Fast Operating System Emulation

on an Exokernel Architecture

by

Russell Thomas Hunt

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 23, 1997

© 1997 Massachusetts Institute of Technology. All rights reserved.

Signature of Author

Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by

Gregory R. Ganger
Postdoctoral Associate
Thesis Supervisor

M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Fast Operating System Emulation on an Exokernel Architecture

by

Russell Thomas Hunt

Submitted to the Department of Electrical Engineering and Computer Science

May 23, 1997

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

ABSTRACT

Operating system (OS) emulation consists of running a binary executable intended for a particular OS on a different OS. This ability allows users to run more programs and OS manufacturers to jumpstart new OSs with non-native programs.

There are several OSs that can run binaries compiled for the same processor but for different operating systems. There are also systems that can run multiple OS servers at the same time (e.g. Mach/microkernel architectures). Unfortunately, on these systems, one must either place the emulation code directly in the OS to gain speed or place the code in user space for flexibility and lose direct, and thus quick, access to OS internals.

This thesis describes how one can take advantage of the exokernel architecture to make OS emulation both flexible, in the sense that it runs in user-space, and quick, in the sense that it can run fast. This combination has not been achieved on monolithic or microkernel architectures.

To demonstrate the approach, this thesis describes a prototype emulator (XPOSE) that runs on an Intel-based exokernel (XOK) with a UNIX-like library OS (ExOS) and can run many programs compiled and linked for the monolithic OpenBSD UNIX OS. All emulation code is in user space and, because the emulator is linked to the library OS, it can access OS internals. Measurements show that XPOSE runs OpenBSD programs with very little overhead.

Finally, the mechanisms XPOSE uses to intercept system calls makes it useful for debugging and performance analysis. Statistics gathered with XPOSE indicate where effort should be focused to reduce the number of kernel calls per library OS call.

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

Thesis Supervisor: Gregory R. Ganger
Title: Postdoctoral Associate
Acknowledgments

I would like to thank XOK and ExOS experts David Mazieres, Tom Pinckney, and Hector Briceño, my advisors Greg Ganger and Frans Kaashoek, Costa Sapuntzakis for helping me understand WINE. Also, I would like to thank DARPA for helping to fund this project. Most of all, I would like to thank Mom, Dad, and Moosey.

I dedicate this thesis to my parental units.
## Contents

1 Introduction .................................................................................................................. 8  
1.1 The primary reasons for emulating ........................................................................ 8  
1.2 Other reasons for emulating .................................................................................... 9  
1.3 Ways to emulate an OS ............................................................................................ 10  
1.4 The rest of this thesis .............................................................................................. 10  

2 OS Emulator Design and the Exokernel Architecture ............................................. 12  
2.1 What is an exokernel? ............................................................................................... 12  
2.2 OS emulation on an exokernel ............................................................................... 13  

3 XPOSE, a Prototype OS Emulator ............................................................................. 16  
3.1 Virtual memory layout problems and loading the emulated program .................. 16  
3.2 OS call emulation .................................................................................................... 18  

4 Performance Results .................................................................................................... 20  
4.1 Execution environment ............................................................................................ 20  
4.2 Microbenchmarks .................................................................................................... 20  
4.3 Application Performance ....................................................................................... 22  
4.4 Implication: system call overhead insignificant .................................................... 23  

5 Analysis of Calls between ExOS and XOK ............................................................... 25  
5.1 Analysis ................................................................................................................... 25  

6 Related Work ............................................................................................................... 29  
6.1 Mach/microkernels ................................................................................................. 29  
6.2 DOS Emulation: Windows NT, Windows 95, OS/2 .............................................. 30  
6.3 Windows Emulation: WINE, Willow, WABI ..................................................... 31  
6.4 The Intel Binary Compatibility Specification (iCBS) ........................................... 32
Chapter 1

1 Introduction
Operating system (OS) emulation involves running a binary executable intended for a particular OS on a different OS. This ability allows users to run more programs and OS manufacturers to jumpstart new OSs with non-native programs. Previous solutions to this problem have been either slow or inflexible. This thesis describes how one can take advantage of the exokernel architecture to make OS emulation both fast and flexible.

The three main contributions of this thesis are:
1) a new approach to OS emulation in user space based on the exokernel architecture,
2) a demonstration of the simplicity and speed of an actual prototype emulator built on an exokernel architecture,
3) and an analysis of statistics about how often a library OS calls into its kernel, gathered using the prototype emulator.

The remainder of this chapter examines, in more detail, why OS emulation is useful and gives some mechanisms for emulation. The discussion includes a high level description of the approach taken in this thesis. Finally, there is an outline of the rest of the thesis.

