

Performance Improvement in Xen

Mahesh Palekar -mpalekar@cc.gatech.edu

Motivation:

Xen is a virtual machine monitor for x86 architecture that supports execution of multiple guest operating systems. There is always an overhead associated with virtualization. The performance of Linux is better than its counterpart on Xen (Xeno-Linux).

Xen runs in privilege ring 0, guest OS runs in privilege ring 1 and applications run in ring 3. A guest OS cannot be run on top of Xen directly in ring 0 due to security isolation between guest OS and Xen. Modifications have been made to linux kernel to para-virtualize it in order to run on Xen, here-after termed as Xeno-Linux. In such a system there is system call overhead caused by protection domain switching. The cost of protection domain switching is more severe in 64-bit architectures, where a domain switch between application and OS involves address-space switch. This happens since 64-bit x86 architecture has only 2 rings of protection (3 and 0), and hence application and OS kernel must both run in ring 3. In this work we only focus on 32-bit x86 architecture. System call overhead affects applications that make a lot of system calls the most like I/O applications. Eliminating system call overhead should improve performance of these applications.

In our proposed solution guest OS and applications will run in the same protection domain. This will be done by making segment descriptors of user program point to kernel descriptors. The system calls library would be changed to do a local procedure call rather than generating a software exception. This is possible because the guest OS and application are running in the same protection domain, i.e they have same address space as well as segments.

In a trusted environment like a server running trusted applications, application vs application and application vs kernel protection is not needed and hence application can also run in kernel mode.

Motivating Example: In-Vehicle Systems

In-vehicle systems run 3 major kinds of applications:

1. Collision avoidance application: These are safety related applications. These applications increase the perception of the driver beyond his field of vision and assist the driver with autonomous assistance. These applications have hard real time guarantees, need high reliability, availability.
2. Traffic Management Applications: These applications help to ease the task of driving by providing traffic and related information like weather etc. to the driver. An example application could be calculating an optimum route to the destination on the current traffic data. These applications have relatively soft real time guarantees and will have less priority than collision avoidance applications.
3. Infotainment: These applications include providing Internet, Gaming, Mass Marketing etc. These applications have least priority and have different requirements depending on the specific applications.

As all these types of applications have different requirements, hence each category may need specialized OS tuned to its requirements. None of the applications of one category should be able to affect applications of other category i.e isolation between categories is required. An unreliable infotainment application should not lead to whole system crash affecting the safety related applications and thus affecting the collision avoidance applications which help to avoid accidents. These requirements are strong enough to assume that in vehicle systems will use virtualization/VMM. Collision avoidance applications have to satisfy hard real time guarantees. Major part of collision avoidance applications is sending data on the network. In Xen, in order to send packets on the network, there are 2 control transfers associated with transfer of data between application and guest OS. One way to improve performance of these applications is to remove the system call overhead between applications and guest OS. Thus avoiding two control transfers per send or receive of data from network. By removing the protection between applications and guest OS, security is compromised. But as collision avoidance applications are the most important ones, they are assumed to be trusted.

Related Work:

In order to provide better performance by minimizing the kernel user boundary crossing, SPIN [5] provides features for extending the kernel. Programmers write their kernel extensions in Modula -3 programming language and insert them into the kernel. SPIN kernel is also protected from malicious kernel extensions by using a type safe language – Modula 3 for writing extensions. But a huge amount of effort will be needed to convert the current systems into SPIN architecture. We also do not care about security as we assume that the applications are trusted. On the other hand, only small changes need to be made to the current kernel in order to remove the Kernel-User boundary crossing. There are other works like Exokernel [4] etc, but they also need a lot of changes to the current system to be implemented.

Kernel Mode Linux [1] is very similar to our work. But a kernel mode linux does not make much sense. Because a malicious application may crash the whole system. In case of Xen, a malicious application can only crash the OS it is running on. Rest of the system remains unaffected. No one else has tried developing kernel mode Xeno-Linux.

Solution

User programs usually run in User mode i.e privilege level 3. Guest OS runs in privilege level 1. Our approach consists of running the user applications in privilege level 1 and thus eliminating the user-kernel boundary crossing.

IA-32 Architecture:

IA-32 architecture provides support for segmentation. It also provides virtual memory mechanism. Linux exploits the segmented flat memory model. IA-32 has 4 privilege levels 0-3. In traditional Operating Systems, Kernel runs in privilege level 0 while applications run in privilege level 3. In Xen, the hypervisor runs in privilege level 0, guest OS (XenoLinux) runs in privilege level 1 and applications run in level 3.

A privilege level of a running program is determined by the segment which is pointed by the CS segment register. In IA-32 segment is defined in a segment descriptor. Segment descriptor provides the base address of a segment, access rights, types and usage information. Each segment descriptor

has a segment selector associated with it. The lower order two bits of the selector specify the privilege level of the selector or segment.

Proposed Solution :

The solution can be divided into two parts

- **Method to execute user programs in kernel mode**

Make user selectors point to kernel descriptors. This needs changes to be made to segment.h file. This leads to stack starvation problem discussed below. The segment limit for user applications is 0-3 GB while kernel can access 0-4 GB. In order to make system calls as light as procedure calls, user should be able to access 3-4GB address space also. This can be done by making changes to the uaccess.h file

- **Make system calls in Kernel Mode.**

In Linux system call must be invoked by executing the int 0x80 assembly language instructions which raises the programmed exception that has vector 128. The system call number is passed in the eax register. Application mostly does not call the system call directly, it calls it through the libc library wrapper routines. Each call in the libc library is generally a syscallX() macro where X is the number of parameters. So in order to make a system call from the kernel mode, syscall0 macros will be changed to call the system call service routine through the syscall table rather than using int 0x80.

Problems:

- **Stack Starvation[1]:**

This problem happens because of interrupt handling mechanism of Linux. If an interrupt occurs in kernel mode, IA-32 CPU pushes CS segment, EIP register, EFLAGS onto the stack. If an interrupt occurs in the user mode, an IA-32 CPU switches a stack from the user mode to kernel mode. CPU pushes CS, EIP and EFLAGS, SS and ESP registers onto the kernel stack. When a user program is executing in kernel mode accesses its stack not mapped by MMU, page fault occurs and IA-32 CPU tries to interrupt the running program and jumps to page fault handler. CPU can't do this as the program executes in the kernel mode and there is no stack to save execution state. In order to notify this error, it generates a double fault, but as there is no stack to save the current state of the system, it resets. Stack starvation problem can be solved by using task management facility of the IA-32 architecture. In order to handle interrupt with task, it must be set in the Interrupt Descriptor Table.

References:

1. Maeda Toshiyuki, Safe Execution of User programs in Kernel Mode using Typed Assembly Language, Masters Thesis.
2. Saraiya Purav, Performance Improvement in Xen. Project Report.
3. Paul Barham et.al.Xen and the Art of Virtualization. SOSP 2003.
4. Dawson R. Engler, Frans Kaashoek and James O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management ", Proceedings of the 15th ACM Symposium on Operating System Principles, ACM, December 1995.
5. Brian Bershad et al., " Extensibility, Safety and Performance in the SPIN Operating System ", Proceedings of the 15th ACM Symposium on Operating System Principles, December 1995.

6. Intel Corporation. IA-32 Intel Architecture Software Developer's Manual.
<http://developer.intel.com/>

7. Bovet, Daniel P., and Marco Cesati. Understanding the LINUX KERNEL. 1st ed. O'Reilly, 2003.