An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel

by

Charles L. Coffing

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 Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 1999

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

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

An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel

by

Charles L. Coffing

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Computer Science

Abstract

This thesis presents the design and implementation of an x86 virtual machine monitor that allows multiple operating systems to run concurrently under xok, MIT's x86-based exokernel. The monitor and modified xok demonstrate how to share several processor-defined structures, such as the GDT, LDT, and IDT, between multiple operating systems. Xok was modified to allow controlled modifications to these structures. Linux was used as the reference guest operating system. The guest operating system is part of the trusted code base, but the monitor itself is not. Xok, the monitor, and the guest operating system coexist on the same pagetable to minimize TLB flushes. Real-mode virtualizations of disk and video devices were done. The resulting monitor was benchmarked against several other solutions, and thoughts for future improvements are presented.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The paradigm of running exactly one operating system per personal computer is being threatened. The idea of running more than one operating system simultaneously is attracting interest, academically and commercially, because there now exists not only the power to do so but also the need. As Gordon Moore noted in a 1965 speech, the transistor counts and processing power of chips tend to double every 18 months. This trend, now known as Moore's Law, has held up well for thirty years and likely will continue in the foreseeable future. Additionally, in recent years the choice in PC operating systems has increased significantly beyond the mass-market standard of Windows. Multimedia-oriented operating systems are appearing and growing in importance, such as BeOS, which was first released on x86 in 1997. Linux, an open-source UNIX clone for personal computers, was first written in 1991 but had an explosion of popularity in 1998, with shipments growing 212% for the year in the server market, according to a study by International Data Corporation. In addition, numerous research operating systems are being or have been developed for the x86, each with their own advantages.

There are numerous reasons for wanting to run multiple operating systems on a single PC. The most obvious reason is an economical one. As PCs become more powerful, an increasing number of uses do not fully tax the processor. Instead of having a separate machine for each task, a new PC should be able to handle all tasks. Having the ability to simultaneously run more than one operating system on a PC makes this consolidation of tasks on a single PC more likely.

Such an arrangement also allows users to focus on using the best application for the task at hand, rather than having to be aware of issues of application availability on a given operating system. This allows the computer to viewed as a more general tool.

Running multiple operating systems on a single PC may also reduce development and support costs, since the migration from a legacy operating system to a more current one can be made smoother. Old and new applications (and operating systems) may be run side-by-side for a period of time to ensure that the new environment is working properly before disposing of the old.

Another motivation is convenience. It is not uncommon to find a developer who prefers to develop code in a UNIX environment, but boots Windows to write or to play a game. Rebooting into another operating system is counter-productive. It interrupts the continuity of work, forces applications to be shut down and restarted, and prevents the machine from being used as a server. Ideally, any application should run on a PC without rebooting, regardless of the operating system for which it was written.

1.2 Terminology

The discussion in this thesis involves the "guest" and the "host". "Guest" refers to the operating system(s) and associated applications that are executing under control of an emulator or virtual machine monitor. The emulator or monitor in conjunction with the supporting operating system (if any) is referred to as the "host" environment. The host retains control of the machine and multiplexes hardware resources between all native applications (if any) and the various guests.

The hosting operating system in this thesis is xok. Xok is an implementation of the MIT exokernel on the x86 processor [7]. Exokernels provide very few traditional operating system abstractions; these are instead available through libraries. The monitor presented in this thesis uses libexos, a library operating system providing a POSIX interface.

1.3 Design strategy

Some of the aforementioned scenarios may be solved by simply having applications written for one operating system run under another. A binary emulator that reimplements one operating system's API within another is a possible solution. Other times having the entire operating system present is desirable; for this, a virtual machine monitor or full emulation may be used. Numerous emulators and monitors exist that offer either of these solutions. However, the problems with emulators and most monitors are multiple. Full emulation is often an order of magnitude or more slower than running code natively. Virtual machine monitors often do not offer full use of the processor and other hardware to the guest operating system. Binary emulation has its own problems, with the primary one being the difficulty of implementation. The API of the guest operating system may be proprietary, rapidly evolving, or both. This may leave the binary emulator always unable to run the current crop of applications.

This thesis attempts to answer the problem at a more fundamental level by demonstrating how to multiplex the x86 hardware among protected mode operating systems with a virtual machine monitor. This allows guest operating systems to fully use the 32-bit mode of the processor, avoids the difficulty of tracking the APIs of numerous operating systems, and is faster than full emulation since instructions are run natively on the hardware when possible.

Specifically, this thesis seeks to allow a Linux-based operating system to run under xok, so that both Linux and xok native applications may coexist. The monitor has been designed in such a way that it will be an incremental step to allow other operating systems to run under xok in the future.

Emulators and virtual machine monitors are certainly not new fields, but this project examines several areas that are not well researched. Virtual machine monitors exist for the x86, but few allow protected mode operating systems to run. Of those, fewer still have source code available to show how the virtualization was implemented. This thesis seeks to offer the emulated operating system the full power of the hardware without having to pay a significant performance penalty.

1.4 Thesis organization

This thesis is organized as follows: Chapter 2 reviews related work in the fields of emulation and virtualization and contrasts them in terms of speed and ease of implementation. Chapter 3 gives background information on the x86 architecture and then Chapter 4 examines what problems this poses for a protected mode virtual machine monitor. Chapter 5 discusses the issues that shaped the design of the monitor and presents the high-level solution. Chapter 6 examines how memory is shared between all operating systems, while still giving each the impression of owning the machine. Chapter 7 examines difficulties introduced by the segmented architecture of the x86. Chapter 8 examines interrupt handling issues, and then hardware virtualization is discussed in Chapter 9. The implementation of the monitor and usage is discussed in Chapter 10. The role xok played in this thesis, along with the security of the modified xok, is discussed in Chapter 11. Chapter 12 presents some performance numbers for this monitor and compares these against other popular x86 emulators and monitors. Finally, the conclusion is presented in Chapter 13.

Chapter 2

Related Work

2.1 Full emulation

One way to achieve the goal of allowing multiple operating systems to run on a single processor is through full emulation of the x86 processor and peripherals. Some software packages do take this approach, but the performance penalty is substantial, often an order of magnitude or more. More expensive hardware may yield a several-fold increase in emulation speed, but this likely isn't enough to make up for the overhead of emulation. Worse, with a solution involving more expensive hardware, the original financial advantage of utilizing an average, inexpensive personal computer to run multiple operating systems is lost.

Bochs[1] is a full x86 emulator. It runs on many platforms and features full emulation of the x86 hardware, including the processor, memory, BIOS, and devices. Bochs emulates all instructions, even if it is running on x86 hardware. This full emulation allows it to faithfully reproduce the behavior of the x86 regardless of the hosting hardware. Bochs has been reported to successfully run Linux, DOS, Windows 95, and other systems. However, benchmarking shows that Bochs may run programs nearly two orders of magnitude slower than native execution on the same hardware (Table 2.1).

[1] http://www.bochs.com

Operating environment	InfoZip 2.2; FreeDOS beta 3
None (native execution)	11.7
DosEMU 0.98.6; Linux (16-bit virtual machine)	14.7
VMWare build 135; Linux (32-bit virtual machine)	139
Bochs 990219a; Linux (full emulation)	982

Time in seconds taken to compress a 1.44 megabyte file with maximal compression. All tests were performed on a Pentium Pro 200.

Table 2.1: Speed comparison of some emulation techniques

2.2 Binary emulation

Alternative implementations of operating systems' APIs are another approach to solving the problem so long as the entire operating system itself is not needed, but it is a potentially difficult solution. Wine[2] is an implementation of the Windows 3.x and Win32 APIs on UNIX. This project has been under active development since 1993; after six years some (arguably now obsolete) 16 bit programs run under UNIX but few 32 bit ones do. While reimplementing an operating system's API within a host operating systems allows for convenient information sharing between the two, such as being able to cut and paste between Windows and X Window programs with Wine, it clearly has a slow development cycle for proprietary APIs. This is definitely a problem since there exist closed-source operating systems that would be useful to support now, such as Windows 9x, Windows NT, and BeOS.

2.3 Virtual machine monitor

Another option is to share the desktop PC between operating systems through the use of a virtual machine monitor. A virtual machine monitor runs code natively on

[2]http://www.winehq.com

the CPU when possible; only privileged instructions are trapped and emulated by the monitor's "trap handlers". This yields better performance in general than full emulation and may be less difficult than reverse engineering and reimplementing proprietary and ever-changing operating system APIs. Furthermore, a virtual machine monitor need not be specific to a particular operating system as does binary emulation. A virtual machine monitor recreates an entire machine that could potentially run a number of operating systems.

A simple example of a virtual machine monitor is DOSemu[3], a monitor for x86-based Unices that targets running MS-DOS compatible operating systems. It creates a virtual 16-bit x86 machine under which any DOS may run, complete with virtual hardware devices and an alternative implementation of the BIOS. Accesses to the hardware and privileged instructions are trapped and emulated by the monitor, but otherwise the code is allowed to run natively using the processor's virtual 86 mode. Performance penalties are around 25% for CPU and disk intensive applications (Table 2.1), which is much better than full emulation. However, this specific virtual machine monitor may be of limited practical use since it is emulating hardware that is more than a decade old. Most software of interest will not run in this 16-bit environment.

A more radical approach is to change the monitor from being an application under a host operating system to being a thin operating system itself. Disco takes this approach. It is a thin monitor that allows commodity operating systems to be run on scalable multiprocessors, with low overhead. Disco is designed for FLASH, a non-uniform memory access (NUMA) machine. In each of four uniprocessor benchmarks (pmake, engineering, raytrace, and database) overhead due to Disco increased the execution time by no more than 16% [2]. Not only does Disco allow multiple operating systems to run on the same machine, but it also allows uniprocessor operating systems to take advantage of multiprocessor hardware. Despite Disco's strengths, it was originally not written to run on x86 machines. Considering that x86 machines are inexpensive, increasingly powerful, and enjoy wide support, a practical solution must support them.

---

[3]http://www.dosemu.org

VMWare[4] is a recently announced commercial x86 virtual machine monitor, developed by several of the original Disco developers. VMWare allows multiple protected mode operating systems to coexist simultaneously under Windows NT or Linux. It virtualizes the processor and peripherals well, for a much lower cost than full emulation. However, since it is commercial little information is available regarding its implementation.

[4]http://www.vmware.com

Chapter 3

x86 Background

To aid full understanding of the virtualization issues and solutions presented in later chapters, a brief overview of the x86 architecture is presented here.

3.1 History

Intel's x86 line of microprocessors is usually regarded as beginning with the 8086 and 8088 chips in 1979, although these chips were influenced by the earlier 4004, 8008, and 8080 chips. The x86 has seen numerous upgrades since that time. Notable points in the evolution of the x86 architecture include the 32-bit 80386, which added a fully functional protected mode; the 80486, which integrated the math coprocessor and added the ability to invalidate individual TLB entries, and the Pentium, which was the Intel's first super-scalar x86 chip. Interestingly, current x86 processors are still able to run most software written for the 8086 and 8088. In order to support current needs while retaining backwards compatibility, modern x86 processors have four modes of operation: real mode, protected mode, virtual 86 mode, and system management mode.

3.2 Operating modes

While in real mode, the processor is 16 bits internally with a 20-bit physical address space. To allow the one megabyte of physical memory to be addressed with the 16-bit registers, two registers are used—one to denote which 64 k segment of memory to use, and the other to find an offset within that segment. This is referred to as a segmented memory model. Real mode programs often segregate their data into different memory segments for convenience, but the processor offers no protection between them. Any process can corrupt the memory of another. Because of this, real mode is not suitable for a multi-user or multi-tasking operating system. The first generation of x86 chips (the 8088 and 8086) supported only real mode. All subsequent x86 processors power up in real mode for backward compatibility.

The 80386 and later processors offer a new mode of operation called protected mode. (The 80286 offered a protected mode that lacked some important features and thus is not considered here.) Protected mode is the preferred mode for modern operating systems on the x86. This mode offers 32-bit registers, a 32-bit memory address space, a memory management unit, and hardware-based protection between processes. Processes are assigned privilege levels ranging from 0 (most privileged) to 3 (least privileged). Chips that support protected mode still power on in real mode; a sequence of instructions must be executed to switch to protected mode [6].

Virtual 86 mode is another mode supported by 386 and later processors. It exists to allow legacy real mode applications to safely execute within a protected mode operating system. Virtual 86 mode is similar to real mode except that the processor traps many instructions, allowing them to be emulated by a virtual machine monitor.

The final mode is system management mode (SMM). This mode is typically used only by hardware control programs embedded in the firmware of laptops, such as power management software. This mode is transparent to all other modes and operating systems, and is therefore not addressed in this thesis.

Real mode has hardware virtualization support through virtual 86 mode; protected mode has no equivalent. This thesis focuses on how to virtualize protected mode through software. Because of the focus on protected mode, some aspects of this mode are reviewed in more detail below.

3.3 Segmentation

Protected mode memory management is accomplished on the x86 processors through the use of both segments and page tables. Segmentation is a method of partitioning the processor's addressable memory space (also called the linear address space) into a set of potentially overlapping smaller regions. These regions may hold code, program data or stack, or system structures.

The x86 processor supports memory segmentation in each of its operating modes. Real mode, system management mode, and virtual 86 mode all share the same 64 k segment design. Protected mode offers a much more powerful segmentation model. Like the others, this segmentation model offers an elementary form of memory management through the division of the linear address space, but it also adds protection and typing of segments.

Segments may be marked as either read-write or read-only and as either data or code. Attempting to use the segment in an invalid way (such as trying to write to a read-only segment) causes the processor to generate an exception. Segments also offer protection based on privilege levels; they may be marked to allow only processes with at least a given privilege level to access them. This is discussed in more detail in Section 3.5.

Segments are described by small structures called descriptors. The descriptor of a segment stores the size and location of that segment, the privilege level, and access settings such as read-only versus read-write. Segment descriptors may be held in the global descriptor table, or the GDT. The GDT is an array of segment descriptors. It exists in a segment of its own. To access memory in a given segment, a far pointer must be used to create a logical address. A far pointer consists of a segment selector (that is, a 16-bit index into the GDT) and a 32-bit offset within the specified segment. Only one GDT is supported by the x86; this GDT should be under control of the operating system. The GDT is defined by loading a base pointer and a length into the `GDTR` register. Applications may define their own segments (if the operating system is written to support this) by using local descriptor tables (LDTs). LDTs are similar to GDTs except that there may be zero or more LDTs [6]. The current LDT is defined by loading the `LDTR` register. Some operating systems, such as Linux, automatically create one LDT per process.

Segments that are currently being used must have their segment descriptors loaded into special segment registers. The `CS` register holds the segment descriptor for the segment containing the currently executing code. The stack segment descriptor is loaded into the `SS` register. The registers `DS`, `ES`, `FS`, and `GS` all may hold data segment descriptors. Segment registers actually consist of a hidden portion and a visible portion. Reading from a segment register will yield only the visible portion. It mainly consists of the segment selector—the index into the descriptor table. The hidden portion is actually a cached copy of the descriptor from the descriptor table. Often-needed information, such as the base, size, and protections of the segment, are available to the processor from the hidden portion of the register. This saves bus cycles since repeated references to the GDT or LDT are avoided. The hidden portions of the segment registers are often collectively referred to as the segment descriptor cache. Instructions that load segment registers implicitly load both the visible and hidden portions.

Segmentation cannot be disabled. However, creating a 4 gigabyte code segment, stack segment, and data segment essentially removes segmentation from view.

3.4 Paging

Paging is implemented in the x86 through a two level page table when using 4 kilobyte pages, or using just one level of indirection with 4 megabyte pages. The root of the page table is pointed to by the `CR3` register. Unlike segmentation, paging is optional. Most Unix-like operating systems on the x86 define 4 gigabyte segments and implement protection entirely through the paging mechanism.

Near pointers specify an offset from the base of the current segment to yield an address in the 32-bit linear address space. Far pointers specify both a segment and an offset. In either case, the resulting linear address is used as the index into the page table.

3.5 Privilege levels

While in protected mode, the processor can execute instructions at any of four privilege levels. Applications run at level 3, the lowest privilege level. This is often referred to in Intel documentation as the "application privilege level". The operating system kernel runs at 0, the highest level. This is also known as the supervisor privilege level. Levels 1 and 2 are unused in many operating systems. These levels are as privileged as level 0 from the point of view of the paging hardware, and therefore may be used for kernel services or drivers.

The privilege level of a process is defined by a two-bit field in the process's code segment selector. This field is called the requestor privilege level, or RPL. Segment descriptors are also marked with a privilege level, called the descriptor privilege level, or DPL. In general, processes assigned to the application privilege level cannot access memory through a descriptor marked with a higher privilege level. This may be used to prevent an application from reading memory owned by the operating system. Also, processes cannot implicitly transfer control to less privileged segments, such as by using an interrupt descriptor to trap to a lower privilege level. A given privilege level should only trust itself and higher levels, and the hardware enforces this.

