7th International Conference on Communication, Computing and Virtualization 2016

A NOVEL FRAMEWORK FOR HYPERVISOR DESIGN

M. Ambika Research Scholar a, Dr. Kolasani Ramchand H Rao b

a Research Scholar, Dept of Computer Science, Acharya Nagarjuna University, Guntur, India -522006

b Professor, Department of Computer Science, ASN Degree College Tenali,

Abstract

The paper discusses the constructive framework for writing hypervisor on the top of the VM. Hypervisor plays a vital role in monitoring VM in cloud. Hypervisors are system programs that virtualizes the running of the architecture. These are critical, safe, small and hard program to write and debug or evaluate. Hypervisors are system software programs that virtualize the architecture they run on. They are typically small, safety-critical, and hard to debug, which makes them a feasible and interesting target for formal verification. Previous functional verifications of system software were all based on interactive theorem proving, requiring substantial human effort complemented by expert proven knowledge. In this paper we present the framework of a hypervisor and basic implementation module of hypervisor and simulator module of hypervisor.

The virtual environment is controlled and managed by software known as the hypervisor. The hypervisor executes at a higher privilege level than the “guest” operating systems; the hypervisor can access any memory assigned to any guest operating system. Furthermore, a guest operating system cannot access resources in the system unless permitted to do so by the hypervisor.

Key words - hypervisor, VMM, OS and Guest

Introduction

In current information technology (IT) parlance, virtualization refers to the ability to partition the hardware

Email id : ambica.mangalagiri@gmail.com
resources of a system such that each partition appears to be a complete hardware platform that can execute an operating system and applications independently from the operating system and applications executing concurrently within other such partitions. Virtualization is becoming an increasingly popular feature implemented in the IT infrastructure of industrial, commercial, and academic enterprises because of a number of benefits including cost reduction, application security, and flexibility. The potential for cost reduction is the main reason driving the popularity of virtualization today. By consolidating servers dedicated to several different applications onto a single physical computer supporting multiple virtual servers, enterprises can reduce the number of physical computers that must be acquired with concomitant savings in power supply, cooling, space, and maintenance costs. The energy savings also translates into goodwill that an enterprise can generate by appearing to be “green” (i.e., by being environmentally friendly). The ability to execute an application in its own dedicated virtualized processor and operating system environment has the potential for improving the security of application. Because each application runs in isolation, a single compromised application or operating system environment does not necessarily mean that all the applications executing on other virtual environments are also compromised.

The flexibility of an enterprise’s IT resources can be improved through virtualization because each virtual platform can execute a different operating system. This enables, for example, a software development team to test its software on a variety of operating systems (and versions) without having to setup the environments on separate physical machines or having to install and reinstall the operating system on a single computer. By utilizing virtualization, the team can test the different versions of their software on demand. The virtual environment is controlled and managed by software known as the hypervisor. The hypervisor executes at a higher privilege level than the “guest” operating systems; the hypervisor can access any memory assigned to any guest operating system. Furthermore, a guest operating system cannot access resources in the system (e.g., memory) unless permitted to do so by the hypervisor.

**Background**

The Intel 64 architecture provides virtualization support through extensions, referred to as VMX, to the instruction set and microprocessor functionality. These extensions enable the creating of a low-level layer of software, referred to as the virtual machine monitor (VMM) or hypervisor [1]. The hypervisor is responsible for partitioning the physical resources of the system into separate virtual machines, each capable of executing an independent operating system. Note, Intel also implements I/O virtualization technology (VT-d) that is only briefly addressed is discussed in this paper [2]. A number of open-source research projects exploring the implementation of hypervisors exist. One of the earliest is the Blue-Pill project by Joanna Rutkowska [3]. The blue pill project is primarily a vehicle to demonstrate how the processor’s virtualization features can be exploited to create virtually undetectable VMM-based malware. The Blue-Pill project implements a thin hypervisor that minimizes the number of resources virtualized. We have used a similar design philosophy in our course in order to reduce the amount of code necessary to explore the issues with virtualization technology in a classroom environment.

