MODERN OPERATING SYSTEMS

Fifth Edition

Andrew S. Tanenbaum
Herbert Bos

MODERN OPERATING SYSTEMS

FIFTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

7

VIRTUALIZATION AND THE CLOUD

In some situations, an organization has a multicomputer but does not actually want it. A common example is where a company has an email server, a Web server, an FTP server, some e-commerce servers, and others. These all run on different computers in the same equipment rack, all connected by a high-speed network, in other words, a multicomputer. One reason all these servers run on separate machines may be that one machine cannot handle the load, but another is reliability: management simply does not trust the operating system to run 24 hours a day, 365 or 366 days a year, with no failures. By putting each service on a separate computer, if one of the servers crashes, at least the other ones are not affected. This is good for security also. Even if some malevolent intruder manages to compromise the Web server, he will not immediately have access to sensitive emails also—a property sometimes referred to as **sandboxing**. While isolation and fault tolerance are achieved this way, this solution is expensive and hard to manage because so many machines are involved.

Mind you, these are just two out of many reasons for keeping separate machines. For instance, organizations often depend on more than one operating system for their daily operations: a Web server on Linux, a mail server on Windows, an e-commerce server for customers running on macOS and a few other services running on various flavors of UNIX. Again, this solution works, but cheap it is definitely not.

What to do? A possible (and popular) solution is to use virtual machine technology. Virtual machines technology itself is quite old, dating back to the 1960s,

but the way we use it today is a little different. The main idea is that a **VMM** (**Virtual Machine Monitor**) creates the illusion of multiple (virtual) machines on the same physical hardware. A VMM is also known as a **hypervisor**. As discussed in Sec. 1.7.5, we distinguish between type 1 hypervisors which run on the bare metal, and type 2 hypervisors that may make use of all the wonderful services and abstractions offered by an underlying operating system. Either way, **virtualization** allows a single computer to host multiple virtual machines, each potentially running a completely different operating system.

The advantage of this approach is that a failure in one virtual machine does not bring down any others. On a virtualized system, different servers can run on different virtual machines, thus maintaining the partial-failure model that a multicomputer has, but at a lower cost and with easier maintainability. Moreover, we can now run multiple different operating systems on the same hardware, benefit from virtual machine isolation in the face of attacks, and enjoy other good stuff.

Of course, consolidating servers like this is like putting all your eggs in one basket. If the server running all the virtual machines fails, the result is even more catastrophic than the crashing of a single dedicated server. The reason virtualization works, however, is that most service outages are due not to faulty hardware, but to ill-designed, unreliable, buggy, and poorly configured software, emphatically including operating systems. With virtual machine technology, the only software running in the highest privilege mode is the hypervisor, which has two orders of magnitude fewer lines of code than a full operating system, and thus two orders of magnitude fewer bugs. A hypervisor is simpler than an operating system because it does only one thing: emulate multiple copies of the bare metal (most commonly the Intel x86 architecture, although ARM is becoming popular in datacenters also).

Running software in virtual machines has still other advantages in addition to strong isolation. One of them is that having fewer physical machines saves money on hardware and electricity and takes up less rack space. For a company such as Amazon, Google or Microsoft, which may have hundreds of thousands of servers doing a huge variety of different tasks at each data center, reducing the physical demands on their data centers represents a huge cost savings. In fact, server companies sometimes locate their data centers in the middle of nowhere—just to be close to, say, hydroelectric dams (and cheap energy). Virtualization also helps in trying out new ideas. Typically, in large companies, individual departments or groups think of an interesting idea and then go out and buy a server to implement it. If the idea catches on and hundreds or thousands of servers are needed, the corporate data center expands. It is often hard to move the software to existing machines because each application often needs a different version of the operating system, its own libraries, configuration files, and more. With virtual machines, each application can take its own environment with it.

Another advantage of virtual machines is that checkpointing and migrating virtual machines (e.g., for load balancing across multiple servers) is much easier than

migrating processes running on a normal operating system. In the latter case, a fair amount of critical state information about every process is kept in operating system tables, including information relating to open files, alarms, signal handlers, and more. When migrating a virtual machine, all that have to be moved are the memory and disk images, since all the operating system tables move, too.

Another use for virtual machines is to run legacy applications on operating systems (or operating system versions) no longer supported or which do not work on current hardware. These can run at the same time and on the same hardware as current applications. In fact, the ability to run at the same time applications that use different operating systems is a big argument in favor of virtual machines.

Yet another important use of virtual machines is for software development. A programmer who wants to make sure his software works on Windows 10, Windows 11, several versions of Linux, FreeBSD, OpenBSD, NetBSD, and macOS, among other systems no longer has to get a dozen computers and install different operating systems on all of them. Instead, she merely creates a dozen virtual machines on a single computer and installs a different operating system on each one. Of course, she could have partitioned the hard disk and installed a different operating system in each partition, but that approach is more difficult. First of all, standard PCs support only four primary disk partitions, no matter how big the disk is. Second, although a multiboot program could be installed in the boot block, it would be necessary to reboot the computer to work on a new operating system. With virtual machines, all of them can run at once, since they are really just glorified processes.

Perhaps the most important and buzzword-compliant use case for virtualization nowadays is found in the **cloud**. The key idea of a cloud is straightforward: outsource your computation or storage needs to a well-managed data center run by a company specializing in this and staffed by experts in the area. Because the data center typically belongs to someone else, you will probably have to pay for the use of the resources, but at least you will not have to worry about the physical machines, power, cooling, and maintenance. Because the isolation offered by virtualization, cloud-providers can allow multiple clients, even competitors, to share a single physical machine. Each client gets a piece of the pie. At the risk of stretching the cloud metaphor, we mention that early critics maintained that the pie was only in the sky and that real organizations would not want to put their sensitive data and computations on someone else's resources. By now, however, virtualized machines in the cloud are used by countless organization for countless applications, and while it may not be for all organizations and all data, there is no doubt that cloud computing has been a tremendous success.

We mentioned that hypervisor-based virtualization allows administrators to run multiple isolated operating systems on the same hardware. In case there is no need for *different* operating systems, just multiple instances of the same one, there is an alternative to hypervisors, known as **OS-level virtualization**. As the name indicates, it is the operating system that creates multiple virtual environments for user

space—commonly referred to as **containers** or **jails**. There are many differences with hypervisor-based virtualization. The main one is that while a container may *look* like an isolated computer (with its own devices, its own memory, etc.), the underlying operating system is fixed and cannot be replaced by another. A second important difference is that since all containers use the same underlying operating system (and hence share its resources), the isolation between containers is less complete than that between virtual machines. A third difference is that, precisely because the containers directly use the services of a single operating system, they are often more lightweight and efficient than hypervisor-based solutions. In this chapter, our main focus is on hypervisor-based virtualization, but we will discuss OS-level virtualization also.

7.1 HISTORY

With all the hype surrounding virtualization in recent years, we sometimes forget that by Internet standards virtual machines are ancient. As early as the 1960s. IBM experimented with not just one but two independently developed hypervisors: **SIMMON** and **CP-40**. While CP-40 was a research project, it was reimplemented as **CP-67** to form the control program of **CP/CMS**, a virtual machine operating system for the IBM System/360 Model 67. Later, it was reimplemented again and released as **VM/370** for the System/370 series in 1972. The System/370 line was replaced by IBM in the 1990s by the System/390. This was essentially just a name change since the underlying architecture remained the same for reasons of backward compatibility. Of course, the hardware technology was greatly improved and the newer machines were bigger and faster than the older ones, but as far as virtualization was concerned, nothing changed. In 2000, IBM released the z-series, which supported 64-bit virtual address spaces but was otherwise backward compatible with the System/360. All of these systems supported virtualization decades before it became popular on the x86.

In 1974, two computer scientists at UCLA, Gerald Popek and Robert Goldberg, published a seminal paper ("Formal Requirements for Virtualizable Third Generation Architectures") that listed exactly what conditions a computer architecture should satisfy in order to support virtualization efficiently (Popek and Goldberg, 1974). It is impossible to write a chapter on virtualization without referring to their work and terminology. Famously, the well-known x86 architecture that also originated in the 1970s did not meet these requirements for decades. It was not the only one. Nearly every architecture since the mainframe also failed the test. The 1970s were very productive, seeing also the birth of UNIX, Ethernet, the Cray-1, Microsoft, and Apple—so, despite what your parents may say, the 1970s were not just about disco!

In fact, the real **Disco** revolution started in the 1990s, when researchers at Stanford University developed a new hypervisor by that name and went on to found **VMware**, a virtualization giant that offers type 1 and type 2 hypervisors and now

rakes in billions of dollars in revenue (Bugnion et al., 1997, Bugnion et al., 2012). Incidentally, the distinction between "type 1" and "type 2" hypervisors is also from the seventies (Goldberg, 1972). VMware introduced its first virtualization solution for x86 in 1999. In its wake other products followed: **Xen**, **KVM**, **VirtualBox, Hyper-V, Parallels**, and many others. It seems the time was right for virtualization, even though the theory had been nailed down in 1974 and for decades IBM had been selling computers that supported—and heavily used—virtualization. In 1999, it became popular among the masses, but new it was not, despite the massive attention it suddenly gained.

OS-level virtualization, while not as old as hypervisors, has a history that stretches back quite some time also. In 1979, UNIX v7 introduced a new system call: chroot (change root). The system call takes as its single argument a path name, for instance /home/hjb/my_new_root, which is where it will create a new "root" directory for the current process and all of its children. Thus, when the process reads a file /README.txt (a file in the root directory), it really accesses /home/hjb/my_new_root/README.txt. In other words, the operating system has created a separate environment on disk to which the process is confined. It cannot access files from directories other than those in the subtree below the new root (and we had better make sure that all files the process ever needs will be in this subtree, because they are literally all it can access).

A kind soul might consider this enough to be called a virtual environment, but clearly it is an exceedingly poor man's version of that. For instance, while the visibility of the file system is limited to a subtree, there is no isolation of processes or their privileges. Thus, a process inside the subtree can still send signals to processes outside the subtree. Similarly, a process with administrator (or "root") privileges cannot be properly locked inside the chroot subtree: having root privileges, it can do *anything*, including breaking out of the confined file name space. Another problem is that processes in different chroot subtrees still have access to all the IP addresses assigned to the machine and there is no isolation between packets sent from this process and those sent by processes in other subtrees. In other words, these are not virtual machines at all.

In 2000, Poul-Henning Kamp and Robert Watson extended the chroot isolation in the FreeBSD operating system to create what they referred to as **FreeBSD Jails** (Kamp and Watson, 2000). They partitioned resources much more pervasively: Jails had their own file system name spaces, their own IP addresses, their own (limited) root processes, etc. Their elegant solution was hugely influential and soon similar features appeared in other operating systems, often partitioning even more resources, such as memory or CPU usage. In 2008, Linux Containers (LXC) was released. Based on an earlier resource partitioning project at Google, LXC offers partitioning of resources of a collection of processes through containers. However, the popularity of containers really exploded with the launch of **Docker** in 2013. Docker uses OS-level virtualization to allow software to be packaged in containers that hold all the code, libraries, and configuration files needed to run it.

Many people use the term "virtualization" to refer exclusively to hypervisor-based solutions, and use the term "containerization" to talk about OS-level virtualization. While we emphasize that in reality both are forms of virtualization, we reluctantly adopt the same convention in this chapter. So, unless we explicitly say otherwise (e.g., when we talk about containers), you may assume that henceforth virtualization refers to the hypervisor variety.

7.2 REQUIREMENTS FOR VIRTUALIZATION

If we leave OS-level virtualization aside, it is important that virtual machines act just like the real McCoy. In particular, it must be possible to boot them like real machines and install arbitrary operating systems on them, just as can be done on the real hardware. It is the task of the hypervisor to provide this illusion and to do it efficiently. Indeed, hypervisors should score well in three dimensions:

1. **Safety**: The hypervisor should have full control of the virtualized resources.
2. **Fidelity**: The behavior of a program on a virtual machine should be identical to that of the same program running on bare hardware.
3. **Efficiency**: Much of the code in the virtual machine should run without intervention by the hypervisor.

An unquestionably safe way to execute the instructions is to consider each instruction in turn in an **interpreter** (such as Bochs) and perform exactly what is needed for that instruction. Some instructions can be executed directly, but not too many. For instance, the interpreter may be able to execute an INC (increment) instruction simply as is, but instructions that are not safe to execute directly must be simulated by the interpreter. For instance, we cannot really allow the guest operating system to disable interrupts for the entire machine or modify the page-table mappings. The trick is to make the operating system on top of the hypervisor think that it has disabled interrupts, or changed the machine's page mappings. We will see how this is done later. For now, we just want to say that the interpreter may be safe, and if carefully implemented, perhaps even hi-fi, but the performance sucks. To also satisfy the performance criterion, we will see that VMMs try to execute most of the code directly.

Now let us turn to fidelity. Virtualization has long been a problem on the x86 architecture due to defects in the Intel 386 architecture that were automatically carried forward into new CPUs for 20 years in the name of backward compatibility. In a nutshell, every CPU with kernel mode and user mode has a set of instructions that behave differently when executed in kernel mode than when executed in user mode. These include instructions that do I/O, change the MMU settings, and so on. Popek and Goldberg called these **sensitive instructions**. There is also a set of

instructions that cause a trap if executed in user mode. Popek and Goldberg called these **privileged instructions**. Their paper stated for the first time that a machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions. In simpler language, if you try to do something in user mode that you should not be doing in user mode, the hardware should trap. Unlike the IBM/370, which had this property, Intel's 386 did not. Quite a few sensitive 386 instructions were ignored if executed in user mode or executed with different behavior. For example, the POPF instruction replaces the flags register, which changes the bit that enables/disables interrupts. In user mode, this bit is simply not changed. As a consequence, the 386 and its successors could not be virtualized, so they could not support a hypervisor directly.

Actually, the situation is even worse than sketched. In addition to the problems with instructions that fail to trap in user mode, there are instructions that can read sensitive state in user mode without causing a trap. For example, on x86 processors prior to 2005, a program can determine whether it is running in user mode or kernel mode by reading its code-segment selector. An operating system that did this and discovered that it was actually in user mode might make an incorrect decision based on this information.