3.6 Registers

The x86 has 6 general purpose registers, 2 more for stack management, the previously mentioned segment registers, and a number of others reserved for special purposes. One of the special purpose registers is the `EFLAGS` register. This register holds a motley set of system status flags. Some flags (such as the carry flag or zero flag) are set according to the results of instructions. Other flags (such as the virtual mode flag and the IOPL flag) may be set by the operating system to control the functioning of the system.

3.7 Task management

The x86 processor supports several task management primitives while in protected mode. A task is defined as a unit of work that the processor can dispatch, execute, and suspend. It is described by the task state segment, or TSS. The TSS contains fields to save processor state (i.e., registers) when a task is suspended. It also includes a pointer to a page table and I/O permission bitmap to use with that task.

The TSS does not dictate the privilege level of the task it describes, since the `CS` register is not set from the TSS. The privilege level is set when a task is invoked. Due to hardware enforced protections, each privilege level must have a separate stack. Because of these two facts, the TSS contains stack segment descriptors and stack pointers for privilege levels 0, 1, and 2.

A multitasking operating system may use the TSS fully by assigning a separate TSS to each process, or it may elect to define only a single TSS to be shared among the operating system and applications. In the latter case, the saving and restoring of state, such as registers and page table pointer, must be done manually by the operating system.

3.8 Interrupts and exceptions

Interrupts come in two varieties on the x86: hardware-generated and software-generated. Hardware-generated interrupts come from a device such as the timer or keyboard. An example of a software interrupt is a system call, generated by an explicit `int n` instruction in the application. Exceptions may be generated when the CPU detects an error condition. Certain exceptions, such as a breakpoint exception, may also be generated with the appropriate `int` instruction in software.

A handler is the function designed to deal with an interrupt or exception. Interrupts and exceptions are directed to the appropriate handler through the interrupt descriptor table, or IDT. This table is defined by the operating system and holds up to 256 descriptors. Descriptors in the IDT are also referred to as gates, since control is transferred through them. Most descriptors contain a reference to the code segment in which the handler is to run, the offset of the handler in that segment, a minimum privilege level that is required to service the interrupt or exception, and a field that identifies it as an interrupt gate or trap gate. Interrupt and trap gates differ slightly in how they modify the state of the registers when called. A third type of gate is the task gate, which is used for hardware-based task switching.

The IDT is defined by setting the `IDTR` register to point to the structure in memory. Only the operating system may write to the `IDTR`.

Chapter 4

Problems Virtualizing the x86

The x86 line of processors have a number of protected mode aspects that cannot be virtualized fully by a virtual machine monitor. The following sections detail these virtualization challenges.

4.1 Privilege levels and segmentation

Typically, virtual machine monitors run guest operating systems at a lower privilege level than normal so that certain instructions may be trapped and emulated. It may be important that the guest operating system does not realize it is running at a privilege level besides level 0, which it expects. This is not always possible. The following example demonstrates why virtualization of the privilege level is difficult.

When an interrupt or exception to a higher privilege level occurs, the stack segment and stack pointer must be changed to use the higher privileged stack. The new descriptor and pointer are located in the TSS of the new task. The old stack selector and pointer are saved on the new stack so that execution may be resumed later. However, if the interrupt or exception does not cause a change in the privilege level—for example, if an exception occurs in the kernel—neither the privilege level nor the stack change, and therefore the processor does not save the stack segment and stack pointer.

The operating system must differentiate these cases to correctly handle the stack.

One potential way to distinguish the two cases is to look at the code segment selector from the interrupted task [9]; the processor pushes it onto the stack whether or not a privilege level change occurs. This selector's RPL is the privilege level of the interrupted process. If the guest operating system checks for recursive traps to the kernel by comparing the RPL value against 0, it will always fail since monitors do not run the guest at privilege level 0. The operating system will then handle some types of traps incorrectly, corrupt the stack, and crash.

The x86 privilege level model is not easily virtualizable because privilege levels, such as the DPL and RPL, are embedded within segment descriptors and selectors. Segment descriptors and selectors are often manipulated directly by the processor with no opportunity for virtualization. This potentially exposes the lower than expected privilege level.

4.2 Segment descriptor cache

It is possible at times that the segment descriptor cache does not accurately reflect the current state of a descriptor in the descriptor table. This would occur if the descriptor is changed (by either writing to it or reloading the appropriate descriptor table register) without reloading the segment register. In such a case, the hidden portion of the segment register contains values from the previous descriptor and therefore may be called "stale".

Section 11.2 of [5] recommends that all segment registers in use be explicitly reloaded immediately after modifying the associated descriptor. Presumably, this is suggested to keep the descriptor cache in agreement with the descriptor table, and therefore keep the code easily understandable. However, the processor's behavior is well defined even if the segment registers are not immediately reloaded. Using stale segment registers is entirely valid but potentially makes the assembly instructions confusing to a casual observer.

Stale segment registers pose a difficulty to virtual machine monitors. Imagine a scenario in which the `CS` register is loaded, the associated segment descriptor is later modified via an `lgdt` instruction, and then an instruction involving the `CS` register traps to the monitor before the register has been explicitly reloaded. Depending on the implementation of the monitor, it may need to read attributes of the `CS` register during emulation of the instruction. Reading from the descriptor table would yield fresh attributes, while if the instructions had run natively on the processor, the stale attributes would have been used. The true values are stored in the hidden portion of the segment register, but there is no easy or efficient way to read them. To make the x86 more easily virtualized, the hidden portion of the segment register should be visible.

4.3 Paging

The paging mechanism is not so tightly integrated with the processor's concept of privileges as are the segments, and therefore does not pose such serious virtualization issues. The most immediately obvious issue is how to allocate physical pages among xok, its applications, and the guest operating systems. This question is not specific to the x86 processor and has been solved by other monitors, including [2].

A more subtle problem is that of balancing the need for a fast monitor against the problem of virtual address collisions. If the monitor and the guest reside in different page tables, they are separated from one another by the hardware. The guest operating system may use the same virtual addresses as are used by the monitor; since they are on different page tables these addresses map to different physical pages. This separation comes at a cost of speed. Each trap to the monitor requires the page tables to be reloaded, flushing the TLBs. This is a major performance penalty. The L4 kernel designers, for example, discovered that avoiding TLB flushes on calls between privilege levels improved IPC performance between 2 and 6 times [4]. IPC involving the guest applications might not have to be trapped by the monitor (depending on the monitor's design), but privileged instructions including device I/O would suffer this penalty. Having the monitor and the and guest OS on the same page table avoids these flushes but makes makes managing virtual addresses more complex.

4.4 Untrappable instructions

Some instructions that ideally should be trapped and virtualized are untrappable in protected mode, regardless of the privilege level. In particular, `popf` and `pushf` may not be trapped outside of virtual 86 mode. `popf` pops the top of the stack into the `EFLAGS` register; `pushf` pushes the current value of `EFLAGS` onto the stack. Since `EFLAGS` itself contains flags that control which instructions will trap at a given privilege level, this register should be virtualized.

The `popf` instruction is strange in that it will not trap while in protected mode, but its effect varies depending on the current privilege level. If the privilege level is not 0, `popf` quietly refuses to change some flags in `EFLAGS` such as the IOPL setting and virtual 86 mode flag. `pushf` pushes an exact copy of `EFLAGS` onto the stack at all privilege levels. Since there is no way to trap and emulate attempted changes to `EFLAGS` nor push virtualized values onto the stack, virtualization is incomplete.

Even if a particular guest operating system is not directly affected by these holes in the virtualization, the untrappable instructions will still prevent other monitors from running within the monitor presented here. As just one example, DOSemu uses virtual 86 mode to create a 16-bit virtual machine within a protected mode operating system. To start virtual 86 mode, the hosting operating system sets the virtual 86 mode flag in `EFLAGS` and returns to the DOS application, presumably with the processor now in virtual 86 mode. However, if this hosting operating system is itself within a virtual machine monitor and therefore not running at privilege level 0, the bit in `EFLAGS` will fail to be set and virtual 86 mode will not start, causing DOSemu to crash.

4.5 Summary of virtualization challenges

The x86 poses a number of virtualization challenges. The difficulties fall into three general categories:

1. **Tight integration between items which are not strictly related.** The segmentation architecture (segment selectors and descriptors) and privilege levels are often manipulated as one by the processor, potentially exposing the privilege level.

2. **Hidden portions of registers.** The segment descriptor cache cannot easily be read, potentially leading to incorrect virtualization by the monitor.

3. **Untrappable instructions.** Numerous instructions cannot be trapped, which potentially could cause the virtualization to fail. Some of these instructions which ideally should be trappable are: `popf`, `pushf`, `sidt`, `sgdt`, and `sldt`.

Chapter 5

Design

5.1 Assumptions

A central assumption in this thesis is that the guest operating system neither is malicious nor contains bugs. A reasonable virtualization of the machine hides pointers to processor-defined structures such as the page table and GDT, keeping a well-behaved guest out of the hosting operating system's structures. To a correctly functioning benign guest, the hosting operating system and monitor are invisible. While the monitor presented in this thesis does that, it does not provide strong hardware protection between the guest operating system and xok. For reasons related to speed of the guest and ease of implementation, the guest, xok, and monitor share the same page tables. The guest operating system runs at a lower privilege level than xok, but the x86 paging hardware treats levels 0, 1, and 2 as the same. This makes it possible for a wild or malicious pointer in the guest operating system to corrupt xok data structures.

Not all bugs in the guest operating system lead to instability of xok. Kernel panics, null pointers, wild pointers which do not hit xok, and any other errors which cause an exception can be handled gracefully by the monitor. Only wild pointers which hit xok or xok structures will cause problems.

This was seen as an acceptable risk since there is at least some reason to trust operating systems like Linux—they tend to be quite stable on their own. Furthermore, this assumption allows the guest to be placed on the same page table as xok and the monitor, avoiding many TLB flushes during emulation. Strict stability in the face of an untrusted guest operating system was traded for speed and ease of implementation.

No such assumptions must be made about the virtual machine monitor application. Since it runs at the lowest privilege level, occasionally it must request xok to perform some task on its behalf. These requests are made through system calls. The monitor is given no unrestricted access to hardware or xok system structures, and therefore cannot crash the system.

Another guiding assumption was that strict virtualization was not necessary to run Linux as a guest operating system. If Linux does not use a feature of the x86 that is difficult to virtualize, perfect virtualization of that feature is not necessary.

5.2 Solution overview

An overview of the solution implemented in this thesis is presented in Table 5.1. The emulation model chosen for this thesis was a virtual machine monitor implemented under xok. The monitor itself, which includes initialization code, trap handlers, and hardware virtualization code, runs as a normal application.

The guest operating system runs at privilege level 1 so that privileged instructions may be trapped. The trap handlers run at privilege level 3. To allow the monitor to handle all traps that occur in the guest, a trap trampoline was added to xok that potentially redirects traps to the monitor. The overhead of these traps is reduced by having the guest, xok, and monitor share the same page table, which avoids some TLB flushes. Sharing the page table leads to the possibility of contention between the guest, xok, and monitor over virtual addresses. This was avoided by placing xok and the monitor at virtual addresses known to be unused by the guest.

Hardware virtualization is handled by the monitor. This includes disk and video, as well as physical memory. To coordinate the allocation of physical memory pages between xok and the guest, the guest's page table is write protected. Attempted writes to the page table are trapped and handled by the monitor.

Problem	Solution
Emulation model	Application-level virtual machine monitor implemented under xok.
Privileged instructions	Run guest operating system at a lower privilege level so instructions may be trapped and emulated.
Some traps from normal xok apps must go to kernel	Wrote a trap trampoline that redirects traps to user-space as needed.
Trap overhead	Guest, xok, and monitor are on same page table to reduce TLB flushes.
Virtual address conflicts	Placed xok and monitor at virtual addresses known to be unused by guest.
Physical page allocation	Trap writes to guest page table; remap to free physical page.
Contention over devices	Virtualize devices.
`pushf` and `popf` are untrappable	None; Linux assumes the values it tried to set are in affect and does not check. Nested virtual machines may fail.

Table 5.1: Solution overview

The following sections examine in in more detail how the high level design decisions were made. More specific topics, such as the rational behind the memory management system, are saved until the appropriate chapter.

5.3 Solution

5.3.1 Emulation model

Based on the benchmarks presented in Table 2.1, it was felt that full emulation was too slow to be useful in a production environment on average PCs. This left a choice between a virtual machine monitor and binary emulation. The virtual machine model was selected over binary emulation due to flexibility. A virtual machine may potentially run any of a number of operating systems; binary emulation inherently targets only one.

5.3.2 Handler placement

Initially, putting the handlers in xok's kernel segment was seen as a potential design. This essentially meant compiling the handlers as part of xok. The handlers would have direct access to all xok functions without being restricted to specific kernel entry points. This might make implementation easier and would cause the emulation of some instructions to be faster since no cross-boundary calls would be needed.

Another potential advantage of having the handlers in xok is that it would yield a cleaner virtual address space, since all code upon which the guests depend is centrally located. The importance of having a clean virtual address space is discussed in Section 6.2.6. However, it turned out to be easy to ensure user-level handlers would not conflict with the guest, so this was no longer compelling.

The primary drawback of the in-kernel approach is that it goes against the exokernel philosophy. Xok strives to separate management and abstraction from protection. Xok focuses on protection, but the handlers are a specific implementation of an abstraction. Inserting the handlers into xok violates this separation and does not take advantage of the flexibility xok offers to applications.

Implementing the handlers within xok would increase the size of the trusted code base. This is undesirable because it could reduce the stability of the system. Kernel-space designs tend to be harder to implement and debug than user-space. Additionally, writing the monitor as an application could highlight other areas xok could be changed to offer more flexibility to other applications.

In the end it was decided that placing the handlers within xok would make implementation and debugging more difficult, without offering many clear advantages.

5.3.3 Segmentation

The guest operating system must be allowed to create and use a complete 4 gigabyte segment. The guest is not aware that it is running within a virtual machine, and it therefore expects to be able to create and use full-sized segments. However, xok has two main code segments: a 3.2 gigabyte user segment followed by a 0.8 gigabyte kernel segment. The monitor, running with application privileges, cannot access all addresses in the upper gigabyte and therefore cannot emulate guest instructions in this range.

The general solution to this is to remove segmentation from xok. This has not yet been done, since Linux does not attempt to run code in the upper 0.8 gigabytes. This is discussed in more detail in Section 7.1.

5.3.4 Privilege levels

An operating system normally runs at privilege level 0, but at this level instructions cannot be trapped. Instead, the guest operating system is allowed to run at privilege level 1 at most, so that privileged instructions may be trapped and emulated by the monitor. Segment descriptors created by the guest with a DPL of 0 are forced by the monitor to have a DPL of 1. Effectively, level 0 is collapsed together with level 1 for the guest.

This approach could potentially cause problems in guests that want to use privilege level 0 for the kernel and 1 for drivers. Collapsing them together under the virtual machine monitor removes some hardware protection between the two components of the operating system. The hardware would prevent the kernel (level 0) from trapping to the driver (level 1), for example. Under the monitor, this would be allowed. With our assumption of a non-buggy guest operating system, this is not a problem; the guest should never attempt such a call.

Having the two privilege levels collapsed into one changes the way the stack is handled even for valid calls between the two. Normally, for calls between privilege levels, a stack segment descriptor and stack pointer are pushed and popped automatically during the call and return. This will not occur if both levels are collapsed into privilege level 1, leading to a corrupted the stack. Linux only uses levels 3 (for applications) and 0 (for the kernel)[1] so this is not an issue for the particular guest this thesis focuses on.

[1]Consult `linux/arch/i386/kernel/process.c`, which sets up the TSS.

Collapsing levels 0 and 1 together does not affect page-level protections, since from the perspective of the paging hardware privilege levels 0, 1, and 2 are all considered the same—that is, "supervisor" privilege.

Since the virtual machine monitor runs as an application, it runs at privilege level 3. It is tempting in some situations to allow it access to higher privilege levels so that it may emulate some instructions more easily, but this is not a clean solution. Such a situation occurs if the instruction that the monitor is emulating requires access to a supervisor level page of the guest. Applications cannot read or write supervisor level pages. Giving the monitor a higher privilege level would allow it to emulate this instruction, but would also require that the monitor be trusted. This is undesirable. The simple solution used here is to use an existing system call in xok that allows the protections of a page to be changed, so long as the page is not owned by xok itself. Once emulation is completed, the protections may be restored so that the guest operating system continues to enjoy hardware protection from the guest applications. Thus the monitor may handle these situations while still running at privilege level 3.

5.4 Multiple guest operating systems

Keeping the virtual machine monitor separate from xok allows it to leverage off of xok's process management code and scheduler. Starting a guest operating system is as simple as `exec`-ing the monitor application; issues regarding page tables are handled by the kernel as they would be for any other application. A little extra effort does need to be invested to multitask multiple guest operating systems, however. All guest operating systems share some common structures such as the GDT. It is the responsibility of the monitor to save any modifications to the GDT before yielding the processor and to restore them upon regaining the processor. If only one guest operating system is active, these steps need not be done, saving some overhead.

Chapter 6

Memory Management

Memory is a vital resource which must be shared intelligently between xok, xok applications, and guest operating systems. Only xok knows the status (used or free) of each physical page of memory. Guest operating systems do not know this and are not, in general, written to collaborate their allocation of physical pages with other entities. The virtual machine monitor along with xok must force this cooperation to take place while still providing guest operating systems with the illusion that they each have full use of the physical memory.

