Device Driver isolation using virtual machines

Sushrut Shirole

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Dr. Godmar Back, Chair
Dr. Keith Bisset
Dr. Kirk Cameron

Dec 12, 2013
Blacksburg, Virginia

Keywords: Device driver, Virtual machines, domain,

Device driver isolation using virtual machines.

Sushrut Shirole

(ABSTRACT)

In majority of today's operating system architectures, kernel is tightly coupled with the device drivers. In such cases, failure in critical components can lead to system failure. A malicious or faulty device driver can make the system unstable, thereby reducing the robustness. Unlike user processes, a simple restart of the device driver is not possible. In such circumstances a complete system reboot is necessary for complete recovery. In a virtualized environment or infrastructure where multiple operating systems execute over a common hardware platform, cannot afford to reboot the entire hardware due to a malfunctioning of a third party device driver.

The solution we implement exploits the virtualization to isolate the device drivers from the kernel. In this implementation, a device driver serves the user process by running in a separate virtual machine and hence is isolated from kernel. This proposed solution increases the robustness of the system, benefiting all critical systems.

To support the proposed solution, we implemented a prototype based on linux kernel and Xen hypervisor. In this prototype we create an independent device driver domain for Block device driver. Our prototype demonstrate that a block device driver can be run in a separate domain.

We isolate device drivers from the kernel with two different approaches and compare both the results. In first approach, we implement the device driver isolation using an interrupt-based inter-domain signaling facility provided by xen hypervisor called event channels. In second approach, we implement the solution, using spinning threads. In second approach user application puts the request in request queue asynchronously and independent driver domain spins over the request queue to check if a new request is available. Event channel is an interrupt-based inter-domain mechanism and it involves immediate context switch, however, spinning doesn't involve immediate context switch and hence can give different results than event channel mechanism.

Acknowledgments

Acknowledgments goes here

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A system is judged by the quality of the services it offers and its ability to function reliably. Even though reliability of operating systems has been studied for several decades, it remains a major concern today. The characteristics of operating systems which make them unstable are size and complexity.

Software reliability study shows that 6 to 16 number of errors/1000 executable lines can be found within a module[6, 36]. Linux kernel has over 15 million lines of code. If we assume minimum estimate, Linux kernel has around 90,000 bugs. Researchers have shown that the error rate for device drivers is higher than the error rate for the rest of the kernel[11]. Considering the fact that the operating system predominantly consists of device drivers, the faults in device drivers make the operating system unreliable[11].

1.1 Problem Statement

In order to make a system reliable, it is essential that a device driver code does not contain any bug. However, finding all the bugs and fixing them is difficult since bug fixes introduces new lines of code resulting in new bugs.

In modern operating systems, memory protection is a way to control memory access rights. The memory protection prevents a process from accessing memory that has not been allocated to it. The memory protection prevents a bug within a process from affecting other processes, or the operating system[14, 32]. Monolithic kernel component do not have the same level of isolation the user level applications have. Unlike user applications, monolithic kernel has hundreds of procedures linked together. As a result, any portion of the kernel can access and potentially overwrite any kernel data structure used by an unrelated component. Such a non-existent isolation between kernel and device driver causes a bug in device drivers to corrupt the memory of the other kernel components. This memory corruption might lead to system crash. Hence the underlying cause of unreliability in the operating system is the tight coupling between device driver and Linux kernel.

In the past, solutions to increase the reliability of a system running monolithic kernel, based on virtualization has been proposed by LeVasseur et. al. [22], Tanenbaum et. al.[36], Nooks[35], Soltesz et. al. [33]. These solutions improve the reliability of the system by executing device drivers in an isolated environment from the kernel. The use of virtual machines has a well-deserved reputation for extremely good fault isolation. Since none of the virtual machines are aware of the other virtual machines, malfunctioning of one virtual machine cannot spread to the others. Xen hypervisor also provides a similar platform to isolate device driver from the monolithic kernel. The platform is called driver domain [1].

Despite the advances in virtualization technology, the overhead of I/O virtualization significantly affects the performance of applications[4, 34, 25]. In this thesis, we propose and evaluate an optimization for improving the performance of the driver domain under the Xen.

1.2 Proposed Solution

In a virtualized environment, all virtual machines run as separate user processes in different address spaces. Thus, to exploit the memory protection capability between virtual machines, xen runs a device driver with a minimalistic kernel in a separate domain, and runs user applications and a kernel in the guest domain. As a result, a device driver is isolated from the Linux kernel, making it impossible for the device driver to corrupt any kernel data structure in the virtual machine running user applications.

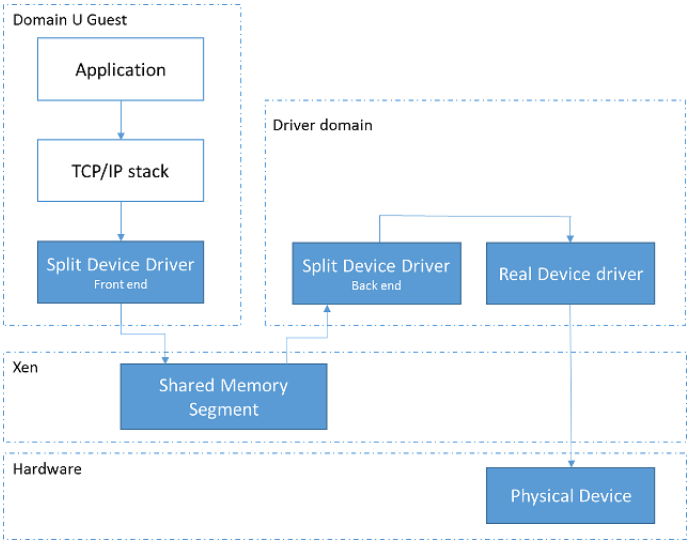

Figure 1.1: Split device driver model

As shown in Figure 1.1, Xen follows a split device driver model. Xen has a front end driver in the guest operating system and a back end driver in the driver domain. The front end and the back end driver transfer data between domains over a channel that is provided by the Xen virtual machine monitor. Within the driver domain, back end driver is used to demultiplex incoming data to the device and to multiplex outgoing data between the device and the guest domain[1]

The Xen design follows an interrupt based approach in the communication channel[4]. This interrupt based model requires the context switches[4]. In a multitasking system, context switch refers to the switching of the CPU from one process or thread to another. Context switch makes multitasking possible. At the same time, context switch causes unavoidable system overhead[23, 26]. Hence the overhead is caused by the communication channel in the Xen driver domain.

We propose and implement an alternate approach for improving the performance of the driver domain. Further, we conduct a comparative analysis and evaluate the performances of both the approaches.

1.3 Core Contributions

The core contributions of this project are listed below.

1. The implementation of the Xen driver domain concept using alternate design.
2. The performance comparison of the alternate approach with the xen driver domain.

1.4 Organization

This section gives the organization and roadmap of the thesis.