This problem was finally solved when Intel and AMD introduced virtualization in their CPUs starting in 2005 (Uhlig et al., 2005). On the Intel CPUs it is called **VT** (**Virtualization Technology**); on the AMD CPUs it is called **SVM** (**Secure Virtual Machine**). We will use the term VT in a generic sense below. Both were inspired by the IBM VM/370 work, but they are slightly different. The basic idea is to create environments in which virtual machines can be run. When a guest operating system is started up in an environment, it continues to run there until it causes an exception and traps to the hypervisor, for example, by executing an I/O instruction. The set of operations that trap is controlled by a hardware bitmap set by the hypervisor. With these extensions, the classical **trap-and-emulate** virtual machine approach becomes possible.

The astute reader may have noticed an apparent contradiction in the description thus far. On the one hand, we have said that x86 was not virtualizable until the architecture extensions introduced in 2005. On the other hand, we saw that VMware launched its first x86 hypervisor in 1999. How can both be true at the same time? The answer is that the hypervisors before 2005 did not really run the original guest operating system. Rather, they *rewrote* part of the code on the fly to replace problematic instructions with safe code sequences that emulated the original instruction. Suppose, for instance, that the guest operating system performed a privileged I/O instruction, or modified one of the CPU's privileged control registers (like the CR3 register which contains a pointer to the page directory). It is important that the consequences of such instructions are limited to this virtual machine and do not affect other virtual machines, or the hypervisor itself. Thus, an unsafe I/O instruction was replaced by a trap that, after a safety check, performed an equivalent instruction and returned the result. Since we are rewriting, we can

use the trick to replace instructions that are sensitive, but not privileged. Other instructions execute natively. The technique is known as **binary translation**; we will discuss it more detail in Sec. 7.4.

There is no need to rewrite all sensitive instructions. In particular, user processes on the guest can typically run without modification. If the instruction is nonprivileged but sensitive and behaves differently in user processes than in the kernel, that is fine. We are running it in userland anyway. For sensitive instructions that are privileged, we can resort to the classical trap-and-emulate, as usual. Of course, the VMM must ensure that it receives the corresponding traps. Typically, the VMM has a module that executes in the kernel and redirects the traps to its own handlers.

A different form of virtualization is known as **paravirtualization**. It is quite different from **full virtualization**, because it never even aims to present a virtual machine that looks just like the actual underlying hardware. Instead, it presents a machine-like software interface that explicitly exposes the fact that it is a virtualized environment. For instance, it offers a set of **hypercalls**, which allow the guest to send explicit requests to the hypervisor (much as a system call offers kernel services to applications). Guests use hypercalls for privileged sensitive operations like updating the page tables, but because they do it explicitly in cooperation with the hypervisor, the overall system can be simpler and faster.

It should not come as a surprise that paravirtualization is not new either. IBM's VM operating system has offered such a facility, albeit under a different name, since 1972. The idea was revived by the Denali (Whitaker et al., 2002) and Xen (Barham et al., 2003) virtual machine monitors. Compared to full virtualization, the drawback of paravirtualization is that the guest has to be aware of the virtual machine API. This means it needs to be customized explicitly for the hypervisor.

Before we delve more deeply into type 1 and type 2 hypervisors, it is important to mention that not all virtualization technology tries to trick the guest into believing that it has the entire system. Sometimes, the aim is simply to allow a process to run that was originally written for a different operating system and/or architecture. We therefore distinguish between full system virtualization and **process-level virtualization**. While we focus on the former in the remainder of this chapter, process-level virtualization technology is used in practice also. Well-known examples include the WINE compatibility layer that allows Windows application to run on POSIX-compliant systems like Linux, BSD, and macOS, and the process-level version of the QEMU emulator that allows applications for one architecture to run on another.

7.3 TYPE 1 AND TYPE 2 HYPERVISORS

Goldberg (1972) distinguished between two approaches to virtualization. One kind of hypervisor, dubbed a **type 1 hypervisor**, is illustrated in Fig. 7-1(a). Technically, it is like an operating system, since it is the only program running in the

most privileged mode. Its job is to support multiple copies of the actual hardware, called **virtual machines**, similar to the processes a normal operating system runs.

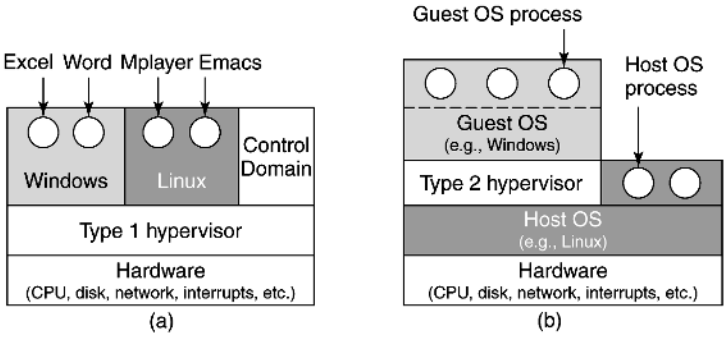

Figure 7-1. Location of type 1 and type 2 hypervisors.

In contrast, a **type 2 hypervisor**, shown in Fig. 7-1(b), is a different kind of animal. It is a program that relies on, say, Windows or Linux to allocate and schedule resources, very much like a regular process. Of course, the type 2 hypervisor still pretends to be a full computer with a CPU and various devices. Both types of hypervisor must execute the machine's instruction set in a safe manner. For instance, an operating system running on top of the hypervisor may change and even mess up its own page tables, but not those of others.

The operating system running on top of the hypervisor in both cases is called the **guest operating system**. For a type 2 hypervisor, the operating system running on the hardware is called the **host operating system**. The first type 2 hypervisor on the x86 market was **VMware Workstation** (Bugnion et al., 2012). In this section, we introduce the general idea. A study of VMware in more detail follows in Sec. 7.12.

Type 2 hypervisors, sometimes referred to as **hosted hypervisors**, depend for much of their functionality on a host operating system such as Windows, Linux, or macOS. When it starts for the first time, it acts like a newly booted computer and expects to find a DVD, USB drive, or CD-ROM containing an operating system in the drive. This time, however, the drive could be a virtual device. For instance, it is possible to store the image as an ISO file on the hard drive of the host and have the hypervisor pretend it is reading from a proper DVD drive. It then installs the operating system to its **virtual disk** (again really just a Windows, Linux, or macOS file) by running the installation program found on the DVD. Once the guest operating system is installed on the virtual disk, it can be booted and run. It is completely unaware that it has been tricked.

The various categories of virtualization we have discussed are summarized in the table of Fig. 7-2 for both type 1 and type 2 hypervisors. For each combination of hypervisor and kind of virtualization, some examples are given.

Virtualizaton method	Type 1 hypervisor	Type 2 hypervisor
Virtualization without HW support	ESX Server 1.0	VMware Workstation 1
Paravirtualization	Xen 1.0	VirtualBox 5.0+
Virtualization with HW support	vSphere, Xen, Hyper-V	VMware Fusion, KVM, Parallels
Process virtualization		Wine

Figure 7-2. Examples of hypervisors. Type 1 hypervisors run on the bare metal, whereas type 2 hypervisors use the services of an existing host operating system.

7.4 TECHNIQUES FOR EFFICIENT VIRTUALIZATION

Virtualizability and performance are important issues, so let us examine them more closely. Assume, for the moment, that we have a type 1 hypervisor supporting one virtual machine, as shown in Fig. 7-3. Like all type 1 hypervisors, it runs on the bare metal. The virtual machine runs as a user process in user mode, and as such is not allowed to execute sensitive instructions (in the Popek-Goldberg sense). However, the virtual machine runs a guest operating system that thinks it is in kernel mode (although, of course, it is not). We will call this **virtual kernel mode**. The virtual machine also runs user processes, which think they are in user mode (and really are in user mode).

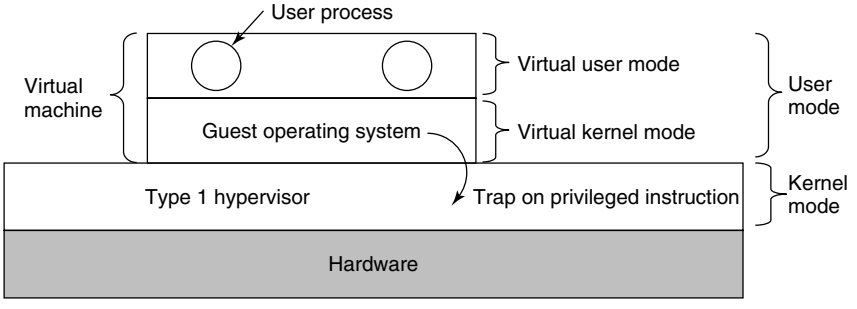

Figure 7-3. When the operating system in a virtual machine executes a kernel-only instruction, it traps to the hypervisor if virtualization technology is present.

What happens when the guest operating system (which thinks it is in kernel mode) executes an instruction that is allowed only when the CPU really is in kernel mode? Normally, on CPUs without VT, the instruction fails and the operating system crashes. On CPUs with VT, when the guest operating system executes a sensitive instruction, a trap to the hypervisor does occur, as illustrated in Fig. 7-3. The hypervisor can then inspect the instruction to see if it was issued by the guest operating system in the virtual machine or by a user program in the virtual machine. In the former case, it arranges for the instruction to be carried out; in the latter case, it

emulates what the real hardware would do when confronted with a sensitive instruction executed in user mode.

7.4.1 Virtualizing the Unvirtualizable

Building a virtual machine system is relatively straightforward when VT is available, but what did people do before that? For instance, VMware released a hypervisor well before the arrival of the virtualization extensions on the x86. Again, the answer is that the software engineers who built such systems made clever use of **binary translation** and hardware features that *did* exist on the x86, such as the processor's **protection rings**.

For many years, the x86 has supported four protection modes or rings. Ring 3 is the least privileged. This is where normal user processes execute. In this ring, you cannot execute privileged instructions. Ring 0 is the most privileged ring that allows the execution of any instruction. In normal operation, the kernel runs in ring 0. The remaining two rings are not used by any current operating system. In other words, hypervisors were free to use them as they pleased. As shown in Fig. 7-4, many virtualization solutions therefore kept the hypervisor in kernel mode (ring 0) and the applications in user mode (ring 3), but put the guest operating system in a layer of intermediate privilege (ring 1). As a result, the kernel is privileged relative to the user processes and any attempt to access kernel memory from a user program leads to an access violation. At the same time, the guest operating system's privileged instructions trap to the hypervisor. The hypervisor does some sanity checks and then performs the instructions on the guest's behalf.

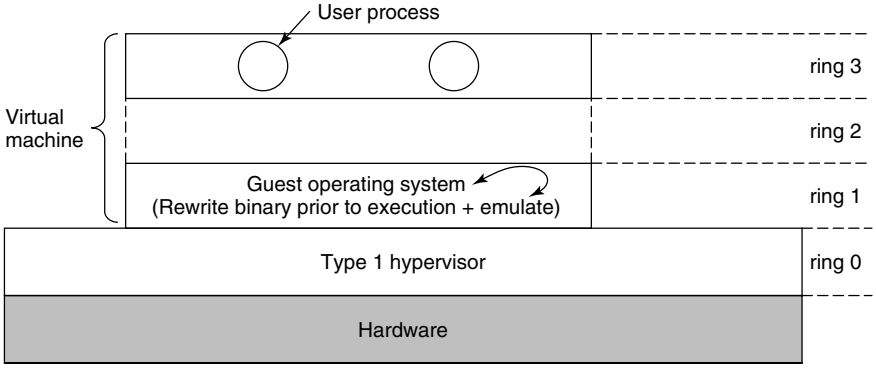

Figure 7-4. The binary translator rewrites the guest operating system running in ring 1, while the hypervisor runs in ring 0.

As for the sensitive instructions in the guest's kernel code: the hypervisor makes sure they no longer exist. To do so, it rewrites the code, one basic block at a time. A **basic block** is a short, straight-line sequence of instructions that ends with

a branch. By definition, a basic block contains no jump, call, trap, return, or other instruction that alters the flow of control, except for the very last instruction which does precisely that. Just prior to executing a basic block, the hypervisor first scans it to see if it contains sensitive instructions (in the Popek and Goldberg sense), and if so, replaces them with a call to a hypervisor procedure that handles them. The branch on the last instruction is also replaced by a call into the hypervisor (to make sure it can repeat the procedure for the next basic block). Dynamic translation and emulation sound expensive, but typically are not. Translated blocks are cached, so no translation is needed in the future. Also, most code blocks do not contain sensitive or privileged instructions and thus can execute natively. In particular, as long as the hypervisor configures the hardware carefully (as is done, for instance, by VMware), the binary translator can ignore all user processes; they execute in nonprivileged mode anyway.

After a basic block has completed executing, control is returned to the hypervisor, which then locates its successor. If the successor has already been translated, it can be executed immediately. Otherwise, it is first translated, cached, then executed. Eventually, most of the program will be in the cache and run at close to full speed. Various optimizations are used, for example, if a basic block ends by jumping to (or calling) another one, the final instruction can be replaced by a jump or call directly to the translated basic block, eliminating all overhead associated with finding the successor block. Again, there is no need to replace sensitive instructions in user programs; the hardware will just ignore them anyway.

On the other hand, it is common to perform binary translation on all the guest operating system code running in ring 1 and replace even the privileged sensitive instructions that, in principle, could be made to trap also. The reason is that traps are very expensive and binary translation leads to better performance.

So far we have described a type 1 hypervisor. Although type 2 hypervisors are conceptually different from type 1 hypervisors, they use, by and large, the same techniques. For instance, VMware ESX Server (a type 1 hypervisor first shipped in 2001) used exactly the same binary translation as the first VMware Workstation (a type 2 hypervisor released two years earlier).

However, to run the guest code natively and use exactly the same techniques requires the type 2 hypervisor to manipulate the hardware at the lowest level, which cannot be done from user space. For instance, it has to set the segment descriptors to exactly the right value for the guest code. For faithful virtualization, the guest operating system should also be tricked into thinking that it is the true and only operating system, with full control of all the machine's resources and with access to the entire address space (4 GB on 32-bit machines). When the guest operating system finds another system (the host kernel) squatting in its address space, first one will not be amused.

Unfortunately, this is exactly what happens when the guest runs as a user process on a regular operating system. For instance, in Linux a user process has access to just 3 GB of the 4-GB address space, as the remaining 1 GB is reserved for the

kernel. Any access to the kernel memory leads to a trap. In principle, it is possible to take the trap and emulate the appropriate actions, but doing so is expensive and typically requires installing the appropriate trap handler in the host kernel. Another (obvious) way to solve the two-kings problem is to reconfigure the system to remove the host operating system and actually give the guest the entire address space. However, doing so is clearly not possible from user space either.

