Recall the need for a kernel, running in privileged mode, to isolate itself from "bad" user pointers. For small amounts of data, the expedient solution is to simply copy the data upon entry to the system call. Special copy functions that safely copy user data to kernel memory are employed. These copies explicitly check that the referenced memory is valid before copying. From then on, the kernel may manipulate the structure as normal data, without concern for a surprise page fault.

2.3.2 Splitting System Calls

For calls involving more data, copying at kernel entry time may be too expensive. In such cases, there are a number of alternatives. A simple approach is to always use special access functions throughout the kernel to dereference these pointers into user space. This may involve modifying code that assumes it is working with "normal" pointers, but in a stylized way that usually does not require deep understanding of the modified code. This technique has a flaw however, when common kernel routines are called by system calls with user pointers *and* by kernel code with pointers to kernel memory.

A solution involves two fairly simple techniques. First, break all system calls into two pieces. A first, "thin", layer copies small user data, leaving only pointers to large typeless data as user pointers. Then, the "thick" layer performs most of the real system call work, but this layer can be called from inside the kernel since it assumes that all necessary copying has been done. This eliminates most of these problems, since kernel code can simply call the "thick" layer which assumes it is being called on kernel data. However, since large buffers are not copied by the thin layer, functions that take pointers to such data cannot be cleanly separated this way. The answer to this problem is a practical one. Empirically, kernels will rarely need to call other kernel routines that take large buffers, since such pointers are generally for transferring large chunks of data, something that the kernel will not need to do with itself. Finally, there is always a brute force approach. One can simply supply an extra argument for each pointer argument to the thick layer. This argument simply flags the pointer as user or kernel memory.

2.3.3 Signals

Mandatory notification, despite its greater importance, presents a simpler problem. Because mandatory notifications must be serviced immediately and they can be serviced outside the context of normal execution, Posix signals (or a close derivative) provide an adequate solution. This thesis however, is more concerned with the needs of "voluntary" notification, since it is appropriate for dealing with I/O.

2.3.4 Channels

To provide functionality along the lines of exokernel interfaces, a general notification mechanism is necessary that meets the needs of voluntary notification efficiently, and compatibly with other Posix abstractions.

This thesis proposes to leverage the existing `select()` infrastructure to address this need. Whenever there is an I/O notification need, the communication should be expressed as a write to a file descriptor. This will allow an application to block, waiting for a single event, by calling `read` on the descriptor, or to block for any one of a number of events (including normal, Posix events) using `select()`. For some interfaces, this will be completely natural, DPF will be an example. For others, where it is inefficient to copy the data itself through a file descriptor interface, the *channel* is introduced.

A channel is simply a new type of object that may be associated with a file descriptor, similar in spirit to a Posix pipe. The key difference is that the kernel is responsible for one end of the "pipe", rather than having applications at each end. The kernel may signal an application by writing a byte to the channel. In the existing prototype, the actual data written is immaterial. If an application wishes to wait on two separate events at the same time (and must distinguish between them), it simply uses two different channels. This interface was chosen to remain as close to the Xok interface as possible. However, considering the overhead involved in allocating what amounts to a pipe per channel, and the system limit on open file handles, a more efficient interface which wrote request identifying data (such as the device and disk block of a disk request) to the channel would be preferable. In this way, a single channel could be used for any number of pending notifications.

Although this thesis will only explicitly use channels in XN, they seem to be appropriate for expressing many notification needs. Their main weakness is that unlike wakeup predicates, important events must be predetermined by the kernel rather than flexibly chosen by the application. That is, there must be an explicit kernel interface that accepts a channel and promises to notify in case of a particular event.

Chapter 3

Implementing Xok Interfaces on Linux

This chapter will take a hands on approach to the issues raised in the previous chapter. It will describe Linux, and details of DPF and XN. Most importantly, it will describe how the details of these exokernel interfaces were modified in order to fit into Linux's existing structure. Finally, for each of DPF and XN, there will be discussion of how things might have been done differently. The discussions will focus on how a project with either more or less resources might have proceeded to gain more or less of the benefits obtained by the effort detailed here. However, the discussion of XN will also be guided by hindsight. It will detail how a different approach to integrating XN may be more successful.

3.1 Linux

Linux is a nearly Posix compatible system with some extensions. This work is based on Linux 2.0.30. The 2.0.x series of kernels is considered "stable", as compared to the 2.1.x series which evolves quickly by further development. 2.0.x was chosen for its stability, despite the fact that some improvements in the 2.1.x series would have made this work easier. Unless otherwise specified, "Linux" refers to the 2.0.x series on the x86 architecture.

For this thesis, the most important subsystems are memory management, the buffer cache, and networking infrastructure. A brief overview, mentioning only those points that affect later design decisions follows.

3.1.1 Memory Management

Linux divides the 4 GB virtual address space between the kernel and the application. The kernel is located above 3 GB, the application below.

The kernel's 1 GB of virtual address space is further subdivided. Beginning at the "bottom" of its address space, the kernel maps all physical memory. Keep in mind that, due to possible swapping, this does not mean imply that Linux can arbitrarily write to all the memory of any process at any time. Beyond this, of course, both kernel and user address space are further subdivided, but the exact divisions are of little consequence.

When the kernel is executing, pointers into user space must be accessed with special macros, rather than by simple dereference. This requirement goes beyond checking to determine that the pointer is valid. The macros use inline assembly to access user space by way of the `fs` segment register.

3.1.2 Buffer Cache

Linux uses a dynamically sized buffer cache to cache reads (including some read ahead) and writes. Writes are buffered for up to 30 seconds, unless they are written through a file descriptor that has been open with the `O_SYNC` option, in which case the cache is write through. All writes for a particular file may also be flushed to disk using `fsync()`.

Perhaps the most important "feature" of the Linux buffer cache is that it is highly tuned to perform the kinds of operations that Posix normally uses. For instance, requesting a buffer may be a blocking operation, since Linux may need to free buffers for reuse. Further, dirty buffers maintain a timestamp which controls when they will be committed to disk. These features will tend to get in the way when one prefers a low level interface that simply returns errors when operations would block, and allows the application to make policy decisions such as when a buffer should be committed to disk.

