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[3][12]. 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[4]. Considering the fact that the operating system predominantly consists of device drivers, the faults in device drivers make the operating system unreliable[4].

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 these 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[5][9]. 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.

1.2 Proposed Solution

The reliability of a system can be improved by executing device drivers in an isolated environment from the kernel. The dependency of device drivers on other kernel components such as memory management, scheduler etc. make it difficult to isolate device drivers from the kernel.

Our solution here adapts the concept of 'virtualization based fault protection in operating systems' proposed by LeVasseur et. al. [7], Tanenbaum et. al.[12], Nooks[11], Soltesz et. al. [10]. The solution we implement runs a special program called hypervisor. Hypervisor is commonly used to run multiple operating systems in parallel, by exploiting the hardware. 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.

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, we run a device driver with minimalistic kernel in one virtual machine, and run user applications and a kernel together in the other virtual machine. 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.

1.3 Core Contributions

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

1. The implementation of the device driver isolation concept using two approaches.
2. The performance comparison of the approaches.

The first approach implements device driver isolation using interrupt based communication between front end and back end device drivers. Xen uses the same approach for implementation of the driver domain[1]. This interrupt based model requires context switches[2]. The second approach implements the communication between front end and back end device drivers using spinning of threads. The spinning based model does not require the context switches. This report presents the performance comparison of both the approaches. In addition, the report aims to find if the performance degradation can be attributed to the context switches.

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

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] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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