
Virtual Machine Monitors

36.1 Introduction

Years ago, IBM sold mainframes to large organizations, and a problem arose: what if the organization wanted to run different operating systems on the machine? (some applications were developed on one OS, and some on others, and thus the problem) As a solution, IBM introduced yet another level of indirection in the form of a **virtual machine monitor**, or **VMM** or just **monitor** for short.

Specifically, the monitor sits between one or more operating systems and the hardware and gives the illusion to each running OS that it controls the machine. Behind the scenes, however, the monitor actually is in control of the hardware, and must multiplex running OSes across the physical resources of the machine. Indeed, the VMM serves as an operating system for operating systems, but at a much lower level; the OS must still think it is interacting with the physical hardware. Thus, **transparency** is a major goal of VMMs.

Today, VMMs have become popular again for a multitude of reasons. Server consolidation is one such reason. In many settings, people run services on different machines which run different operating systems (or even OS versions), and yet each

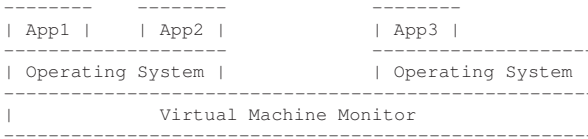


Figure 36.1: A VMM under Two Operating Systems

machine is lightly utilized. In this case, virtualization enables an administrator to **consolidate** multiple OSES onto fewer hardware platforms, and thus lower costs and ease administration. Virtualization has also become popular on desktops, as many users wish to run one operating system (say Linux or Mac OS X) but still have access to native applications on a different platform (say Windows). Thus, for this and many other reasons, virtualization is back again and likely here to stay.

This resurgence began in the mid-to-late 1990's, and was led by a group of researchers at Stanford headed by Professor Mendel Rosenblum. His group's work on Disco [1], a VMM for the MIPS processor, was an early effort that revived VMMs and eventually led that group to the founding of VMware [2], now a market leader in virtualization technology. In this note, we will discuss the primary technology underlying Disco and through that window try to understand how virtualization works.

36.2 Virtualizing the CPU

To run a **virtual machine** (e.g., an OS and its applications) on top of a virtual machine monitor, the basic technique that is used is **direct execution**. Thus, when we wish to "boot" a new OS on top of the VMM, we simply jump to the address of the first instruction and let the OS begin running. It is as simple as that.

Assume we are running on a single processor, and that we wish to multiplex between two virtual machines, that is, be-

tween two OSes and their respective applications. In a manner quite similar to an operating system switching between running processes (a **context switch**), a virtual machine monitor must perform a **machine switch** between running virtual machines. Thus, when performing such a switch, the VMM must save the entire machine state of one OS (including registers, PC, and unlike in a context switch, any privileged hardware state), restore the machine state of the to-be-run VM, and then jump to the PC of the to-be-run VM and thus complete the switch. Note that the to-be-run VM's PC may be within the OS itself (i.e., the system was executing a system call) or it may simply be within a process that is running on that OS (i.e., a user-mode application).

We get into some slightly trickier issues when a running application or OS tries to perform some kind of **privileged operation**. For example, on a system with a software-managed TLB, the OS will use special privileged instructions to update the TLB with a translation before restarting an instruction that suffered a TLB miss. In a virtualized environment, the OS cannot be allowed to perform privileged instructions, because then it controls the machine rather than the VMM beneath it. Thus, the VMM must somehow intercept attempts to perform privileged operations and thus retain control of the machine.

A simple example of how a VMM must interpose on certain operations arises when a running process on a given OS tries to make a system call. For example, the process may be trying to `open()` a file, or may be calling `read()` to get data from it, or may be calling `fork()` to create a new process. In a system without virtualization, a system call is achieved with a special instruction; on MIPS, it is a **trap** instruction, and on x86, it is the `int` (an interrupt) instruction with the argument `0x80`. Here is an open system call in assembly on FreeBSD:

On UNIX-based systems, `open()` takes three arguments:

```
int open(char *path, int flags, mode_t mode);
```

and you can see in the code above how `open()` is implemented: first, the arguments get pushed onto the stack (`mode, flags, path`),

```
open:
    push    dword mode
    push    dword flags
    push    dword path
    mov     eax, 5
    push    eax
    int     80h
    add    esp, byte 16
```

Figure 36.2: `open()` System Call on FreeBSD; See [3] for Details

then a 5 gets pushed onto the stack, and then “int 80h” is called, which transfers control to the kernel. The 5, if you were wondering, is the pre-agreed upon convention between user-mode applications and the kernel for the `open()` system call; different system calls would place different numbers onto the stack (in the same position) before calling the interrupt instruction.