3.1.3 Networking

Linux's network implementation, like other Posix systems, has a fairly generic interface to sockets. This allows various protocols (TCP/IP, AX.25, Appletalk, IPX...) to work with the standard `socket()` interface. The code is organized to allow a particular implementation to set up callbacks to implement important, protocol-specific pieces of the interface. For instance, protocol-specific function pointers are called when the socket is created, connected or destroyed, as well as for each each read and write.

On the other hand, Linux does show its roots as a TCP/IP-only system. The "generic" socket code as well as the generic socket data structures (`struct socket` and `struct sock`) have a significant number of TCP/IP related details tangled in. Fortunately, since Linux offers a `SOCK_PACKET` interface that bypasses much of these details, it was fairly easy to determine what was important to all sockets and what was a TCP/IP related anachronism.

A final note. Linux's network buffers, `struct sk_buff`, are particularly nice for exokernel interfaces. From Linux's beginning, they were planned as linear buffers, in contrast to BSD's mbufs. This means that copies, when necessary, are fast and memory allocations in the networking code occur less often and are less likely to cause fragmentation.

3.2 DPF

DPF (Dynamic Packet Filters) is a system for efficient, generic demultiplexing of network packets to the applications that should receive them. As in other packet filtering systems, an application communicates its desires to the network subsystem by creating a "filter" — effectively a small bit of code written in a "little language" that is especially suited to the simple matching operations needed to determine the intended recipient of a packet. This filter is combined with all other filters that are currently in use to create one “meta-filter” which, when given a packet as input, returns the ID of the original filter that matches the packet. The packet is then delivered to the application associated with that ID. DPF differs from other packet filtering system in that this “meta-filter” is created by dynamically compiling the downloaded filters into native machine code. The demultiplexing operation becomes a function call.

There are two interfaces to consider when designing for DPF’s inclusion in an OS, corresponding to the two directions of information flow mentioned earlier. First, there is the simple issue of sending filter “programs” to the kernel. Second, and more complicated, one must consider how applications will actually receive packet data that the kernel has determined they are to receive.

Getting filter data into the kernel is a fairly simple matter. As filters are fairly small, perhaps 64 to 256 bytes, and filter downloading is a one time operation per connection, little performance is lost by copying the filter upon system call entry. In fact, even Xok, which could have avoided this overhead relatively easily, copies filters — the performance hit is so small that an optimization has never been implemented.

Providing packet data to the appropriate application is a more interesting problem. On Xok, besides supplying code for a filter, an application provides a block of its own memory to the kernel for use as a “packet ring”. When the kernel determines that a packet belongs to a filter, it copies the data directly into the provided memory (bookkeeping data is also manipulated so that the application knows, for instance, how much data has been copied).

Reproducing such an interface on Linux has many of the problems delineated earlier, making it both difficult and undesirable. On Linux, as on any Unix-like system, it is unnatural to modify the memory of a process that is not currently scheduled. Since packets arrive asynchronously, and are serviced at interrupt-time, the application that will receive a given packet will generally not be scheduled. Furthermore, pages that an applications has designated as a “packet ring” would need to be pinned in unpageable memory to allow them to receive packets at any time. This is complicated on Linux, since no other abstraction has such a requirement. The final straw is the difficulty in using such an interface, were it implemented. Without wakeup predicates, an application would need to busy wait, constantly checking the packet ring's meta data to determine when the kernel has put a new packet into the ring. The solution, already proposed, is to express the communication with file descriptors, thereby allowing `select()` to wait on events without spinning the CPU.

Our technique is to integrate DPF into Linux as a new socket type. This provides an excellent tradeoff between implementation effort and expected performance gain, as well as integrating perfectly with an event driven application's use of `select()` to avoid busy waiting. Adding a new type of socket to Linux is a fairly easy process, since there are multiple examples to follow. The interface for adding a new socket is intended to allow developers to write only the most socket-specific code (much like the vnode interface to filesystems). Using this existing code makes it easy to ensure that resource accounting is correct, since the socket independent code takes care of reference counts, duplication at `fork()` time, and deallocation. Finally, the interface is easy to use at the Posix application level. It performs as the user expects, just as any other other (datagram oriented) socket might. A more detailed report of the effort expended appears in Chapter 4.

3.2.1 Doing More

It is worth considering how a more ambitious project might gain more benefits. For receiving packets, the current implementation is already exactly equivalent to the Xok interface in terms of copies. Further, DPF is targeted at high performance servers, in which sending packets is certain to be more common that receiving them. These two facts lead to the conclusion that sending should receive attention next.

So far, sending packets to a DPF socket has received little attention in this thesis. This is because it essentially "came for free". Linux already supports a `SOCK_PACKET` interface for sending raw packets over the network. In terms of *flexibility*, this is exactly what Xok provides. However, there is room for improvement in the implementation, to bring network performance closer to that of Xok. The idea is for writing to a DPF socket to avoid copies. This requires a number of things. First, either applications or the kernel must have a mechanism for locking pages into physical memory. If the kernel provides this mechanism automatically, it must be a reference counted system (in case two or more packets from the same page are being sent), otherwise it will be up to the application to maintain reference counts, unlocking the page only after the packet has been sent to the network card. In either case, this system would require notification when the packet has been sent and the memory can be used again. Channels would handle this need quite well. This interface is particularly attractive when the ethernet card in question can perform DMA, since combining this interface with `mmap` or an application accessible buffer cache allows data to proceed from disk to memory to network without the CPU ever traversing it.

For the most ambitious, it may be possible to increase performance again by applying the same direct memory access techniques to reads. That is, allow the kernel to write incoming packet data directly to an application's memory. Although the number of copies would not be reduced (one copy to kernel memory must occur before running DPF to demultiplex the packet), there would be a reduction in latency. When an application is notified of the new data (again, using a channel), the packet will already have been copied to the application's memory. Although there is likely to be less benefit from this optimization over copyless writes, they require similar infrastructure (a simplified memory model), so the incremental cost of this work should be quite small.