1. Chapter 2 gives the background on memory protection, virtualization, Xen Hypervisor, inter-domain communication, processes and threads.
2. Chapter 3 gives the introduction to design of the system to isolate device driver.
3. Chapter 4 discusses the detailed design and implementation to isolate device driver.
4. Chapter 5 evaluates the performance of Independent device driver with different designs.
5. Chapter 6 reviews the related work in the area of kernel fault tolerance.
6. Chapter 7 concludes the report and lists down the topics where this work can be extended.

Chapter 2

Background

This section gives a background on operating system concepts such as Processes, threads, Memory protection, Virtualization and Hypervisor.

2.1 Processes and threads

2.1.1 Process

Process is a program in execution or an abstraction of a running program. Process can be called as the most central concept in an operating system. In early days, computer systems allowed only one program to be executed at a time. These programs had complete control of the system and hence could access all the resources of the system without any complications. However, modern computer systems allow multiple programs to be executed concurrently. To load and run multiple programs, operating systems require control over resource access and allocation. Isolation of the various programs is needed for resource allocation, which results in need of a process.[32].
A program is a passive entity, it is not a process by itself. Program is a file stored on a disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when the file is loaded into the memory[32].

A process is more than the program code, or the text section. It includes the current activity, with the help of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data such as function parameters, return addresses, and local variables, and a data section, which contains global variables. A process may include a heap, which is a memory that is dynamically allocated during process run time[32].

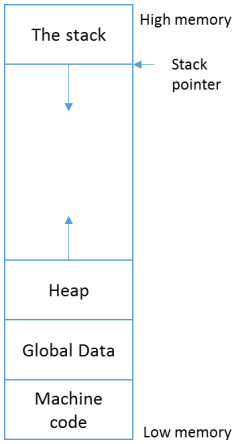

Figure 2.1: Program's memory map

2.1.2 Threads

Each process has an address space. A process has either single or multiple threads of control in that same address space. Threads run as if they were separate processes although they share the address space. [32]

Thread is also called as a light-weight process. The implementation of threads and processes differ in each operating system, but in most cases, thread is contained inside a process. Multiple threads can exist within the same process and share resources such as code, and data segment, while different processes do not share these resources. If a process has multiple threads then it can perform more than one task at a time.

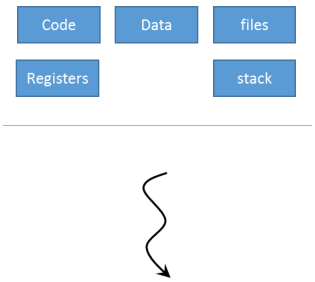

Figure 2.2: Single threaded process

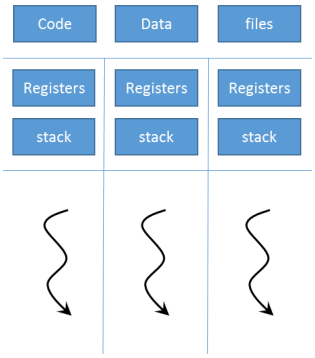

Figure 2.3: Multithreaded process

2.1.3 Context Switch

Multithreading is implemented by time division multiplexing on a single processor. In time division multiplexing, the processor switches between different threads. The switch between threads is called as the context switch. The context switch makes the user feel that the threads or tasks are running concurrently. However, on a multi-core system or multiprocessor system, threads can run truly concurrently, with every processor or core executing a separate thread simultaneously.

In a context switch the state of a process is stored and restored, so that the execution can be resumed from the same point at a later time. The state of the process is also called as context. The context is determined by the processor and the operating system. Context switching makes it possible for multiple processes or threads to share a single processor. Usually context switches are computationally intensive. Switching between two process requires good amount of computation and time to save and load registers, memory maps, and updating various tables[32] .

2.1.4 Spinlocks and spinning

Spinlock is a lock which causes a thread trying to acquire it to spin continuously checking if the lock is available. Spinning is a technique in which a process repeatedly checks to see if a condition is true. Spinlock is one of the locking mechanisms designed to work in a multiprocessing environment. Spinlocks are similar to the semaphores, except that when a process finds the lock closed by another process, it spins around continuously. Spinning is implemented by executing an instruction in a loop[7].

Whenever a lock is not available, a CPU either spins or does a context switch. In a uniprocessor environment, the waiting process keeps spinning for the lock. However, the other process holding the lock might not have a chance to release it, because of which spin lock could deteriorate the performance in a uniprocessor environment. In a multiprocessor environment, spin locks can be more efficient. Overhead for spinlocks is very small. On the other hand, a context switch takes a significant amount of time, so it is more efficient for each process to keep its own CPU and simply spin while waiting for a resource[7]. Because spinlocks avoid overhead from operating system process re-scheduling or context switching, spinlocks are efficient if threads are only likely to be blocked for a short period. For the above reasons, spinlocks are often used inside operating system kernels.

2.2 Memory protection

The memory protection mechanism of computer systems control the access to objects. The main goal of memory protection is to prevent malicious misuse of the system by users or programs. Memory protection also ensures that each shared resource is used only in accordance with the system policies. In addition, it also helps to ensure that errant programs cause minimal damage. However, memory protection systems only provide the mechanisms for enforcing policies and ensuring reliable systems. It is up to the administrators, programmers and users to implement those mechanisms[32, 18]. The following subsections explain how these policies are implemented at kernel level and user level.

2.2.1 User level

Typically in a monolithic kernel, the lowest X GB of memory is reserved for user processes (In 32-bit architecture 3GB is reserved for user level). The upper $'VM - X'$ GB is reserved for kernel (In 32 bit architecture 1 GB is reserved for kernel). This upper 1 GB is restricted to $CPL0$ ($ring0$) only. The kernel puts its private data structures in the upper 1GB and always accesses them at the same virtual address, irrespective of what processes are running.

Figure 2.4: Physical memory

At user space, each application runs as a separate process. Each process is associated with an address space and believes that it owns the entire memory, starting with the virtual address 0. However, a translation table translates every memory reference by these processes from virtual to physical addresses. The translation table maintains $< base, bound >$ entry. If a process tries to access virtual address which is out of $'base + bound'$ then error is reported by the OS, otherwise physical address $'base + virtualaddress'$ is returned. This allows multiple processes to be in memory with protection. Since address translation provides protection, a process cannot access to other processes addresses, nor about the OS addresses.

Consider an example in below diagram.

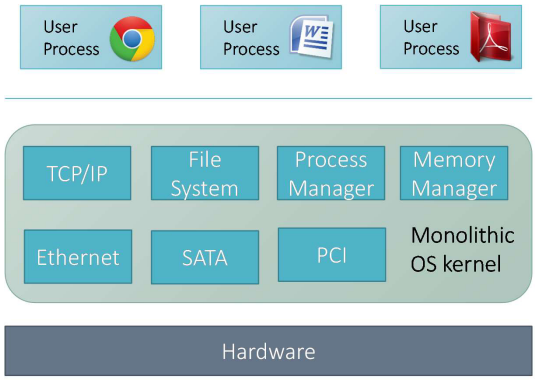

Figure 2.5: User space

In the above system Chromium browser, word processor and pdf reader are running as a 3 different processes in user space.

The word process hits a bug and tries to corrupt the memory out of the address space.

