Performance analysis of driver domain.

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,

Copyright 2013, Sushrut Shirole

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 the 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 studies shows that 6 to 16 bugs/1000 lines can be found within a typical module [7]. The Linux kernel has over 15 million lines of code. If we assume minimum estimate, Linux kernel may contain 90,000 bugs. Researchers have shown that device drivers has more bugs than the rest of the kernel [12]. Considering the fact that the operating system predominantly consists of device drivers, the bugs in device drivers make the operating system unreliable [12].

1.1 Problem Statement

The reliability of the system is influenced by the number of bugs in the device driver and the system should have small number of bugs in the device driver code. However, fixing all the bugs is difficult since bug fixes might introduce new lines of code resulting in new bugs. Modern operating systems provide isolation from bugs using memory protection. Memory protection is a way to control memory access rights. It prevents a process from accessing memory that has not been allocated to it. Memory protection prevents a bug within a process from affecting other processes, or the operating system [15, 35]. Linux kernel modules do not have the same level of isolation the user level applications have. Unlike user applications, Linux 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. The underlying cause of unreliability in the operating system is the lack of isolation 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. [25], Xen isolated driver domain [17]. 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 [5, 37, 28]. 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. Xen does not include device drivers for all

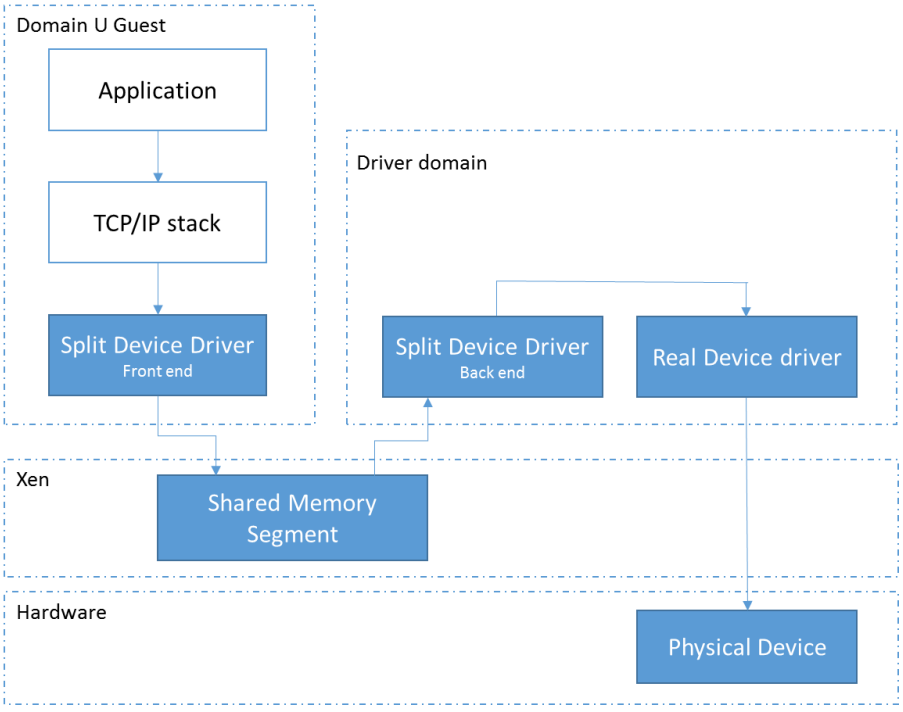

Figure 1.1: Split device driver model

the devices, adding support for all the devices would be a duplication of effort. Instead, Xen delegates hardware support to a guests. The guest typically runs in priviledged domain,

although it is possible to delegate hardware to guests in other domains. Model in which hardware support is delegated to a guest is called split device driver model [11]. Xen uses the same split device driver model for driver domain as shown in the figure 1.1.

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 [5]. In this communication model, front end and back end notifies each others the receipt of a service request and corresponding responses by sending an interrupt. This interrupt based model requires the context switches [5]. 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 [26, 29]. Hence the overhead is caused by the communication channel in the Xen driver domain.

In our proposed solution, a thread in the back end driver spins for the service request, and the front end driver spins for the availability of the corresponding responses. As a spinning does not involve any context switch, our solution performs better than the Xen device driver domain.

In this thesis, we re-implement the Xen driver domain and call it as Isolated Device Driver

(IDDR). The proposed solution is implemented with IDDR as a base code.

1.3 Core Contributions

The core contributions of this project are listed below.

1. Re-implementation of the Xen driver domain - Isolated Device Driver (IDDR).

2. Improve the performance of IDDR by implementing thread based communication channel instead of interrupt based communication channel.

3. The performance comparison of the thread based IDDR and interrupt based IDDR.

1.4 Organization

This section gives the organization and roadmap of the thesis.

1. Chapter 2 gives the background on Processes, threads, Memory protection, Virtualization, Hypervisor and inter-domain communication.

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.[35].

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[35].

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[35].

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. [35]

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

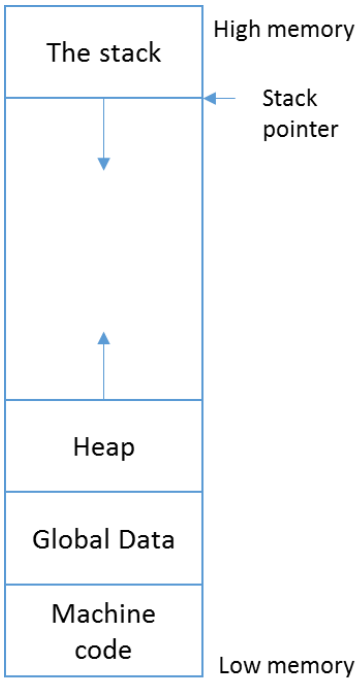

Figure 2.1: Program's memory map

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.

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

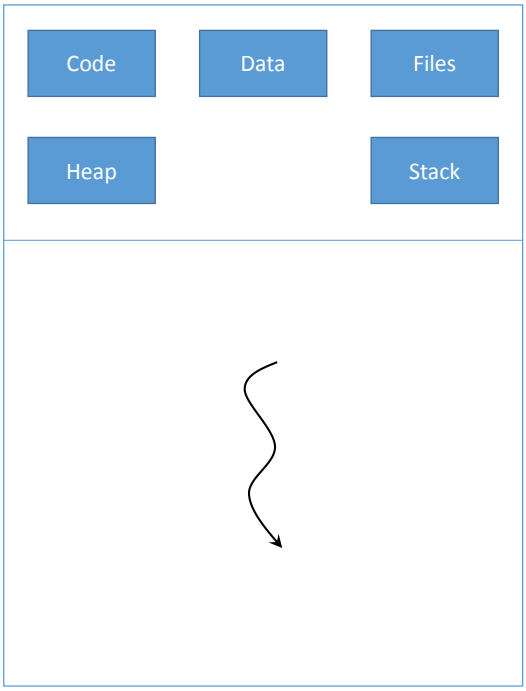

Figure 2.2: Single threaded process

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[35] .

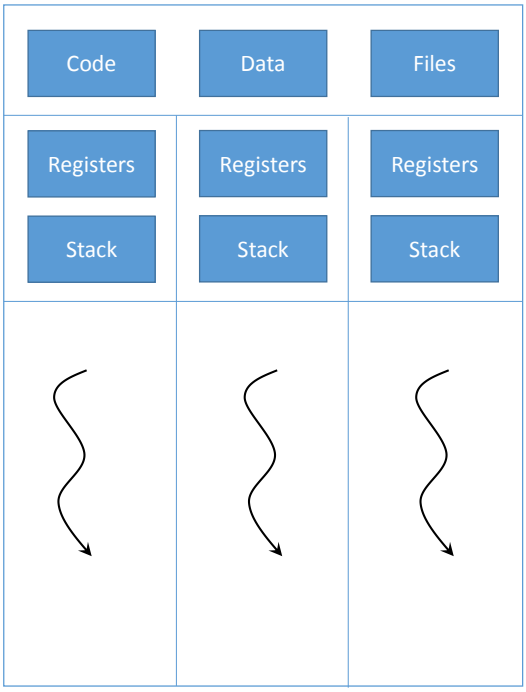

Figure 2.3: Multithreaded process

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[8].

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[8]. 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[35, 20]. 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.

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

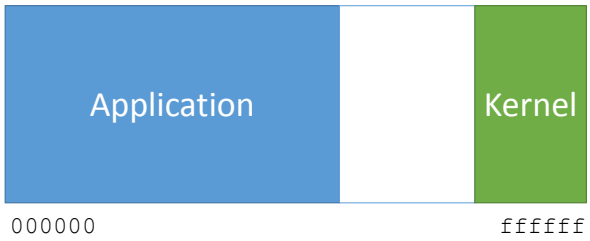

Figure 2.4: Physical memory

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.

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.