Likewise, the hypervisor needs to handle the interrupts to do the right thing, for instance when the disk sends an interrupt or a page fault occurs. Also, if the hypervisor wants to use trap-and-emulate for privileged instructions, it needs to receive the traps. Again, installing trap/interrupt handlers in the kernel is not possible for user processes.

Most modern type 2 hypervisors therefore have a kernel module operating in ring 0 that allows them to manipulate the hardware with privileged instructions. Of course, manipulating the hardware at the lowest level and giving the guest access to the full address space is all well and good, but at some point the hypervisor needs to clean it up and restore the original processor context. Suppose, for instance, that the guest is running when an interrupt arrives from an external device. Since a type 2 hypervisor depends on the host's device drivers to handle the interrupt, it needs to reconfigure the hardware completely to run the host operating system code. When the device driver runs, it finds everything just as it expected it to be. The hypervisor behaves just like teenagers throwing a party while their parents are away. It is okay to rearrange the furniture completely, as long as they put it back exactly as they found it before the parents come home. Going from a hardware configuration for the host kernel to a configuration for the guest operating system is known as a **world switch**. We will discuss it in detail when we discuss VMware in Sec. 7.12.

It should now be clear why these hypervisors work, even on unvirtualizable hardware: sensitive instructions in the guest kernel are replaced by calls to procedures that emulate these instructions. No sensitive instructions issued by the guest operating system are ever executed directly by the true hardware. They are turned into calls to the hypervisor, which then emulates them.

7.4.2 The Cost of Virtualization

One might naively expect that CPUs with VT would greatly outperform software techniques that resort to translation, but measurements showed a mixed picture (Adams and Agesen, 2006). It turns out that the trap-and-emulate approach used by VT hardware generates a lot of traps, and traps are very expensive on modern hardware because they ruin CPU caches, TLBs, and branch prediction tables internal to the CPU. In contrast, when sensitive instructions are replaced by calls to hypervisor procedures within the executing process, none of this context-switching overhead is incurred. As Adams and Agesen show, depending on the workload, sometimes software would beat hardware. For this reason, some type 1 (and

type 2) hypervisors do binary translation for performance reasons, even though the software will execute correctly without it. In recent years, this situation changed and state-of-the-art CPUs and hypervisors are quite efficient with hardware virtualization. For instance, VMware no longer has a binary translator.

With binary translation, the translated code itself may be either slower or faster than the original code. Suppose, for instance, that the guest operating system disables hardware interrupts using the CLI instruction ("clear interrupts"). Depending on the architecture, this instruction can be very slow, taking many tens of cycles on certain CPUs with deep pipelines and out-of-order execution. It should be clear by now that the guest's wanting to turn off interrupts does not mean the hypervisor should really turn them off and affect the entire machine. Thus, the hypervisor must turn them off for the guest without really turning them off. To do so, it may keep track of a dedicated **IF** (**Interrupt Flag**) in the virtual CPU data structure it maintains for each guest (making sure the virtual machine does not get any interrupts until the interrupts are turned off again). Every occurrence of CLI in the guest will be replaced by something like "VirtualCPU.IF = 0", which is a very cheap move instruction that may take as little as one to three cycles. Thus, the translated code is faster. Still, with modern VT hardware, usually the hardware beats the software.

On the other hand, if the guest operating system modifies its page tables, this is very costly. The problem is that each guest operating system on a virtual machine thinks it "owns" the machine and is at liberty to map any virtual page to any physical page in memory. However, if one virtual machine wants to use a physical page that is already in use by another virtual machine (or the hypervisor), something has to give. We will see in Sec. 7.6 that the solution is to add an extra level of page tables to map "guest physical pages" to the actual physical pages on the host. Not surprisingly, mucking around with multiple levels of page tables is not cheap.

7.5 ARE HYPERVISORS MICROKERNELS DONE RIGHT?

Both type 1 and type 2 hypervisors work with unmodified guest operating systems, but have to jump through hoops to get good performance. We have seen that paravirtualization takes a different approach by modifying the source code of the guest operating system instead. Rather than performing sensitive instructions, the paravirtualized guest executes hypercalls. In effect, the guest operating system is acting like a user program making system calls to the operating system (the hypervisor). When this route is taken, the hypervisor must define an interface consisting of a set of procedure calls that guest operating systems can use. This set of calls forms what is effectively an **API** (**Application Programming Interface**) even though it is an interface for use by guest operating systems, not application programs.

Going one step further, by removing all the sensitive instructions from the operating system and just having it make hypercalls to get system services like I/O,

we have turned the hypervisor into a microkernel, like that of Fig. 1-26. The idea, explored in paravirtualization, is that emulating peculiar hardware instructions is an unpleasant and time-consuming task. It requires a call into the hypervisor and then emulating the exact semantics of a complicated instruction. It is far better just to have the guest operating system call the hypervisor (or microkernel) to do I/O, and so on.

Indeed, some researchers have argued that we should perhaps consider hypervisors as "microkernels done right" (Hand et al., 2005). The first thing to mention is that this is a highly controversial topic and some researchers have vocally opposed the notion, arguing that the difference between the two is not fundamental to begin with (Heiser et al., 2006). Others suggest that compared to microkernels, hypervisors may not even be that well suited for building secure systems, and advocate that they be extended with kernel functionality like message passing and memory sharing (Hohmuth et al., 2004). Finally, some researchers have made the argument that perhaps hypervisors are not even "operating systems research done right" (Roscoe et al., 2007). Since nobody said anything about operating system textbooks done right (or wrong)—yet—we think we do right by exploring the similarity between hypervisors and microkernels a bit more.

The main reason the first hypervisors emulated the complete machine was the lack of availability of source code for the guest operating system (e.g., for Windows) or the vast number of variants (e.g., for Linux). Perhaps in the future the hypervisor/microkernel API will be standardized, and subsequent operating systems will be designed to call it instead of using sensitive instructions. Doing so would make virtual machine technology easier to support and use.

The difference between true virtualization and paravirtualization is illustrated in Fig. 7-5. Here we have two virtual machines being supported on VT hardware. On the left is an unmodified version of Windows as the guest operating system. When a sensitive instruction is executed, the hardware causes a trap to the hypervisor, which then emulates it and returns. On the right is a version of Linux modified so that it no longer contains any sensitive instructions. Instead, when it needs to do I/O or change critical internal registers (such as the one pointing to the page tables), it makes a hypervisor call to get the work done, just like an application program making a system call in standard Linux.

In Fig. 7-5, we have shown the hypervisor as being divided into two parts separated by a dashed line. In reality, only one program is running on the hardware. One part of it is responsible for interpreting trapped sensitive instructions, in this case, from Windows. The other part of it just carries out hypercalls. In the figure the latter part is labeled "microkernel." If the hypervisor is intended to run only paravirtualized guest operating systems, there is no need for the emulation of sensitive instructions and we have a true microkernel, which just provides very basic services such as process dispatching and managing the MMU. The boundary between a type 1 hypervisor and a microkernel is vague already and will probably get even less clear as hypervisors begin acquiring more and more functionality and

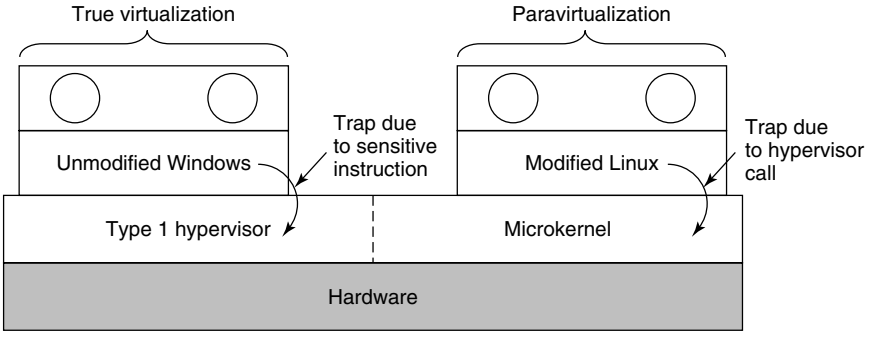

Figure 7-5. True virtualization and paravirtualization.

hypercalls, as seems likely. Again, this subject is controversial, but it is increasingly clear that the program running in kernel mode on the bare hardware should be small and reliable and consist of thousands, not millions, of lines of code.

Paravirtualizing the guest operating system raises a number of serious issues. First, if the sensitive (i.e., kernel-mode) instructions are replaced with calls to the hypervisor, how can the operating system run on the native hardware? After all, the hardware does not understand these hypercalls. And second, what if there are multiple hypervisors available in the marketplace, such as VMware, the open source Xen originally from the University of Cambridge, and Microsoft's Hyper-V, all with somewhat different hypervisor APIs? How can the kernel be modified to run on all of them?

Amsden et al. (2006) have proposed a solution. In their model, the kernel is modified to call special procedures whenever it needs to do something sensitive. Together these procedures, called the **VMI** (**Virtual Machine Interface**), form a low-level layer that interfaces with the hardware or hypervisor. These procedures are designed to be generic and not tied to any specific hardware platform or to any particular hypervisor.

An example of this technique is given in Fig. 7-6 for a paravirtualized version of Linux they call VMI Linux (VMIL). When VMI Linux runs on the bare hardware, it has to be linked with a library that issues the actual (sensitive) instruction needed to do the work, as shown in Fig. 7-6(a). When running on a hypervisor, say VMware or Xen, the guest operating system is linked with different libraries that make the appropriate (and different) hypercalls to the underlying hypervisor. In this way, the core of the operating system remains portable yet is hypervisor friendly and still efficient.

Other proposals for a virtual machine interface have also been made. Another popular one is called **paravirt ops**. The idea is conceptually similar to what we described above, but different in the details. Essentially, a group of several Linux vendors that included companies like IBM, VMware, Xen, and Red Hat advocated

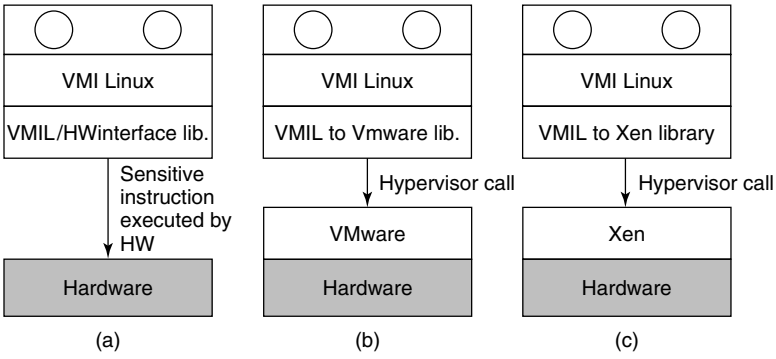

Figure 7-6. VMI Linux running on (a) the bare hardware, (b) VMware, and (c) Xen.

a hypervisor-agnostic interface for Linux. The interface, included in the mainline kernel from version 2.6.23 onward, allows the kernel to talk to whatever hypervisor is managing the physical hardware.

7.6 MEMORY VIRTUALIZATION

So far we have addressed the issue of how to virtualize the CPU. But a computer system has more than just a CPU. It also has memory and I/O devices. They have to be virtualized, too. Let us see how that is done.

Modern operating systems nearly all support virtual memory, which is basically a mapping of pages in the virtual address space onto pages of physical memory. This mapping is defined by (multilevel) page tables. Typically the mapping is set in motion by having the operating system set a control register in the CPU that points to the top-level page table. Virtualization greatly complicates memory management. In fact, it took hardware manufacturers two tries to get it right.

Suppose, for example, a virtual machine is running, and the guest operating system in it decides to map its virtual pages 7, 4, and 3 onto physical pages 10, 11, and 12, respectively. It builds page tables containing this mapping and loads a hardware register to point to the top-level page table. This instruction is sensitive. On a VT CPU, it will trap; with dynamic translation it will cause a call to a hypervisor procedure; on a paravirtualized operating system, it will generate a hypercall. For simplicity, let us assume it traps into a type 1 hypervisor, but the problem is the same in all three cases.

What does the hypervisor do now? One solution is to actually allocate physical pages 10, 11, and 12 to this virtual machine and set up the actual page tables to map the virtual machine's virtual pages 7, 4, and 3 to use them. So far, so good.

Now suppose a second virtual machine starts and maps its virtual pages 4, 5, and 6 onto physical pages 10, 11, and 12 and loads the control register to point to its page tables. The hypervisor catches the trap, but what should it do? It cannot use this mapping because physical pages 10, 11, and 12 are already in use. It can find some free pages, say 20, 21, and 22, and use them, but it first has to create new page tables mapping the virtual pages 4, 5, and 6 of virtual machine 2 onto 20, 21, and 22. If another virtual machine starts and tries to use physical pages 10, 11, and 12, it has to create a mapping for them. In general, for each virtual machine the hypervisor needs to create a **shadow page table** that maps the virtual pages used by the virtual machine onto the actual pages the hypervisor gave it.

Worse yet, every time the guest operating system changes its page tables, the hypervisor must change the shadow page tables as well. For example, if the guest OS remaps virtual page 7 onto what it sees as physical page 200 (instead of 10), the hypervisor has to know about this change. The trouble is that the guest operating system can change its page tables by just writing to memory. No sensitive operations are required, so the hypervisor does not even know about the change and certainly cannot update the shadow page tables used by the actual hardware.

A possible (but clumsy) solution is for the hypervisor to keep track of which page in the guest's virtual memory contains the top-level page table. It can get this information the first time the guest attempts to load the hardware register that points to it because this instruction is sensitive and traps. The hypervisor can create a shadow page table at this point and also map the top-level page table and the page tables it points to as read only. A subsequent attempts by the guest operating system to modify any of them will cause a page fault and thus give control to the hypervisor, which can analyze the instruction stream, figure out what the guest OS is trying to do, and update the shadow page tables accordingly. It is not pretty, but it is doable in principle.

Another, equally clumsy, solution is to do exactly the opposite. In this case, the hypervisor simply allows the guest to add new mappings to its page tables at will. As this is happening, nothing changes in the shadow page tables. In fact, the hypervisor is not even aware of it. However, as soon as the guest tries to access any of the new pages, a fault will occur and control reverts to the hypervisor. The hypervisor inspects the guest's page tables to see if there is a mapping that it should add, and if so, adds it and reexecutes the faulting instruction. What if the guest removes a mapping from its page tables? Clearly, the hypervisor cannot wait for a page fault to happen, because it will not happen. Removing a mapping from a page table happens by way of the INVLPG instruction (which is really intended to invalidate a TLB entry). The hypervisor therefore intercepts this instruction and removes the mapping from the shadow page table also. Again, not pretty, but it works.

