

3

This chapter discusses the fundamental concepts of virtualization, its evolution, and various models and technologies used in cloud computing environments.

Since its inception, virtualization has been sporadically explored and adopted, but in the last few years there has been a consistent and growing trend to leverage this technology. Virtualization technologies have gained renewed interested recently due to the confluence of several phenomena:

- *Increased performance and computing capacity.* Nowadays, the average end-user desktop PC is powerful enough to meet almost all the needs of everyday computing, with extra capacity that is rarely used. Almost all these PCs have resources enough to host a virtual machine manager and execute a virtual machine with by far acceptable performance. The same consideration applies to the high-end side of the PC market, where supercomputers can provide immense compute power that can accommodate the execution of hundreds or thousands of virtual machines.
- *Underutilized hardware and software resources.* Hardware and software underutilization is occurring due to (1) increased performance and computing capacity, and (2) the effect of

limited or sporadic use of resources. Computers today are so powerful that in most cases only a fraction of their capacity is used by an application or the system. Moreover, if we consider the IT infrastructure of an enterprise, many computers are only partially utilized whereas they could be used without interruption on a 24/7/365 basis. For example, desktop PCs mostly devoted to office automation tasks and used by administrative staff are only used during work hours, remaining completely unused overnight. Using these resources for other purposes after hours could improve the efficiency of the IT infrastructure. To transparently provide such a service, it would be necessary to deploy a completely separate environment, which can be achieved through virtualization.

- *Lack of space.* The continuous need for additional capacity, whether storage or compute power, makes data centers grow quickly. Companies such as Google and Microsoft expand their infrastructures by building data centers as large as football fields that are able to host thousands of nodes. Although this is viable for IT giants, in most cases enterprises cannot afford to build another data center to accommodate additional resource capacity. This condition, along with hardware underutilization, has led to the diffusion of a technique called *server consolidation*,¹ for which virtualization technologies are fundamental.
- *Greening initiatives.* Recently, companies are increasingly looking for ways to reduce the amount of energy they consume and to reduce their carbon footprint. Data centers are one of the major power consumers; they contribute consistently to the impact that a company has on the environment. Maintaining a data center operation not only involves keeping servers on, but a great deal of energy is also consumed in keeping them cool. Infrastructures for cooling have a significant impact on the carbon footprint of a data center. Hence, reducing the number of servers through server consolidation will definitely reduce the impact of cooling and power consumption of a data center. Virtualization technologies can provide an efficient way of consolidating servers.
- *Rise of administrative costs.* Power consumption and cooling costs have now become higher than the cost of IT equipment. Moreover, the increased demand for additional capacity, which translates into more servers in a data center, is also responsible for a significant increment in administrative costs. Computers—in particular, servers—do not operate all on their own, but they require care and feeding from system administrators. Common system administration tasks include hardware monitoring, defective hardware replacement, server setup and updates, server resources monitoring, and backups. These are labor-intensive operations, and the higher the number of servers that have to be managed, the higher the administrative costs. Virtualization can help reduce the number of required servers for a given workload, thus reducing the cost of the administrative personnel.

These can be considered the major causes for the diffusion of hardware virtualization solutions as well as the other kinds of virtualization. The first step toward consistent adoption of virtualization technologies was made with the wide spread of virtual machine-based programming languages: In 1995 Sun released Java, which soon became popular among developers. The ability to integrate small Java applications, called *applets*, made Java a very successful platform, and with the

¹Server consolidation is a technique for aggregating multiple services and applications originally deployed on different servers on one physical server. Server consolidation allows us to reduce the power consumption of a data center and resolve hardware underutilization.

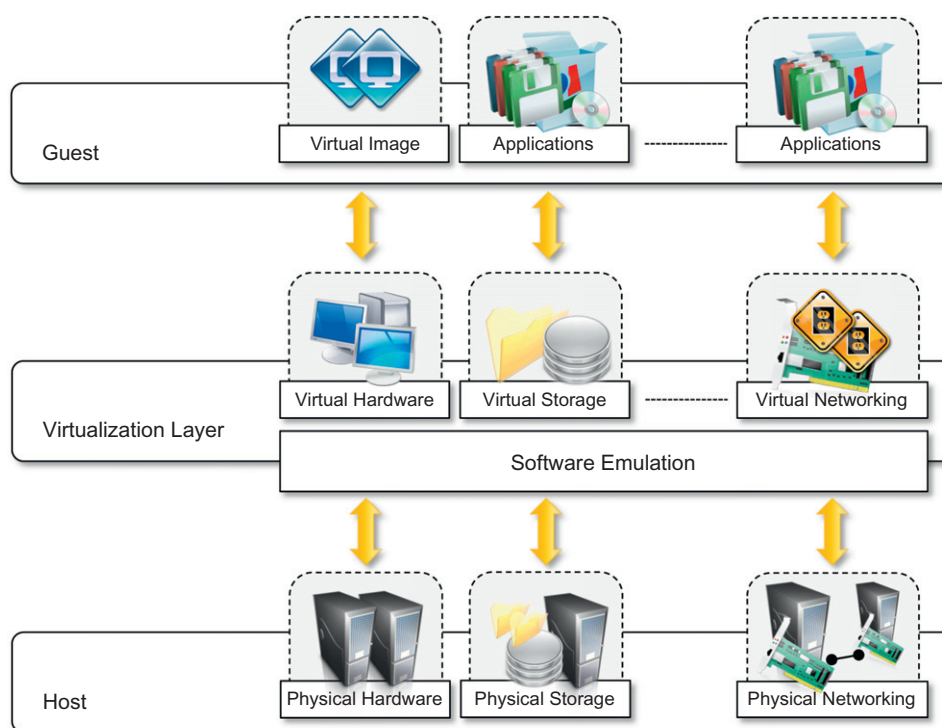
beginning of the new millennium Java played a significant role in the application server market segment, thus demonstrating that the existing technology was ready to support the execution of managed code for enterprise-class applications. In 2002 Microsoft released the first version of .NET Framework, which was Microsoft's alternative to the Java technology. Based on the same principles as Java, able to support multiple programming languages, and featuring complete integration with other Microsoft technologies, .NET Framework soon became the principal development platform for the Microsoft world and quickly became popular among developers. In 2006, two of the three "official languages" used for development at Google, Java and Python, were based on the virtual machine model. This trend of shifting toward virtualization from a programming language perspective demonstrated an important fact: The technology was ready to support virtualized solutions without a significant performance overhead. This paved the way to another and more radical form of virtualization that now has become a fundamental requisite for any data center management infrastructure.

3.2 Characteristics of virtualized environments

Virtualization is a broad concept that refers to the creation of a virtual version of something, whether hardware, a software environment, storage, or a network. In a virtualized environment there are three major components: *guest*, *host*, and *virtualization layer*. The *guest* represents the system component that interacts with the virtualization layer rather than with the host, as would normally happen. The *host* represents the original environment where the guest is supposed to be managed. The *virtualization layer* is responsible for recreating the same or a different environment where the guest will operate (see Figure 3.1).

Such a general abstraction finds different applications and then implementations of the virtualization technology. The most intuitive and popular is represented by *hardware virtualization*, which also constitutes the original realization of the virtualization concept.² In the case of hardware virtualization, the guest is represented by a system image comprising an operating system and installed applications. These are installed on top of virtual hardware that is controlled and managed by the virtualization layer, also called the *virtual machine manager*. The host is instead represented by the physical hardware, and in some cases the operating system, that defines the environment where the virtual machine manager is running. In the case of virtual storage, the guest might be client applications or users that interact with the virtual storage management software deployed on top of the real storage system. The case of virtual networking is also similar: The guest—applications and users—interacts with a virtual network, such as a *virtual private network (VPN)*, which is managed by specific software (VPN client) using the physical network available on the node. VPNs are useful for creating the illusion of being within a different physical network and thus accessing the resources in it, which would otherwise not be available.

²Virtualization is a technology that was initially developed during the mainframe era. The IBM CP/CMS mainframes were the first systems to introduce the concept of hardware virtualization and hypervisors. These systems, able to run multiple operating systems at the same time, provided a backward-compatible environment that allowed customers to run previous versions of their applications.

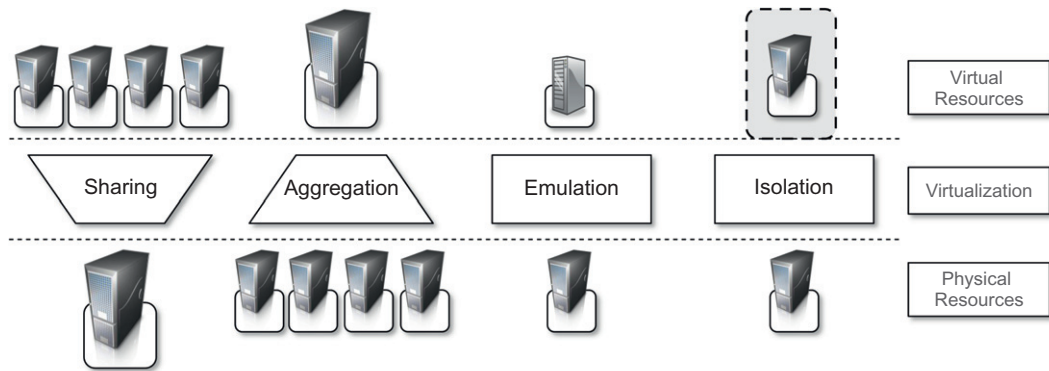
**FIGURE 3.1**

The virtualization reference model.

The main common characteristic of all these different implementations is the fact that the virtual environment is created by means of a *software program*. The ability to use software to emulate such a wide variety of environments creates a lot of opportunities, previously less attractive because of excessive overhead introduced by the virtualization layer. The technologies of today allow profitable use of virtualization and make it possible to fully exploit the advantages that come with it. Such advantages have always been characteristics of virtualized solutions.

3.2.1 Increased security

The ability to control the execution of a guest in a completely transparent manner opens new possibilities for delivering a secure, controlled execution environment. The virtual machine represents an emulated environment in which the guest is executed. All the operations of the guest are generally performed against the virtual machine, which then translates and applies them to the host. This level of indirection allows the virtual machine manager to *control* and *filter* the activity of the guest, thus preventing some harmful operations from being performed. Resources exposed by the host can then be hidden or simply protected from the guest. Moreover, sensitive

**FIGURE 3.2**

Functions enabled by managed execution.

information that is contained in the host can be naturally hidden without the need to install complex security policies. Increased security is a requirement when dealing with untrusted code. For example, applets downloaded from the Internet run in a sandboxed³ version of the *Java Virtual Machine (JVM)*, which provides them with limited access to the hosting operating system resources. Both the JVM and the .NET runtime provide extensive security policies for customizing the execution environment of applications. Hardware virtualization solutions such as VMware Desktop, VirtualBox, and Parallels provide the ability to create a virtual computer with customized virtual hardware on top of which a new operating system can be installed. By default, the file system exposed by the virtual computer is completely separated from the one of the host machine. This becomes the perfect environment for running applications without affecting other users in the environment.

3.2.2 Managed execution

Virtualization of the execution environment not only allows increased security, but a wider range of features also can be implemented. In particular, *sharing*, *aggregation*, *emulation*, and *isolation* are the most relevant features (see Figure 3.2).

- *Sharing*. Virtualization allows the creation of a separate computing environments within the same host. In this way it is possible to fully exploit the capabilities of a powerful guest, which would otherwise be underutilized. As we will see in later chapters, sharing is a particularly important feature in virtualized data centers, where this basic feature is used to reduce the number of active servers and limit power consumption.

³The term sandbox identifies an isolated execution environment where instructions can be filtered and blocked before being translated and executed in the real execution environment. The expression sandboxed version of the Java Virtual Machine (JVM) refers to a particular configuration of the JVM where, by means of security policy, instructions that are considered potential harmful can be blocked.