Since the access to the address is restricted because of memory protection, the word processor does not crash the other application or system.

The bug in the word processor might lead to crashing itself.

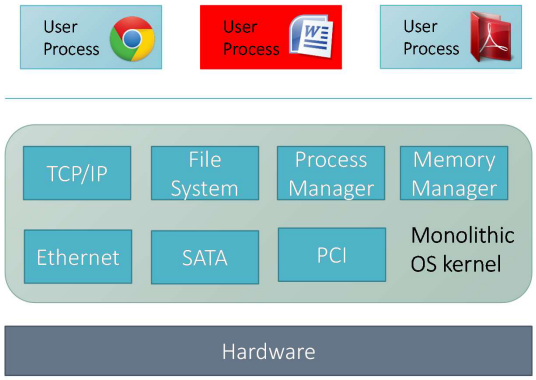

Figure 2.6: User space: Word processor hits a bug

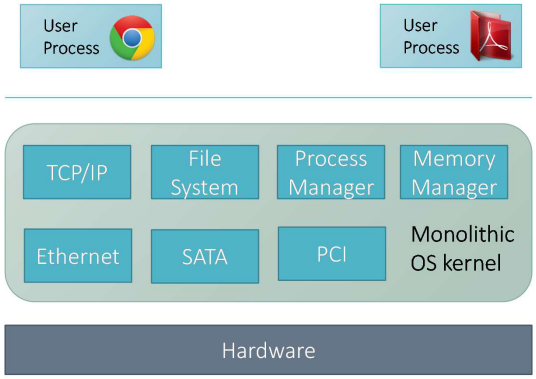

Figure 2.7: User space : Word processor crashes and system is still intact

2.2.2 kernel level

Kernel reserves upper $1Gb$ of virtual memory for its internal use. The page table entries of this region are marked as protected so that pages are not visible or modifiable in the user mode. This reserved region is divided into two regions. First region contains page table references to every page in the system. It is used to do translation of address from physical to virtual when kernel code is executed. The core of the kernel and all the pages allocated by page allocator lies in this region. The other region of kernel memory is used by the memory allocator such as *vmalloc*(), the allocated memory is mapped by kernel modules using *kmap*() or *vremap*(). Since an operating system maps physical addresses directly, kernel components do not have memory protection similar to that of the user space. At kernel level any code running at CPL 0 can access the 1 GB of kernel memory, and hence any kernel component can access and corrupt the kernel data structure.
Consider an example below.

1. In the system shown in figure 2.8, Chromium browser, word processor and pdf reader are running as 3 different processes in the user space and many different kernel components are running.

2. The network driver hits a bug, and corrupts a kernel data structure. This memory corruption might crash the other components, and might lead to a system crash.

3. Ideally with proper memory protection policy implementation and proper decoupling

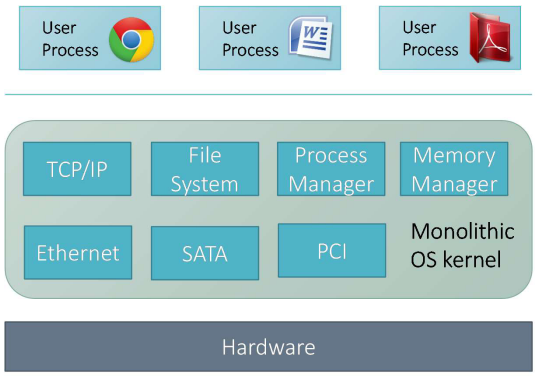

Figure 2.8: Kernel space

between network device driver and kernel, only network device driver and applications using network device driver (chromium in this case) should have been crashed.

4. But because of tight coupling between device driver and kernel, complete system crashes.

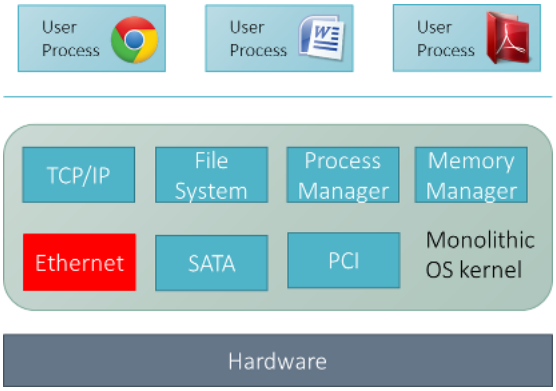

Figure 2.9: Kernel space

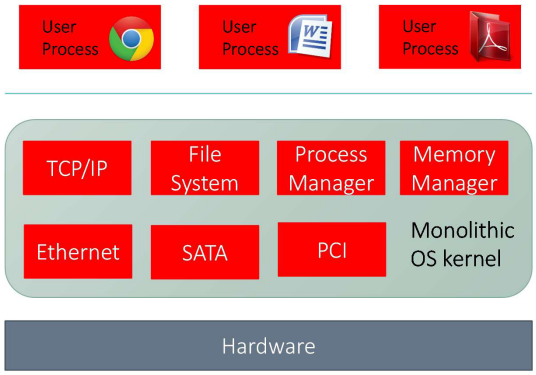

Figure 2.10: Kernel space

2.3 Virtualization

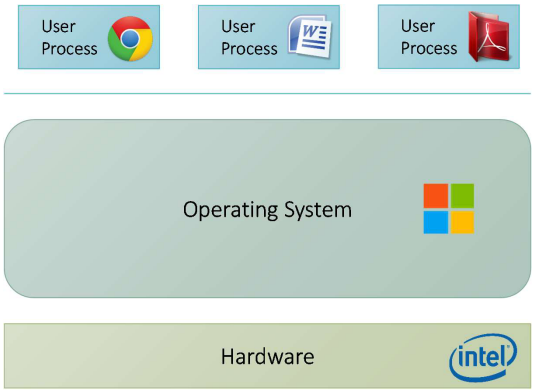

Figure 2.11: Operating System Architecture

Virtualization, is the act of creating a virtual version of hardware platform, operating system, storage device, or computer network resources etc. In operating system virtualization, the software allows a hardware to run multiple operating system images at the same time. Virtualization was invented to allow large expensive main frames to be easily shared among different application environments.[24] Originally introduced for VM/370 [12], the idea later emerged for modern platforms [8, 31].

With the high hardware prices there was a need to share a single computer system between multiple users. This introduced the need to provide isolation between the users, which was achieved by providing the time-sharing systems. Virtualization concept started with the need to provide such isolation between users.

The addition of user and kernel modes on processors protected the operating system code from user programs. A set of privileged instructions reserved for the operating system software could run only in the kernel mode. The invention of Memory protection and later virtual memory made it possible to separate address spaces. The address space is assigned to different processes to share the system's physical memory and ensures that the use of memory is mutually exclusive by different applications. Before introduction of virtualization these enhancements were sufficient within an operating system. But the need to run different applications and users and different operating systems on a single physical machine could be satisfied only by virtualization.[13]

Virtualization has the capability to share the underlying hardware resources and still provides

isolated environment to each operating system. In virtualization each operating system runs independently from the other on its own virtual processors. Because of this isolation the failures in an operating system are contained. Virtualization is implemented in many different ways. It can be implemented either with or without hardware support. Also operating system might require some changes in order to run in a virtualized environment, or it can also function without making any changes.[15]. It has been shown that virtualization can be utilized to provide better security and robustness for operating systems[16, 22, 30].