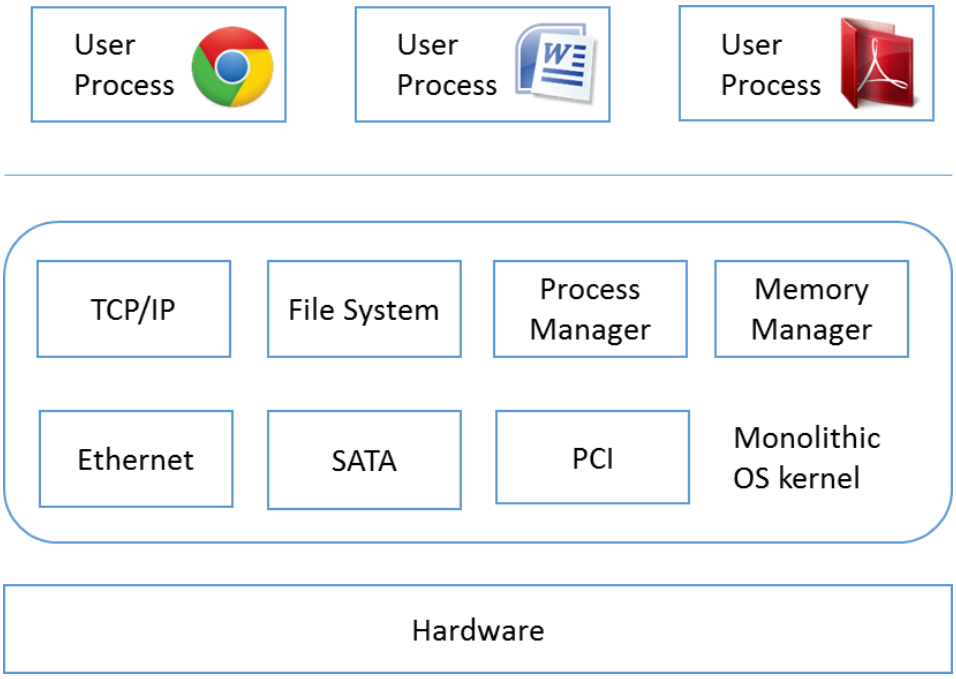

Figure 2.5: User space

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

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

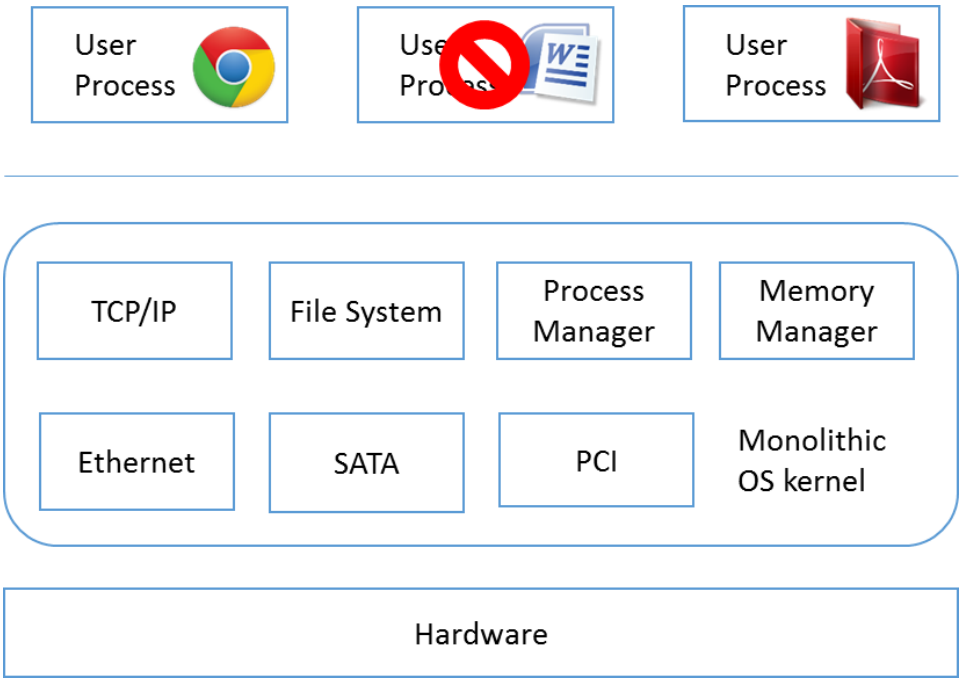

Figure 2.6: User space: Word processor hits a bug

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.

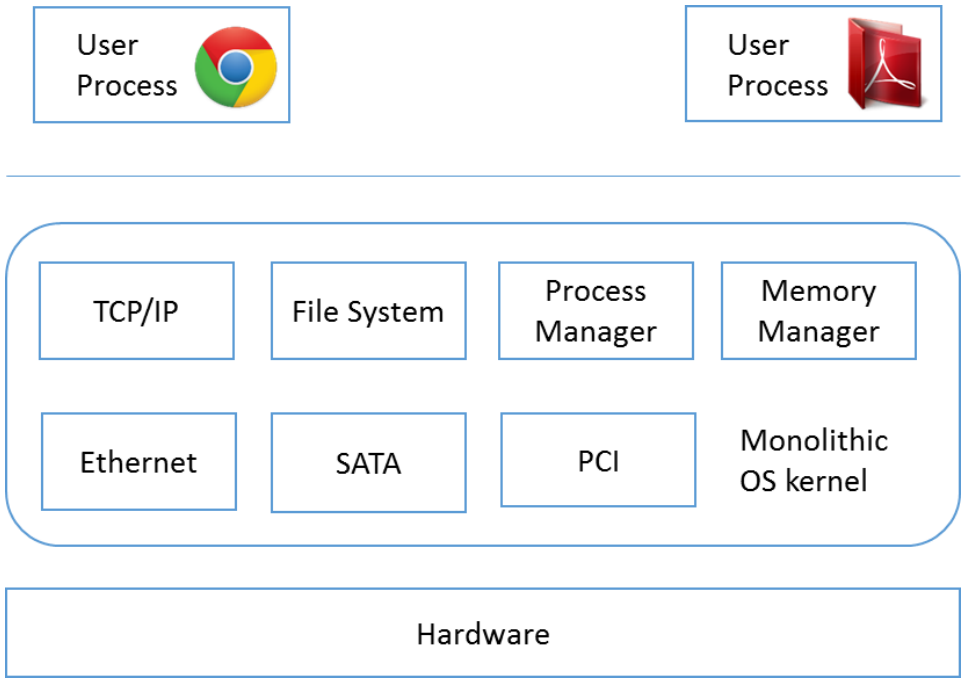

Figure 2.7: User space : Word processor crashes and system is still intact

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 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.

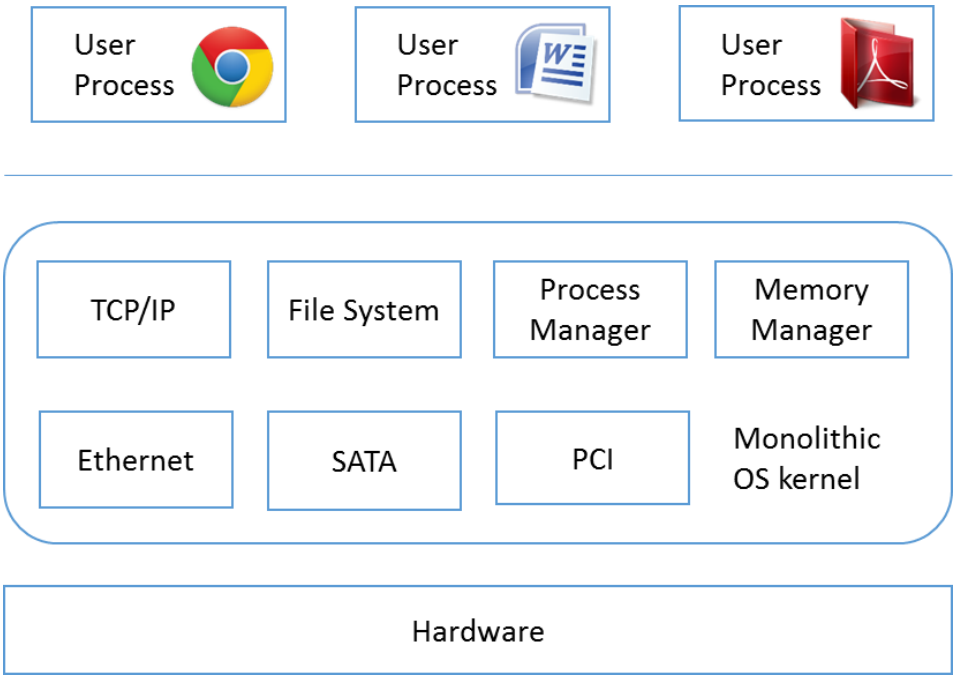

Figure 2.8: Kernel space

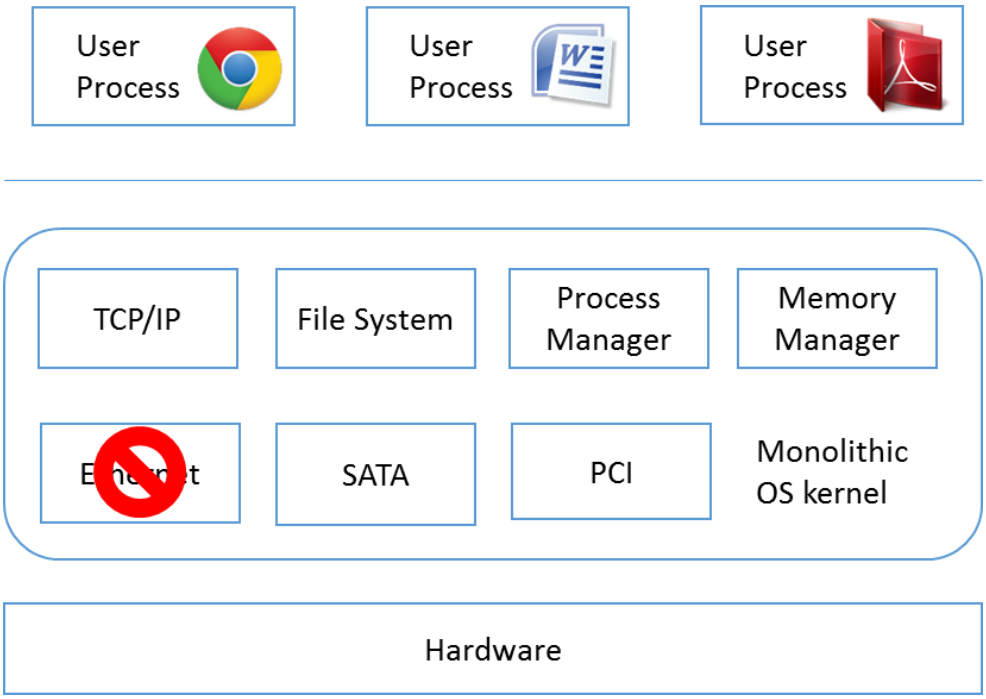

Figure 2.9: Kernel space

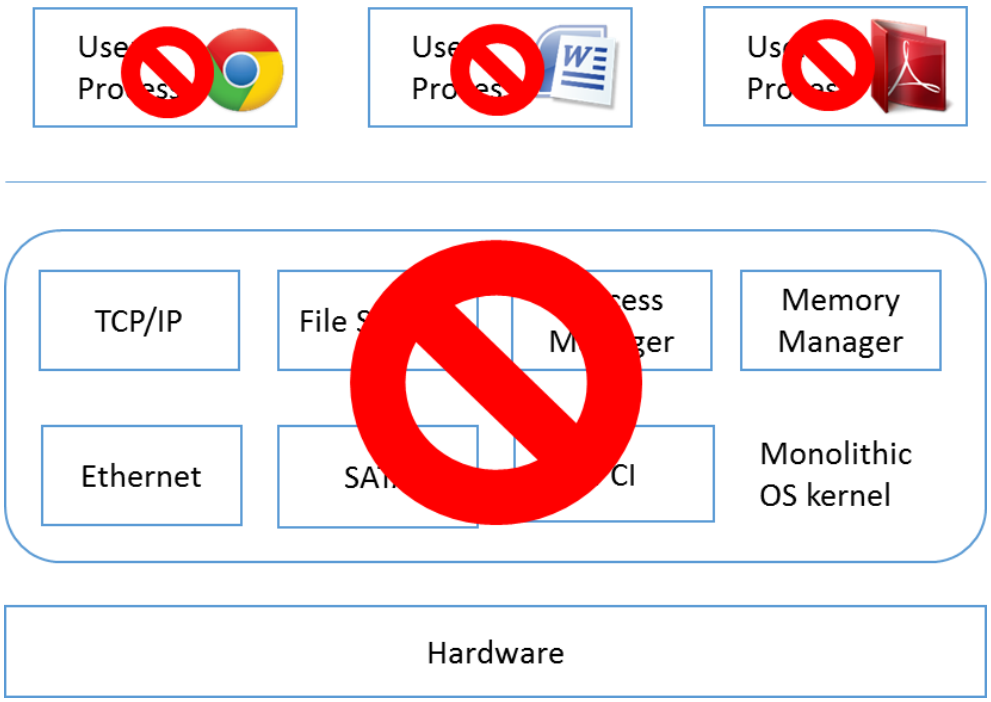

Figure 2.10: Kernel space

2.3 Virtualization

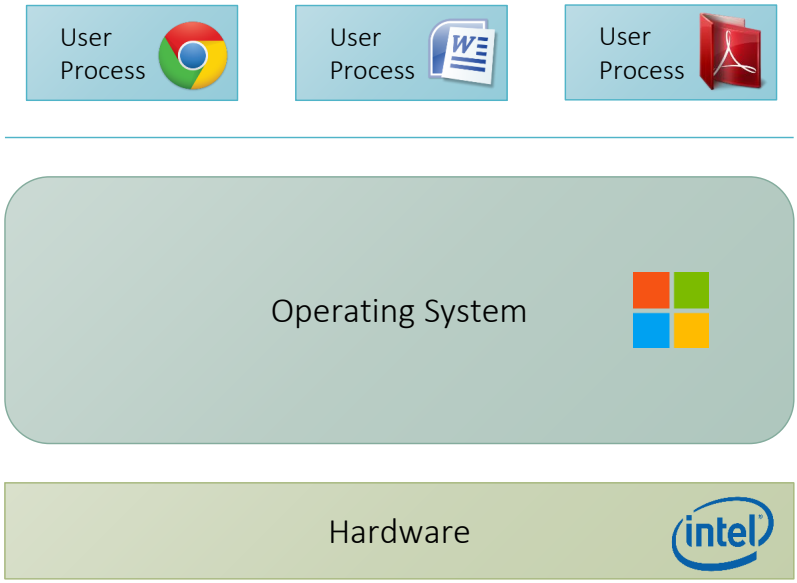

Figure 2.11: Operating System Architecture

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.[27] Originally introduced for VM/370 [13], the idea later emerged for modern platforms [9, 34].