Both of these techniques incur many page faults, and page faults are expensive. We typically distinguish between "normal" page faults that are caused by guest programs that access a page that has been paged out of RAM, and page faults that are related to ensuring the shadow page tables and the guest's page tables are in

sync. The former are known as **guest-induced page faults**, and while they are intercepted by the hypervisor, they must be reinjected into the guest. This is not cheap at all. The latter are known as **hypervisor-induced page faults** and they are handled by updating the shadow page tables.

Page faults are always expensive, but especially so in virtualized environments, because they lead to so-called VM exits. A **VM exit** is a situation in which the hypervisor regains control. Consider what the CPU needs to do for such a VM exit. First, it records the cause of the VM exit, so the hypervisor knows what to do. It also records the address of the guest instruction that caused the exit. Next, a context switch is done, which includes saving all the registers. Then, it loads the hypervisor's processor state. Only then can the hypervisor start handling the page fault, which was expensive to begin with. Oh, and when it is all done, it should reverse these steps. The whole process may take tens of thousands of cycles, or more. No wonder people bend over backward to reduce the number of exits.

In a paravirtualized operating system, the situation is different. Here the paravirtualized OS in the guest knows that when it is finished changing some process' page table, it had better inform the hypervisor. Consequently, it first changes the page table completely, then issues a hypervisor call telling the hypervisor about the new page table. Thus, instead of a protection fault on every update to the page table, there is one hypercall when the whole thing has been updated, obviously a more efficient way to do business.

Hardware Support for Nested Page Tables

The cost of handling shadow page tables led chip makers to add hardware support for **nested page tables**. Nested page tables is the term used by AMD. Intel refers to them as **EPT** (**Extended Page Tables**). They are similar and aim to remove most of the overhead by handling the additional page-table manipulation all in hardware, all without any traps. Interestingly, the first virtualization extensions in Intel's x86 hardware did not include support for memory virtualization at all. While these VT-extended processors removed many bottlenecks concerning CPU virtualization, poking around in page tables was as expensive as ever. It took a few years for AMD and Intel to produce the hardware to virtualize memory efficiently.

Recall that even without virtualization, the operating system maintains a mapping between the virtual pages and the physical page. The hardware "walks" these page tables to find the physical address that corresponds to a virtual address. Adding more virtual machines simply adds an extra mapping. As an example, suppose we need to translate a virtual address of a Linux process running on a type 1 hypervisor like Xen or VMware ESX Server to a physical address. In addition to the **guest virtual addresses**, we now also have **guest physical addresses** and subsequently **host physical addresses** (sometimes referred to as **machine physical addresses**). We have seen that without EPT, the hypervisor is responsible for

maintaining the shadow page tables explicitly. With EPT, the hypervisor still has an additional set of page tables, but now the CPU is able to handle much of the intermediate level in hardware also. In our example, the hardware first walks the "regular" page tables to translate the guest virtual address to a guest physical address, just as it would do without virtualization. The difference is that it also walks the extended (or nested) page tables without software intervention to find the host physical address, and it needs to do this every time a guest physical address is accessed. The translation is illustrated in Fig. 7-7.

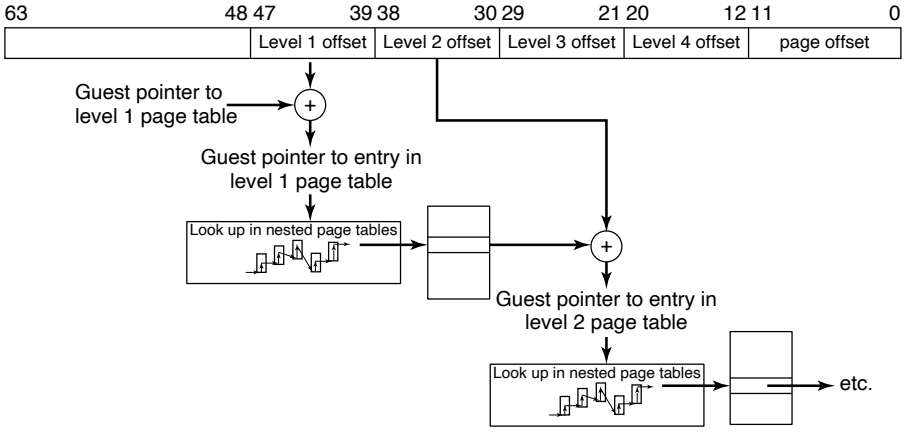

Figure 7-7. Extended/nested page tables are walked every time a guest physical address is accessed—including the accesses for each level of the guest's page tables.

Unfortunately, the hardware may need to walk the nested page tables more frequently then you might think. Let us suppose that the guest virtual address was not cached and requires a full page-table lookup. Every level in paging hierarchy incurs a lookup in the nested page tables. In other words, the number of memory references grows quadratically with the depth of the hierarchy. Even so, EPT dramatically reduces the number of VM exits. Hypervisors no longer need to map the guest's page table read only and can do away with shadow page-table handling. Better still, when switching virtual machines, it just changes this mapping, the same way an operating system changes the mapping when switching processes.

Reclaiming Memory

Having all these virtual machines on the same physical hardware all with their own memory pages and all thinking they are the king of the mountain is great—until we need the memory back. This is particularly important in the event of **overcommitment** of memory, where the hypervisor pretends that the total amount of memory for all virtual machines combined is more than the total amount of real

memory present on the system. In general, this is a good idea, because it allows the hypervisor to admit more and more beefy virtual machines at the same time. For instance, on a machine with 32 GB of memory, it may run three virtual machines each thinking it has 16 GB of memory. Clearly, this does not fit. However, perhaps the three machines do not really need the maximum amount of physical memory at the same time. Or perhaps they share pages that have the same content (such as the Linux kernel) in different virtual machines in an optimization known as **deduplication**. In that case, the three virtual machines use a total amount of memory that is less than 3 times 16 GB. We will discuss deduplication later; for the moment the point is that what looks like a good distribution now may be a poor distribution as the workloads change. Maybe virtual machine 1 needs more memory, while virtual machine 2 could do with fewer pages. In that case, it would be nice if the hypervisor could transfer resources from one virtual machine to another and make the system as a whole benefit. The question is, how can we take away memory pages safely if that memory is given to a virtual machine already?

In principle, we could use yet another level of paging. In case of memory shortage, the hypervisor would then page out some of the virtual machine's pages, just as an operating system may page out some of an application's pages. The drawback of this approach is that the hypervisor should do this, and the hypervisor has no clue about which pages are the most valuable to the guest. It is very likely to page out the wrong ones. Even if it does pick the right pages to swap (i.e., the pages that the guest OS would also have picked), there is still more trouble ahead. For instance, suppose that the hypervisor pages out a page *P*. A little later, the guest OS also decides to page out this page to disk. Unfortunately, the hypervisor's swap space and the guest's swap space are not the same. In other words, the hypervisor must first page the contents back into memory, only to see the guest write it back out to disk immediately. Not very efficient.

A common solution is to use a trick known as **ballooning**, where a small balloon module is loaded in each VM as a pseudo device driver that talks to the hypervisor. The balloon module may inflate at the hypervisor's request by allocating more and more pinned pages, and deflate by deallocating these pages. As the balloon inflates, memory scarcity in the guest increases. The guest operating system will respond by paging out what it believes are the least valuable pages—which is just what we wanted. Conversely, as the balloon deflates, more memory becomes available for the guest to allocate. In other words, the hypervisor tricks the operating system into making tough decisions for it. In politics, this is known as passing the buck (or the euro, pound, yen, etc.).

7.7 I/O VIRTUALIZATION

Having looked at CPU and memory virtualization, we next examine I/O virtualization. The guest operating system will typically start out probing the hardware to find out what kinds of I/O devices are attached. These probes will trap to the

hypervisor. What should the hypervisor do? One approach is for it to report back that the disks, printers, and so on are the ones that the hardware actually has. The guest will then load device drivers for these devices and try to use them. When the device drivers try to do actual I/O, they will read and write the device's hardware device registers. These instructions are sensitive and will trap to the hypervisor, which could then copy the needed values to and from the hardware registers, as needed.

But here, too, we have a problem. Each guest OS could think it owns an entire disk partition, and there may be many more virtual machines (hundreds) than there are actual disk partitions. The usual solution is for the hypervisor to create a file or region on the actual disk for each virtual machine's physical disk. Since the guest OS is trying to control a disk that the real hardware has (and which the hypervisor understands), it can convert the block number being accessed into an offset into the file or disk region being used for storage and do the I/O.

It is also possible for the disk that the guest is using to be different from the real one. For example, if the actual disk is some brand-new high-performance disk (or RAID) with a new interface, the hypervisor could advertise to the guest OS that it has a plain old IDE disk and let the guest OS install an IDE disk driver. When this driver issues IDE disk commands, the hypervisor converts them into commands to drive the new disk. This strategy can be used to upgrade the hardware without changing the software. In fact, this ability of virtual machines to remap hardware devices was one of the reasons VM/370 became popular: companies wanted to buy new and faster hardware but did not want to change their software. Virtual machine technology made this possible.

Another interesting trend related to I/O is that the hypervisor can take the role of a virtual switch. In this case, each virtual machine has a MAC address and the hypervisor switches frames from one virtual machine to another—just like an Ethernet switch would do. Virtual switches have several advantages. For instance, it is very easy to reconfigure them. Also, it is possible to augment the switch with additional functionality, for instance for additional security.

I/O MMUs

Another I/O problem that must be solved somehow is the use of DMA, which uses absolute memory addresses. As might be expected, the hypervisor has to intervene here and remap the addresses before the DMA starts. However, hardware already exists with an **I/O MMU**, which virtualizes the I/O the same way the MMU virtualizes the memory. I/O MMU exists in different forms and shapes for many processor architectures. Even if we limit ourselves to the x86, Intel and AMD have slightly different technology. Still, the idea is the same. This hardware eliminates the DMA problem.

Just like regular MMUs, the I/O MMU uses page tables to map a memory address that a device wants to use (the device address) to a physical address. In a

virtual environment, the hypervisor can set up the page tables in such a way that a device performing DMA will not trample over memory that does not belong to the virtual machine on whose behalf it is working.

I/O MMUs offer different advantages when dealing with a device in a virtualized world. **Device pass through** allows the physical device to be directly assigned to a specific virtual machine. In general, it would be ideal if device address space were exactly the same as the guest's physical address space. However, this is unlikely—unless you have an I/O MMU. The MMU allows the addresses to remapped transparently, and both the device and the virtual machine are blissfully unaware of the address translation that takes place under the hood.

Device isolation ensures that a device assigned to a virtual machine can directly access that virtual machine without jeopardizing the integrity of the other guests. In other words, the I/O MMU prevents rogue DMA traffic, just as a normal MMU prevents rogue memory accesses from processes—in both cases accesses to unmapped pages result in faults.

DMA and addresses are not the whole I/O story, unfortunately. For completeness, we also need to virtualize interrupts, so that the interrupt generated by a device arrives at the right virtual machine, with the right interrupt number. Modern I/O MMUs therefore support **interrupt remapping**. Say, a device sends a message signaled interrupt with number 1. This message first hits the I/O MMU that will use the interrupt remapping table to translate to a new message destined for the CPU that currently runs the virtual machine and with the vector number that the VM expects (e.g., 66).

And finally, having an I/O MMU allows 32-bit devices to access memory above 4 GB. Normally, such devices are unable to access (e.g., DMA to) addresses beyond 4 GB, but the I/O MMU can easily remap the device's lower addresses to any address in the physical larger address space.

Device Domains

A different approach to handling I/O is to dedicate one of the virtual machines to run a standard operating system and reflect all I/O calls from the other ones to it. This approach is enhanced when paravirtualization is used, so the command being issued to the hypervisor actually says what the guest OS wants (e.g., read block 1403 from disk 1) rather than being a series of commands writing to device registers, in which case the hypervisor has to play Sherlock Holmes and figure out what it is trying to do. Xen uses this approach to I/O, with the virtual machine that does I/O called **domain 0**.

I/O virtualization is an area in which type 2 hypervisors have a practical advantage over type 1 hypervisors: the host operating system contains the device drivers for all the weird and wonderful I/O devices attached to the computer. When an application program attempts to access a strange I/O device, the translated code can call the existing device driver to get the work done. With a type 1 hypervisor,

the hypervisor must either contain the driver itself, or make a call to a driver in domain 0, which is somewhat similar to a host operating system. As virtual machine technology matures, future hardware is likely to allow application programs to access the hardware directly in a secure way, meaning that device drivers can be linked directly with application code or put in separate user-mode servers (as in MINIX3), thereby eliminating the problem.

Single Root I/O Virtualization

Directly assigning a device to a virtual machine is not very scalable. With four physical networks you can support no more than four virtual machines that way. For eight virtual machines you need eight network cards, and to run 128 virtual machines—well, let's just say that it may be hard to find your computer buried under all those network cables.

Sharing devices among multiple hypervisors in software is possible, but often not optimal because an emulation layer (or device domain) interposes itself between hardware and the drivers and the guest operating systems. The emulated device frequently does not implement all the advanced functions supported by the hardware. Ideally, the virtualization technology would offer the equivalence of device pass through of a single device to multiple hypervisors, without any overhead. Virtualizing a single device to trick every virtual machine into believing that it has exclusive access to its own device is much easier if the hardware actually does the virtualization for you. On PCIe, this is known as single root I/O virtualization.

SR-IOV (**Single Root I/O Virtualization**) allows us to bypass the hypervisor's involvement in the communication between the driver and the device. Devices that support SR-IOV provide an independent memory space, interrupts and DMA streams to each virtual machine that uses it (Intel, 2011). The device appears as multiple separate devices and each can be configured by separate virtual machines. For instance, each will have a separate base address register and address space. A virtual machine maps one of these memory areas (used for instance to configure the device) into its address space.

SR-IOV provides access to the device in two flavors: **PF** (**Physical Functions**) and (**Virtual Functions**). PFs are full PCIe functions and allow the device to be configured in whatever way the administrator sees fit. Physical functions are not accessible to guest operating systems. VFs are lightweight PCIe functions that do not offer such configuration options. They are ideally suited for virtual machines. In summary, SR-IOV allows devices to be virtualized in (up to) hundreds of virtual functions that trick virtual machines into believing they are the sole owner of a device. For example, given an SR-IOV network interface, a virtual machine is able to handle its virtual network card just like a physical one. Better still, many modern network cards have separate (circular) buffers for sending and receiving data, dedicated to this virtual machines. For instance, the Intel I350 series of network cards has eight send and eight receive queues.

