CHAPTER 14

Virtual Machines

Learning Objectives

After studying this chapter, you should be able to:

- Discuss Type 1 and Type 2 virtualization.
- Explain container virtualization and compare it to the hypervisor approach.
- Understand the processor issues involved in implementing a virtual machine.
- Understand the memory management issues involved in implementing a virtual machine.
- Understand the I/O management issues involved in implementing a virtual machine.
- Compare and contrast VMware ESXi, Hyper-V, Xen, and Java VM.
- Explain the operation of the Linux virtual machine.

This chapter focuses on the application of virtualization to operating system design. Virtualization encompasses a variety of technologies for managing computing resources by providing a software translation layer, known as an abstraction layer, between the software and the physical hardware. Virtualization turns physical resources into logical, or virtual, resources. Virtualization enables users, applications, and management software operating above the abstraction layer to manage and use resources without needing to be aware of the physical details of the underlying resources.

The first three sections of this chapter deal with the two main approaches to virtualization: virtual machines and containers. The remainder of the chapter looks at some specific systems.

14.1 VIRTUAL MACHINE CONCEPTS

Traditionally, applications have run directly on an operating system (OS) on a personal computer (PC) or on a server, with the PC or server running only one OS at a time. Thus, the application vendor had to rewrite parts of its applications for each OS/platform they would run on and support, which increased time to market for new features/functions, increased the likelihood of defects, increased quality testing efforts, and usually led to increased price. To support multiple OSs, application vendors needed to create, manage, and support multiple hardware and OS infrastructures, a costly and resource-intensive process. One effective strategy for dealing with this problem is known as **hardware virtualization**. Virtualization technology enables a single PC or server to simultaneously run multiple OSs or multiple sessions of a single OS. A machine with virtualization software can host numerous applications, including those that run on different OSs, on a single platform. In essence, the host OS can support a number of **virtual machines (VMs)**, each of that has the characteristics of a particular OS and, in some versions of virtualization, the characteristics of a particular hardware platform. A VM is also referred to as a *system virtual machine*, emphasizing that it is the hardware of the system that is being virtualized.

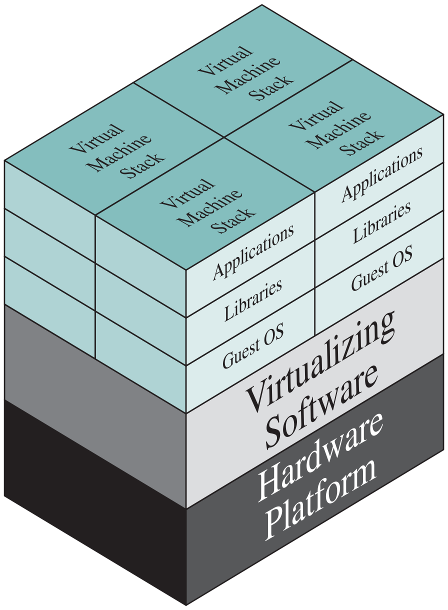

Figure 14.1 Virtual Machine Concept

Virtualization is not a new technology. During the 1970s, IBM mainframe systems offered the first capabilities that would allow programs to use only a portion of a system's resources. Various forms of that ability have been available on platforms since that time. Virtualization came into mainstream computing in the early 2000s when the technology was commercially available on x86 servers. Organizations were suffering from a surfeit of servers due to a Microsoft Windows-driven "one application, one server" strategy. Moore's Law drove rapid hardware improvements outpacing software's ability, and most of these servers were vastly underutilized, often consuming less than 5% of the available resources in each server. In addition, this overabundance of servers filled datacenters and consumed vast amounts of power and cooling, straining a corporation's ability to manage and maintain their infrastructure. Virtualization helped relieve this stress.

The solution that enables virtualization is a **virtual machine monitor (VMM)**, or commonly known today as a **hypervisor**. This software sits between the hardware and the VMs acting as a resource broker. Simply put, it allows multiple VMs to safely coexist on a single physical server host and share that host's resources. Figure 14.1 illustrates this type of virtualization in general terms. On top of the hardware platform sits some sort of virtualizing software, which may consist of the host OS plus specialized virtualizing software or a simply a software package that includes host OS functions and virtualizing functions, as explained subsequently. The virtualizing software provides abstraction of all physical resources (such as processor, memory, network, and storage) and thus enables multiple computing stacks, called virtual machines, to be run on a single physical host.