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.[14]

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.[16]. It has been shown that virtualization can be utilized to provide better security and robustness for operating systems[17, 25, 33].

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[4]. 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 [6, 11], KVM[21, 24], VMware ESX[4],and VirtualBox[10].

There are two types of hypervisors [19]

Type 1 hypervisors are also called as native hypervisors or bare metal hypervisors. Type 1

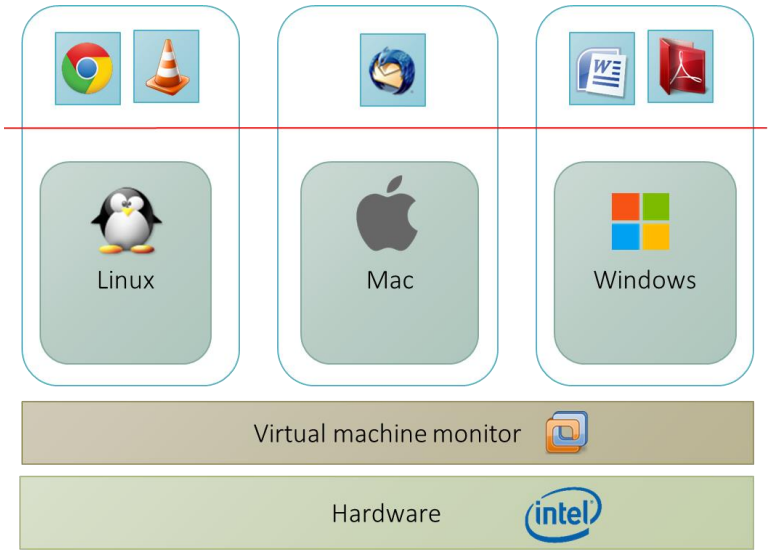

Figure 2.12: Virtualization

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[6], VMware ESX/ESXi[4] and Microsoft Hyper-V.

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[37, 10].

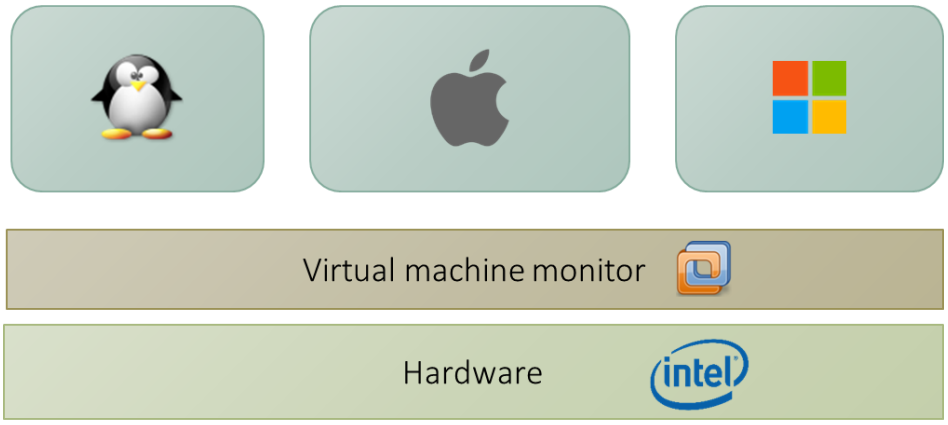

Figure 2.13: Type 1 hypervisor

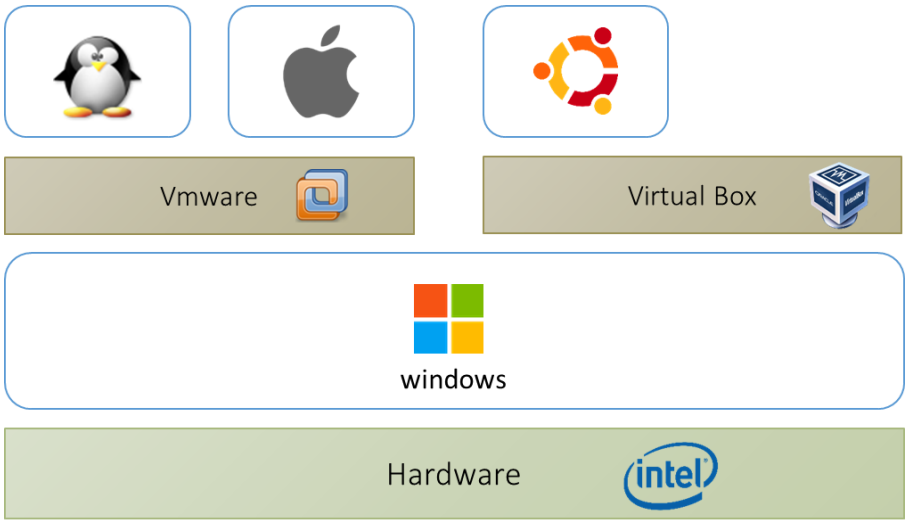

Figure 2.14: Type 2 hypervisor

2.3.2 Xen Hypervisor

Xen[6] is a widely known Type I hypervisor that allows execution of virtual machines in guest domains[23]. Figure 2.14 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 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 $Dom0$. $Dom0$ 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 $Dom0$.

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[11].

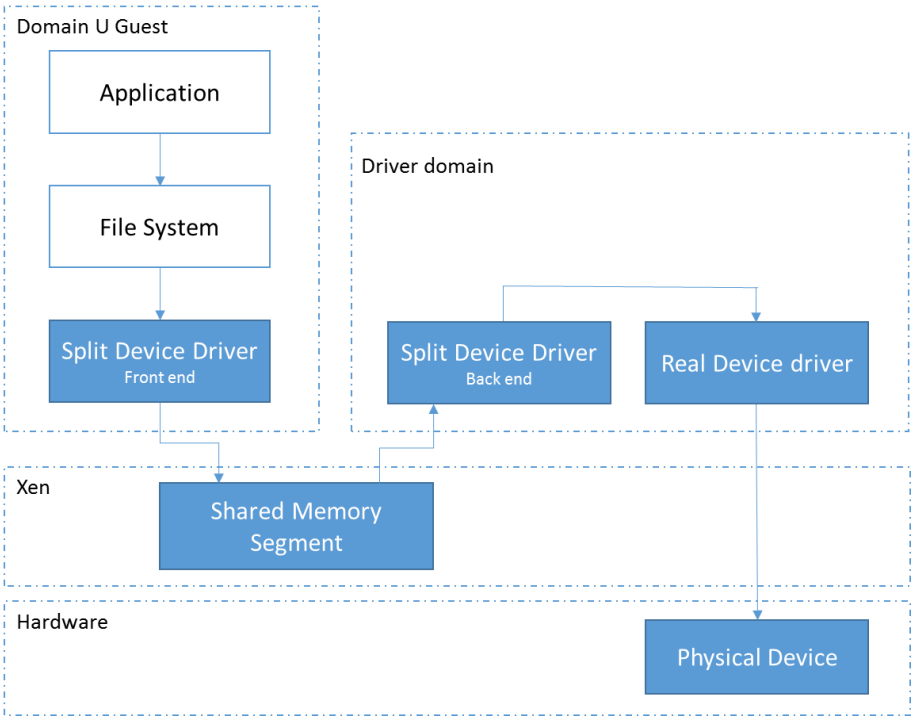

Figure 2.15: Xen split device driver

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.

In driver domain implementation, xen uses shared I/O ring buffers and event channel[6, 30,

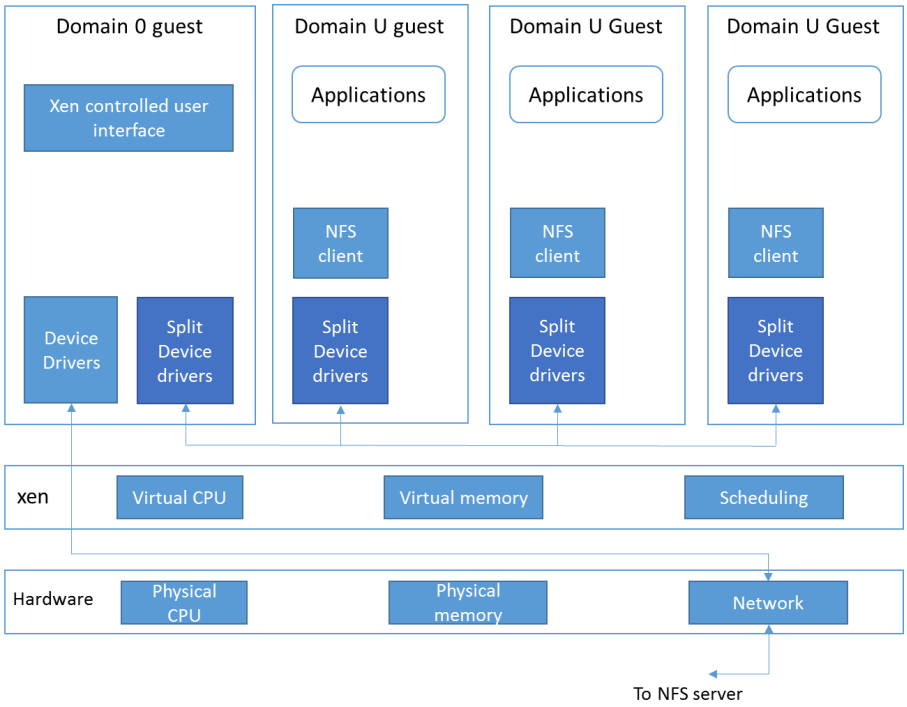

Figure 2.16: Xen

31].

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.

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.

