Usermode Kernel : Running the Kernel in Userspace in VM Environments

by

Sharath George

M.Tech., Atal Bihari Vajpayee - Indian Institute of Information Technology and Management, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver)

June, 2008

Abstract

In many instances of virtual machine deployments today, virtual machine instances are created to support a single application. Traditional operating systems provide an extensive framework for protecting one process from another. In such deployments, this protection layer becomes an additional source of overhead as isolation between services is provided at an operating system level and each instance of an operating system supports only one service. This makes the operating system the equivalent of a process from the traditional operating system perspective. Isolation between these operating systems and indirectly the services they support, is ensured by the virtual machine monitor in these deployments. In these scenarios the process protection provided by the operating system becomes redundant and a source of additional overhead. We propose a new model for these scenarios with operating systems that bypass this redundant protection offered by the traditional operating systems. We prototyped such an operating system by executing parts of the operating system in the same protection ring as user applications. This gives processes more power and access to kernel memory bypassing the need to copy data from user to kernel and vice versa as is required when the traditional ring protection layer is enforced. This allows us to save the system call trap overhead and allows application programmers to directly call kernel functions exposing the rich kernel library. This does not compromise security on the other virtual machines running on the same physical machine, as they are protected by the VMM. We illustrate the design and implementation of such a system with the Xen hypervisor and the XenoLinux kernel.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge those who gave me invaluable help during the duration of this research. Without their support, I could never have accomplished this work. First of all, my sincere gratitude goes to my supervisors, Mike Feeley and Andrew Warfield, for their tremendous patience and support in providing me with extraordinary guidance and leading me to the right direction of research. I would also like to thank Geoffrey Lefebvre for his continuous stream of ideas that helped me immensly to solve the problems that I came across during the course of this thesis.

Chapter 1

Introduction

1.1 Motivation

Virtualization is enjoying an explosive growth in today's IT infrastructure. This is a tremendous shift in technology with many advantages like server consolidation, increased security, disaster recovery, and server migration. With the continuous increase of the number of cores in each system, it is now possible to run many virtual machines on a single physical machine. This is achieved by adding a thin software layer between the Operating System kernel and the hardware, called the Virtual Machine Monitor. The VMM manages the hardware resources, allowing the common resources to be shared among all the virtual machines running on the system. The resource management in itself consumes a part of the processing power and memory from the physical system which is absent in non-virtualized environments. In certain specific commercial deployments of virtualization, each virtual machine is configured to provide only one service. However each virtual machine runs a full operating system within itself. These operating systems are multi-tasking operating systems which support process isolation preventing one process from accessing other processes. Protection mechanisms include page level protections for operating system pages, utilizing separate address spaces for different processes, and code verification paths in system call handlers ensuring denial of access to memory that is not accessible to the process. These security features in the operating systems come with an additional cost. Some of these security features become redundant in virtual machine deployments where each operating system provides a single service. The main motivation of this thesis is to try to minimize these additional costs by bypassing these redundant security features in the operating system.

1.2 Introduction to Virtualization

Virtualization in the computing world refers to abstraction of computing resources. It hides the actual characteristics of the computing resources from the users of virtualization. Hardware virtualization makes one physical resource appear as many virtual resources. In the context of this thesis we broadly refer to the hardware virtualization wherein a software layer multiplexes a single physical computer to many consumers, each consuming one instance of the virtual computer invisible to the details of multiplexing the physical resource.

Traditionally operating systems run on bare hardware providing important functionality like memory management, process management etc to other applications running on the operating system. These operating systems expect to control and access an entire physical machine.

With virtualization, we introduce another layer between the operating dystem and the hardware which exposes an interface that is provided by the hardware to the operating system running above it. We refer to this layer as a Virtual Machine Monitor (VMM). The VMM directly controls the hardware resources, and multiplexes resources such as CPU, memory and devices to be shared amongst multiple operating systems running above the VMM. So the consumers of this kind of virtualization are the operating systems. These operating systems may or may not be aware of the existence of the VMM but for the most part, these operating systems function as if they were executing directly on the hardware.

1.2.1 Advantages of Virtualization

Virtual Machine Monitors are usually much smaller than today's popular macrokernel operating systems, hence they can be argued to be more secure than the operating systems they support. This is primarily because the interface exposed by the VMM is significantly narrower than the interface exposed by traditional operating systems.

Operating systems running on a VMM get a virtualized view of the hardware and are not aware of the other virtual machines running on the system. This isolation between the virtualization is managed by the VMM which manages the hardware resources and allocates them to the virtual machines above.

Today's physical machines are getting more and more powerful. This causes single server deployments on these machines to be inefficient as a lot of processing power remains idle. Virtualization allows multiple server

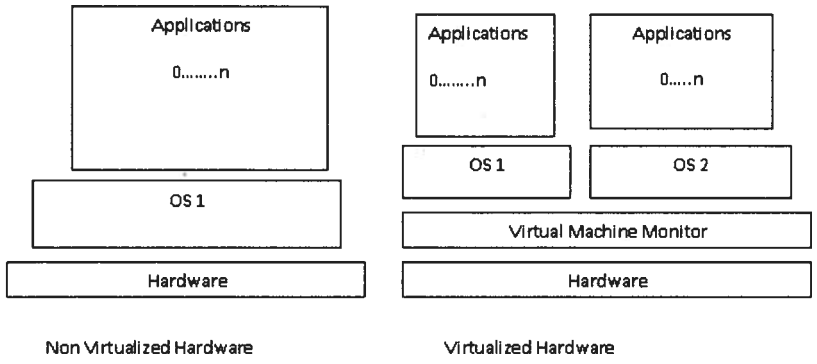

Figure 1.1: Virtualized vs Non virtualized systems

deployments on the same physical machine removing the need for multiple machines providing great logistic and power saving value.

1.2.2 Cost of Virtualization

Virtualization adds an extra layer to the software stack. The VMM has to manage all the hardware resources and allocate these resources appropriately to the operating systems running above it. Managing these resources in itself is done in software and uses a part of the total computing resources provided by the system. This is the additional overhead presented by the Virtual Machine Monitor. This cost ensures effective and safe isolation between the various Operating Systems which act as consumers for the virtualization layer.

1.3 Thesis

In virtualization deployments, a thin VMM executes directly on the underlying physical machine. Above this layer, we deploy one or more Virtual Machines. Each of these virtual machines run an entire operating system in them. Mostly these operating systems are slight variants of existing operating systems used in the non virtualized world and were designed to share

an entire physical machine among various applications/processes running on them. By design these protect one process from interfering with another. To implement this design requirement, these operating systems protect themselves from modification by processes. This is effected using the various protection mechanisms that today's processors provide to achieve this. These operating systems allow very controlled interaction between the processes it supports with the operating system kernel itself as well as with other processes. This control is necessary for process isolation when the OS supports more than one critical process which needs isolation from the other processes running on the System.

In many commercial deployments of virtual machines, each virtual machine is made to support only one service. This is mostly because the process isolation supports provided by today's operating systems do not meet the security standards demanded by certain critical IT infrastructures. However, each of these virtual machines does provide support for process isolation which becomes a source for overhead in these deployments. Isolation is effected by the VMM itself by isolating one virtual machine from another. In this scenario, it is safe to remove the process isolation mechanisms that are provided by the operating systems executing in the virtual machines, as these run only one critical process.

In this thesis we propose to remove the operating system level protection provided by the Linux operating system to run on the Xen hypervisor. We specifically target removing the paging protection that the Linux kernel utilizes to prevent processes from accessing the operating system memory directly. We also modify the system call path exposed by the operating system to bypass a protection ring transition to make use of this reduced protection. This allows unmodified processes to derive performance benefit offered by the lower level of protection. Applications could achieve still greater performance benefit by calling directly into internal kernel functions which are now accessible by removal of the paging protection. We propose that this customized kernel could be deployed in specific deployments where each virtual machine supports only one critical process/service. The operating system interface is still maintained as current applications expect the OS to offer simplified interfaces to its devices and memory. However we expose additional interfaces when removing the operating system level protection that applications can choose to use for better performance.

1.4 Thesis Outline

The remainder of the thesis text is summarized as follows.