Each VM includes an OS, called the guest OS. This OS may be the same as the host OS or a different one. For example, a guest Windows OS could be run in a VM on top of a Linux host OS. The guest OS, in turn, supports a set of standard library functions and other binary files and applications. From the point of view of the applications and the user, this stack appears as an actual machine, with hardware and an OS; thus, the term *virtual machine* is appropriate. In other words, it is the hardware that is being virtualized.

The number of guests that can exist on a single host is measured a **consolidation ratio**. For example, a host that is supporting 4 VMs is said to have a consolidation ratio of 4 to 1, also written as 4:1 (see Figure 14.1). The initial commercially available hypervisors provided consolidation ratios of between 4:1 and 12:1, but even at the low end, if a company virtualized all of their servers, they could remove 75% of the servers from their datacenters. More importantly, they could remove the cost as well, which often ran into the millions or tens of millions of dollars annually. With fewer physical servers, less power and less cooling was needed. Also this leads to fewer cables, fewer network switches, and less floor space. Server consolidation became, and continues to be, a tremendously valuable way to solve a costly and wasteful problem. Today, more virtual servers are deployed in the world than physical servers, and virtual server deployment continues to accelerate.

We can summarize the key reasons the organizations use virtualization as follows:

- **Legacy hardware**: Applications built for legacy hardware can still be run by virtualizing (emulating) the legacy hardware, enabling the retirement of the old hardware.
- **Rapid deployment**: As discussed subsequently, whereas it may take weeks or longer to deploy new servers in an infrastructure, a new VM may be deployed in a matter of minutes. As explained subsequently, a VM consists of files. By duplicating those files, in a virtual environment there is a perfect copy of the server available.
- **Versatility**: Hardware usage can be optimized by maximizing the number of kinds of applications that a single computer can handle.
- **Consolidation**: A large-capacity or high-speed resource, such a server can be used more efficiently by sharing the resource among multiple applications simultaneously.
- **Aggregating**: Virtualization makes it easy to combine multiple resources in to one virtual resource, such as in the case of storage virtualization.
- **Dynamics**: With the use of virtual machines, hardware resources can be easily allocated in a dynamic fashion. This enhances load balancing and fault tolerance.
- **Ease of management**: Virtual machines facilitate deployment and testing of software.
- **Increased availability**: Virtual machine hosts are clustered together to form pools of compute resources. Multiple VMs are hosted on each of these servers and, in the case of a physical server failure, the VMs on the failed host can be quickly and automatically restarted on another host in the cluster. Compared with providing this type of availability for a physical server, virtual environments can provide higher availability at significantly less cost and with less complexity.

Commercial VM offerings by companies such as VMware and Microsoft are widely used on servers, with millions of copies having been sold. A key aspect of server virtualization is, in addition to the capability of running multiple VMs on one machine, VMs can be viewed as network resources. Server virtualization masks server resources, including the number and identity of individual physical servers, processors, and OSs, from server users. This makes it possible to partition a single host into multiple independent servers, conserving hardware resources. It also makes it possible to quickly migrate a server from one machine to another for load balancing, or for dynamic switchover in the case of machine failure. Server virtualization has become a central element in dealing with "big data" applications and in implementing cloud computing infrastructures.

In addition to their use in server environments, these VM technologies also are used in desktop environments to run multiple OSs, typically Windows and Linux.

14.2 HYPERVISORS

There is no definitive classification of the various approaches that have been taken to the development of virtual machines. Various methods of classification are discussed in [UHLI05], [PEAR13], [RPSE04], [ROSE05], [NAND05], and [GOLD11]. This section examines the concept of a hypervisor, which is the most common basis for classifying virtual machine approaches.

Hypervisors

Virtualization is a form of abstraction. Much like an OS abstracts the disk I/O commands from a user through the use of program layers and interfaces, virtualization abstracts the physical hardware from the virtual machines it supports. The virtual machine monitor or hypervisor is the software that provides this abstraction. It acts as a broker, or traffic cop, acting as a proxy for the guests (VMs) as they request and consume resources of the physical host.