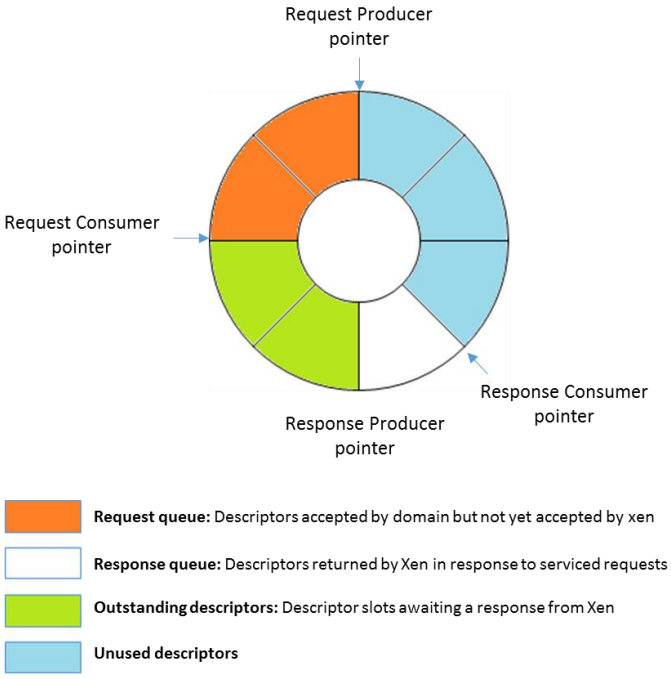

Figure 2.17: Ring I/O buffer

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

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.

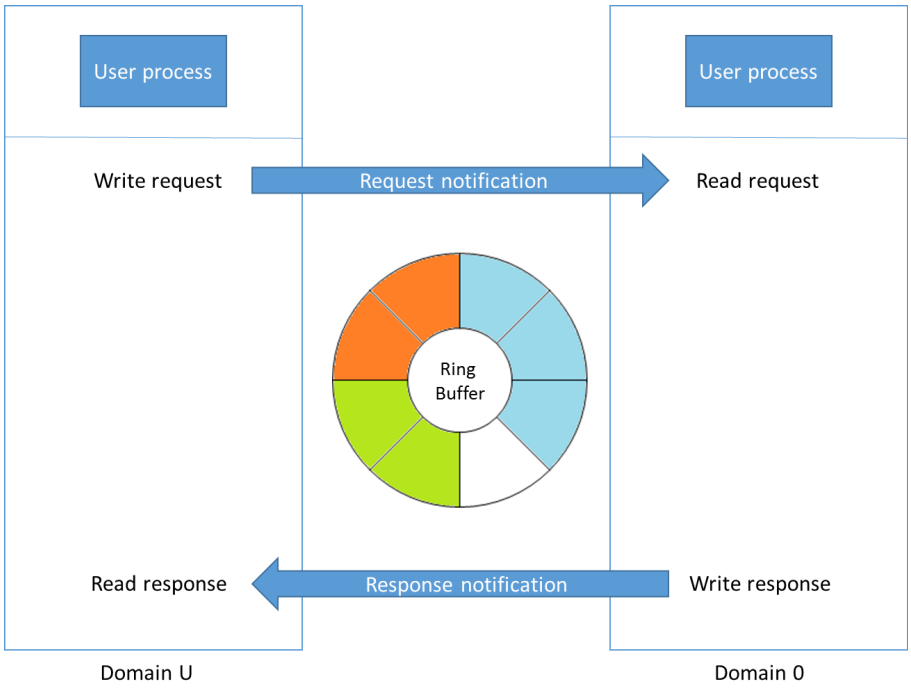

Figure 2.18: Ring I/O buffer

Chapter 3

System Introduction

3.1 Design Goals

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

In this section we will discuss each goal briefly.

3.1.1 Performance improvement

Xen driver domain system is the Xen hypervisor based solution to isolate the device driver from monolithic kernel. Even though Xen driver domain system provides better robustness for operating systems, it deteriorates the performance. The reasons for the performance deterioration can be due to the introduction of an extra layer of hypervisor, data copy overhead or overhead of communication between the domains. For data intensive operations such as read and write, Xen driver domain system transfers data between hardware and the driver domain domU. Also, the same data is transferred between driver domain domU and the application domain. The extra copy from the driver domain domU to the application domain explains the reasons for the performance degradation. The Xen driver domain system runs the device driver in a separate domU. The application domain and driver domain domU communicate with each other in order to send requests and get responses from the device driver. The communication between domains adds an overhead to the system performance. Our goal is to minimize the overhead during communication between the driver domain domU and the application domain.

3.1.2 Strong isolation

One of the main goals of the Xen driver domain system is to provide strong isolation. The Xen driver domain system implementation adds an extra layer of isolation in the design which provides fault isolation between kernel and the device driver. It also adds the ability

to manage individual components independently, thus, increasing the availability of the system. While exploring the opportunities to improve the performance of the Xen driver domain system, it is essential that the main goal of the Xen driver domain system is not compromised.

3.1.3 Compatibility and transparency

The extension of existing OS structure usually results in a large number of broken applications. In the past, Microkernels required effort to retain compatibility with the existing applications[22]. In order to provide compatibility with applications, either an emulation layer was implemented [22], or a multiserver operating system [18, 39] was built on top of a microkernel. Since the Xen driver domain system maintains compatibility between existing device drivers and applications, it is one of the basic goals of our solution to achieve the compatibility and transparency. Our solution makes sure that the performance improvement implementation will not require any changes in the device driver and the application to run the system.

3.2 System overview

The architectural overview of the system is presented in Figure 3.1 . An overview of the

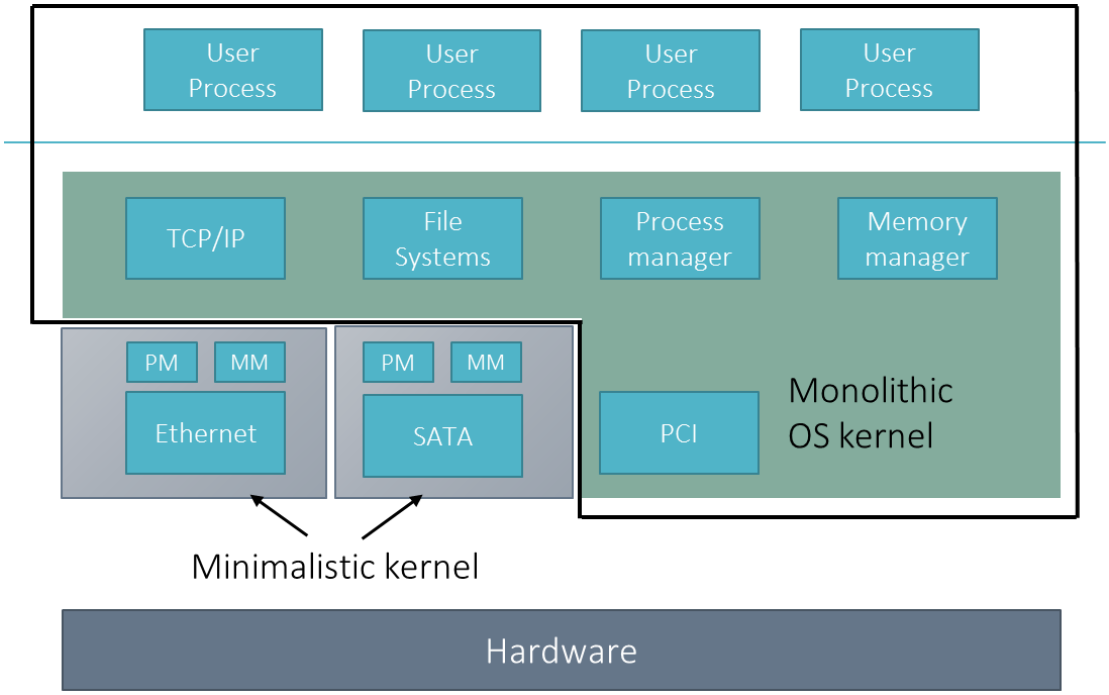

Figure 3.1: Overview

architecture shows that the Xen driver domain system partitions an existing kernel into multiple independent components. The user applications and Linux kernel run in a domain called as the *application domain*. The device driver which needs to be isolated from the kernel, executes in the separate domain called as the *driver domain*. Since in the Xen driver domain system implementation, a device driver cannot run standalone in a separate domU, hence the *driver domain* domU runs a minimalistic kernel with the device driver. The *driver domain* domU does not run any applications. The application domain is dedicated to

the core system tasks such as process management, scheduling, user memory management, and IPC. The sole task of driver domain domU is to handle requests coming from the user processes managed by the application domain.

3.3 System components

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

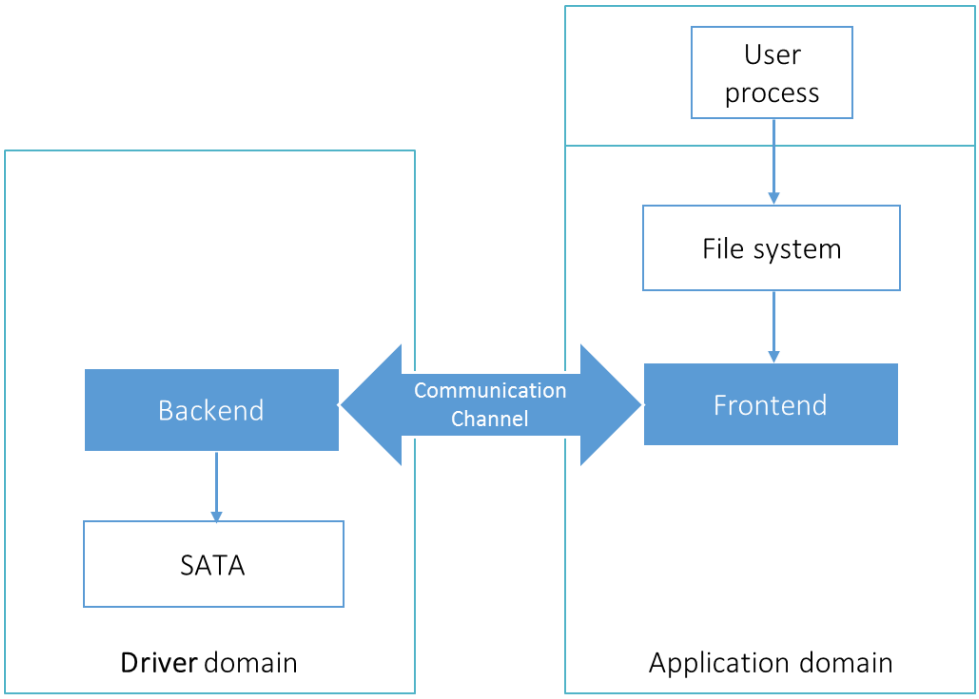

Figure 3.2: System Components

3.3.1 Front end driver

As mentioned earlier in section 3.1.3, transperency is one the design goals of the Xen driver domain system, which requires us to avoid any changes to the kernel as well as the device driver. In Xen driver domain system, the device driver runs in the driver domain domU and

the user applications run in the application domain. Hence, the user application will not know about the device driver execution in the driver domain domU. Thus, it is not possible for the application to send requests to the driver in the driver domain domU, without making any changes to the kernel. As a result, Xen driver domain system runs a piece of a code called *front end* in an application domain. The front end driver acts as a substitute for the device driver. The main functionality of the front end driver is to accept requests from the user application, process the requests, enqueue the requests for the driver domain domU and notify the driver domain domU. It also takes care of the processing of the responses and ending the corresponding requests.