2.3.1 Hypervisor

Hypervisor is a piece of computer software, firmware or hardware that creates and runs virtual machines. Operating system virtualization is achieved by inserting a hypervisor between the guest operating system and the underlying hardware. Most of the literature presents hypervisor synonymous to virtual machine monitor (VMM). While, VMM is a software layer specifically responsible for virtualizing a given architecture, a hypervisor is an operating system with a VMM. The operating system may be a general purpose one, such as Linux, or it may be developed specifically for the purpose of running virtual machines[3]. A computer on which a hypervisor is running one or more virtual machines, is defined as a host machine. Each virtual machine is called a guest machine. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. Multiple instances of a variety of operating systems may share the virtualized hardware resources. Among widely known hypervisors are Xen [5, 10], KVM[19, 21], VMware ESX[3],and VirtualBox[9].

There are two types of hypervisors [17]

Type 1 hypervisors are also called as native hypervisors or bare metal hypervisors. Type 1 hypervisor runs directly on the host's hardware to control the hardware and to manage guest operating systems. A guest operating-system, thus, runs on another level above the hypervisor. Type 1 hypervisor represents the classic implementation of virtual-machine architectures such as SIMMON, and CP/CMS. Modern equivalents include Oracle VM Server for SPARC, Oracle VM Server, the Xen hypervisor[5], VMware ESX/ESXi[3] and Microsoft Hyper-V.

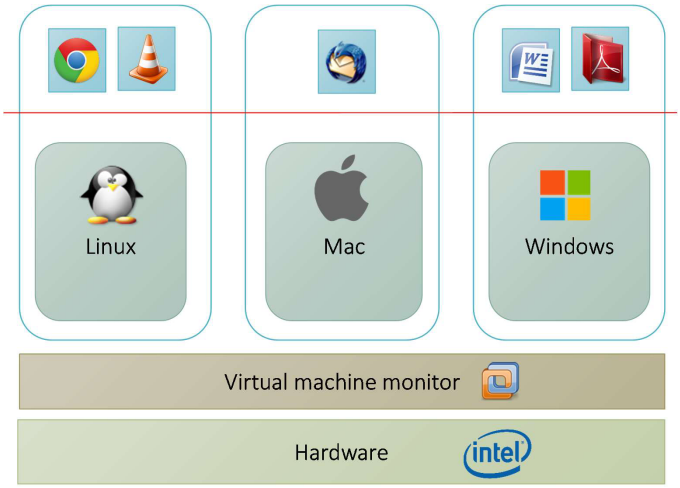

Figure 2.12: Virtualization

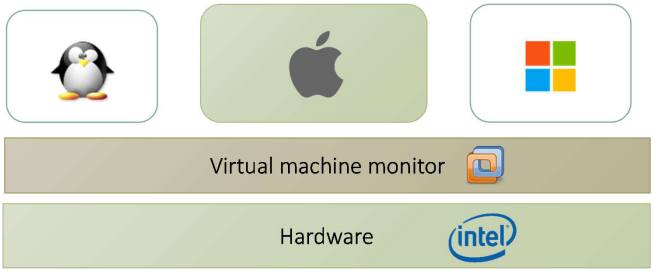

Figure 2.13: Type 1 hypervisor

Type 2 hypervisors are also called as hosted hypervisors. Type 2 hypervisor runs within a conventional operating-system environment. Type 2 hypervisor runs at a distinct second software level whereas, guest operating systems run at the third level above hardware. VMware Workstation and VirtualBox are some of the examples of Type 2 hypervisors[34, 9].

2.3.2 Xen Hypervisor

Xen[5] is a widely known Type I hypervisor that allows execution of virtual machines in guest domains[20]. Figure **??** represents a diagram showing the different layers of a Type I hypervisor system. The hypervisor itself forms the lowest layer, which consists of the hypervisor kernel and Virtual Machine Monitors. The kernel has direct access to the hardware and

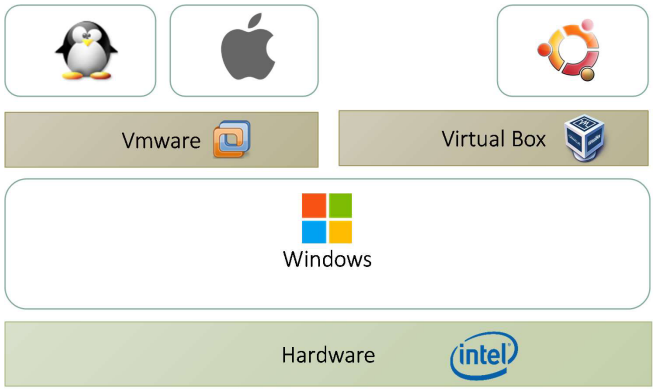

Figure 2.14: Type 2 hypervisor

is responsible for resource allocation, resource scheduling and resource sharing. A hypervisor is a layer responsible for virtualizing and providing resources to a given operating system. The purpose of a hypervisor is to allow guests to be run. Xen runs guests in environments known as domains. Domain 0 is the first guest to run, and has elevated privileges. Xen loads a Domain 0 guest kernel during boot. Other unpriviledged domains are called as domain U. Xen hypervisor does not include device drivers. Device management is included in privileged domain *Dom*0. *Dom*0 uses the device drivers which are present in its guest kernel implementation. However, *domU* accesses devices using a split device driver architecture. In the split device architecture a front end driver in a guest domain communicates with a back end driver in *Dom*0.

Figure 2.15 shows what happens to a data when it is sent by an application running in a domU guest. First, it travels through the file system as it would normally. However, at the end of the stack the normal block device driver does not exist. Instead, a simple piece of code called the front end puts the data into the shared memory. The other half of the split device driver called the back end, running on the dom0 guest, reads the data from the buffer, sends it way down to the real device driver. The data is written on actual physical device. In conclusion, split device driver can be explained as a way to move data from the domU guests to the dom0 guest, usually using ring buffers in shared memory[10].

Xen provides an inter-domain memory sharing API accessed through the guest kernel extensions, and an interrupt-based inter-domain signaling facility called event channels to implement the efficient inter-domain communication. Split drivers use memory sharing APIs to implement I/O device ring buffers to exchange data across domains.