A virtual machine is a software construct that mimics the characteristics of a physical server. It is configured with some number of processors, some amount of RAM, storage resources, and connectivity through the network ports. Once that VM is created, it can be powered on like a physical server, loaded with an OS and software solutions, and utilized in the manner of a physical server. Unlike a physical server, this virtual server only sees the resources it has been configured with, not all of the resources of the physical host itself. This isolation allows a host machine to run many virtual machines, each of them running the same or different copies of an OS, sharing RAM, storage and network bandwidth, without problems. An OS in a virtual machine accesses the resource that is presented to it by the hypervisor. The hypervisor facilitates the translation and I/O from the virtual machine to the physical server devices, and back again to the correct virtual machine. In this way, certain privileged instructions that a "native" OS would be executing on its hosts hardware are trapped and run by the hypervisor as a proxy for the virtual machine. This creates some performance degradation in the virtualization process, though over time both hardware and software improvements have minimalized this overhead.

A VM instance is defined in files. A typical virtual machine can consist of just a few files. There is a configuration file that describes the attributes of the virtual machine. It contains the server definition, how many virtual processors (vCPUs) are allocated to this virtual machine, how much RAM is allocated, to which I/O devices the VM has access, how many network interface cards (NICs) are in the virtual server, and more. It also describes the storage that the VM can access. Often that storage is presented as virtual disks that exist as additional files in the physical file system. When a virtual machine is powered on, or instantiated, additional files are created for logging, for memory paging, and other functions. Because a VM essentially consists of files, certain functions in a virtual environment can be defined simpler and quicker than in a physical environment. Since the earliest days of computers, backing up data has been a critical function. Since VMs are already files, copying them produces not only a backup of the data but also a copy of the entire server, including the OS, applications, and the hardware configuration itself.

A common method to rapidly deploy new VMs is through the use of templates. A template provides a standardized group of hardware and software settings that can be used to create new VMs configured with those settings. Creating a new VM from a template consists of providing unique identifiers for the new VM, and having the provisioning software build a VM from the template and adding in the configuration changes as part of the deployment.

Hypervisor Functions The principal functions performed by a hypervisor are the following:

- **Execution management of VMs**: Includes scheduling VMs for execution, virtual memory management to ensure VM isolation from other VMs, context switching between various processor states. Also includes isolation of VMs to prevent conflicts in resource usage and emulation of timer and interrupt mechanisms.
- **Devices emulation and access control**: Emulating all network and storage (block) devices that different native drivers in VMs are expecting, mediating access to physical devices by different VMs.
- **Execution of privileged operations by hypervisor for guest VMs**: Certain operations invoked by guest OSs, instead of being executed directly by the host hardware, may have to be executed on its behalf by the hypervisor, because of their privileged nature.
- **Management of VMs (also called VM lifecycle management)**: Configuring guest VMs and controlling VM states (e.g., Start, Pause, and Stop).
- **Administration of hypervisor platform and hypervisor software**: Involves setting of parameters for user interactions with the hypervisor host as well as hypervisor software.

Type 1 Hypervisor There are two types of hypervisors, distinguished by whether there is an OS between the hypervisor and the host. A type 1 hypervisor (see Figure 14.2a) is loaded as a software layer directly onto a physical server, much like an OS is loaded. The type 1 hypervisor can directly control the physical resources of