The Xen Hypervisor is an open-source community supported virtualization implementation that is intended
to provide robust virtualization capabilities to end users [4]. Because of its completeness, Xen is a relatively heavy-weight implementation of virtualization. Although an excellent tool for implementing virtualization in realistic environments, we found the source code for Xen to be unsuitable for classroom activity. Students, however, are encouraged to browse the source codes for Xen and Blue-Pill as examples of how to implement specific virtualization functionality. VMware is well-known commercial vendor of virtualization technology. Although their source code is not openly available, many of our students utilize their software and are familiar with it. Furthermore, VMware releases numerous technical papers that describe issues and solutions that arise in hypervisor design (e.g., performance tradeoffs [5]). From the perspective of the physical processor, the hypervisor needs to perform the following two main functions (a) save and restore the state of the virtual processor – this includes the user-visible general purpose registers as well as the system registers (e.g., mode control registers, memory management unit’s registers and the interrupt controller’s registers). Essentially, whenever a guest operating system accesses these registers or executes any instruction that updates the system’s state or reports on important system state, the processor, instead of executing the instruction, invokes the hypervisor. The hypervisor emulates the execution of the instruction, storing the results in appropriate general purpose registers, if required (i.e., any changes to the general purpose registers expected from the normal execution of the instruction are reported back to the guest environment in the guest’s general purpose registers). The hypervisor then resumes execution of the guest at the point after the instruction that caused the hypervisor to be invoked (see figure 1). Volume 3, System Programming Guide in the Intel® 64 and IA-32 Architectures Software Developer’s Manual series [6] describes the processor’s VMX technology in detail. Volume 2: Instruction Set Reference [7].

Proposed Work

Hypervisors virtualized all of the computing resources and provide abstractions to the guest OS, they can theoretically view any aspect of the guest OS’s state. as shown in Table I, different types of hypervisors virtualized different logical resources; consequently, they have different views.

Native hypervisors directly push guest code to run on the hardware. As such, they cannot continuously monitor the execution of the guest OS (unless they use single-step execution). Instead, they intercept control from interrupts or other hardware events. They therefore have a snapshot view of the VM state, which is often acquired when certain hardware events occur and allow the hypervisor to regain control. Hypervisors of this type can observe the following: Since

—CPU registers: All of the CPU registers can be read by the hypervisor when it gains control because it runs at the highest privilege level.
—Guest OS memory: The entire guest OS memory state can also be observed. However, hypervisors only have access to physical addresses, which have to be translated to virtual addresses while accessing each specific memory cell.
—Hard disk contents: Similar to the memory image, the content of the guest OS’s disk image, if not encrypted, is also visible to the hypervisor.

—Hardware events: All hardware-level events, including timers, interrupt, and exceptions, can also be observed.

—I/O traffic: The hypervisor also oversees all I/O traffic; including network traffic, disk I/O, and keystrokes.

**FRAME WORK OF HYPERVISOR**

This section discusses how the device driver is used for inserting the hypervisor layer below the operating system, how the hypervisor is structured in order to enable students to do a majority of the code in C, and how the hypervisor interacts with the guest operating system.

**The Device Driver**

A full discussion on writing device drivers is beyond the scope of this paper; we describe the salient points relevant to the delivery of the hypervisor and how it can be used to help with debugging and monitoring the hypervisor during development. Device drivers based on the Windows Driver Model (WDM) architecture are characterized by six functions DriverEntry(),

DriverUnload(), DispatchCreate(), DispatchClose(), DispatchWrite(), and DispatchRead(); and one data structure, the DeviceExtension. The DeviceExtension structure holds data that uniquely describes the state of this
instance of the driver and establishes the context of the driver instance. The DeviceExtension structure contains fields mandated by the WDM architecture and other fields unique to the driver. We added fields containing pointers to important hypervisor data structures and debugging areas to the DeviceExtension for our driver. These data structures are described in more detail below.

The `CreateDevice()` function is called once during the lifetime of the device driver when it is first loaded by the system; it can be thought of as the “main()” function of the driver. This function requests Windows to setup the driver’s instance and to allocate space for the DeviceExtension. It also sets up the pointers to the other five characteristic functions in the DeviceExtension so they can be called by the operating systems when needed (the WDM architecture uses an event-driven programming model for device drivers). The function also establishes a symbolic name that user mode applications can use to connect to the device driver and informs Windows that the device driver will not use buffered I/O (i.e., the driver will directly access the user-space memory in the application when the application makes read/write requests to the driver).