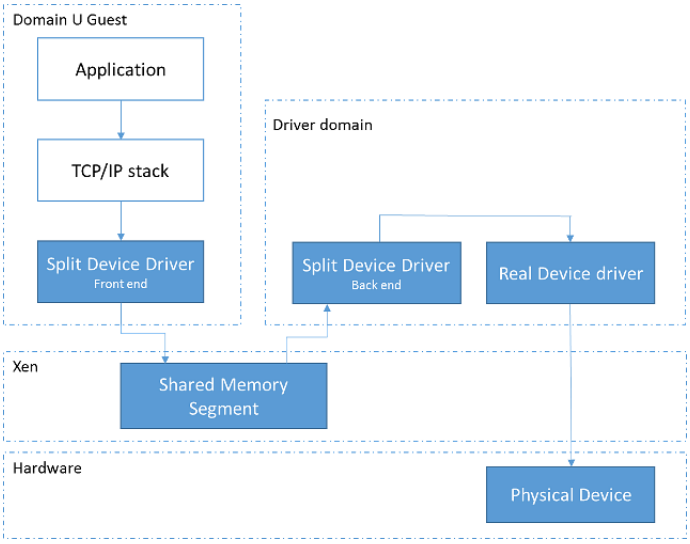

Figure 2.15: Xen split device driver

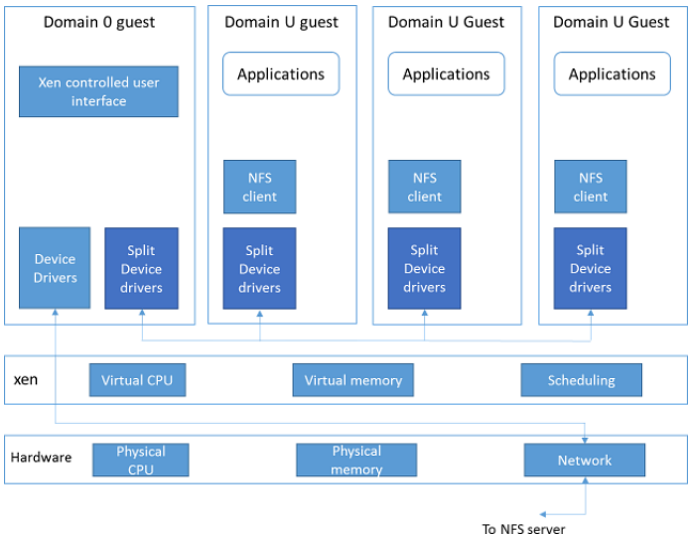

Figure 2.16: Xen

In driver domain implementation, xen uses shared I/O ring buffers and event channel[5, 27, 28].

2.3.2.1 Hypercalls and events

Hypercalls and event channels are two mechanisms that exist for interactions between Xen and domains. A hypercall is a software trap from a domain to the Xen, just as a syscall is a software trap from an application to the kernel[2]. Domains use the hypercalls to request privileged operations like updating pagetables.
Event channel is to Xen hypervisor as hardware interrupt is to operating system. Event channel is used for sending asynchronous notifications between domains. Event notifications are implemented by updating a bitmap. After scheduling pending events from an event queue, the event callback handler is called to take appropriate action. The callback handler is responsible for resetting the bitmap of pending events, and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen readable software flag: this is analogous to disabling interrupts on a real processor. Event notifications can be compared to traditional UNIX signals acting to flag a particular type of occurrence. For example, events are used to indicate that new data has been received over the network, or a virtual disk request has completed.

2.3.2.2 Data Transfer: I/O Rings

Hypervisor introduces an additional layer between guest OS and I/O devices. Xen provides a data transfer mechanism that allows data to move vertically through the system with minimum overhead. Two main factors have shaped the design of I/O transfer mechanism which are resource management and event notification.
Figure 2.18 shows the structure of I/O descriptor ring. I/O descriptor ring is a circular queue of descriptors allocated by a domain. These descriptors do not contain I/O data. However, I/O data buffers are allocated separately by the guest OS and is indirectly referenced by these I/O descriptors. Access to I/O ring is based around two pairs of producer-consumer pointers.

1. Request producer pointer: domains place requests on a ring by advancing request producer pointer.

2. Request consumer pointer: Xen removes requests which are pointed by request producer pointer by advancing a request consumer pointer.

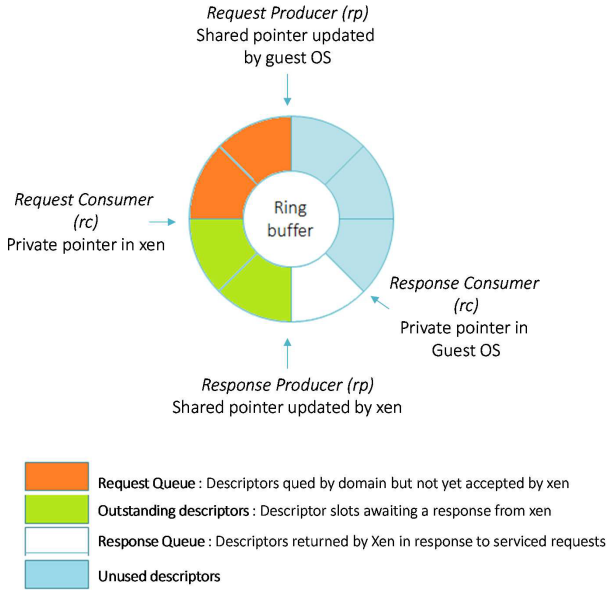

Figure 2.17: Ring I/O buffer

3. Response producer pointer: Xen places responses on a ring by advancing response producer pointer.

4. Response consumer pointer: Domains remove responses which are pointed by response producer pointer by advancing a response consumer pointer.

The requests are not required to be processed in order. I/O rings are generic to support different device paradigms. For example, a set of *requests* can provide buffers for read data of virtual disks; subsequent *responses* then signal the arrival of data into these buffers.
The notification is not sent for production of each request and response. A domain can en-queue multiple requests and responses before notifying the other domain. This allows each domain to trade-off between latency and throughput.

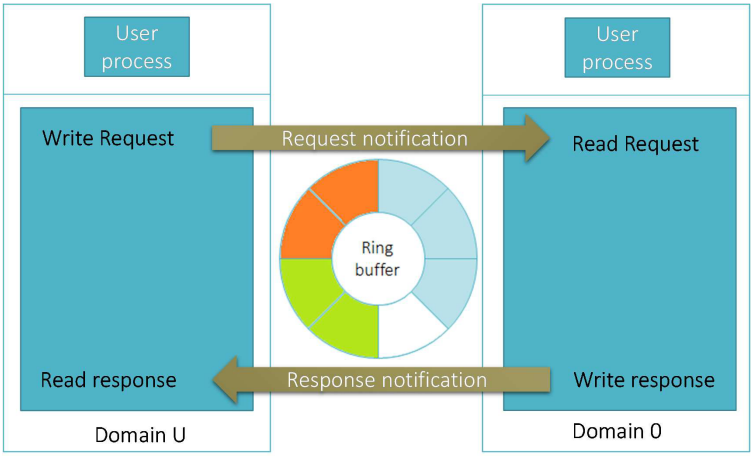

Figure 2.18: Ring I/O buffer

Chapter 3

System Introduction

3.1 Design Goals

The goal of the solution we are implementing is to provide full isolation between device driver and monolithic kernel, at the same time avoiding modifications to the device drivers. The most important goal of this thesis is to minimize the performance penalty because of the communication between the domains. In the system implementation we explore opportunities to minimize the overhead because of the communication module at the same time we achieve the original goals of the system such as application compatibility and transperency and strong isolation.

In this section we will discuss each goal briefly.