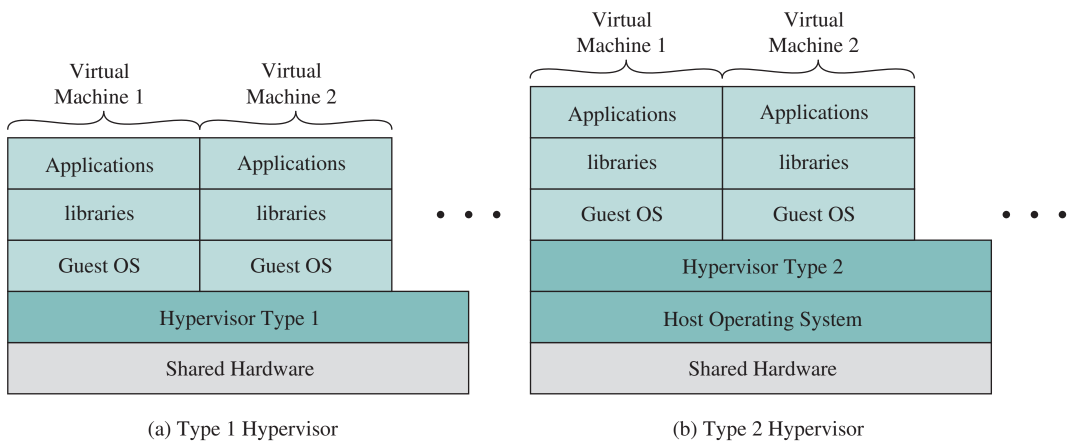

Figure 14.2 Type 1 and Type 2 Hypervisors

the host. Once it is installed and configured, the server is then capable of supporting virtual machines as guests. In mature environments, where virtualization hosts are clustered together for increased availability and load balancing, a hypervisor can be staged on a new host. Then, that new host is joined to an existing cluster, and VMs can be moved to the new host without any interruption of service. Some examples of type 1 hypervisors are VMware ESXi, Microsoft Hyper-V, and the various Xen variants.

Type 2 Hypervisor A type 2 hypervisor exploits the resources and functions of a host OS and runs as a software module on top of the OS (see Figure 14.2b). It relies on the OS to handle all of the hardware interactions on the hypervisor's behalf. Some examples of type 2 hypervisors are VMware Workstation and Oracle VM Virtual Box.

Key differences between the two hypervisor types are as follows:

- Typically, type 1 hypervisors perform better than type 2 hypervisors. Because a type 1 hypervisor doesn't compete for resources with an OS, there are more resources available on the host, and by extension, more virtual machines can be hosted on a virtualization server using a type 1 hypervisor.
- Type 1 hypervisors are also considered to be more secure than the type 2 hypervisors. Virtual machines on a type 1 hypervisor make resource requests that are handled external to that guest, and they cannot affect other VMs or the hypervisor they are supported by. This is not necessarily true for VMs on a type 2 hypervisor, and a malicious guest could potentially affect more than itself.
- Type 2 hypervisors allow a user to take advantage of virtualization without needing to dedicate a server to only that function. Developers who need to run multiple environments as part of their process, in addition to taking advantage of the personal productive workspace that a PC OS provides, can do both with a type 2 hypervisor installed as an application on their LINUX or Windows desktop. The virtual machines that are created and used can be migrated or

copied from one hypervisor environment to another, reducing deployment time and increasing the accuracy of what is deployed, reducing the time to market of a project.

Paravirtualization

As virtualization became more prevalent in corporations, both hardware and software vendors looked for ways to provide even more efficiencies. Unsurprisingly, these paths led to both software-assisted virtualization and hardware-assisted virtualization. **Paravirtualization** is a software-assisted virtualization technique that uses specialized APIs to link virtual machines with the hypervisor to optimize their performance. The OS in the virtual machine, Linux or Microsoft Windows, has specialized paravirtualization support as part of the kernel, as well as specific paravirtualization drivers that allow the OS and hypervisor to work together more efficiently with the overhead of the hypervisor translations. This software-assisted offers optimized virtualization support on servers with or without processors that provide virtualization extensions. Paravirtualization support has been offered as part of many of the general Linux distributions since 2008.

Although the details of this approach differ among the various offerings, a general description is as follows (see Figure 14.3). Without paravirtualization, the guest OS can run without modification if the hypervisor emulates the hardware. In this case, calls from the guest OS drivers to the hardware are intercepted by the hypervisor, which does any necessary translation for native hardware and redirects the call to real driver. With paravirtualization, the source code of an OS is modified to run as a guest OS in a specific virtual machine environment. Calls to the hardware are replaced to calls to the hypervisor, which is able to accept these calls and redirect them without