In Chapter 2 we discuss the previous research that had taken the direction that we move within this research.

In Chapter 3 we discuss the current technology that we are looking at modifying for our model. We discuss the protection mechanisms provided by the x86 hardware, the Xen Hypervisor and the Linux Kernel modified to run on the Xen Hypervisor.

In Chapter 4 we discuss the various changes we made to the existing system to implement our proposed architecture.

In Chapter 5 we discuss the interface and library we developed to make it easy for applications to be developed utilizing the kernel functions directly enhancing the interface provided by the operating system.

In Chapter 6 we analyze the performance benefits that we have seen using this model.

In Chapter 7 we conclude the thesis and identify further directions of research and development that his model allows.

Chapter 2

Related Work

There are many instances of past research that try to allow user programs to run in kernel space. The most important difference between these and our work is the objective. Most of the previous research deals with trying to make the user programs run in the kernel safely without crashing the kernel. We assume that the kernel is expendable as the protection is already provided by the hypervisor and the kernel exists mainly to provide a service interface to the user process.

2.1 Kernel Mode Linux

Kernel Mode Linux proposed by Toshiyuki Maeda. [5], is a modified Linux kernel which allows user programs to be started as kernel threads. They modify the kernel to start user processes with segment registers pointing to the kernel segment descriptors. This makes the user processes run in ring 0. The x86 architecture allows one stack for each protection ring, and when there is a ring change the stack is automatically switched by the hardware. This modified architecture for KML implies that such a process has a single stack in the user space and no kernel stack. Since the user programs run in ring 0 they are able to call the kernel service routines for the system calls, bypassing the system call context switch boundary.

The main issue faced in this model is what the authors describe as the stack starvation problem. User memory is not guaranteed by the kernel to be non pagable. This implies that as the user stack expands it could expand into a virtual address range which is currently paged out to disk by the kernel. This would cause a page fault, on trying to access this memory. Normally, a page fault causes a trap into kernel mode and the stack would switch to the kernel mode stack which is guaranteed to be in memory. In the absence of a kernel mode stack, and having the kernel use the same user mode stack causes yet another fault as the page is still not in memory making it impossible for the page fault handler to run. This would cause a double fault, then a triple fault and so on. The authors handle this issue by

making changes to the double fault handler and associating a different stack context for the double fault handler which handles this condition specifically.

KML shares similar motivations as our thesis in the intention to reduce OS protection overhead for trusted processes though targeted at different scenarios. Our method is different from the KML framework in many ways. The primary difference between their approach and ours is that, in KML the user processes run with a higher privilege level (CPL of 0). In our approach we run parts of the kernel with a privilege level of user programs (CPL of 3). In the model proposed they propose maintaining security of kernel from exploitation by user processes by using Typed Assembly Language. We assume that the kernel is expendable as we assume the security required protecting the services on the physical machine is already provided by the Virtual Machine Monitor running beneath the kernel. The performance difference between the two approaches is hard to compare, as their patch for the kernel is not available for the XenoLinux kernel, while our approach is customized for the XenoLinux kernel.

2.2 Exokernel Project

The Exokernel architecture proposed by Engler et al. [4] [3], argues that in standard operating systems only the kernel can manage system resources, and that there is a performance benefit to be derived if some control over these resources is given to applications. The proposed architecture features an Exokernel which protects the resources (such as disk blocks and memory pages) of one process from another but allows each process to manage its own resources. The architecture suggests that commonly used abstractions/functions like those provided by current operating systems be linked as libraries creating libOSes (See Figure 2.1). The authors have developed an exokernel (Xok) and a libOS (ExOS) for the x86 architecture which allows many applications for the Unix environment to work unmodified on Xok/ExOS. At the same time this architecture allows applications to be customized to manage their own resources and use only a subset of ExOS to perform better.

The system we propose is similar to this approach with the role of the exokernel being played by the Xen hypervisor which protects one domain from another. While the kernel-user interface is available to the application programmer, the application programmer can also directly access kernel functions which manage its own resources in way that is not otherwise possible using just the system call interface. This allows for resource cus-

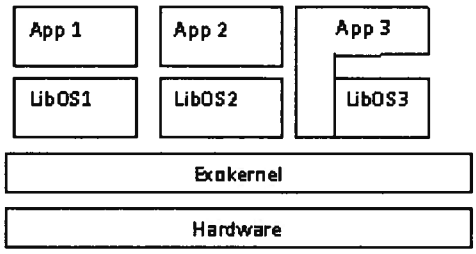

Figure 2.1: Exokernel architecture

tomization, as made possible in the exokernel architecture, for applications in Linux. This setup provides the additional advantage of running all Linux applications without modifications or recompilation. However our model does not currently address sharing higher level abstractions of data blocks such as file systems across domains in Xen.

2.3 Kernel Webservers - Khttpd, Tux

Web servers form a significant portion of server roles, and are very performance sensitive. This has caused a lot of performance optimizations in various kernels to make web servers perform better. Many web servers have been developed to run in kernel space for the added performance benefit that it offers. The khttpd [6] project is a web server that runs as a kernel module, while Tux [1] is integrated into the kernel more directly. Both these benefit from using kernel functions directly and using kernel memory. Thus they are not required to make repeated journeys across the User – Kernel boundary which is a significant cause of overhead. Both these web servers perform significantly better than web servers that exist entirely in user space. This goal of better performance by bypassing the kernel protection overhead is similar as in our thesis.

Moving code to kernel space requires that the code be as small and uncomplicated as possible as vulnerabilities in this code would compromise the entire system. This limitation causes these web servers to serve only static pages, as dynamic pages need more complicated interpreters and runtimes which significantly increase the required trusted computing base if pushed into the kernel. Such implementations pass on dynamic page requests back to user space traversing the protection ring boundary. This negates the ad-

vantage of having a kernel web server, when dynamic content forms a major chunk of the workload.

Our approach is to allow any application to run unmodified to save the cost associated with system calls, and at the same time allow applications to use kernel functions directly if that is beneficial from a performance standpoint. Our application that is able to send packets to any host using underlying kernel functions goes on to prove that modifying web servers to use the kernel functions directly does provide a major performance boost.

2.4 Infokernel

This project proposed by Andrea C. Arpaci-Dusseauet al. [2], allows user processes to change certain kernel mechanisms like file cache replacement and file and directory placement. This requires the kernel to expose values of certain key data structures to be visible to the user processes. For the published work, the authors have effected this exposure to user space by marking certain pages to be read by user space, rather than using system calls. The authors then establish that, with this information provided to applications, applications can modify their behavior to perform better by implementing different mechanisms in user space. In our work, we allow user programs to access all the kernel data structures, allowing such performance optimizations to the applications easily. The application that we wrote to inspect the process data structures in the kernel is an effort in the direction of this paper. The model we propose makes it easy to program applications that can easily peek into kernel data structures for useful information and also change hardcoded kernel policies for application benefit.

2.5 Application Specific Linux

This work by Lamia Youssef et al. [7] [8], is mainly motivated by the need to increase performance in high performance computing applications. The authors argue that in the batch execution model for running high performance computing applications, the cost of installing a customized Linux kernel to instantiate the application would not be very high. Thus they argue that performance enhancements can be attained by customizing the Linux kernel to work better for a specific application than following generic policies that benefit all the applications that the Linux kernel supports in a generic way. The authors envision a system that would automatically customize the Linux kernel for a particular application using software tools. However, such

a tool hasn't been developed yet and they currently use profiling tools such as KernProf to generate profile data when running the application for which the kernel has to be customized. This profile data is inspected to manually find code paths that can be optimized specific to the profile data. The authors have developed customized versions of Linux for specific applications that prove the benefit of this approach.

ASL shares the same goal as in our thesis to reduce overhead in the operating system in virtualized environments. However, our model is different from this approach as we do not need different versions of the kernel for each application to provide better performance. It is potentially easier to customize applications directly than profiling and reengineering the kernel based on its runtime data. This also saves the cost associated with re-instantiating different kernel versions for each application.