6.1 Requirements

The design of the memory management system of the monitor was driven by several requirements. First, it should make efficient use of physical memory. A guest that requires little memory should not blindly be awarded the same amount as a more memory-hungry one. It is also desirable that the memory management system of the monitor have low overhead. The amortized speed of the guests' accesses to memory should not be significantly slower than those of normal xok applications. Finally, the design must be correct. A guest operating system should not be able to tell that it does not exclusively own the page tables and physical memory.

6.1.1 Efficiency

Running multiple operating systems on one computer naturally consumes more memory than running a single operating system. Duplicate kernels, kernel state, and page tables all require the use of extra memory. A paging mechanism in libexos, although currently unimplemented, could mitigate the requirement for more memory but at the cost of speed. Overcommitting the physical memory resources between the operating systems will cause thrashing by the paging mechanism. On the other hand, awarding a guest little physical memory may keep xok applications more responsive, but the guest's performance will suffer. A balance needs to be found.

Statically partitioning physical memory between guests is not a good solution. Static memory partitions potentially waste memory since it is not always known a priori how many guest operating systems are to be run or what their memory requirements will be. Dynamic memory partitions are not suitable in this situation either; operating systems do not expect the size of physical memory to change while they are running. An ideal solution would allocate physical pages of memory only as needed so that smaller guests do not tax the system inordinately, yet impose an upper limit to control larger guests.

6.1.2 Speed

Some central authority must control the allocation of physical memory. A guest operating system does not expect to have to share physical memory with another operating system. Xok maintains information regarding the allocation state of pages, but it should not contain logic to control the guest's allocation of pages. This leaves the monitor application to mediate physical memory allocation. The obvious question, however, is how to do this at a reasonable amortized speed.

As was demonstrated in Section 2.1, complete virtualization of the x86 results in very poor performance. Trapping only those instructions that refer to memory in an attempt to multiplex physical memory is not acceptable either, as will be argued here. The x86 has only 8 general purpose registers [6], a relatively small number. By way of comparison, the PowerPC 601 processor offers 32 [8]. The variety of addressing modes of the x86 attempts to compensate for the small number of registers. The x86 `add` instruction, for example, can retrieve operands from and store the result to memory or registers. The 601's `add` instruction, by contrast, may only access registers. In general, the x86's instruction set is much more memory intensive. While it may be feasible on other platforms, trying to trap and emulate instructions that access memory so that physical pages may be shared essentially degenerates into full emulation—or slower, once trap overhead is considered—on the x86.

To retain acceptable performance, only select instructions should be trapped and emulated. The majority, even those that access memory, should run natively. To achieve this, it is obvious that the memory management unit of the x86 must be used as much as possible, not avoided or emulated, while sharing the physical memory.

6.1.3 Correctness

Bit 2 of the page directory entry and the page table entry is the "user/supervisor" flag. This flag helps moderate access to pages of virtual memory. For example, a page marked with the supervisor flag may be accessed by an operating system but not by an application. As another example, copy-on-write is implemented by using the supervisor flag and R/W flag in conjunction with the WP flag of the CR0 register. In other words, this flag erects protection between an operating system and its applications, and some operating systems further use it to control paging. It is therefore vital that the semantics of this flag are preserved even though the guest runs at an unusual privilege level.

The active page table may not agree with what the guest expects, depending on the implementation of the monitor. Some mappings may have need to be changed so that physical memory may be shared. This may confuse the guest operating system and cause it to fail. For example, in `linux/mm/memory.c`, the function `free_page()` is called on physical pages read directly from the page table. If Linux were to read a page table entry that had been modified by the monitor, the free page list in Linux would become corrupted. Therefore, the monitor's modifications to the page table must be undetectable to the guest.

6.2 Design

The guest executes within a hosting environment that has paging enabled and is in protected mode at all times. Other modes of operation (real mode or protected mode with paging disabled) can be emulated for the guest without changing xok's mode. The simplest case is real mode; the guest expects to boot in this mode. It is emulated by using the processor's virtual 86 mode and mapping the first megabyte of the virtual address space to serve as physical memory. Protected mode without paging can also be virtualized. The "physical" memory seen by the guest is actually the first n megabytes of the monitor's virtual address space. Finally, protected mode with paging enabled is made possible by carefully handling page tables. The following sections examine these mode virtualizations in more detail.

6.2.1 Setup

Before the guest operating system boots in virtual 86 mode, the monitor maps one megabyte of memory into virtual addresses `0x00000` through `0xffff` in its own page table. This simulates the physical memory addressable by the processor in real mode. An additional 64 k is mapped immediately following the first megabyte, but it refers back to the first 64 k. This simulates the wraparound that occurs in real mode when the segment and offset overflow the 20-bit physical addressing lines.

One step of the guest's initialization sequence that must be performed before enabling paging is to determine the amount of physical memory installed in the computer. Linux determines physical memory size by querying the BIOS[1]. Other operating systems such as BSD and xok actually walk the memory[2] by writing and reading back values. If the value read does not match the value written, the algorithm concludes it has passed the end of physical memory and stops.

[1] Refer to `linux/arch/i386/boot/setup.S`.
[2] Xok uses FreeBSD's `probe_for_memory()` function.

The monitor can virtualize both methods of discovering the memory size. The monitor includes a virtualized BIOS (examined in more detail in Section 9.1) that returns the memory size as defined by the monitor's configuration file. To handle probing, the monitor refuses to allocate physical pages beyond the virtualized memory size. Since these pages are not mapped, attempts to write to them will trap. The monitor ignores such write attempts. Read attempts will also trap; zero is returned. This convinces the probing algorithm that it has reached the end of physical memory. The amount of memory present in a system should be virtualized to be smaller than it is in reality to avoid the thrashing problem mentioned in the requirements.

It is possible for the guest to enter protected mode and enable paging as separate steps. If the guest is in protected mode with paging disabled, physical memory is emulated by mapping the appropriate number of pages into the monitor's address space starting at virtual address `0x0`. The original mappings for the the megabyte used by virtual 86 mode are preserved; the wrap-around mappings for the next 64 k are removed and replaced.

Since protected mode with paging enabled is the primary mode used by Linux, the next section is dedicated to discussing how this mode is virtualized.

6.2.2 Page table management

As discussed in Section 6.1.1, the memory management unit of the x86 must be used by the guest; it cannot be emulated without sacrificing performance. The capstone of the memory management unit is the `CR3` register. This register must be loaded with a pointer to a valid page table for the memory management unit to work correctly. The page table will not agree with what the guest expects since it will contain mappings for the monitor and xok, as outlined in Chapter 5. Far more troubling is that this page table may contain mappings to physical pages other than what the guest originally set, since the monitor may change mappings to share physical memory among all guests.

Fortunately, accesses to the `CR3` register by tasks at non-zero privilege levels fault. This allows the monitor to virtualize the `CR3` register and maintain two separate page tables. The first is managed by the monitor and used by the memory management unit. The other is a "placebo" page table. It is maintained by the guest, but it is not used by the hardware.

The placebo page table exists to preserve the integrity of the guest's mappings and avoid the possibility of corrupting the guest's internal state with the monitor's mappings. This page table is created by the guest operating system. Xok first learns of it when the guest attempts to set a reference to it in the `CR3` register.

The true page table is managed by the monitor. When the monitor first starts, this page table contains mappings for xok and the monitor itself. Modifications made by the guest to the placebo page table must be reflected in some way in the true page table so that the memory management unit sees the change. However, the physical page mapped by the guest may already be in use by xok or an xok application. Because of this, the monitor must find an available physical page (through an xok system call) and associate the guest's requested virtual address with the new physical page. The guest's permission bits (the low 12 bits of the page table entry) are retained in the new entry; only the physical page is changed. The new entry is then inserted at the correct virtual address in the monitor's page table. The translation from the guest's requested physical page to the actual allocated physical page is remembered by the monitor in case the same physical page is requested again. This ensures consistency in the allocation of physical pages.

Writes to `CR3` by the guest will trap. This gives the monitor an opportunity to examine the guest's proposed page table. The monitor will walk the entire placebo page table, translating and then inserting each entry into the monitor's page table. The guest's page table is left untouched during this entire process. Once all mappings have been processed and inserted into the monitor's table, the monitor write protects the guest's page table[3]. This allows future writes to the guest's page table to be trapped, translated, and inserted into the monitor's table before emulating the original write.

[3] Supervisor processes are forced to honor the read-only bit when the WP bit of CR0 set. Xok has this bit set to implement copy-on-write.

Reads from CR3 by the guest also trap. To handle these instructions, the monitor returns a pointer to the placebo page table. To the guest, the page table appears as the guest intended to create it.

6.2.3 Changing page tables

In most modern multitasking operating systems, applications are protected from each other by being placed on separate page tables. Each application has the kernel mapped into its page table so kernel calls do not require TLB flushes. Each application, however, is on a separate page table. Switching between applications requires CR3 to be reloaded. Xok and Linux both operate this way.

The cost of this switch is largely due to the TLB misses it causes; the cost of the actually loading CR3 is relatively small. Under the monitor, however, emulating the load to CR3 can be expensive. The register is not actually changed since applications do not have the privilege to do so. Instead, the existing page table is modified to reflect the change which would have occurred. The entries for the first application are removed and replaced by those of the second.

This has the potential for being extremely slow. A simple implementation (as was the first implementation in this thesis) is to perform a separate system call for each modified entry. This leads to performance worse than full virtualization, as shown in Chapter 12. Xok offers a better solution: batched system calls. The system call numbers and arguments are queued until a sufficiently large number are pending. They are then sent to xok to be handled, avoiding the cost of repeatedly calling to and returning from xok.

Batched system calls can help when changing page tables, but little can be done with the current design to help performance if the guest operating system iterates over the page table, changing entries one at a time. Such a situation will result in repeated traps to the monitor and repeated system calls.

6.2.4 Page faults

Guest page faults are a difficult issue for the monitor. The monitor lacks the high level information needed to determine the cause of certain classes of page faults. An attempted read to a nonexistent page by a guest application may be a copy-on-write fault or a bug in a guest application—the monitor cannot tell. Only the guest operating system has access to its internal data structures, which are needed to decide how to handle such a fault. For example, Linux uses the following code to check if a page fault due to a write should cause a copy on write:

```
/*
 * Do we need to copy?
 */
if (mem_map[MAP_NR(old_page)].count != 1) {
        ...
```

The `mem_map` array is a data structure internal to Linux. The monitor obviously cannot emulate the behavior of Linux's page fault handler. Other faults, such as those that occur when the guest attempts to modify its page table, must be caught and handled by the monitor.

These facts dictate that page faults initially turn control over to the monitor. The monitor decides if this is a case it is capable of handling, such an attempt to write to the protected page table. If so, the monitor handles it and returns.

If the fault is one the monitor cannot handle (such as an application fault, copy-on-write, or demand paging), control must be turned over to the guest's handler. Redirecting faults to the guest is examined in Chapter 8.

6.2.5 Tracking protected structures

The guest must be prevented from writing directly to a number of processor-defined structures. The placebo page table must be protected for reasons already explained. The GDT and LDT must be protected for reasons that will be explained in Chapter 7. This is complicated slightly by the fact that the page table may have multiple entries pointing to the same physical page, all of which may not necessarily have the same protections. Although the original mapping for a given structure is write protected by the monitor when the structure is created, other mappings may be added later that aren't. This may exposes the structure and allow the guest to modify it without monitor intervention.

An example of this problem occurs with GDT in Linux. While booting, Linux identity maps the first four megabytes of memory and loads the kernel there. It later adds a mapping for the kernel starting at `0xc0000000`. The GDT is originally loaded at a low virtual address but becomes unprotected at a high one. It is clear that write protecting these structures at creation is not sufficient; all subsequent mappings must be write protected also.

To ensure that all references to such structures remain write protected, the monitor tracks protected structures by the locations of their physical pages, as numbered by the guest. (Recall from Section 6.2.2 that the guest's concept of physical pages does not match xok's.) Every guest mapping inserted into the monitor's page table is checked against this list of physical pages to be protected. Mappings that match have their read only bit set before being inserted into the monitor's page table.

The guest's physical page numbering is used instead of xok's since the former will tend to allocate multi-page structures contiguously. After remapping to xok's physical pages, these structures may no longer be contiguous. The code to manage protected ranges is simplified by using the numbering scheme that keeps pages of the same structure contiguous.

6.2.6 Virtual address collisions

Careful management of page tables—one active and one placebo per guest operating system—is one piece of the solution to allow physical memory to be shared among xok and the guest operating systems. However, the virtual memory space potentially needs to be shared between xok and each guest too.

Because the guest shares a page table with xok and the monitor, there are regions that the guest should not and cannot use. Xok and vital system structures such as xok's GDT and IDT use regions of the virtual address space. These things are never unmapped (xok is not self-paging, and the GDT and IDT are vital for handling exceptions) and therefore the virtual addresses they occupy may never be used by the guest, the monitor, or any other processes. The monitor also occupies a region of the virtual address space; the guest may not use this region either. However, a guest operating system assumes it is free to use any and all of the x86's four gigabyte virtual address space. This leads to the potential for a collision.

A virtual address collision occurs when a guest operating system tries to map a virtual address that is currently being used by xok or the monitor. Virtual address collisions between guest operating systems are not possible since each guest operating system has its own page table. The possibility for conflict exists only between a guest and the hosting environment.

Separate address spaces

One possible approach to solve this problem is to discard the original design and keep xok and the monitor on a page table separate from the guest operating system's page table. This lessens the problem of virtual address collisions at the cost of speed. Every privileged instruction in the guest operating system is trapped and emulated. Each trap would require that the page tables be changed, flushing the TLB. This is analogous to the IPC calls investigated in L4; flushing the TLBs on calls to the kernel slowed the calls down 2 to 6 times [4]. For guest kernel routines that frequently attempt to access privileged registers or modify protected structures such as the GDT, the slowdown can be significant.

It is also worth noting that xok is not designed to change page tables for traps. It would be possible to change the tasking model of xok to use a TSS for each task and allow the processor to switch TSSs (and hence, page tables) for traps to xok. However, this would impose the performance penalty of flushing the TLB on all xok applications, not just the guest.

Even with separate page tables the problem is not entirely solved. At least xok's interrupt descriptor table and the global descriptor table must be mapped into the guest's address space for task switching to occur. In other words, the possibility for virtual address collisions still exists to a small degree.

Relocatable and self-paging host

Another possible solution to handle these collisions in the general case would require that xok and the monitor be relocatable during runtime or be self-paging. In either case, the page fault handler in the monitor must guard against inserting guest mappings at virtual addresses already claimed by the xok or the monitor. Should such a mapping be requested, the handler would either relocate or page out part of the hosting environment.

Paging the hosting environment (xok and the monitor) potentially has a large performance hit due to "hot spots" between it and the guest. Suppose guest code is placed at a virtual address that conflicts with the hosting environment. This forces a portion of xok or the monitor to be paged out. Every trap (except page faults) to the monitor might require parts of the hosting environment to be paged back in.

Making xok and the monitor pageable is not enough by itself to solve the problem of virtual address collisions. Some portions of the kernel are inherently not pageable since they are used in the paging process itself: the IDT, GDT, and page fault handler. These portions would have to be relocatable in case their virtual addresses were requested by a guest.

Making both xok and the monitor relocatable is a more attractive option since it does not suffer hot spots. In such a design, if the page fault handler detects a virtual address collision, it would attempt to find a space in the virtual address space unused by the guest and move xok and/or the monitor there. It would be best to find a region unused by *all* xok processes so that xok could be mapped globally. This would reduce the number of TLB flushes. After moving, the page fault handler would have to perform various fix-ups such as correcting the offsets in the interrupt descriptor table. This fix-up would need to be done to all guest operating systems' page tables. This would be a time-consuming task, but it should reach steady state quickly—that is, it would soon find a region of the virtual address space that none of the guests

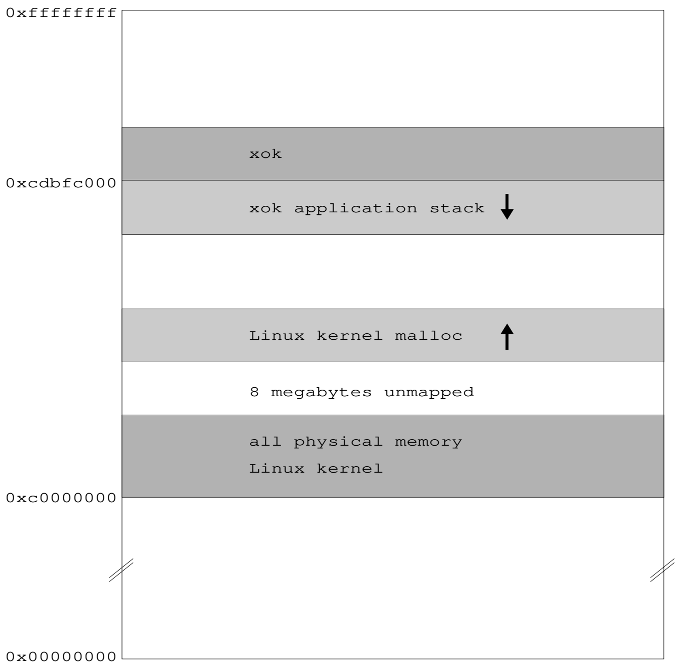

Figure 6-1: Virtual memory footprint of Linux overlaid on xok

ever use. However, it is likely more complex to implement than a self-paging kernel, and it assumes all guest operating systems in use at a given time share a common hole in their virtual memory footprints.