3.3 XN

XN provides a mechanism for applications to gain low level access to disks without sacrificing protection. Each application may individually manipulate disk blocks, including meta data blocks, without concern that another application may corrupt meta data or access blocks without permission. Without doubt, XN is a more radical idea than DPF. While DPF can be compared to other packet filtering systems and presents only one abstraction, XN is unique to Xok and is a large system to digest.

All this is not to say that XN is overly complex. Rather it solves a complex problem. Using XN, the disk essentially becomes a statically typed heap in which disk blocks are the fundamental objects. An application that wants raw access to disk blocks must "convince" XN that it is playing by the rules of those disk blocks. Whenever a disk block is allocated, it is allocated either as a named root or, more commonly, as a child of a block that has already been allocated. XN uses a novel mechanism, Untrusted Deterministic Functions (UDFs), to let applications define what it means for a block to be allocated to another block. The upshot of this is that applications have almost completely control of the individual blocks on the disk, allowing user level decisions concerning allocation patterns and on disk structure, without sacrificing interprocess protection. The price is a fairly rich interface through which the application must describe what it intends to do before it does it.

The interface to XN on Xok is quite wide (see Appendix A). Approximately 20 system calls allow applications to create new types of disk blocks as well as allocate, free, read, modify, and commit these new types of blocks — if the rules imposed by their types are followed. Further, in Xok, cached data blocks may be mapped directly into the address space of a properly privileged application (that is, an application which is allowed to read or write that disk block).

Even more than DPF, bringing Xok's XN interface directly to Linux presents a number of difficulties. First, like DPF, XN's method of dealing with user pointers was designed for a system in which the kernel could transparently dereference user pointers (after confirming, early in the system call, that they were valid). In DPF, the expedient solution was simply to copy the structures in question. To some extent, that solution was possible on XN as well. However, since system calls in XN often involve pointers to entire pages of memory destined for the disk, it was necessary to avoid copies in these cases. Ostensibly, the solution is simply to replace accesses of memory through these pointers with accesses that use Linux's user pointer access routines. A minor complication is that XN, as implemented on Xok, calls some of its own system calls from within the kernel, for example to write kernel memory (superblocks, freemap blocks).

Since Linux's user to kernel access routines (correctly) fail to access kernel memory, a combinations of two strategies was employed. First, the splitting strategy of separating system calls into a thin isolation layer and thick work layer was used. However, some calls remained that called system calls with untyped pointers to kernel memory, which is still a problem, as indicated earlier. Although this could be fixed with extra arguments flagging calls that originate from XN, such calls occurred so infrequently that it was easier to change them to call lower level routines that did not use the user to kernel routines.

A more difficult case of using pointers to user memory occurred in the XN interface to asynchronous I/O. On Xok, an applications may call:

```
xn_err_t sys_xn_writeback(db_t db, size_t nblocks, int* ptr);
```

to asynchronously commit the `nblocks` disk blocks starting at `db` through to disk. Each time a block is committed, `*ptr` is incremented. This presents a difficulty similar to asynchronous packet arrival in DPF. Since the application that makes a disk request will not necessarily be scheduled when the request completes, it is difficult for the kernel to write to the application's memory. Further, as with DPF, an application would be unable to sleep in wait for the write to it's memory. On Xok, wakeup predicates solve this difficulty, but recall that wakeup predicates have been dismissed from inclusion in this work. To summarize, XN requires a mechanism to signal applications, but that mechanism should not require writing to the process's memory.

Again, the problem can be solved by expressing the communication as writes to a file descriptor. In particular, since the overhead of copying XN's large data arguments over a descriptor would be exorbitant, channels are appropriate. To use XN asynchronously, an application creates a channel (with `sys_channel()`), and passes it to the asynchronous XN calls which previously required a pointer to an integer to be incremented. When the kernel would have incremented the integer, it simply writes a byte to the channel. This allows the application to behave exactly like an application on Xok with respect to XN, but still use existing Posix abstractions.

3.3.1 Other Approaches

There are two major ways in which a similar project could proceed differently. The first way corresponds with doing less work, and, not surprisingly, getting fewer advantages. If the entirety of XN seems like too much work, and protected sharing is not a high priority, then applications may simply use raw partitions. The one feature of XN that may still be useful to add is asynchrony. This would allow applications written for the asynchronous exokernel disk interface to be ported with far less trouble, they would simply lack the protection of XN. Taking a minimalist approach to this project, one might implement two simple system calls: `int prefetch(int fd, int block, int n, int chn)` and a similarly typed `commit`. This would request that the `n` blocks beginning at `block` in `fd` be read into the buffer cache, notifying the channel `chn`, when the operation is complete. `Commit` would request that any buffer in the buffer cache backing the blocks in question be written to disk, again notifying `chn` when complete. Using a file descriptor open to a raw device, and using `mmap()` after being notified of the read, an application would have the same level of control (but not protection) that an applications using XN has.

If, on the other hand, a project is willing to spend more time than the approach of this thesis, other advantages can be delivered. A more ambitious approach should focus on the fundamentals of the target operating system first. Time spent there will pay off later as the project attempts to pull in more and more exokernel interfaces. Memory management should be regularized, the buffer cache should be simplified. Once this is done, XN can be integrated with relative ease using the techniques above to handle problems. The main difference is that using the system's buffer cache (which should have been simplified, but remained in place) now allows easy access to existing primitives for mapping kernel pages into user memory, and the added complexity of maintaining two buffer caches is reduced. One can imagine this greater integration greatly simplifying access to flexibility that exokernels promise. For instance, applications might use the existing, in kernel filesystem, but use XN primitive for asynchronous access and block level write-back control.

Chapter 4

Results

The goal of this chapter is to evaluate the changes made to Linux. First, the engineering effort required for the changes is evaluated. Although it is fairly difficult to measure the amount of engineering effort that goes into a project, a few metrics help to provide some understanding. We consider, for instance, the amount of change required in Linux, the amount required in the exokernel code, and finally, the amount of additional new code needed to "glue" the two together.

Next, performance is evaluated. For DPF, a version of Cheetah is measured to show DPF's performance in a network intensive application. For XN, benchmarks measure performance on primitive read and write operations.