Chapter 3

Protection Mechanisms in Intel IA-32 Hardware, Xen and Linux

3.1 Overview

In this section, we will introduce the existing protection mechanisms that are provided by the Intel x86 32 bit processors. The hardware provides many options to enforce protection that is not currently used in Linux or a system virtualized with Xen and so only a subset of the protection mechanisms provided by the processor are outlined in this chapter. In this chapter we also discuss how Linux and Xen utilize these protection mechanisms provided by the hardware to ensure isolation between processes and virtual machines respectively.

3.2 Privilege Levels

The Intel x86 processors allow for 4 privilege levels (from 0 to 3) to be used by software with increasing privileges. These levels are generally referred to as protection rings as shown in Figure 3.1. The innermost ring has maximum privileges with decreasing privileges as we move to outward rings. The outermost ring (ring 3) is least privileged.

Usually operating systems, including the native Linux Kernel uses only two of these privilege levels, with the operating system kernel running in ring 0 and the user processes running in ring 3. In a virtualized system with Xen, three of these levels are used with the Xen hypervisor running in ring 0, the paravirtualized Linux kernel in ring 1 and the user processes in ring 3.

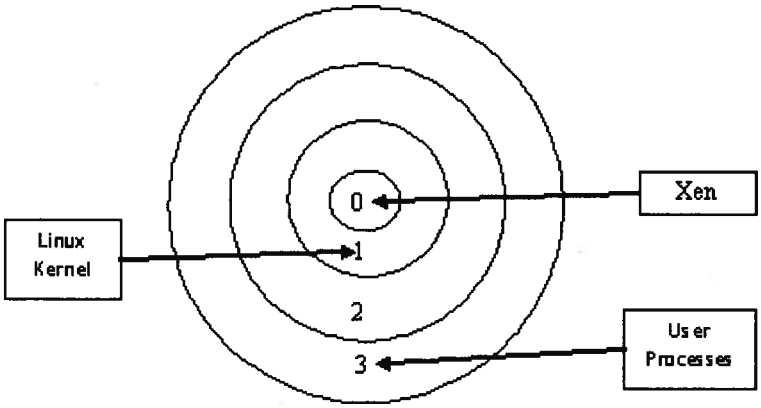

Figure 3.1: Protection rings in x86

3.3 Memory Management Protection

3.3.1 Memory Addressing

The x86 architecture defines three different kinds of memory addresses: Logical, Linear and Physical addresses.

A logical address is that which is used in a machine language instruction and is defined by a segment identifier and an offset.

A logical address is converted to a linear address by the segmentation unit (detailed in Section 3.3.2) by adding the offset to the segment base address, and is a number that varies from 0 to 0xffffffff. This allows linear addresses to map a total of 4GB of memory (in a 32 bit system).

A physical address is the actual hardware location of the memory addressed in the memory chip. The paging unit (detailed in Section 3.3.5) converts a virtual address to a physical address by looking up the physical address corresponding to a virtual address from the page tables.

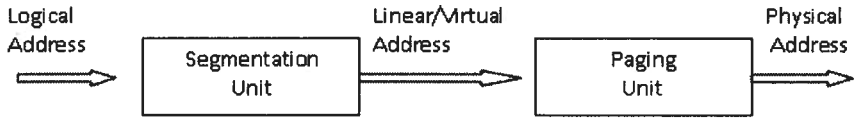

Figure 3.2: Memory unit architecture in x86

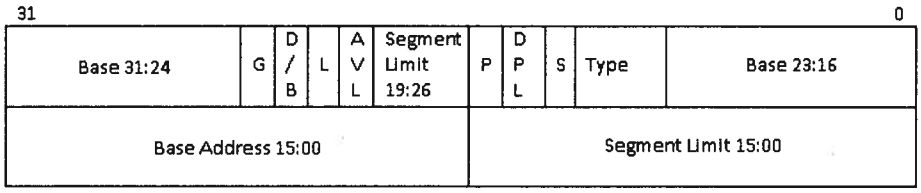

Figure 3.3: Segment descriptor format in x86

3.3.2 Segmentation

A segment in memory is defined in a data structure called a segment descriptor. A segment descriptor is 64 bits long and has the structure as shown in Figure 3.3.

The base address specifies the starting address of the segment and the segment limit specifies the size of the segment, thus fully defining the extent of virtual memory addressable in this segment.

The DPL bits specify the privilege level of the segment descriptor.

The Type bits specify the type of the segment that the descriptor defines. There are many types of segments; however of interest to us are only the types Code, Data and Task State Segment (TSS).

All the Segment Descriptors are stored either in a Global Descriptor Table (GDT) or a Local Descriptor Table. The processor provides specific segment registers whose function is to point to one of the descriptors in one of these tables. These segment registers contain Segment Selectors. A segment selector contains the index of a segment descriptor in the GDT/LDT and a Requestor Privilege Level(RPL).

The segment registers in x86 architecture are cs, ss, ds, es, fs and gs. Of these, three have specific purposes; the Code Segment register (cs) points to a segment containing the instructions of the current program, the Stack Segment register (ss) points to the segment containing the current program stack, and the Data Segment register (ds) points to the segment containing global and static data.

The RPL of the current cs register refers to the protection ring the processor is currently executing in, and is called the Current Privilege Level (CPL).

3.3.3 Segmentation in Linux

A native Linux kernel uses a flat segmentation structure and does not use segmentation to protect the kernel memory from the user processes. It defines 4 segments: a kernel code segment, a kernel data segment, a user code segment and a user data segment. All these 4 segments have a base address of 0x00000000 and a limit of 0xffffffff allowing all the segments to access the entire 4GB of virtual memory. Even though the user segments have a DPL of 3 while the kernel segments have a DPL of 0, a user process can address the kernel memory from its own segment. Linux uses paging for protecting its kernel from user processes.

3.3.4 Segmentation in Xen

Xen uses the segmentation protection offered by the processor and sets up the segment registers so that the top 64 MB of the memory is beyond the segment limits of the paravirtualized kernel and the user processes. This protects the hypervisor from being accessed/modified by the guest kernels. In this scenario, if the guest kernels try to access the Xen memory, the processor raises a General Protection Exception.

3.3.5 Paging

The paging unit of the processor converts a virtual address to the corresponding physical address. To achieve this the operating system has to set up page tables that map a virtual address to its corresponding physical address. The architecture requires the page tables to be set up in a two tier hierarchy. The first tier is a set of page directories. A page directory data structure contains pointers to the page table data structures. Entries in the page table data structure contain the base addresses of pages in physical memory.

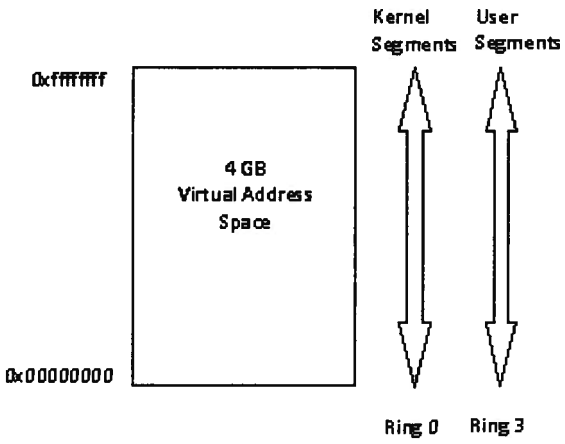

Figure 3.4: Native virtual address range in x86

The most significant 10 bits of the virtual address is an offset from the current Page Directory giving the address of the Page Table that maps the address. The next 10 bits of the address offsets in that particular page table to give the Page Table entry that stores the base address of the physical page which contains the memory location. The last 12 bits of the virtual address is the offset from the base address of the page to physical address corresponding to the virtual address.

3.3.6 Paging Protection in Linux

The native Linux kernel uses the protection offered by the Paging unit to protect itself from being accessed or modified by user processes. Both the Page Directory and the Page Table entries have a User/Supervisor Flag which allow for their access control based on the Current Privilege Level of the processor. However this is a 1 bit protection, and if for an entry this bit is set as a "User Page" then all the rings can access the entry. Otherwise the entry can be accessed only when the CPL of the processor is 0, 1 or 2 and cannot be accessed if the CPL of the processor is 3. This is sufficient protection for Linux because it uses only 2 rings (i.e. 0 and 3).