Avoiding collisions

Whether xok and the monitor are fully relocatable or a combination of pageable and relocatable, the solution to virtual address collisions is technically possible. However, both solutions would require a large amount of work while solving only a small portion of the overall problem presented in this thesis. A simpler solution is to avoid the problem in the first place. Collisions can be avoided entirely if the virtual memory usage patterns of the guests are known a priori. In such a case, xok and the monitor can be written to load themselves at addresses that are known to be unused by the guests.

This thesis simplifies the problem of virtual address collisions by focusing specifically on running Linux as the guest operating system. The virtual memory footprint of Linux is known, and xok can be written to accommodate this usage pattern at compile time. To generalize this to handle arbitrary guests, a common unused area of the virtual address space must exist among the guests or one of the above solutions may need to be implemented.

As documented on page 73 of [1], the first three gigabytes of Linux's virtual address space are allocated to the user; the upper gigabyte is reserved for the kernel. A patch was applied to the Linux kernel that allowed a keystroke to display a map of the upper one gigabyte of memory while Linux was running. From this it was discovered that Linux maps physical memory starting at virtual address `0xc0000000`. Memory `malloc`'d by the kernel starts eight megabytes after the end of physical memory, and grows upwards. The rest of the Linux kernel space is unmapped. The virtual memory layout of Linux on a test machine with 64 megabytes of physical RAM is shown in 6-1. Xok resides entirely above `0xcdbfc000`; the user stack grows down from there. A conflict will occur between xok and the guest Linux kernel only if Linux is given well over 200 megabytes of virtualized physical memory. If more is desired, xok must be recompiled to load at a higher virtual address.

6.3 Imperfect virtualization

6.3.1 Monitor location

Currently the monitor resides of memory location `0x800000`; shared libraries, including libexos, start at `0x1000000`. For better virtualization, these must be moved much higher in the virtual address space, above xok, so that they do not conflict with Linux applications. Moving the monitor itself is not difficult. Offsets are defined in `exopc/bin/exosloader/os/shexec.c` and `exopc/GNUmakefile.global` which control the placement of applications. Development tools such as `gcc` expect to place the text segment at `0x800000` by default, but a setting in the makefile will override this. Libexos, however, is not as simple. It places system structures and shared regions of memory starting at `0x30000000`. This can conflict with Linux. Libwafer is a minimal implementation of libexos without these problems. To reduce virtual address collisions, the monitor should be migrated to libwafer. This has not yet been done.

6.3.2 Four megabyte pages

The Pentium and later processors support a four megabyte page size optimization. A bit in the page directory entry selects if the entry points to a page table with 1024 separate 4 kilobyte pages, or to a single 4 megabyte page. Storing a kernel on a single 4 megabyte page saves TLB entries for other pages and reduces the size of the page table.

Linux optionally supports this optimization via a setting in `linux/include/asm/pgtable.h`. Currently neither xok system calls nor the monitor support four megabyte pages, so this option is turned off. True support would require modifying a system call interface in xok to allow applications to set the page directory entry; while certainly possible, this has not yet been done.

6.3.3 Guests

Placing xok, the monitor, and the guest on the same page table makes it impossible for xok to host an exact copy of itself as a guest. Although such an arrangement would not allow any new applications to be run, it may still be desirable for testing or fault isolation purposes. The virtualization is not good enough to support this situation.

Chapter 7

Segments

The segmentation issues discussed in this chapter apply only to protected mode segmentation. Segments in real mode pose no difficult virtualization issues since they lack integration with privilege levels and have no global table for storage of segment descriptors.

7.1 Xok segmentation

Xok has two main code segments: a 3.2 gigabyte user segment followed by a 0.8 gigabyte kernel segment. Additional smaller segments exist for Ethernet buffers. Xok was originally written using segmentation in an attempt to achieve high IPC performance. Segmentation proved to not be vital for high speed IPC, but was not removed. As previously touched upon in Section 5.3.3, the monitor may need access to 4 gigabyte segments. Xok's original segmentation should be removed, giving the monitor and other applications 4 gigabyte code and data segments. Protection between xok and its applications would still be present through the settings in the page table.

While the removal of segmentation is necessary to handle arbitrary guests, Linux did not require it. Linux, including its applications and kernel, resides entirely in addresses accessible though the original 3.2 gigabyte user segment. Therefore, removing the segmentation was a low priority and has not yet been done.

7.2 Descriptor tables

The x86 supports two types of segment descriptor tables: the global descriptor table (GDT) and the local descriptor table (LDT). Exactly one GDT must be defined for a protected mode operating system, but zero or more LDTs may be defined.

7.2.1 GDT

The x86 has a single register, called the `GDTR`, to hold a pointer to the GDT. This means that only a single GDT may be referenced at a time. This is not a difficulty when running only one operating system: A single GDT is defined and controlled by that operating system. For example, in Linux modifications to the GDT, such as the addition of a new TSS descriptor during the creation of a new process[1], are controlled solely by the Linux kernel. Allowing multiple operating systems to share the processor complicates matters. Conflicts can occur if no single authority handles additions and deletions to and from the GDT. Obviously some method must be used to mediate access to the GDT, otherwise one operating system may inadvertently use or change a descriptor defined by another.

Access to the GDT, and LDT as discussed in the next section, should be controlled for a more subtle reason. As detailed in Section 4.1, descriptors contain privilege level fields embedded in them, called DPLs. If a guest were allowed to insert a segment descriptor with a descriptor privilege level of 0, many uses of it would trap unnecessarily since the guest does not run at privilege level 0. To avoid this overhead, 0 DPLs should be changed to 1.

The initial solution attempted in this thesis was to reload the `GDTR` register when changing from one operating system to another. Problems with this solution quickly became apparent. A task switch is initiated in xok by a periodic clock interrupt. This interrupt is handled according to a gate in the IDT. The IDT may contain task gate descriptors, interrupt gate descriptors, or trap gate descriptors. Each of these

[1]See the `set_tss_desc` macro, which is defined in `linux/include/asm-i386/system.h` and used in `linux/arch/i386/kernel/process.c`.

descriptor types contains a code segment selector, referencing the code segment in which the code is to run. Therefore, at every clock tick a descriptor in the GDT is referenced. This descriptor must be the code segment expected by xok. There is no opportunity to replace a guest's GDT with xok's GDT before handling the clock interrupt. (Putting the descriptor in the LDT instead of the GDT and always keeping that LDT present does not help, since LDTs are referenced through the GDT. A descriptor in the GDT is still needed.) The code segment descriptor for xok must always be present in the GDT, and therefore, GDTs cannot be switched.

The solution implemented in this thesis is to overlay the guest's GDT on top of the xok's, changing any 0 DPLs to 1 in the process. The x86 allows up to 8192 descriptors in the GDT[6] yet xok uses only 10 of these[2]. By repositioning these 10 much later in the GDT, empty space is left at the beginning for the guest's descriptors. The reverse solution—overlaying xok's 10 descriptors on the guest's GDT—is not always possible since the guest's GDT is not guaranteed to have unused space.

Linux defines at most 1032 descriptors in the GDT. Eight descriptors are defined for the operating system itself, along with a TSS descriptor and an LDT descriptor for each of the 512 possible tasks[3]. To make ample room for the Linux GDT, xok was modified to leave the first 1536 (1.5 k) descriptors unused. Xok's original 10 descriptors immediately follow. The GDT was not created with the maximal number of empty descriptors simply to save several kilobytes of RAM. If in the future other guests require more space in the GDT, the area unused by xok can be increased.

Multiple guests

This solution is sufficient to allow a single guest to coexist with xok. Attempting to run multiple guests, however, leads to conflicts between the guests in the lower-numbered descriptors of the GDT. To deal with this, the system call that restarts the guest also optionally reloads the guest's GDT entries. The guest GDT does not need to be explicitly saved before yielding the processor, since two copies exist—the

[2]These 10 are defined in `exopc/sys/xok/mmu.h`.

[3]Refer to `linux/include/asm-i386/system.h` and `linux/include/linux/tasks.h`.

guest's unmodified copy, and the volatile version in xok's GDT. It is important that the guest GDT be restored in the same system call that restarts the guest so that these two actions are performed atomically.

If only a single guest is expected to be used, the overhead of restoring the guest's descriptors can be avoided. Currently this is set via a compile-time setting. When compiled for a single guest, xok is faster but unsafe for multiple guests; when compiled for multiple guests it is slower but safe. It would not be difficult to make xok dynamically recognize if more than one guest is running, allowing it to adjust its optimizations as necessary.

7.2.2 LDT

A guest may define multiple LDTs by inserting an appropriate descriptor for each into the GDT. Even though the x86 allows multiple LDTs to be created, only one may be active at a time since there is a single `LDTR` register, as mentioned in Section 3.3. Thankfully, having a single register does not introduce the same difficulties as it did with the GDT. No particular descriptor must be loaded in an LDT at each clock interrupt, since xok does not use LDTs at all. Still, LDTs need to be virtualized because of the privilege levels embedded in the descriptors, and to share the `LDTR` register if running multiple guests.

Virtualization of the LDT might be attempted with steps similar to that of the GDT: copy the table elsewhere, modifying the DPLs as necessary, while leaving the original intact. Since xok does not use an LDT, space would have to be allocated for the copy, either by the monitor or xok. The modified copy would be made, and the `LDTR` register would be set to it. If the guest were to read at the original memory location, the unmodified LDT would be seen.

This would be a poor solution for a number of reasons. First, the actual location of the LDT is changed. The guest operating system can see this, and therefore has access to the descriptors with modified DPLs. (This is also a problem with the GDT, but it must be done this way to share the GDT with xok. The LDT does not need to be shared with xok.) Furthermore, since each Linux task has a separate LDT, the proposed virtualization would add extra overhead to multitasking that occurs within Linux. Every task switch within Linux resets the `LDTR`, repeatedly requiring an LDT to be copied and modified. Modified LDTs cannot be cached, since the monitor is not aware of process creation and termination (and hence, LDT creation and termination) within Linux.

A better solution is to modify the DPLs of the guest's LDT directly. This still allows the guest to see the modified privilege levels, but at least the `LDTR` register has the expected value. Furthermore, it is faster to modify the guest's DPLs directly rather than making a copy and modifying that. This is the solution used in the monitor.

Multiple guests

Care must be taken to correctly share the `LDTR` register when multiple guest operating systems are running. Having an arbitrary value in the `LDTR` will not affect xok since it does not use LDTs. It would, however, affect other guests. The value that one guest loaded into the `LDTR` register almost certainly will not work for another guest.

Guest LDTs and their LDT descriptors in the GDT do not need to be saved before yielding the processor since these are not shared state. Even the `LDTR` register itself does not need to be explicitly saved, since the monitor internally saves the value set from the previous emulation of `lldt`. However, the `LDTR` register must be reloaded before restarting the guest since other guests may have changed it. The x86 provides hardware support via the TSS to automatically reload `LDTR` upon task switches. Xok does not use the task switching capabilities of the TSS; it does it through software. Therefore care must be taken to be sure `LDTR` is reloaded atomically with respect to restarting the guest. To guarantee atomicity, the register is reloaded within the same system call that restarts the guest.

As with the GDT, having only one guest active leads to an optimization. In such a case the `LDTR` does not need to be restored when the guest regains the processor. This relieves xok of the burden of validating the guest's LDT before installing it.

7.2.3 Trapping descriptor table modifications

Attempts to create descriptors or modify those in active descriptor tables (tables referred to by `GDTR` or `LDTR`) must be trapped and emulated. Descriptors may be created or modified in either of two ways. An operating system may create a descriptor table in memory and then set `GDTR` or `LDTR` to define that region of memory as containing descriptors. The other method is to directly modify the memory of an existing descriptor table that is referenced by `GDTR` or `LDTR`. Both of these methods must be trapped and emulated.

Setting `GDTR` and `LDTR`

The instructions to set `GDTR` and `LDTR` (`lgdt` and `lldt`, respectively) trap when the privilege level is not 0. The monitor takes advantage of this to capture attempts by the guest to set either register. If the `GDTR` is to be set, the monitor copies descriptors from the guest's proposed GDT into xok's GDT. All references to privilege level 0 are forced to 1 before inserting the descriptor into xok's GDT. A new system call was added to xok to allow these descriptors to be set in a safe way:

```
int sys_set_gdt (u_int k, u_int i, struct seg_desc *sd)
```

This system call is the doorway through which the guest gains the increased powers of privilege level 1. The system call therefore first verifies that the caller has root permissions by examining the capability `k`. It also verifies that the privilege level of the descriptor `sd` is not zero, and that `i` is less than the number of descriptors given to the guest so that xok descriptors are not affected. It then copies the descriptor pointed to by `sd` into xok's GDT at offset `i`. After the system call returns, the monitor write-protects the guest's original copy of the GDT. Throughout this process, the original guest GDT is left untouched. Also, the `GDTR` register is not modified; it remains pointing to xok's GDT.

Handling attempts to set the `LDTR` is similar but not identical. The `lldt` instruction traps to the monitor, which iterates over the table changing 0 privilege levels to

1. The LDT is then made active through a new system call:

```
int sys_set_ldt (u_int k, u_int16_t sel)
```

The `sel` argument is the selector on which the `lldt` instruction is to be executed. Before returning, the monitor write-protects the LDT so that DPL modifications may be trapped and emulated.

Modifying memory directly

Since guest descriptor tables are write protected, attempts to directly modify the memory may be trapped and emulated. For the GDT, the original write is performed in the monitor exactly as it would have occurred natively on the processor, but the modified descriptor is inserted into xok's GDT via the aforementioned system call. For the LDT, modifications are made directly after ensuring that the privilege level is not being set to 0: Zero privilege levels are instead written as 1.

7.2.4 Segment descriptor cache

The general virtualization problem with the segment descriptor cache was presented in Section 4.2. A specific example of this problem occurs for Linux in `linux/arch/i386/kernel/head.S`. The GDT is set with `lgdt`, then the IDT is set with `lidt`, and only afterwards are `CS` and `DS` reloaded. The `lidt` traps to the monitor, where the `CS` descriptor is used to calculate the virtual address of the instruction to emulate. The monitor does not have access to the stale `CS` descriptor as the code expects; it can only read the descriptor in the newly created GDT. The monitor needs to know what segment descriptor the processor had cached when it trapped.

Although difficult, it is not impossible to read and write the segment descriptor cache. Entering system management mode (SMM) saves all state of the CPU, including the hidden portions of the segment registers, to memory. The copy of the state may be modified as desired while in this mode. Upon exiting from SMM, the state is directly reloaded into the CPU with very little validation by the processor. Usually

SMM is entered due to a timeout from the chipset, such as when a device should be powered down. However, according to [3], chipsets often allow SMM to be invoked explicitly.

Even after ignoring the performance hit of entering and exiting SMM for each trap to the monitor and the difficulty of directly calling SMM despite chipset differences, this is not a clean solution. To truly yield the hidden portions of the the guest's segment registers, the code to enter SMM must be executed while in the guest. Allowing a timer interrupt to directly return control to xok before saving the state via SMM would modify the segment descriptor cache before it was saved. Because of these problems, SMM was never viewed as a viable route to a solution.

The solution initially attempted in this thesis was to virtualize the segment descriptor cache by having the monitor cache the previous segment descriptor(s) before emulating an instruction that would, if executed directly on the processor, make some part of the segment descriptor cache stale. In the previous Linux example, that instruction is `lgdt`, since by definition it will define a new descriptor table and make the segment descriptor cache entries for all six segment registers stale. In this case, all six descriptors associated with the segment registers should be cached before loading `GDTR`. This allows their values to be used during emulation of the following `lidt` instruction.

Of course, in some cases the soon-to-be stale entries hold real mode segments. Descriptors are not used in real mode, but an appropriate one may be created based on the properties of real mode and then cached. The base of the descriptor is the value of the segment register, shifted left by four bits. The limit is 65536. The DPL is 3, since no protections exist in real mode.

Unfortunately, this solution simply does not work. Instructions that would make some part of the segment descriptor cache stale may be trapped, but their counterparts—instructions that freshen the cache—usually cannot be trapped. The `pop` and `mov` instructions may load segment registers with new selectors, synchronizing entries of the segment descriptor cache with the values in a descriptor table. But since these instructions are untrappable, the virtualized segment descriptor cache in the monitor becomes outdated. When next invoked, the monitor will see that an entry in the virtual segment descriptor cache disagrees with the descriptor in the guest's descriptor table. There is no way for the monitor to tell if the hidden portion of the segment register should agree with the descriptor in the guest's table or the entry in the monitor's virtualized segment descriptor cache.