1.1 The primary reasons for emulating
Both users and operating system manufacturers benefit from an operating system’s ability to run non-native binaries. Users benefit for at least the three following reasons: Suppose, for example, there is a user with one primary application. Normally, such a user would be required to use the OS for which that application was written. However, if that OS can be emulated, the user can use a cheaper OS, a more reliable OS, or a faster OS. Most importantly, the user is no longer constrained. Also, if a user has more than one primary application, but each requires a different OS, then he no longer has to have multiple computers or reboot every time he wants to run a different application. Third, if a user's application is no longer supported by its manufacturers, and thus unlikely to ever be ported to another OS, he can still run it under emulation.
Manufacturers of OSs also benefit from OS emulation because an OS that can support a larger application base has access to a larger potential base of users. Introducing a new OS is potentially difficult because users are often more concerned about certain applications on their current OS than they are about their OS. Even if they would like a faster, more flexible, more reliable operating system, they will not switch to the new OS unless it runs their favorite applications. Therefore, the new OS has a better chance of surviving and thriving if it can start out with a larger base of applications. Even if all of today’s important programs were written with cross-platform development toolkits, new OSs could still make use of emulators until a sufficient number of applications had been recompiled or ported, assuming that the new OS structure was not too difficult or different to incorporate into the toolkit.

1.2 Other reasons for emulating
In addition to the primary benefits of OS emulation, there are several incidental benefits. For example, executing a program on some emulators can give detailed information about what that program is doing. Emulation usually involves intercepting system calls, I/O, and some memory references with a handler. This handler could easily record every system call along with its arguments, for example. This may not be useful to the average user, but it could certainly be used for debugging purposes, reverse engineering and research into performance. However, there are other tools designed specifically for those purposes that are as good as or likely better suited than an OS emulator. Thus, this thesis does not focus on this aspect of emulation.

Emulating an entire system can be important in studying that system, either for understanding its performance or for prototyping [1]. This would involve emulating the OS, processor, peripherals and drivers, and requires inserting code to make detailed measurements. However, emulation to facilitate understanding or study of performance, although interesting, is not a primary goal of this thesis.

Finally, one might want to make a program believe it is running on a kind of hardware that it is not or to correct some flaw in (or make a small modification to) the real OS or hardware. Once again, such emulation in not central to this thesis.
1.3 Ways to emulate an OS
This thesis focuses on how to emulate OSs on their native architectures. In particular, it focuses on emulating a PC version of OpenBSD UNIX on the PC exokernel, XOK.

There are two main aspects of emulating systems calls. The first is the location of the emulator code. One method is to place the code in the kernel, which is appropriate when one monolithic kernel is emulating another (e.g., when OpenBSD emulates NetBSD). Another method is to place the code in user space but in a separate server process. Microkernels use this approach, as described in section 6.1. Finally, the emulator code can be collocated in the user space of the emulated program. XPOSE, the prototype emulator discussed in this thesis uses this method for two reasons: 1) redirection (discussed next) is quicker, and 2) the emulator and its library OS have direct access to the process structures and memory of the emulated program.

The second main aspect of system call emulation involves the transfer of control from the emulated program to the emulator. There are three methods. First, it is sometimes possible to modify a binary executable and replace all system call instructions with instructions to transfer control to another location in the same address space, to the kernel, or to another process. However, variable length machine instructions, dynamic code generation, and self-modifying code make this difficult and faulty. Second, the original instructions can transfer control to the kernel as usual, at which point the kernel must determine whether the process needs emulation and then dispatch appropriately. Third, some processors, such as the Intel x86s, can be configured to route control (to the kernel, another process, or another location in the same process) after execution of the system call machine instruction used by the emulated program. XPOSE takes advantage of this feature to transfer directly to another location in user space, thus reducing expensive kernel and process boundary crossings.

1.4 The rest of this thesis
The remainder of this thesis is organized as follows: Chapter 2 examines emulator design on an exokernel and specifically how the exokernel architecture can be exploited to achieve fast and flexible emulation. Chapter 3 details the implementation of XPOSE (eXokernel Prototype Operating System Emulator), which demonstrates the concepts
discussed in chapter 2. Chapter 4 evaluates the performance of XPOSE compared to native XOK and native OpenBSD with both microbenchmarks and applications. Chapter 5 presents some previously unknown statistics about system calls and kernel calls in the library OS, ExOS. The statistics indicate where effort should be focused to reduce the number of kernel calls per library OS call. Chapter 6 examines related work. Chapter 7 concludes.
Chapter 2

2 OS Emulator Design and the Exokernel Architecture

This chapter briefly describes the exokernel architecture and discusses OS emulator design on an exokernel.

2.1 What is an exokernel?

An exokernel is an operating system kernel that securely multiplexes the hardware, providing flexibility and performance[2]. Standard operating systems have large monolithic kernels that contain everything from TCP/IP networking and filesystem code to virtual memory management. However, an exokernel contains only enough functionality to protect processes and their resources from one another. The rest of the code is contained in a library operating system (libOS) that can be linked to each program (see Error! Reference source not found.). In this way, an application can use abstractions from a standard shared library OS or, if it has some task that needs to be performed particularly fast or which does not fit the standard OS abstractions, it can access the hardware at a lower level.