- *Aggregation.* Not only is it possible to share physical resource among several guests, but virtualization also allows aggregation, which is the opposite process. A group of separate hosts can be tied together and represented to guests as a single virtual host. This function is naturally implemented in middleware for distributed computing, with a classical example represented by cluster management software, which harnesses the physical resources of a homogeneous group of machines and represents them as a single resource.
- *Emulation.* Guest programs are executed within an environment that is controlled by the virtualization layer, which ultimately is a program. This allows for controlling and tuning the environment that is exposed to guests. For instance, a completely different environment with respect to the host can be emulated, thus allowing the execution of guest programs requiring specific characteristics that are not present in the physical host. This feature becomes very useful for testing purposes, where a specific guest has to be validated against different platforms or architectures and the wide range of options is not easily accessible during development. Again, hardware virtualization solutions are able to provide virtual hardware and emulate a particular kind of device such as *Small Computer System Interface (SCSI)* devices for file I/O, without the hosting machine having such hardware installed. Old and legacy software that does not meet the requirements of current systems can be run on emulated hardware without any need to change the code. This is possible either by emulating the required hardware architecture or within a specific operating system sandbox, such as the MS-DOS mode in Windows 95/98. Another example of emulation is an arcade-game emulator that allows us to play arcade games on a normal personal computer.
- *Isolation.* Virtualization allows providing guests—whether they are operating systems, applications, or other entities—with a completely separate environment, in which they are executed. The guest program performs its activity by interacting with an abstraction layer, which provides access to the underlying resources. Isolation brings several benefits; for example, it allows multiple guests to run on the same host without interfering with each other. Second, it provides a separation between the host and the guest. The virtual machine can filter the activity of the guest and prevent harmful operations against the host.

Besides these characteristics, another important capability enabled by virtualization is *performance tuning*. This feature is a reality at present, given the considerable advances in hardware and software supporting virtualization. It becomes easier to control the performance of the guest by finely tuning the properties of the resources exposed through the virtual environment. This capability provides a means to effectively implement a quality-of-service (QoS) infrastructure that more easily fulfills the service-level agreement (SLA) established for the guest. For instance, software-implementing hardware virtualization solutions can expose to a guest operating system only a fraction of the memory of the host machine or set the maximum frequency of the processor of the virtual machine. Another advantage of managed execution is that sometimes it allows easy capturing of the state of the guest program, persisting it, and resuming its execution. This, for example, allows virtual machine managers such as Xen Hypervisor to stop the execution of a guest operating system, move its virtual image into another machine, and resume its execution in a completely transparent manner. This technique is called *virtual machine migration* and constitutes an important feature in virtualized data centers for optimizing their efficiency in serving application demands.

3.2.3 Portability

The concept of *portability* applies in different ways according to the specific type of virtualization considered. In the case of a hardware virtualization solution, the guest is packaged into a virtual image that, in most cases, can be safely moved and executed on top of different virtual machines. Except for the file size, this happens with the same simplicity with which we can display a picture image in different computers. Virtual images are generally proprietary formats that require a specific virtual machine manager to be executed. In the case of programming-level virtualization, as implemented by the JVM or the .NET runtime, the binary code representing application components (jars or assemblies) can be run without any recompilation on any implementation of the corresponding virtual machine. This makes the application development cycle more flexible and application deployment very straightforward: One version of the application, in most cases, is able to run on different platforms with no changes. Finally, portability allows having your own system always with you and ready to use as long as the required virtual machine manager is available. This requirement is, in general, less stringent than having all the applications and services you need available to you anywhere you go.

3.3 Taxonomy of virtualization techniques

Virtualization covers a wide range of emulation techniques that are applied to different areas of computing. A classification of these techniques helps us better understand their characteristics and use (see [Figure 3.3](#)).

The first classification discriminates against the service or entity that is being emulated. Virtualization is mainly used to emulate *execution environments*, *storage*, and *networks*. Among these categories, *execution virtualization* constitutes the oldest, most popular, and most developed area. Therefore, it deserves major investigation and a further categorization. In particular we can divide these execution virtualization techniques into two major categories by considering the type of host they require. *Process-level* techniques are implemented on top of an existing operating system, which has full control of the hardware. *System-level* techniques are implemented directly on hardware and do not require—or require a minimum of support from—an existing operating system. Within these two categories we can list various techniques that offer the guest a different type of virtual computation environment: bare hardware, operating system resources, low-level programming language, and application libraries.

3.3.1 Execution virtualization

Execution virtualization includes all techniques that aim to emulate an execution environment that is separate from the one hosting the virtualization layer. All these techniques concentrate their interest on providing support for the execution of programs, whether these are the operating system, a binary specification of a program compiled against an abstract machine model, or an application. Therefore, execution virtualization can be implemented directly on top of the hardware by the operating system, an application, or libraries dynamically or statically linked to an application image.

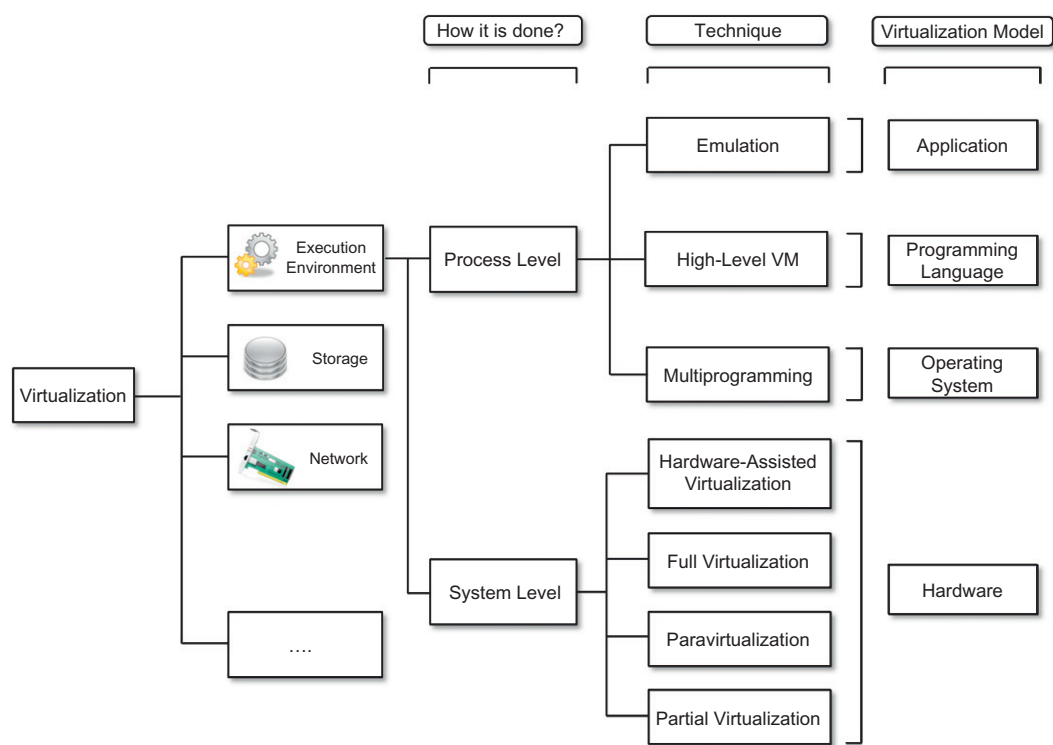


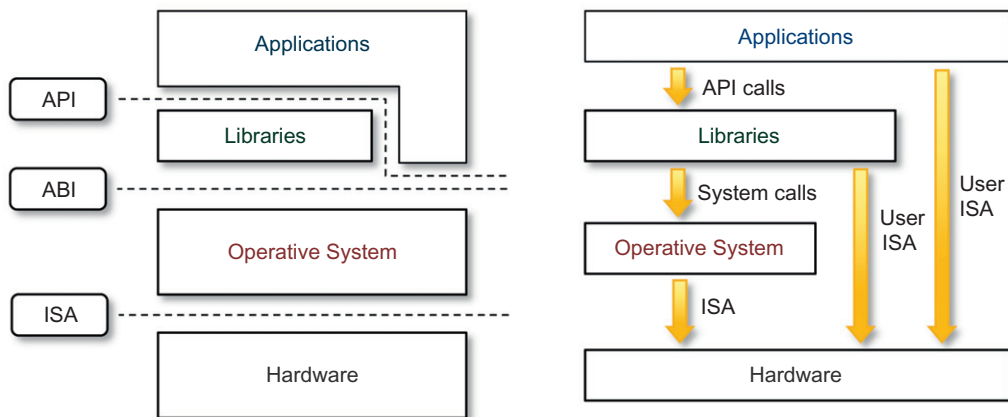
FIGURE 3.3

A taxonomy of virtualization techniques.

3.3.1.1 Machine reference model

Virtualizing an execution environment at different levels of the computing stack requires a reference model that defines the interfaces between the levels of abstractions, which hide implementation details. From this perspective, virtualization techniques actually replace one of the layers and intercept the calls that are directed toward it. Therefore, a clear separation between layers simplifies their implementation, which only requires the emulation of the interfaces and a proper interaction with the underlying layer.

Modern computing systems can be expressed in terms of the reference model described in Figure 3.4. At the bottom layer, the model for the hardware is expressed in terms of the *Instruction Set Architecture (ISA)*, which defines the instruction set for the processor, registers, memory, and interrupt management. ISA is the interface between hardware and software, and it is important to the operating system (OS) developer (*System ISA*) and developers of applications that directly manage the underlying hardware (*User ISA*). The *application binary interface (ABI)* separates the operating system layer from the applications and libraries, which are managed by the OS. ABI covers details such as low-level data types, alignment, and call conventions and defines a format for executable programs. System calls are defined at this level. This interface allows portability of applications and libraries across operating systems that

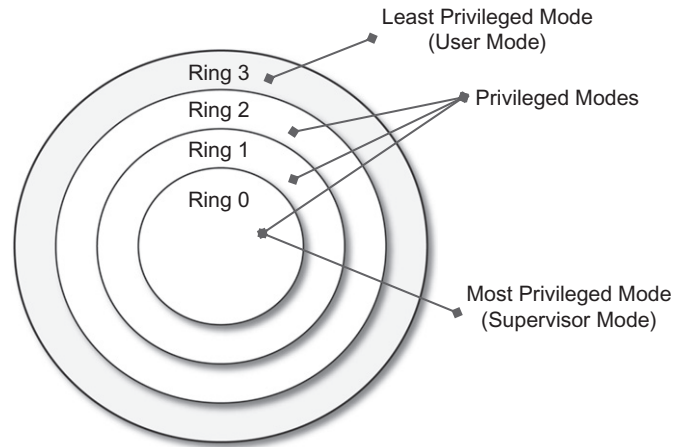
**FIGURE 3.4**

A machine reference model.

implement the same ABI. The highest level of abstraction is represented by the *application programming interface (API)*, which interfaces applications to libraries and/or the underlying operating system.

For any operation to be performed in the application level API, ABI and ISA are responsible for making it happen. The high-level abstraction is converted into machine-level instructions to perform the actual operations supported by the processor. The machine-level resources, such as processor registers and main memory capacities, are used to perform the operation at the hardware level of the central processing unit (CPU). This layered approach simplifies the development and implementation of computing systems and simplifies the implementation of multitasking and the coexistence of multiple executing environments. In fact, such a model not only requires limited knowledge of the entire computing stack, but it also provides ways to implement a minimal security model for managing and accessing shared resources.

For this purpose, the instruction set exposed by the hardware has been divided into different security classes that define who can operate with them. The first distinction can be made between *privileged* and *nonprivileged* instructions. Nonprivileged instructions are those instructions that can be used without interfering with other tasks because they do not access shared resources. This category contains, for example, all the floating, fixed-point, and arithmetic instructions. Privileged instructions are those that are executed under specific restrictions and are mostly used for sensitive operations, which expose (*behavior-sensitive*) or modify (*control-sensitive*) the privileged state. For instance, behavior-sensitive instructions are those that operate on the I/O, whereas control-sensitive instructions alter the state of the CPU registers. Some types of architecture feature more than one class of privileged instructions and implement a finer control of how these instructions can be accessed. For instance, a possible implementation features a hierarchy of privileges (see Figure 3.5) in the form of ring-based security: *Ring 0*, *Ring 1*, *Ring 2*, and *Ring 3*; Ring 0 is in the most privileged level and Ring 3 in the least privileged level. Ring 0 is used by the kernel of the OS, rings 1 and 2 are used by the OS-level services, and Ring 3 is used by the user. Recent systems support only two levels, with Ring 0 for supervisor mode and Ring 3 for user mode.

**FIGURE 3.5**

Security rings and privilege modes.

