Teaching Hypervisor Design, Implementation, and Control to Undergraduate Computer Science and Computer Engineering Students

Yoginder S. Dandass
Mississippi State University
yogi@cse.msstate.edu

Samuel T. Shannon
Mississippi State University
yogi@cse.msstate.edu

David A. Dampier
Mississippi State University
yogi@cse.msstate.edu

Abstract

The study of design issues and implementation techniques for hypervisors is becoming an increasingly important aspect of operating systems pedagogy. There is a demand for students, especially in the field of information assurance, who understand the security issues exposed by the improper use of virtualization functionality provided by modern processors and how virtualization can be exploited to improve system security. Furthermore, students need to understand the process isolation vs. performance tradeoffs that must be made when designing hypervisors.

This paper describes the experience of the authors in teaching a single-semester course to undergraduate students in designing, implementing, and debugging a hypervisor for an Intel 64 processor. Advanced topics in the course include how to capture and manage I/O and interrupt events in the hypervisor. The paper also discusses the use of a PCIe-based hardware module for monitoring and debugging the hypervisor implementation.

1. Introduction

In current information technology (IT) parlance, *virtualization* refers to the ability to partition the hardware 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 (simultaneously if required).

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.

The newer x86 compatible processors by Intel and AMD provide robust processor virtualization support, along with multiple execution cores. These features are enabling the efficient utilization of virtualized platforms in realistic environments – further increasing the popularity of virtualization. However, virtualization also presents significant security challenges if not implemented correctly. A compromised or malicious hypervisor can enable stealthy attacks on the entire set of applications executing on the computer while evading detection by denying or redirecting memory accesses by

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.554

operating system-based malware detection tools such a virus scanners. Furthermore, sophisticated hypervisor-based malware can simply disable well-known defense mechanisms built into operating systems.

We have created an elective course at Mississippi State University targeted towards computer science and computer engineering students who want to gain a deeper understanding of virtualization beyond the material presented in the required OS course. A majority of the students who take this course are interested in the field of information assurance. As described below, this course also discusses how modern operating systems such as Windows 7 and Linux organize processes, virtual paged memory, interrupts, and privilege levels with the support of the Intel 64-bit architecture processors.

This paper describes how the course is structured. Section 2 introduces virtualization from the processor's perspective and describes existing virtualization implementations. Section 3 describes the prerequisite knowledge expected of students and the equipment and software tools required for the hands-on laboratory work. Section 4 describes the organization of the student-developed hypervisor. Section 5 describes a hardware-based debugging and monitoring system for the hypervisor. Section 6 concludes with a discussion of our observations from the first offering of this course.

2. 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 in our course and is not 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 Hyprvisor 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] describes the VMX instructions. In addition, students need to have a basic understanding of interrupt handling, protected mode paged memory management, and processor privilege levels. Much of this information is provided in *Volume 3*, *System Programming Guide*.

It is important to note that although we chose to target the 64-bit Windows 7 operating system as the guest of our hypervisor implementation, we did not need to know the internal implementation details for Windows 7. We were able to read processor state in order to determine just enough information to successfully enable a hypervisor for Windows 7 without having to patch any system binaries. This is a critical observation because the source code for Windows is proprietary, making patching difficult. Furthermore, we wanted to develop a hypervisor that is portable to any operating system with minimal effort.