![Exokernel and Monolithic Architecture Diagram](image)

**Figure 1** – Process-Kernel relationships on the exokernel architecture versus the traditional monolithic kernel architecture.
2.2 OS emulation on an exokernel

OS emulation conceptually takes place on top of the library OS, not directly on the exokernel itself (see Figure 2). However, the exokernel architecture can make an OS emulator easy to develop and fast to execute.

![Figure 2 - Structure of traditional monolithic architecture versus an emulator and emulated program on exokernel architecture. Objects are dependent on and can only access the object(s) on which they directly sit. Kernel space is below the dark line and user space above.](image)

On an exokernel, an OS emulator can run in the same address space as the emulated program. When the emulated program performs a system call as it would for its original OS, control is transferred to the exokernel which then transfers control back to the emulator running in user space. The exact mechanism for doing this varies from processor to processor. For example, on MIPS processors, there is only one machine instruction for performing system calls. Thus, checking a user settable flag would be necessary in order to determine whether the call was a true exokernel system call or whether to bounce control back to a user-specified location in user space. On the other hand, Intel x86 processors have many machine instructions that can be used for system calls and each one can be individually routed to any location, including user space. Thus, as long as the system call instruction used for the emulated OS and the exokernel are different, control can go directly to the emulator without going through the kernel. Otherwise, extra checking would again be necessary.

Adding such rerouting abilities to current OSs would not be difficult. However, an emulator on a standard monolithic OS should stay in the kernel to take advantage of the internal OS structures and functionality. With an exokernel architecture, the emulator should be in user space to take advantage of the library OS and its structures and functionality, as shown in Figure 3 and Figure 4. Generally speaking, an exokernel
architecture can be more efficient on “standard” system calls, such as open or gettimeofday, for three reasons:

1) The library OS version of the system call may not have to transfer control to kernel (which is expensive on some processors).

2) A library OS does not have to verify that arguments which are pointers actually point to valid locations (which is also expensive). This is allowed because the exokernel architecture lets a library OS crash the process it is linked to as long as other processes linked to the same library OS are not affected.

3) A library OS does not have to deal with overhead arising from the interdependencies within the monolithic structure (expensive).

Fortunately, an OS emulator built for an exokernel-based system would still reap the benefits of reasons two and three even if it has to execute the expensive system call machine instruction.

---

**Figure 3** - System call on an exokernel architecture, and a direct kernel call demonstrating the flexibility of the architecture. A system call to the library OS might not call into the kernel, or it might call into it several times.
Figure 4 - An "normal" emulated system call and an emulated system call that does not call the libOS (for example, the `brk` call). A system call to the library OS might not call into the kernel, or it might call into it several times.
Chapter 3

3 XPOSE, a Prototype OS Emulator

This chapter describes a prototype OS emulator built for an exokernel system, including how it initializes and loads the emulator and emulated program. The last section explains in detail the process by which the prototype emulator receives control from and returns control to the emulated program.

XPOSE runs unmodified OpenBSD 2.0 (Intel/PC version) UNIX binaries on the ExOS version 2.0 library OS running on the XOK exokernel for Intel x86 processors. It works by directing the system call machine instruction used by OpenBSD to transfer control to a particular location in the user part of the virtual memory address space. From there, the emulator code takes control. This code emulates the system call and returns any results to the emulated program. There are approximately 155 OpenBSD system calls, of which approximately 90 are partially or fully supported by ExOS and XPOSE. With these system calls, most programs can run. Many simple UNIX utilities have been tested on XPOSE. In addition, the Mosaic web browser executes properly.

The majority of the OpenBSD system calls can be emulated by making direct calls to the ExOS equivalents. However, one interesting call that could not directly call the library OS version is the brk call. It adjusts the “break” or address of the first unmapped page after the program in user space. The library OS version maintains the “break” for XPOSE, but XPOSE must maintain a second “break” for the emulated program.

3.1 Virtual memory layout problems and loading the emulated program

Standard OpenBSD binaries are linked such that their code segment starts on the second page in virtual memory (at address 0x1000). Unfortunately, XOK originally used this location for XOK-specific data and code. Therefore, it was necessary to modify the kernel so that this region could be relocated on a process by process basis. More generally, OS emulation of this sort requires that no virtual address space regions be hard-coded.

There is another conflict between ExOS and XPOSE. ExOS programs are currently linked to be loaded at an 8 megabyte offset (address 0x800000). This can cause problems because the OpenBSD program and XPOSE are running in the same address
space. Therefore, the OpenBSD code and data sections from the binary must fit below the 8 megabytes so that they do not interfere with XPOSE. In addition, the OpenBSD program, near the bottom of the address space, has a heap that grows upward and thus overwrites XPOSE if it reaches the 8 megabyte offset. To solve this space problem (for programs that use more than 8 megabytes of memory) it was necessary to move XPOSE higher into the address space. Because the current binary executable layout for ExOS and the corresponding ExOS code which loads programs have the 8 megabyte address hardcoded into them, it was simplest to link the emulator for a higher address (specifically 0x1900000, the higher the emulator, the more memory the emulated program can use) and then to create a special program (called “run”) to load the emulator at that location. A permanent solution (to eliminate the “run” program) requires that ExOS be able to recognize multiple binary executable formats and load the programs at whatever location they are linked for.