The chapter will conclude with a discussion of flexibility. That section will determine, regardless of the actual performance involved, whether all of the same fundamental operations are possible in Linux's version of DPF and XN as on Xok's.

4.1 Engineering effort

DPF

From a software engineering point of view, the addition of DPF is quite a success.

The first metric is number, type, and extent of modifications to Linux. These results are summarized in Table 4.1. It seems clear, both from the size and types

File	Modifications	Lines
net/Config.in	Add `CONFIG_DPF` option.	1
net/protocols.c	Add `dpf_proto_init` to table of protocols	6
net/ipv4/packet.c	Removed `static` from `packet_sendmsg`	1
net/ipv4/af_inet.c	Removed `static` from `inet_ioctl`	1
net/core/dev.c	Called into `dpf_rcv()` to demux packets	20

Table 4.1: Linux Modifications For DPF

of changes made, that the required Linux modifications were almost trivial. Only `net/core/dev.c` required changes that were anything beyond boilerplate. Interestingly, even that code was copied almost line for line from the IP bridging code which also requires access to raw frames, and may remove those frames from later processing by the network stack. It was a pleasant surprise to find exactly the needed code at exactly the needed entry point. (Note that this also demonstrates a use of DPF. Given this low level entry point, IP bridging could be implemented with DPF at user level, a feature that previously required kernel modification.)

Next, it would make sense to look at the number of changes made to DPF (and vcode), in order to reconcile any assumptions made about kernel interfaces that are not true on Linux. Fortunately, there were no such changes, beyond those necessary for the Linux build environment. These included Makefile changes, changes to `#include` lines, and trivial wrappers for routines such as `printf` and `malloc` [1].

Finally, beyond changes to either Linux or exokernel code, additional code was necessary. This is the glue code that lets Linux networking code to use `AF_DPF` as a new socket type for reading and writing; as well as allocating, compiling, and freeing filters in response to the opening and closing of DPF sockets. These additions are confined to `net/dpf/af_dpf.c` and amount to only 208 lines, including comments and blank lines. Furthermore, about half of this code follows the template of any of Linux's other socket types (INET, IPX, Appletalk, AX.25, etc.) making it fairly easy to write. For the most part, it consists of the implementations of various callbacks into DPF code that Linux will make in response to system calls, process exits, packet

[1] Linux offers `printk` and `kmalloc` with nearly identical interfaces.

File	Modifications	Lines
include/asm-i386/unistd.h	Define constants for XN system calls	26
arch/i386/kernel/entry.S	Add entry points for XN system calls	27
fs/buffer.c	Callback XN and notify channels	26
include/linux/mm.h	Add Counted Page Locking	5
mm/page_io.c	Use new page locking	2
fs/nfs/bio.c	Use new page locking	5

Table 4.2: Linux Modifications For XN

receptions, etc.

XN

Although more lines of Linux code were modified for XN (see Table 4.1), the complexity of the added code was still quite small. Only the changes in `fs/buffer.c` and `linux/include/mm.h` contained "real" modifications. Other changes were simply to wrap page locking into access functions, define constants, or add to the list of system calls.

Unlike DPF, XN required modifications beyond build environment changes. One explanation for this is the relative immaturity of XN. While DPF was first used over two years ago, and, significantly, has been ported from it's original incarnation in a previous exokernel; XN is still not in a "polished" form, even in Xok. Therefore, a number of portability issues came into play while migrating the code. Many of these issues were essentially trivial, even if time consuming. XN used a single hard-coded device number, called the same internal routines with kernel pointers and user pointers, and used Xok buffer cache structures explicitly. These issues were generally easy to solve, and a reasonable case could be made for contending that even unfamiliar developers would have little trouble making the changes required.

Other changes required somewhat more effort. Changing XN to use channels, rather than incrementing an integer in user space is an example. While the change itself was systematic, and therefore relatively easy, it would be disingenuous to discount design time. A number of other ideas were considered, including an enhanced signal mechanism and enhancing (and therefore deeply understand) Linux's memory management to allow the interface to work as it had on Xok. Only after considerable thought, including an examination of the way the various proposals would interact with applications, was the idea of a channel solidified and preferred. However, since channels are a general mechanism, similar projects may be able to leverage the idea in other areas to shorten development time.

Some issues seemed less tractable, and though solved for the most part, still seem like rough edges. The most egregious is the inability to memory map a page that has been read in by XN. This is so clearly unacceptable, that one hesitates to call it a "rough edge". The only reason it exist is because the XN buffer cache is separate from Linux's normal buffer cache, thus the primitives for memory mapping the memory into application virtual address space are not immediately usable. They must be better understood before they are applied to memory outside what Linux considers to be the buffer cache. But this points out a failing with respect to engineering effort. Clearly, the right thing to do is to unify the Linux and XN buffer caches. Arbitrary partitions of this sort are an unnecessary source of added complexity. They are currently separate only because the Linux buffer cache was sufficiently complex that a separate buffer cache seemed the more expedient solution. These are issues that, while troublesome, one may be willing to forgive. They occur, not because the exokernel interface is complicated, but because the Linux interface is. It would be unsurprising to learn that the work could be easily righted by someone more familiar with the Linux buffer cache within a few days.

4.2 Performance

4.2.1 DPF

The performance of DPF was measured on Cheetah using workloads designed to avoid disk accesses. Thus the effects of low level access to the network can be observed, without clouding the issue with disk access. We expect to outperform other webservers dramatically with small files, with more modest gains as Cheetah's packet saving tricks become lost in the noise of larger transfers. We do not expect to be able utilize the network as well as Cheetah on Xok, since our network writes perform copies.