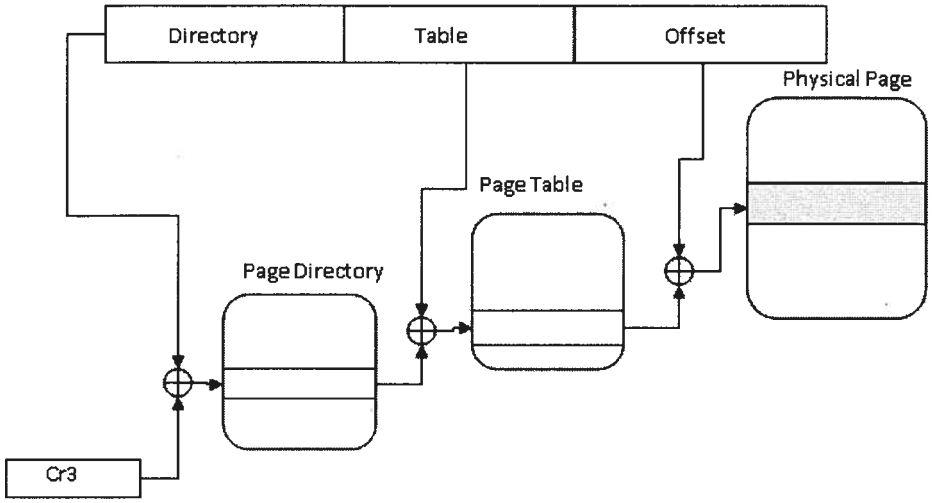

Figure 3.5: Paging in x86

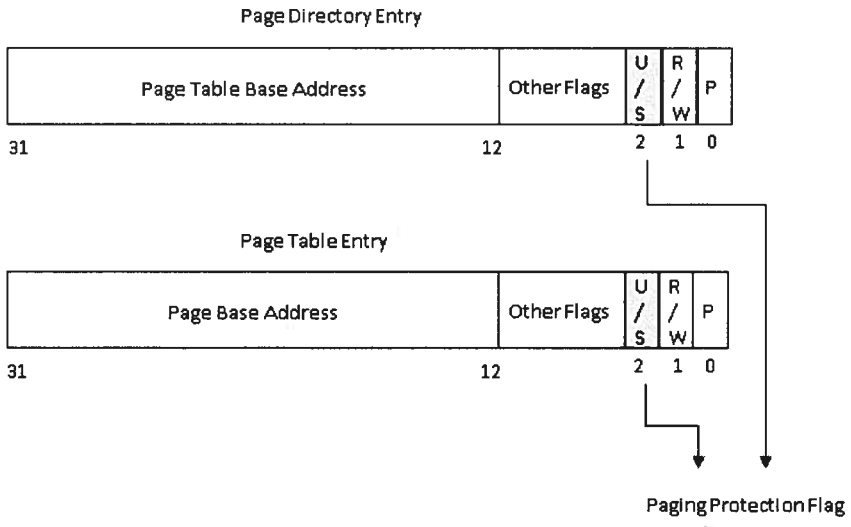

Figure 3.6: Page table entry data structures

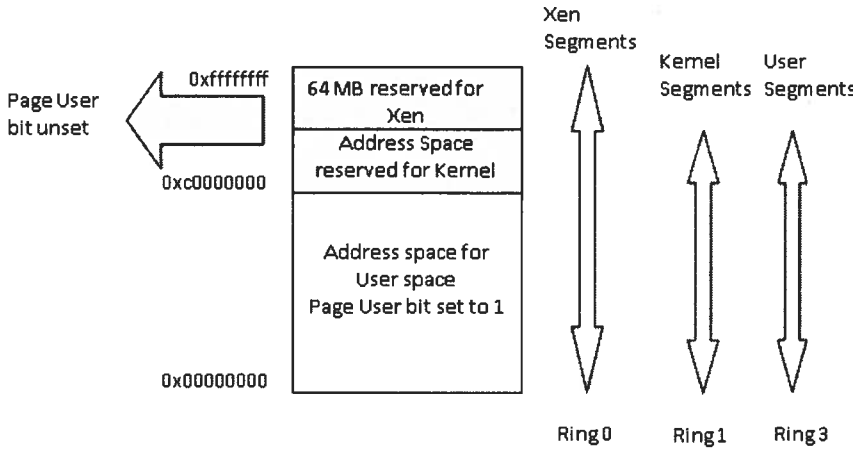

Figure 3.7: Interrupts and Exceptions

3.3.7 Paging Protection in Xen

Xen uses segmentation to protect itself from the guest kernels as discussed in the segmentation section of this chapter. The paravirtualized Linux kernel protects itself from the user processes in the same manner as a native Linux kernel: using the protection offered by the paging mechanism. So in a Xen system, all the Page Directory/Table entries associated with the Xen and Linux kernel memory are not marked as page user while the rest of the memory is marked page user.

3.4 Interrupts and Exceptions

An Interrupt is an event that alters the sequence of executions of the processor. An interrupt might be synchronous (when the interrupt is raised due to the execution of an instruction by the processor) or asynchronous (when the interrupt is raised by an external hardware event, e.g. a keystroke).

When an interrupt is raised, it has to be handled by the processor. To define how the processor handles interrupts, the hardware has to be preconfigured with an Interrupt Descriptor Table (IDT). This table associates each interrupt signal with an associated Interrupt Service Routine (ISR) which the processor starts executing when the corresponding interrupt is raised. Each entry in the IDT also specifies the privilege level the routine has to be

executed in and the CPU switches to the corresponding privilege level when it transfers execution to the ISR.

3.4.1 Hardware handling of an interrupt

The processor checks the DPL of the ISR with the CPL. If they are the same then the processor loads the EFLAGS, CS, and EIP registers onto the stack before transferring execution to the ISR.

If they are different then the processor switches stacks. The processor gets the address of the stack it has to switch to from the Task State Structure that is pointed to by the TR register. On this new stack the processor pushes the SS and the ESP registers which point to the top of the old stack. Then it pushes the EFLAGS, CS and EIP registers onto the new stack, and transfers execution to the ISR.

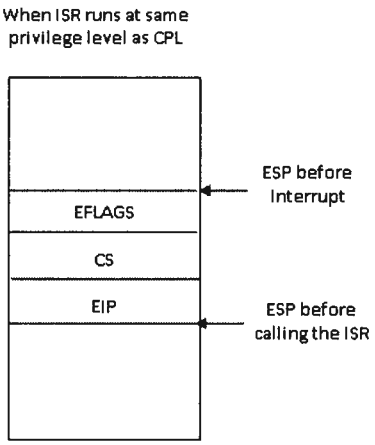

Figure 3.8: ISR handling

3.4.2 Interrupt Handling in Xen

Xen does not allow the guest kernel to set up the IDT for the processor, but allows the guest kernel to pass on an IDT it desires to Xen by means of a hypercall. Xen makes its own IDT on behalf of the guest kernel that the processor accesses. However, when it does this the stack that the hardware jumps to is at privilege level 0 and is a stack accessible only to Xen.

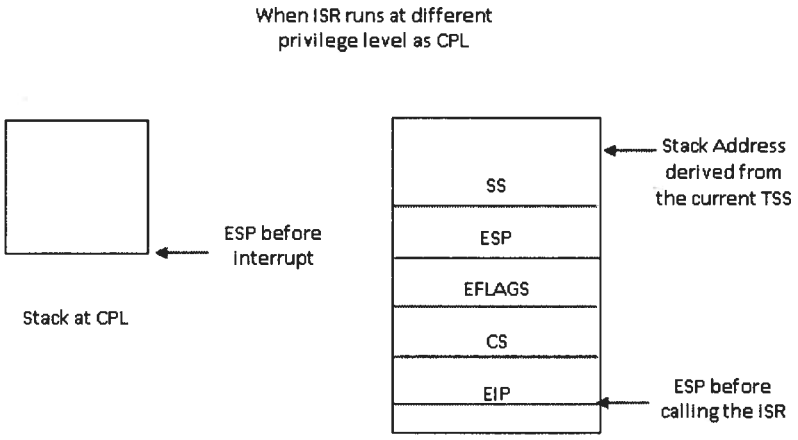

Figure 3.9: ISR handling

Xen emulates the hardware behavior to the OS by creating a bounce frame on the Linux Kernel stack just as the x86 hardware would do. However, Xen does not turn the control directly to the Interrupt Service Routine as set by the Linux kernel. Instead, the Xen hypervisor jumps to the "hypervisor_callback" routine defined in the XenoLinux kernel after creating a bounce frame on the kernel stack of the domain.