To summarize, here is an example command line invocation of XPOSE running Mosaic (an OpenBSD program for browsing World Wide Web pages):

```
run 0x1900000 emulate Mosaic http://www.mit.edu/
```

The program “run” is loaded at the regular 8 megabyte offset. It loads the emulator program (XPOSE) “emulate” at address 0x1900000, passes on the rest of the command line, and transfers control to XPOSE. XPOSE relocates the special XOK-specific code and data mentioned above to a very high address. Then, it sets up the system call handler, loads the program to be emulated at address 0x1000 and transfers control to it along with any remaining command line options. Figure 5 shows the layout of memory after the emulator has been loaded.
Figure 5 – XPOSE and the emulated OpenBSD program virtual memory layout.

3.2 OS call emulation

Once the emulated program gains control it executes instructions as usual until it gets to the particular instruction which would normally transfer control to the kernel: the system call instruction. Instead of transferring control to the kernel, XOK has been programmed to instruct the processor to transfer control back to a particular address in user space for the instruction which OpenBSD uses as its system call instruction. Before transferring initial control to the emulated program, XPOSE sets up a special handler at the particular address.

Before executing the system call instruction, an OpenBSD program pushes the arguments to the system call onto the stack. It then calls a special system call wrapper, which pushes a return address onto the stack and places the number of the requested system call in a register (specifically, AX). Eventually, the wrapper code executes the system call instruction, at which point the processor flags and a second return address are pushed onto the stack and control passes to the emulator handler (see Figure 6). Here is what the handler does once it gains control:

1) Saves the registers in a structure
2) Saves flags and the two return addresses from the stack into a structure, leaving only the system call arguments on the stack.

3) Uses the system call number in AX to call a function from an array of functions (normally, the ExOS function which corresponds to the system call specified in AX).

4) Upon return from the ExOS function, deals with error conditions such as “File not found” for open by setting certain flags.

5) Restores flags and return address to the stack.

6) Restores registers.

7) Returns from the system call back to the emulated program.

If a program makes a system call that has not been implemented then XPOSE will print out a message indicating the system call and exit. It can also be set to print warnings whenever a system call is made which has not been tested or fully implemented.

---

**Figure 6 - The contents of the stack at the point when XPOSE gains control.**

One of the more useful, but less central, abilities of XPOSE is its debug feature. When run with the proper options, it will create a log file containing all the system calls a program makes along with their arguments and return values. This is accomplished by making all the functions in the array of functions point to a debugging handler which can write out to the log file before and after emulating the system call. Other forms of instrumentation can be applied in the same way.
Chapter 4

4 Performance Results
This chapter details the machine used to make the measurements and discusses both microbenchmark- and application-level performance results comparing native XOK/ExOS to native OpenBSD and emulated OpenBSD. The experiments show that overhead from running OpenBSD programs on XPOSE is minimal.

4.1 Execution environment
All measurements were taken on a 120 MHz Intel Pentium (with PCI bus) machine running either OpenBSD 2.0 (Intel/PC version) UNIX or the XOK exokernel with ExOS version 2.0 library OS. The machine has 64 megabytes of RAM and a Conner CFP1080S SCSI disk drive connected via FAST SCSI-2. The machine was hooked up to a shared 10 megabit/second ethernet with an SMC ethernet card. The level of variation in the test results indicate that any time spent processing broadcast messages was insignificant. All code was compiled with gcc version 2.7.2.1 using "-O6" optimization level.

4.2 Microbenchmarks
This microbenchmarks in this section are intended to measure the overhead involved in transferring control between a process and the OS, be it in the kernel or a library or in XPOSE. In other words, the measurements show the amount of time spent on the "journey" to and from the OS not including the "real" work of the system call. These measurements give an indication as to whether the extra time spent getting to the emulator code is significant.

These measurements were gathered using the following piece of code:

```c
{  
int i;
  for (i=0; i < 1000000; i++)
    getpid();
}
```

This code is used to determine the overhead involved in trapping to kernel space. The getpid system call was chosen because it does so little that it can be treated as a null system call. The current time was recorded (using the gettimeofday system call)
before and after the loop and the difference was used to calculate the elapsed time. Cycle count estimates were calculated by multiplying the total time by the speed of the test machine (120 MHz) and dividing by the number of times the loop was executed (1000000 times).