Performance is analyzed over a number of variables by measuring changes from an initial, default configuration. The server is a PPro 200 MHz with 128 MB of memory, three 100 Mb/s ethernet cards (Intel EtherExpress), and a SCSI disk system (which should be of little consequence). The default workload is three client machines, each making requests with a maximum of 10 simultaneous connections for a single document. Each client machine makes requests to a different ethernet card on the server. From this setup, parameters are varied over a number of axes. The effects of changing document size, numbers of simultaneous connections, and available CPU cycles are all explored. All experiments are repeated with Apache and Harvest. Apache is used because it is currently the most used web server on the Internet, and generally respected for its speed. Harvest is used because it was designed explicitly for speed and performs optimizations similar in spirit to Cheetah's. Additionally, Cheetah on Xok was compared to Harvest on similar hardware, therefore an approximation of Cheetah on Xok's performance on the default hardware can be obtained.

Benchmarks are run on document sizes ranging from empty to 1 MB, to explore the effect of document request size. Results are summarized in Figure 4.2.1.

Clearly, Cheetah is most effective for small documents where it is approximately a factor of two faster than Harvest (which in turn leads Apache by a wide margin). With small documents, the relative effects of using fewer packets on connect and close is greater. Cheetah's advantage shrinks as document size grows, but even in the "steady state" of 1 MB documents, Cheetah leads its competitors by over a third. This can be attributed to a number of specializations that Cheetah employs, such as precomputed checksums and delaying writes long enough to use maximally sized packets. It should be noted that the ease with which these enhancements were incorporated is due, in part, to an exokernel design that allows debugging and experimentation outside of the kernel.

Compared to Cheetah on Xok however, there is room for improvement. For in-

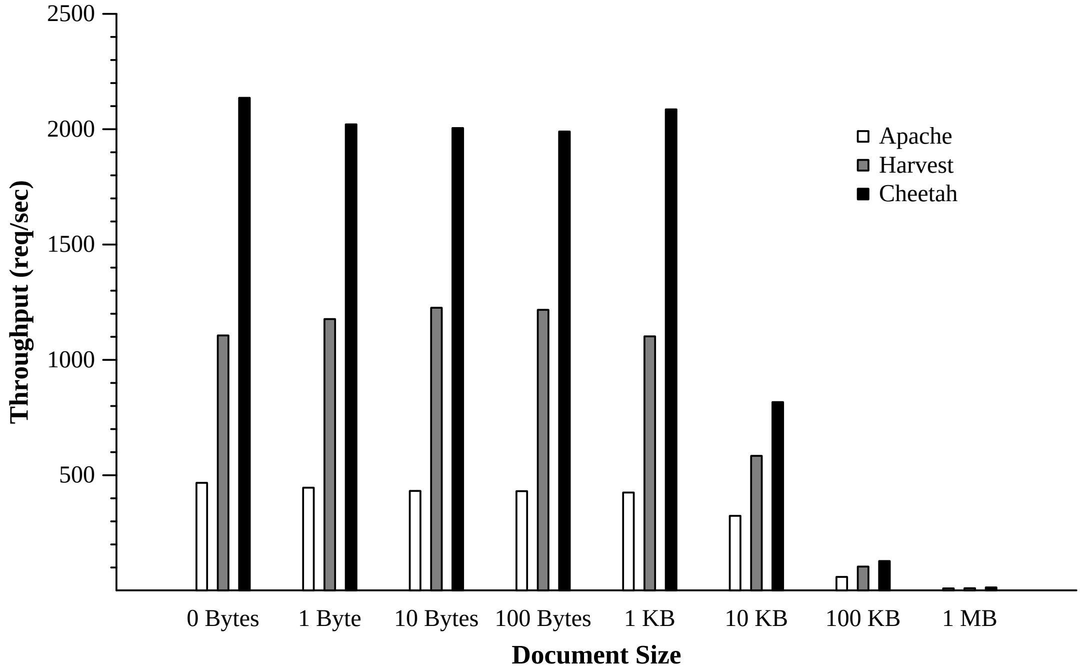

Figure 4-1: Effect of document size on throughput for several HTTP/1.0 servers. **Harvest** represents the Harvest proxy cache, version 1.4.pl3. **Apache** represents the Apache web server, version 1.1.3. **Cheetah** represents Cheetah running under Linux, using DPF for network access.

stance, with 10 KB files on Xok, Cheetah leads Harvest by approximately a factor of 4. This effect is most likely due to Xok's copyless interface to network writes from the buffer cache, and is likely to be addressed in later work. A more dramatic indication of the wins of a copyless write interface is in the maximum bandwidth of the two versions of Cheetah. On Xok, Cheetah achieves 29.3 MB/s. On Linux, this is more than halved, at 12.8 MB/s.

Next, benchmarks are rerun with the number of connections per client machine varied. The results, summarized in figure 4.2.1, demonstrate the effects a changing number of simultaneous connections. In essence, the number of client connections has little impact on the throughput of any of the measured servers.

The effect of CPU load is simulated on the server by running the original benchmark while running a simple script to waste CPU time. The results are summarized in Table 4.2.1. The data, however, requires some explanation.