3.5 System Calls

System calls make up the interface that the kernel exposes to the application programmer. User programs are able to make a system call by triggering a software interrupt (int 0x80) which causes the hardware to jump to the interrupt handler for that interrupt.

Since this call is an inter privilege level call, the user stack cannot be used to pass the parameters to the system call. So the user program has to move the parameters into the registers before triggering the soft interrupt. Usually standard libraries like libc have wrapper functions around each system call, which move the parameters to the registers and cause the software interrupt so that these details are abstracted away from the common application developer.

Most processors support an alternate mechanism to make a system call into the kernel, using the "sysenter" and "sysexit" instructions. These in-

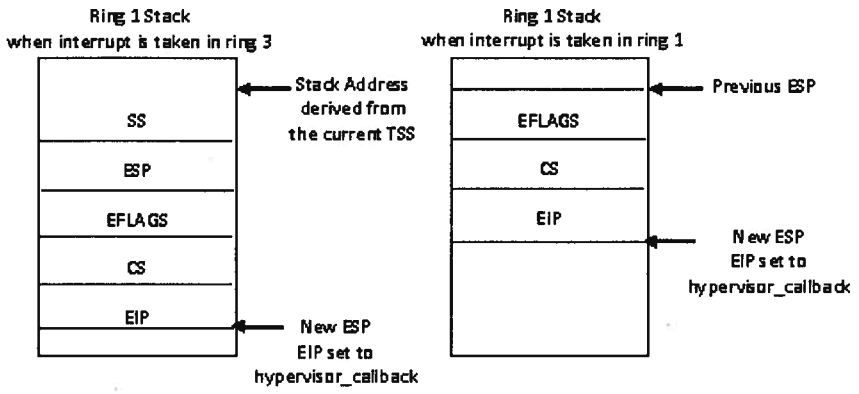

Figure 3.10: Xen interrupt handling

structions provide a faster mechanism for the application to make system calls and if available, it is faster if this is used. However some old processors do not have these instructions in the instruction set. To make the kernel backward compatible, as well as utilize the sysenter mechanism if available, the Linux kernel shares a virtual syscall page with the user processes i.e. the "vsyscall" page. This page contains the instructions to make the software trap into the kernel: the "int 0x80 " method if the processor does not support the newer sysenter mechanism, or the sysenter instruction. This allows a user process to make a call into this page and let the kernel decide the best way to make the trap, according to the available hardware. Normally a system call invocation from a common application program has a flow like in Figure 3.11.

3.5.1 Processes and Kernel Mode Stacks

Each user thread has a kernel mode stack associated with it. This is the stack used by the process when it is executing in the kernel mode. This stack switch happens when a hard/soft interrupt is taken during thread execution. As the kernel control paths don't contain many nested function calls, this stack is assigned only 1 page frame. This page frame also contains the thread_info data structure associated with that thread. This structure

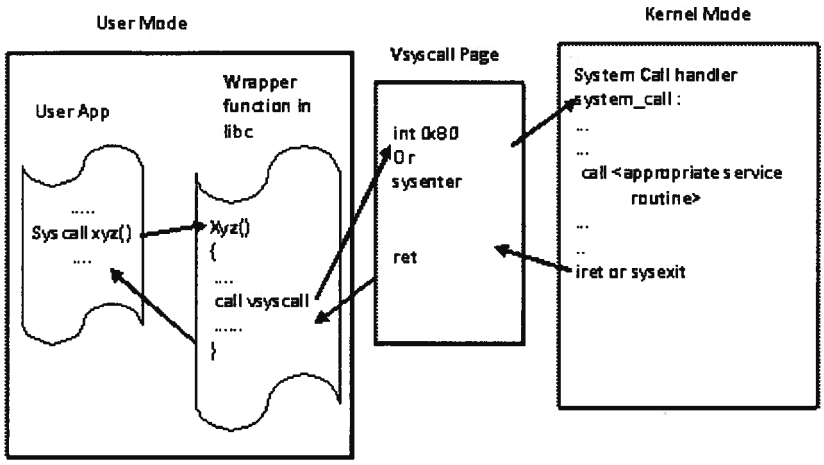

Figure 3.11: System call entry point to kernel

contains important properties of the thread including a pointer to the task structure. Many kernel functions use information in this structure. The address of this structure is calculated with kernel macros masking the value of the current ESP register, as there is a guarantee that the current stack pointer is in the same page as that of the thread_info structure. The memory layout of the kernel mode stack and the thread_info structure is as outlined in Figure 3.12.

3.5.2 Kernel Entry-Exit flow

All the points of entry into the XenoLinux kernel and exit from the XenoLinux kernel are handled by routines coded in assembly in entry-xen.S. The code flow from the entry points to the exit points are detailed in the flowchart in Figure 3.13. For simplicity the figure omits the entry points associated with exception handlers, return entry points for a thread fork, etc.

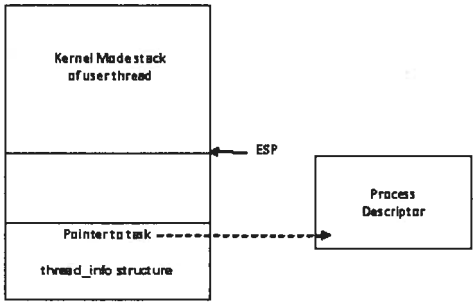

Figure 3.12: Process data structures in the Linux Kernel

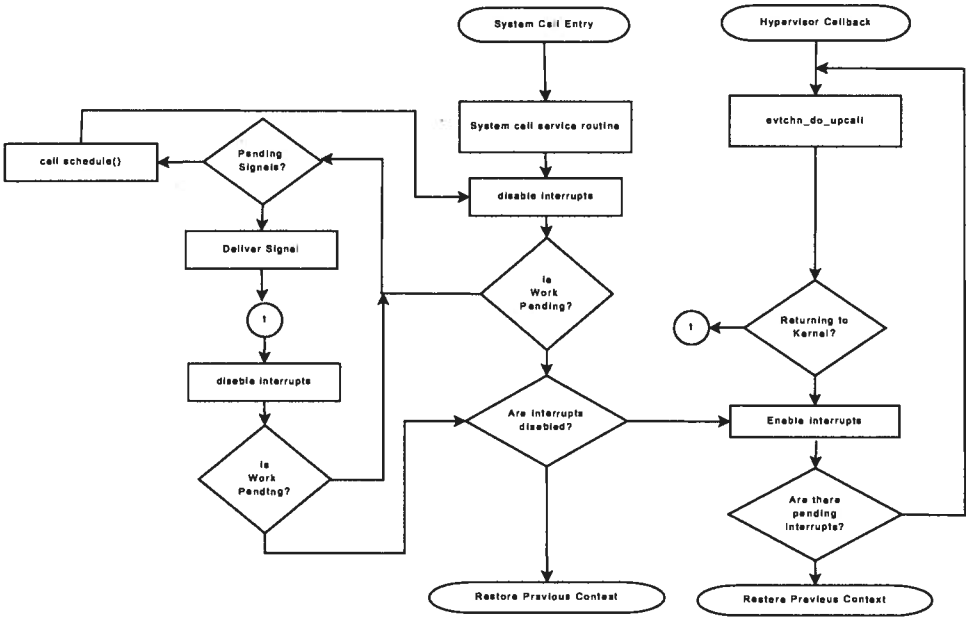

Figure 3.13: System call execution flow in Linux kernel

Chapter 4

Implementation of Usermode Kernel

4.1 Overview

The main design goal for our thesis is to remove the additional protection layers that are present in the operating system that protects the operating system from the applications. Implementation of this design involves some fundamental changes in the existing implementation of the Xen hypervisor as well as the XenoLinux kernel controlling the virtual machines.