7.8 VIRTUAL MACHINES ON MULTICORE CPUS

The combination of virtual machines and multicore CPUs creates a whole new world in which the number of CPUs available can be set by the software. If there are, say, four cores, and each can run, for example, up to eight virtual machines, a single (desktop) CPU can be configured as a 32-node multicomputer if need be, but it can also have fewer CPUs, depending on the software. Never before has it been possible for an application designer to first choose how many CPUs he wants and then write the software accordingly. This is clearly a new phase in computing.

Moreover, virtual machines can share memory. A typical example where this is useful is a single server hosting multiple instances of the same operating systems. All that has to be done is map physical pages into the address spaces of multiple virtual machines. Memory sharing is already available in deduplication solutions. **Deduplication** does exactly what you think it does: avoids storing the same data twice. It is a fairly common technique in storage systems, but it is now appearing in virtualization as well. In Disco, it was known as **transparent page sharing** (which requires modification to the guest), while VMware calls it **content-based page sharing** (which does not require any modification). In general, the technique revolves around scanning the memory of each of the virtual machines on a host and hashing the memory pages. Should some pages produce an identical hash, the system has to first check to see if they really are the same, and if so, deduplicate them, creating one page with the actual content and two references to that page. Since the hypervisor controls the nested (or shadow) page tables, this mapping is straightforward. Of course, when either of the guests modifies a shared page, the change should not be visible in the other virtual machine(s). The trick is to use **copy on write** so the modified page will be private to the writer.

If virtual machines can share memory, a single computer becomes a virtual multiprocessor. Since all the cores in a multicore chip share the same RAM, a single quad-core chip could easily be configured as a 32-node multiprocessor or a 32-node multicomputer, as needed.

The combination of multicore, virtual machines, hypervisor, and microkernels is going to radically change the way people think about computer systems. Current software cannot deal with the idea of the programmer determining how many CPUs are needed, whether they should be a multicomputer or a multiprocessor, and how minimal kernels of one kind or another fit into the picture. Future software will have to deal with these issues. If you are a computer science or engineering student or professional, you could be the one to sort out all this stuff. Go for it!

7.9 CLOUDS

Virtualization technology played a crucial role in the dizzying rise of cloud computing. There are many clouds. Some clouds are public and available to anyone willing to pay for the use of resources, others are private to an organization.

Likewise, different clouds offer different things. Some give their users access to physical hardware, but most virtualize their environments. Some offer the bare machines, virtual or not, and nothing more, but others offer software that is ready to use and can be combined in interesting ways, or platforms that make it easy for their users to develop new services. Cloud providers typically offer different categories of resources, such as "big machines" versus "little machines."

For all the talk about clouds, few people seem really sure about what they are exactly. The National Institute of Standards and Technology, always a good source to fall back on, lists five essential characteristics:

1. **On-demand self-service.** Users should be able to provision resources automatically, without requiring human interaction.
2. **Broad network access.** All these resources should be available over the network via standard mechanisms so that heterogeneous devices can make use of them.
3. **Resource pooling.** The computing resource owned by the provider should be pooled to serve multiple users and with the ability to assign and reassign resources dynamically. The users generally do not even know the exact location of "their" resources or even which country they are located in.
4. **Rapid elasticity.** It should be possible to acquire and release resources elastically, perhaps even automatically, to scale immediately with the users' demands.
5. **Measured service.** The cloud provider meters the resources used in a way that matches the type of service agreed upon.

7.9.1 Clouds as a Service

In this section, we will look at clouds with a focus on virtualization and operating systems. Specifically, we consider clouds that offer direct access to a virtual machine, which the user can use in any way he sees fit. Thus, the same cloud may run different operating systems, possibly on the same hardware. In cloud terms, this is known as **IAAS** (**Infrastructure As A Service**), as opposed to **PAAS** (**Platform As A Service**, which delivers an environment that includes things such as a specific OS, database, Web server, and so on), **SAAS** (**Software As A Service**), which offers access to specific software, such as Microsoft Office 365, or Google Apps), **FAAS** (**Function As A Service**), which helps you deploy applications to the cloud, and many other types of as-a-service. One example of an IAAS cloud is Amazon AWS, which happens to be based on the Xen hypervisor and counts multiple hundreds of thousands of physical machines. Provided you have the cash, you can have as much computing power as you need.

Clouds can transform the way companies do computing. Overall, consolidating the computing resources in a small number of places (conveniently located near a power source and cheap cooling) benefits from economy of scale. Outsourcing your processing means that you need not worry so much about managing your IT infrastructure, backups, maintenance, depreciation, scalability, reliability, performance, and perhaps security. All of that is done in one place and, assuming the cloud provider is competent, done well. You would think that IT managers are happier today than 10 years ago. However, as these worries disappeared, new ones emerged. Can you really trust your cloud provider to keep your sensitive data safe? Will a competitor running on the same infrastructure be able to infer information you wanted to keep private? What law(s) apply to your data (for instance, if the cloud provider is from the United States, is your data subject to the PATRIOT Act, even if your company is in Europe)? Once you store all your data in cloud X, will you be able to get them out again, or will you be tied to that cloud and its provider forever, something known as **vendor lock-in**?

7.9.2 Virtual Machine Migration

Virtualization technology not only allows IAAS clouds to run multiple different operating systems on the same hardware at the same time, it also permits clever management. We have already discussed the ability to overcommit resources, especially in combination with deduplication. Now we will look at another management issue: what if a machine needs servicing (or even replacement) while it is running lots of important machines? Probably, clients will not be happy if their systems go down because the cloud provider wants to replace a disk drive.

Hypervisors decouple the virtual machine from the physical hardware. In other words, it does not really matter to the virtual machine if it runs on this machine or that machine. Thus, the administrator could simply shut down all the virtual machines and restart them again on a shiny new machine. Doing so, however, results in significant downtime. The challenge is to move the virtual machine from the hardware that needs servicing to the new machine without taking it down at all.

A slightly better approach might be to pause the virtual machine, rather than shut it down. During the pause, we copy over the memory pages used by the virtual machine to the new hardware as quickly as possible, configure things correctly in the new hypervisor and then resume execution. Besides memory, we also need to transfer storage and network connectivity, but if the machines are close, this can be relatively fast. We could make the file system network-based to begin with (like NFS, the network file system), so that it does not matter whether your virtual machine is running on hardware in server rack 1 or 3. Likewise, the IP address can simply be switched to the new location. Nevertheless, we still need to pause the machine for a noticeable amount of time. Less time perhaps, but still noticeable.

Instead, what modern virtualization solutions offer is something known as **live migration**. In other words, they move the virtual machine while it is still operational. For instance, they employ techniques like **pre-copy memory migration**. This means that they copy memory pages while the machine is still serving requests. Most memory pages are not written much, so copying them over is safe. Remember, the virtual machine is still running, so a page may be modified after it has already been copied. When memory pages are modified, we have to make sure that the latest version is copied to the destination, so we mark them as dirty. They will be recopied later. When most memory pages have been copied, we are left with a small number of dirty pages. We now pause very briefly to copy the remaining pages and resume the virtual machine at the new location. While there is still a pause, it is so brief that applications typically are not affected. When the downtime is not noticeable, it is known as a **seamless live migration**.

7.9.3 Checkpointing

Decoupling of virtual machine and physical hardware has additional advantages. In particular, we mentioned that we can pause a machine. This in itself is useful. If the state of the paused machine (e.g., CPU state, memory pages, and storage state) is stored on disk, we have a snapshot of a running machine. If the software makes a royal mess of the still-running virtual machine, it is possible to just roll back to the snapshot and continue as if nothing happened.

The most straightforward way to make a snapshot is to copy everything, including the full file system. However, copying a multiterabyte disk may take a while, even if it is a fast disk. And again, we do not want to pause for long while we are doing it. The solution is to use **copy on write** solutions, so that data are copied only when absolutely necessary.

Snapshotting works quite well, but there are issues. What to do if a machine is interacting with a remote computer? We can snapshot the system and bring it up again at a later stage, but the communicating party may be long gone. Clearly, this is a problem that cannot be solved.

7.10 OS-LEVEL VIRTUALIZATION

So far, we studied hypervisor-based virtualization, but in the beginning of this chapter we mentioned that there is this thing called OS-level virtualization also. Instead of presenting the illusion of a machine, the idea is to create isolated user space environments, also known as a **containers** or **jails**, as mentioned earlier.

As much as possible, each container and all the processes in it are isolated from the other containers and the rest of the system. For instance, they may all have their own file system "name space": a subtree that starts from a root that the administrator created with the chroot system call. A process in this name space

cannot access other name spaces (subtrees). However, having your own root directory is not enough. For proper isolation, containers also need separate name spaces for process identifiers, user identifiers, network interfaces (and the associated IP addresses), IPC endpoints, etc. Furthermore, isolation of (and limits on) memory and CPU usage would also be nice. Perhaps you can think of a few other resources that should be restricted?

We have already seen that limiting access to a particular file system name space using a system call like something like chroot is relatively simple: the operating system remembers that for this group of processes all file operations are relative to the new root. If one of the processes opens a file in /home/hjb/, the operating system knows that this is relative to the new root. We can apply similar tricks to the other name spaces. For instance, when a process opens all network interfaces on the system, the operating makes sure to open only the network interface(s) assigned to the group/container. Similarly, process and user identifiers can be virtualized, so that when a process sends a signal to the process with process identifier 6293, the operating system translates that number to the "real" process identifier. All of this is straightforward. However, how does one partition resources such as memory and CPU usage?

In general, we need a way to track resource usage for groups of processes for a wide variety of resources. Different operating systems have different solutions. One of the better known ones, the Linux **cgroups** (control groups) feature, we will use for illustration purposes. It allows administrators to organize processes in sets known as cgroups, and to monitor and limit a cgroup's usage of various kinds of resources. Cgroups are flexible in that they do not prescribe in advance the exact resources to track. Thus, any resource that can be tracked and restricted can be added. By attaching a resource controller (sometimes referred to as a "subsystem") for a particular resource to a cgroup, it will monitor and/or limit the corresponding resource access for all processes that are a member of that cgroup.

It is possible to limit the usage of one set of resources (such as memory, CPU, and the bandwidth for block I/O) for process P_1 and another set (for instance, only block I/O bandwidth) for process P_2. To do so, we first create two cgroups $C_{CPU+Mem}$ and C_{Blkio}. We then attach the CPU and memory controller to the first cgroup, and the block I/O controller to the second. Finally, we make P_1 a member of both $C_{CPU+Mem}$ and C_{Blkio}, and P_2 a member of only C_{Blkio}.

This in itself is still not enough. Often, we do not want to control the CPU usage of P_1 and P_2 *together,* but rather isolate the CPU usage of P_1 and P_2 *individually* also. As we shall see, cgroups elegantly solve this problem.

It is important to realize that resource control is often possible at different levels of granularity. Take the CPU. At a fine granularity, we can divide the CPU time on a core among processes dynamically using scheduling. We have discussed scheduling at length in Chap. 2. At a much coarser granularity, we can simply divide the cores of a computer, restricting the processes in a cgroup to, say, 4 of the 16 cores. Whatever they do, their processes will not run on the other 12 cores.

While fine-grained CPU control can also be used, core partitioning is actually very popular in practice. The mechanism goes back to the first years of this millennium.

Back in 2004, programmers at Bull SA (the French company that distributed Multics from 1975 until 2000) came up with the notion of **cpusets**. Cpusets allow administrators to associate specific CPUs or cores and subsets of memory to a group of processes. Since then, cpusets have been extended and modified by different programmers from different organizations, among them Paul Menage at Google who also played a leading role in the development of cgroups. This is no coincidence: cpusets match the controller model of cgroups to a t, allowing them to limit a cgroup's usage of CPU and memory at a coarse granularity. In particular, cpusets allow one to assign to a cgroup a set of CPUs and memory nodes—where a memory node simply refers to a node that contains memory, for instance in a NUMA system. Thus, administrators can specify that this cgroup may use these CPU cores and that all its memory will be allocated from the memory at these nodes only. Coarse, but effective!

Moreover, cpusets are hierarchical. In other words, it is possible to subpartition the resources in a parent cpuset into child cpusets. Thus, the root cpuset contains all CPUs and all memory nodes, all level-1 cpusets are subpartitions of its resources, all level-2 cpusets are subpartitions of their level-1 cpusets, and so on. Cgroups are similarly hierarchical. Given the example above, we can create two child cgroups in the parent cgroup $C_{CPU+Mem}$. By attaching different subpartitions of the cpuset with each child cgroup, administrators can ensure that different groups of process keep out of each other's hair.

Using concepts such as cgroups, cpusets, and name spaces, OS-level virtualization allows the creation of isolated containers without resorting to hypervisors or hardware virtualization. These containers have been around for many years, but they really started taking off after the introduction of convenient platforms such as Docker, Kubernetes, and Microsoft Azure Container Registry that help administrators to build, deploy, and manage them.

Compared to hypervisor-based virtual machines, containers are generally more lightweight: faster to start and more efficient in resource consumption. They have other advantages also. For instance, system administration is easier if we need to maintain only a single operating system rather than a separate operating system per virtual machine.

However, there are downsides also. First, you cannot run multiple operating systems on the same machine. If you want to run Windows and UNIX at the same time, containers will not do you much good. Second, while the isolation is pretty good, it is by no means absolute, as the different containers still share the same operating system and may interfere with each other there. If the operating system has static limits on certain resources (such as the number of open files) and one container uses (almost) all of them, the other containers will be in trouble. Similarly, a single vulnerability in the operating system endangers all the containers. In comparison, the isolation offered by hypervisors is considerably stronger. Also,

some researchers argue that hypervisor-based virtualization need not be more heavyweight than containers at all, as long as you reduce the virtual machines to a unikernel (Manco et al., 2017).

7.11 CASE STUDY: VMWARE

Since 1999, VMware, Inc. has been the leading commercial provider of hypervisor-based virtualization solutions with products for desktops, servers, the cloud, and now even on cell phones. It provides not only hypervisors but also the software that manages virtual machines on a large scale.

We will start this case study with a brief history of how the company got started. We will then describe VMware Workstation, a type 2 hypervisor and the company's first product, the challenges in its design and the key elements of the solution. We then describe the evolution of VMware Workstation over the years. We conclude with a description of ESX Server, VMware's type 1 hypervisor.