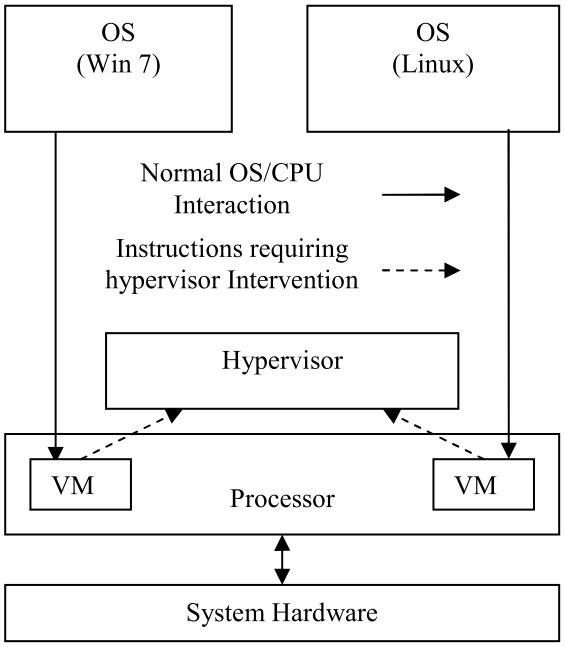

OS
(Win 7)
OS
(Linux)
Normal OS/CPU Interaction
Instructions requiring hypervisor Intervention
Hypervisor
VM
Processor
VM
System Hardware

Figure 1: Organization of operating systems in a virtualized environment

3. Course Environment

The course is a split-level course, open to undergraduate and graduate students who have taken an operating systems course at least at the undergraduate level. They need to have a fundamental understanding of how hardware and operating system software interacts in order to provide paged-mode memory isolation to processes. The Intel 64 architecture-specific paged memory implementation details are reviewed as part of this course on virtualization. Students are also expected to be proficient C programmers and have some experience with assembly language. However, the use of x86 assembly language is minimized in this course (we provide much of the assembly language code to the students in advance). Students are able to learn sufficient assembly language in order to modify supplied code and to contribute small functions where needed.

Most of the code developed and studied in this course is written in C. Because we chose Windows 7 as the target operating systems environment for this course, we use Microsoft's C compiler included in the latest version of the device driver kit (DDK) for code development. This compiler provides a number of intrinsic functions corresponding to specific processor instructions. However, we chose to develop and provide to the students a library of functions written in assembly language in order give students a better idea of how the instructions are used. This technique does not result in the most efficient code because of the function call/return overheads that can be optimized away by the compiler when intrinsic functions are used instead. However, by writing a separate library, we are not dependent on extensions to the C language provided by any specific compiler vendor.

We chose to use the *Netwide Assembler* (NASM) for assembling our x86 assembly language source code into an object file that is linked into our driver executable [8]. NASM is a free assembler that supports the complete IA-32 and Intel 64 instruction set. NASM also requires programmers to use a relatively unambiguous syntax with minimal use of directives as compared to Microsoft's assembler (MASM) which is included in the Windows DDK. Another advantage of NASM over several other assemblers is that is can also be used in Linux environments.

We use the device driver kit to implement a simple kernel mode device driver that inserts our hypervisor into the Windows 7 environment. The device driver is needed because instructions necessary to begin the execution of the hypervisor are *privileged* (*i.e.*, they require the processor to be in kernel mode), and therefore, cannot be executed directly by a user-level application.

Using a device driver for delivering a hypervisor adds some complexity to the laboratory environment because Windows 7 requires drivers to be signed before it executes them. Therefore, we need to create a test certificate, self-sign the device driver using the test certificate, and instruct windows to execute drivers with self-signed certificates. Windows will only execute drivers with self-signed certificates when it is in test mode which is enabled by issuing the command "`bcdedit.exe -set TESTSIGNING ON`" and then rebooting the system. When Windows 7 boots in the test mode, the text "Test Mode" appears at the bottom right corner in the background of the main screen.

Students need to use two separate computer systems for the hands-on laboratory exercises. One system, designated as the *development system*, is dedicated to code development and compiling – it is not used for testing the driver and the hypervisor. In most cases, students chose to use their personal laptops as the development system. This computer can run either windows 7 or 32-bit windows XP (we had no issues cross-compiling 64-bit device driver code for Windows 7 on a development system running 32-bit Windows XP). The second computer, designated as the *test system*, needs to be running 64-bit Windows 7 and have a processor that supports VMX. In our laboratory, we constructed the test computer using Intel's LA1155 chipset and the i7-2600 processor (Sandy Bridge). The VMX functionality was enabled in the BIOS of the test machine. A folder on the test machine was mapped onto the development machine over the network in order to facilitate the delivery of new versions of the device drivers to the test machine.