3.3.2 Back end driver

The kernel and the device driver provide a functionality to accept requests from the user application running in the same domain. The kernel is not capable of accepting the requests from the application running in a separate domain without making any changes to the kernel code. 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. In order to avoid making any changes to the device driver and kernel, a piece of code called the *back end driver* runs in the driver domain domU. The responsibility of a back end driver is to accept requests from the application domain and forward them to the device driver. The back end driver sends the responses and notifies the application domain after receiving the responses from the device driver.

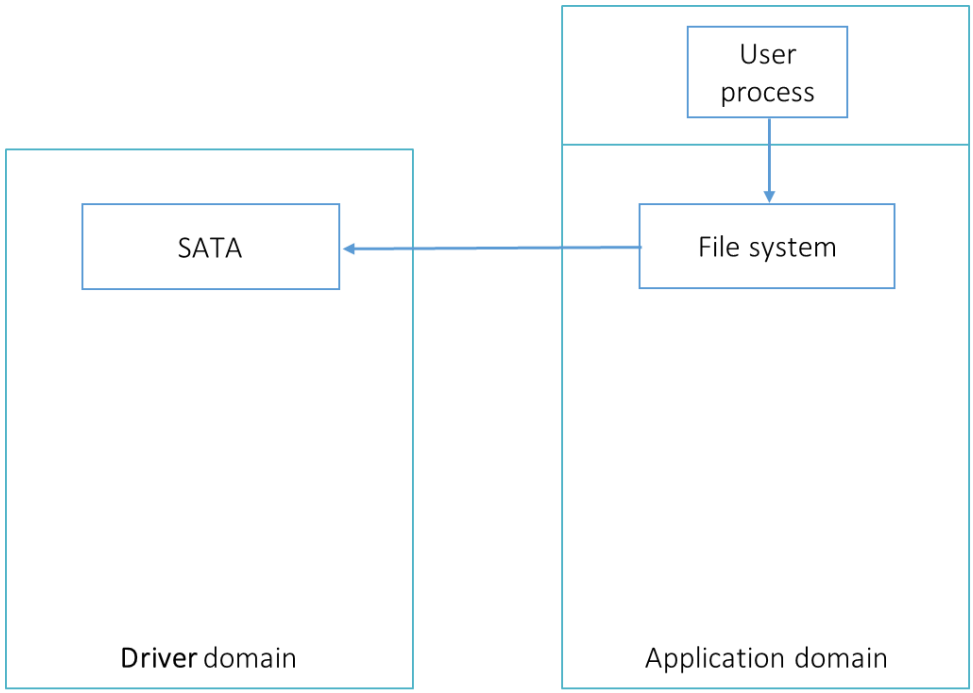

Figure 3.3: Conceptual design of driver domain

3.3.3 Communication module

The communication module is a communication channel between the front end driver and the back end driver. Unlike the back end and the front end driver, the communication module is not a separate physical entity or a kernel module. The communication channel exists in the front and the back end driver. It is logically divided into three parts. The responsibility of the first part is to forward the requests from the application domain to the driver domain domU as well as to forward the responses from the driver domain domU to the application domain. The responsibility of the second part is to share the read and write data and the responsibility of the third part is to notify the other domain upon the occurrence of

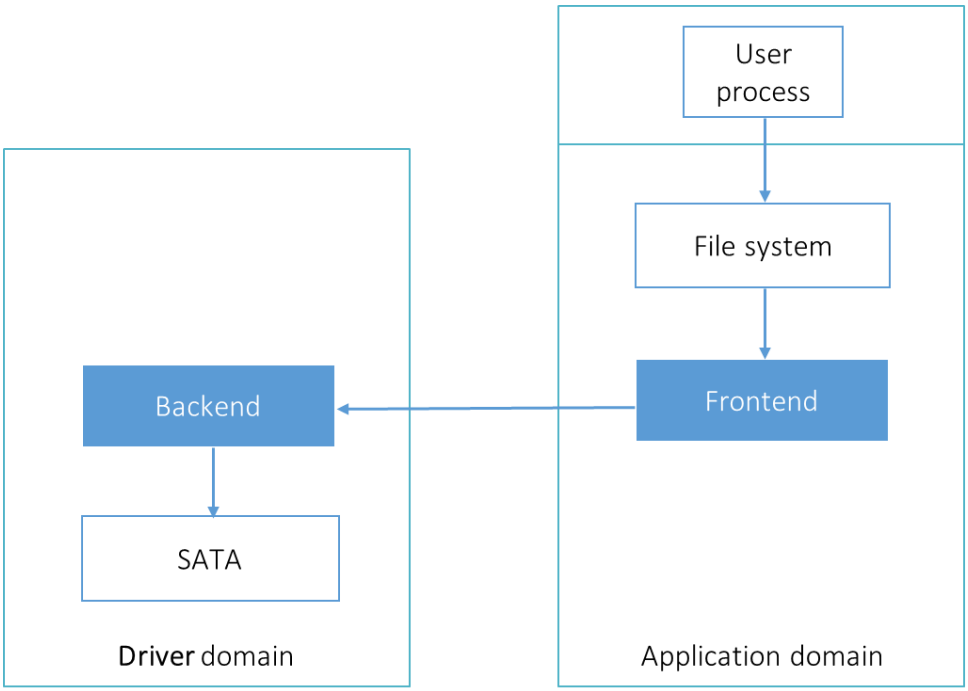

Figure 3.4: Back end and front end drivers

a particular event.

Figure 3.5 illustrates the role of the communication model.

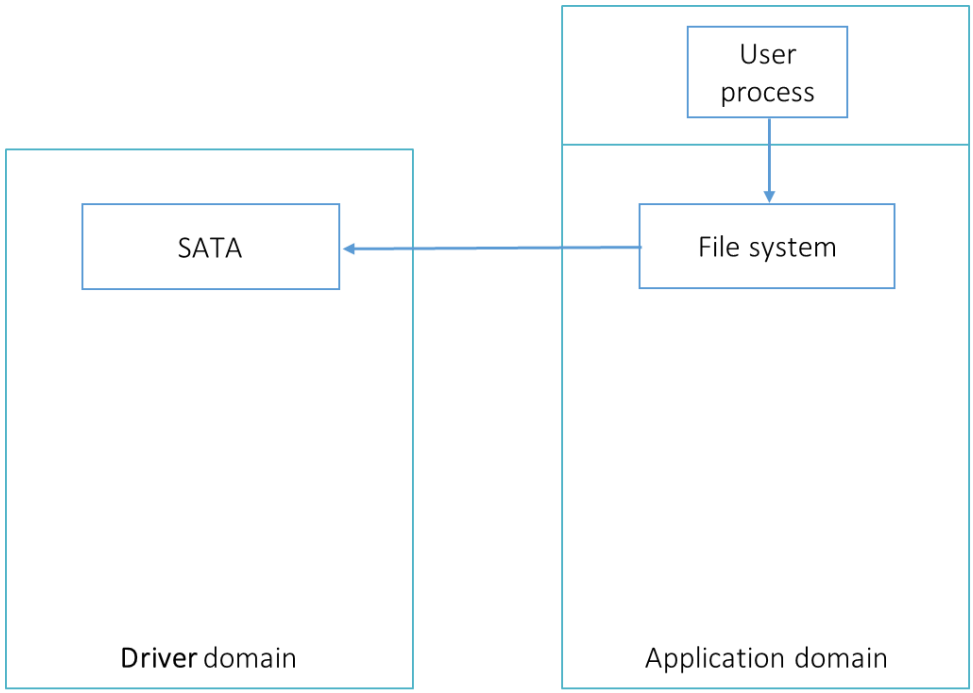

Figure 3.5: Communication module

3.4 System design

Figure 3.6 shows the architectural overview of the modern operating system with a monolithic kernel and Figure 3.7 illustrates the concept of the Xen driver domain.

The following section explains the design evolution of the Xen driver domain in brief. Although the solution to provide more protection at the kernel level is to isolate the device driver from the Linux kernel, it is not possible to run a standalone device driver. A device driver is dependent on the kernel components such as scheduler, memory management unit etc. Hence an instance of a minimalistic kernel runs with the device driver removing the

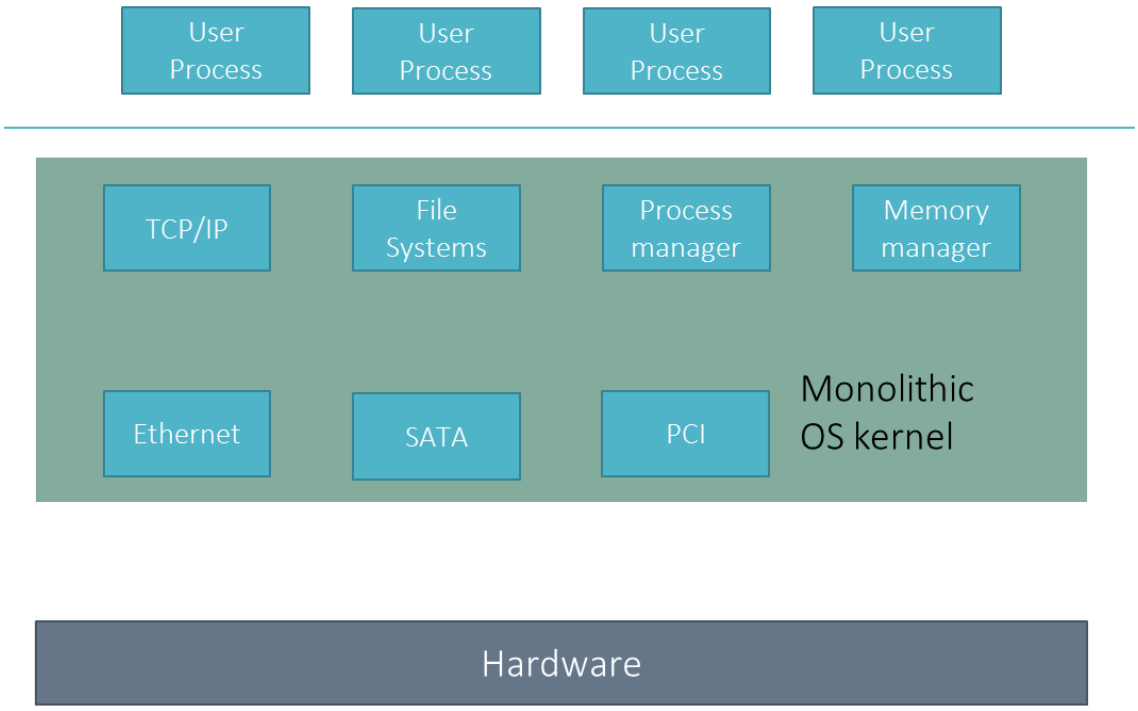

Figure 3.6: Tightly coupled System

dependency. Figure **??** represents the concept.