All the current systems support at least two different execution modes: *supervisor mode* and *user mode*. The first mode denotes an execution mode in which all the instructions (privileged and nonprivileged) can be executed without any restriction. This mode, also called *master mode* or *kernel mode*, is generally used by the operating system (or the hypervisor) to perform sensitive operations on hardware-level resources. In user mode, there are restrictions to control the machine-level resources. If code running in user mode invokes the privileged instructions, hardware interrupts occur and trap the potentially harmful execution of the instruction. Despite this, there might be some instructions that can be invoked as privileged instructions under some conditions and as nonprivileged instructions under other conditions.

The distinction between *user* and *supervisor* mode allows us to understand the role of the *hypervisor* and why it is called that. Conceptually, the hypervisor runs above the supervisor mode, and from here the prefix *hyper-* is used. In reality, hypervisors are run in supervisor mode, and the division between privileged and nonprivileged instructions has posed challenges in designing virtual machine managers. It is expected that all the sensitive instructions will be executed in privileged mode, which requires supervisor mode in order to avoid traps. Without this assumption it is impossible to fully emulate and manage the status of the CPU for guest operating systems. Unfortunately, this is not true for the original ISA, which allows 17 sensitive instructions to be called in user mode. This prevents multiple operating systems managed by a single hypervisor to be isolated from each other, since they are able to access the privileged state of the processor and change it.⁴ More recent implementations of ISA (Intel VT and AMD Pacifica) have solved this problem by redesigning such instructions as privileged ones.

By keeping in mind this reference model, it is possible to explore and better understand the various techniques utilized to virtualize execution environments and their relationships to the other components of the system.

⁴It is expected that in a hypervisor-managed environment, all the guest operating system code will be run in user mode in order to prevent it from directly accessing the status of the CPU. If there are sensitive instructions that can be called in user mode (that is, implemented as nonprivileged instructions), it is no longer possible to completely isolate the guest OS.

1.5 Operating-System Operations

As mentioned earlier, modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt.

Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

1.5.1 Dual-Mode and Multimode Operation

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

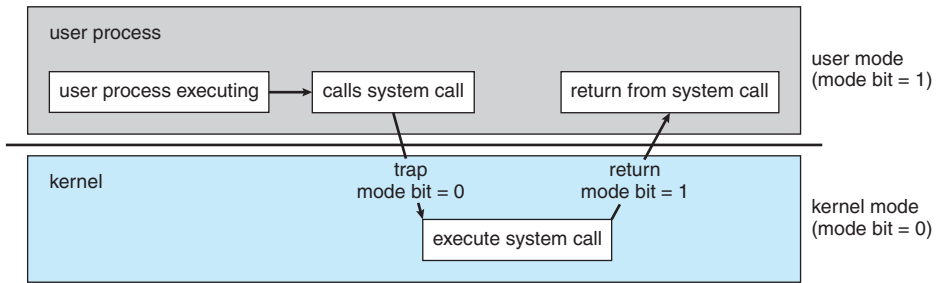


Figure 1.10 Transition from user to kernel mode.

At the very least, we need two separate *modes* of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.10. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. As we shall see throughout the text, there are many additional privileged instructions.

The concept of modes can be extended beyond two modes (in which case the CPU uses more than one bit to set and test the mode). CPUs that support virtualization (Section 16.1) frequently have a separate mode to indicate when the **virtual machine manager (VMM)**—and the virtualization management software—is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so. Sometimes, too, different modes are used by various kernel

components. We should note that, as an alternative to modes, the CPU designer may use other methods to differentiate operational privileges. The Intel 64 family of CPUs supports four *privilege levels*, for example, and supports virtualization but does not have a separate mode for virtualization.

We can now see the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems (such as MIPS) have a specific `syscall` instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time, with potentially disastrous results. Modern versions of the Intel CPU do provide dual-mode operation. Accordingly, most contemporary operating systems—such as Microsoft Windows 7, as well as Unix and Linux—take advantage of this dual-mode feature and provide greater protection for the operating system.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space—then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

1.5.2 Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

We can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts, and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

1.6 Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one). For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. As we shall see in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given the name of the file as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the terminal. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process. A program is a *passive* entity, like the contents of a file stored on disk, whereas a process

is an *active* entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU, for example.

The operating system is responsible for the following activities in connection with process management:

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

We discuss process-management techniques in Chapters 3 through 5.

1.7 Memory Management

As we discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory-

management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the *hardware* design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and who is using them
- Deciding which processes (or parts of processes) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Memory-management techniques are discussed in Chapters 8 and 9.

1.8 Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file*. The operating system maps files onto physical media and accesses these files via the storage devices.

1.8.1 File-System Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass-storage media, such as tapes and disks, and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files

- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques are discussed in Chapters 11 and 12.

1.8.2 Mass-Storage Management

As we have already seen, because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the principal on-line storage medium for both programs and data. Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory. They then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and the algorithms that manipulate that subsystem.

There are, however, many uses for storage that is slower and lower in cost (and sometimes of higher capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD and DVD drives and platters are typical **tertiary storage** devices. The media (tapes and optical platters) vary between **WORM** (write-once, read-many-times) and **RW** (read-write) formats.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary and tertiary storage management are discussed in Chapter 10.

1.8.3 Caching

Caching is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a

temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance. Figure 1.11 compares storage performance in large workstations and small servers. Various replacement algorithms for software-controlled caches are discussed in Chapter 9.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory. In addition, solid-state disks may be used for high-speed storage that is accessed through the file-system interface. The bulk of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of a hard-disk failure. Some systems automatically archive old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost (see Chapter 10).

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.11 Performance of various levels of storage.

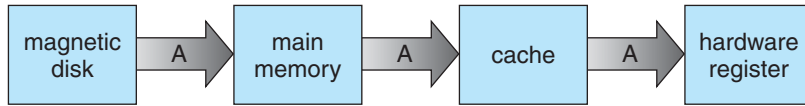


Figure 1.12 Migration of integer A from disk to register.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (see Figure 1.12). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (Figure 1.6). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 17.

1.8.4 I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O

devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed in Section 1.2.3 how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 13, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

1.9 Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 14.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents,

Virtual Machines



The term *virtualization* has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware and that behaves toward them as native hardware would but that also protects, manages, and limits them.

This chapter delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this chapter describes these options. One option is to add virtual machine support to the kernel. Because that implementation method is the most pertinent to this book, we explore it most fully. Additionally, hardware features provided by the CPU and even by I/O devices can support virtual machine implementation, so we discuss how those features are used by the appropriate kernel modules.

CHAPTER OBJECTIVES

- To explore the history and benefits of virtual machines.
- To discuss the various virtual machine technologies.
- To describe the methods used to implement virtualization.
- To show the most common hardware features that support virtualization and explain how they are used by operating-system modules.

16.1 Overview

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. This concept may seem similar to the layered approach of operating system implementation (see Section 2.7.2), and in some ways it is. In the case of virtualization, there is a layer that creates a virtual system on which operating systems or applications can run.

Virtual machine implementations involve several components. At the base is the **host**, the underlying hardware system that runs the virtual machines. The **virtual machine manager (VMM)** (also known as a **hypervisor**) creates and runs virtual machines by providing an interface that is *identical* to the host (except in the case of paravirtualization, discussed later). Each **guest** process is provided with a virtual copy of the host (Figure 16.1). Usually, the guest process is in fact an operating system. A single physical machine can thus run multiple operating systems concurrently, each in its own virtual machine.

Take a moment to note that with virtualization, the definition of “operating system” once again blurs. For example, consider VMM software such as VMware ESX. This virtualization software is installed on the hardware, runs when the hardware boots, and provides services to applications. The services include traditional ones, such as scheduling and memory management, along with new types, such as migration of applications between systems. Furthermore, the applications are in fact guest operating systems. Is the VMware ESX VMM an operating system that, in turn, runs other operating systems? Certainly it acts like an operating system. For clarity, however, we call the component that provides virtual environments a VMM.

The implementation of VMMs varies greatly. Options include the following:

- Hardware-based solutions that provide support for virtual machine creation and management via firmware. These VMMs, which are commonly found in mainframe and large to midsized servers, are generally known as **type 0 hypervisors**. IBM LPARs and Oracle LDOMs are examples.
- Operating-system-like software built to provide virtualization, including VMware ESX(mentioned above), Joyent SmartOS, and Citrix XenServer. These VMMs are known as **type 1 hypervisors**.

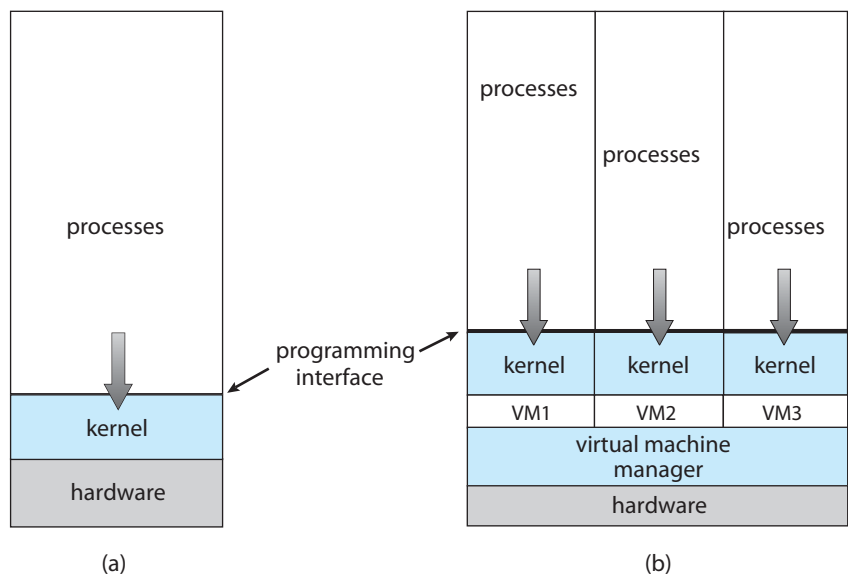


Figure 16.1 System models. (a) Nonvirtual machine. (b) Virtual machine.

INDIRECTION

“All problems in computer science can be solved by another level of indirection”—David Wheeler “. . . except for the problem of too many layers of indirection.”—Kevlin Henney

- General-purpose operating systems that provide standard functions as well as VMM functions, including Microsoft Windows Server with HyperV and RedHat Linux with the KVM feature. Because such systems have a feature set similar to type 1 hypervisors, they are also known as type 1.
- Applications that run on standard operating systems but provide VMM features to guest operating systems. These applications, which include VMware Workstation and Fusion, Parallels Desktop, and Oracle Virtual-Box, are **type 2 hypervisors**.
- **Paravirtualization**, a technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance.
- **Programming-environment virtualization**, in which VMMs do not virtualize real hardware but instead create an optimized virtual system. This technique is used by Oracle Java and Microsoft.Net.
- **Emulators** that allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU.
- **Application containment**, which is not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system. Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs “contain” applications, making them more secure and manageable.

The variety of virtualization techniques in use today is a testament to the breadth, depth, and importance of virtualization in modern computing. Virtualization is invaluable for data-center operations, efficient application development, and software testing, among many other uses.

16.2 History

Virtual machines first appeared commercially on IBM mainframes in 1972. Virtualization was provided by the IBM VM operating system. This system has evolved and is still available. In addition, many of its original concepts are found in other systems, making it worth exploring.

IBM VM370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM approach involved disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual machines. Clearly, it could not allocate a disk drive to each virtual machine. The solution was to provide virtual disks—termed **minidisks** in IBM’s VM operating system. The minidisks are identical

to the system's hard disks in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once the virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS—a single-user interactive operating system.

For many years after IBM introduced this technology, virtualization remained in its domain. Most systems could not support virtualization. However, a formal definition of virtualization helped to establish system requirements and a target for functionality. The virtualization requirements stated that:

1. A VMM provides an environment for programs that is essentially identical to the original machine.
2. Programs running within that environment show only minor performance decreases.
3. The VMM is in complete control of system resources.

These requirements of fidelity, performance, and safety still guide virtualization efforts today.