3.1.1 Performance improvement

Many of the solutions to decouple device driver from linux kernel use virtualization. The performance of the system we would be comparing with in this thesis is the device driver domain implementation by Xen. Even though virtualization provides better robustness for operating systems, it deteriorates the performance. The reasons for the performance deterioration can be introduction of an extra layer of hypervisor, data copy overhead and

communication between the domains. In case of the solutions based on xen hypervisor, applications running on dom 0 has to go through the virtual monitor to access the hardware. This degrades the performance of the system. For data intensive operations such as read and write, virtualization based solution needs to copy data from hardware to the domain running isolated driver and then from driver domain to the domain running applications. Introduction of extra copy from driver domain to application domain is also responsible for the performance degradation. Virtualization based solution needs to run driver in a separate domain. To send request and get response from device driver, application domain and driver domain communicates with each other. Communication between domains adds an overhead to the system performance. We strive to minimize the overhead during communication between driver and application domain.

3.1.2 Strong isolation

Strong isolation is the one of the main goals of the virtualization based solutions we aim to improve performance of. In device driver domain implementation of xen device driver domain adds extra layer of isolation in the design which provides fault isolation between kernel and device driver. Virtualization also adds the ability to manage individual components independently increasing the availability of the system.

3.1.3 Compatibility and transperency

Usually extending existing OS structure leads to the result in a large number of broken applications. Microkernels historically required effort to retain compatibility with existing applications[**?**]. To provide compatibility with applications, either an emulation layer was implemented [**?**], or a multiserver operating system [**?**, 36] was built on top of a microkernel. Since device driver domain of xen maintains compatibility between existing drivers and applications, it was one of the basic need of our implementation. Our implementation made sure that design wouldnt require any changes in the device driver and application to run the system.

3.2 System overview

The architectural overview of the system is presented in Figure 1.

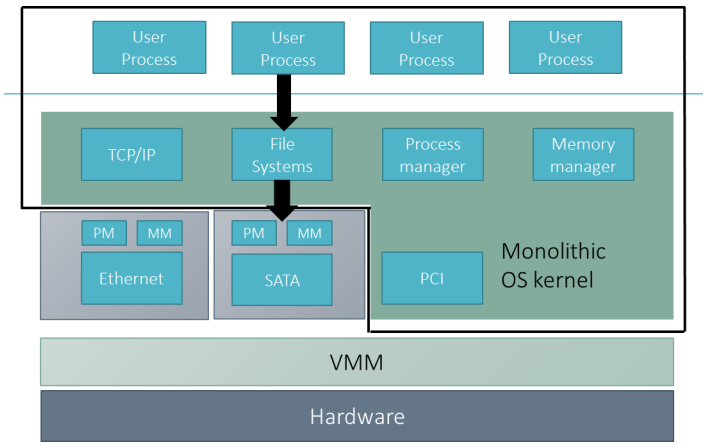

Figure 3.1: Overview

The architecture partitions an existing kernel into multiple independent component. The user applications and linux kernel run in a domain called as *application domain*, and device driver which needs to be isolated from the kernel executes in the sepetate domain called as *driver domain*. Sticking to the driver domain design goal, we would not want to make any changes to the device driver code. However, a device driver cannot run standalone in a separate domain, hence *driver domain* runs a minimilistic kernel with the device driver. Here *driver domain* does not run any applications. Also user would be unaware of driver domainand hence the system would be transperent.

In case of multiple device drivers being decoupled from linux kernel, each device driver would run in a different instance of a device domain, which represents a light-weight subset of kernel functionality dedicated to a particular function. However, irrespective of that there would be only one application domain, and it is dedicated to core system tasks such as process management, scheduling, user memory management, and IPC. The sole task of driver domains is to handle requests coming from the user processes managed by the primary domain.

Although current design assume that there is only one application domain in which user processes can run; it could be extended to support multiple user environments, each having its own primary and set of driver domains. [1, 28]

3.3 System components

3 main components of the design are Front end driver, Back end driver, Communication module.

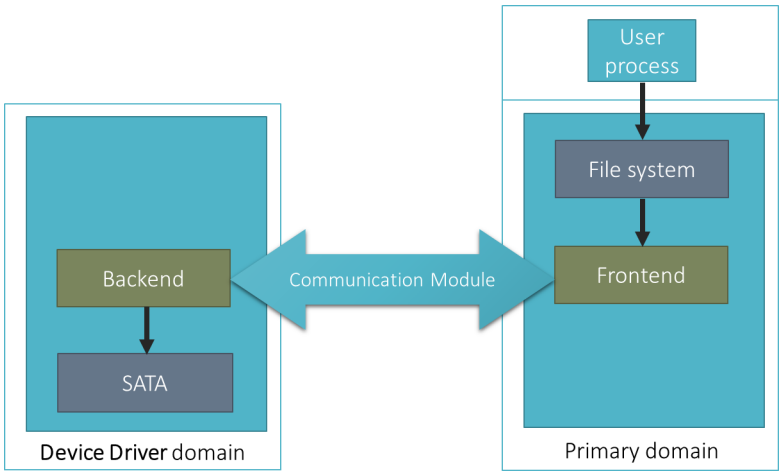

Figure 3.2: System Components

3.3.1 Front end driver

Transperency is one the design goal of the driver domain system. Any changes to operating system and operating system APIs would be against the transperency goal. Also another system design goal is to avoid any changes to device drivers. However, device driver runs in a separate driver domain and, user applications in Application domain. Since the user application wouldn't know that device driver is in a remote driver domain, and it won't be possible for application to send request to driver in driver domain without making any changes to Operating System API. Hence we introduced a kernel module *front end* which runs in a application domain, and acts as a psuedo device driver for user application. For user application, front end driver would acts as a substitute for device driver. Front end driver would accept request from the user application and forward it to the driver domain. The main functionality of front end driver is to accept request from the user application, process the request, enqueue the request for driver domain and notify the driver domain. After that front end driver will also take care of processing the responses and ending the

corresponding request.

3.3.2 Back end driver

Operating system and device driver provides an API to accept request from user application running in same domain. The operating system is not capable of accepting requests from application running in a separate domain without making any changes to it. At the same time, device driver is not capable of sending responses back to the application domain without making any changes to the device driver code. And since in design goals we mention that we avoid to make any changes to the device driver code, *back end driver* is introduced. The responsibilty of back end driver is to accept requests from application domain and forward the requests to device driver. Upon receiving reponses from the device driver, back end driver sends back the responses, and notifies the application domain.

3.3.3 Communication module

Communication module is the communication channel between front end driver and back end driver. Unlike backend and front end driver, communication module is not a separate physical entity or a kernel module. Part of the communication module is in front end driver and part is in backend driver. Communication channel is logically divided into three parts. Responsibility of 1st part is to forward the requests from application domain to driver domain and to forward the responses from driver domain to application domain. Responsibility of the second part is to share the read write data. And responsibility of the third part to notify other domain upon a particular event.

3.4 System design

Following section explain the system design.