On the test machine using XPOSE (with the loop code compiled for OpenBSD), the loop takes 850000 microseconds or roughly $1 \times 10^8$ cycles to complete, which gives 100 cycles per loop. Table 1 identifies where the 100 cycles are spent. The trap instruction (31 cycles) and trap return instruction (10 cycles) are expensive (these two numbers came from [3]). The other values in the table were calculated with microbenchmarks using the gettimeofday plus large loop method.

Table 2 compares the overheads of the various mechanisms for transferring control to an OS or XPOSE. An emulated system call that makes one call to XOK will still have less overhead than a native OpenBSD system call, whereas a non-emulated call on ExOS could have up to two calls to XOK before exceeding the overhead on an OpenBSD system call. The large OpenBSD system call overhead appears to be caused by an inordinate number of checks that are required for every call. From this micro-perspective, at least, the additional emulator overhead appears to be moderate.

<table>
<thead>
<tr>
<th>Cause</th>
<th>approx. cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop overhead</td>
<td>6</td>
</tr>
<tr>
<td>between “call getpid” and actual trap to system (OpenBSD places some wrapper code around each system call.)</td>
<td>17</td>
</tr>
<tr>
<td>int $0x80 (system call instruction)</td>
<td>31</td>
</tr>
<tr>
<td>overhead of adjusting stack, etc. plus calling library OS getpid</td>
<td>33</td>
</tr>
<tr>
<td>iret (return from trap)</td>
<td>10</td>
</tr>
<tr>
<td>unknown/error</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

Table 1 - Breakdown of system call overhead on XPOSE.
System overhead (cycles)
OpenBSD – system call 270
XOK emulator (XPOSE) – system call 100
XOK native – kernel call 127
XOK native – libOS call 11

Table 2 - Comparison of system call overheads. libOS call: What normal OSs call system calls are really regular procedure calls to the library OS on XOK, which is why the libOS call overhead is so little.

4.3 Application Performance
While the microbenchmark results are encouraging, application performance is much more important because system call overhead does not matter for programs that make few system calls. Unfortunately, comparing an application running in all three situations (native OpenBSD, emulated OpenBSD, and native XOK/ExOS) is currently difficult because ExOS is largely untuned. In particular, the input/output subsystems that many system calls use are substantially different than OpenBSD’s. For example, the NFS code on ExOS is slow and the local filesystem is designed differently from standard UNIX filesystems. Thus, comparisons between running times of applications on ExOS and OpenBSD are difficult to interpret and best addressed in a more comprehensive study. Instead, measurements have been taken for several programs running on native ExOS/XOK and under XPOSE. Originally, this information was meant to give an idea of the performance effect of the extra system call overhead on the application. However, as explained later, this is unlikely to be the cause of the differences in running times seen in Table 3.

The following list explains the applications tested and their inputs (the test files were taken from a directory off the author’s home directory):

**cksum** computes and outputs a CRC and byte count for 65 files of various sizes (21 megabytes total).

**compress** reads a file, writes out a compressed version, and deletes the original (65 files of various sizes, 21 megabytes total, uncompressed).

**uncompress** reads a file, writes out a decompressed version, and deletes the original (65 files of various sizes, 21 megabytes total, uncompressed).

**cp** copies 65 files of various sizes, 21 megabytes total.
**diff** compares 2 equivalent files (not related to the other 65 files) of 10.5 megabytes each.

**encrypt** takes a 500 kilobyte file from stdin and sends an encrypted version to stdout (uses -m option).

**grep** searches 64 files of various sizes (18 megabytes total) for a substring and prints out lines from the files that contain the substring. One file of the 65 had to be removed because grep on ExOS uses a lot of memory and XOK/ExOS currently has no page swapping mechanism.

The input files for **cksum**, **diff**, **encrypt**, and **grep** fit within the buffer-cache.

<table>
<thead>
<tr>
<th></th>
<th>native XOK in secs (std. dev.)</th>
<th>XPOSE in secs (std. dev.)</th>
<th>% slowdown</th>
<th>total number of emulated system calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cksum</strong></td>
<td>5.44 (0.018)</td>
<td>5.54 (0.024)</td>
<td>1.8</td>
<td>1593</td>
</tr>
<tr>
<td><strong>Compress</strong></td>
<td>65.63 (0.34)</td>
<td>66.01 (0.96)</td>
<td>0.6</td>
<td>9697</td>
</tr>
<tr>
<td><strong>Uncompress</strong></td>
<td>40.58 (0.27)</td>
<td>42.53 (0.17)</td>
<td>4.8</td>
<td>9583</td>
</tr>
<tr>
<td><strong>cp</strong></td>
<td>16.60 (0.11)</td>
<td>16.70 (0.12)</td>
<td>0.6</td>
<td>1205</td>
</tr>
<tr>
<td><strong>diff</strong></td>
<td>2.09 (0.087)</td>
<td>2.17 (0.0093)</td>
<td>3.8</td>
<td>5146</td>
</tr>
<tr>
<td><strong>encrypt</strong></td>
<td>17.34 (0.021)</td>
<td>20.16 (0.0092)</td>
<td>16.3</td>
<td>97</td>
</tr>
<tr>
<td><strong>grep</strong></td>
<td>1.65 (0.015)</td>
<td>1.68 (0.0088)</td>
<td>1.8</td>
<td>360</td>
</tr>
</tbody>
</table>

Table 3 - Comparison of several applications. Each test was run ten times and the results averaged.