By the late 1990s, Intel 80x86 CPUs had become common, fast, and rich in features. Accordingly, developers launched multiple efforts to implement virtualization on that platform. Both **Xen** and **VMware** created technologies, still used today, to allow guest operating systems to run on the 80x86. Since that time, virtualization has expanded to include all common CPUs, many commercial and open-source tools, and many operating systems. For example, the open-source *VirtualBox* project (<http://www.virtualbox.org>) provides a program that runs on Intel x86 and AMD64 CPUs and on Windows, Linux, Mac OS X, and Solaris host operating systems. Possible guest operating systems include many versions of Windows, Linux, Solaris, and BSD, including even MS-DOS and IBM OS/2.

16.3 Benefits and Features

Several advantages make virtualization attractive. Most of them are fundamentally related to the ability to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems.

A potential disadvantage of isolation is that it can prevent sharing of resources. Two approaches to provide sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can

send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software. Of course, the VMM is free to allow any number of its guests to use physical resources, such as a physical network connection (with sharing provided by the VMM), in which case the allowed guests could communicate with each other via the physical network.

One feature common to most virtualization implementations is the ability to freeze, or **suspend**, a running virtual machine. Many operating systems provide that basic feature for processes, but VMMs go one step further and allow copies and **snapshots** to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then **resume** where it was, as if on its original machine, creating a **clone**. The snapshot records a point in time, and the guest can be reset to that point if necessary (for example, if a change was made but is no longer wanted). Often, VMMs allow many snapshots to be taken. For example, snapshots might record a guest's state every day for a month, making restoration to any of those snapshot states possible. These abilities are used to good advantage in virtual environments.

A virtual machine system is a perfect vehicle for operating-system research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

Furthermore, the operating system runs on and controls the entire machine, meaning that the system must be stopped and taken out of use while changes are made and tested. This period is commonly called **system-development time**. Since it makes the system unavailable to users, system-development time on shared systems is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this latter problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation is disrupted only when a completed and tested change is ready to be put into production.

Another advantage of virtual machines for developers is that multiple operating systems can run concurrently on the developer's workstation. This virtualized workstation allows for rapid porting and testing of programs in varying environments. In addition, multiple versions of a program can run, each in its own isolated operating system, within one system. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.

A major advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, since many lightly used systems can be combined to create one more heavily used system.

Consider, too, that management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators. One of the tools that make this possible is **templating**, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs. Other features include managing the patching of all guests, backing up and restoring the guests, and monitoring their resource use.

Virtualization can improve not only resource utilization but also resource management. Some VMMs include a **live migration** feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, the evacuated host can be maintained, and then the guests can be migrated back. This operation occurs without downtime and without interruption to users.

Think about the possible effects of virtualization on how applications are deployed. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, the application could be preinstalled on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become easier, less tuning would be required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much easier than the usual steps of uninstalling and reinstalling. For widespread adoption of this methodology to occur, though, the format of virtual machines must be standardized so that any virtual machine will run on any virtualization platform. The “Open Virtual Machine Format” is an attempt to provide such standardization, and it could succeed in unifying virtual machine formats.

Virtualization has laid the foundation for many other advances in computer facility implementation, management, and monitoring. **Cloud computing**, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, which others can access via the Internet. Many multiuser games, photo-sharing sites, and other web services use this functionality.

In the area of desktop computing, virtualization is enabling desktop and laptop computer users to connect remotely to virtual machines located in remote data centers and access their applications as if they were local. This practice can increase security, because no data are stored on local disks at the user’s site. The cost of the user’s computing resource may also decrease. The user must have networking, CPU, and some memory, but all that these system components need to do is display an image of the guest as it runs remotely (via

a protocol such as [RDP](#)). Thus, they need not be expensive, high-performance components. Other uses of virtualization are sure to follow as it becomes more prevalent and hardware support continues to improve.

16.4 Building Blocks

Although the virtual machine concept is useful, it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine. This is especially a challenge on dual-mode systems, where the underlying machine has only user mode and kernel mode. In this section, we examine the building blocks that are needed for efficient virtualization. Note that these building blocks are not required by type 0 hypervisors, as discussed in Section 16.5.2.

The ability to virtualize depends on the features provided by the CPU. If the features are sufficient, then it is possible to write a VMM that provides a guest environment. Otherwise, virtualization is impossible. VMMs use several techniques to implement virtualization, including trap-and-emulate and binary translation. We discuss each of these techniques in this section, along with the hardware support needed to support virtualization.

One important concept found in most virtualization options is the implementation of a [virtual CPU \(VCPU\)](#). The VCPU does not execute code. Rather, it represents the state of the CPU as the guest machine believes it to be. For each guest, the VMM maintains a VCPU representing that guest's current CPU state. When the guest is context-switched onto a CPU by the VMM, information from the VCPU is used to load the right context, much as a general-purpose operating system would use the PCB.

16.4.1 Trap-and-Emulate

On a typical dual-mode system, the virtual machine guest can execute only in user mode (unless extra hardware support is provided). The kernel, of course, runs in kernel mode, and it is not safe to allow user-level code to run in kernel mode. Just as the physical machine has two modes, however, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call, an interrupt, or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode in the virtual machine.

How can such a transfer be accomplished? The procedure is as follows: When the kernel in the guest attempts to execute a privileged instruction, that is an error (because the system is in user mode) and causes a trap to the VMM in the real machine. The VMM gains control and executes (or “emulates”) the action that was attempted by the guest kernel on the part of the guest. It then returns control to the virtual machine. This is called the [trap-and-emulate](#) method and is shown in Figure 16.2. Most virtualization products use this method to one extent or other.

With privileged instructions, time becomes an issue. All nonprivileged instructions run natively on the hardware, providing the same performance

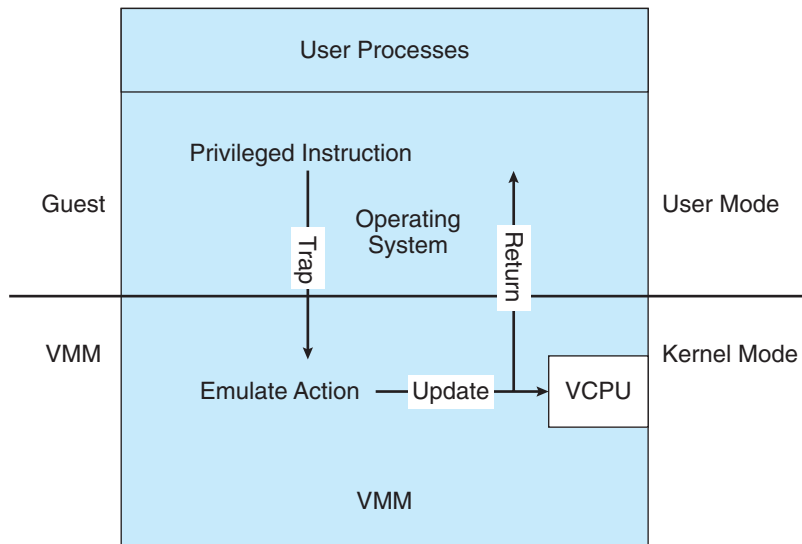


Figure 16.2 Trap-and-emulate virtualization implementation.

for guests as native applications. Privileged instructions create extra overhead, however, causing the guest to run more slowly than it would natively. In addition, the CPU is being multiprogrammed among many virtual machines, which can further slow down the virtual machines in unpredictable ways.

This problem has been approached in various ways. IBM VM, for example, allows normal instructions for the virtual machines to execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be emulated and hence execute more slowly. In general, with the evolution of hardware, the performance of trap-and-emulate functionality has been improved, and cases in which it is needed have been reduced. For example, many CPUs now have extra modes added to their standard dual-mode operation. The VCPU need not keep track of what mode the guest operating system is in, because the physical CPU performs that function. In fact, some CPUs provide guest CPU state management in hardware, so the VMM need not supply that functionality, removing the extra overhead.

16.4.2 Binary Translation

Some CPUs do not have a clean separation of privileged and nonprivileged instructions. Unfortunately for virtualization implementers, the Intel x86 CPU line is one of them. No thought was given to running virtualization on the x86 when it was designed. (In fact, the first CPU in the family—the Intel 4004, released in 1971—was designed to be the core of a calculator.) The chip has maintained backward compatibility throughout its lifetime, preventing changes that would have made virtualization easier through many generations. Let's consider an example of the problem. The command `popf` loads the flag register from the contents of the stack. If the CPU is in privileged mode, all of the flags are replaced from the stack. If the CPU is in user mode, then only some flags are replaced, and others are ignored. Because no trap is generated if `popf` is executed in user mode, the trap-and-emulate procedure is rendered

useless. Other x86 instructions cause similar problems. For the purposes of this discussion, we will call this set of instructions *special instructions*. As recently as 1998, /Judi 1998 doesn't seem that recent using the trap-and-emulate method to implement virtualization on the x86 was considered impossible because of these special instructions.

This previously insurmountable problem was solved with the implementation of the **binary translation** technique. Binary translation is fairly simple in concept but complex in implementation. The basic steps are as follows:

1. If the guest VCPU is in user mode, the guest can run its instructions natively on a physical CPU.
2. If the guest VCPU is in kernel mode, then the guest believes that it is running in kernel mode. The VMM examines every instruction the guest executes in virtual kernel mode by reading the next few instructions that the guest is going to execute, based on the guest's program counter. Instructions other than special instructions are run natively. Special instructions are translated into a new set of instructions that perform the equivalent task—for example changing the flags in the VCPU.

Binary translation is shown in Figure 16.3. It is implemented by translation code within the VMM. The code reads native binary instructions dynamically from the guest, on demand, and generates native binary code that executes in place of the original code.

The basic method of binary translation just described would execute correctly but perform poorly. Fortunately, the vast majority of instructions would execute natively. But how could performance be improved for the other instructions? We can turn to a specific implementation of binary translation, the VMware method, to see one way of improving performance. Here, caching

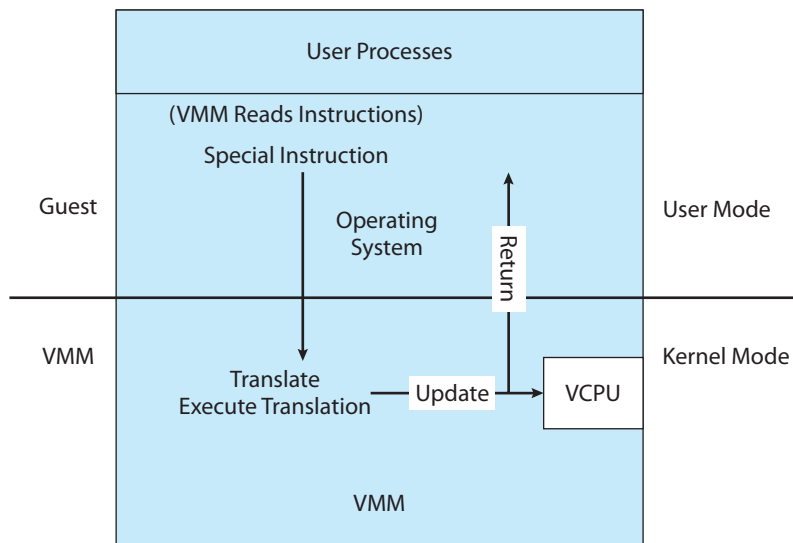


Figure 16.3 Binary translation virtualization implementation.

provides the solution. The replacement code for each instruction that needs to be translated is cached. All later executions of that instruction run from the translation cache and need not be translated again. If the cache is large enough, this method can greatly improve performance.

Let's consider another issue in virtualization: memory management, specifically the page tables. How can the VMM keep page-table state both for guests that believe they are managing the page tables and for the VMM itself? A common method, used with both trap-and-emulate and binary translation, is to use **nested page tables (NPTs)**. Each guest operating system maintains one or more page tables to translate from virtual to physical memory. The VMM maintains NPTs to represent the guest's page-table state, just as it creates a VCPU to represent the guest's CPU state. The VMM knows when the guest tries to change its page table, and it makes the equivalent change in the NPT. When the guest is on the CPU, the VMM puts the pointer to the appropriate NPT into the appropriate CPU register to make that table the active page table. If the guest needs to modify the page table (for example, fulfilling a page fault), then that operation must be intercepted by the VMM and appropriate changes made to the nested and system page tables. Unfortunately, the use of NPTs can cause TLB misses to increase, and many other complexities need to be addressed to achieve reasonable performance.