The primary mechanism by which the Linux kernel protects itself from unauthorized access be user mode applications is by utilizing the paging protection facilities enforced by the hardware as discussed in Section 3.3.6. We discuss how in our implementation we bypass this protection feature. The Linux kernel is loaded into memory by the Xen Hypervisor at boot time of a virtual machine. For this reason, we have to modify the Xen Hypervisor as well as the Linux kernel to effect this change. This is discussed in detail in Section 4.2

Just as the hardware behaves differently based on which protection layer the processor is executing in; the Xen hypervisor is aware of the current privilege level the guest Virtual machine is executing in. Xen behaves differently for interrupt delivery to each guest based on the privilege level of executing code in the guest. Since we make fundamental changes to the way privilege levels are managed in the guest Linux kernel, we have to make certain modifications to the interrupt delivery mechanisms within the Xen Hypervisor. These changes are detailed in Section 4.3.

In our thesis, we propose a model in which the existing applications can be supported without any modification but at the same time still enjoy a performance benefit. Currently all applications interact with the Operating System through system calls. When an application makes a system call, we traverse the boundary separating the OS from the application. Since natively the protection mechanisms are in place, this traversal is much more

expensive than a normal function call as the execution privilege level of the code has to change. In our proposed model, we execute the Operating System code in the same privilege level as the application making the costly ring transition unnecessary. In order to support existing applications, rather than modifying the entire system call interface, we modify the common entry point to all system calls. We establish a different entry point to execute operating system code which is detailed in Section 4.4. We then modify the entry point to all system calls to point to this new entry we created allowing applications to benefit from our model without modification. This change is explained in Section 4.5

Implementing these high level changes in the current system introduces additional complexities that have to be handled within code. Some of these changes that were necessary are explained in Sections 4.6 and 4.7.

4.2 Page Table modification

As explained in Section 3.3, the XenoLinux kernel protects its memory from the user process address space with the "Page User" flag in the page table entries. Pages which have this flag set in the corresponding page table entry can be accessed by all rings, but pages with this flag cleared in their page table entry can be accessed only by rings 0, 1 and 2. Since the user processes run in ring 3, the kernel pages are inaccessible to the user programs. In our architecture we modify the page table entries of the kernel page to have the user page set. This allows the user processes to access the kernel pages. However the Xen memory is still protected from the Domain Kernels and the user programs by the x86 Segmentation protection mechanisms (Section 3.2).

The new modified memory structure in our implementation is depicted in Figure 4.1.

4.3 Xen Interrupt delivery modification

In our proposed architecture, the kernel code could be executed either in ring 3 or in ring 1. However, the interrupt delivery mechanism present in Xen behaves differently when an interrupt takes place when the CPU is running in ring 1 from that when it is running in ring 3.

Eventhough we may be running the kernel code from ring 3, we still use the kernel stack when we execute kernel code. This creates a problem, as the default behavior of Xen uses the kernel stack from the top for the interrupt

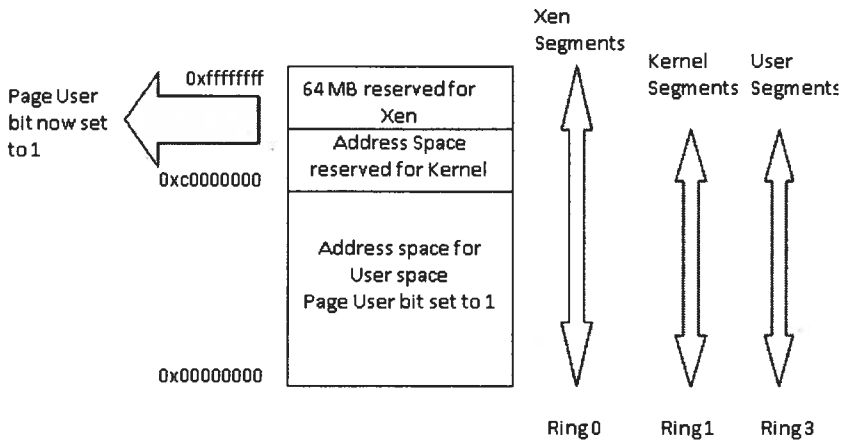

Figure 4.1: Modified memory structure for our model

handler, when an interrupt is taken in ring 3. We modify this behavior, to add an additional check in the Xen hypervisor to see if the current stack pointer is greater that 0xc0000000 (which implies a kernel stack) when an interrupt is taken in ring 3. If this condition is true, it implies that we are running kernel code in ring 3 and we should use the stack bottom for the interrupt handler rather than the stack top (See Figure 4.2).

4.4 Alternate System Call Path entry

To allow us to save the trap cost associated with a system call, we convert all the system calls into direct function calls into the kernel instead of causing a software interrupt. However, we still leave the existing system call handler in the kernel allowing system call invocation through a software interrupt as well. We add a new routine which almost mirrors the system call handler.

When a software interrupt occurs the hardware automatically switches the stack before it calls the interrupt handler. So the default system call handler runs in the kernel stack . When we make a function call into the kernel this stack switching does not happen automatically. We have to manually switch the stack in our alternate path for direct system calls. It is important that the kernel functions run in the kernel stack and not use the user stack, as the kernel stack is aligned to a pointer to the task