7.11.1 The Early History of VMware

Although the idea of using virtual machines was popular in the 1960s and 1970s in both the computing industry and academic research, interest in virtualization was totally lost after the 1980s and the rise of the personal computer industry. Only IBM's mainframe division still cared about virtualization. Indeed, the computer architectures designed at the time, and in particular Intel's x86 architecture, did not provide architectural support for virtualization (i.e., they failed the Popek/Goldberg criteria). This is extremely unfortunate, since the 386 CPU, a complete redesign of the 286, was done a decade after the Popek-Goldberg paper, and the designers should have known better.

In 1997, at Stanford, three of the future founders of VMware had built a prototype hypervisor called Disco (Bugnion et al., 1997), with the goal of running commodity operating systems (in particular UNIX) on a very large scale multiprocessor then being developed at Stanford: the FLASH machine. During that project, the authors realized that using virtual machines could solve, simply and elegantly, a number of hard system software problems: rather than trying to solve these problems within existing operating systems, one could innovate in a layer **below** existing operating systems. The key observation of Disco was that, while the high complexity of modern operating systems made innovation difficult, the relative simplicity of a virtual machine monitor and its position in the software stack provided a powerful foothold to address limitations of operating systems. Although Disco was aimed at very large servers, and designed for the MIPS architecture, the authors realized that the same approach could equally apply, and be commercially relevant, for the x86 marketplace.

And so, VMware, Inc. was founded in 1998 with the specific goal of bringing virtualization to the x86 architecture and the personal computer industry. VMware's first product (VMware Workstation) was the first virtualization solution available for 32-bit x86-based platforms. The product was first released in 1999, and came in two variants: **VMware Workstation for Linux**, a type 2 hypervisor that ran on top of Linux host operating systems, and **VMware Workstation for Windows**, which similarly ran on top of Windows NT. Both variants had identical functionality: users could create multiple virtual machines by specifying first the characteristics of the virtual hardware (such as how much memory to give the virtual machine, or the size of the virtual disk) and could then install the operating system of their choice within the virtual machine, typically from the (virtual) CD-ROM.

VMware Workstation was largely aimed at developers and IT professionals. Before the introduction of virtualization, a developer routinely had two computers on his desk, a stable one for development and a second one where he could reinstall the system software as needed. With virtualization, the second test system became a virtual machine.

Soon, VMware started developing a second and more complex product, which would be released as ESX Server in 2001. ESX Server leveraged the same virtualization engine as VMware Workstation, but packaged it as part of a type 1 hypervisor. In other words, ESX Server ran directly on the hardware without requiring a host operating system. The ESX hypervisor was designed for intense workload consolidation and contained many optimizations to ensure that all resources (CPU, memory, and I/O) were efficiently and fairly allocated among the virtual machines. For example, it was the first to introduce the concept of ballooning to rebalance memory between virtual machines (Waldspurger, 2002).

ESX Server was aimed at the server consolidation market. Before the introduction of virtualization, IT administrators would typically buy, install, and configure a new server for every new task or application that they had to run in the data center. The result was that the infrastructure was very inefficiently utilized: servers at the time were typically used at 10% of their capacity (during peaks). With ESX Server, IT administrators could consolidate many independent virtual machines into a single server, saving time, money, rack space, and electrical power.

In 2002, VMware introduced its first management solution for ESX Server, originally called Virtual Center, and today called vSphere. It provided a single point of management for a cluster of servers running virtual machines: an IT administrator could now simply log into the Virtual Center application and control, monitor, or provision thousands of virtual machines running throughout the enterprise. With Virtual Center came another innovation, **VMotion** (Nelson et al., 2005), which allowed the live migration of a running virtual machine over the network. For the first time, an IT administrator could move a running computer from one location to another without having to reboot the operating system, restart applications, or even lose network connections.

7.11.2 VMware Workstation

VMware Workstation was the first virtualization product for 32-bit x86 computers. The subsequent adoption of virtualization had a profound impact on the industry and on the computer science community: in 2009, the ACM awarded its authors the **ACM Software System Award** for VMware Workstation 1.0 for Linux. The original VMware Workstation is described in a detailed technical article (Bugnion et al., 2012). Here we provide a summary of that paper.

The idea was that a virtualization layer could be useful on commodity platforms built from x86 CPUs and primarily running the Microsoft Windows operating systems (a.k.a. the **WinTel** platform). The benefits of virtualization could help address some of the known limitations of the WinTel platform, such as application interoperability, operating system migration, reliability, and security. In addition, virtualization could easily enable the coexistence of operating system alternatives, in particular, Linux.

Although there existed decades' worth of research and commercial development of virtualization technology on mainframes, the x86 computing environment was sufficiently different that new approaches were necessary. For example, mainframes were **vertically integrated**, meaning that a single vendor engineered the hardware, the hypervisor, the operating systems, and most of the applications.

In contrast, the x86 industry was (and still is) disaggregated into at least four different categories: (a) Intel and AMD make the processors; (b) Microsoft offers Windows and the open source community offers Linux; (c) a third group of companies builds the I/O devices and peripherals and their corresponding device drivers; and (d) a fourth group of system integrators such as HP and Dell put together computer systems for retail sale. For the x86 platform, virtualization would first need to be inserted without the support of any of these industry players.

Because this disaggregation was a fact of life, VMware Workstation differed from classic virtual machine monitors that were designed as part of single-vendor architectures with explicit support for virtualization. Instead, VMware Workstation was designed for the x86 architecture and the industry built around it. VMware Workstation addressed these new challenges by combining well-known virtualiza4G tion techniques, techniques from other domains, and new techniques into a single solution.

We now discuss the specific technical challenges in building VMware Workstation.

7.11.3 Challenges in Bringing Virtualization to the x86

Recall our definition of hypervisors and virtual machines: hypervisors apply the well-known principle of **adding a level of indirection** to the domain of computer hardware. They provide the abstraction of virtual machines: multiple copies of the raw underlying hardware, each running an independent operating system

instance. The virtual machines are isolated from other virtual machines, appear each as a duplicate of the underlying hardware, and ideally run with the same speed as the real machine. VMware adapted these core attributes of a virtual machine to an x86-based target platform as follows:

1. **Compatibility**. The notion of an "essentially identical environment" meant that any x86 operating system, and all of its applications, would be able to run without modifications as a virtual machine. A hypervisor needed to provide sufficient compatibility at the hardware level such that users could run whichever operating system (down to the update and patch version) they wished to install within a particular virtual machine, without restrictions.
2. **Performance**. The overhead of the hypervisor had to be sufficiently low that users could use a virtual machine as their primary work environment. As a goal, the designers of VMware aimed to run relevant workloads at near native speeds, and in the worst case to run them on then-current processors with the same performance as if they were running natively on the immediately prior generation of processors. This was based on the observation that most x86 software was not designed to run only on the latest generation of CPUs.
3. **Isolation**. A hypervisor had to guarantee the isolation of the virtual machine without making any assumptions about the software running inside. That is, a hypervisor needed to be in complete control of resources. Software running inside virtual machines had to be prevented from any access that would allow it to subvert the hypervisor. Similarly, a hypervisor had to ensure the privacy of all data not belonging to the virtual machine. A hypervisor had to assume that the guest operating system could be infected with unknown, malicious code (a much bigger concern today than during the mainframe era).

There was an inevitable tension between these three requirements. For example, total compatibility in certain areas might lead to a prohibitive impact on performance, in which case VMware's designers had to compromise. However, they ruled out any trade-offs that might compromise isolation or expose the hypervisor to attacks by a malicious guest. Overall, four major challenges emerged:

1. **The x86 architecture was not virtualizable**. It had virtualization-sensitive, nonprivileged instructions, which violated the Popek and Goldberg criteria for strict virtualization. For example, the POPF instruction has a different (yet nontrapping) semantics depending on whether the currently running software is allowed to disable interrupts or not. This ruled out the traditional trap-and-emulate approach to virtualization. Even engineers from Intel Corporation were convinced their processors could not be virtualized in any practical sense.

2. **The x86 architecture was of daunting complexity**. The x86 architecture was a notoriously complicated CISC architecture, including legacy support for multiple decades of backward compatibility. Over the years, it had introduced four main modes of operations (real, protected, v8086, and system management), each of which enabled in different ways the hardware's segmentation model, paging mechanisms, protection rings, and security features (such as call gates).
3. **x86 machines had diverse peripherals**. Although there were only two major x86 processor vendors, the personal computers of the time could contain an enormous variety of add-in cards and devices, each with their own vendor-specific device drivers. Virtualizing all these peripherals was infeasible. This had two implications: it applied to both the front end (the virtual hardware exposed in the virtual machines) and the back end (the real hardware that the hypervisor needed to be able to control) of peripherals.
4. **Need for a simple user experience**. Classic hypervisors were installed in the factory, similar to the firmware found in today's computers. Since VMware was a startup, its users would have to add the hypervisors to existing systems after the fact. VMware needed a software delivery model with an easy installation procedure to speed-up adoption.

7.11.4 VMware Workstation: Solution Overview

This section describes at a high level how VMware Workstation addressed the challenges mentioned in the previous section.

VMware Workstation is a type 2 hypervisor that consists of distinct modules. One important module is the VMM, which is responsible for executing the virtual machine's instructions. A second important module is the VMX, which interacts with the host operating system.

The section covers first how the VMM solves the nonvirtualizability of the x86 architecture. Then, we describe the operating system-centric strategy used by the designers throughout the development phase. Next, we look at the design of the virtual hardware platform, which addresses half of the peripheral diversity challenge. Finally, we discuss the role of the host operating system, in particular the interaction between the VMM and VMX components.

Virtualizing the x86 Architecture

The VMM runs the actual virtual machine; it enables it to make forward progress. A VMM built for a virtualizable architecture uses a technique known as trap-and-emulate to execute the virtual machine's instruction sequence directly, but

safely, on the hardware. When this is not possible, one approach is to specify a virtualizable subset of the processor architecture, and port the guest operating systems to that newly defined platform. This technique is known as paravirtualization (Barham et al., 2003; Whitaker et al., 2002) and requires source-code level modifications of the operating system. Put bluntly, paravirtualization modifies the guest to avoid doing anything that the hypervisor cannot handle. Paravirtualization was infeasible at VMware because of the compatibility requirement and the need to run operating systems whose source code was not available, in particular Windows.

An alternative would have been to employ an all-emulation approach. In this, the instructions of the virtual machines are emulated by the VMM on the hardware (rather than directly executed). This can be quite efficient; prior experience with the SimOS (Rosenblum et al., 1997) machine simulator showed that the use of techniques such as **dynamic binary translation** running in a user-level program could limit overhead of complete emulation to a factor-of-five slowdown. Although this is *quite* efficient, and certainly useful for simulation purposes, a factor-of-five slowdown was clearly inadequate and would not meet the desired performance requirements.

The solution to this problem combined two key insights. First, although trap-and-emulate direct execution could not be used to virtualize the entire x86 architecture all the time, it could actually be used some of the time. In particular, it could be used during the execution of application programs, which accounted for most of the execution time on relevant workloads. The reasons is that these virtualization sensitive instructions are not sensitive all the time; rather they are sensitive only in certain circumstances. For example, the POPF instruction is virtualization-sensitive when the software is expected to be able to disable interrupts (e.g., when running the operating system), but is not virtualization-sensitive when software cannot disable interrupts (in practice, when running nearly all user-level applications).

Figure 7-8 shows the modular building blocks of the original VMware VMM. We see that it consists of a direct-execution subsystem, a binary translation subsystem, and a decision algorithm to determine which subsystem should be used. Both subsystems rely on some shared modules, for example to virtualize memory through shadow page tables, or to emulate I/O devices.

The direct-execution subsystem is preferred, and the dynamic binary translation subsystem provides a fallback mechanism whenever direct execution is not possible. This is the case for example whenever the virtual machine is in such a state that it could issue a virtualization-sensitive instruction. Therefore, each subsystem constantly reevaluates the decision algorithm to determine whether a switch of subsystems is possible (from binary translation to direct execution) or necessary (from direct execution to binary translation). This algorithm has a number of input parameters, such as the current execution ring of the virtual machine, whether interrupts can be enabled at that level, and the state of the segments. For example, binary translation must be used if any of the following is true:

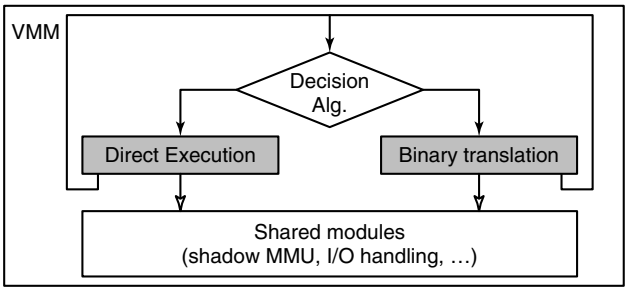

Figure 7-8. High-level components of the VMware virtual machine monitor (in the absence of hardware support).

1. The virtual machine is currently running in kernel mode (ring 0 in the x86 architecture).
2. The virtual machine can disable interrupts and issue I/O instructions (in the x86 architecture, when the I/O privilege level is set to the ring level).
3. The virtual machine is currently running in real mode, a legacy 16-bit execution mode used by the BIOS among other things.

The actual decision algorithm contains a few additional conditions. The details can be found in Bugnion et al. (2012). Interestingly, the algorithm does not depend on the instructions that are stored in memory and may be executed, but only on the value of a few virtual registers; therefore, it can be evaluated very efficiently in just a handful of instructions.

The second key insight was that by properly configuring the hardware, particularly using the x86 segment protection mechanisms carefully, system code under dynamic binary translation could also run at near-native speeds. This is very different than the factor-of-five slowdown normally expected of machine simulators.

The difference can be explained by comparing how a dynamic binary translator converts a simple instruction that accesses memory. To emulate such an instruction in software, a classic binary translator emulating the full x86 instruction-set architecture would have to first verify whether the effective address is within the range of the data segment, then convert the address into a physical address, and finally to copy the referenced word into the simulated register. Of course, these various steps can be optimized through caching, in a way very similar to how the processor cached page-table mappings in a translation-lookaside buffer. But even such optimizations would lead to an expansion of individual instructions into an instruction sequence.