Although it might seem that the binary translation method creates large amounts of overhead, it performed well enough to launch a new industry aimed at virtualizing Intel x86-based systems. VMware tested the performance impact of binary translation by booting one such system, Windows XP, and immediately shutting it down while monitoring the elapsed time and the number of translations produced by the binary translation method. The result was 950,000 translations, taking 3 microseconds each, for a total increase of 3 seconds (about 5%) over native execution of Windows XP. To achieve that result, developers used many performance improvements that we do not discuss here. For more information, consult the bibliographical notes at the end of this chapter.

16.4.3 Hardware Assistance

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more feature-rich and stable the virtual machines can be and the better they can perform. In the Intel x86 CPU family, Intel added new virtualization support in successive generations (the VT-x instructions) beginning in 2005. Now, binary translation is no longer needed.

In fact, all major general-purpose CPUs are providing extended amounts of hardware support for virtualization. For example, AMD virtualization technology (**AMD-V**) has appeared in several AMD processors starting in 2006. It defines two new modes of operation—host and guest—thus moving from a dual-mode to a multimode processor. The VMM can enable host mode, define the characteristics of each guest virtual machine, and then switch the system to guest mode, passing control of the system to a guest operating system that is running in the virtual machine. In guest mode, the virtualized operating system thinks it is running on native hardware and sees whatever devices are included in the host's definition of the guest. If the guest tries to access a

virtualized resource, then control is passed to the VMM to manage that interaction. The functionality in Intel VT-x is similar, providing root and nonroot modes, equivalent to host and guest modes. Both provide guest VCPU state data structures to load and save guest CPU state automatically during guest context switches. In addition, **virtual machine control structures (VMCSs)** are provided to manage guest and host state, as well as the various guest execution controls, exit controls, and information about why guests exit back to the host. In the latter case, for example, a nested page-table violation caused by an attempt to access unavailable memory can result in the guest's exit.

AMD and Intel have also addressed memory management in the virtual environment. With AMD's RVI and Intel's EPT memory management enhancements, VMMs no longer need to implement software NPTs. In essence, these CPUs implement nested page tables in hardware to allow the VMM to fully control paging while the CPUs accelerate the translation from virtual to physical addresses. The NPTs add a new layer, one representing the guest's view of logical-to-physical address translation. The CPU page-table walking function includes this new layer as necessary, walking through the guest table to the VMM table to find the physical address desired. A TLB miss results in a performance penalty, because more tables must be traversed (the guest and host page tables) to complete the lookup. Figure 16.4 shows the extra translation work performed by the hardware to translate from a guest virtual address to a final physical address.

I/O is another area improved by hardware assistance. Consider that the standard direct-memory-access (DMA) controller accepts a target memory address and a source I/O device and transfers data between the two without operating-system action. Without hardware assistance, a guest might try to set up a DMA transfer that affects the memory of the VMM or other guests. In CPUs that provide hardware-assisted DMA (such as Intel CPUs with VT-d), even DMA has a level of indirection. First, the VMM sets up **protection domains** to tell the CPU which physical memory belongs to each guest. Next, it assigns the I/O devices to the protection domains, allowing them direct access to those memory regions and only those regions. The hardware then transforms the address in a DMA request issued by an I/O device to the host physical memory address associated with the I/O. In this manner DMA transfers are passed through between a guest and a device without VMM interference.

Similarly, interrupts must be delivered to the appropriate guest and must not be visible to other guests. By providing an interrupt remapping feature, CPUs with virtualization hardware assistance automatically deliver an interrupt destined for a guest to a core that is currently running a thread of that guest. That way, the guest receives interrupts without the VMM's needing to intercede in their delivery. Without interrupt remapping, malicious guests can generate interrupts that can be used to gain control of the host system. (See the bibliographical notes at the end of this chapter for more details.)

16.5 Types of Virtual Machines and Their Implementations

We've now looked at some of the techniques used to implement virtualization. Next, we consider the major types of virtual machines, their implementation, their functionality, and how they use the building blocks just described to

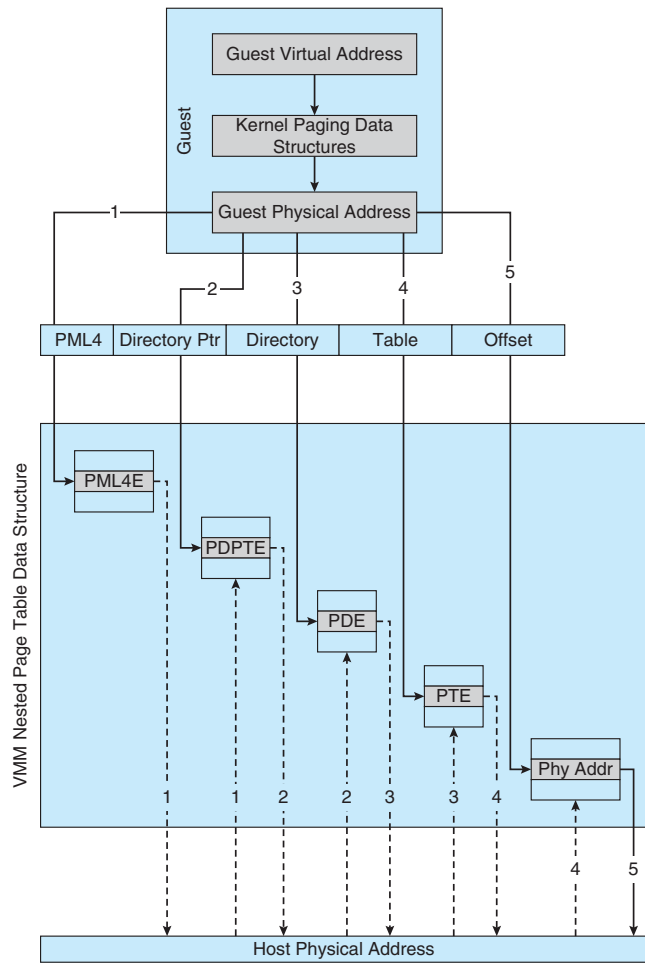


Figure 16.4 Nested page tables.

create a virtual environment. Of course, the hardware on which the virtual machines are running can cause great variation in implementation methods. Here, we discuss the implementations in general, with the understanding that VMMs take advantage of hardware assistance where it is available.

16.5.1 The Virtual Machine Life Cycle

Let’s begin with the virtual machine life cycle. Whatever the hypervisor type, at the time a virtual machine is created, its creator gives the VMM certain parameters. These parameters usually include the number of CPUs, amount of memory, networking details, and storage details that the VMM will take into account when creating the guest. For example, a user might want to create a new guest with two virtual CPUs, 4 GB of memory, 10 GB of disk space, one network interface that gets its IP address via DHCP, and access to the DVD drive.

The VMM then creates the virtual machine with those parameters. In the case of a type 0 hypervisor, the resources are usually dedicated. In this situation, if there are not two virtual CPUs available and unallocated, the creation request

in our example will fail. For other hypervisor types, the resources are dedicated or virtualized, depending on the type. Certainly, an IP address cannot be shared, but the virtual CPUs are usually multiplexed on the physical CPUs as discussed in Section 16.6.1. Similarly, memory management usually involves allocating more memory to guests than actually exists in physical memory. This is more complicated and is described in Section 16.6.2.

Finally, when the virtual machine is no longer needed, it can be deleted. When this happens, the VMM first frees up any used disk space and then removes the configuration associated with the virtual machine, essentially forgetting the virtual machine.

These steps are quite simple compared with building, configuring, running, and removing physical machines. Creating a virtual machine from an existing one can be as easy as clicking the “clone” button and providing a new name and IP address. This ease of creation can lead to **virtual machine sprawl**, which occurs when there are so many virtual machines on a system that their use, history, and state become confusing and difficult to track.

16.5.2 Type 0 Hypervisor

Type 0 hypervisors have existed for many years under many names, including “partitions” and “domains”. They are a hardware feature, and that brings its own positives and negatives. Operating systems need do nothing special to take advantage of their features. The VMM itself is encoded in the firmware and loaded at boot time. In turn, it loads the guest images to run in each partition. The feature set of a type 0 hypervisor tends to be smaller than those of the other types because it is implemented in hardware. For example, a system might be split into four virtual systems, each with dedicated CPUs, memory, and I/O devices. Each guest believes that it has dedicated hardware because it does, simplifying many implementation details.

I/O presents some difficulty, because it is not easy to dedicate I/O devices to guests if there are not enough. What if a system has two Ethernet ports and more than two guests, for example? Either all guests must get their own I/O devices, or the system must provide I/O device sharing. In these cases, the hypervisor manages shared access or grants all devices to a **control partition**. In the control partition, a guest operating system provides services (such as networking) via daemons to other guests, and the hypervisor routes I/O requests appropriately. Some type 0 hypervisors are even more sophisticated and can move physical CPUs and memory between running guests. In these cases, the guests are paravirtualized, aware of the virtualization and assisting in its execution. For example, a guest must watch for signals from the hardware or VMM that a hardware change has occurred, probe its hardware devices to detect the change, and add or subtract CPUs or memory from its available resources.

Because type 0 virtualization is very close to raw hardware execution, it should be considered separately from the other methods discussed here. A type 0 hypervisor can run multiple guest operating systems (one in each hardware partition). All of those guests, because they are running on raw hardware, can in turn be VMMs. Essentially, the guest operating systems in a type 0 hypervisor are native operating systems with a subset of hardware made available to them. Because of that, each can have its own guest operating

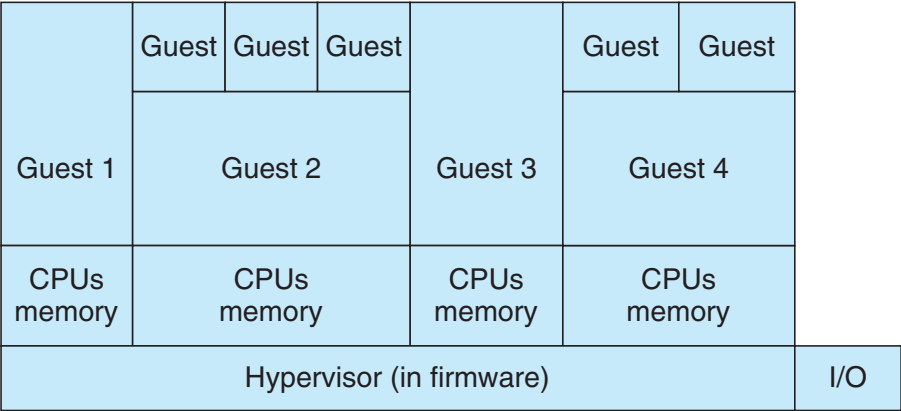


Figure 16.5 Type 0 hypervisor.

systems (Figure 16.5). Other types of hypervisors usually cannot provide this virtualization-within-virtualization functionality.

16.5.3 Type 1 Hypervisor

Type 1 hypervisors are commonly found in company data centers and are in a sense becoming “the data-center operating system.” They are special-purpose operating systems that run natively on the hardware, but rather than providing system calls and other interfaces for running programs, they create, run, and manage guest operating systems. In addition to running on standard hardware, they can run on type 0 hypervisors, but not on other type 1 hypervisors. Whatever the platform, guests generally do not know they are running on anything but the native hardware.

Type 1 hypervisors run in kernel mode, taking advantage of hardware protection. Where the host CPU allows, they use multiple modes to give guest operating systems their own control and improved performance. They implement device drivers for the hardware they run on, because no other component could do so. Because they are operating systems, they must also provide CPU scheduling, memory management, I/O management, protection, and even security. Frequently, they provide APIs, but those APIs support applications in guests or external applications that supply features like backups, monitoring, and security. Many type 1 hypervisors are closed-source commercial offerings, such as VMware ESX while some are open source or hybrids of open and closed source, such as Citrix XenServer and its open Xen counterpart.

By using type 1 hypervisors, data-center managers can control and manage the operating systems and applications in new and sophisticated ways. An important benefit is the ability to consolidate more operating systems and applications onto fewer systems. For example, rather than having ten systems running at 10 percent utilization each, a data center might have one server manage the entire load. If utilization increases, guests and their applications can be moved to less-loaded systems live, without interruption of service. Using snapshots and cloning, the system can save the states of guests and duplicate those states—a much easier task than restoring from backups or installing manually or via scripts and tools. The price of this increased manageability

is the cost of the VMM (if it is a commercial product), the need to learn new management tools and methods, and the increased complexity.