The solution implemented in this thesis is a collection of patches, since no robust solution could be found. When protected mode is entered, a "transition" flag is set in the monitor and remains set until `CS` is reloaded with the protected mode descriptor. This flag warns the monitor to still calculate segment bases by treating the segment registers as real mode registers. Such a flag was not needed for the `lidt` instruction since, by chance, the page tables had already been set up in such a way that using the incorrect descriptor during emulation of `lidt` still yielded the correct result. While these tricks are not perfect, they allow Linux to run. The problems with this solution are listed in Section 7.3.3.

The task register, `GDTR`, and `LDTR` have also have hidden portions which cache descriptors. These registers do not present a problem since the instructions which freshen the cache (`ltr`, `lgdt`, and `lldt`, respectively) are trapped by the monitor.

7.3 Imperfect virtualization

7.3.1 Descriptor tables

Perfect virtualization of the GDT is not possible since `sgdt`, the assembly instruction to read and store the contents of the `GDTR` register, is untrappable. If executed by the guest, this instruction will yield a pointer to xok's GDT instead of the guest's GDT as desired. Furthermore, xok's GDT is larger than expected (to accommodate the 10 xok descriptors) and contains descriptors with modified DPLs. This failure of the virtualization could be patched somewhat if the GDT could be read protected; this would offer the monitor a chance to return descriptors with the expected DPLs even though the location and size of the table itself would still be wrong. This is impossible since the guest runs at a supervisor privilege level, and pages may only be read protected from applications.

The LDT's analogue of the `sgdt` instruction is `sldt`. This instruction is not trappable either, but this does not matter. The monitor has not moved the LDT, so the offset and size returned by this instruction are correct. Even so, virtualization of the LDT is not perfect since it is visible to the guest that any DPLs inserted as 0 have been changed to 1.

In practice, the imperfections in the virtualization of the descriptor tables do not matter for Linux. A search with `grep` through the Linux source tree revealed that neither `sgdt` nor `sldt` are performed. This is not surprising since Linux creates the GDT and LDTs at known memory locations. The GDT location is known at compile time, and LDTs may be located through their descriptors in the GDT.

Failing to completely hide the modified DPLs does not matter either for Linux. The descriptors in the GDT are created with static DPLs and are never examined. Even though new TSS and LDT descriptors are created dynamically when a process is created, their DPLs are also known at compile time. Nor do LDT descriptors present problems. In general the LDT may be used by the operating system, but Linux uses it only for applications. Therefore no descriptors with a 0 DPL will ever be inserted into the LDT.

7.3.2 Privilege levels and segmentation

Exceptions and interrupts that occur in the kernel may cause the processor to construct a different stack than those that occur in an application, as was explained in Section 4.1. One of the items saved on the stack during an exception or interrupt is the code segment selector from the interrupted task. This selector contains the privilege level of the processor when it was interrupted. This privilege level is often used to deduce the format and appropriate handling of the stack.

Since the guest operating system runs at privilege level 1 instead of the expected level of 0, the test used to distinguish the two forms of the stack may fail. If the test is for equality between the selector privilege level and 0, it will always fail. This will corrupt the stack when the guest kernel has an exception or interrupt. If the test is for equality with 3, it will work correctly.

Linux accepts hardware interrupts while in slow handlers and system calls, to avoid losing these interrupts. To correctly deal with Linux and other similar guests, a way must be found to allow the test to be performed correctly when the kernel is interrupted. Exceptions in the kernel are not such an important issue, since most kernels will not experience them except for bugs. A few systems will, such as page faults in the rare self-paging kernels.

Linux tests the privilege level for equality with 0. The test is defined in `linux/include/asm/ptrace.h` and performed in `linux/kernel/sched.c`:

```
#define user_mode(regs) ((VM_MASK & (regs)->eflags) || (3 & (regs)->cs))

if (!user_mode(regs)) {
     ...
```

Because of this, trapping from the Linux kernel directly back into the Linux kernel would be incorrectly identified as coming from a user context and the stack would be mishandled.

One possible solution to this problem would be to modify the source code of Linux so that the previous test is instead a test for inequality with 3. This is, however, an unsatisfying solution since it is an obvious admission of failed virtualization. If guest operating systems tend require source code modifications before they work within the monitor, the possibility of eventually running binary-only operating systems such as Windows is slim.

Were exceptions and interrupts allowed to propagate directly from the guest kernel back to the guest kernel, this would indeed lead to imperfect virtualization. Fortunately, the exception and interrupt handling routines developed in Chapter 8 pass control through the monitor, offering the chance to rewrite the stack as necessary. In this case, the monitor fully emulates the effect of using the gate, including pushing a privilege level 0 selector into the stack. This truly makes it appear to the guest that it is running at privilege level 0.

7.3.3 Segment descriptor cache

The segment descriptor cache is not perfectly virtualized by the monitor. Although the solution presented in Section 7.2.4 works for Linux, it can certainly be broken by poorly written code in other guest operating systems. The potential for the problem appears only when executing instructions that cause part of the segment descriptor cache to become stale (such as those that enter or exit protected mode or modify a descriptor table), and then immediately using that stale descriptor on an instruction that traps.

Most applications on most guest operating systems, especially those applications designed to be portable, will not be affected since they do not perform such operations. The only relevant instructions an application might perform are those that modify an LDT. Emulators such as Wine often need such functionality to recreate appropriate segmentation for their target.

In fact, even applications that may modify the LDT will usually not be affected, since operating systems typically require modifications to the LDT to be done through a system call. The return from the system call will will restore `SS`, `CS`, and `DS` since these, at least, were used while in the kernel. It might also reload `ES`, `FS`, and `GS` or prevent their descriptors from being modified. This would prevent the application from depending on stale values.

This is the case with Linux. Modifications to the application's LDT are performed through the `write_ldt()` function in `linux/arch/i386/kernel/ldt.c`. The `SS`, `CS`, `DS`, and `ES` registers are reloaded on return from all system calls, and the `write_ldt()` function will return an `EBUSY` error if an attempt is made to modify a descriptor currently referenced in the application by either the `FS` or `GS` registers. Therefore, Linux applications will never see a failure in the virtualization of the segment descriptor cache.

Descriptors used by a kernel tend to be static; they are created at startup and are not modified. The only modifications Linux makes to the GDT while running are for applications. Affected registers are certainly reloaded before control is returned to the application.

In practice, therefore, the only places problems are likely to show up on guest operating systems are during booting while loading the GDT and when entering protected mode. If a guest operating system survives booting up, it will likely not experience other problems relating to the segment descriptor cache. Since the problem space turns out to be quite restricted, disassembling guest operating systems that present problems and modifying the monitor to handle the situation is a feasible solution.

Chapter 8

Interrupts and Exceptions

This chapter describes how to virtualize interrupts and exceptions. Recall from Section 3.8 that exactly one interrupt descriptor table (IDT) is active at a given time. Xok defines and loads its IDT when it boots. No function exists within xok to change the IDT since the semantics of an interrupt or exception, and therefore the proper handling of it, is static. However, the guest operating system will attempt to install its own IDT when it starts up. This chapter examines how these IDTs are made to coexist.

8.1 Xok interrupt descriptor table

Xok's IDT directs interrupts and exceptions from any of three sources to the correct handlers. Hardware interrupts are directed to the appropriate handler in the kernel so that the device that generated the interrupt may be serviced. Some exceptions, such as page faults and double faults, are also directed into xok. Less critical exceptions, along with a number of unused software interrupts, are sent directly to user-space so the application may handle them. The remaining software interrupts are reserved for system calls.

Access to the descriptors in the IDT is controlled by a descriptor privilege level (DPL) embedded in each descriptor. If the CPL is less privileged (i.e., larger) than the DPL of the target descriptor, a general protection fault is generated and the

Xok interrupt	Description	DPL of gate
0, 1, 3-7, 9-12, 16-31	exceptions: with user definable handlers	3
2, 8, 13, 14	exceptions: some user definable handlers	0
32 - 47	hardware interrupts	0
48 - 57, 59 - 95	software interrupts: system calls	3
58	software interrupt: system call	2
96 - 255	software interrupts: user definable handlers	3

Table 8.1: Xok interrupt descriptor table

interrupt or exception is not serviced. Hardware interrupts and exceptions are treated as if they came from privilege level 0, regardless of the privilege level in effect when it was generated, so such interrupts and exceptions will always be serviced. Xok uses the DPLs to prevent an application from explicitly calling to exception and hardware interrupt gates as if they were system calls. A brief listing of xok's interrupt descriptors and their DPLs is presented in Table 8.1.

Hardware interrupt 0 (also referred to as IRQ 0) is important in the following discussion. In xok, it is handled by descriptor number 32. IRQ 0 is generated by the clock circuitry of the motherboard at a software-defined frequency. It is this interrupt that drives the periodic switching of tasks in xok, and indeed in all other preemptively multitasked operating systems on the x86.

8.2 Potential solutions to share the IDT

The interrupt descriptor that handles IRQ 0 for xok must not be changed. If it is changed to point somewhere besides xok's scheduler, the scheduler will not be called and xok applications will cease to be scheduled. Nevertheless, the guest may wish to place its own descriptor at this offset in the IDT. Linux uses this descriptor to refer to its own scheduler. IRQ 0 is just one example of a descriptor that must be shared, but in fact all descriptors must be shared. A way must be found to resolve this contention over the IDT.

8.2.1 Switching the IDT

One possible solution is to allow controlled switching of the IDT, loading and unloading the guest and xok IDTs as needed. This method only works if xok and the guest have placed their hardware interrupts at the same offset within the IDT. Both xok and Linux place hardware interrupts at offsets 32 through 47.

To implement this scheme, xok would load the `IDTR` with a modified version of the guest's IDT just before giving control of the CPU to the guest. All hardware interrupt gates and certain exception gates in this IDT would have been modified to refer to the xok routines instead of the guest routines. The guest's descriptor for IRQ 0, for example, would be modified to return control to xok's scheduler. The general protection faults and page faults would also return control to xok, so that the monitor would have the opportunity to emulate privileged instructions. Execution would then be transfered to the guest. The guest would retain control of the processor until a general protection fault, page fault, or hardware interrupt occurs.

Xok should immediately restore its own IDT when it regains control. Restoring it soon after entry allows any exceptions due to bugs in xok to be handled correctly. More relevant, however, is that the original IDT must be reloaded before scheduling any other xok applications. If the guest was interrupted because of a fault, the fault is forwarded to the monitor to be handled. After the monitor completes emulation, it calls back into xok, requesting that the guest's IDT be restored and guest execution be resumed.

An attractive aspect of this scheme is that it allows the guest's IDT to be used almost unchanged. Slight overhead is incurred for general protection faults, page faults, and hardware interrupts; all other interrupts and exceptions in both the guest and xok are handled at normal speed. The overhead itself is actually quite small. The `lidt` instruction is an inexpensive operation to perform—optimally 6 cycles on the Pentium [5], and timing tests indicate the actual cost is similar on the Pentium Pro, at about 7 cycles. Modifying the guest IDT to return control to xok is relatively costly but could be done only once and cached in the process structure within xok.

This solution is also safe, if implemented correctly. All entry and exit to and from the guest is through xok, which is able to set and reset `IDTR`.

This solution has two main drawbacks, however. First, it requires a number of changes to critical sections of xok to modify, save, and restore IDTs. While these changes are fairly simple, localized changes such as system calls are preferred, all else being equal. More damaging is the requirement that the guest and xok store their hardware interrupt descriptors at the same offset in the IDT. Linux and xok happen to share this trait, but it is preferred to not impose artificial requirements on the guest so that the monitor may support other guest operating systems in the future.

8.2.2 Multiplexing the IDT

A better solution would be to multiplex the IDT in some fashion. If the monitor were able to gain control whenever the guest caused an exception or attempted to execute a software interrupt, the state change caused by calling through the interrupt, trap, or task gate could be simulated. The key point in such a design is that the IDT would not actually need to contain the guest's gates. This means that hardware interrupts do not need to be at the same offsets in the IDT as xok's, which potentially supports more types of guest operating systems than the first solution.

It is possible to trap all guest calls to the IDT, and therefore this solution is the one implemented in this thesis. The full solution is presented in the next section.

8.3 Multiplexing the IDT

To multiplex xok's IDT, interrupts and exceptions that originate in a guest must be trapped and directed to the monitor. This redirecting must be done without excessive overhead and without disturbing other xok applications.

<table>
<tr><th>Xok interrupt number</th><th>Description</th><th colspan="2">Interrupt or exception occurs in guest PL:
1 2 3</th></tr>
<tr><td>0, 1, 3-7, 9-12, 16-31</td><td>user definable handlers</td><td>#gpf due to lower privilege of handler</td><td>user-defined interrupt</td></tr>
<tr><td>2, 8, 13, 14</td><td>exceptions</td><td colspan="2">modify xok to redirect via trampoline</td></tr>
<tr><td>32-47</td><td>hardware interrupts</td><td colspan="2">#gpf due to DPL of 0</td></tr>
<tr><td>48-95</td><td>system calls</td><td colspan="2">modify xok to redirect via trampoline</td></tr>
<tr><td>96-255</td><td>user definable handlers</td><td>#gpf due to lower privilege of handler</td><td>user-defined interrupt</td></tr>
</table>

Table 8.2: Gaining control of a guest interrupt or exception

8.3.1 Catching guest exceptions and interrupts

Every time the guest either executes a software interrupt or generates an exception, control must be transferred to the monitor. This allows the monitor to emulate the state change that normally occurs during a control transfer through a gate, without requiring that the gate actually be present in the IDT.

Control must be transferred from the guest to the monitor regardless of the interrupt number or privilege level from which it originated. Some combinations are easy to trap due to privilege level violations. Others required minor modifications to xok. An overview of how all combinations of interrupts and privilege levels are trapped and directed to the monitor is presented in Table 8.2. Linux does not execute any services at privilege level 2; however, it is presented in the table for completeness.

User definable handlers

The interrupts and exceptions listed in the first and last rows of the table are serviced by application-defined handlers. Interrupts 96 through 255 may be generated by software; the others are those exceptions that are not directed to xok. Regardless of the source, all of these are handled the same. Xok's IDT transfers control for these interrupts and exceptions to a predefined location in the application's address space.

The application must have a few instructions at that location to jump to its desired handler.

For interrupts and exceptions of this type that occur in a guest application, xok's user-defined interrupts work well at transferring control to the monitor. But since the control transfer does not change privilege levels, the guest's stack segment and stack pointer are not saved automatically. The monitor must be sure to save this state before resetting the segment registers and running the main handler.

It is also possible to trap the user-defined interrupts and exceptions when they occur within the guest operating system. As before, the processor attempts to transfer control to a predefined location in the monitor's address space. However, the monitor's code segment is less privileged than the privilege level in effect when the interrupt or exception occurred. That is, the RPL of the code segment referred to by the gate is numerically larger than the CPL. The x86 catches this error and generates a general protection fault instead. This successfully transfers control to the monitor. The monitor checks if the source of the exception was a software interrupt by examining the guest's current instruction. Exceptions, on the other hand, cannot be immediately identified; the original exception is lost and replaced by the trap. To discover the original exception, the monitor attempts to emulate the instruction that generated the fault. Any error discovered which would result in an exception is passed on to the guest.

Initially, it was unclear if some contributory exceptions (exceptions 0, 10, 11, and 12) and page faults (14) that occur in the guest kernel could be forwarded to the monitor in this way. The x86 is unable to handle some combinations of faults serially. For example, a general protection fault (13) cannot be handled while in a stack fault (12) handler. Instead of raising a general protection fault while in the handler, the x86 will generate a double fault. The state of the processor is undefined after a double fault, making it impossible to reliably restart the previous handler. It was unclear from [6] if control could be redirected from the originally intended gate to the general protection fault handler without causing a double fault. Experiments showed that this was in fact possible. Apparently, a double fault will not occur unless the first exception is actually entered successfully. If it is not, the second fault replaces the first.

Exceptions

Exceptions, such as a general protection fault or a page fault, are asserted by the processor when an error condition is detected. Before this thesis was started, xok's IDT sent general protection faults directly to a user defined handler, without first going through xok. This was not suitable for the interrupt handling model presented here. Exceptions and interrupts that occur in the guest's kernel cannot go through a gate to a less privileged handler; such a situation generates a general protection fault as explained in the previous section. The general protection fault gate, therefore, must be able to catch and handle these exceptions. It cannot immediately send them to user-space; it must direct them to xok.

To deal with general protection faults, a `general_protection_fault()` function was added to xok. This function examines the code segment selector saved on the stack. If the selector is from the guest (currently any of the first 1536 descriptors), control is transferred to the monitor's handler via a "trampoline" function, `fault_trampoline()`. This function rebuilds the application's stack to include the guest's stack segment selector and pointer, modifies xok's stack to return to the application handler, and then returns to redirect the interrupt or exception to user-space. (A pointer to the monitor's handler was passed into xok when the guest was first scheduled.) However, if the exception did not come from the guest, `general_protection_fault()` rebuilds the stack and transfers control back to the application, as if the exception had not caused a privilege level change at all. This emulates the original behavior of the gate so that xok applications see no change. In short, the `general_protection_fault()` function exists only to satisfy the privilege level protections of the processor. Unfortunately this adds overhead to general protection faults for all xok applications. This overhead seemed acceptable, since this is only in the critical path for emulation applications and for these it was necessary.