### 4.4 Implication: system call overhead insignificant

For most of these programs XPOSE’s overhead is minimal. Even for those programs that approach 10,000 system calls total, the overhead of XPOSE calls (100 cycles/call) sums to less than 1/100th of a second (on a 120 MHz machine), which is minuscule given the total running times.

There are several possible explanations for the slowdown seen in Table 3. First, the libc part of the library OS with which the native ExOS programs were linked was compiled with the highest (gcc "-O6" option) optimizations while the libc with which the OpenBSD programs were linked was compiled with only default optimizations. Thus, the native XOK/ExOS programs should be expected to run faster. Unfortunately, programs linked with a non-optimized version of ExOS were unstable (i.e., ExOS simply does not work correctly when compiled without "-O6") so that comparisons were not possible.
Second, the libc library used with ExOS is not exactly the same as the OpenBSD version. One or the other version could be less well tuned than the other. Some test runs using the Pentium performance counters indicate that the slowdown of encrypt (the test program with the greatest slowdown) may be caused by an inordinate number of code cache misses and an increased number of executed instructions. Other possibilities include memory layout conflicts and misaligned data accesses. Since encrypt makes very few system calls, all of this contention must be occurring in the main section of the program or in the standard libraries, rather than in XPOSE.
Chapter 5

5 Analysis of Calls between ExOS and XOK

XPOSE was not initially intended to provide performance analysis or debugging facilities. However, this chapter presents some statistics which were easy to gather using XPOSE and which are not available elsewhere. The findings suggest that while many system calls do not enter the kernel, the most “popular” system calls enter the kernel often. At the same time, the data indicate that decreasing the number of kernel calls may be simple.

5.1 Analysis

Interestingly, XPOSE has given the first indication of just how many kernel calls an ExOS system call makes. Although ExOS could be instrumented to record such information, it has not been. However, because XPOSE creates a single point through which all system calls must go, recording such information is simple. It has been able to record which and how many XOK calls are made from each system call in a given run. The results indicate that there are many system calls that never or rarely enter the kernel. However, the more popular system calls, such as read, write, open, break, and unlink tend to enter the kernel quite often, in one particular case as many as 83 times on average per system call. The most popular kernel call for the tested utilities appears to be the virtual memory mapping call, followed by buffer-cache manipulation calls. Although this information does not require an emulator to discover, it shows that, especially in initial development, an emulator’s debugging and performance monitoring abilities can be useful, even if only as verification. In addition, XPOSE measures the number of kernel calls per system call as seen by the user. So even if open calls break, any kernel calls break makes will be attributed to open. Without XPOSE, differentiating between intra-ExOS system calls and extra-ExOS system calls may have been more difficult.

One potential win for library OSs is that they are not required to enter the kernel, which is expensive on some machines. The detailed system/kernel call data XPOSE creates gives an indication of where system call slowdown may be occurring, and thus where efforts at tuning should be focused. For example, system calls break and unlink both appear to take more time and make more kernel calls during compress
and *uncompress* than should be necessary. Though the data varied substantially from program to program, Table 4 and Table 5 are two representative examples of the data collected.

<table>
<thead>
<tr>
<th>Name of OS Call</th>
<th>Name of Kernel Call</th>
<th>Number of OS Calls</th>
<th>Number of Kernel Calls</th>
<th>Kernel/OS Calls</th>
<th>Number of Yields</th>
<th>Approx. CPU ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>self_insert_pte</td>
<td>5235</td>
<td>8666</td>
<td>1.7</td>
<td>657</td>
<td>296</td>
</tr>
<tr>
<td></td>
<td>bc_read_and_insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_write_dirty_bufs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>self_insert_pte</td>
<td>3166</td>
<td>12021</td>
<td>3.8</td>
<td>170</td>
<td>205</td>
</tr>
<tr>
<td></td>
<td>bc_insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_read_and_insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_write_dirty_bufs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>open</td>
<td></td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>close</td>
<td></td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>unlink</td>
<td>self_insert_pte</td>
<td>65</td>
<td>5405</td>
<td>83.2</td>
<td>70</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>bc_write_dirty_bufs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td></td>
<td>62</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>chown</td>
<td></td>
<td>62</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>self_insert_pte</td>
<td>395</td>
<td>13251</td>
<td>33.5</td>
<td>10</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chflags</td>
<td></td>
<td>62</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>readlink</td>
<td></td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ed0xmit</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>utimes</td>
<td></td>
<td>62</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>stat</td>
<td>self_insert_pte</td>
<td>195</td>
<td>98</td>
<td>0.5</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

26
Table 4 – Breakdown of kernel calls per system call for a single run of `compress`. The total number of system calls was 9697. The `self_insert_pte` kernel call associates a virtual memory page with a physical memory page. Blank cells are either N/A or can be considered 0.