Another type of type 1 hypervisor includes various general-purpose operating systems with VMM functionality. In this instance, an operating system such as RedHat Enterprise Linux, Windows, or Oracle Solaris performs its normal duties as well as providing a VMM allowing other operating systems to run as guests. Because of their extra duties, these hypervisors typically provide fewer virtualization features than other type 1 hypervisors. In many ways, they treat a guest operating system as just another process, albeit with special handling provided when the guest tries to execute special instructions.

16.5.4 Type 2 Hypervisor

Type 2 hypervisors are less interesting to us as operating-system explorers, because there is very little operating-system involvement in these application-level virtual machine managers. This type of VMM is simply another process run and managed by the host, and even the host does not know virtualization is happening within the VMM.

Type 2 hypervisors have limits not associated with some of the other types. For example, a user needs administrative privileges to access many of the hardware assistance features of modern CPUs. If the VMM is being run by a standard user without additional privileges, the VMM cannot take advantage of these features. Due to this limitation, as well as the extra overhead of running a general-purpose operating system as well as guest operating systems, type 2 hypervisors tend to have poorer overall performance than type 0 or 1.

As is often the case, the limitations of type 2 hypervisors also provide some benefits. They run on a variety of general-purpose operating systems, and running them requires no changes to the host operating system. A student can use a type 2 hypervisor, for example, to test a non-native operating system without replacing the native operating system. In fact, on an Apple laptop, a student could have versions of Windows, Linux, Unix, and less common operating systems all available for learning and experimentation.

16.5.5 Paravirtualization

As we've seen, paravirtualization takes a different tack than the other types of virtualization. Rather than try to trick a guest operating system into believing it has a system to itself, paravirtualization presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the paravirtualized virtual hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

The Xen VMM, which is the leader in paravirtualization, has implemented several techniques to optimize the performance of guests as well as of the host system. For example, as we have seen, some VMMs present virtual devices to guests that appear to be real devices. Instead of taking that approach, the Xen VMM presents clean and simple device abstractions that allow efficient I/O, as well as good communication between the guest and the VMM about device I/O. For each device used by each guest, there is a circular buffer shared by the guest and the VMM via shared memory. Read and write data are placed in this buffer, as shown in Figure 16.6.

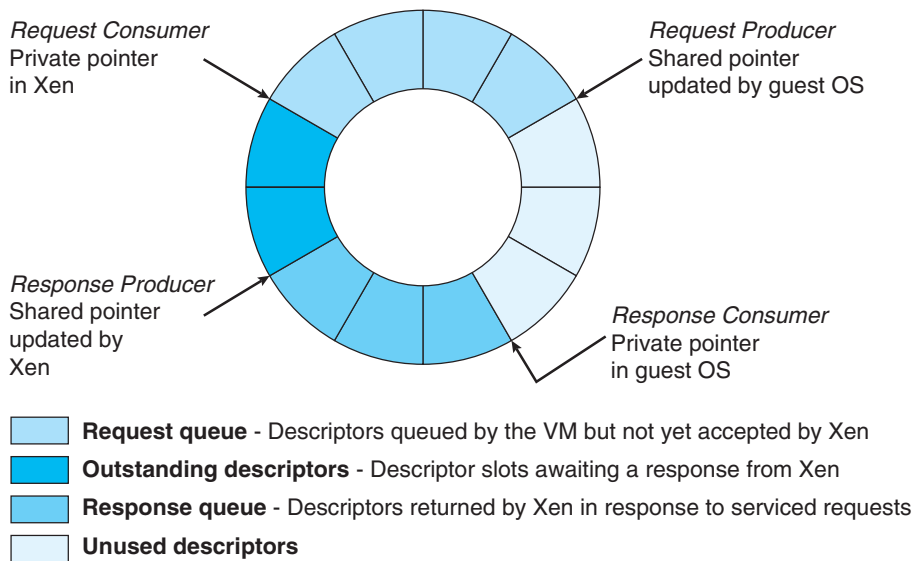


Figure 16.6 Xen I/O via shared circular buffer.

For memory management, Xen does not implement nested page tables. Rather, each guest has its own set of page tables, set to read-only. Xen requires the guest to use a specific mechanism, a **hypercall** from the guest to the hypervisor VMM, when a page-table change is needed. This means that the guest operating system's kernel code must be changed from the default code to these Xen-specific methods. To optimize performance, Xen allows the guest to queue up multiple page-table changes asynchronously via hypercalls and then check to ensure that the changes are complete before continuing operation.

Xen allowed virtualization of x86 CPUs without the use of binary translation, instead requiring modifications in the guest operating systems like the one described above. Over time, Xen has taken advantage of hardware features supporting virtualization. As a result, it no longer requires modified guests and essentially does not need the paravirtualization method. Paravirtualization is still used in other solutions, however, such as type 0 hypervisors.

16.5.6 Programming-Environment Virtualization

Another kind of virtualization, based on a different execution model, is the virtualization of programming *environments*. Here, a programming language is designed to run within a custom-built virtualized environment. For example, Oracle's Java has many features that depend on its running in the **Java virtual machine (JVM)**, including specific methods for security and memory management.

If we define virtualization as including only duplication of hardware, this is not really virtualization at all. But we need not limit ourselves to that definition. Instead, we can define a virtual environment, based on APIs, that provides a set of features that we want to have available for a particular language and programs written in that language. Java programs run within the JVM

environment, and the JVM is compiled to be a native program on systems on which it runs. This arrangement means that Java programs are written once and then can run on any system (including all of the major operating systems) on which a JVM is available. The same can be said for **interpreted languages**, which run inside programs that read each instruction and interpret it into native operations.

16.5.7 Emulation

Virtualization is probably the most common method for running applications designed for one operating system on a different operating system, but on the same CPU. This method works relatively efficiently because the applications were compiled for the same instruction set as the target system uses.

But what if an application or operating system needs to run on a different CPU? Here, it is necessary to translate all of the source CPU's instructions so that they are turned into the equivalent instructions of the target CPU. Such an environment is no longer virtualized but rather is fully emulated.

Emulation is useful when the host system has one system architecture and the guest system was compiled for a different architecture. For example, suppose a company has replaced its outdated computer system with a new system but would like to continue to run certain important programs that were compiled for the old system. The programs could be run in an emulator that translates each of the outdated system's instructions into the native instruction set of the new system. Emulation can increase the life of programs and allow us to explore old architectures without having an actual old machine.

As may be expected, the major challenge of emulation is performance. Instruction-set emulation can run an order of magnitude slower than native instructions, because it may take ten instructions on the new system to read, parse, and simulate an instruction from the old system. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run more slowly than it did on its native hardware. Another challenge for emulator writers is that it is difficult to create a correct emulator because, in essence, this task involves writing an entire CPU in software.

In spite of these challenges, emulation is very popular, particularly in gaming circles. Many popular video games were written for platforms that are no longer in production. Users who want to run those games frequently can find an emulator of such a platform and then run the game unmodified within the emulator. Modern systems are so much faster than old game consoles that even the Apple iPhone has game emulators and games available to run within them.

16.5.8 Application Containment

The goal of virtualization in some instances is to provide a method to segregate applications, manage their performance and resource use, and create an easy way to start, stop, move, and manage them. In such cases, perhaps full-fledged virtualization is not needed. If the applications are all compiled for the same operating system, then we do not need complete virtualization to provide these features. We can instead use application containment.

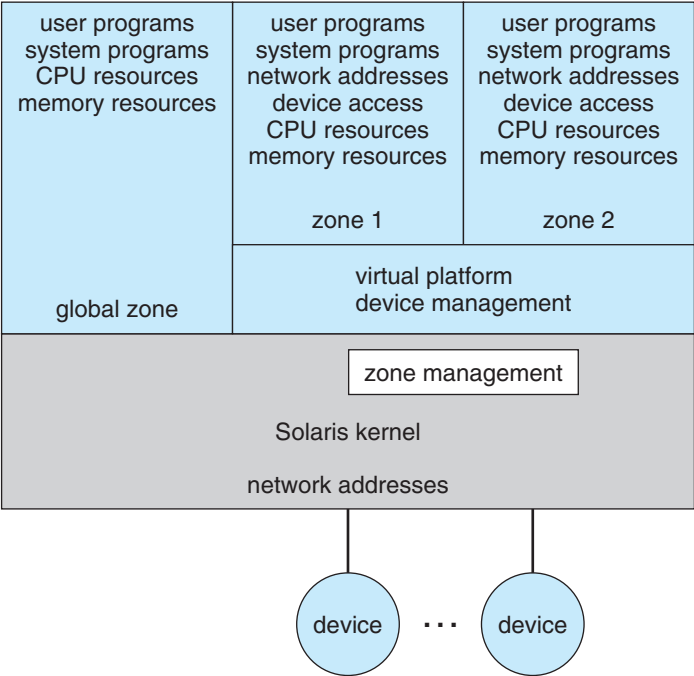


Figure 16.7 Solaris 10 with two zones.

Consider one example of application containment. Starting with version 10, Oracle Solaris has included **containers**, or **zones**, that create a virtual layer between the operating system and the applications. In this system, only one kernel is installed, and the hardware is not virtualized. Rather, the operating system and its devices are virtualized, providing processes within a zone with the impression that they are the only processes on the system. One or more containers can be created, and each can have its own applications, network stacks, network address and ports, user accounts, and so on. CPU and memory resources can be divided among the zones and the system-wide processes. Each zone in fact can run its own scheduler to optimize the performance of its applications on the allotted resources. Figure 16.7 shows a Solaris 10 system with two containers and the standard “global” user space.

16.6 Virtualization and Operating-System Components

Thus far, we have explored the building blocks of virtualization and the various types of virtualization. In this section, we take a deeper dive into the operating-system aspects of virtualization, including how the VMM provides core operating-system functions like scheduling, I/O, and memory management. Here, we answer questions such as these: How do VMMs schedule CPU use when guest operating systems believe they have dedicated CPUs? How can memory management work when many guests require large amounts of memory?

16.6.1 CPU Scheduling

A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system. The virtualization software presents one or more virtual CPUs to each of the virtual machines running on the system and then schedules the use of the physical CPUs among the virtual machines.

The significant variations among virtualization technologies make it difficult to summarize the effect of virtualization on scheduling. First, let's consider the general case of VMM scheduling. The VMM has a number of physical CPUs available and a number of threads to run on those CPUs. The threads can be VMM threads or guest threads. Guests are configured with a certain number of virtual CPUs at creation time, and that number can be adjusted throughout the life of the VM. When there are enough CPUs to allocate the requested number to each guest, the VMM can treat the CPUs as dedicated and schedule only a given guest's threads on that guest's CPUs. In this situation, the guests act much like native operating systems running on native CPUs.

Of course, in other situations, there may not be enough CPUs to go around. The VMM itself needs some CPU cycles for guest management and I/O management and can steal cycles from the guests by scheduling its threads across all of the system CPUs, but the impact of this action is relatively minor. More difficult is the case of **overcommitment**, in which the guests are configured for more CPUs than exist in the system. Here, a VMM can use standard scheduling algorithms to make progress on each thread but can also add a fairness aspect to those algorithms. For example, if there are six hardware CPUs and 12 guest-allocated CPUs, the VMM could allocate CPU resources proportionally, giving each guest half of the CPU resources it believes it has. The VMM can still present all 12 virtual CPUs to the guests, but in mapping them onto physical CPUs, the VMM can use its scheduler to share them appropriately.

Even given a scheduler that provides fairness, any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively affected by virtualization. Consider a time-sharing operating system that tries to allot 100 milliseconds to each time slice to give users a reasonable response time. Within a virtual machine, this operating system is at the mercy of the virtualization system as to what CPU resources it actually receives. A given 100-millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, resulting in very poor response times for users logged into that virtual machine. The effect on a real-time operating system can be even more serious.

The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all of the cycles and indeed that they are scheduling all of those cycles. Commonly, the time-of-day clocks in virtual machines are incorrect because timers take longer to trigger than they would on dedicated CPUs. Virtualization can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines.

To correct for this, a VMM will have an application available for each type of operating system that system administrators install into the guests. This

application corrects clock drift and can have other functions such as virtual device management.