The `DriverUnload()` function is called when the driver is exited; it frees any memory allocated by the driver that has not already been released back to the operating system and deletes the symbolic link.

The `DispatchCreate()` function is called when an application opens a connection to the driver via the `CreateFile()` Windows function. In `DispatchCreate()`, we allocate a page (naturally aligned 4-Kbytes) of memory for each of the following regions required by the processor to support virtualization:

(a) The VMXON region

(b) The virtual machine control structure (VMCS), and (c) the MSR bitmap. The function also allocates two 64-Kbyte areas of physically contiguous memory, one for the hypervisor’s stack and another to hold general purpose debugging information to assist in the hypervisor’s development.

The `DispatchClose()` function is called when the application closes its connection to the device driver. This function ensures that the hypervisor has been terminated properly and frees all hypervisor-related memory regions (e.g., VMXON region, VMCS, stack, and debugging).

The `DispatchRead()` and `DispatchWrite()` functions are called when the user application calls `FileRead()` and `FileWrite()` Windows functions, respectively. Other than the direction of the data transfer expected by Windows, these functions are essentially identical. Because the driver can directly accessing the user’s buffers, the driver can read and write from/to the buffer in both functions. Therefore, we can program a user-level application to write “commands” and their parameters into the I/O buffer. The device driver, in the `DispatchRead()` and `DispatchWrite()` functions, can interpret the commands, execute the actions requested, and write the results into the buffer. When the `FileRead()` and `FileWrite()` completes, the application can read and display the results reported back in the buffer. This arrangement where the device driver acts as the application’s kernel-mode surrogate enables the user application to perform privileged activities, such as installing and testing the hypervisor. We use only the `DispatchRead()` function to control the device driver in this course. However, students are free to use either function as they see fit. The user application requests the installation and removal of the hypervisor using this mechanism.
The Hypervisor

When the DispatchRead() function receives the command to install the hypervisor, it initializes the VMXON region and the VMCS. Initialization of the VMXON region is straightforward; it only involves clearing the memory and writing the VMX version information loaded from a machine-specific register (MSR) at a specific location in the VMXON region. Next the managing of the computer such as memory management (including paging and virtual memory), I/O, and interrupt handling. Note that the operating system is unaware of the existence of the hypervisor and the hypervisor is invoked asynchronously by the processor in response to certain events. Therefore, the hypervisor’s code and data is stored in locked (i.e., immovable, always resident) areas of memory.

The VMCS contains the following three broad types of fields: (a) control, (b) guest state, and (c) host (i.e., hypervisor’s) state. The control fields are used by the hypervisor to define the events that cause the physical processor to stop the execution of the virtual machine and return control to the hypervisor. These fields also provide information to the hypervisor as to the reason why the hypervisor is being invoked (e.g., which instruction caused the virtual machine to exit, the instruction’s length, and which registers are affected). This information can be used by the hypervisor to emulate the execution of the instruction/event and resume the execution of the virtual machine.

The guest state fields hold the contents of the control, descriptor, and segment registers that are loaded into the virtual processor when the guest code begins or resumes execution. The hypervisor initializes these fields to setup a new virtual machine and the physical processor stores the contents of the virtual processor’s control registers back into these fields when the virtual machine execution stops and control is returned to the hypervisor. The hypervisor can then modify these registers as needed in order to emulate the execution of the instruction or event that cause the hypervisor to be invoked before resuming the execution of the virtual machine. Interestingly, the general-purpose, floating point, and MMX and XMM registers are not saved automatically and must be saved and restored by the hypervisor if they are modified in the hypervisor. This is the same behavior required by interrupt handlers. Note that the hypervisor may need to modify the content of some general purpose registers in order to reflect the result of emulating an instruction/event back to the virtual machine. Windows 7 appeared to be a significant challenge until we realized that we could simply reflect the current control register context into the guest state fields in the VMCS; our device driver is running in kernel-mode as part of Windows 7 when the DispatchRead() function is called. This means that the current content of the control registers are correct for continued execution of the operating system within a virtual machine. Therefore, we query and write the contents of the various control registers to the corresponding host state fields in the VMCS.