Exceptions 2, 8, and 14 (non-maskable interrupt, double fault, and page fault, respectively) always transfer control to handlers in xok. Conceptually these are handled similarly to general protection faults, although the implementation is simpler since these exceptions never supported sending control directly to application handlers. To give control to the monitor when these exceptions occur within the guest, the handlers in xok have been modified to test the code segment selector in which the exception occurred. If the selector refers to the guest portion of the GDT, the trampoline function is called to forward the interrupt or exception to the monitor's handler. In theory, all three exceptions could be passed to the monitor and then on to the guest. In practice, receiving a non-maskable interrupt or double fault would likely indicate a bug in the virtualization. Because of this, these two exceptions are not passed on to the guest; instead, the monitor's debugger is started.

Hardware interrupts

Gates 32 through 47 are particularly simple. The DPLs of these gates are 0, but the guest is never allowed to run at this level. Any attempts by a guest to use them will fail with a general protection fault, transferring control to the monitor.

System calls

Interrupts 48 through 95 are reserved in xok for system calls. The DPLs of all these gates are 3 to allow applications to access them. (One has a DPL of 2, although this system call is deprecated.) A guest application may inadvertently use one of these interrupts, expecting to obtain a service from the guest kernel. Instead, the request would incorrectly get directed to xok. The processor cannot prevent a guest from using these interrupts (by protecting them with a more privileged DPL) without interfering with system calls performed by xok applications. Instead, to catch the use of these gates by the guest, a small test has been inserted into the code that is executed upon entry into xok. This code is generated automatically by `exopc/sys/conf/src/genvec.c` when xok is built, so it is not necessary to manually update any system calls. `genvec.c` was modified to prefix the entry code with the following test for interrupt gates that have a DPL of 2 or 3:

```
cmpl $GD_NULLS*8, -4(%esp)
jl _fault_trampoline
```

The first instruction compares the numerical value `GD_NULLS*8`, the number of bytes of the GDT allocated for the guest, against the `CS` selector pushed onto the stack. If the `CS` selector refers to a descriptor defined by the guest, control is sent to the trampoline, which in turn returns control to the monitor.

Gate number 50 in xok is the `yield` system call. The monitor developed in [2] embedded such calls in the idle loop of the guest kernel to lower CPU usage when the guest is not doing useful work. Similar functionality has been implemented in this monitor. A useful artifact of the IDT virtualization is that the `yield` system call does not need to be unprotected from the guest for this to work, since the call is not made directly from the guest to xok. Instead, all interrupts trap to the monitor. If the interrupt number matches a magic "idle interrupt" as defined in the monitor's configuration file, the monitor yields to xok and skips the instruction. Obviously, the idle interrupt must be one otherwise unused by the guest operating system, and the setting in the monitor's configuration file is therefore specific to a particular guest operating system.

8.3.2 Handling exceptions in the monitor

Once the monitor has gained control through whichever means, it must decide how to handle the interrupt or exception. A few exceptions may be handled entirely by the monitor without calling the guest's handler. In particular, exceptions that are generated solely to aid virtualization are handled by the monitor.

Page faults caused by an attempt to write to a protected structure, such as the GDT, are handled in this way. These are identified by comparing the virtual address of the write against all the addresses of all protected structures. If no match is found, the fault is forwarded to the guest operating system.

General protection faults that occur in the guest kernel are usually handled entirely in the monitor. Under the assumption that the guest kernel does not contain bugs, general protection faults indicate that an instruction needs to be emulated. If an exception is discovered while emulating the instruction (that is, if there actually is a bug in the guest kernel), the exception is forwarded to the guest kernel's handler in hopes that it may recover or shut down cleanly.

General protection faults that occur in a guest application are handled according to the appropriate descriptor in the guest's IDT. Emulation is not performed for guest applications; the appropriate handling of an application's interrupts and exceptions is decided by the guest kernel and IDT. The monitor simply helps implement this by directing interrupts and exceptions to the intended handler.

Double faults are not passed on to the guest. While the assumption of a non-buggy guest implies that double faults will not occur within the guest, they did occur while the monitor was being developed due to bugs in the monitor. Instead of passing a double fault to the guest, a debugger in the monitor is invoked.

8.3.3 Emulating interrupt, task, and trap gates

To give an interrupt or exception to the guest, the monitor must manipulate the guest's stack and registers as if the interrupt had actually occurred and control had been transferred through the appropriate gate of the guest's IDT. The exact effect on the stack and registers depends upon the type of gate and if a privilege level change would take place. The gate is read directly from the guest's IDT by the monitor.

Three types of gates exist: task, interrupt, and trap gates. Task gates are used by operating systems that take advantage of the x86's hardware task switching model. A task gate references a TSS that holds the state of the desired task. To emulate a call through a task gate, the monitor finds the desired TSS, performs validity checks as the processor would, and then loads the state from the TSS into the guest's registers.

Interrupt and trap gates are handled almost identically. The code segment descriptor referred to by the gate is retrieved and checked for validity. Some guest state—the `CS`, `EIP`, `EFLAGS`, and `SS` and `ESP` if the privilege level will be changed—is saved on the guest's current stack. `CS` is set to the new code segment, and the instruction pointer is set to the offset specified in the gate. Certain flags are cleared in `EFLAGS`, depending on whether the gate is an interrupt or trap gate. If a privilege level change is to occur, SS and ESP are loaded from the guest's TSS. (The monitor retains a pointer to the guest's TSS from when the guest last set it.)

8.3.4 Returning to the guest

If the exception was handled in the monitor, the guest's state was updated according to the emulated instruction and the guest's instruction pointer was advanced to the next instruction. Otherwise, the state was modified as if the processor had just jumped through the interrupt, trap, or task gate. In either case, the next step is to load the guest's state back into the processor and return control to the guest so it may resume execution.

If the guest operating system supports directing interrupts or exceptions directly back to applications as xok does, control may be efficiently returned to the guest application. Since the privilege levels of both the monitor and guest application are three, the monitor may `iret` to the guest application. This possibility is depicted in Figures B-1 and B-2.

Often, however, the monitor will be returning to the guest's kernel, not to the guest application, since most operating systems handle interrupts and exceptions in the kernel. This presents a problem, as the x86 prevents returning from an interrupt handler to a higher privilege level.

To solve this, a new system call was added to xok that the monitor uses to invoke the guest. The system call returns control to a guest with the specified state, after checking that the state does not violate the security of xok. The system call has the following prototype:

```
int sys_run_guest (u_int k, host_tf *tf, u_int cr0, u_int handler)
```

This system call has some security concerns as will be discussed in Chapter 11. Only certain xok applications, such as the monitor, should be allowed to execute code with supervisor level privileges. Therefore requires root permissions in the capability `k`. `tf` is a pointer to a trap frame to be restored. This frame includes the state of all segment registers, general purpose registers, flags register, and instruction pointer. The next argument, cr0, holds the virtualized state of the guest's CR0 register. Only the status of the Task Switched bit is updated in the true CR0 register. Finally, handler is a pointer to the monitor's single handler entry point.

Before instantiating the registers with values from the trap frame, several checks for validity are performed. The code and stack segment selectors must be among those reserved for the guest. The RPL of the proposed code segment selector must not be 0. Also, the new instruction pointer should not refer to xok or the monitor. If any check fails, the system call returns to the monitor with an error.

If the checks pass, xok proceeds to load the guest's state. The TS bit of CR0 is set or cleared as needed. Next, the stack pointer is set to point to the trap frame, all general purpose registers are loaded, the ES and DS segment registers are loaded, and an iret is performed:

```
movl _tf, %esp  ; restore from tf, the guest's trapframe
popal           ; restore general purpose registers
popl %es        ; restore ES register; xok used it
popl %ds        ; restore DS register; xok used it
iret            ; restore EIP, CS, EFLAGS and return to guest
```

The iret restores the CS and SS registers from the stack. The FS and GS registers are not restored since xok does not use them. Before sys_run_guest() is invoked for the very first time, the monitor must load the FS and GS registers manually. Afterwards, these registers are only saved and restored as xok task switches to and from the guest, but they are not touched during interrupts and exceptions.

8.3.5 Summary

The guest's IDT is virtualized by trapping and emulating all attempts to use the gates in it. This results in any of three main control transfer sequences, depending on which descriptor is used and from from what privilege level it is requested. See Figures B-1, B-2, and B-3 for graphical depictions of this.

8.4 PIC virtualization

Code from DOSemu was used to virtualize the programmable interrupt controller, or PIC. Writes to the PIC control, among other things, where hardware interrupts are located. By default, hardware interrupts are directed to IDT offsets 0 through 15, which would conflict with the processor's own exceptions. Linux reprograms the PIC with a series of `out` instructions in `linux/arch/i386/boot/setup.S` to move the hardware interrupts to offset 32. These instructions are trapped and virtualized so that the monitor knows where to send simulated hardware interrupts.

The PIC also controls which hardware interrupts are enabled and disabled. This is virtualized and respected by the monitor. For example, if the guest disables IRQ 0, it does not expect to have its scheduler called. Xok still schedules its applications, and therefore the guest may lose and regain the CPU. If the guest's virtualized IRQ 0 is disabled, the monitor will restart execution of the guest at exactly the same point it lost the CPU. If on the other hand IRQ 0 is enabled, the monitor will restart the guest in its scheduler when the CPU is regained.

8.5 Imperfect virtualization

Among other things, the PIC controls the rate at which timer interrupts are generated by the clock circuitry. The instructions to set this rate are trapped but are not currently virtualized. The guest operating system gets a simulated IRQ 0 every time the monitor is scheduled by xok. The guest's scheduler may then schedule its own applications. This rate is set by xok and is unrelated to the rate the guest attempted to program the PIC. Timing sensitive applications may not function correctly because of this.

Chapter 9

Device Virtualization

Only rudimentary virtualization of some devices was completed for this thesis. That is reviewed here, along with ideas on virtualizing other devices.

9.1 BIOS

A small virtualized BIOS image is mapped into memory at `0xf000` before the guest is booted. This image contains only tables describing the virtualized hardware but no code such as BIOS interrupt handlers. The interrupt functions supported by a real BIOS, such as those to write to the screen or disk, actually exist as part of the virtual 86 mode interrupt handler in the monitor.

Table 9.1 lists the data tables present in the virtualized BIOS. The handler for the first interrupt number listed is implemented within the monitor. This handler returns a pointer to system configuration data stored in the BIOS at the offset listed. The other vectors are different; they are actually not interrupts. Real mode applications, potentially including the boot sequence of protected mode operating systems, read this hardware-defined data by looking up the vector number in the real mode IDT, which yields the offsets and data listed.

BIOS interrupt vector	offset	description
0x15, AX = 0xc0	0xe6f5	System configuration data
0x1d	0xf0a4	Video parameter table
0x1e	0xefc7	Diskette parameter table
0x41	0xe401	Hard disk parameter table

Table 9.1: Data tables in the virtualized BIOS

9.2 Keyboard

A partial virtualization of the keyboard was implemented for real mode through the BIOS interrupts. Keys may be read via `int 16` with `AX` set to 0, and the availability of a key may be checked with the same interrupt with `AX` set to 1. The subfunctions to read shift states and set the keyboard repeat rate have not been implemented.

The keyboard has not been virtualized for protected mode guests.

9.3 Video

Virtualizing text mode video is a relatively simple process since text screens are regions of memory mapped at a known locations. Standard color text modes (that is, text modes for CGA, EGA, and VGA cards) consist of four kilobytes of memory mapped at offset `0xb8000`. Text modes for Hercules cards are mapped at offset `0xb0000`. Text is written to the screen either through a BIOS `int 0x10` video call (while in real mode) or by writing directly to memory (while in either real mode or protected mode). The appropriate page of memory at either `0xb0000` or `0xb8000` is identity mapped (the virtual address is mapped to the same physical address) when the guest operating system is started in virtual 86 mode so that writes to them are seen by the video hardware.

The `int 0x10` BIOS call for writing to the screen is implemented within the monitor, not the virtualized BIOS. This incurs extra overhead for BIOS writes from real mode, but this is not significant since the targeted guests primarily use protected mode. Linux only uses this BIOS interrupt to display a few messages while booting, so ease of implementation was more important than speed.

Writing directly to memory, as is done in protected mode, currently incurs no overhead. Although the core of the monitor (the portion that virtualizes the x86 itself) was designed to handle multiple, simultaneous guests, the video virtualization has not been developed to that extent. This is due in part to the immaturity of xok's console subsystem. If xok had a better console subsystem, access to the screen would need to be more tightly controlled. Currently, if multiple guests run under xok, their displays will intermingle in the physical display. Linux, by contrast, virtualizes the text screen so that several "hot keys" may switch between its applications. A signal is sent to the Linux application when it gains or loses the physical console. Xok does not offer this. If it did, the page of memory containing the text would be unmapped for the guest when the monitor lost access to the physical console, perhaps to another guest or to a native xok application. Reads and writes to the text screen would trap since the pages would be unmapped, allowing video to be virtualized entirely. When the monitor regained the console, pages could be remapped to these regions and the display recreated. In summary, the text virtualization offered by the monitor may mature more as xok matures.

Graphics mode is far more difficult to virtualize than text for several reasons. First, there is no single standard to adhere to. Many graphics chipsets exist, each with their own interface. Even if a single chipset is picked, a register-level virtualization is very complex. During the initial stages of the project, graphics virtualization code was taken from the DOSemu sources, in hopes of leveraging off of this existing work. Time constraints prevented this from being developed.

9.4 Disk

A fairly complete real mode virtualization of the disk interface was implemented through the BIOS interrupts. The disk has not been virtualized for protected mode. Linux loads the entire kernel into memory via the BIOS disk interrupts then proceeds to boot it. Implementation of BIOS disk interrupts allowed testing of protected mode code to be done, even without a virtualized disk interface for protected mode.

The real mode virtualization includes interrupts to read, write, and verify sectors. The virtualization will also return the disk parameters when queried and let new ones be set. Some subfunctions of the interrupt do nothing but return a success code to the caller, such as a request to position a drive's heads. Other subfunctions are virtualized by returning an error code indicating that the function is not supported, such as is done when the disk change status is queried.

Full disk virtualization in protected mode is difficult for the same reasons protected mode video virtualization is difficult. Many chipsets exist that are incompatible at the register level. One possible solution, discussed in [2] and applicable to both video and disk, is to create a device driver for the guest operating system that is aware of the monitor and hosting operating system. Such a driver could translate video or disk calls originating in the guest operating system into calls understood by xok.

Chapter 10

Monitor Implementation

This chapter gives a view into the implementation, configuration, and usage of the monitor. Wherever possible, the monitor was written in a way that simplified the process of decoding and emulating instructions. In some places this meant using lookup tables to pick the correct emulation function; in others, creating highly parameterized functions to avoid reimplementing (perhaps incorrectly) similar emulation algorithms. That in combination with a generous number of assertions created a monitor that was be relatively easy to debug.

10.1 Exception handling

General protection faults, page faults, and other exceptions that occur in the guest's kernel are first sent to xok then directly to the main entry point in the monitor. Other exceptions originating in a guest application do not pass through xok. Instead they are first routed through small assembly wrapper functions in the monitor, defined in `exopc/bin/monitor/entry.S`. Since a privilege level change did not occur, the processor did not automatically save the stack segment selector and stack pointer from the guest. The wrappers ensure that all such state is saved before calling the main handler function, `exc_C_handler_header()`. This is the main entry point into the monitor; it contains setup code that is relevant for all exceptions. It sets a pointer to the recently saved guest state so that it may be manipulated during emulation, validates the guest's state (if compiled with debugging enabled), parses any instruction prefixes, and verifies that the exception did not in fact come from the monitor itself since that would indicate a bug. Once this common code is completed, the appropriate handler is selected and called. If the exception came from a guest application, it is forwarded to the guest operating system. Otherwise, control is transfered to `exc_13_C_handler()` for general protection faults or `exc_14_C_handler()` for page faults. Other exceptions could have turned control over to the monitor, but as discussed in Section 8.3.1, these are caught by the monitor as general protection faults when they come from the guest's kernel.

The general protection fault routine is table driven. The first instruction byte after the prefixes is used as an index into a table of emulation functions. Not all possible instruction bytes have an emulation function in the table, since many instructions will never trap to the monitor. A few others are not yet implemented since Linux did not use at them all or did not use them in a way that would cause a trap. Since Linux was used as the reference guest operating system, instructions it used and trapped on were given priority over those that it did not use or did not trap on. If an unimplemented instruction is encountered, the monitor's debugger is started, showing the instruction and state of the virtual machine.

Some instruction bytes on the x86 are actually prefixes to two byte opcodes. The `0x0f` opcode is one example, and the coprocessor has its own range of two byte opcodes. The emulation function for `0x0f` is itself a lookup function, keying on the next instruction byte.

Each emulation function must finish decoding the instruction and then emulate it. For instructions with no operands, no more decoding needs to be performed. More complex instructions may have to decode a register, memory location, or even a variable offset from a constant memory location. Logic to do this and other fundamental instruction decoding tasks was placed in highly parameterized functions so it could be reused often.