16.6.2 Memory Management

Efficient memory use in general-purpose operating systems is one of the major keys to performance. In virtualized environments, there are more users of memory (the guests and their applications, as well as the VMM), leading to more pressure on memory use. Further adding to this pressure is that VMMs typically overcommit memory, so that the total memory with which guests are configured exceeds the amount of memory that physically exists in the system. The extra need for efficient memory use is not lost on the implementers of VMMs, who take great measures to ensure the optimal use of memory.

For example, VMware ESX uses at least three methods of memory management. Before memory optimization can occur, the VMM must establish how much real memory each guest should use. To do that, the VMM first evaluates the maximum memory size of each guest as dictated when it is configured. General-purpose operating systems do not expect the amount of memory in the system to change, so VMMs must maintain the illusion that the guest has that amount of memory. Next, the VMM computes a target real memory allocation for each guest based on the configured memory for that guest and other factors, such as overcommitment and system load. It then uses the three low-level mechanisms below to reclaim memory from the guests. The overall effect is to enable guests to behave and perform as if they had the full amount of memory requested although in reality they have less.

1. Recall that a guest believes it controls memory allocation via its page-table management, whereas in reality the VMM maintains a nested page table that re-translates the guest page table to the real page table. The VMM can use this extra level of indirection to optimize the guest's use of memory without the guest's knowledge or help. One approach is to provide double paging, in which the VMM has its own page-replacement algorithms and pages to backing-store pages that the guest believes are in physical memory. Of course, the VMM has knows less about the guest's memory access patterns than the guest does, so its paging is less efficient, creating performance problems. VMMs do use this method when other methods are not available or are not providing enough free memory. However, it is not the preferred approach.
2. A common solution is for the VMM to install in each guest a pseudo-device driver or kernel module that it controls. (A **pseudo-device driver** uses device-driver interfaces, appearing to the kernel to be a device driver, but does not actually control a device. Rather, it is an easy way to add kernel-mode code without directly modifying the kernel.) This **balloon** memory manager communicates with the VMM and is told to allocate or deallocate memory. If told to allocate, it allocates memory and tells the operating system to pin the allocated pages into physical memory. Recall that pinning locks a page into physical memory so that it cannot be moved or paged out. The guest sees memory pressure because of these pinned pages, essentially decreasing the amount of physical memory it has available to use. The guest then may free up other physical memory

to be sure it has a sufficient pool of free memory. Meanwhile, the VMM, knowing that the pages pinned by the balloon process will never be used, removes those physical pages from the guest and allocates them to another guest. At the same time, the guest is using its own memory-management and paging algorithms to manage the available memory, which is the most efficient option. If memory pressure within the entire system decreases, the VMM will tell the balloon process within the guest to unpin and free some or all of the memory, allowing the guest more pages for its use.

3. Another common method for reducing memory pressure is for the VMM to determine if the same page has been loaded more than once. If this is the case, to the VMM reduces the number of copies of the page to one and maps the other users of the page to that one copy. VMware, for example, randomly samples guest memory and creates a hash for each page sampled. That hash value is a “thumbprint” of the page. The hash of every page examined is compared with other hashes already stored in a hash table. If there is a match, the pages are compared byte by byte to see if they really are identical. If they are, one page is freed, and its logical address is mapped to the other’s physical address. This technique might seem at first to be ineffective, but consider that guests run operating systems. If multiple guests run the same operating system, then only one copy of the active operating-system pages need be in memory. Similarly, multiple guests could be running the same set of applications, again a likely source of memory sharing.

16.6.3 I/O

In the area of I/O, hypervisors have some leeway and can be less concerned with exactly representing the underlying hardware to their guests. Because of all the variation in I/O devices, operating systems are used to dealing with varying and flexible I/O mechanisms. For example, operating systems have a device-driver mechanism that provides a uniform interface to the operating system whatever the I/O device. Device-driver interfaces are designed to allow third-party hardware manufacturers to provide device drivers connecting their devices to the operating system. Usually, device drivers can be dynamically loaded and unloaded. Virtualization takes advantage of such built-in flexibility by providing specific virtualized devices to guest operating systems.

As described in Section 16.5, VMMs vary greatly in how they provide I/O to their guests. I/O devices may be dedicated to guests, for example, or the VMM may have device drivers onto which it maps guest I/O. The VMM may also provide idealized device drivers to guests, which allows easy provision and management of guest I/O. In this case, the guest sees an easy-to-control device, but in reality that simple device driver communicates to the VMM which sends those requests to a more complicated real device through a more complex real device driver. I/O in virtual environments is complicated and requires careful VMM design and implementation.

Consider the case of a hypervisor and hardware combination that allows devices to be dedicated to a guest and allows the guest to access those devices directly. Of course, a device dedicated to one guest is not available to any other guests, but this direct access can still be useful in some circumstances.

The reason to allow direct access is to improve I/O performance. The less the hypervisor has to do to enable I/O for its guests, the faster the I/O can occur. With Type 0 hypervisors that provide direct device access, guests can often run at the same speed as native operating systems. When type 0 hypervisors instead provide shared devices, performance can suffer by comparison.

With direct device access in type 1 and 2 hypervisors, performance can be similar to that of native operating systems if certain hardware support is present. The hardware needs to provide DMA pass-through with facilities like VT-d, as well as direct interrupt delivery to specific guests. Given how frequently interrupts occur, it should be no surprise that the guests on hardware without these features have worse performance than if they were running natively.

In addition to direct access, VMMs provide shared access to devices. Consider a disk drive to which multiple guests have access. The VMM must provide protection while sharing the device, assuring that a guest can access only the blocks specified in the guest's configuration. In such instances, the VMM must be part of every I/O, checking it for correctness as well as routing the data to and from the appropriate devices and guests.

In the area of networking, VMMs also have work to do. General-purpose operating systems typically have one Internet protocol (IP) address, although they sometimes have more than one—for example, to connect to a management network, backup network, and production network. With virtualization, each guest needs at least one IP address, because that is the guest's main mode of communication. Therefore, a server running a VMM may have dozens of addresses, and the VMM acts as a virtual switch to route the network packets to the addressed guest.

The guests can be “directly” connected to the network by an IP address that is seen by the broader network (this is known as **bridging**). Alternatively, the VMM can provide a **network address translation (NAT)** address. The NAT address is local to the server on which the guest is running, and the VMM provides routing between the broader network and the guest. The VMM also provides firewalling, moderating connections between guests within the system and between guests and external systems.

16.6.4 Storage Management

An important question in determining how virtualization works is this: If multiple operating systems have been installed, what and where is the boot disk? Clearly, virtualized environments need to approach the area of storage management differently from native operating systems. Even the standard multiboot method of slicing the root disk into partitions, installing a boot manager in one partition, and installing each other operating system in another partition is not sufficient, because partitioning has limits that would prevent it from working for tens or hundreds of virtual machines.

Once again, the solution to this problem depends on the type of hypervisor. Type 0 hypervisors do tend to allow root disk partitioning, partly because these systems tend to run fewer guests than other systems. Alternatively, they may have a disk manager as part of the control partition, and that disk manager provides disk space (including boot disks) to the other partitions.

Type 1 hypervisors store the guest root disk (and configuration information) in one or more files within the file systems provided by the VMM. Type 2 hypervisors store the same information within the host operating system's file systems. In essence, a **disk image**, containing all of the contents of the root disk of the guest, is contained within one file in the VMM. Aside from the potential performance problems that causes, it is a clever solution, because it simplifies copying and moving guests. If the administrator wants a duplicate of the guest (for testing, for example), she simply copies the associated disk image of the guest and tells the VMM about the new copy. Booting that new VM brings up an identical guest. Moving a virtual machine from one system to another that runs the same VMM is as simple as halting the guest, copying the image to the other system, and starting the guest there.

Guests sometimes need more disk space than is available in their root disk image. For example, a nonvirtualized database server might use several file systems spread across many disks to store various parts of the database. Virtualizing such a database usually involves creating several files and having the VMM present those to the guest as disks. The guest then executes as usual, with the VMM translating the disk I/O requests coming from the guest into file I/O commands to the correct files.

Frequently, VMMs provide a mechanism to capture a physical system as it is currently configured and convert it to a guest that the VMM can manage and run. Based on the discussion above, it should be clear that this **physical-to-virtual (P-to-V)** conversion reads the disk blocks of the physical system's disks and stores them within files on the VMM's system or on shared storage that the VMM can access. Perhaps not as obvious is the need for a **virtual-to-physical (V-to-P)** procedure for converting a guest to a physical system. This step is sometimes needed for debugging: a problem could be caused by the VMM or associated components, and the administrator could attempt to solve the problem by removing virtualization from the problem variables. V-to-P conversion can take the files containing all of the guest data and generate disk blocks on a system's disk, recreating the guest as a native operating system and applications. Once the testing is concluded, the native system can be reused for other purposes when the virtual machine returns to service, or the virtual machine can be deleted and the native system can continue to run.

16.6.5 Live Migration

One feature not found in general-purpose operating systems but found in type 0 and type 1 hypervisors is the live migration of a running guest from one system to another. We mentioned this capability earlier. Here, we explore the details of how live migration works and why VMMs have a relatively easy time implementing it while general-purpose operating systems, in spite of some research attempts, do not.

First, consider how live migration works. A running guest on one system is copied to another system running the same VMM. The copy occurs with so little interruption of service that users logged in to the guest, and network connections to the guest, continue without noticeable impact. This rather astonishing ability is very powerful in resource management and hardware administration. After all, compare it with the steps necessary without virtualization: warning users, shutting down the processes, possibly moving the

binaries, and restarting the processes on the new system, with users only then able to use the services again. With live migration, an overloaded system can have its load decreased live with no discernible disruption. Similarly, a system needing hardware or system changes (for example, a firmware upgrade, hardware addition or removal, or hardware repair) can have guests migrated off, the work done, and guests migrated back without noticeable impact on users or remote connections.

Live migration is made possible because of the well-defined interfaces between guests and VMMs and the limited state the VMM maintains for the guest. The VMM migrates a guest via the following steps:

1. The source VMM establishes a connection with the target VMM and confirms that it is allowed to send a guest.
2. The target creates a new guest by creating a new VCPU, new nested page table, and other state storage.
3. The source sends all read-only memory pages to the target.
4. The source sends all read-write pages to the target, marking them as clean.
5. The source repeats step 4, as during that step some pages were probably modified by the guest and are now dirty. These pages need to be sent again and marked again as clean.
6. When the cycle of steps 4 and 5 becomes very short, the source VMM freezes the guest, sends the VCPU's final state, sends other state details, sends the final dirty pages, and tells the target to start running the guest. Once the target acknowledges that the guest is running, the source terminates the guest.

This sequence is shown in Figure 16.8.

We conclude this discussion with a few interesting details and limitations concerning live migration. First, for network connections to continue uninterrupted, the network infrastructure needs to understand that a MAC

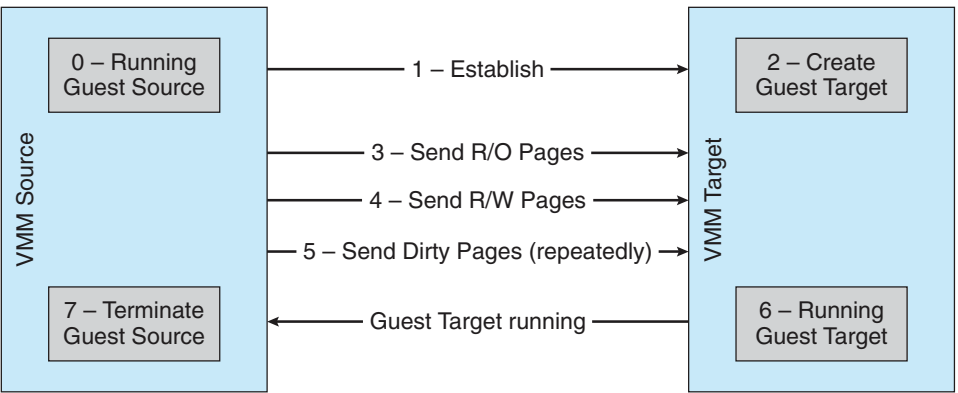


Figure 16.8 Live migration of a guest between two servers.

address—the hardware networking address—can move between systems. Before virtualization, this did not happen, as the MAC address was tied to physical hardware. With virtualization, the MAC must be movable for existing networking connections to continue without resetting. Modern network switches understand this and route traffic wherever the MAC address is, even accommodating a move.