The VMware binary translator performs *none* of these steps in software. Instead, it configures the hardware so that this simple instruction can be reissued

with the identical instruction. This is possible only because the VMware VMM (of which the binary translator is a component) has previously configured the hardware to match the exact specification of the virtual machine: (a) the VMM uses shadow page tables, which ensures that the memory management unit can be used directly (rather than emulated) and (b) the VMM uses a similar shadowing approach to the segment descriptor tables (which played a big role in the 16-bit and 32-bit software running on older x86 operating systems).

There are, of course, complications and subtleties. One important aspect of the design is to ensure the integrity of the virtualization sandbox, that is, to ensure that no software running inside the virtual machine (including malicious software) can tamper with the VMM. This problem is generally known as **software fault isolation** and adds run-time overhead to each memory access if the solution is implemented in software. Here also, the VMware VMM uses a different, hardware-based approach. It splits the address space into two disjoint zones. The VMM reserves for its own use the top 4 MB of the address space. This frees up the rest (that is, 4 *GB* – 4 MB, since we are talking about a 32-bit architecture) for the use by the virtual machine. The VMM then configures the segmentation hardware so that no virtual machine instructions (including ones generated by the binary translator) can ever access the top 4-MB region of the address space.

A Guest Operating System Centric Strategy

Ideally, a VMM should be designed without worrying about the guest operating system running in the virtual machine, or how that guest operating system configures the hardware. The idea behind virtualization is to make the virtual machine interface identical to the hardware interface so that all software that runs on the hardware will also run in a virtual machine. Unfortunately, this approach is practical only when the architecture is virtualizable and simple. In the case of x86, the overwhelming complexity of the architecture was clearly a problem.

The VMware engineers simplified the problem by focusing only on a selection of supported guest operating systems. In its first release, VMware Workstation supported officially only Linux, Windows 3.1, Windows 95/98, and Windows NT as guest operating systems. Over the years, new operating systems were added to the list with each revision of the software. Nevertheless, the emulation was good enough that it ran some unexpected operating systems, such as MINIX 3, perfectly, right out of the box.

This simplification did not change the overall design—the VMM still provided a faithful copy of the underlying hardware, but it helped guide the development process. In particular, engineers had to worry only about combinations of features that were used in practice by the supported guest operating systems.

For example, the x86 architecture contains four privilege rings in protected mode (ring 0 to ring 3) but no operating system uses ring 1 or ring 2 in practice (save for OS/2, a long-dead operating system from IBM). So rather than figure out

how to correctly virtualize ring 1 and ring 2, the VMware VMM simply had code to detect if a guest was trying to enter into ring 1 or ring 2, and, in that case, would stop execution of the virtual machine. This not only removed unnecessary code, but more importantly it allowed the VMware VMM to assume that ring 1 and ring 2 would never be used by the virtual machine, and therefore that it could use these rings for its own purposes. In fact, the VMware VMM's binary translator runs at ring 1 to virtualize ring 0 code.

The Virtual Hardware Platform

So far, we have primarily discussed the problem associated with the virtualization of the x86 processor. But an x86-based computer is much more than its processor. It also has a chipset, some firmware, and a set of I/O peripherals to control disks, network cards, CD-ROM, keyboard, etc.

The diversity of I/O peripherals in x86 personal computers made it impossible to match the virtual hardware to the real, underlying hardware. Whereas there were only a handful of x86 processor models in the market, with only minor variations in instruction-set level capabilities, there were thousands of I/O devices, most of which had no publicly available documentation of their interface or functionality. VMware's key insight was to **not** attempt to have the virtual hardware match the specific underlying hardware, but instead have it always match some configuration composed of selected, canonical I/O devices. Guest operating systems then used their own existing, built-in mechanisms to detect and operate these (virtual) devices.

The virtualization platform consisted of a combination of multiplexed and emulated components. Multiplexing meant configuring the hardware so it can be directly used by the virtual machine, and shared (in space or time) across multiple virtual machines. Emulation meant exporting a software simulation of the selected, canonical hardware component to the virtual machine. Figure 7-9 illustrates that VMware Workstation used multiplexing for processor and memory and emulation for everything else.

For the multiplexed hardware, each virtual machine had the illusion of having one dedicated CPU and a configurable, but a fixed amount of contiguous RAM starting at physical address 0.

Architecturally, the emulation of each virtual device was split between a front-end component, which was visible to the virtual machine, and a back-end component, which interacted with the host operating system (Waldspurger and Rosenblum, 2012). The front-end was essentially a software model of the hardware device that could be controlled by unmodified device drivers running inside the virtual machine. Regardless of the specific corresponding physical hardware on the host, the front end always exposed the same device model.

For example, the first Ethernet device front end was the AMD PCnet "Lance" chip, once a popular 10-Mbps plug-in board on PCs, and the back end provided

	Virtual Hardware (front end)	Back end
Multiplexed	1 virtual x86 CPU, with the same instruction set extensions as the underlying hardware CPU	Scheduled by the host operating system on either a uniprocessor or multiprocessor host
	Up to 512 MB of contiguous DRAM	Allocated and managed by the host OS (page-by-page)

Emulated	PCI Bus	Fully emulated compliant PCI bus
	4x IDE disks 7x Buslogic SCSI Disks	Virtual disks (stored as files) or direct access to a given raw device
	1x IDE CD-ROM	ISO image or emulated access to the real CD-ROM
	2x 1.44 MB floppy drives	Physical floppy or floppy image
	1x VMware graphics card with VGA and SVGA support	Ran in a window and in full-screen mode. SVGA required VMware SVGA guest driver
	2x serial ports COM1 and COM2	Connect to host serial port or a file
	1x printer (LPT)	Can connect to host LPT port
	1x keyboard (104-key)	Fully emulated; keycode events are generated when they are received by the VMware application
	1x PS-2 mouse	Same as keyboard
	3x AMD Lance Ethernet cards	Bridge mode and host-only modes
	1x Soundblaster	Fully emulated

Figure 7-9. Virtual hardware configuration options of the early VMware Workstation, ca. 2000.

network connectivity to the host's physical network. Ironically, VMware kept supporting the PCnet device long after physical Lance boards were no longer available, and actually achieved I/O that was orders of magnitude faster than 10 Mbps (Sugerman et al., 2001). For storage devices, the original front ends were an IDE controller and a Buslogic Controller, and the back end was typically either a file in the host file system, such as a virtual disk or an ISO 9660 image, or a raw resource such as a drive partition or the physical CD-ROM.

Splitting front ends from back ends had another benefit: a VMware virtual machine could be copied from computer to another computer, possibly with different hardware devices. Yet, the virtual machine would not have to install new device drivers since it only interacted with the front-end component. This attribute, called **hardware-independent encapsulation**, has a huge benefit today in server environments and in cloud computing. It enabled subsequent innovations such as suspend/resume, checkpointing, and the transparent migration of live virtual machines across physical boundaries (Nelson et al., 2005). In the cloud, it allows

customers to deploy their virtual machines on any available server, without having to worry of the details of the underlying hardware.

The Role of the Host Operating System

The final critical design decision in VMware Workstation was to deploy it "on top" of an existing operating system. This classifies it as a type 2 hypervisor. The choice had two main benefits.

First, it would address the second part of peripheral diversity challenge. VMware implemented the front-end emulation of the various devices, but relied on the device drivers of the host operating system for the back end. For example, VMware Workstation would read or write a file in the host file system to emulate a virtual disk device, or draw in a window of the host's desktop to emulate a video card. As long as the host operating system had the appropriate drivers, VMware Workstation could run virtual machines on top of it.

Second, the product could install and feel like a normal application to a user, making adoption easier. Like any application, the VMware Workstation installer simply writes its component files onto an existing host file system, without perturbing the hardware configuration (no reformatting of a disk, creating of a disk partition, or changing of BIOS settings). In fact, VMware Workstation could be installed and start running virtual machines without requiring even rebooting the host operating system, at least on Linux hosts.

However, a normal application does not have the necessary hooks and APIs necessary for a hypervisor to multiplex the CPU and memory resources, which is essential to provide near-native performance. In particular, the core x86 virtualization technology described above works only when the VMM runs in kernel mode and can furthermore control all aspects of the processor without any restrictions. This includes the ability to change the address space (to create shadow page tables), to change the segment tables, and to change all interrupt and exception handlers.

A device driver has more direct access to the hardware, in particular if it runs in kernel mode. Although it could (in theory) issue any privileged instructions, in practice a device driver is expected to interact with its operating system using well-defined APIs, and does not (and should never) arbitrarily reconfigure the hardware. And since hypervisors call for a massive reconfiguration of the hardware (including the entire address space, segment tables, exception and interrupt handlers), running the hypervisor as a device driver was also not a realistic option.

Since none of these assumptions are supported by host operating systems, running the hypervisor as a device driver (in kernel mode) was also not an option.

These stringent requirements led to the development of the VMware Hosted Architecture. In it, as shown in Fig. 7-10, the software is broken into three separate and distinct components.

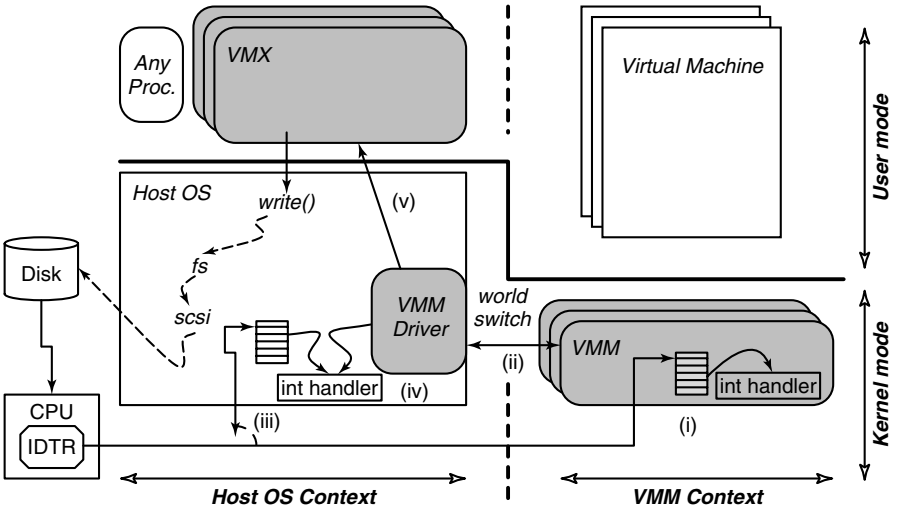

Figure 7-10. The VMware Hosted Architecture and its three components: VMX, VMM driver, and VMM.

These components each have different functions and operate independently from one another:

1. A user-space program (the **VMX**) which the user perceives to be the VMware program. The VMX performs all UI functions, starts the virtual machine, and then performs most of the device emulation (front end), and makes regular system calls to the host operating system for the back end interactions. There is typically one multithreaded VMX process per virtual machine.

2. A small kernel-mode device driver (the **VMX driver**), which gets installed within the host operating system. It is used primarily to allow the VMM to run by temporarily suspending the entire host operating system. There is one VMX driver installed in the host operating system, typically at boot time.

3. The VMM, which includes all the software necessary to multiplex the CPU and the memory, including the exception handlers, the trap-and-emulate handlers, the binary translator, and the shadow paging module. The VMM runs in kernel mode, but it does not run in the context of the host operating system. In other words, it cannot rely directly on services offered by the host operating system, but it is also not constrained by any rules or conventions imposed by the host operating system. There is one VMM instance for each virtual machine, created when the virtual machine starts.

VMware Workstation appears to run on top of an existing operating system, and, in fact, its VMX does run as a process of that operating system. However, the VMM operates at system level, in full control of the hardware, and without depending on any way on the host operating system. Figure 7-10 shows the relationship between the entities: the two contexts (host operating system and VMM) are peers to each other, and each has a user-level and a kernel component. When the VMM runs (the right half of the figure), it reconfigures the hardware, handles all I/O interrupts and exceptions, and can therefore safely temporarily remove the host operating system from its virtual memory. For example, the location of the interrupt table is set within the VMM by assigning the IDTR register to a new address. Conversely, when the host operating system runs (the left half of the figure), the VMM and its virtual machine are equally removed from its virtual memory.

This transition between these two totally independent system-level contexts is a world switch. The name itself emphasizes that everything about the software changes during a world switch, in contrast with the regular context switch implemented by an operating system. Figure 7-11 shows the difference between the two. The regular context switch between processes "A" and "B" swaps the user portion of the address space and the registers of the two processes, but leaves a number of critical system resources unmodified. For example, the kernel portion of the address space is identical for all processes, and the exception handlers are also not modified. In contrast, the world switch changes everything: the entire address space, all exception handlers, privileged registers, etc. In particular, the kernel address space of the host operating system is mapped only when running in the host operating system context. After the world switch into the VMM context, it has been removed from the address space altogether, freeing space to run both the VMM and the virtual machine. Although this sounds complicated, this can be implemented quite efficiently and takes only 45 x86 machine-language instructions to execute.

The careful reader will have wondered: what of the guest operating system's kernel address space? The answer is simply that it is part of the virtual machine address space, and is present when running in the VMM context. Therefore, the guest operating system can use the entire address space, and in particular the same locations in virtual memory as the host operating system. This is very specifically what happens when the host and guest operating systems are the same (e.g., both are Linux). Of course, this all "just works" because of the two independent contexts and the world switch between the two.

The same reader will then wonder: what of the VMM area, at the very top of the address space? As we discussed above, it is reserved for the VMM itself, and those portions of the address space cannot be directly used by the virtual machine. Luckily, that small 4-MB portion is not frequently used by the guest operating systems since each access to that portion of memory must be individually emulated and induces noticeable software overhead.

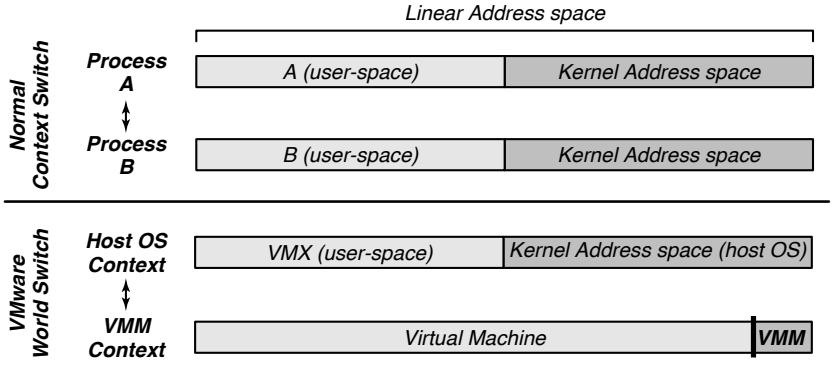

Figure 7-11. Difference between a normal context switch and a world switch.