—-

It is not possible to run multiple monolithic kernels over the same hardware without any virtual machine monitor. Thus, virtualization is used for running multiple monolithic kernels on a virtual machine monitor. —

Figure 3.8 and Figure 3.9 explains the effects of a malicious activity occurring in the device driver isolated from the Linux kernel. When a device driver running in a driver domain domU hits a bug, it crashes the kernel of the driver domain and hence the driver domain itself. In addition, applications expecting a response from the driver domain might hang or

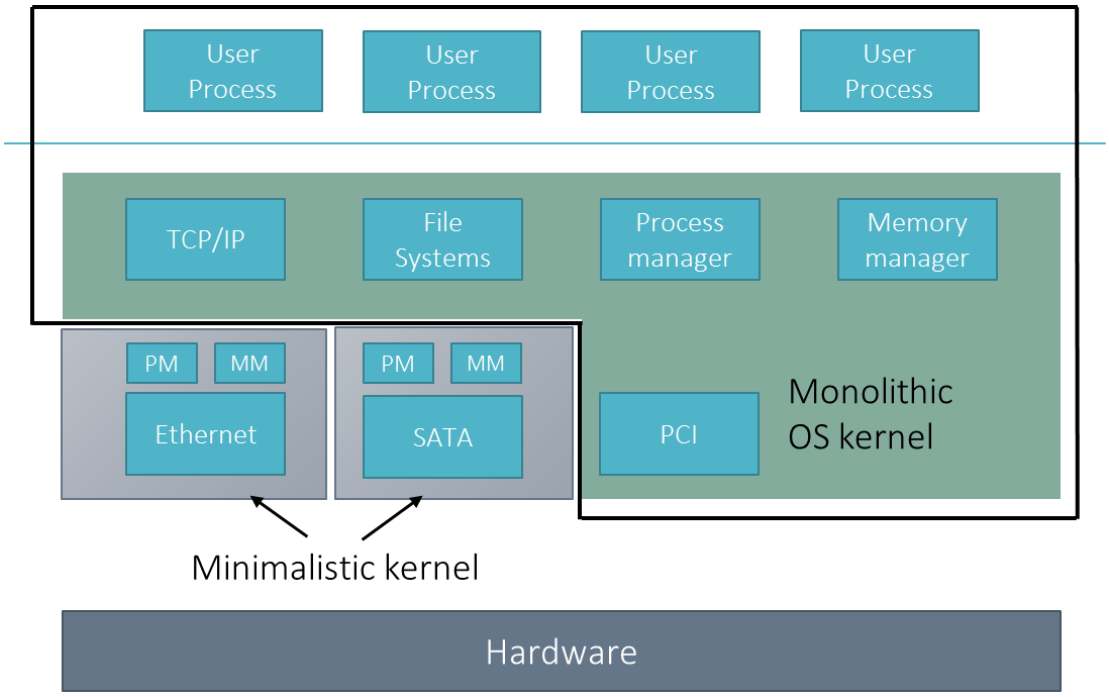

Figure 3.7: System with kernel and isolated device driver

crash waiting for the response. But due to the address space separation of the application domain and the driver domain, the application domain will remain intact.

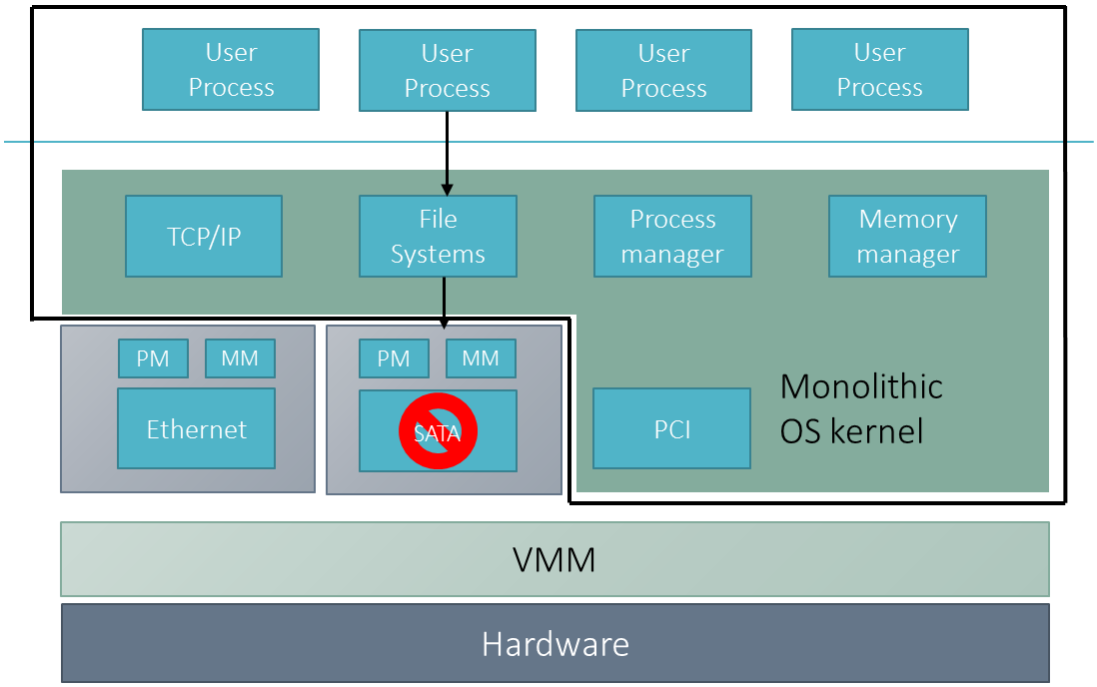

Figure 3.8: Device driver crash

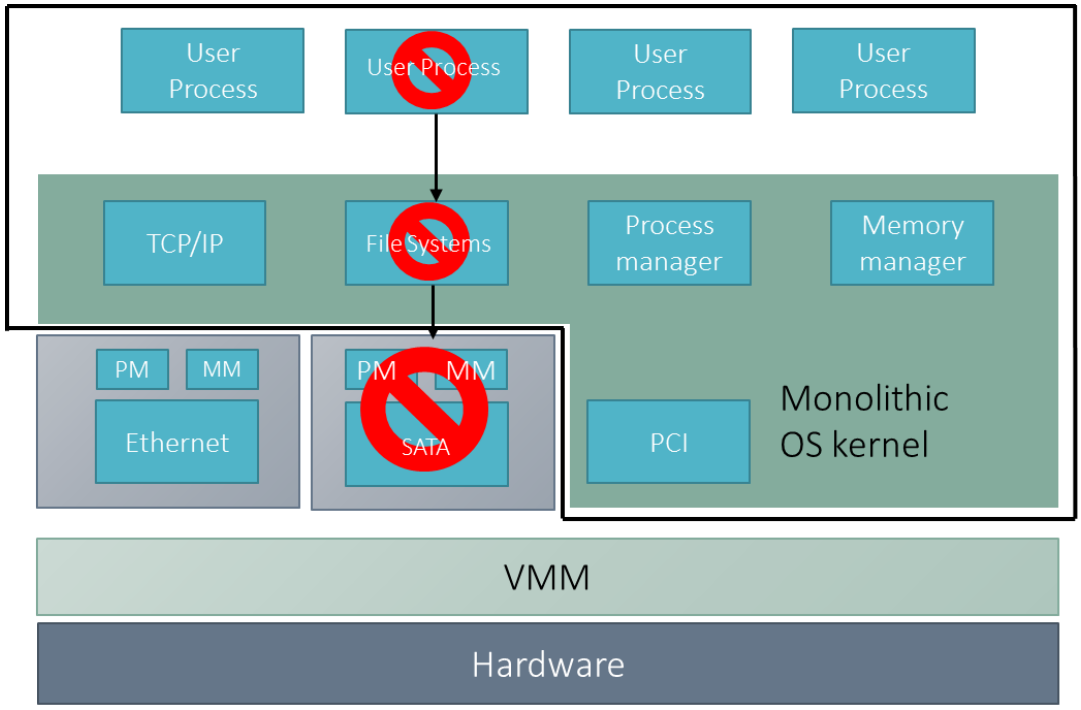

Figure 3.9: High Availability

Chapter 4

System Design and Implementation

This chapter describes the specific implementation details of the Xen driver domain system and its communication channel.

4.1 Implementation Overview

The Xen driver domain system is implemented with linux kernel 3.5.0 and Xen hypervisor 4.2.1. For the prototype, we implemented the Xen driver domain system with the block device driver. Both the application domain and the driver domain domU run the same linux kernel. The following table summarizes our implementation efforts of the Xen driver domain system.

Component	Number of Lines
Linux Kernel	7
Xen	250
Front-end Driver	647
Back-end Driver	752
Total	tt

As we observe from the table, the block device driver is unchanged, small number of changes were made to linux kernel and Xen hypervisor. However, we maintain the application compatibility.

4.2 Implementation

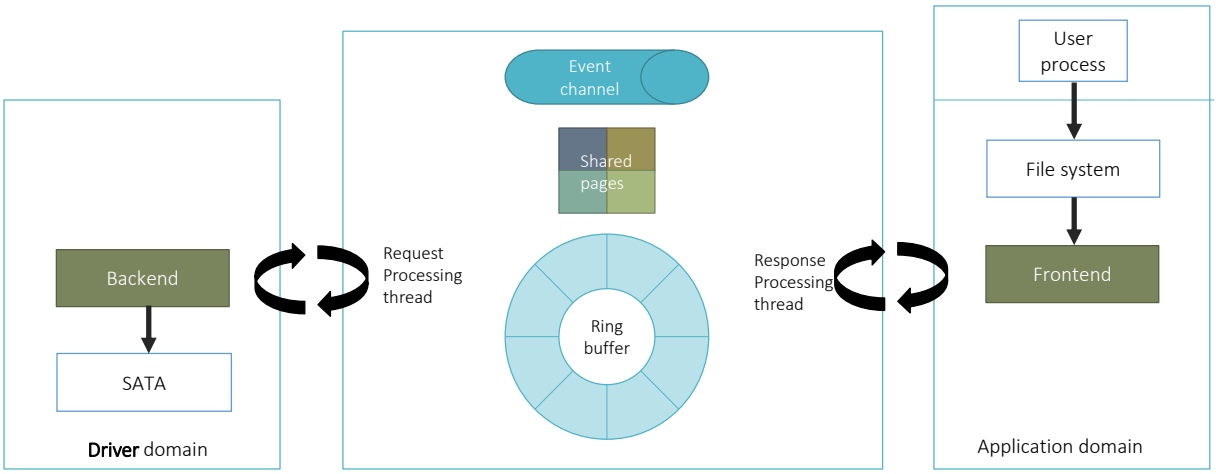

Figure 4.1: Implementation overview

4.2.1 Communication component

The most important component in the Xen driver domain system implementation is the communication component.

This section will describe the implementation details of the communication channel of xen driver domain system and the implementation details to improve performance of the communication channel.

1. In the original implementation of the communication channel, event channel is used for notifying driver domain domU and application domain when a request or response is available in the shared request and response queue.

2. In order to improve the performance of the system, we implement the communication channel in which the front end driver thread spins for the availablity of responses, and a dedicated thread in the back end driver spins for requests. In case of unavailablity of requests and responses, threads go to sleep. In this implementaion event channel is used only to wake these threads up from the remote domain.

The following subsections describe the implementation details of the communication channel with and without performance improvement measures.