We developed a number of scripts to help generate test certificates, compile and self-sign the drivers, transport the driver to the test system, and to startup the driver. We also installed Microsoft's *DebugView* tool on the test system [9]. DebugView is used for displaying kernel-mode and Win32 debug output generated by the kernel and device drivers. We found that the liberal use of debugging output statements was, in many cases, easier than using a kernel-mode debugger.

4. Structure of the 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.

4.1. 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 then 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.

4.2. 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 leaded from a machine-specific register (MSR) at a specific location in the VMXON region.

Initialization of the VMCS is more complex because it establishes the contexts of the hypervisor (the host) and the virtual machine in which the operating system (the guest) is executing.

One of the overarching goals of this course was to minimize the amount of coding required by the students so that they could focus on the virtualization concepts rather than spend large amounts of time setting up virtual machine contexts. Therefore, we designed our hypervisor development laboratory assignments along the lines of the Blue-Pill project such that the hypervisor monitors the smallest possible subset of processor status. Additional monitoring and control functionality is added as required for advanced experiments. This means that the operating system continues to control all aspects 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.

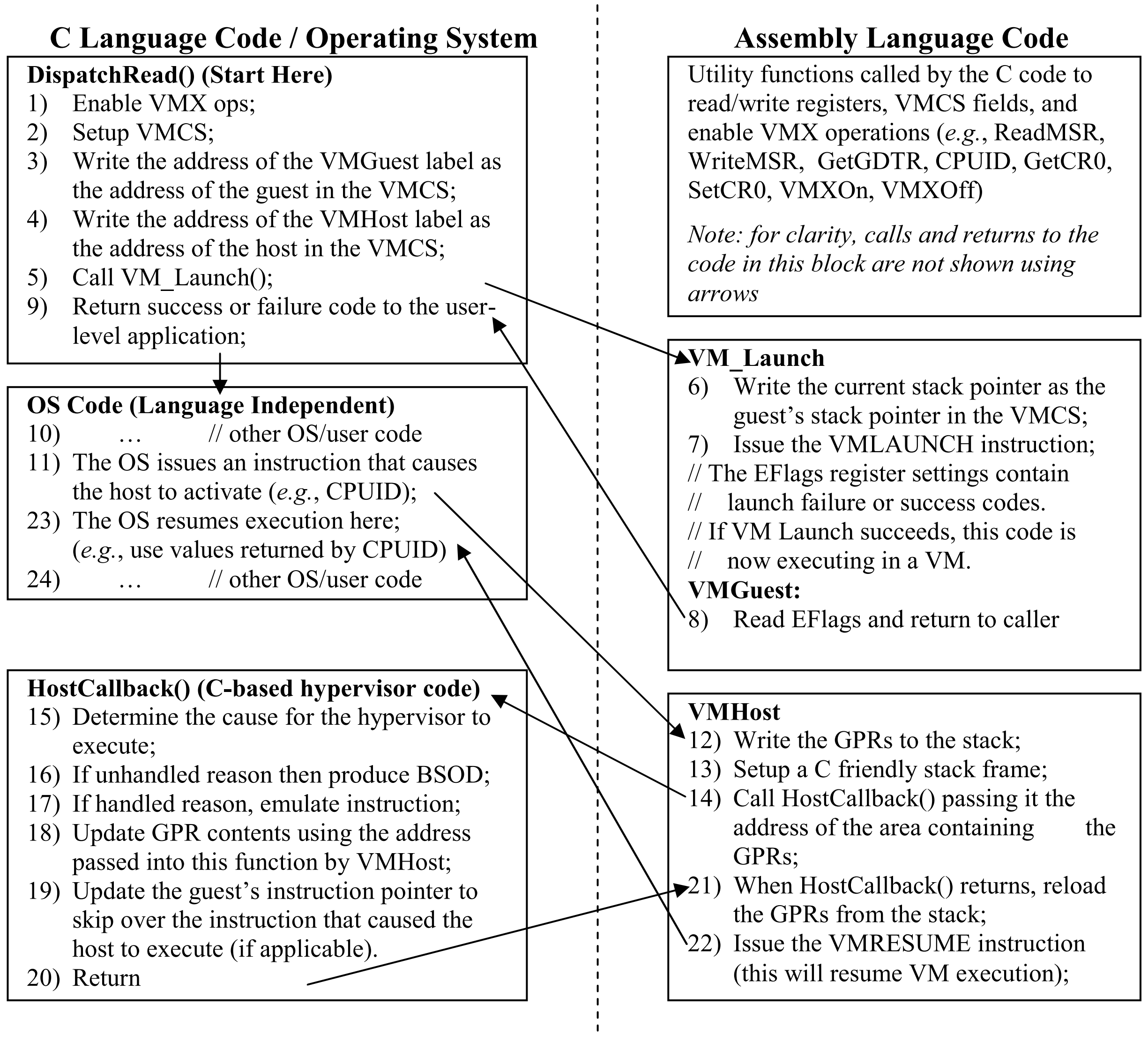
The guest state fields hold the contents of the control, descriptor, and segment registers that are loaded into the virtual processor when the guest code begins or resumes execution. The hypervisor initializes these fields to setup a new virtual machine and the physical processor stores the contents 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.

