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

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.

Independent Device Driver (IDD) is an infrastructure which exploits the virtualization to isolate the device drivers from the kernel. In IDD, 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 independent device driver domain for Block device driver. Our prototype demonstrate that a block device driver can be run in a separate domain.

We implement the IDD with two different approaches and compare both the results. In first approach, we implement the Independent Device Driver with 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 services it offers and its ability to function reliably. Even though reliability of operating system has been studied for several decades, it remains a major concern even today. The characteristics of operating systems which makes them unstable and insecure are size and complexity. If we consider Linux kernel; it has over 15 million lines of code. One of the software reliability study shows that a code contains 6 to 16 bugs per 1000 lines of executable code[1][8]. Some other studies even say 2 to 75 bugs per 1000 lines of code [6]. Using a minimum estimate of 6 bugs per 1000 lines, the Linux kernel has around 90,000 bugs. Also one of the study shows that the device drivers have error rate 3 to 7 times higher than ordinary code[2]. So considering the fact that 70% of the operating system consists of device drivers, our calculation of 90000 bugs can be an understatement[2]. To make a system reliable, finding all these bugs and fixing them is definitely not a feasible option considering the fact that bug fixes introduces new lines of code and hence new bugs.

1.1 Problem Statement

Large size of system makes it impossible for one person to understand the code. At the same time complexity and tightly coupled device driver and linux kernel makes it difficult to isolate a fault in a device driver. The difficulty in isolating faults in a device driver makes system unreliable and less robust, thus, the problem we try to solve here is of tightly coupled operating system.

Monolithic kernel components doesnt have the kind of isolation user level applications has. In a monolithic kernel, one of the millions lines of the kernel can overwrite kernel data structure used by an unrelated component and crash the system. This threat can be reduced by executing each device driver in an isolated environment from kernel. However, a device driver is dependent upon many kernel components like memory management, scheduler etc., hence it is difficult to isolate device driver from the kernel and execute it separately.

1.2 Proposed Solution

Unlike user applications, modern operating system kernel contains couple of hundred or thousand procedures linked together as a single program[8]. Any one of the millions lines of the kernel can overwrite/corrupt kernel data structure.

In modern operating systems ”Memory protection” is a way to control memory access rights. Memory protection prevents a process from accessing memory that has not been allocated to it, preventing a line of code within a process affecting other processes, or the operating system[3][7]. Unlike user applications, kernel has hundreds of procedure linked together, which makes it difficult to prevent an access to kernel data structure with memory protection.

The idea is to run a special program called a hypervisor. Hypervisor is capable of running multiple operating systems at the same time called a virtual machine[4]. Hypervisor is commonly used to allow different operating systems such as Linux, Oracle, and Windows, to run at the same time, or to exploit the hardware. The use of virtual machines has a well-deserved reputation for extremely good fault isolation. Since none of the virtual machines even know about the other ones, malfunctioning of one virtual machine cannot spread to the other[5].

We propose an infrastructure which exploits the virtualization to isolate such linked procedures. In a virtualized environment, each virtual machine has an allocated memory, and because of memory protection mechanism, one virtual machine cannot affect the memory of any other virtual machine.

Thus, by exploiting this property we run a kernel and device driver in a separate virtual machine, whereas user application and a kernel runs in a same virtual machine, making it impossible to corrupt kernel data structure by a device driver running in another virtual machine. The isolation of a device driver and a kernel can be achieved by running the device driver in a separate virtual machine other than the virtual machine running the user application. Our solution here adapts the concept, protection of faults in an operating system[5].

1.3 Core Contributions

Technical core contributions of this project can be divided into two parts.

1. Device driver isolation implementation

2. Performance comparison of spinning and event channel.

These contributions are listed below.

1.3.1 Device driver isolation

An approach based on virtualization to decouple device driver from linux kernel, and partition existing kernel into application domain, and isolated device driver domain. The application domain would serve system calls related to core linux kernel functionality, whereas the isolated device driver would serve system calls related to corresponding device driver.

1.3.2 Performance comparison

We implement request and response availability notification with mechanism provided by Xen, which follows interrupt based model. We implement the same part with spinning of threads to avoid context switch and compare the performance of isolated device driver.
We aim to find if there is any performance degradation in system because of context switch.

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

2.1 Memory protection

2.1.1 User space

2.1.2 kernel space

2.2 Virtualization

2.2.1 Hypervisor

2.2.2 Xen Hypervisor

2.2.2.1 Hypercalls and events

2.2.2.2 Data Transfer: I/O Rings

2.3 Processes and threads

2.3.1 Processes

2.3.2 Threads

Chapter 3

System Introduction

3.1 Design Goals

3.2 System overview

3.3 System components

3.3.1 Front end module

3.3.2 Back end Module

3.3.3 Communication module

3.4 System design

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

4.2.2.1.6 Dequeue response

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] Galvin P. Silberschatz, A. and G Gagne. *Operating System Concepts.* Wiley, 2009.

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