The CPU wasting script is always run for the duration of the benchmark. It is

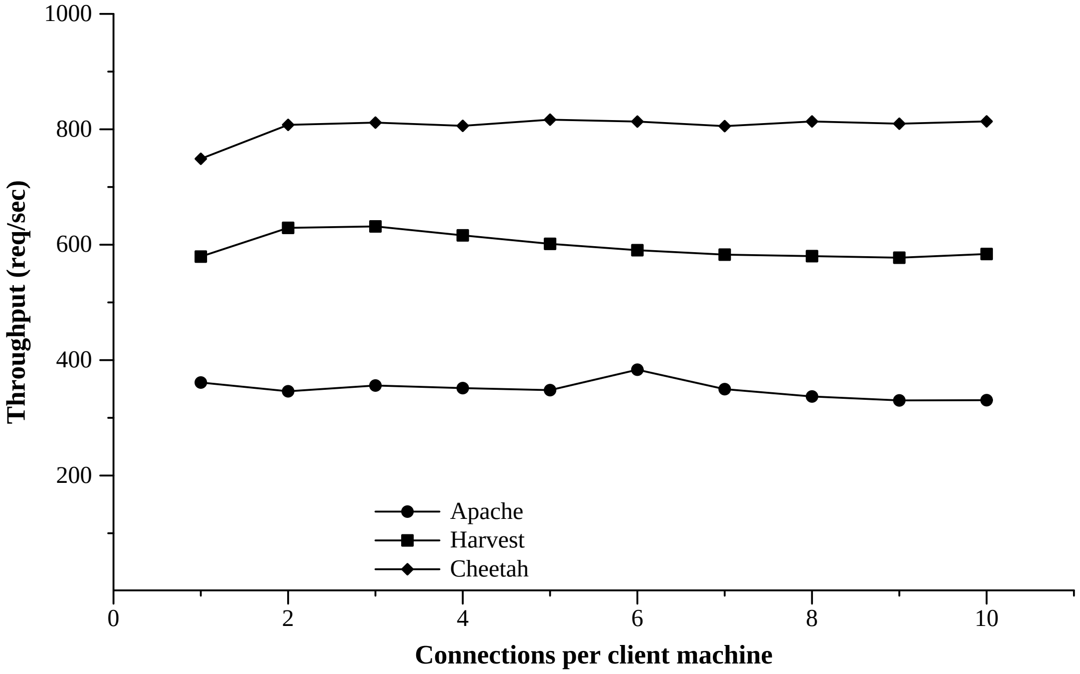

Figure 4-2: Effect of simultaneous connections on throughput. Servers are the same as previous experiments. Document size is 10 KB. Connections refers to the maximum number of simultaneous connections from each of three client machines.

	Idle	Slowdown
Apache	34.6%	26.6%
Harvest	49.2%	49.9%
Cheetah	49.6%	51.1%

Table 4.3: Effect of CPU usage on throughput. Servers are the same as is previous experiments. **Idle** represents the relative progress of a CPU intensive application while running at the same time as the server is serving documents vs. running on a quiescent system. **Slowdown** represents the percent decrease in throughput for the servers when serving pages while the CPU intensive job is running.

restarted at the beginning of each run to compensate for the Unix scheduler's tendency to lower the priority of CPU bound processes. In addition to running throughout the entire length of a benchmarking run, the script outputs a progress count after it has been running 10 seconds (all runs take longer than 10 seconds). The first column shows how far the script progressed in 10 second while running in tandem with the benchmark, as opposed to running alone for 10 seconds. This is a measure of idle time; larger number indicate more available time for other processes while the server is running. The second column indicates how badly throughput has been degraded by the increased CPU load. Higher numbers indicate increased degradation.

These results demonstrate that Cheetah on Linux degrades under load nearly identically to Harvest, while handling load considerably better than Apache. It may be speculated that, to some extent, these numbers overstate the ill effects of load on Cheetah. Since TCP handling is normally done at interrupt time. All processes are charged equally for the CPU time required to decode incoming packets. However, in the case of Cheetah, TCP is handled explicitly in user space. Thus, Cheetah's network utilization is charged exclusively to it. Arguably, this is a more equitable situation (Druschel make similar observations in [1]), but since the other servers are not operating under it, it may provide a slightly unfair comparison. As exokernel interfaces are integrated into an OS, these issues would be eliminated, as every process would be using a user-space library to handle TCP.

4.2.2 XN

XN provides functionality that is currently unavailable on Linux. Therefore, unlike DPF it is difficult to simply choose an application making heavy use of the facility on Linux and compare its performance to a similar application written to use XN. XN is intended to provide low level access to a disk, including the ability to specialize on disk layout, an optimization that a general purpose file system does not allow. The mere *existence* of this feature is a significant result, but it can not be compared directly to the status quo numerically.

The closest to which a highly specialized application may approach this flexibility

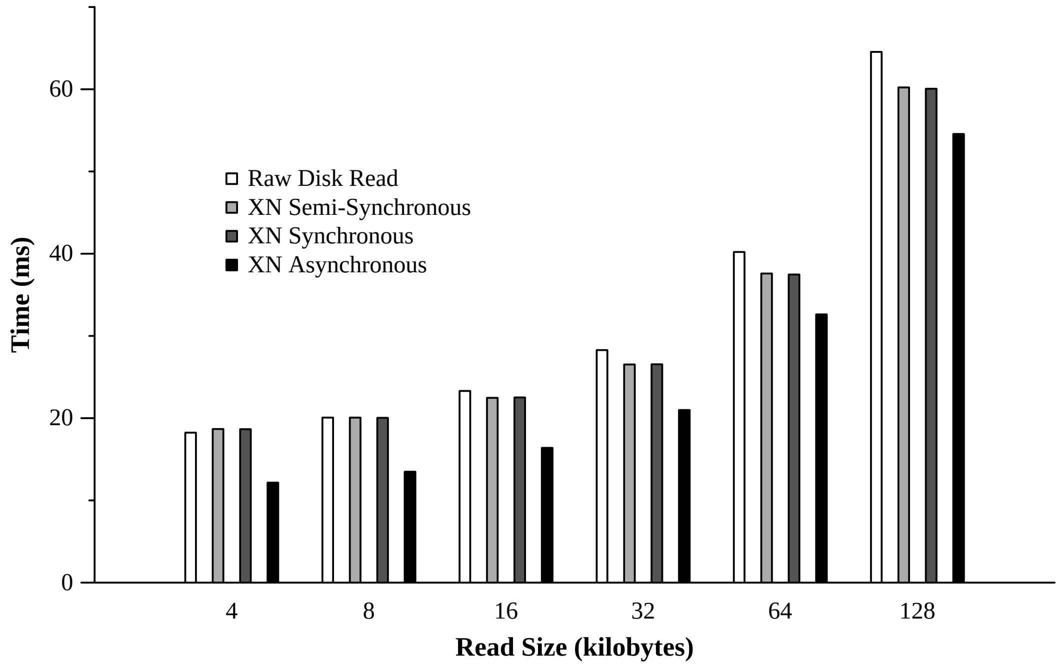

Figure 4-3: Raw Disk Reads vs. XN. 4 MB of data is read using various request sizes. Each request begins at a random location on disk. **Raw Disk Read** represents the use of `read` on a file descriptor for the raw SCSI disk. **XN Semi-Synchronous** represents the use of XN in two steps, a call to read a contiguous group of blocks, followed by calls to wait for the read to complete. **XN Synchronous** represents using XN to read blocks, specifying that the read should not return until completed. **XN Asynchronous** represents using XN to request reads of all 4 MB of data, then waiting for all requests to complete.

on a Posix system is by the use of raw access to a disk device. This allows reads and writes to any block on the disk. This is less satisfying, since it foregoes all protection and concedes complete control to the application, however, it is the only avenue available. Further skewing a direct comparison, XN provides an asynchronous interface that Posix lacks.