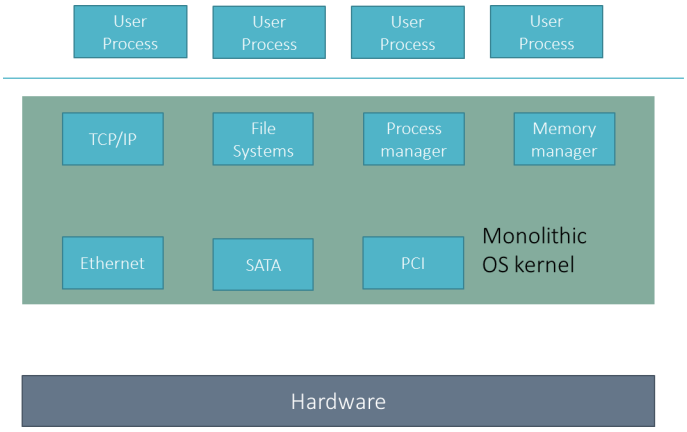

Figure 3.3: Tightly coupled System

Figure **??** shows the architecturial overview of modern operating system with a monolithic kernel. As we have seen in previous chapter, because of tight coupling between kernel and device driver, a bug in a device driver can crash the system. We implement a driver domain to decouple device driver from linux kernel. Figure **??** illustrates the concept.

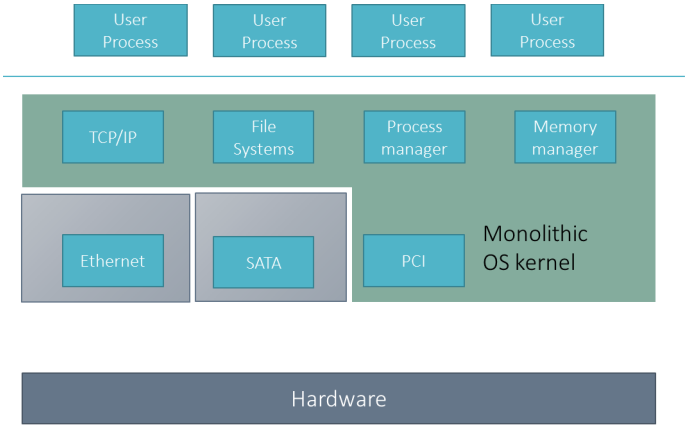

Figure 3.4: System with kernel and decoupled, standalone device driver

Though solution to provide more protection at kernel level is to isolate device driver from Linux kernel, it is not possible to run a standalone device driver. A Device driver is dependent

on kernel components such as scheduler, memory management unit. It is not possible to run standalone device driver without making any changes. Hence a minimal kernel provides the functionality on which device driver is dependent on. Figure **??** shows that each device driver runs with a minimalistic version of kernel.

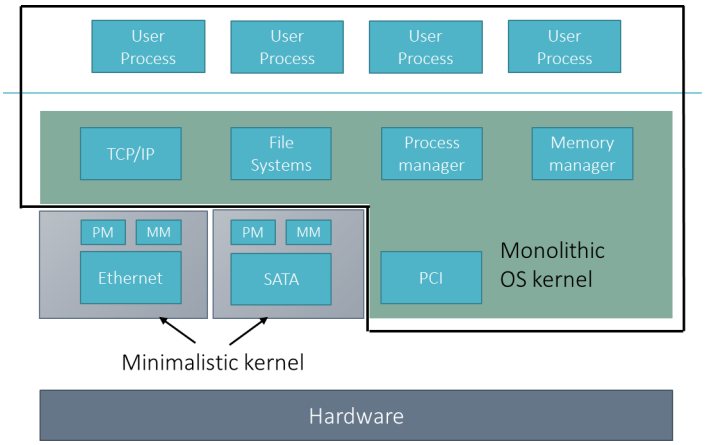

Figure 3.5: Each instance of device driver runs with a kernel

However, it is not possible to run multiple monolithic kernels over same hardware without any virtual machine monitor. Virtualization could be used to run multiple monolithic kernels on an virtual machine monitor.

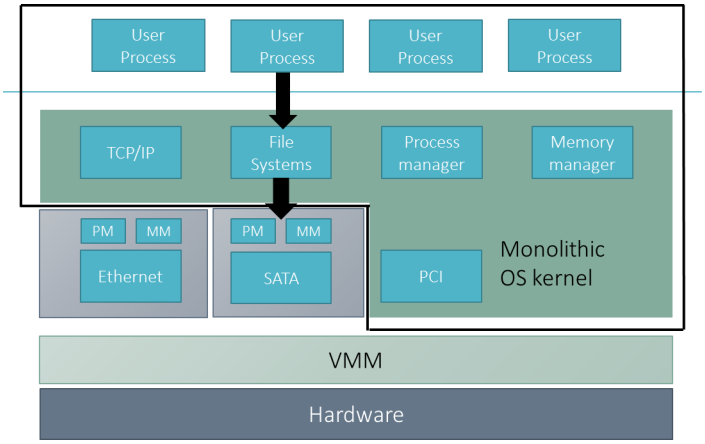

Figure 3.6: All the instances of operating system execute over VMM

Figure **??** and Figure **??** explains the effect of a malicious driver when it is isolated from Linux kernel. When a device driver running in a driver domain domain hits a bug, it

will crash the kernel of the driver domain and hence domain executing itself. In addition, applications in an application domain which are expecting a response from the driver domain might crash. But because of the address space separation of application domain and driver domain, system will be intact.

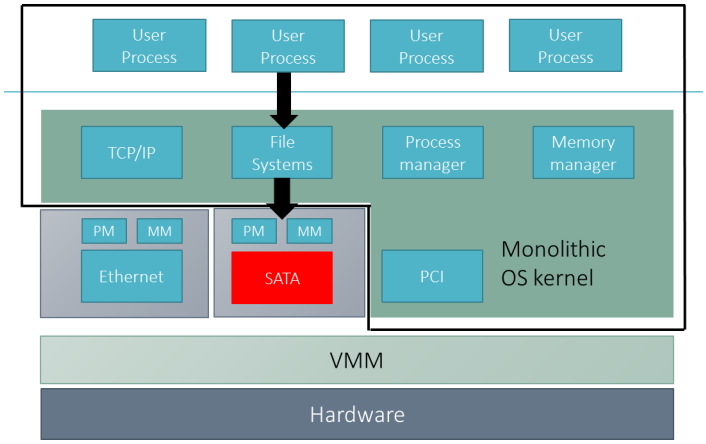

Figure 3.7: Device driver crash

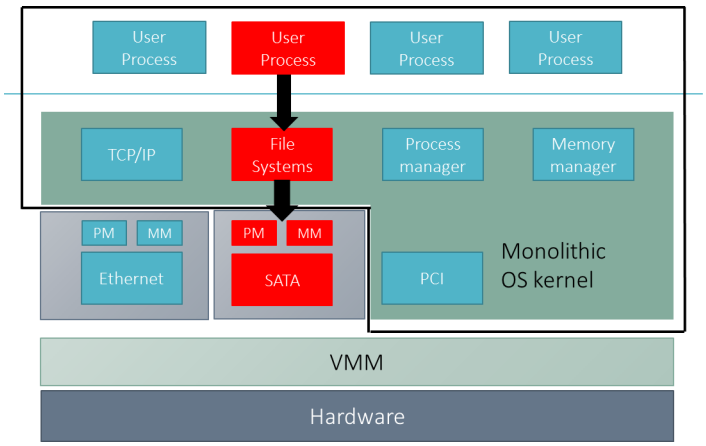

Figure 3.8: High Availability

Following section explains the evolution and introduction of different components of the system.