During course development, the process of establishing the control register context for a virtual machine capable of executing 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 2 depicts the organization of the hypervisor code and its control flow at a high level. Hypervisor setup begins when the device driver's DispatchRead() function receives the command to install the hypervisor. The DispatchRead() function calls several utility functions written in assembly language to enable VMX operations and to initialize the VMXON region and the VMCS. 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. 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. After the VMCS is setup, the DispatchRead function calls the VM_Launch() function written in assembly language. VM_Launch writes the current stack pointer value into the guest's stack pointer field in the VMCS and then issues the VMLAUNCH instruction.

If the VMLUANCH succeeds, the virtual processor begins execution by loading the context from control context from the VMCS – essentially


C Language Code / Operating System

DispatchRead() (Start Here)
1) Enable VMX ops;
2) Setup VMCS;
3) Write the address of the VMGuest label as the address of the guest in the VMCS;
4) Write the address of the VMHost label as the address of the host in the VMCS;
5) Call VM_Launch();
9) Return success or failure code to the user-level application;

OS Code (Language Independent)
10) … // other OS/user code
11) The OS issues an instruction that causes the host to activate (*e.g.*, CPUID);
23) The OS resumes execution here; (*e.g.*, use values returned by CPUID)
24) … // other OS/user code

HostCallback() (C-based hypervisor code)
15) Determine the cause for the hypervisor to execute;
16) If unhandled reason then produce BSOD;
17) If handled reason, emulate instruction;
18) Update GPR contents using the address passed into this function by VMHost;
19) Update the guest's instruction pointer to skip over the instruction that caused the host to execute (if applicable).
20) Return

Assembly Language Code

Utility functions called by the C code to read/write registers, VMCS fields, and enable VMX operations (*e.g.*, ReadMSR, WriteMSR, GetGDTR, CPUID, GetCR0, SetCR0, VMXOn, VMXOff)

Note: for clarity, calls and returns to the code in this block are not shown using arrows

VM_Launch
6) Write the current stack pointer as the guest's stack pointer in the VMCS;
7) Issue the VMLAUNCH instruction;
// The EFlags register settings contain
// launch failure or success codes.
// If VM Launch succeeds, this code is
// now executing in a VM.
VMGuest:
8) Read EFlags and return to caller