The host state fields contain the processor context that should be loaded into the control and segment registers when the hypervisor is executed. For our initial implementation of a thin hypervisor, we chose to setup the control register context to be identical to that of Windows 7. This kept us from having to setup a separate paged memory infrastructure; saving us considerable virtual memory design and programming effort. Figure 3 depicts the organization of the hypervisor code and its control flow at a high level.

1. Hypervisor setup begins when the device driver’s DispatchRead() function receives the command to install the hypervisor.
2. The DispatchRead() function calls several utility functions written in assembly language to enable VMX operations and to initialize the VMXON region and the VMCS.

3. The guest and host instruction pointer fields in the VMCS are filled with the addresses of the instructions following the VMGuest and VMHost assembly language labels, respectively.

4. These labels are exported by NASM and the linker sets up the correct values when all the component object files are linked together into the driver’s executable file.

5. After the VMCS is setup, the DispatchRead function calls the VM_Launch() function written in assembly language.

6. VM_Launch writes the current stack pointer value into the guest’s stack pointer field in the VMCS and then issues the VMLAUNCH instruction.

7. If the VMLUANCH succeeds, the virtual processor begins execution by loading the context from control context from the VMCS – essentially leaving the control registers unchanged (because they have been setup in the VMCS to have the same values they have currently).

8. The virtual machine starts executing at the instruction following the VMGuest assembly language label. This code reads the EFlags register which contains the success/failure flag settings resulting from the execution of VMLAUNCH instruction and returns these values back to the DispatchRead() function.

9. If VMLAUNCH fails, the virtual machine does not execute.

IMPLEMENTATION OF HYPERVISOR

Hypervisor main function is implemented in C part and assembly portions part for hypervisor entry and exit. The execution process starts from guest on the host. The architecture step is shown in figure. C parts program should contain Hypervisors start from

Ds_start 1. data segment size DS_Size 2. the number of guest 3. the max physical page index of guest
4. Max virtual page index for guest

Data structure
(i) The architecture executes under a functional interrupt occurs, under sim vamp until some interrupt occurs.

(ii) The kernel is entered and the interrupted state (here: the processor registers) must be saved, which must be implemented in assembly rather than in pure C. The semantics of assembly instructions can be fully expressed by sim vamp and the effects of the complete code by the contracts of a function kernel entry(h)

(iii) The kernel’s main function kernel main(il) implemented in C then handles the interrupt. (iv) Exiting the kernel and switching to the user again requires an assembly implementation, the effects of which we specify by the contracts of the function
Fig 3: Hypervisor Code organization and Control Flow

Fig 4: Stack Layout for the Host Code

Fig 5: Flow chart of Working of Hypervisor

Working of Hypervisors

The flow diagram or chart gives the hypervisor working from initialization to the exit call.

Performance Evaluate of Hypervisor

The performance of the Hypervisor is maintained on the following constraints

Flexibility - How many constraints are imposed on the monitor

Security - How well the deployment type provides for security coverage

Invisibility - How difficult the presence of the monitor is to detect from within the VM

Speed - How much system slowdown occurs compared to no monitor running

Space - How much storage capability the deployment type possesses
CONCLUSION

We proposed a framework for hypervisor. It is very difficult in designing hypervisor, but we have attempted the framework modules of the hypervisor. We can extend our work in building a generic framework for hypervisor design and development.

REFERENCES


MANGALAGIRI AMBIKA, Research Scholar, ANU in Computer Science and Engg Department. She has authored engineering curriculum book for programming named "Data Structures Using C". Possess 7 Years of professional experience as Assistant Professor in Computer Science and Engineering. Underwent certified training for concepts of Public(AWS & Advanced AWS) and Private (Open Stack)

Dr.Kolasani Ramchand H Rao graduated from Gulbarga University and received his doctoral Degree from Acharya nagarjuna University. And he is working as Professor in Computer Science & Engg at ASN Women's Engineering College ,Tenali,Guntur(Dt),Andhra pradesh and presently under his guidance 8 members are doing their doctoral degree in Acharya Nagarjuna University. He has published 20 research papers in IEEE & ACM Conferences,