Ring buffer: I/O rings

In the implementation of communication channel, for both approaches, ring buffer is used as a request and response queue. Ring buffer is a shared I/O ring explained in section 2.3.2. A ring buffer is divided into front ring and back ring. The front ring is used as a request queue and the back ring is used as a response queue. The front end driver recieves the request from an application, and converts the request into a format which can be understood by the back end driver. The front end device driver then checks for a free space in the request queue for a new request, and allocates the space for the new request using the function `RING_GET_REQUEST`.

```
ring_req = RING_GET_REQUEST(&info.main_ring, info.main_ring.req_prod_pvt);
if(ring_req == NULL){
 printk("NULL RING_GET_REQUEST\n");
 BUG_ON(1);
 return 1;
}
ring_req->seq_no = id;
ring_req->sector_number = blk_rq_pos(req);
ring_req->data_direction=write;
```

After batching sufficient requests together, the front end device driver pushes the requests to the front ring using the function `RING_PUSH_REQUESTS_AND_CHECK_NOTIFY`

```
static inline void flush_requests(idd_irq_info_t *flush_info)
{
 int notify;
 RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(&flush_info->main_ring, notify);
 notify_remote_via_irq(flush_info->ring_irq);
}
```

Shared pages

Since the ring buffer is not large enough to hold the read and write data of responses and requests, we use it only for sharing the requests and responses. In order to share an actual data we use shared pages.

Grant Table

Grant tables are a mechanism provided by the Xen hypervisor for sharing and transferring frames between the domains. It is an interface for granting foreign access to machine frames and sharing memory between underprivileged domains provided by the Xen hypervisor. In Xen, each domain has a respective grant table data structure, which is shared with the Xen hypervisor. The grant table data structure is used by Xen to verify the access permission other domains have on the page allocated by a domain[3].

Grant References

Grant references are the entries in the grant table. A grant reference entry has every detail about the shared page, which removes the dependency on the real machine address of the shared page. Since there exists a fully virtualized memory, the biggest difficulty in sharing the memory correctly between domains is knowing its correct machine address. Removing the dependency with the real machine address makes it possible to share the memory between domains.[11, 5, 3]

We use grant table so that the application domain grants the driver domain access to the the shared page, while retaining the ownershipn. The front end driver grants memory access to the back end driver, so that back end driver may read or write data into the shared memory as requested.

The steps we implement are:

1. The application domain creates a grant access reference, and shares the reference id (ref) to the block device driver domain by enqueueing a request in the request queue (front I/O ring) .

2. The block device driver domain reads the request and the reference id, and uses the reference to map the foreign access granted frame.

3. The block device driver domain performs the memory access.

4. The block device driver domain unmaps the granted frame.

5. The application domain removes its grant.

1. Granting foreign domain access

```
for_each_sg(info.sg, sg, ring_req->nr_segments, i) {
  buffer_mfn = pfn_to_mfn(page_to_pfn(sg_page(sg)));
  fsect = sg->offset >> KERNEL_SECTOR_SHIFT;
  lsect = fsect + (sg->length >> KERNEL_SECTOR_SHIFT) - 1;
```

```
  ref = gnttab_claim_grant_reference(&gref_head);
  BUG_ON(ref == -ENOSPC);

  gnttab_grant_foreign_access_ref(ref, info.domid,.
  buffer_mfn, rq_data_dir(req));

  info.shadow[id].frame[i] = mfn_to_pfn(buffer_mfn);
  ring_req->seg[i] = (struct idd_request_segment) {
      .gref       = ref,
      .first_sect = fsect,
      .last_sect  = lsect };
}
```

2. Mapping foreign frames

```
for (i = 0; i < nseg; i++) {
  flags = GNTMAP_host_map ;
  if (pending_req->operation != 0)
     flags |= GNTMAP_readonly;

  gnttab_set_map_op(&map[i], vaddr(pending_req, i),
  flags, req->seg[i].gref, DOMZERO);
}
```

```
ret = gnttab_map_refs(map, NULL, &backend.pending_page(pending_req, 0), nseg);
BUG_ON(ret);


for (i = 0; i < nseg; i++) {
  if (unlikely(map[i].status != 0)) {
    printk("invalid buffer -- could not remap it\n");
    map[i].handle = IDD_INVALID_HANDLE;
    ret |= 1;
  }


  pending_handle(pending_req, i) = map[i].handle;
  if(ret)
    continue;
seg[i].buf = map[i].dev_bus_addr |.
(req->seg[i].first_sect << KERNEL_SECTOR_SHIFT);
}
```

3. Unmapping foreign frames

```
for (i = 0; i < req->nr_pages; i++) {
handle = pending_handle(req, i);
  if(handle == IDD_INVALID_HANDLE)
```

```
        continue;
      gnttab_set_unmap_op(&unmap[invcount], vaddr(req, i),
          GNTMAP_host_map, handle);
      pending_handle(req, i) = IDD_INVALID_HANDLE;
      pages[invcount] = virt_to_page(vaddr(req, i));
      invcount++;
    }
    ret = gnttab_unmap_refs(unmap, pages, invcount, 0);
```

4. End foreign access

```
for (i = 0; i < s->req.nr_segments; i++) {
gnttab_end_foreign_access(s->req.seg[i].gref, 0, 0UL);
}
```

`gnttab_end_foreign_access` does not revoke access; it only prevents further mappings. Since `gnttab_end_foreign_access` does not revoke access it is used after the block device driver has unmapped the frame[11, 5].

Event Channel

Event channel is a mechanism provided by xen hypervisor for event notification. Xen Driver domain implementation use event channel to send notifications between domains that request and response is available in the request and response queue.

However, in the implementation to improve the performance of the communication channel and hence the system we spin threads to share request and responses between domains and event channel is used only to wake up the both threads.

Event channel : Hypercall interface

`EVTCHNOP_alloc_unbound` is a hypercall to allocate a new event channel port. Allocated event channel port can be connected by remote domain if,

1. Specied domain exist
2. A free port exist in the specified domain.

Less privileged domains can allocate only their own ports, privileged domains can also allocate ports in other domains[11, 5]. `bind_evtchn_to_irqhandler` is used for assigning an interrupt handler for a notification. In driver domain implementation, back end driver allocates an event channel while initializing, and assigns an interrupt handler.

```
ring_alloc.dom = DOMID_SELF;
ring_alloc.remote_dom = DOMZERO;
smp_mb();
err = HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound, &ring_alloc);
if (unlikely(err != 0))
  goto end2;
```

Event channel: Other interfaces

`bind_interdomain_evtchn_to_irqhandler` is used for connecting to existing event channel as well as assigning an interrupt handler for handling a notification. In driver domain implementation back end driver allocates the event channel, and binds the interrupt handler to allocated event channel using `bind_interdomain_evtchn_to_irqhandler`.

```
err = bind_evtchn_to_irqhandler(ring_alloc.port, irq_ring_interrupt,
  0, "syscall_backend_irq_ring", &backend);
```

In driver domain implementation, front end driver connects to the event channel allocated by backend, using interface `bind_interdomain_evtchn_to_irqhandler`.

```
err = bind_interdomain_evtchn_to_irqhandler(data.domid, data.ring_port,
    irq_ring_interrupt, 0, "ring_irq", NULL);
if (unlikely(err < 0)) {
printk(KERN_WARNING "Cannot bind (main) event channel\n");
goto end2;
}
```

4.2.2 Application domain

Application domain is the domain running user applications. In the monolithic linux kernel, usually an user process sends the read write request to file system, which sends the read and

write request to the block device driver. The block device driver serves the request and send back a response to the file system, which further sends the response to the user process.
In Xen driver domain implementation block device runs separately in a driver domain. When user process sends a request to the file system, the file system needs to forward the request to storage domain. Like explained in the section 3.3.1, in the implementation of xen driver domain, a piece of code is introduced which forwards the request to the domU running device drivers.

Front end driver

The piece of code, which forwards the request to the domU running device driver is called as a Front end driver. The core responsibility of the front end driver is:

1. To provide an interface which appears as a block device to upper layer in the stack.
2. Accept a request from the upper layer.
3. Create a new request which can be understood by storage domain.
4. Enqueue new request into request queue.

Implementation details of front end driver is split into 4 stages.

1. Initialization
2. Create request

3. Enqueue request

4. Dequeue response

Initialization

During the initialization process the front end driver creates an interface for all the block devices. After creating the interface for all block devices, front end driver creates a kernel thread `read_response_thread`

```
info.response_thread = kthread_run(idd_read_response, &info, "read_response_thread");
```

The core functionality of the `read_response_thread` is to dequeue the responses available in the ring buffer. However, there might be a case when no request is available in the ring buffer, but the request is expected to be present in the future. In such cases, `read_response_thread` thread spins for the responses.

The `read_response_thread` thread goes into sleep state after spinning for some time- threshold.

Obviously, a thread shouldn't sleep unless it is assured that somebody else, somewhere, will wake it up. The code doing the waking up job must also be able to identify the thread to be able to do its job. We use a data structure called a wait queue to find the sleeping thread. A wait queue is a list of threads, all waiting for a specific event[**?**, 8].

In Linux kernel like all other lists, a wait queue is managed by a wait queue head, of a data type `wait_queue_head_t`, and is defined in $< linux/wait.h >$. A wait queue head is defined

and initialized statically as follows: `DECLARE_WAIT_QUEUE_HEAD(name);` and dynamicly as follows: `wait_queue_head_t my_queue; init_waitqueue_head(&my_queue);`

```
info.response_thread = kthread_run(idd_read_response, &info, "read_response_thread");
init_waitqueue_head(&info.wq);
info.waiting_rsps=1;
```

Create request

Front end driver dequeues the request submitted to the driver interface by an user process or the file system, and then converts the request into a request of structure type `idd_request_t`. The structure is as below:

```
struct idd_request {
  int data_direction;
  uint8_t nr_segments;
  uint64_t sector_number;
  struct idd_request_segment {
      grant_ref_t gref;
      uint8_t first_sect, last_sect;
  }seg[IDD_MAX_SEGMENTS_PER_REQUEST];
  uint64_t seq_no;
}__attribute__((__packed__));
```

Member of the structure are explained below. `data_direction` : Flag to tag if request is read or write. `nr_segments` : Number of `idd_request_segments`. `gref` : Grant reference grant table entry. `first_sect` and `last_sect` : first and last sector in frame to transfer. `seq_no` : To track if any request and response is lost.

Enqueue request

Like explained in section 4.2.1, `RING_PUSH_REQUESTS_AND_CHECK_NOTIFY` is used for flushing request to the ring buffer, we use the same API to enqueue the request to the request queue.

```
RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(&flush_info->main_ring, notify);
```

However, if the `read_request_thread` running in back end driver which accepts the requests is sleeping, then waking up the `read_request_thread` is more important task of front end driver. Wake up signal is sent to back end driver using an event channel. Before sending the notification, we check the state of the thread.

```
status = atomic_read(&(flush_info->main_ring.sring->req_status));
smp_mb();

if(status == SLEEPING){
  notify_remote_via_irq(flush_info->ring_irq);
```

Status of the thread is saved in the shared memory. We use the atomic variables to save the state of threads to avoid race conditions.

Dequeue response

Response is dequeued from the ring buffer by the `read_response_thread`. To dequeue the response from response queue, we use the ring buffer API `RING_GET_RESPONSE`. However, the important part in this stage it managing the `read_response_thread` thread.