Figure **??** shows the component overview of the system design explained in the figure **??**. Since one of the goal of this design is to keep device driver and OS unchanged, an user application would not be aware of the remote device driver, and can not send the request

to remote domain. Hence front end module was introduced in the design. Like explained in the previous section, responsibility of front end module is to forward the request from user application to remote domain and return the response from remote domain to user application. Similarly device driver wont be aware that the request has come from a remote domain and wont be able to return the response to remote domain. Hence back end module was introduced in the design. Whose responsiblity is to forward the response to the primary domain. Figure **??** illustrates the role of front end and backend module.

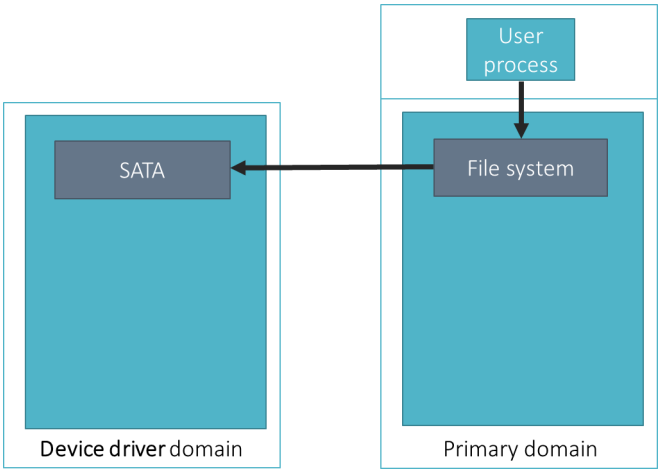

Figure 3.9: Component wise view of Figure 4

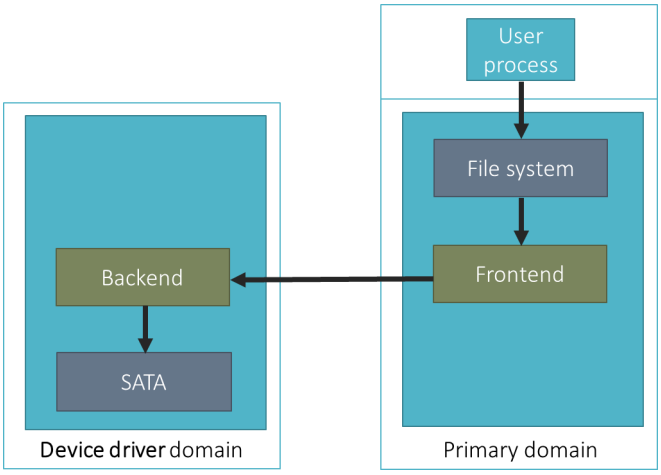

Figure 3.10: Introduction of front end and back end modules

It is not possible to share the data, requests and responses between domains without any communication model. Figure **??** illustrates the role of communication model.

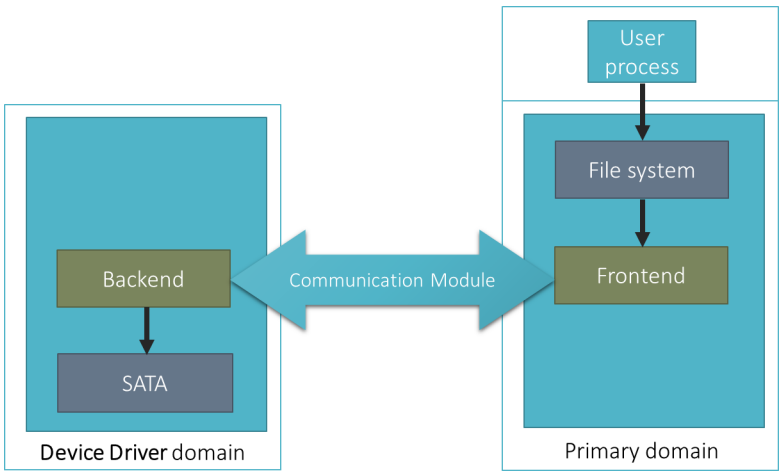

Figure 3.11: Introduction of Communication module

Chapter 4

System Design and Implementation

4.1 Implementation Overview

4.2 Implementation

4.2.1 Communication component

4.2.1.1 Ring buffer

4.2.1.2 Shared pages

4.2.1.2.1 Hypercall interface

4.2.1.2.2 Other interfaces

4.2.2 Application domain

4.2.2.1 Front end driver

4.2.2.1.3 Initialization

4.2.2.1.4 Create request

4.2.2.1.5 Enqueue request

Chapter 5

Related Work

Related work goes here

aspos paper has good related work

5.1 Driver protection approaches

5.2 Existing Kernel designs

Chapter 6

Evaluation

6.1 Goals and Methodology

6.1.1 Goals

6.1.2 Experiment Set Up

Experiment Set Ups

6.2 System Overhead

6.2.1 Copy Overhead

6.3 Results with event channel

Results goes here

6.4 Results with spinning

Results goes here

6.5 Comparision

Chapter 7

Conclusion and Future Work

7.1 Contributions

7.2 Future Work

The idea is make the system general enough to support multiple disaster relief studies.

Bibliography

[1]

[2] hypercall. `http://wiki.xen.org/wiki/Hypercall`.

[3] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4):3–18, December 2010.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[6] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation0. *Commun. ACM*, 27(1):42–52, January 1984.

[7] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[8] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[9] Fernando L. Camargos, Gabriel Girard, and Benoit D. Ligneris. Virtualization of Linux servers: a comparative study. In *2008 Ottawa Linux Symsposium*, pages 63–76, July 2008.

[10] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[12] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.

[13] Simon Crosby and David Brown. The virtualization reality. *Queue*, 4(10):34–41, December 2006.

[14] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[15] Ulrich Drepper. The cost of virtualization. *Queue*, 6(1):28–35, January 2008.

[16] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[17] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, New York, NY, USA, 1973. ACM.

[18] G. Scott Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[19] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166), February 2008.

[20] Samuel T. King and et al. Operating system support for virtual machines.

[21] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[22] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[23] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[24] Daniel A. Menasc. Virtualization: Concepts, applications, and performance modeling, 2005.

[25] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[26] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. *SIGPLAN Not.*, 26(4):75–84, April 1991.

[27] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 116–132, New York, NY, USA, 2013. ACM.

[28] year = 2013 Nikolaev, Ruslan and Back, Godmar, title = Design and Implementation of the VirtuOS Operating System.

[29] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM.

[30] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, July 2004.

[32] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[33] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[34] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[35] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[36] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure. *Computer*, 39:44–51, 2006.