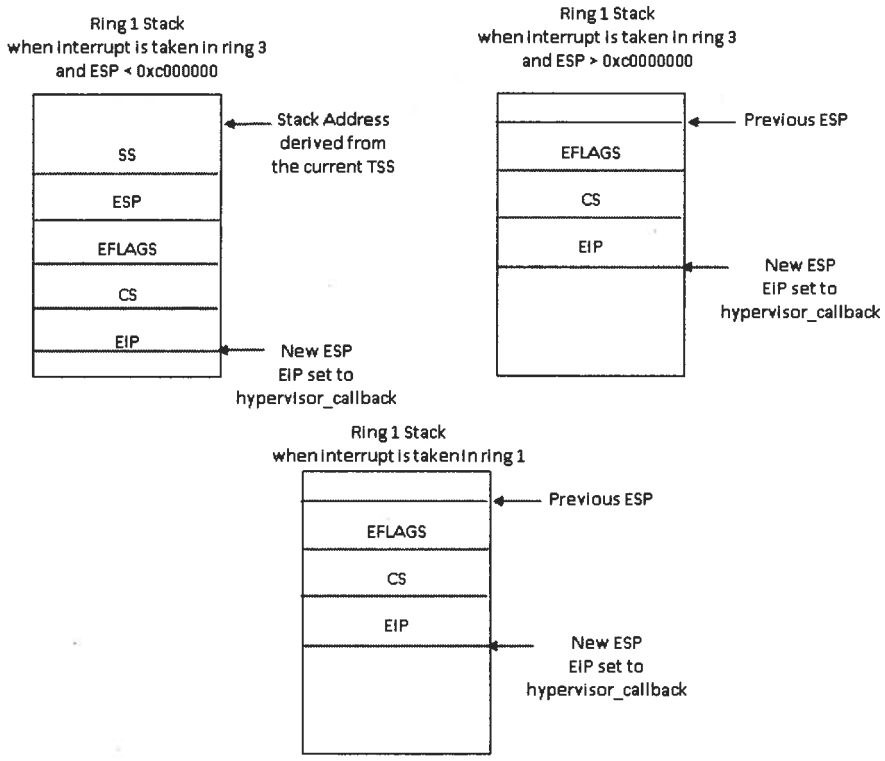

Figure 4.2: Xen interrupt delivery modification for our model

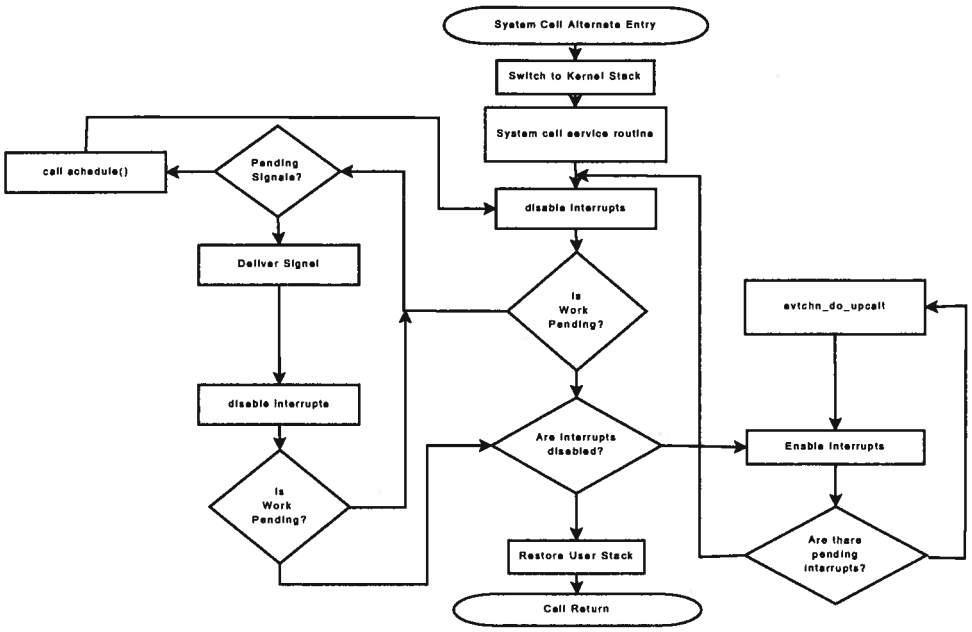

Figure 4.3: Code flow from alternate system call entry path

structure. Many kernel functions and macros find the address of this pointer by masking the current ESP. The code flow of our alternate system call entry path is outlined by the flowchart in Figure 4.3.

4.5 Vsyscall page modification

The vsyscall page is the single point of entry for all the system calls from the user processes. Even though it is possible to execute the "int 0x80" instruction directly, common libc implementations usually make a call into the vsyscall page. In our kernel, we modify the vsyscall page to make a direct call into the alternate system call path entry that we introduced in the kernel. This automatically bypasses the trap calls made for all system calls from library implementations that use the vsyscall page (See Figure 4.4).

4.6 Ring 1 to Ring3 and Ring3 to Ring1

Once we allow the processes to run parts of the kernel in ring 3, we have to handle exit code paths from the kernel carefully. In the scenario that we

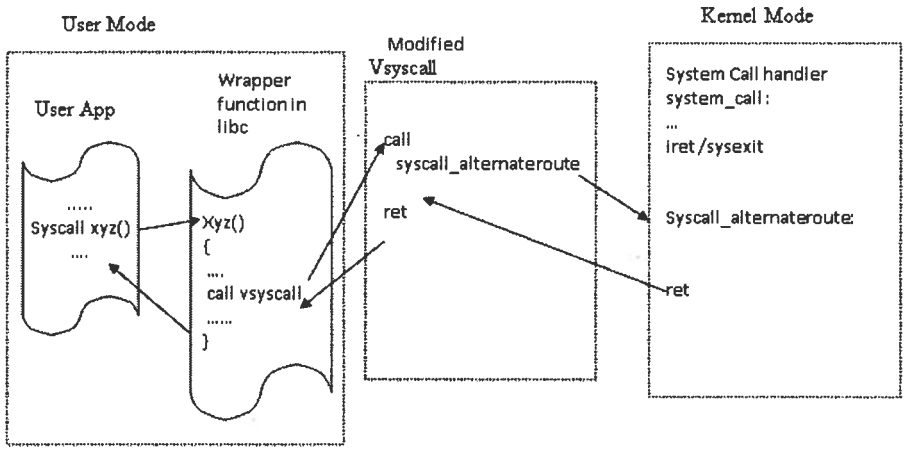

Figure 4.4: Modified system call entry point in Vsyscall page

have created it is possible that certain threads are running in the kernel in ring 3 and certain other kernel threads are running in the kernel in ring 1.

However it is possible that a timer interrupt would bring about a context switch from the thread running kernel code in ring 1 to a thread running kernel code in ring 3 or vice versa. When we switch threads we do not change the rings for the new thread based on the ring it was running in when it was context switched out.

This brings in the possibility that when we exit a kernel path back to user space, we might be in a ring that we weren't expecting to be in. We had to add additional checks around the exit paths from the kernel based on the current privilege level of the thread. If the thread is already in ring 3, we have to use a "ret" instruction and use an "iret" otherwise.

4.7 Page Fault Handler modification

The page fault handler in the Linux kernel behaves differently when there is a page fault when running in ring 1, from when there is a page fault in ring 3. This is because page faults in the kernel can occur only at specific points and each of these points has an exception handler associated with it.

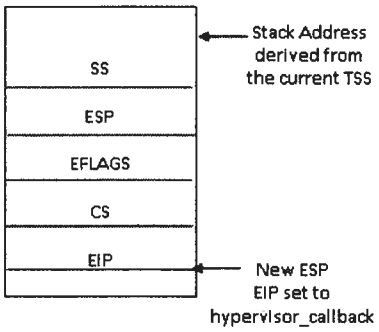

Figure 4.5: Stack on exiting from kernel thread

In the native kernel the page fault handler determines the ring at which the page fault occurred to determine if it was executing in the kernel or a user process. In our architecture, it is possible to be executing kernel code in ring 3, so a check on the ring level is not sufficient. This check was modified to check the EIP at page fault. An EIP greater than 0xc0000000 indicates kernel code and a trap in this region would be handled different from that in user space.

Chapter 5

Utilizing the Kernel Library

5.1 Overview

In this section, we describe how we implemented the framework to make calls to the kernel functions from user space, exposing the kernel library to applications. As we mentioned in the previous section, we marked the page user bit in all the page table entries corresponding to the kernel memory, so that the hardware does not deliver a page protection fault when the kernel memory is accessed from userspace. However, we still need to switch stacks to align ESP with the pointer to the task structure as discussed in the previous section.

5.2 Middleware Library

We need explicit code to switch stacks whenever a user programs wants to make a call to the kernel function as we do not want to trap into the kernel to fall into a higher privilege ring. We achieve this by writing a small wrapper function that the user program has to call in order to call any kernel function. As a proof of concept, we have currently implemented a function with the following prototype
*void * kernelcall(<kernel function name symbol>, <no of arguments> ,... <variable no of arguments>)*
So for example a user application needing to call printk with a test message can make the call with the following call
kernelcall(printk,1,"Hello World");
This function internally implements the following.

1. Switch to the kernel stack for the corresponding thread, aligning the ESP with the pointer to the task structure.

2. Saves registers in the kernel stack and restores them on exit, because many functions require that the stack be setup in this way.

3. Pushes the parameters to the function from the user stack to the current kernel stack.
4. Makes a call to the specified kernel function.
5. Copies the return value into register EAX.
6. Switches back to the user stack.

5.3 Symbol Table Import

To simplify the exporting of the symbol table and linking this middleware library, we have currently used the symbol table file generated on building the kernel (System.map) to generate a header file which converts each symbol name to their corresponding addresses. Our middleware library uses these converted addresses to make direct calls into these addresses.

5.4 Proof of Concept – Applications using Kernel functions

To demonstrate the ease of this interface to make customized applications using the kernel library, 2 applications were developed.

5.4.1 Application to monitor process state and kernel stack

From the symbol table of the kernel build, we not only get the addresses of the kernel functions but also of the key data structures. Since we can read the kernel memory, we can access the kernel data structure as if we were accessing global variables. This allowed the creation of an application that iterates over the process list and peeks into the task structure of the processes. This allowed us to view the kernel stack, and other interesting process state information. This proves the ease of which applications can be developed to view kernel information, and allows application customization like that proposed in the Infokernel paper [2].

5.4.2 Application to send packets to a different host/network

To gauge the performance benefit offered by using the kernel functions directly from applications we have utilized our kernel library that we developed

to develop an application that can send arbitrary data to any host on the network using UDP.

This customized application uses internal kernel functions to copy memory directly from the user memory and write headers directly into allocated socket buffers. These buffers are then directly passed on to the network hardware driver which sends this buffer onto the wire. The performance of this application is compared with another that utilizes the socket programming interfaces that the kernel provides to applications in a native system. The code path length from the function which sends the data to the initiation of the network device driver is significantly lower when we use the kernel functions directly. The benchmark results of this application are analyzed in Section 6.4. This goes on to further prove that current applications can be easily modified to use underlying kernel functions rather than using the native API exposed by the kernel without much effort to gain performance benefit.