When a trap instruction is executed, it usually does a number of interesting things. Most important in our example here is that it first transfers control (i.e., changes the PC) to a well-defined **trap handler** within the operating system. The OS, when it is first starting up, establishes the address of such a routine with the hardware (also a privileged operation!) and thus upon subsequent traps, the hardware knows where to start running code to handle the trap. At the same time of the trap, the hardware also does one other crucial thing: it changes the mode of the processor from **user mode** to **kernel mode**. In user mode, operations are restricted, and attempts to perform privileged operations will lead to a trap and likely the termination of the offending process; in kernel mode, on the other hand, the full power of the machine is available, and thus all privileged operations can be executed.

Thus, in a traditional setting (again, without virtualization), the flow of control would be like what you see in Table 36.1.

On a virtualized platform, things are a little more interesting. When an application running on an OS wishes to perform a system call, it does the exact same thing: executes a trap instruction with the arguments carefully placed on the stack (or

Process	Hardware	Operating System
1. execute instructions (add, load, etc.)		
2. system call: trap to OS	3. Switch to kernel mode; jump to trap handler	4. In kernel mode; handle system call Return from trap
	5. Switch to user mode; return to executing user code	
6. Resume executing user code		

Table 36.1: Executing a System Call

in registers). However, it is the VMM that controls the machine, and thus the VMM who has installed a trap handler that will first get executed in kernel mode.

So what should the VMM do to handle this system call? The VMM doesn't really know **how** to handle the call; after all, it does not know the details of each OS that is running and therefore does not know what each call should do. What the VMM does know, however, is **where** the OS's trap handler is. It knows this because when the OS booted up, it tried to install its own trap handlers; when the OS did so, it was trying to do something privileged, and therefore trapped into the VMM; at that time, the VMM recorded the necessary information (i.e., where this OS's trap handlers are in memory). Now, when the VMM receives a trap from a user process running on the given OS, it knows exactly what to do: it jumps to the OS's trap handler and lets the OS handle the system call as it should. When the OS is finished, it executes some kind of privileged instruction to return from the trap ("rett" on MIPS), which again bounces into the VMM, which then realizes that the OS is trying to return from the trap and thus performs a real return-from-trap

and thus returns control to the user and puts the machine back in user mode. The entire process is depicted in Tables 36.2 and 36.3, both for the normal case without virtualization and the case with virtualization (we leave out the exact hardware operations from above to save space).

Process	Operating System
1. trap	
	2. OS trap handler: decode trap and execute appropriate syscall routine; when done: return from trap
3. start running again	

Table 36.2: System Call Flow Without Virtualization

Process	Operating System	VMM
1. trap		
		2. Process trapped! call OS trap handler
	3. OS trap handler: decode trap and execute appropriate syscall routine; when done: try return from trap instruction	
		4. OS tried return from trap! do a real return from trap
5. start running again		

Table 36.3: System Call Flow with Virtualization

As you can see from the figures, a lot more has to happen when virtualization is going on. Certainly, because of the extra jumping around, virtualization might indeed slow down system calls and thus could hurt performance.

You might also notice that we have one remaining question: what mode should the OS run in? It can't run in kernel mode, because then it would have unrestricted access to the hardware. Thus, it must run in some less privileged mode than before, be able to access its own data structures, and simultaneously prevent access to its data structures from user processes.

In the Disco work, Rosenblum and colleagues handled this problem quite neatly by taking advantage of a special mode pro-

vided by the MIPS hardware known as supervisor mode. When running in this mode, one still doesn't have access to privileged instructions, but one can access a little more memory than when in user mode; the OS can use this extra memory for its data structures and all is well. On hardware that doesn't have such a mode, one has to run the OS in user mode and use memory protection (page tables and TLBs) to protect OS data structures appropriately.

36.3 Virtualizing Memory

You should now have a basic idea of how the processor is virtualized: the VMM acts like an OS and schedules different virtual machines to run, and some interesting interactions occur when privilege levels change. But we have left out a big part of the equation: how does the VMM virtualize memory?

Each OS normally thinks of OS as a linear array of pages, and assigns each page to itself or user processes. The OS itself, of course, already virtualizes memory for its running processes, such that each process has the illusion of its own private address space. Now we must add another layer of virtualization underneath the OS, so that multiple OSes can share the actual physical memory of the machine, and we must do so transparently.

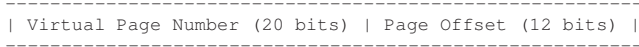
To understand this process, we must first make sure we understand what happens when there is no virtual machine monitor. We thus review what happens on a MIPS-based system during address translation. Let us assume a user process generates an address (this could be for an instruction fetch or an explicit load or store, for example); by definition, the process generates a **virtual address**, as its address space has been virtualized by the OS. It is the role of the OS, with help from the hardware, to turn this into a **physical address** and thus be able to fetch the desired contents from physical memory. The act of turning a virtual address into a physical address is called **address translation**, as you might recall.