Despite these caveats, a comparison of disk access on a raw partition with access provided by XN is presented. For each benchmark, and for each request size, 4 MB of data is transferred. For instance, in measuring 16 KB write performance, 16 KB of data is written to 256 random (block aligned) locations.

Results are summarized in Figures 4-2 and 4-3. The experiment compares raw disk access to access using XN in various ways. The first two XN measurements show the relative performance of using XN for reads compared to raw access. In both case

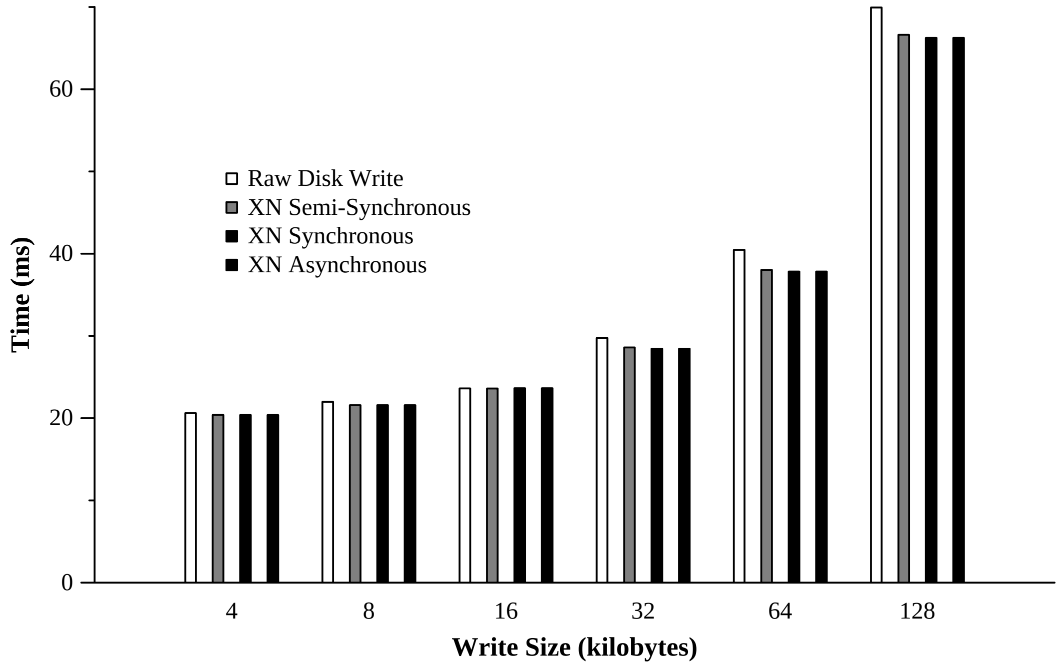

Figure 4-4: Raw Disk Writes vs. XN. Experimental setup is identical to Figure 4-2, with the exception that writes, rather than reads, are performed.

XN is used synchronously. The difference between the first two XN measurements is whether the the application's interface to XN is explicitly synchronous or makes an initial asynchronous request to XN then immediately waits on the supplied channel for the request to complete. This measures the overhead of notification. The final XN benchmark attempts to demonstrate a simple advantage of XN's flexible interface. In that test, all requests (for 4 MB total, no matter what the request size) are made asynchronously. After all requests are made, the test program waits for all to complete. This allows some latency hiding, though real applications with more interesting workloads may be able to use this interface even more profitably.

One caveat to keep in mind is that XN's numbers for synchronous requests only beat raw disk requests because of the lack of copying from/to the buffer cache. If the data must be copied out to be manipulated (as is, unfortunately, the case in the current system), the results are identical. This is to be expected, of course. There is no way that XN is going to be *faster* than raw disk access, unless that raw disk access were implemented exceedingly badly, which would make the result somewhat meaningless anyway.

Two results are important to note. First, the difference between explicitly asking XN to operate synchronously and asking for asynchronous access followed immediately by waits is lost in the noise. This indicates that the channel notification technique has vanishingly small overhead. Second, even in this very limited example of latency hiding, performance is measurably increased. This argues favorably for the utility of an asynchronous interface to the disk.

4.3 Flexibility

4.3.1 DPF

In terms of the flexibility to send and receive packets of the user's choice, DPF on Linux is identical to DPF on Xok. There are absolutely no constraints on the contents of outgoing packets, and incoming packets may be selected with the exact same "little language" for constructing filters. This means that DPF on Linux can be freely used to implement any protocol specializations the Xok's DPF enables.

However, considering a wider definition of flexibility — one that includes the flexibility to control *all* possible resources — Linux's DPF is missing some options that Xok provides. For the most part, these are discussed at the end of Section 3.2, which details how and why these options may be useful. Essentially, they boil down to the fact that Linux's DPF interface for data is streamed through a file descriptor. This means that some useful optimizations, those that require more explicit application management of memory used to buffer packets, are impossible. This includes zero-copy disk to network transmits, avoiding duplicate buffering in the network and disk cache, and fast transmits of identical data to multiple recipients.

4.3.2 XN

The situation with XN is very much like DPF. At a high-level, all of the flexibility of Xok's XN is now available under Linux. Blocks may be allocated as the application intends, different applications may share data that they both know how to use correctly, and XN's consistency guarantees preserve this protection across reboots.

However, there are caveats. XN on Linux does not use the normal Linux buffer cache. This was a decision of expediency in the face of a complicated interface, not a measure of what is "right". Because of this decision, there is no mechanism to map XN buffer cache pages into a process's address space. Thus, one advantage of XN, a copyless interface, its lost.