VMHost
12) Write the GPRs to the stack;
13) Setup a C friendly stack frame;
14) Call HostCallback() passing it the address of the area containing the GPRs;
21) When HostCallback() returns, reload the GPRs from the stack;
22) Issue the VMRESUME instruction (this will resume VM execution);

Figure 2: Hypervisor Code organization and Control Flow

leaving the control registers unchanged (because they have been setup in the VMCS to have the same values they have currently). 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. If VMLAUNCH fails, the virtual machine does not execute. However, the physical processor continues executing the instructions following the VMLAUNCH instructions – these are the same as above, returning the content of EFlags to DispatchRead(). When control returns to DispatchRead(), the function examines the return value in order to determine whether or not the virtual machine launch was successful (and whether or not it is now running in a virtual machine).

Once the virtual machine is running and the driver returns control back to the operating system, the host code is only invoked when certain instructions are called or any events specified in the VMCS control fields occur. The host always begins execution at the instruction following the VMHost label. This assembly language code saves the content of the general purpose registers onto the host's stack (the stack pointer register is automatically loaded by the processor from the VMCS when the host begins execution). The code also sets up a C-language friendly stack frame before calling the HostCallback() function written in C.

Microsoft describes the 64-bit C/C++ and assembly language application binary interface (ABI) in the Microsoft Software Developer Network (MSDN) documentation online [10]. According to the ABI, 64-bit return codes are returned in the RAX register. Parameters P_1, P_2, …, P_n are passed to functions from left to right order, P_1 in register RCX, P_2 in register RDX, P_3 in register R8, P_4 in register R9, and P_5 through P_n on the stack. Space for the first 4 parameters is also reserved on the stack so that the compiler/assembly language programmer can save their contents if needed. The caller sets up and cleans up the stack. Registers RDI, RSI, RBX, RBP, and R12-R15 must be preserved by the called function (*i.e.*, our assembly language functions). Figure 3 depicts the layout for the stack when the HostCallback C function is called by the assembly language host code.

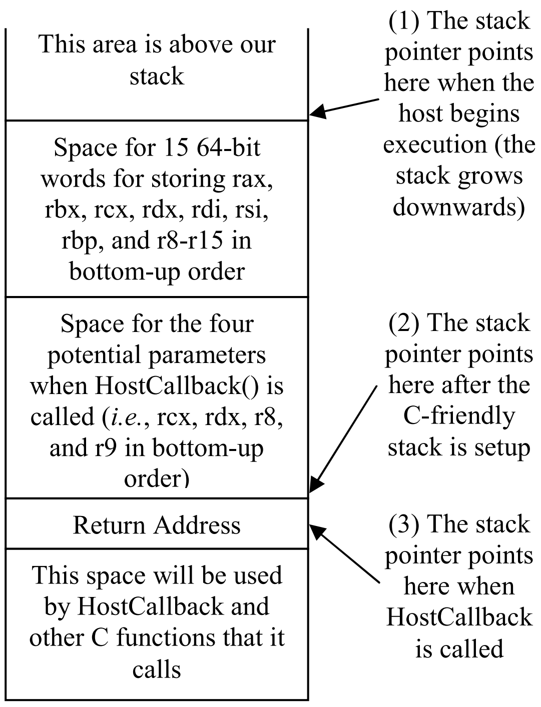

This area is above our stack
Space for 15 64-bit words for storing rax, rbx, rcx, rdx, rdi, rsi, rbp, and r8-r15 in bottom-up order
Space for the four potential parameters when HostCallback() is called (*i.e.*, rcx, rdx, r8, and r9 in bottom-up order)
Return Address
This space will be used by HostCallback and other C functions that it calls
(1) The stack pointer points here when the host begins execution (the stack grows downwards)
(2) The stack pointer points here after the C-friendly stack is setup
(3) The stack pointer points here when HostCallback is called