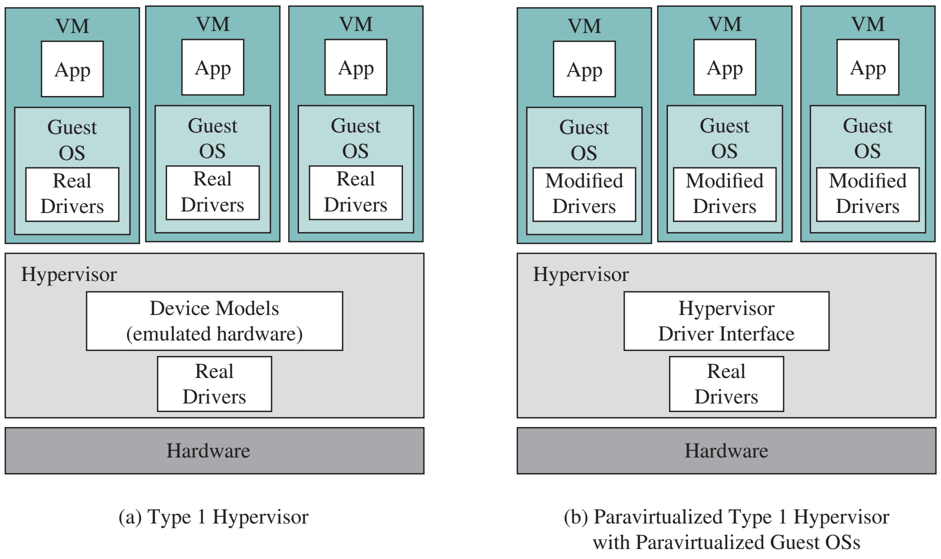

Figure 14.3 Paravirtualization

modification to the real drivers. This arrangement is faster with less overhead than a non-paravirtualized configuration.

Hardware-Assisted Virtualization

Similarly, processor manufacturers AMD and Intel added functionality to their processors to enhance performance with hypervisors. AMD-V and Intel's VT-x designate the hardware-assisted virtualization extensions that the hypervisors can take advantage of during processing. Intel processors offer an extra instruction set called Virtual Machine Extensions (VMX). By having some of these instructions as part of the processor, the hypervisors no longer need to maintain these functions as part of their codebase, the code itself can be smaller and more efficient, and the operations they support are much faster as they occur entirely on the processor. This hardware-assisted support does not require a modified guest OS in contrast with paravirtualization.

Virtual Appliance

A virtual appliance is standalone software that can be distributed as a virtual machine image. Thus, it consists of a packaged set of applications and guest OS. It is independent of hypervisor or processor architecture, and can run on either a type 1 or type 2 hypervisor.

Deploying a pre-installed and pre-configured application appliance is far easier than preparing a system, installing the app, and configuring and setting it up. Virtual appliances are becoming a de-facto means of software distribution and have spawned a new type of business—the virtual appliance vendor.

In addition to many useful application-oriented virtual appliances, a relatively recent and important development is the security virtual appliance (SVA). The SVA is a security tool that performs the function of monitoring and protecting the other VMs (User VMs), and is run outside of those VMs in a specially security-hardened VM. The SVA obtains its visibility into the state of a VM (including processor state, registers, and state of memory and I/O devices) as well as the network traffic between VMs, and between VMs and the hypervisor, through the *virtual machine introspection* API of the hypervisor. NIST SP 800-125 (*Security Recommendations for Hypervisor Deployment*, October 2014) points out the advantages of this solution. Specifically, the SVA is:

- Not vulnerable to a flaw in the Guest OS
- Independent of the virtual network configuration and does not have to be reconfigured every time the virtual network configuration changes due to migration of VMs or change in connectivity among VMs resident on the hypervisor host.

14.3 CONTAINER VIRTUALIZATION

A relatively recent approach to virtualization is known as **container virtualization**. In this approach, software, known as a **virtualization container**, runs on top of the host OS kernel and provides an isolated execution environment for applications. Unlike hypervisor-based VMs, containers do not aim to emulate physical servers. Instead, all containerized applications on a host share a common OS kernel. This eliminates the resources needed to run a separate OS for each application and can greatly reduce overhead.

Kernel Control Groups

Much of the technology for containers as used today was developed for Linux and Linux-based containers are by far the most widely used. Before turning to a discussion of containers, it is useful to introduce the concept of Linux kernel control group. In 2007 [MENA07], the standard Linux process API was extended to incorporate the containerization of user environment so as to allow grouping of multiple processes, user security permission and system resource management. Initially referred to as *process containers*, in late 2007, the nomenclature changed to *control groups* (cgroups) to avoid confusion caused by multiple meanings of the term *container* in the Linux kernel context, and the control groups functionality was merged into the Linux kernel mainline in kernel version 2.6.24, released in January 2008.

Linux process namespace is hierarchical, in which all the processes are child of common boot time process called init. This forms a single process hierarchy. The kernel control group allows multiple process hierarchies to coexist in single OS. Each hierarchy is attached to system resources at configuration time.

Cgroups provide:

- **Resource limiting**: Groups can be set to not exceed a configured memory limit.
- **Prioritization**: Some groups may get a larger share of CPU utilization or disk I/O throughput.
- **Accounting**: This measures a group's resource usage, which may be used, as an example, for billing purposes.
- **Control**: Freezing groups of processes, their checkpointing and restarting.