Chapter 6

Performance Benchmarks of System Call Optimization

6.1 Overview

In this section we present the performance improvement obtained by bypassing the system call trap, and making a direct call into the kernel system call service routine. We present both microbenchmarks and application level benchmarks showing the performance benefit gained by our approach.

6.2 Lmbench Microbenchmarks

We used the lmbench suite to microbenchmark the performance benefit that is provided by skipping the system call trap, and making a call to the system call entry point. We conducted multiple runs and averaged the data to normalize the results. The results are shown in Figure 6.1. A more detailed table showing differences for various system calls is shown in Table 6.1.

From the results it is evident that we save almost 0.23 microseconds on each system call that is made by the application. For applications that make a lot of system calls, we stand to gain substantial performance benefit.

6.3 Application Benchmarks

6.3.1 Find benchmark

For this test, we searched for a file in the Linux kernel source directory, using find. This test was measured with the time command. The results are recorded in Table 6.2.

6.3.2 Iperf Benchmark

Iperf is a benchmark tool to measure TCP and UDP bandwidth performance. It is a simple tool and has a server which accepts data and a client

System Call	**Native (s)**	**Modified (s)**	**Difference (s)**	**% Benefit**
Simple syscall	0.3092	0.0744	0.2348	75.9
Simple read	0.4694	0.2519	0.2175	46.3
Simple write	0.4316	0.1985	0.2331	54.0
Simple stat	1.2787	1.0288	0.2499	19.5
Simple fstat	0.4794	0.2688	0.2106	43.9
Simple open/close	2.3187	1.8732	0.4455	19.2
Select on 10 fd's	0.9203	0.6874	0.2329	25.3
Select on 100 fd's	3.4407	3.1656	0.2751	8.0
Select on 250 fd's	7.5625	7.3122	0.2503	3.3
Select on 500 fd's	14.4567	14.2481	0.2086	1.4
Select on 10 tcp fd's	0.9606	0.7228	0.2378	24.8
Select on 100 tcp fd's	5.4229	5.1931	0.2298	4.2
Select on 250 tcp fd's	12.9724	12.8167	0.1557	1.2
Select on 500 tcp fd's	25.6526	24.9628	0.6898	2.7
Signal handler installation	0.7183	0.4937	0.2246	31.3
Signal handler overhead	1.6948	1.7453	-0.0505	-3.0
Protection fault	0.8939	0.9801	-0.0862	-9.6
Pipe latency	10.1135	9.3661	0.7474	7.4
AF_UNIX sock stream latency	19.0139	18.0882	0.9257	4.9
Process fork+exit	274.7	272.05	2.65	1.0

Table 6.1: System call performance benchmarks

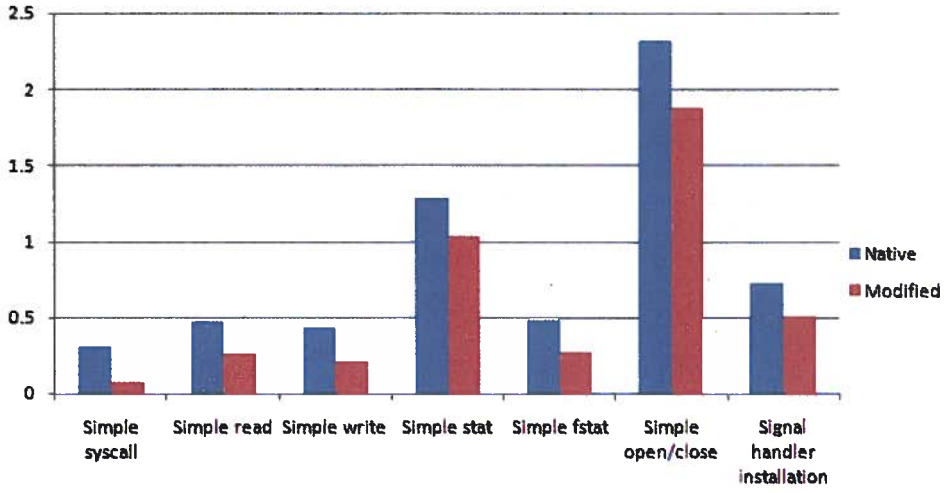

Figure 6.1: System call performance benchmark

Benchmark	Time (ms)	Modified (ms)	Difference (ms)	% Benefit
find	340	327	13	3.8

Table 6.2: find application benchmark

Iperf benchmark	Native (Gbits/s)	Modified (Gbits/s)	Difference (Gbits/s)	% Benefit
TCP Throughput	3.55	3.73	0.18	4.8

Table 6.3: iperf application benchmark

Kernel Library benchmark	Native Library Application (Delay ns)	Kernel Library Application (Delay ns)	Difference (ns)	% Benefit
Delay for sending 100, 5KB packets	630	350	280	44

Table 6.4: Kernel library application benchmark

which sends data. For this experiment we ran both the server and the client on a Xen domain with the system call trap optimizations, with the data transfer taking place over the loopback interface. The results of this experiment are shown in Table 6.3.

6.4 Kernel Library Benchmark

To illustrate the benefit offered by utilizing kernel functions directly, we created an application utilizing the kernel library functions that sends raw packets onto the network. We compare this against a similar user mode program which sends the same data over the network but utilizing the interface provided by the common libraries included with Linux.

For the test setup, we benchmark a process that sends 100 UDP packets each of 5KB to the network. We use socket functions provided by the kernel for the native application. We compare the performance of this application against the custom application which uses the kernel library that we expose using our model. In this applications we directly modify kernel memory and construct the network headers ourselves bypassing the socket functions and customizing the path for performance. This packet so constructed is passed onto the network card directly. The results are recorded in Table 6.4.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

We have proposed a new architecture model providing performance benefit in virtual machine deployments where each virtual machine runs only a single service providing application. We have prototyped a version of Xen and the XenoLinux kernel that implements this model. This model allows unmodified versions of applications to run on this prototype kernel while gaining performance benefits. The benchmarks on this model prove that there is sufficient advantage on further investigating this approach towards commercial deployments.

Our model and proof of concept applications also prove that customized applications developed/modified for this infrastructure can gain tremendous performance benefit by utilizing the kernel functions/data structures directly.

This infrastructure also allows greater power flexibility in debugging applications by allowing access to kernel data structures. This also allows easy control over kernel policies which are otherwise hard to change in native deployments.

7.2 Future Work

Stress tests on certain applications sometimes cause the kernel to page fault at certain applications. This is an unresolved bug in the current implementation of this model which could not be fixed due to time constraints. Further evaluation of this bug is necessary for deployments using this model.

To further prove the performance benefit of this model, popular webservers could be modified to use kernel functions directly for performance critical paths and tested on this architecture to study the deployment benefits of this model.

The current implementation of our proposed model handles only single processor systems. This could be enhanced to support multiple processors with relative ease as the model does not need architectural revisions to handle multiple processors. However applications utilizing the kernel libraries need to be coded more carefully to handle locking mechanisms for multiple processors to prevent deadlocks.

The current architecture model is specific for the 32 bit Linux kernel. In 64 bit systems the segmentation and paging model supported by the hardware and by Xen is significantly different and a different model needs to be proposed for this architecture.

Bibliography

[1] Tux web server. *http://www.redhat.com/docs/manuals/tux/.*

[2] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, N.C. Burnett, T.E. Denehy, T.J. Engle, H.S. Gunawi, J.A. Nugent, and F.I. Popovici. Transforming policies into mechanisms with infokernel. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105, 2003.

[3] D.R. Engler. *The Exokernel Operating System Architecture.* PhD thesis, Massachusetts Institute of Technology, 1998.

[4] DR Engler, MF Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, 1995.

[5] T. Maeda. *Safe Execution of User programs in Kernel mode using Typed Assembly Language.* PhD thesis, The University of Tokyo, 2002.

[6] A. van de Ven. kHTTPd: Linux HTTP accelerator, 1999.

[7] L. Youseff, R. Wolski, and C. Krintz. Application Specific Linux.

[8] L. Youseff, R. Wolski, and C. Krintz. Linux kernel specialization for scientific application performance. Technical report, Technical Report UCSB Technical Report 2005-29, Univ. of California, Santa Barbara, Nov 2005.