Assume we have a 32-bit virtual address space and a 4-KB

Process	Operating System
1. load from memory: TLB miss! trap into OS	2. OS TLB miss trap handler: figure out VPN of address and lookup VPN in page table; if present, get PFN and update TLB /w priv. instructions; return from trap
3. start running again instruction is retried; this time a TLB hit	

Table 36.4: TLB Miss Flow without Virtualization

page size. Thus, our 32-bit address is chopped into two parts: a 20-bit virtual page number (VPN), and a 12-bit offset:



The role of the OS, with help from the TLB, is to translate the VPN into a valid physical page frame number (PFN) and thus produce a fully-formed physical address which can be sent to physical memory to fetch the proper data. In the common case, we expect the TLB to handle the translation in hardware, thus making the translation fast. When a TLB miss occurs (at least, on a system with a software-managed TLB), the OS must get involved to service the miss, as depicted here in Table 36.4.

As you can see from this flow, a TLB miss causes a trap into the OS, which handles the fault by looking up the VPN in the page table and installing the translation (VPN-to-PFN for this PID) in the TLB.

With a virtual machine monitor underneath the OS, however, things again get a little more interesting. Let's examine the flow again (Table 36.5).

As you can see, the picture gets a little more complicated. Now, the VMM is managing memory underneath the OS and thus must interpose on these attempts by the OS to install direct virtual-to-physical mappings and redirect such mappings

Process	Operating System	Virtual Machine Monitor
1. load from memory TLB miss! trap into OS		2. TLB miss handler Call into OS TLB miss handler to actually do the work
	3. OS TLB miss trap handler figure out VPN of address and lookup VPN in page table; if present, get PFN and update TLB /w priv. instructions	4. Trap handler: Somebody (OS) is trying to update the TLB; See that OS is trying to install VPN-to-PFN mapping; Change mapping to desired VPN-to-MFN mapping; jump back to OS, pretending that TLB update worked
	5. return from trap	6. Trap handler Somebody (OS) is trying to return from a trap Thus, do real return from trap
7. start running again instruction is retried; this time a TLB hit		

Table 36.5: TLB Miss Flow with Virtualization

to the mappings desired by the VMM. In other words, the VMM is actually managing what we call real **machine memory** underneath the “physical” memory that the OS thinks it is managing. Thus, each process on a given OS thinks it has its own virtual address space, which the OS multiplexes across physical memory. The VMM, in turn, multiplexes different OS’s physical memories onto the actual machine memory. By inserting another level of indirection, the VMM can control which pages are allocated to which OSes.

This set of actions also hints at how a VMM must manage the virtualization of physical memory for each running OS; just like

the OS has a page table for each process, the VMM must track the physical-to-machine mappings for each virtual machine it is running. These **per-machine page tables** (as opposed to per-process page tables) would thus be consulted in the VMM TLB miss handler in order to determine which machine page a particular “physical” page maps to, and even, for example, if it is **present** in machine memory at the current time (i.e., the VMM could have swapped it to disk to reduce memory pressure).

36.4 Other Issues

There are a huge number of other issues one could discuss about virtualization. One general problem is what some have called the **semantic gap** between the VMM and the OS. The VMM, in general, doesn’t know what the OS is trying to accomplish and this situation can often lead to inefficiencies. For example, an OS, when it has nothing else to run, will sometimes go into an **idle loop** just spinning and waiting for the next interrupt to occur:

```
while (1)
    ; // the idle loop
```

It makes sense to spin like this if the OS in charge of the entire machine and thus knows there is nothing else that needs to run. However, when a VMM is running underneath two different OSes, one in the idle loop and one usefully running user processes, it would be useful for the VMM to know that one of the OSes is now idle so it can give more CPU time to the OS that is doing useful work. There are many other similar examples, e.g., the OS knows which “physical” pages are free but to the VMM they look like the OS is using them; all require either some intelligence on the VMM’s part to infer what is happening or a rewrite of a small portion of the OS to pass the VMM the needed information and thus let it make better decisions.

Many other topics could also be discussed:

- I/O virtualization

- Hosted virtualization where an operating system is running and a VMM is loaded later on the side
- Memory management in more detail
- Paravirtualization, where the OS is modified to be more easily virtualized

But for now, this summary is good enough.

36.5 Summary

Virtualization is in a renaissance. For a multitude of reasons, users and administrators want to run multiple OSes on the same machine at the same time. The key is that VMMs generally provide this service **transparently**; the OS above has little clue that it is not actually controlling the hardware of the machine. The key method that VMMs use to do so is **interposition**; by getting involved on critical privileged events such as traps, the VMM can completely control how machine resources are allocated while preserving the illusion that the OS requires.

References

[1] “Disco: Running Commodity Operating Systems on Scalable Multiprocessors”

Edouard Bugnion, Scott Devine, Kinshuk Govil, Mendel Rosenblum.
SOSP '97.

[2] VMware corporation.

Available: <http://www.vmware.com/>
There, I saved you a google search.

[3] “FreeBSD Developers’ Handbook:

Chapter 11 x86 Assembly Language Programming”

Available: <http://www.freebsd.org/doc/en/books/developers-handbook/x86-system-calls.html>