Figure 3: Stack Layout for the Host Code

4.3. Debugging the Hypervisor

During the initial development phase of the hypervisor, we were plagued by frequent kernel crashes known as the "blue screen of death" (BSOD). These BSOD events were caused because of bugs in the setup of the VMCS fields. It is difficult to debug some of these events because the BSOD immediately replaces any debugging output captured and displayed by the DebugView tool and our user-level application. Sometimes, the test computer would freeze before the debugging information was displayed. Because these BSODs were caused by incorrect setup of the control registers, a kernel debugger also could not be used effectively because the kernel was unstable.

Our initial solution to this debugging problem was to use the PC speaker peripheral to produce tones of varying frequency to indicate progress made by our hypervisor code after the virtual machine launch was initiated. The PC speaker peripheral is a very simple device existing on the I/O bus that only requires about nine assembly instructions to produce a tone from the speaker. By embedding tone producing codes of different frequencies at strategic points in our code, we were able to discern whether or not the VMLAUNCH instruction was successful and if our host code and HostCallback() functions were being called properly.

The system became stable once we corrected the VMCS setup errors. We are now able to debug the hypervisor code using conventional means. For example, we make extensive use of the KdPrint() macro to send kernel-level debugging output to be viewed by the DebugView tool. We also produce our own BSODs for certain conditions (*e.g.*, unhandled reasons for hypervisor invocation – we clearly cannot return to the operating system without reacting to the event/instruction that caused the hypervisor to execute).

Furthermore, because our hypervisor's context is largely the same as Windows' context, we found that we could successfully use the KdPrint() macro from within our hypervisor code (even though the hypervisor is invoked asynchronously). Clearly, calling operating system functions from within the hypervisor is inadvisable because doing so may disrupt the current internal state of the operating system. What we need is a simple persistent storage device that can be accessed simply without requiring complex device driver software because we should not use Windows drivers from within the hypervisor. Our test computer does not have a traditional serial (COM) port (*i.e.*, only USB ports exist on this computer). Programming the USB or Ethernet devices provided on the motherboard was too complex to be considered as a serious alternative. These constraints led us to begin using an FPGA-based peripheral device as a debugging aid that proved to be of great benefit during the initial development and testing of the hypervisor. This device is described in Section 5 below.

4.3. Hypervisor-based Key Logger

After the students completed a stable implementation of the basic hypervisor, we asked them to incorporate key logging functionality into the hypervisor for a legacy PS2 (*i.e.*, non-USB) keyboard. PS2 keyboards are easier to deal with than USB keyboards because their legacy interrupts and control mechanisms are well documented. However, handling external interrupts adds complexity to the hypervisor because it now needs to respond to interrupts and perform advanced I/O operations. Enabling interrupt handling in the hypervisor is enabled by setting the appropriate fields in the VMCS. Doing so causes the hypervisor to be activated when an external keyboard interrupt is issued, instead of the guest operating system's keyboard interrupt handler. The guest will never receive any external keyboard interrupts unless the hypervisor passes the interrupt along to the virtual processor using the event injection VMX functionality when the guest execution is resumed.

Setting up event injection fields in the VMCS is relatively straightforward; the hypervisor simply copies information related to the interrupt into the event injection fields in the VMCS. However, the hypervisor should pass along interrupts only when the guest is ready to process interrupts (*i.e.*, the guest has not disabled interrupts on the virtual processor). If the guest has disabled interrupts, the hypervisor needs to queue the interrupt events and play them back when the guest is able to accept interrupts.

In order to detect when the guest re-enables interrupts, the hypervisor sets fields in the VMCS to capture the enabling of interrupts by the guest. When the hypervisor is invoked because the guest has just enabled interrupts and there are external interrupts in the interrupt queue maintained by the hypervisor, the hypervisor removes the first interrupt in the queue and injects a corresponding event into the guest's execution.