Certain instructions (`ins`, `movs`, `outs`, `lods`, `stos`, `cmps`, and `scas`) may be preceded by a repeat prefix. Such a prefix indicates that the instruction should be repeated until a condition is no longer true, or no longer false, or until a counter is decremented to zero. Rather than implementing this repeat test repeatedly, once for each instruction emulation function, a generic `repeat` function was created. The emulation functions for these instructions were split into two halves: The first contains setup code, moving computation outside of the repeat loop for speed. The second half is the execution function, implementing one iteration of the instruction. The `repeat` instruction calls the setup function, then calls the execution function according to the logic of the repeat loop. This simplified the implementation of these emulation functions, since repeat logic is concentrated in one location. If the instruction is not preceded by a repeat prefix, the `once` function is called instead, which sets up and then performs only one iteration.

10.2 Compilation

For the monitor to successfully run, both xok and the monitor must have been compiled with the `EXO_HOST` environmental variable set. This setting causes various changes to be made during the compilation of xok, such as allocating extra descriptor entries in the GDT and defining several new system calls to support the monitor. This variable must be set during the monitor compilation since the monitor source code includes several affected xok header files.

A change must also be made to the file `exopc/sys/conf/trap.conf`. The line:

```
0x0d    tNT     $UIDT_VECT_TO_ADDR(0x0d)         # GPF
```

must be changed to:

```
0x0d    iAe     general_protection_fault         # GPF (if EXO_HOST)
```

which will allow general protection faults to be routed correctly. This change must be made rather than relying on the environmental variable since this file is not preprocessed. The `general_protection_fault()` function could, in theory, always be used whether or not monitor support is desired since it will direct faults back to xok applications as expected. However it was desirable to avoid all modifications to xok when monitor support was not compiled in.

The monitor may also be compiled with the `MULTIPLE_GUEST` define set in the makefile, to allow multiple guests to run simultaneously. Not having this defined removes some overhead when running just one guest.

10.3 Configuration

The monitor's configuration is controlled through a configuration file read at startup. This file controls the number of debug messages printed, the debugging environment, the virtual disks present in the system, along with the virtualized size of RAM and type of display adapter. Full documentation for correct usage of these options is contained in the file itself. The default location for this file is `/etc/monitor.conf`.

10.4 Debugger

The monitor contains a simple debugger, which allows the state of the guest to be examined and manipulated. This is not a true debugger; it does not use the processor's ability to set breakpoints or single step. Even so, it proved quite useful during the development of the monitor. When perplexing bugs were encountered, a call to the debugger was hard coded into the monitor, triggered by the state known to accompany the bug. The debugger is also invoked automatically when unhandled instructions or bugs in the guest operating system are encountered.

Configuration options also allow the debugger to be started at special times, such as after initialization but just before transferring control to the guest for the first time, and when a `hlt` instruction is encountered.

Some of the debugger's most basic commands are listed in Table 10.1. The full set may be obtained by typing `help` at the debugger's prompt.

Command	Description
`r`	Display guest registers, data at stack pointer, and data at instruction pointer.
`g`	Go; resume guest execution.
`reg=val`	Change the value of a register. `reg` is a register name; `val` is in hexdecimal. For example, `eip=91b2`.
`dump [[segment:]offset] [n]`	Display `n` bytes of memory starting at `[segment:]offset`.
`search n string`	Search memory for `string` starting at address `n`. Stops when found or when an unmapped page is encountered.
`pte n`	Displays guest pte, host pte, and physical page number for address `n`.
`gdt n`	Display the guest's `n`-th GDT descriptor.
`idt n`	Display the guest's `n`-th IDT descriptor.
`help`	Display a full list of debugger commands.

Table 10.1: Basic debugger commands

10.5 Usage

In its simplest form, the monitor is invoked with no arguments. All configuration data is read from the default configuration file. If a different configuration file is desired, it may be specified with the `-F` option.

10.6 Status

The monitor was developed and tested primarily with the Linux 2.0.36 kernel, and therefore it is expected to work best with Linux. The Linux kernel manages to load itself into memory, detect the memory size, initialize the page tables, define system tables such as the IDT, GDT, and LDT, and then enter protected mode. It then initializes trap handlers, IRQs, the scheduler, calibrates to the CPU speed, and sets up numerous kernel data structures before the virtualization fails. The `buffer_init()` call in `linux/init/main.c` allocates memory, but the memory has not been mapped into the page table at that point. It has not yet been determined if this is due to a bug in the page table virtualization or a failed virtualization in the Linux kernel's `malloc`.

The monitor also has some success at booting FreeDOS, although the lack of a register-level virtualized disk keeps it from loading COMMAND.COM. FreeBSD was also tested, but it fails to correctly detect the size of the virtualized physical memory, causing the monitor to assert.

The monitor was originally based on code taken from early versions of the DOSemu project. However, as the monitor evolved almost all of this code was rewritten. Currently the PIC, keyboard, and video virtualization code are the only parts that are still largely unmodified, since these areas have not been worked on as much as the rest.

Chapter 11

Xok

This chapter discusses xok as it relates to this thesis.

11.1 Helpful xok features

Several features of xok proved useful during the implementation of the monitor. Those are listed here along with a brief explanation of how they were used.

11.1.1 User defined trap handlers

User defined trap handlers allow certain interrupts and exceptions to be redirected to an application-level handler, without the overhead of first passing through the kernel. The monitor uses such handlers to catch interrupts and exceptions from the guest application when possible, reducing the cost of fully virtualizing the IDT.

11.1.2 Batch system calls

Xok also supports gathering a number of system calls together and sending them to the kernel en masse. This batching process avoids the cost of repeatedly calling from the application to xok and returning. System call batching is used by the monitor when a page table pointer is loaded into the `CR3` register. Page table entries are set in groups, rather than one at a time, speeding page table switches.

11.2 Security of the modified xok

The primary goal of this thesis was to demonstrate that a protected mode virtual machine monitor could be implemented as a normal application under xok. Implicit in this goal was a secondary goal: to do so without making xok insecure or unstable, and therefore unsuitable for other xok applications.

There exists a fine, almost invisible, line between trusting the guest operating system yet not trusting the monitor that loads it. This was the original goal. Trusting the guest allows more instructions to be executed natively on the processor, which translates into a faster virtualization. Trusting the guest certainly seems reasonable, since Linux, for example, is trusted when it is run on its own. Yet the monitor, ideally, should not be trusted. The monitor is a complex application which is not as battle-tested as most operating systems. It certainly contains bugs and therefore should not be included in xok's trusted code base.

The distinction between a trusted guest operating system and an untrusted monitor is, in practice, somewhat artificial. The monitor loads the guest, and is therefore being trusted to load one that is non-buggy and non-malicious. In an attempt to lessen this threat, the monitor must have root permissions to start a guest OS (i.e., to call `sys_run_guest`). This prevents malicious users from executing arbitrary code at privilege level 1.

In practice, the desire to not trust the monitor leads to system calls with rigorous error checking. This prevents a buggy monitor from loading incorrect data into system structures and crashing the system. Requiring root permissions to execute guest code at a supervisor privilege level is another form of error checking; this guards against the problem of a malicious guest. Together, these checks make it possible to run a guest operating system while still keeping xok stable.

Chapter 12

Performance

While the monitor has not reached the point where Linux applications may run, largely due to the lack of virtualized devices, the x86 itself is fairly well virtualized. Therefore, several micro benchmarks are presented in this chapter, analyzing specific functions of the monitor rather than the performance of full applications. In lieu of application benchmarks, a single larger benchmark—timing a portion of the Linux kernel boot sequence—was also performed.

12.1 Benchmarking methodology

All benchmarks were performed on a 200 Mhz Pentium Pro. Five measurements were taken for each data point. The highest and lowest numbers were discarded, and the remaining three were averaged. The four environments compared were:

1. Native execution: No monitor or emulator was used.

2. Bochs: The latest version of Bochs, version 990219a, compiled with optimizations and hosted on Linux 2.0.36.

3. VMWare: The latest beta version of VMWare, build 135, hosted on Linux 2.0.36 and using an XFree86 X server optimized for VMWare.

4. Monitor: Compiled with optimizations and debugging symbols enabled, hosted on xok.

The most recent version of VMWare (build 135) was used in these tests. This version is still beta and therefore includes debugging symbols and assertions. The source code to VMWare is not available so these cannot be disabled for speed. VMWare was, however, optimized by using a custom VMWare-aware X server, which removed some overhead associated with virtualizing video devices.

A modified Linux 2.0.36 kernel was used as the guest in all tests. Instructions were inserted into the kernel around the code to be timed. These instructions read a cycle counter at the start and finish of the test and printed the difference. The `rdtsc` ("read time-stamp counter") instruction, present on the Pentium and later processors, was used to obtain the time stamps. The counter is incremented every clock cycle. These timings are a measurement of wall-clock time, since they includes time spent in all tasks. The benchmarks, then, are not perfect measurements of each virtualization environment but rather an indicator of real-world performance. The test machine was idle when the tests were performed.

The `rdtsc` instruction does not trap when called from applications on the x86 unless a bit in the control registers has been set. This bit is not set when the x86 boots, nor does xok set this bit. Therefore Linux was able to time itself with no timing-related overhead under the monitor.

It was important to know that VMWare directly executes `rdtsc`, rather than virtualizing it at a potentially untrue rate or trapping it and incurring overhead. This was ascertained by modifying the boot sequence of the Linux kernel to repeatedly read the time-stamp counter and print the value to the screen. Another program was written that also repeatedly read and displayed the count. The modified Linux kernel was booted in VMWare under Linux; the other program was run in a separate window. The time stamp counts remained perfectly synchronized, demonstrating that VMWare directly executes `rdtsc` as desired.

Originally Bochs did not support emulation of `rdtsc`; instead, it panicked and exited when the instruction was encountered. An "emulation" of the instruction was added to Bochs, which simply runs the instruction and loads the results from the real registers into the virtualized ones. This modification made Bochs nonportable since the emulation function was written with inline x86 assembly, but allowed timing tests to be performed.

12.2 Benchmarks

Each benchmark was done within the context of a Linux kernel, modified to execute and time the micro benchmark. Three micro benchmarks were performed, along with a more comprehensive benchmark.

12.2.1 Computation: gzip

The gzip benchmark times how long it takes to uncompress a 932 k (382 k compressed) Linux kernel during booting. Paging has already been enabled and the kernel has been loaded from disk when the test occurs. This test is strictly computation, and it therefore suggests the optimal performance of the monitor. It may not represent optimal performance of other emulators and monitors, depending upon their design.

12.2.2 Page table insertions: multiple pte

Before uncompressing the kernel, Linux identity maps the first four megabytes of memory. Linux then uncompresses the kernel and performs other setup functions before finishing the memory initialization in `linux/arch/i386/mm/init.c`. This final setup of the page table is the basis of the second benchmark.

This test counts the number of cycles required to populate a page in the page table with 1024 page table entries, mapping four megabytes of memory. Each entry is created by a separate write to the page table. This benchmark indicates how expensive it is to modify a single entry in the page table.

12.2.3 Page table insertions: single pte

Other common manipulations to perform on the page table are to replace the entire table by reloading `CR3` or to replace just a table in it by changing a pointer in the hierarchy. Since Linux changes page tables each time it schedules a processes, it is desirable to keep the emulation of such tasks fast.

This benchmark measures the cycles required to insert a second-level table, containing 1024 page mappings, into the page table by changing a pointer in the top-level table. For the monitor, this includes time taken to remap the guest's physical pages to free pages under xok.

12.2.4 Comprehensive: boot

To gain a more realistic idea of the over all performance of each emulation environment, a portion of the Linux boot sequence was timed. The timer was started immediately after loading the compressed kernel into memory. It was stopped before the call to measure the CPU speed in `linux/init/main.c`. The time taken to calibrate the processor speed depends upon when timer interrupts arrive. This is not a function of emulation efficiency, and therefore including it could skew the results.

12.3 Results and Analysis

Micro-benchmarking results are presented in Table 12.1. These numbers hold some interesting results, and suggest that VMWare and the monitor developed in this thesis do not share a similar design despite the fact that both are virtual machine monitors.

gzip

For the gzip test, the monitor and native execution scored similarly. This shows that for plain computation, the monitor adds no overhead. Oddly, though, the monitor consistently achieved slightly *faster* times than native execution. This could be due to caching issues in any of several ways.

One difference between native execution and execution under the monitor is that the pages allocated by the monitor may already have cache entries; those allocated during native execution do not. It was postulated that there is a processor-dependent setup cost when an empty cache line is to be filled. To eliminate this possibility, the

Hosting environment	gzip	multiple pte	single pte	boot
Native execution	28.1	0.0146	0.000352	170.
Bochs / Linux	11800.	2.94	0.0308	8300.
VMWare / Linux	45.0	0.0394	0.442	111.
Monitor / xok	26.8	17.2	20.3	221.
Monitor (batch) / xok			10.4	

gzip: Uncompress a 382 k Linux kernel during booting.
multiple pte: Individually insert 1024 page mappings into guest's page table.
single pte: Set a pointer to a page table with 1024 existing page mappings.
boot: Partially boot a 282 k Linux kernel.

Table 12.1: Speed comparison of four environments, in millions of cycles

`wbinvd` assembly instruction was inserted into the Linux kernel before the call to uncompress the kernel. This instruction flushes all internal and external caches of main memory. Thus execution under the monitor and native execution both commenced uncompression with empty caches. Nevertheless, execution times were unchanged.

Another difference is that under native execution the physical pages are allocated contiguously. The monitor, however, remaps physical pages, causing the allocation to be more random. It is plausible that executing the gzip algorithm on contiguous physical pages interferes with the processor's caching strategy more than a somewhat random distribution.

Testing, however, proved this not to be the case. The Pentium Pro's performance counters were used to count the number of cycles spent waiting for data after cache misses. For native execution, 0.606 million cycles were spent waiting. Execution under the monitor spent 0.616 million cycles waiting; this number is inflated since one trap and one system call were required to start the performance counter from user-space. Regardless, stalls due to cache misses cannot explain the 1.37 million cycle difference in uncompression time. A satisfactory explanation has not been found.

multiple pte

The monitor scored poorly on both page table tests. For the first test, multiple pte, poor performance was expected. Each write to the page table results in numerous traps and system calls. As illustrated in Figure B-3, the general protection fault caused by the write is delivered to xok, then returned to the monitor where it is handled. The monitor executes at least three system calls to emulate the effects of the write. It must allocate a page in the real page table, and then set the guest page table read-write and back to read-only while performing the original write. In reality, the monitor is not yet well tuned so more system calls than this are performed.

Bochs managed to execute faster than the monitor on this test since it did not have the repeated overhead of exceptions and system calls. Full emulation is expensive at times, but on a small loop containing privileged instructions it may win out over a virtual machine monitor.

VMWare achieved speed comparable to that of native execution. It is unknown how VMWare handles memory beyond the fact that each guest operating system has a page table separate from VMWare. (This was determined by examining the source code for VMWare's Linux kernel modules, although the source code for VMWare itself is not available.) Based on the benchmarks taken here it is possible that VMWare uses a scheme in which pages of memory are mapped on demand. This would move overhead away from page table manipulations and amortize them over the execution of other instructions. If this were combined with dynamic translation of trapped instructions to remove subsequent trap overhead, execution times of loops such as in this benchmark could approach hardware speeds.

single pte

The monitor also did poorly on the second page table test. It was originally expected that the monitor would score better on this benchmark than the previous one since repeated traps into the monitor were not needed. However, the monitor's memory handling functions were designed strongly favoring simplicity over efficiency.

Repeated system calls are made to query items that in a more efficient design would be cached locally. For example, the monitor often validates memory references by reading page table entries through an xok system call. Ideally, the page table should be readable by xok applications to reduce the cost of such operations, although this modification to xok has not been attempted. Another approach would be to read from the guest's page table and then translate the result to true physical pages.

Xok supports queueing system calls together and executing them as a batch. This avoids the overhead of repeatedly calling between the kernel and the application. By rewriting several guest page table routines to batch page table modifications, execution time was cut in half. Further improvements may be possible by batching other types of system calls, such as those to mark guest pages read-write and read-only.

boot

The relative cycle counts for this test were as expected, except for VMWare. It is difficult to hypothesize how VMWare achieved such speed since the source code is not available. The monitor was somewhat slower than native, and Bochs was significantly slower. The execution times of all forms of emulation, except for Bochs, were longer than in all other tests since a larger portion of the kernel was being timed. The kernel used in this test was smaller than the one used in the previous three. Bochs turned in a faster time on this test than it did uncompressing the larger kernel. This suggests that Bochs is slowest on computation and data intensive algorithms.

12.3.1 Monitor invocation costs

It was not possible to directly time all execution paths into and out of the monitor (Figures B-1, B-2, and B-3) since user-level applications cannot yet be loaded. However, accurately measuring the cost of trapping to and from the guest operating system, as depicted in Figure B-3, will give an idea of the costs guest applications will incur.