<table>
<thead>
<tr>
<th>Name of OS Call</th>
<th>Name of Kernel Call</th>
<th>Number of OS Calls</th>
<th>Number of Kernel Calls</th>
<th>Kernel/OS Calls</th>
<th>Number of Yields</th>
<th>Approx. CPU ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read</td>
<td></td>
<td>424</td>
<td>10482</td>
<td>24.7</td>
<td>221</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>self_insert_pte</td>
<td></td>
<td>9918</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_write_dirty_bufs</td>
<td></td>
<td>188</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td>376</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write</td>
<td></td>
<td>362</td>
<td>25985</td>
<td>71.8</td>
<td>285</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>self_insert_pte</td>
<td></td>
<td>20126</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_insert</td>
<td></td>
<td>5091</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_read_and_insert</td>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_write_dirty_bufs</td>
<td></td>
<td>216</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td>512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>open</td>
<td></td>
<td>130</td>
<td>433</td>
<td>3.3</td>
<td>108</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>self_insert_pte</td>
<td></td>
<td>118</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_read_and_insert</td>
<td></td>
<td>41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc_write_dirty_bufs</td>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td>210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close</td>
<td></td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td></td>
<td>19</td>
<td>9</td>
<td>0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>self_insert_pte</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getuid</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>readlink</td>
<td></td>
<td>1</td>
<td>6</td>
<td>6.0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ed0xmit</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wkpred</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>umask</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5 - Breakdown of kernel calls per system call for a single run of cp. The total number of system calls was 1205. The self_insert_pte kernel call associates a virtual memory page with a physical memory page. Blank cells are either N/A or can be considered 0.

The “wkpred” kernel calls are a kind of sleep call that can be used while waiting for I/O. “yield” is not a system call, but together with the “wkpred” count, gives an idea of how much time is spent blocked waiting for I/O. The “CPU ticks” column gives a crude approximation of the total amount of time spent in the system calls. It is calculated by taking the difference between the machine’s “tick counter” at the beginning and end of the system call.

The most obvious point from the two tables above is that the system calls make a lot of kernel calls in the process of mapping memory. In fact, the self_insert_pte call only maps one page at a time. Even though there exists a call which will map in ranges of pages (and ranges are often involved, for example when using malloc, or placing a lot of data on the stack), it is not being used when it should be. Thus, the ExOS implementers now know where to look to fix the excessive kernel call problem.

This data gives a first look at the interaction between library OS calls and kernel calls. While the number of kernel calls is high for some system calls, given the XPOSE data, the problems can likely be fixed with reasonable effort.
Chapter 6

6 Related Work

6.1 Mach/microkernels

The distinguishing feature of microkernels (of which Mach is an example [4]) is their server oriented nature (see Figure 7). Like exokernels, microkernels have a small kernel and a larger OS that runs in user space. However, on a microkernel, the OS runs in a separate process and is called a server. System calls on some implementations bounce from the kernel back to an emulation library and then IPC over to the server, while others go directly to the server. In any case, moving between processes can be relatively expensive, depending on the processor. While part of a microkernel’s purpose is to provide an interface that facilitates emulating OSs, its intention is not to run binaries from other OSs.

![Microkernel Architecture](image)

Figure 7 – Process-Kernel-OS server relationships on the traditional microkernel architecture. Other configurations are possible. For example, the filesystem, network code, and virtual memory management code could each be separate servers.

To run a binary for an OS that the microkernel does not have a server for would require one of two solutions: 1) An emulator must run in the user space of an available OS server. 2) An emulator must run in the process’ address space. The first case would be
fast but harder to maintain and the second would be easier to maintain but lack the direct access to the OS server internal structures and facilities.

In contrast, the exokernel architecture places what would be in the microkernel OS server into libraries in the caller's address space. The intent of XPOSE is to provide an interface to the OS libraries from a local address space system call handler. In essence, the handler is the emulator (it emulates some OS), and it is built on top of whatever library OS the handler is linked with. Finally, the exokernel provides the trapping and protection mechanisms.

To summarize, the microkernel architecture is designed to run a binary compiled specifically for an OS "emulated" on top of the microkernel (see Figure 8). The purpose of XPOSE is to take a binary which was never intended for XOK and execute it without modification. In addition, the mechanisms used in and by XPOSE are more efficient than the standard microkernel approaches.