When the response is not available `read_response_thread` spins for some time, and then goes to sleep. When the response is made available by the backend then depending upon the state of the thread, event channel interrupt is sent by the backend. If `read_response_thread` is in sleeping state then the backend driver will send an interrupt and the front end driver will wake up the thread, otherwise, no action is taken as thread is already spinning for the response.

In interrupt handler shared atomic variable status is read by the front end driver and depending upon state action is taken.

```
status = atomic_read(&(info.main_ring.sring->rsp_status));
if(status == SLEEPING){
  wake_up(&info.wq);
info.waiting_rsps = 1;
}
atomic_set(&(info.main_ring.sring->rsp_status), RUNNING);
```

```
smp_wmb();
```

`read_response_thread` sleeps on the wait queue `info.wq` waiting for `info.waiting_rsps` to be set.

```
wait_event_interruptible(
  info.wq,
  info.waiting_rsps || kthread_should_stop());
```

Response is read using ring buffer API `RING_GET_RESPONSE`.

```
rp = info.main_ring.sring->rsp_prod;
```

```
for (i = info.main_ring.rsp_cons; i != rp; i++) {
  unsigned long id;
  sleep_cond = 0;
  ring_rsp = RING_GET_RESPONSE(&info.main_ring, i);
  id  = ring_rsp->seq_no;
  if (id >= IDD_RING_SIZE)
    continue;
```

We also maintain an shadow table of all requests which are ended upon reading respective successful response.

```
  req  = info.shadow[id].request;
  if(req!=NULL)
    idd_completion(&info.shadow[id], ring_rsp);
  if (add_id_to_freelist(&info, id)) {
    WARN(1, "response to %s (id %ld) couldn't be recycled!\n",
      op_name(ring_rsp->op), id);
    continue;
  }
  error = (ring_rsp->res == 0) ? 0 : -EIO;
  __blk_end_request_all(req, 0);
}
```

Upon reading the responses, thread spins again for more responses and then goes to sleep. We mark the state of the thread as `SLEEPING` and then check for the request queue to avoid race condition. If a request is present in the request queue then the request is served while state of the thread is still `SLEEPING`.

```
status = atomic_read(&(info.main_ring.sring->rsp_status));
if(sleep_cond > THRESHOLD && status==RUNNING){
  atomic_set(&(info.main_ring.sring->rsp_status), SLEEPING);
  info.waiting_rsps = 0;
  sleep_cond = 0;
```

```
  RING_FINAL_CHECK_FOR_RESPONSES(&info.main_ring, more_to_do);
  if (more_to_do)
    goto again;
}
```

4.2.3 Driver domain

Driver domain is the domU running a device driver. In our implementation, driver domain runs block device driver. Usually in monolithic linux kernel an user process sends the read write request to a file system, which sends the read and write request to block device driver. Block device driver serves the request and responses back to file system, which further sends response to user process.

However, in driver domain implementation, block device runs separately in a driver domain. Like explained in a section 3.3.2, a peice of code called as a back end driver runs in a driver domain which accepts a request from application domain and forward the request to the device driver. Upon receiving the reponse from the device driver, back end driver sends back the response, and notifies the application domain.

Back end driver

Back end driver is a kernel module and component of independent block device driver which runs in the storage domain. The core responsibility of back end end driver is :

1. Dequeue resquest from request queue.
2. Convert to BIO request which can be understood by block device driver .
3. Accept response from block device driver.
4. Enqueue response into response queue.

Implementation details of back end driver can be split into 5 stages.

1. Initialization
2. Dequeue request
3. Create BIO.
4. Make response.
5. Enqueue response

Initialization

During initialization process Back end driver creates a kernel thread `read_request_thread`.

```
backend.request_thread = kthread_run(idd_request_schedule, &backend, "read_request_threa
```

The core functionality of the `read_request_thread` is to dequeue the requests available in the request queue. If request is not available in the request queue then thread waits on a wait queue. Similar to front end, back end driver initializes wait queue in initiazation process.

Dequeue request

Request is dequeued from the request queue by the `read_resquest_thread`. To dequeue a request, we use the ring buffer API `RING_GET_RESPONSE`. When the request queue is empty, `read_resquest_thread` spins for some time to checks if new requests are queued, after reaching threshold it goes to sleep. When the request is enqueued by front end driver then depending upon the state of the `read_resquest_thread`, an event channel interrupt is sent by the front end driver. If `read_resquest_thread` is in a sleeping state then front end driver will send an interrupt. In intrupt handler, back end driver will wake up the `read_resquest_thread`. If `read_resquest_thread` is already running then no action is taken. In interrupt handler, status is read from shared atomic variable by the back end driver.

```
status = atomic_read(&(backend.main_ring.sring->req_status));
if(status == SLEEPING){
wake_up(&backend.wq);
backend.waiting_reqs = 1;
}
atomic_set(&(backend.main_ring.sring->req_status), RUNNING);
```

`read_resquest_thread` sleeps on the wait queue `backend.wq` waiting for `backend.waiting_reqs` to be set.

```
wait_event_interruptible(
be->wq,
be->waiting_reqs || kthread_should_stop());
```

Request is read using ring buffer API `RING_GET_REQUESTS`.

```
rc = be->main_ring.req_cons;
rp = be->main_ring.sring->req_prod;
rmb();
while (rc != rp) {
  if (RING_REQUEST_CONS_OVERFLOW(&be->main_ring, rc))
    break;
  memcpy(&req, RING_GET_REQUEST(&be->main_ring, rc), sizeof(req));
  be->main_ring.req_cons = ++rc;
```

Upon reading requests, thread spins again for more responses. If a threshold is reached then the thread goes to sleep. We mark the state of the thread as `SLEEPING` and then check for the request queue one more time to avoid a race condition. If request is present in request queue then that request is served while state of the thread is still `SLEEPING`.

```
status = atomic_read(&(backend.main_ring.sring->req_status));
if(sleep_cond > THRESHOLD && status==RUNNING){
  atomic_set(&(backend.main_ring.sring->req_status), SLEEPING);
```

```
  be->waiting_reqs = 0;
  sleep_cond=0;
  RING_FINAL_CHECK_FOR_REQUESTS(&be->main_ring, more_to_do);
  if (more_to_do)
    goto again;
```

Create BIO

Whenever the request thread receives a request to serve, the request thread creates the `bio` request for the corresponding request. `bio` structure is a basic container for block I/O within a kernel. `bio` structure is defined in $< linux/blk_types.h >$. `bio` structure represents active block I/O operations as a list of segments, and a segment is a chunk of buffers. The `bio` structure provides the capability for the kernel to perform block I/O operations of even a single buffer from multiple locations in memory. Vector I/O such as this is called scatter-gather I/O.

Following is the struct bio:

```
struct bio {
  sector_t    bi_sector;  /* device address in 512 byte sectors */
  struct bio    *bi_next; /* request queue link */
  struct block_device *bi_bdev; /* associated block device */
  unsigned long   bi_flags; /* status, command, etc */
  unsigned long   bi_rw;    /* bottom bits READ/WRITE, top bits priority */
```

```
  unsigned short  bi_vcnt;  /* how many bio_vec's */
  unsigned short  bi_idx;   /* current index into bvl_vec */
  unsigned int    bi_phys_segments; /* Number of segments in this BIO after physical add
  unsigned int    bi_size;  /* residual I/O count */
  unsigned int    bi_seg_front_size;
  unsigned int    bi_seg_back_size;
  unsigned int    bi_max_vecs;  /* max bvl_vecs we can hold */
  atomic_t    bi_cnt;   /* pin count */
  struct bio_vec    *bi_io_vec; /* the actual vec list */
  bio_end_io_t    *bi_end_io;  void      *bi_private;
  bio_destructor_t  *bi_destructor; /* destructor */
  struct bio_vec    bi_inline_vecs[0];
};
```

A request queued into the request queue by the front end driver is in format which is understood by back end driver, but we can not forward the same request to block device. We convert the dequeued request into `bio` request, so that the block device understands the request. To make the `bio` request, we need the associated block device structure `struct block_device`. We get associated block device structure using function `blkdev_get_by_path`.

```
backend.bdev = blkdev_get_by_path("/dev/ramd",
  FMODE_READ | FMODE_WRITE | FMODE_LSEEK |
```

```
  FMODE_PREAD | FMODE_PWRITE, NULL);


backend.dev = MKDEV(MAJOR(backend.bdev->bd_inode->i_rdev),
  MINOR(backend.bdev->bd_inode->i_rdev));
```

Pages from shared memory are mapped and inserted into the `bio` structure using function `bio_add_page`. And other variables are copied into the `bio` structure from the dequeued request.

```
for (i = 0; i < nseg; i++) {
  while ((bio == NULL) || bio_add_page(bio,
  be->pending_page(pending_req, i),
  seg[i].nsec << 9,.
  seg[i].buf & ~PAGE_MASK) == 0) {
    bio = bio_alloc(GFP_KERNEL, nseg - i );
    if (unlikely(bio == NULL))
      goto fail_put_bio;
    biolist[nbio++] = bio;
    bio->bi_bdev = breq.bdev;
    bio->bi_private = pending_req;
    bio->bi_end_io = end_block_io_op;
    bio->bi_sector  = breq.sector_number;
```

```
  }
  breq.sector_number += seg[i].nsec;
}
```

At the end, the newly created `bio` request is sent to the lower layer for execution.

```
for (i = 0; i < nbio; i++){
  submit_bio(op, biolist[i]);
}
```

`bi_end_io` function pointer is a pointer to a callback function. Once bio request is completed, function pointed by `bi_end_io` gets called.

```
bio->bi_end_io = end_block_io_op;
```

Make response and Enqueue

Irrespective of the success or failure of the execution of `bio` request, the back end driver makes a response, which could be understood by the frontend. Like explained in subsection 4.2.3, `bi_end_io` function pointer is a pointer to a callback function. Once bio request is completed, function pointed by `bi_end_io` gets called. We create a new response in this callback function.

In this callback function we complete the `bio` request with function `bio_put`. After that the result gets copied into a newly allocated response structure. The response is enqueued to re-

sponse queue using `RING_GET_RESPONSE` and `RING_PUSH_RESPONSES_AND_CHECK_NOTIFY`. Once the response is enqueued, depending upon the status of the remote thread-`read_response_thread`, a inturrupt signal is sent to the application domain.

```
resp.op = op;
resp.seq_no = id;
resp.priv_data = NULL;
resp.res = st;
spin_lock_irqsave(&be->blk_ring_lock, flags);
memcpy(RING_GET_RESPONSE(&be->main_ring, backend.main_ring.rsp_prod_pvt),&resp, sizeof(r
be->main_ring.rsp_prod_pvt++;
RING_PUSH_RESPONSES_AND_CHECK_NOTIFY(&be->main_ring, notify);
status = atomic_read(&(be->main_ring.sring->rsp_status));
smp_mb();
if(status == SLEEPING){
  notify_remote_via_irq(be->ring_irq);
```

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] hypercall. `http://xenbits.xen.org/docs/4.2-testing/misc/grant-tables.txt`.

[4] 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.

[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. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

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

[17] 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.

[18] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. In *In 9th SIGOPS European Workshop*, pages 109–114, 2000.

[19] 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.

[20] 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.

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

[22] Gernot Heiser and Volkmar Uhlig. Are virtualmachine monitors microkernels done right. *Operat. Syst. Rev*, 40:2006, 2006.

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

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

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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.

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