Container Concepts

Figure 14.4 compares container and hypervisor software stacks. For containers, only a small container engine is required as support for the containers. The container engine sets up each container as an isolated instance by requesting dedicated resources from the OS for each container. Each container app then directly uses the resources of the host OS. Although the details differ from one container product to another, the following are typical tasks performed by a container engine:

- Maintain a lightweight runtime environment and toolchain that manages containers, images and builds.
- Create a process for the container.
- Manage file system mount points.
- Request resources from kernel, such as memory, I/O devices, and IP addresses.

A typical life cycle of Linux-based containers can be understood through different phases of Linux containers:

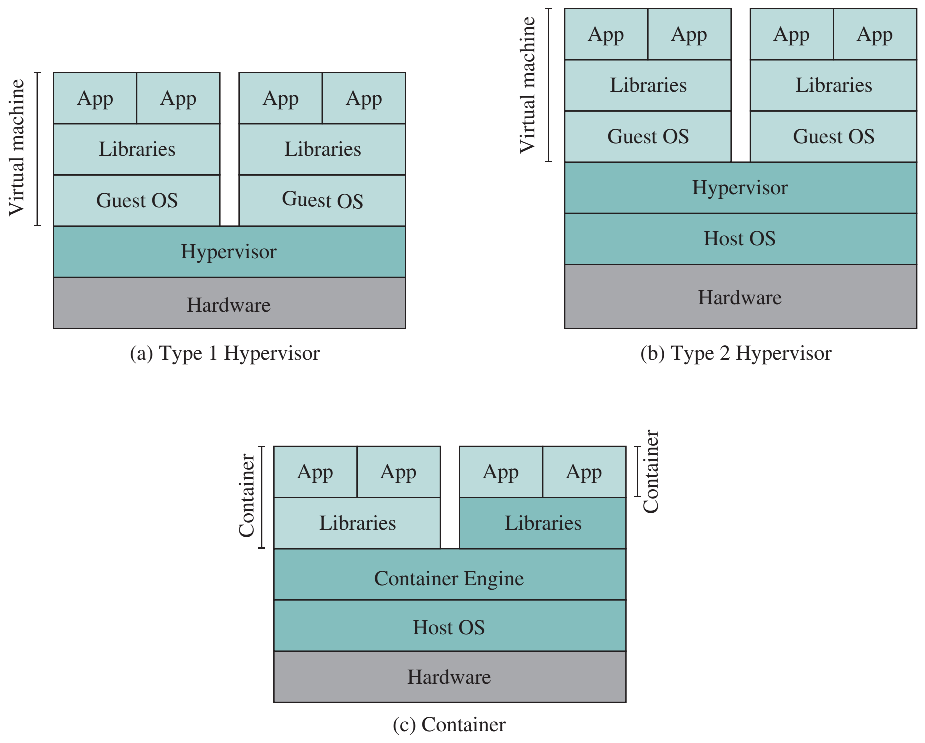

Figure 14.4 Comparison of Virtual Machines and Containers

- **Setup:** Setup phase includes the environment to create and start the Linux containers. A typical example of setup phase is Linux kernel enabled with flags or packages installed so as to allow userspace partition. Setup also includes installation of toolchain and utilities (e.g., lxc, bridge utils) to instantiate the container environment and networking configuration into host OS.
- **Configuration:** Containers are configured to run specific applications or commands. Linux container configuration includes networking parameters (e.g., IP address), root file systems, mount operations, and devices that are allowed access through the container environment. In general, containers are configured to allow execution of an application in controlled system resources (such as upper bound on application memory access).
- **Management:** Once a container is set up and configured, it has to be managed so as to allow seamless bootstrap (start up) and shutdown of the container. Typically, managed operations for a container-based environment include start, stop, freeze, and migrate. In addition, there are meta commands and toolchains that allows controlled and managed allocation of containers in a single node for end user access.

Because all the containers on one machine execute on the same kernel, thus sharing most of the base OS, a configuration with containers is much smaller and lighter weight compared to a hypervisor/guest OS virtual machine arrangement. Accordingly, an OS can have many containers running on top of it, compared to the limited number of hypervisors and guest OSs that can be supported.