We encountered another interesting problem when capturing keystrokes. Keystroke information is consumed when read from the keyboard controller, and therefore is no longer available to be read by the guest operating system's keyboard interrupt handler. Fortunately, it is possible to program the keyboard controller to replay an arbitrary keystroke; our hypervisor exploited this functionality to setup the keyboard to replay the keystroke recorded along with the interrupt information on the keystroke queue when the event is injected into the guest's execution.

Logged keyboard *make* (*i.e.*, press) and *break* (*i.e.*, release) keyboard scan codes are stored in a buffer and reported using the KdPrint() macro. The content of the buffer can also be reported by our hypervisor driver to the hypervisor control user application on demand. This reporting can be easily accomplished because the hypervisor's keystroke buffer is located in the general-purpose buffer allocated by the driver during initialization, and therefore, is accessible by the driver and the hypervisor simultaneously.

5. Peripheral-based Hypervisor Debugging

We had previously developed an FPGA-based PCIe peripheral device for another research project using the low cost Virtex-5 OpenSPARC Evaluation Platform from Digilent Inc [11]. Our PCIe implementation presents two memory mapped regions to the test computer. The first memory region corresponds to 16 32-bit control registers on the FPGA implementation. The second region corresponds to a 64-Kbyte storage area on the FPGA. Upon starting up, our hypervisor device driver queries the physical base addresses of these two memory regions and requests Windows to setup page table entries that allows the driver to access the memories using logical addresses by using the MmMapIoSpace() kernel call. Once the logical addresses are available, the driver and the hypervisor can readily access these memory areas via normal pointer dereference operations.

We use the FPGA to store trace and debugging information in the second memory region. Essentially, we create formatted execution trace strings similar to those created by the KdPrint() macro and write them onto FPGA's memory. By incrementing the buffer pointer after every write operation, we can keep track of guest and host execution in real-time. Furthermore, because the FPGA is supplied by an external power source, independent of the host computer, the FPGA's memory storage is persistent across host computer boot events. This important fact enables us to perform post mortem analysis of the trace after we reboot the test system following a BSOD or system lockup event. This important capability is not available in standard RAM on our computer which was cleared on all reboot events (including warm reboots).

An additional feature of the FPGA-based device is that it is capable of direct memory access (DMA) operations over the PCIe bus. DMA operations are programmed and initiated by writing the physical addresses of the target memory on the host computer, transfer sizes, and transfer direction (*i.e.*, read or write) into the FPGA implementation's control registers. DMA allows us to copy large portions of host memory into the FPGA for offline analysis asynchronously from the physical (and therefore the virtual processors). Although we did not use this feature in this course, it is available if there is any need for performing asynchronous real-time monitoring of the hypervisor. This is a course option we will seriously considering including in future course offerings.

6. Discussion and Conclusions

We offered this course in the spring 2011 to a small group of five motivated undergraduates. Students were asked to program individually while designing collaboratively. Our goal was to foster individual software development effort while simultaneously encouraging an exchange of ideas. A secondary goal was to reduce what can appear to be an overwhelming amount of design work and technical reading required to design and implement a hypervisor. All students, except for one who had time management issues with all of his courses, produced exemplary results.

6.1. Lessons Learned

During our first offering of the course, we made several observations that will lead to improvements for our subsequent offerings. These observations are summarized below.

We spent a considerable amount of time at the beginning of the semester going over the design of the device driver. Much of this discussion is not necessary for the purpose of the hypervisor development. Also, the students can pick up this material by reading the code themselves and by following excellent tutorials online describing how to write WDK architecture device drivers.