![Figure 8 - A system call on a traditional microkernel architecture.](image)

6.2 DOS Emulation: Windows NT, Windows 95, OS/2
OSs that operate on processors derived from the Intel 80386 can easily emulate DOS and standard mode Windows, for example, by using a special mode on those processors for emulating the Intel 8086 chip [3]. The intent of the mode is to forward all protected instructions to handlers. In addition, each DOS program can have its own address space, thus maintaining the protection of the entire system. Because the old chips could only access about one megabyte directly, the intent of the special mode is to have the handler
code higher in the address space of the virtual session. In a sense, having the handler code in the same address space as the binary is similar to the goal of XPOSE. However, the Virtual 8086 mode is a hardware mode created with the specific intent of facilitating the emulation, through software, of older operating systems like DOS. On the other hand, XPOSE intends to emulate OSs in the normal non-Virtual 8086 mode that was never intended for emulation but happens to facilitate it. Windows NT, Windows 95, and OS/2 are widely distributed OSs that take advantage of the Virtual 8086 mode [5]. Other UNIX-like OSs have also used this mode.

6.3 Windows Emulation: WINE, Willow, WABI
There are many packages that try to emulate Microsoft Windows OSs in various ways. WINE is a Windows Emulator that runs on PC UNIXs [6]. Although still in alpha production, it can run many 16-bit Windows programs and some Win32 programs. Trapping some interrupts is still necessary, but the interface to the Win32 API is in user space, so intercepting calls to it are relatively simple. The difficult part is implementing the complex Win32 API. The Windows implementation of the Win32 API eventually calls into a kernel, but the interface to this kernel is undocumented except to special license holders, thus 32-bit Windows emulators must implement all of Win32, and not just calls to the kernel, which might be as hard or harder anyway.

Because the source to WINE is available, it could be helpful in running Windows programs under XOK. There are three options: one, run WINE under XPOSE; two, port WINE to ExOS, the OpenBSD-like OS currently being developed on XOK; three, use some of the framework of WINE, but re-implement much of it directly on top of XOK. The third option would undoubtedly be more impressive, have better performance and take advantage of the special flexibility of XOK. However, the first or second option would be a quicker route to Windows emulation, both 16- and 32-bit.

The purpose of the Willows Toolkit, created by Willows Software, Inc., is to make cross-platform development of programs that use the Windows API easy [7]. It is intended for developers who want their products to run on many platforms, and thus intends that the code be recompiled for different platforms. However, they do have an
Intel processor emulator as well as something called xwin which is similar to WINE. Still, the main purpose of the Willows Toolkit is not related to XPOSE’s.

Wabi is essentially a commercial version of WINE developed separately by SunSoft, Inc [8] [9]. It runs under Solaris on Intel, SPARC, and PowerPC processors. It is more stable than WINE and so far its purpose is to run 16-bit Windows programs, but the developers do plan to extend support to 32-bit programs. Like WINE, of course, it runs in a single separate address space. In addition, however, it actually translates the Windows API calls to X Windows calls for better performance.

6.4 The Intel Binary Compatibility Specification (iCBS)
iCBS is intended for UNIX-like OSs that run on Intel processors [10]. It seeks to provide a specification which allows a single iCBS conforming binary to execute on any iCBS conforming OS. Although it provides specifications on how files should be structured, installed, even on what media they should be stored, it is chiefly concerned with the system call interface and virtual memory layout. Because part of the purpose of XPOSE is to increase the number of programs which will initially run on ExOS, it may be important to provide an iCBS emulator. Interestingly, the iCBS requires shared libraries be able to load at 0xa000 0000 in the process’ address space. Unfortunately, this very high location is currently in a reserved area in XOK.

6.5 Interposition Agents [11]
Intercepting system calls in user space has many uses besides emulation. [11] mentions several of these: system call tracing, customizable filesystem views, implementing logical devices, transparent data compression/encryption, and transactions. That paper presents an object oriented toolkit for creating interposing agents (the intercepting user code), and it wins by taking advantage of all the standard object oriented ideas. However, this thesis is focused on emulation and, in general, maintaining performance. That paper has a good example of the speed problem user space emulators, or in this case a filesystem trace agent, can have. The author says that the agent (user space) version of the FS tracer caused a 64% slowdown (due to lack of direct access to resources in the kernel) while a similar “monolithic implementation” had only a 3% slowdown. Of course, the non-agent
version required modifications to 26 kernel files. Interestingly, an emulator on an exokernel architecture does not require changing library OS or kernel code and is still fast.
Chapter 7

7 Conclusions
This thesis describes a new approach to OS emulation, which exploits the exokernel architecture to realize both speed and flexibility. XPOSE, a prototype emulator based on this approach, correctly executes many OpenBSD binaries on an exokernel-based system for PCs. Measurements show that the overhead of emulation is very low in the worst case and negligible for most real applications. In addition to increasing the application base for the exokernel-based system, XPOSE has been useful in debugging (both correctness and performance) the ExOS library OS. Finally, this thesis presents previously unknown statistics about how this particular library OS interacts with the kernel in this exokernel-based system.
8 References