Going back to Fig. 7-10: it further illustrates the various steps that occur when a disk interrupt happens while the VMM is executing (step i). Of course, the VMM cannot handle the interrupt since it does not have the back-end device driver. In (ii), the VMM does a world switch back to the host operating system. Specifically, the world-switch code returns control to the VMware driver, which in (iii) emulates the same interrupt that was issued by the disk. So in step (iv), the interrupt handler of the host operating system runs through its logic, as if the disk interrupt had occurred while the VMware driver (but not the VMM!) was running. Finally, in step (v), the VMware driver returns control to the VMX application. At this point, the host operating system may choose to schedule another process, or keep running the VMware VMX process. If the VMX process keeps running, it will then resume execution of the virtual machine by doing a special call into the device driver, which will generate a world switch back into the VMM context. As you see, this is a neat trick that hides the entire VMM and virtual machine from the host operating system. More importantly, it provides the VMM complete freedom to reprogram the hardware as it sees fit.

7.11.5 The Evolution of VMware Workstation

The technology landscape has changed dramatically in the decade following the development of the original VMware Virtual Machine Monitor.

The hosted architecture is still used today for state-of-the-art interactive hypervisors such as VMware Workstation, VMware Player, and VMware Fusion (the product aimed at Apple macOS host operating systems), and even in VMware's product aimed at cell phones (Barr et al., 2010). The world switch, and its ability to separate the host operating system context from the VMM context, remains the foundational mechanism of VMware's hosted products today. Although the implementation of the world switch has evolved through the years, for example, to

support 64-bit systems, the fundamental idea of having totally separate address spaces for the host operating system and the VMM remains valid today.

In contrast, the approach to the virtualization of the x86 architecture changed rather dramatically with the introduction of hardware-assisted virtualization. Hardware-assisted virtualizations, such as Intel VT-x and AMD-v were introduced in two phases. The first phase, starting in 2005, was designed with the explicit purpose of eliminating the need for either paravirtualization or binary translation (Uhlig et al., 2005). Starting in 2007, the second phase provided hardware support in the MMU in the form of nested page tables. This eliminated the need to maintain shadow page tables in software. Today, VMware's hypervisors mostly uses a hardware-based, trap-and-emulate approach (as formalized by Popek and Goldberg four decades earlier) whenever the processor supports both virtualization and nested page tables.

The emergence of hardware support for virtualization had a significant impact on VMware's guest operating system centric-strategy. In the original VMware Workstation, the strategy was used to dramatically reduce implementation complexity at the expense of compatibility with the full architecture. Today, full architectural compatibility is expected due to hardware support. The current VMware guest operating system-centric strategy focuses on performance optimizations for selected guest operating systems.

7.11.6 ESX Server: VMware's type 1 Hypervisor

In 2001, VMware released a different product, called ESX Server, aimed at the server marketplace. Here, VMware's engineers took a different approach: rather than creating a type 2 solution running on top of a host operating system, they decided to build a type 1 solution that would run directly on the hardware.

Figure 7-12 shows the high-level architecture of ESX Server. It combines an existing component, the VMM, with a true hypervisor running directly on the bare metal. The VMM performs the same function as in VMware Workstation, which is to run the virtual machine in an isolated environment that is a duplicate of the x86 architecture. As a matter of fact, the VMMs used in the two products use the same source code base, and they are largely identical. The ESX hypervisor replaces the host operating system. But rather than implementing the full functionality expected of an operating system, its only goal is to run the various VMM instances and to efficiently manage the physical resources of the machine. ESX Server therefore contains the usual subsystem found in an operating system, such as a CPU scheduler, a memory manager, and an I/O subsystem, with each subsystem optimized to run virtual machines.

The absence of a host operating system required VMware to directly address the issues of peripheral diversity and user experience described earlier. For peripheral diversity, VMware restricted ESX Server to run only on well-known and certified server platforms, for which it had device drivers. As for the user experience,

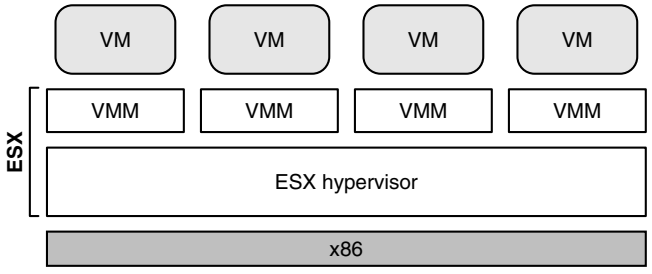

Figure 7-12. ESX Server: VMware's type 1 hypervisor.

ESX Server (unlike VMware Workstation) required users to install a new system image on a boot partition.

Despite the drawbacks, the trade-off made sense for dedicated deployments of virtualization in data centers, consisting of hundreds or thousands of physical servers, and often (many) thousands of virtual machines. Such deployments are sometimes referred today as private clouds. There, the ESX Server architecture provides substantial benefits in terms of performance, scalability, manageability, and features. For example:

1. The CPU scheduler ensures that each virtual machine gets a fair share of the CPU (to avoid starvation). It is also designed so that the different virtual CPUs of a given multiprocessor virtual machine are scheduled at the same time.

2. The memory manager is optimized for scalability, in particular to run virtual machines efficiently even when they need more memory than is actually available on the computer. To achieve this result, ESX Server first introduced the notion of ballooning and transparent page sharing for virtual machines (Waldspurger, 2002).

3. The I/O subsystem is optimized for performance. Although VMware Workstation and ESX Server often share the same front-end emulation components, the back ends are totally different. In the VMware Workstation case, all I/O flows through the host operating system and its API, which often adds overhead. This is particularly true in the case of networking and storage devices. With ESX Server, these device drivers run directly within the ESX hypervisor, without requiring a world switch.

4. The back ends also typically relied on abstractions provided by the host operating system. For example, VMware Workstation stores virtual machine images as regular (but very large) files on the host file system. In contrast, ESX Server has VMFS (Vaghani, 2010), a file

system optimized specifically to store virtual machine images and ensure high I/O throughput. This allows for extreme levels of performance. For example, VMware demonstrated back in 2011 that a single ESX Server could issue 1 million disk operations per second (VMware, 2011).

5. ESX Server made it easy to introduce new capabilities, which required the tight coordination and specific configuration of multiple components of a computer. For example, ESX Server introduced VMotion, the first virtualization solution that could migrate a live virtual machine from one machine running ESX Server to another machine running ESX Server, while it was running. This achievement required the coordination of the memory manager, the CPU scheduler, and the networking stack.

Over the years, new features were added to ESX Server. ESX Server evolved into ESXi, a small-footprint alternative that is sufficiently small in size to be preinstalled in the firmware of servers. Today, ESXi is VMware's most important product and serves as the foundation of the vSphere suite.

7.12 RESEARCH ON VIRTUALIZATION AND THE CLOUD

Virtualization technology and cloud computing are both extremely active research areas. The research produced in these fields is way too much to enumerate. Each has multiple research conferences. For instance, the Virtual Execution Environments (VEE) conference focuses on virtualization in the broadest sense. You will find papers on migration deduplication, scaling out, and so on. Likewise, the ACM Symposium on Cloud Computing (SOCC) is one of the best-known venues on cloud computing. Papers in SOCC include work on fault resilience, scheduling of data center workloads, management and debugging in clouds.

Hardware support for virtualization is now present in nearly all relevant CPU architectures, restoring the Popek and Goldberg architectural principles into practice. Notably, ARM added a new privilege level "EL2" to support hardware virtualization in ARMv8. In mobile platforms, virtualization is often deployed with another hardware feature called TrustZone to allow a "secure" virtual machine to co-exist with the main operating environment (Dall et al., 2016).

Security is perpetually a hot topic for research (Dai et al., 2020; Trach et al., 2020), as is reducing energy usage (Kaffes et al., 2020). With so many data centers now using virtualization technology, the networks connecting these machines are also a major subject of research (Alvarez et al., 2020).

One of the nice things about virtualization hardware is that untrusted code can get direct but safe access to hardware features like page tables, and tagged TLBs. With this in mind, the Dune project (Belay, 2012) did not aim to provide a machine abstraction, but rather a process abstraction. The process is able to enter Dune

mode, an irreversible transition that gives it access to the low-level hardware. Nevertheless, it is still a process and able to talk to and rely on the kernel. The only difference that it uses the VMCALL instruction to make a system call. The Dune approach was later used for research in dataplane operating systems such as IX (Belay, 2017) and finally adapted as the foundation for Google's container solution gVisor (Young, 2019).

Speaking of containers, the cloud is seeing a shift from being a platform for tenants to run virtual machines (specified by a virtual disk image) to a platform used by tenants to run containers specified as Dockerfiles and coordinated by orchestrators such as Kubernetes. Often, the container runs within a virtual machine but the guest operating system is now run by the cloud provider. The latest trend, called "serverless", further decouples the application logic from its environment (Shahrad et al., 2020). The idea here is to allow a service to be automatically spun up to serve a single function (an RPC over HTTPS) without managing any aspect of its operating system environment. Amazon calls its serverless technology firecracker (Barr, 2018) and Google calls it gVisor (Young, 2019). Serverless computing has gained even more popularity with the adoption of the Function-as-a-Service (FAAS) model (Kim and Lee, 2019). Of course, the operating system and indeed the server both still exist in serverless operations. But they are hidden from the developer.

The cloud is therefore much more complex than it was in the beginning with the original AWS EC2 model of virtual machines: networks are virtualized, and applications are encapsulated into containers and serverless models that decouple them from the underlying layers. The cloud is more complex, but also more powerful and central to nearly every computing organization. With this importance comes another consideration: trust. Specifically, how much must a tenant trust the cloud service provider ? The correct answer is obviously "as little as necessary."

To achieve this goal of minimal trust dependencies, Intel introduced SGX as the first architectural extension for "confidential computing." SGX creates enclaves, which are isolated in hardware from the host operating system and other applications, with the content of memory and registers cryptographically encrypted by the hardware. In a way, technologies such as SGX are similar to virtualization extensions: they create new ways for software to isolate itself from each other. SGX has notably been used in research to run entire operating systems such as in Haven (Baumann, 2015) or Linux containers such as in SCONE (Arnautov, 2016).

7.13 SUMMARY

Virtualization is the technique of simulating a computer, but with high performance. Typically one computer runs many virtual machines at the same time. This technique is widely used in data centers to provide cloud computing. In this chapter we looked at how virtualization works, especially for paging, I/O, and multicore systems. We also studied an example: VMWare.

PROBLEMS

1. Give a reason why a data center might be interested in virtualization.
2. Give a reason why a company might be interested in running a hypervisor on a machine that has been in use for a while.
3. Give a reason why a software developer might use virtualization on a desktop machine being used for development.
4. Give a reason why an individual at home might be interested in virtualization. Which type of hypervisor would probably be best for a home user?
5. Why do you think virtualization took so long to become popular? After all, the key paper was written in 1974 and IBM mainframes had the necessary hardware and software throughout the 1970s and beyond.
6. What are the three main requirements for designing hypervisors?
7. Name two kinds of instructions that are sensitive in the Popek and Goldberg sense.
8. Name three machine instructions that are not sensitive in the Popek and Goldberg sense.
9. What is the difference between full virtualization and paravirtualization? Which do you think is harder to do? Explain your answer.
10. Does it make sense to paravirtualize an operating system if the source code is available? What if it is not?
11. Consider a type 1 hypervisor that can support up to n virtual machines at the same time. PCs can have a maximum of four disk primary partitions. Can n be larger than 4? If so, where can the data be stored?
12. Briefly explain the concept of process-level virtualization.
13. Why do type 2 hypervisors exist? After all, there is nothing they can do that type 1 hypervisors cannot do and the type 1 hypervisors are generally more efficient as well.
14. Is virtualization of any use to type 2 hypervisors?
15. Why was binary translation invented? Do you think it has much of a future? Explain your answer.
16. Explain how the x86's four protection rings can be used to support virtualization.
17. State one reason as to why a hardware-based approach using VT-enabled CPUs can perform poorly when compared to translation-based software approaches.
18. Give one case where a translated code can be faster than the original code, in a system using binary translation.
19. VMware does binary translation one basic block at a time, then it executes the block and starts translating the next one. Could it translate the entire program in advance and then execute it? If so, what are the advantages and disadvantages of each technique?
20. What is the difference between a pure hypervisor and a pure microkernel?

21. Briefly explain why memory is so difficult to virtualize well in practice? Explain your answer.

22. Running multiple virtual machines on a PC is known to require large amounts of memory. Why? Can you think of any ways to reduce the memory usage? Explain.

23. Explain the concept of shadow page tables, as used in memory virtualization.

24. One way to handle guest operating systems that change their page tables using ordinary (nonprivileged) instructions is to mark the page tables as read only and take a trap when they are modified. How else could the shadow page tables be maintained? Discuss the efficiency of your approach versus the read-only page tables.

25. Why are balloon drivers used? Is this cheating?

26. Describe a situation in which balloon drivers do not work.

27. Explain the concept of deduplication as used in memory virtualization.

28. Computers have had DMA for doing I/O for decades. Did this cause any problems before there were I/O MMUs?

29. What is a virtual appliance? Why is such a thing useful?

30. PCs differ in minor ways at the very lowest level, things like how timers are managed, how interrupts are handled, and some of the details of DMA. Do these differences mean that virtual appliances are not actually going to work well in practice? Explain your answer.

31. Give one advantage of cloud computing over running your programs locally. Give one disadvantage as well.

32. Give an example of IAAS, PAAS, SAAS, and FAAS.

33. Why is virtual machine migration important? Under what circumstances might it be useful?

34. Migrating virtual machines may be easier than migrating processes, but migration can still be difficult. What problems can arise when migrating a virtual machine?

35. Why is migration of virtual machines from one machine to another easier than migrating processes from one machine to another?

36. What is the difference between live migration and the other kind (dead migration)?

37. What were the three main requirements considered while designing VMware?

38. Why was the enormous number of peripheral devices available not a problem when VMware Workstation was first introduced?

39. VMware ESXi has been made very small. Why? After all, servers at data centers usually have tens of gigabytes of RAM. What difference does a few tens of megabytes more or less make?

40. We have seen that hypervisor-based virtual machines provides better isolation than containers. This is clearly an advantage from a security point of view. However, can you think of any security advantages that containers might have over virtual machines?