Another large period of time was spent by students in debugging the VMCS setup. Now that we have identified, through experience, the sources of most VMCS setup errors, we can point out the potential for these errors to the students. Also, teaching the student effective debugging skills using offline storage, providing fragments of prewritten code (*e.g.*, the VMLaunch function) can significantly reduce the time spent by students in implementation and debugging. Many students stated that doing this work was immensely rewarding. However, we were set back at least three weeks because of VMCS debugging issues and device driver design discussions. This time can be better spent discussing other issues such as the following: (a) how to design memory protection for the hypervisor so that it is completely isolated from the guest environment, (b) how to design a hypervisor to support more than one guest on at the same time, (c) design issues related to distributed hypervisors (on our current test computer we disable all but one core in our multi-core i7 processor when Windows 7 boots), and (d) how to handle nested hypervisors.

The key-logging implementation, although seemingly straightforward in appearance posed several challenges. The issue of destructive reads from the keyboard controller has already been discussed previously. Our initial design for the key logger was to simply ignore interrupts and all the problems associated with event injections. Instead, we chose setup fields in the VMCS to intercept I/O operations (*i.e.*, the IN and OUT) instructions that target the keyboard ports. However, this technique did not work because in modern chipsets, the keyboard controller is also presented at a memory mapped location and the Windows 7 keyboard driver uses the memory mapped addressed of the keyboard controller's registers as opposed to the I/O bus mappings. Therefore, our I/O event traps were never triggered.

As virtualization becomes increasingly popular, there are a number of well-written technical research papers becoming available (*e.g.*, on the VMware website). Advanced undergraduate and graduate students can develop some these ideas into publishable projects and should be encouraged to do so. A specific area of interest we have is to study the impact of hypervisor execution overheads on the runtime performance of applications running on virtualized platforms.

6.2. Future Expansion

One specific area of interest that we were not able to cover in our initial course offering is how to virtualize the physical memory in the system such that the guest operating system is completely isolated from the hypervisor and other potential guests. Chapter 28 in *Volume 3B*, *System Programming Guide, Part 2* in the *Intel® 64 and IA-32 Architectures Software Developer's Manual* series describes in detail how this can be accomplished. This is a complex task and will require considerable design and development effort in advance of our next offering of the course.

Another important topic to consider including in this course is a discussion and implementation of a hypervisor that utilizes Intel's VX-d technology to virtualize the I/O capabilities of the chipset. I/O virtualization, if implemented properly, has the potential to prevent attacks on the system from malicious functionality embedded in the firmware and/or logic of peripheral devices.

7. References

[1] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, 2011, http://www.intel.com/Assets/PDF/manual/253665.pdf, accessed June 14, 2011.

[2] D. Abramson, J, Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Ulig, B. Vembu, and J. Weigert, "Intel Virtualization Technology for Directed I/O," Intel Technology Journal, Vol 10:3, 2006.

[3] J. Rutkowska, "Subverting Vista Kernel for Fun and Profit", Black Hat Briefings, Las Vegas, 2006, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf, accessed June 14, 2011.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in the Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing (Lake George), New York, October 2003.

[5] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of Architectural Support for Programming Languages and Operating Systems Conference*, San Jose, California, October 2006.

[6] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, 2011, http://www.intel.com/Assets/PDF/manual/325384.pdf, accessed June 14, 2011.

[7] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, 2011*, http://www.intel.com/Assets/PDF/manual/325383.pdf, accessed June 14, 2011.

[8] The NASM Development Team, *NASM*, http://www.nasm.us/xdoc/2.09.08/nasmdoc.pdf, accessed June 14, 2011.

[9] M. Russinovich, *DebugView for Windows v4.76*, http://technet.microsoft.com/en-us/sysinternals/bb896647, accessed June 14, 2011.

[10] Microsoft Corporation, "x64 Software Conventions," *MSDN Online Documentation*, http://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx, accessed June 14, 2011.

[11] Xilinx, Inc., *ML505/ML506/ML507 Evaluation Platform User Guide, UG347 (v3.1.2) May 16, 2011*, http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf, accessed June 14, 2011.