Description	Average cycles
Trap from and return to guest	917
Monitor setup	227
System call	323

Table 12.2: Cost breakdown of monitor invocation, in cycles

It is especially interesting to measure these costs considering the high price of page table modifications. The cycle counts obtained in the previous section were large given the number of system calls made. An obvious question is whether the remaining time is spent directing exceptions to and from the monitor—implying a fundamental design flaw in the monitor—or if the time is spent in the kernel servicing the system calls, in which case only the memory system needs reexamined.

To answer this, a test was developed which measured the minimum time required to trap to the monitor, perform an instruction emulation which may call into xok a number of times, and return to the guest. The results of these tests are shown graphically in Figure B-4. The top line represents the cycles required for the full emulation path (including system calls) from guest operating system to monitor and back to guest. The bottom line was obtained by doing the system calls immediately upon entry into the monitor; no monitor initialization or instruction parsing was done. Thus the difference between the two lines represents the minimum number of cycles needed to parse an instruction and invoke an emulation function. The slope of the lines is the cost of invoking an xok system call. These costs are listed numerically in Table 12.2.

These numbers show that the overhead associated with a trap to the monitor is not unreasonably large when compared to the cost of a single system call. The slow emulation of page table operations is a design flaw in the memory virtualization of the monitor, not a general inefficiency in the monitor design itself. Also, the xok system calls to manipulate the page table seem slow, considering that approximately 4100 calls were made for the single pte test under the monitor without system call batching. It is likely that both xok and the monitor could be tuned to improve these benchmarks. However, a better-designed memory virtualization system could potentially avoid both of these problems entirely by not manipulating individual page table entries in this manner.

Chapter 13

Conclusion

13.1 Future work

This monitor proves that x86 virtualization can be done through an application under xok. However, more work needs to be done before it is usable. Some directions for future work are outlined below.

13.1.1 Memory

The memory management model of the monitor should be made more robust. This is currently the weakest and slowest aspect of the virtualization. VMWare pays a performance penalty on general computation, perhaps in part to better virtualize the 32-bit address space. In retrospect, the design tradeoff made in this thesis between high performance and good virtualization of the address space should have favored the latter more. With the current implementation, problems in the memory virtualization abound. Further development and debugging can solve many of the problems mentioned below. However when these problems are viewed together, it becomes clear that that a redesign of the memory handling system would be the cleanest solution.

Pinning pages

Currently, when the guest frees a page table entry, the monitor mirrors this release in the true page table. This causes xok to potentially place the page back on the free list, where other applications may claim it. Therefore, if a guest operating system removes a page from a page table and later adds it back, the contents of the physical page may have changed. This is incorrect virtualization. To fix this, pages should be retained by the monitor even if the guest is no longer referencing them. This will, however, lead to wasted memory if the guest is not going to use the page again.

This problem also surfaces when the monitor simulates the switching of page tables between guest applications by modifying its own page table. Pages owned by the application losing the CPU cannot be completely freed; they must be retained so that they may be restored when the application is rescheduled. Pages are reference counted in xok; removing the page from the page table will cause it to be placed on the free list. Xok does, however, allow pages to be pinned for pending DMA. Such pages will not be placed on the free list even if they are not referenced by a page table. No way currently exists in xok to add a pinned page back to a page table. If this were done, pinning a page may be the solution to this problem, allowing pages not referenced by a page table to be retained.

A simpler and potentially faster solution would be to modify xok to allow an application to optionally use a separate page table, similar to the modifications VMWare loads into the hosting kernel. Applications which do not opt for a separate page table would not be penalized with excessive TLB flushes during system calls, yet the option of a separate address space would be available to guest operating systems.

Segmentation

Segmentation should be removed so that guests which execute instructions in portions of the upper 0.8 gigabytes may be emulated by the monitor. This would allow a wider range of guests to run under the monitor. Better, however, would be to entirely remove the possibility of virtual address conflicts between the guest and the host with the aforementioned xok modification.

Demand paged physical memory

Demand paged physical memory should be implemented so that memory is not wasted on guests which do not require much, as discussed in Section 6.1.1. This has been started but is not yet finished.

Page release hints

Page release hints allow the monitor to free physical pages as they are placed on the guest's free list [2]. This is an optimization that requires a small modification to the guest operating system's source code. This would be useful, but is not implemented.

13.1.2 Supported guest operating systems

The monitor was developed using Linux. Development and testing should be broadened to include more guest operating systems. This would exercise more aspects of the monitor, helping to develop a better virtualization.

13.1.3 Virtualized devices

Devices need to be virtualized. A good disk virtualization is most important since this will allow the operating system to finish booting and load user applications. A full keyboard virtualization should follow. The current handling of text video is tolerable, although a better virtualization and even graphics virtualization would be useful. Finally, a NIC virtualization would allow multiple guests to interact, as was done in [2].

13.1.4 Hardware interrupts

Hardware interrupts need to be virtualized and/or forwarded to the guest operating system. For example, if a packet arrives on the network for the virtualized operating system, a virtual hardware interrupt should be sent to the guest to cause it to retrieve the packet from the virtual network.

13.2 Conclusions

One lesson to take away from this thesis is that the x86 does not need to be perfectly virtualized to host an operating system. The monitor presented here has imperfect virtualization of numerous instructions, yet this does not affect Linux. Certainly operating systems could be constructed which are "hostile" to virtualization, relying on aspects of instructions which are not easily virtualized within a monitor. Linux, however, is not this way.

Memory virtualization proved to be more troublesome than instruction virtualization. In retrospect this is not so surprising. "Simplifying" the problem of memory virtualization by changing as little as possible (forcing the guest operating system down to privilege level 1 instead of 3, and overlaying the address spaces instead of creating a separate page table) resulted in more difficult implementation, difficult to solve bugs, and very low performance when modifying page tables. The rest of the thesis attempted to use the hardware as much as possible; this should have applied to page table handling also. Modifying the host operating system to allow page table switches, as VMWare does, would have resulted in a much cleaner memory implementation.

Xok's mantra of exposing the hardware did not prove as useful to this monitor as was originally expected. This may change as the monitor is developed more fully; for example, explicit revocation of the processor may allow the monitor to save device state when multiple guests are running. In the basic monitor implementation presented here, such functionality was not needed. Xok certainly did have other useful features, though: batch system calls and user-defined trap handlers optimized particular aspects of the virtualization.

Appendix A

Tables

`int sys_run_guest (u_int k, host_tf *tf, u_int cr0, u_int handler)`

Restores the specified state and restarts execution the guest.

`int sys_set_gdt (u_int k, u_int i, struct seg_desc *_sd)`

Sets the specified entry of the GDT to the descriptor pointed to by `_sd`.

`int sys_set_ldt (u_int k, u_int16_t sel)`

Loads the specified selector into the LDTR register.

`int sys_set_tss (u_int k, u_int16_t gdt_selector)`

Loads some state from the TSS descriptor referred to by `gdt_selector` into the current TSS. Only regions of the TSS unused by xok are modified.

Table A.1: Summary of new xok system calls

Appendix B

Figures

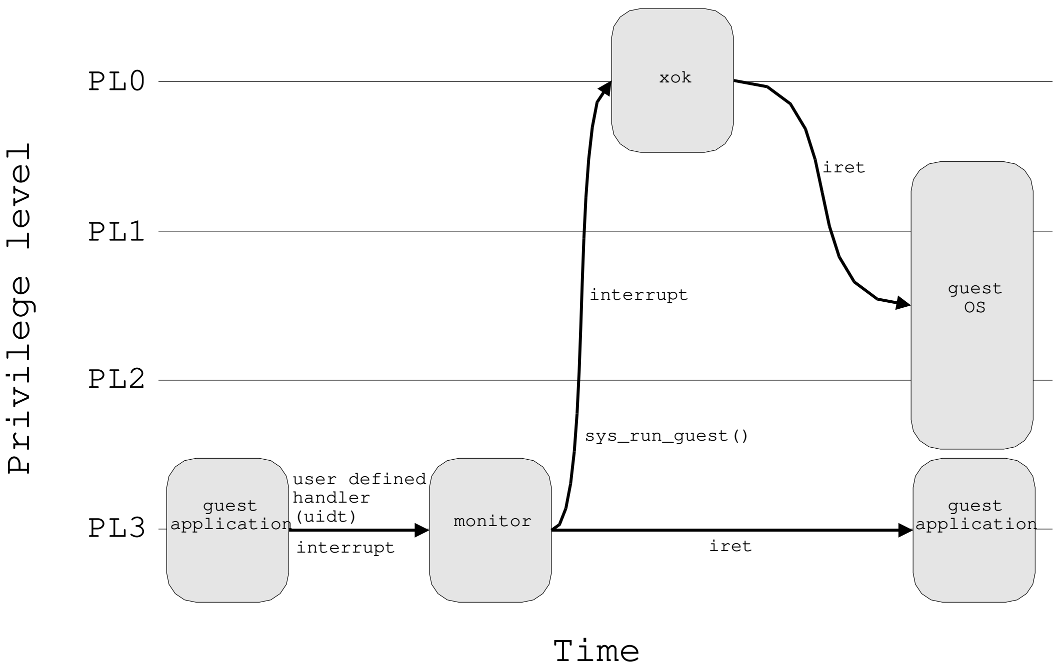

Figure B-1: Interrupt and exception handling from guest application using user-defined handlers.

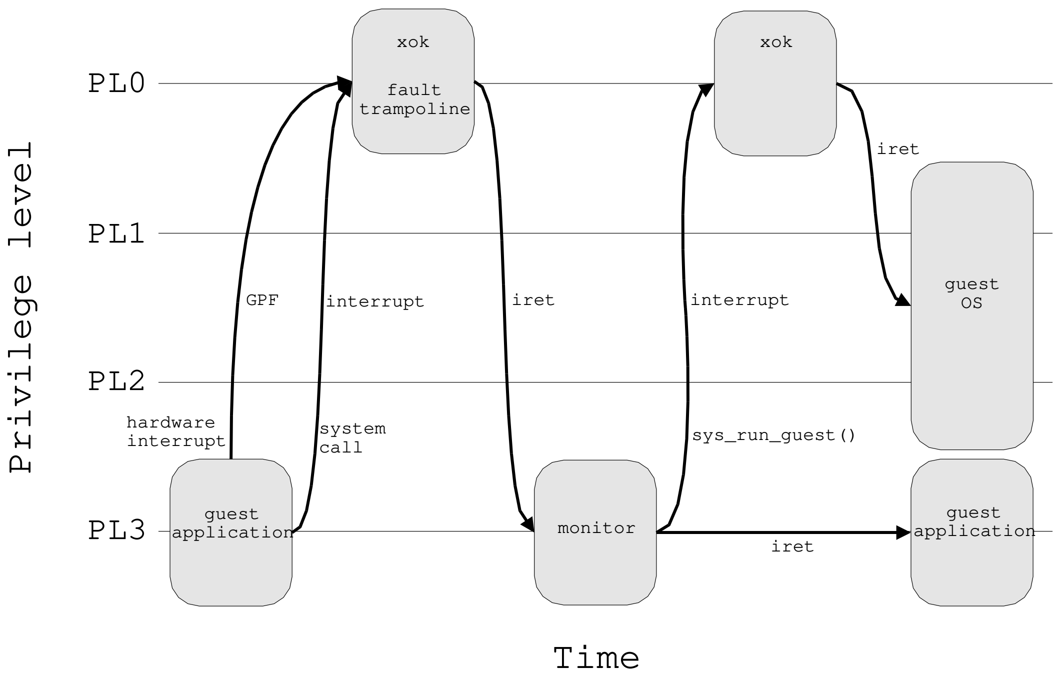

Figure B-2: Interrupt and exception handling from guest application.

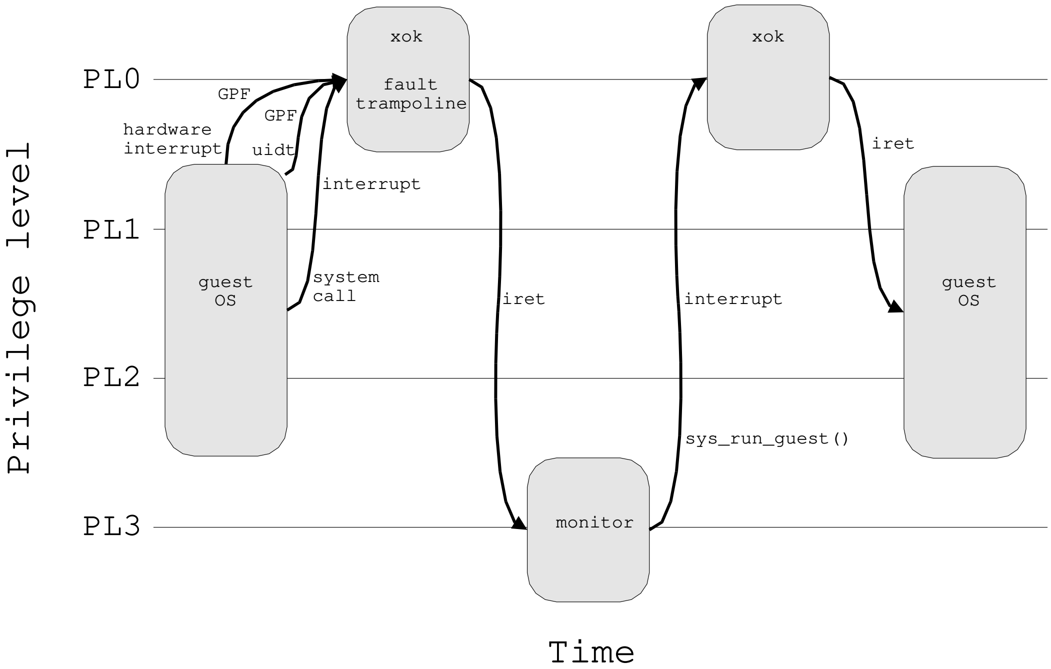

Figure B-3: Interrupt and exception handling from guest operating system.

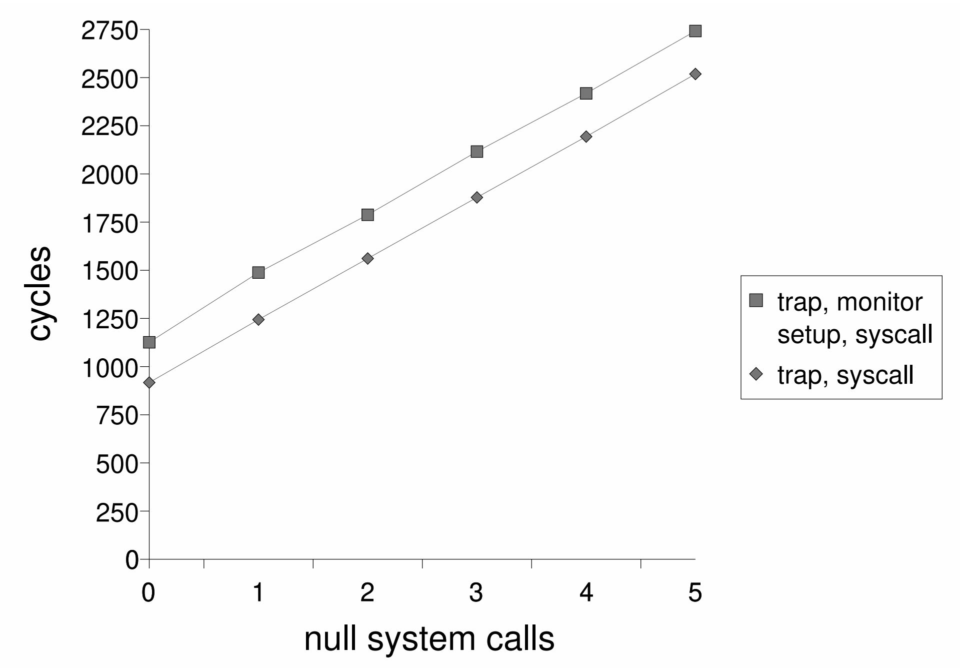

Figure B-4: Minimum cost of trapping from a guest operating system, emulating an instruction with xok system calls, and returning.

Bibliography

[1] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals.* Addison-Wesley, second edition, 1998.

[2] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *Operating Systems Review*, pages 143–156, December 1997.

[3] Robert R. Collins. Undocumented corner: The caveats of Pentium System Management Mode. *Dr. Dobb's Journal of Software Tools*, 22(5):109–??, May 1997.

[4] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schœnberg, and Jean Wolter. The performance of μ-kernel-based systems. *16th ACM Symposium on Operating Systems Principles*, October 1997.

[5] *Architecture and Programming Manual*, volume 3 of *Pentium Processor Family Developer's Manual.* Intel Corporation, Mt. Prospect, Illinois, January 1996.

[6] *Operating System Writer's Manual*, volume 3 of *Pentium Pro Family Developer's Manual.* Intel Corporation, Mt. Prospect, Illinois, January 1996.

[7] M. Frans Kaashoek, Dawson Engler, Gregory Ganger, Hector Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotii, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. *16th Symposium on Operating Systems Principles*, 1997.

[8] *PowerPC 601 RISC Microprocessor User's Manual*, chapter 2, pages 2–6. Motorola, Inc., Phoenix, Arizona, 1993.

[9] Kip Rugger. Why is x86 not virtualizable? *comp.arch*, July 1998. Message-ID: 6pfv7a$561@rugger.nodomain.ca.