Chapter 5

Conclusions

The experience of migrating exokernel interfaces to Linux has produced some valuable insights that should be of general applicability to any such project. The issues encountered have more to do with exokernel interfaces clashing with high level OS abstractions than with particular difficulties with Linux specifically.

The first lesson, perhaps unsurprising, is that it pays to be intimately familiar with the OS to be modified. Exokernel's assume simple interfaces to resources, but a conventional operating system may (after long, slow feature accretion) have fairly complex interfaces to even fairly simple abstractions. Further, even if the interface isn't complex, it may still be challenging to alter the implementation. Performance critical systems may be highly optimized for the high level interface that they export. Although the high level interface may be built upon simpler primitives, it may be difficult to tell if those primitives may be used in novel ways, that is in ways other than the ways they are used by the current system.

None of this is surprising, of course. Operating systems are well known for their complexity. The insight actually comes by comparison. Exokernel interfaces are generally *not* complex. Despite the fact that XN has been called somewhat complex, this is more an issue of internalizing the concepts behind XN's protected sharing of the disk. The actually abstractions are fairly simple - disk blocks, types, and notification.

In an effort to integrate exokernel abstractions into a given OS, it would appear that it is far more important to be an expert with that OS than with the exokernel implementation. This is fortunate, since the reverse would be problematic for any such project, as it is likely to be developers of the OS, rather than exokernel researchers, carrying out such work.

Fortunately, some complicated kernel abstractions can be completely bypassed — this is an important exception to the need for developers who are intimately familiar with the target OS. This was the case with much of DPF which almost entirely avoids the more complicated parts of Linux's network interface. All that was required was a single entry point from all network drivers, which any OS is sure to have. From there, DPF takes over and the network subsystem never sees the packet. TCP/IP development can migrate into user level libraries — easier to experiment with, easier to specialize, and easier to debug.

A final lesson about implementation difficulties concerns the way one should approach an exokernel. It is all too tempting to see performance measurements for an exokernel subsystem and want those results in another, more mature operating system. The obvious approach, then, is to migrate that subsystem to the mature system. This, in fact, is exactly the approach taken in this thesis with XN and DPF. Unfortunately, some of the ease of implementation of exokernel subsystems rests upon the simple interfaces throughout the entire kernel. Specific examples that hampered progress include the lack of a simple interface to the buffer cache and easily decomposable memory management primitives. Those subsystems, though less obviously appealing, would make further development much easier.

To some extent, it is an issue of overall goals. If the final goal of a project is fast, flexible networking, then focusing strictly on DPF may make sense. Where mismatches exist, they can be corrected. However, if the eventual goal of a project is to migrate an entire OS toward an exokernel design, a bottom-up approach, in which important, basic exokernel subsystems are implemented first, will shorten total development time.

Although it may seem that migrating an existing interface to a new system would involve little design, developers would be wise to expect difficulties that require new approaches on the target OS. Generally, these come from the desire to make exokernel interfaces "compatible" with Unix abstractions. This was relatively easy in the case of DPF as a new socket type, but required the introduction of the channel for XN. Though channels are by no means a breakthrough in Computer Science, a number of other interfaces were rejected as unsuitable. The most common reason was either an implementation that seemed too complex for the benefit, or, upon reflection it was realized that the interface could not work efficiently on a Unix-like system. The point is that in an effort to create a gradual migration, there will always be edges between the old and the new that will not fit together.

Appendix A

XN API

Disk Block Allocation/Deallocation	
`sys_xn_alloc`	Allocate disk blocks, record the allocation in the parent.
`sys_xn_free`	Free disk blocks, record the change in the parent.
`sys_xn_swap`	Atomically swap the parents of two sets of disk blocks.
`sys_xn_move`	Change the parent of a set of disk blocks.
`sys_db_find_free`	Find a free range of blocks for later allocation.

Memory Allocation/Deallocation	
`sys_xn_bind`	Associate physical memory (buffer cache entry) with a disk block.
`sys_xn_unbind`	Free the physical memory (buffer cache entry) associated with a disk block.

Reading/Writing	
sys_xn_writeb	Copy bytes into a disk block's buffer.
sys_xn_readb	Copy bytes out of a disk block's buffer.
sys_xn_readin	Read disk blocks into pre-allocated buffer cache entries.
sys_xn_writeback	Commit a buffer cache entry to disk.
sys_xn_read_and_insert	Read disk blocks into the buffer cache (allocates the entries).

Roots	
sys_install_mount	Create a persistent, named root for an application level filesystem.
sys_xn_mount	Reads in the disk block of a named root.

Types	
sys_type_import	Creates a new named, persistent type.
sys_type_mount	Reads in named, persistent type from disk.
sys_xn_read_catalogue	Allows applications to view the type catalog
sys_xn_insert_attr	Associate a type with a buffer cache entry.
sys_xn_read_attr	Copy a buffer cache entry's type information into user space.
sys_xn_delete_attr	Delete a mapping from buffer cache entry to type.
sys_xn_set_type	For unions, set the actual type.

Bookeeping	
`sys_xn_init`	Initializes XN, checks disk for corruption.
`sys_xn_shutdown`	Cleanly shuts down XN. Writes out XN's superblock, freelist, type catalog, etc.
`db_t sys_root`	Find XN's superblock.

Bibliography

[1] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, October 1996. USENIX Assoc.

[2] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, Washington, May 1995.

[3] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of the ACM SIGCOMM'96 Conference on Communication Architectures, Protocols, and Applications*, pages 53–59, Stanford, California, August 1996.

[4] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[5] Gregory R. Ganger and M. Frans Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. In *Proceedings of theUSENIX Technical Conference*, pages 1–17, Anaheim, California, January 1997.

[6] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Brice no, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti,

and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, St. Malo, France, October 1997.

[7] K. Mackenzie, J. Kubiatowicz, A. Agarwal, and M.F. Kaashoek. FUGU: implementing translation and protection in a multiuser, multimodel multiprocessor. Technical Memorandum MIT/LCS/TM503, MIT, October 1994.

[8] Thomas Pinckney. Operating system extensibility through event capture. Master's thesis, Massachusetts Institute of Technology, 1997.