A limitation of live migration is that no disk state is transferred. One reason live migration is possible is that most of the guest's state is maintained within the guest—for example, open file tables, system-call state, kernel state, and so on. Because disk I/O is so much slower than memory access, and used disk space is usually much larger than used memory, disks associated with the guest cannot be moved as part of a live migration. Rather, the disk must be remote to the guest, accessed over the network. In that case, disk access state is maintained within the guest, and network connections are all that matter to the VMM. The network connections are maintained during the migration, so remote disk access continues. Typically, NFS, CIFS, or iSCSI is used to store virtual machine images and any other storage a guest needs access to. Those network-based storage accesses simply continue when the network connections are continued once the guest has been migrated.

Live migration enables entirely new ways of managing data centers. For example, virtualization management tools can monitor all the VMMs in an environment and automatically balance resource use by moving guests between the VMMs. They can also optimize the use of electricity and cooling by migrating all guests off selected servers if other servers can handle the load and powering down the selected servers entirely. If the load increases, these tools can power up the servers and migrate guests back to them.

16.7 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming into fashion as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: the VMware Workstation and the Java virtual machine. As you will see, these virtual machines can typically run on top of operating systems of any of the design types discussed in earlier chapters. Thus, operating-system design methods—simple layers, microkernels, modules, and virtual machines—are not mutually exclusive.

16.7.1 VMware

VMware Workstation is a popular commercial application that abstracts Intel X86 and compatible hardware into isolated virtual machines. VMware Workstation is a prime example of a Type 2 hypervisor. It runs as an application on a host operating system such as Windows or Linux and allows this host system to run several different guest operating systems concurrently as independent virtual machines.

The architecture of such a system is shown in Figure 16.9. In this scenario, Linux is running as the host operating system, and FreeBSD, Windows NT, and

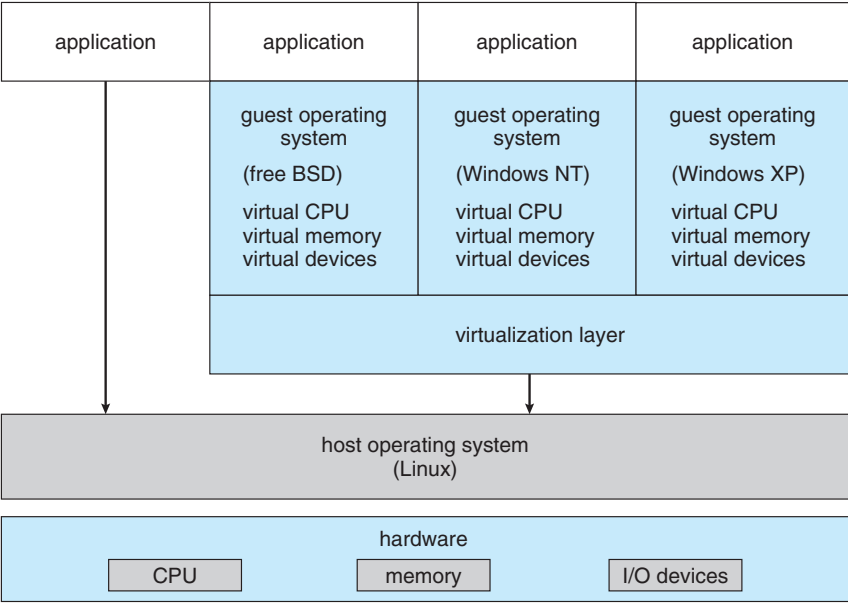


Figure 16.9 VMware Workstation architecture.

Windows XP are running as guest operating systems. At the heart of VMware is the virtualization layer, which abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

The physical disk that the guest owns and manages is really just a file within the file system of the host operating system. To create an identical guest, we can simply copy the file. Copying the file to another location protects the guest against a disaster at the original site. Moving the file to another location moves the guest system. These scenarios show how virtualization can improve the efficiency of system administration as well as system resource use.

16.7.2 The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java provides a specification for a Java virtual machine, or JVM. Java therefore is an example of programming-environment virtualization, as discussed in Section 16.5.6.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 16.10. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the `.class` file is valid Java bytecode and that it does not overflow or underflow the stack. It also

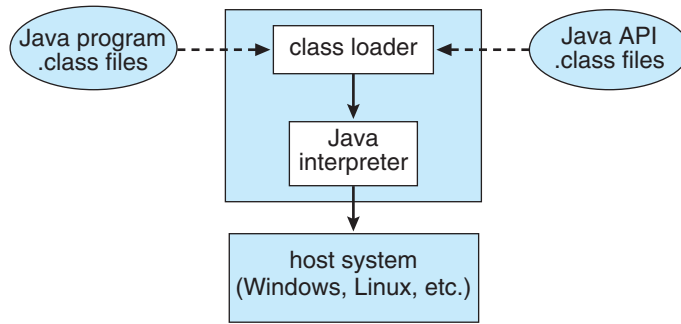


Figure 16.10 The Java virtual machine.

ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a Web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a **just-in-time (JIT)** compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions, and the bytecode operations need not be interpreted all over again. Running the JVM in hardware is potentially even faster. Here, a special Java chip executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

16.8 Summary

Virtualization is a method of providing a guest with a duplicate of a system's underlying hardware. Multiple guests can run on a given system, each believing it is the native operating system in full control of the system. Virtualization started as a method to allow IBM to segregate users and provide them with their own execution environments on IBM mainframes. Since then, with improvements in system and CPU performance and through innovative software techniques, virtualization has become a common feature in data centers and even on personal computers. Because of the popularity of virtualization, CPU designers have added features to support virtualization. This snowball effect is likely to continue, with virtualization and its hardware support increasing over time.

Type 0 virtualization is implemented in the hardware and requires modifications to the operating system to ensure proper operation. These modifications

offer an example of paravirtualization, in which the operating system is not blind to virtualization but instead has features added and algorithms changed to improve virtualization's features and performance. In Type 1 virtualization, a host virtual machine monitor (VMM) provides the environment and features needed to create, run, and destroy guest virtual machines. Each guest includes all of the software typically associated with a full native system, including the operating system, device drivers, applications, user accounts, and so on.

Type 2 hypervisors are simply applications that run on other operating systems, which do not know that virtualization is taking place. These hypervisors do not enjoy hardware or host support so must perform all virtualization activities in the context of a process.

Other facilities that are similar to virtualization but do not meet the full definition of replicating hardware exactly are also common. Programming-environment virtualization is part of the design of a programming language. The language specifies a containing application in which programs run, and this application provides services to the programs. Emulation is used when a host system has one architecture and the guest was compiled for a different architecture. Every instruction the guest wants to execute must be translated from its instruction set to that of the native hardware. Although this method involves some performance penalty, it is balanced by the usefulness of being able to run old programs on newer, incompatible hardware or run games designed for old consoles on modern hardware.

Implementing virtualization is challenging, especially when hardware support is minimal. Some hardware support must exist for virtualization, but the more features provided by the system, the easier virtualization is to implement and the better the performance of the guests. VMMs take advantage of whatever hardware support is available when optimizing CPU scheduling, memory management, and I/O modules to provide guests with optimum resource use while protecting the VMM from the guests and the guests from one another.

Exercises

- 16.1 Describe the three types of traditional virtualization.
- 16.2 Describe the four virtualization-like execution environments and why they are not “true” virtualization.
- 16.3 Describe four benefits of virtualization.
- 16.4 Why can VMMs not implement trap-and-emulate-based virtualization on some CPUs? Lacking the ability to trap-and-emulate, what method can a VMM use to implement virtualization?
- 16.5 What hardware assistance for virtualization can be provided by modern CPUs?
- 16.6 Why is live migration possible in virtual environments but much less possible for a native operating system?

Bibliographical Notes

The original IBM VM system was described in [Meyer and Seawright (1970)]. [Popek and Goldberg (1974)] established the characteristics that help define VMMs. Methods of implementing virtual machines are discussed in [Agesen et al. (2010)].

Virtualization has been an active research area for many years. Disco was one of the first attempts to use virtualization to enforce logical isolation and provide scalability on multicore systems ([Bugnion et al. (1997)]). Based on that and other work, Quest-V used virtualization to create an entire distributed operating system within a multicore system ([Li et al. (2011)]).

Intel x86 hardware virtualization support is described in [Neiger et al. (2006)]. AMD hardware virtualization support is described in a white paper (<http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf>).

KVM is described in [Kivity et al. (2007)]. Xen is described in [Barham et al. (2003)]. Oracle Solaris containers are similar to BSD jails, as described in [Poul-henning Kamp (2000)].

[Agesen et al. (2010)] discuss the performance of binary translation. Memory management in VMware is described in [Waldspurger (2002)]. The problem of I/O overhead in virtualized environments has a proposed solution in [Gordon et al. (2012)]. Some protection challenges and attacks in virtual environments are discussed in [Wojtczuk and Ruthkowska (2011)].

Live process migration research occurred in the 1980s and was first discussed in [Powell and Miller (1983)]. Problems identified in that research left migration in a functionally limited state, as described in [Milojicic et al. (2000)]. VMware realized that virtualization could allow functional live migration and described prototype work in [Chandra et al. (2002)]. VMware shipped the vMotion live migration feature as part of VMware vCenter, as described in VMware VirtualCenter User's Manual Version 1.0 (http://www.vmware.com/pdf/VirtualCenter_Users_Manual.pdf). The details of the implementation of a similar feature in the Xen VMM are found in [Clark et al. (2005)].

Research showing that, without interrupt remapping, malicious guests can generate interrupts that can be used to gain control of the host system is discussed in [Wojtczuk and Ruthkowska (2011)].

Bibliography

- [Agesen et al. (2010)] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, "The Evolution of an x86 Virtual Machine Monitor", *Proceedings of the ACM Symposium on Operating Systems Principles* (2010), pages 3–18.
- [Barham et al. (2003)] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization", *Proceedings of the ACM Symposium on Operating Systems Principles* (2003), pages 164–177.

- [Bugnion et al. (1997)] E. Bugnion, S. Devine, and M. Rosenblum, “Disco: Running Commodity Operating Systems on Scalable Multiprocessors”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1997), pages 143–156.
- [Chandra et al. (2002)] R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum, “Optimizing the Migration of Virtual Computers” (2002), pages 377–390.
- [Clark et al. (2005)] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live Migration of Virtual Machines”, *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation* (2005), pages 273–286.
- [Gordon et al. (2012)] A. Gordon, N. A. N. Har’El, M. Ben-Yehuda, A. Landau, and A. S. and Dan Tsafrir, “ELI: Bare-metal Performance for I/O Virtualization”, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pages 411–422.
- [Kivity et al. (2007)] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux Virtual Machine Monitor”, *Proceedings of the Linux Symposium* (2007).
- [Li et al. (2011)] Y. Li, M. Danish, and R. West, “Quest-V: A Virtualized Multikernel for High-Confidence Systems”, Technical report, Boston University (2011).
- [Meyer and Seawright (1970)] R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System”, *IBM Systems Journal*, Volume 9, Number 3 (1970), pages 199–218.
- [Milojicic et al. (2000)] D. S. Milojicic, F. Douglass, Y. Paindaveine, R. Wheeler, and S. Zhou, “Process Migration”, *ACM Computing Surveys*, Volume 32, Number 3 (2000), pages 241–299.
- [Neiger et al. (2006)] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, “Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization”, *Intel Technology Journal*, Volume 10, (2006).
- [Popek and Goldberg (1974)] G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, *Communications of the ACM*, Volume 17, Number 7 (1974), pages 412–421.
- [Poul-henning Kamp (2000)] R. N. M. W. Poul-henning Kamp, “Jails: Confining the Omnipotent Root”, *Proceedings of the 2nd International System Administration and Networking Conference* (2000).
- [Powell and Miller (1983)] M. Powell and B. Miller, “Process Migration in DEMOS/MP”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1983).
- [Waldspurger (2002)] C. Waldspurger, “Memory Resource Management in VMware ESX Server”, *Operating Systems Review*, Volume 36, Number 4 (2002), pages 181–194.
- [Wojtczuk and Ruthkowska (2011)] R. Wojtczuk and J. Ruthkowska, “Following the White Rabbit: Software Attacks Against Intel VT-d Technology”, *The Invisible Things Lab’s blog* (2011).