Virtual containers are feasible due to resource control and process isolations as explained using techniques such as the kernel control group. This approach allows system resources being shared between multiple instances of isolated containers. Cgroups provides a mechanism to manage and monitor the system resources. The application performance is close to native system performance due to single kernel shared between all userspace container instances, and overhead is only to provide mechanism to isolate the containers via cgroups. Linux subsystems partitioned using control group primitives include filesystem, process namespace, network stack, hostname, IPC, and users.

To compare virtual machines with containers, consider I/O operation in during an application with process P in virtualized environment. In classical system virtualization environment (with no hardware support), process P would be executed inside a guest virtual machine. I/O operation is routed though guest OS stack to emulated guest I/O device. I/O call is further intercepted by hypervisor that forward it through host OS stack to the physical device. In comparison, the container is primarily based on indirection mechanism provided by container framework extensions that have been incorporated into main stream kernel. Here, a single kernel is shared between multiple containers (in comparison with individual OS kernel in system virtual machines). Figure 14.5 gives an overview of the dataflow of virtual machines and containers.

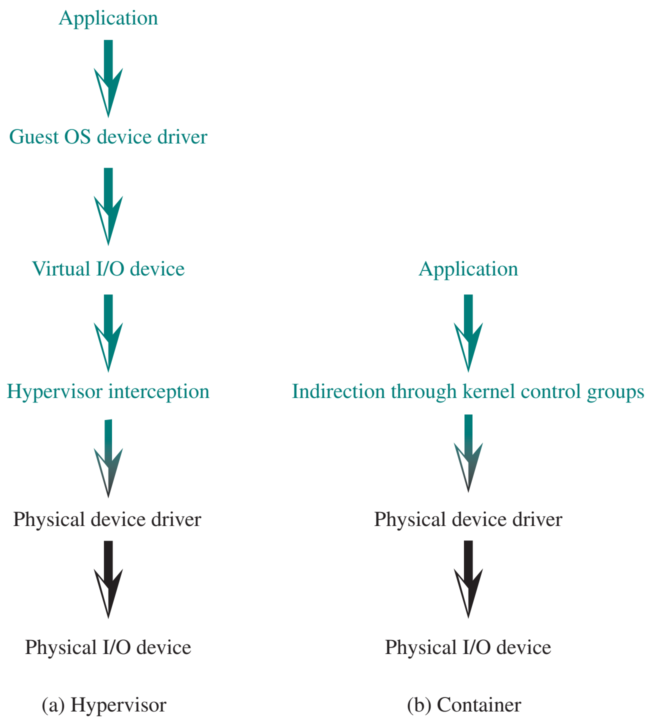

Figure 14.5 Data Flow for I/O Operation via Hypervisor and Container

The two noteworthy characteristics of containers are the following:

1. There is no need for a guest OS in the container environment. Therefore, containers are lightweight and have less overhead compared to virtual machines.
2. Container management software simplifies the procedure for container creation and management.

Because they are light weight, containers are an attractive alternative to virtual machines. An additional attractive feature of containers is that they provide application portability. Containerized applications can be quickly moved from one system to another.

These container benefits do not mean containers are always a preferred alternative to virtual machines, as the following considerations show:

- Container applications are only portable across systems that support the same OS kernel with the same virtualization support features, which typically means Linux. Thus, a containerized Windows application would only run on Windows machines.
- A virtual machine may require a unique kernel setup that is not applicable to other VMs on the host; this requirement is addressed by the use of the guest OS.
- VM virtualization functions at the border of hardware and OS. It's able to provide strong performance isolation and security guarantees with the narrowed interface between VMs and hypervisors. Containerization, which sits in between the OS and applications, incurs lower overhead, but potentially introduces greater security vulnerabilities.

One potential use case, cited in [KERN16], revolves around Kubernetes, an open source container orchestration technology built by Google but now managed by the Cloud Native Computing Foundation (CNCF). The foundation itself operates as a Linux Foundation Collaborative project. As an example, if an administrator dedicates 500 Mbps to a particular application running on Kubernetes, then the networking control plane can be involved in the scheduling of this application to find the best place to guarantee that bandwidth. Or, by working with the Kubernetes API, a network control plane can start making ingress firewall rules that are aware of the container applications.

Container File System

As part of the isolation of a container, each container must maintain its own isolated file system. The specific features vary from one container product to another, but the essential principals are the same.

As an example, we look at the container file system used in OpenVZ. This is depicted in Figure 14.6. The scheduler init is run to schedule user applications and each container has its own init process, which from the hardware nodes perspective is just another running process.

The multiple containers on a host are most likely running the same processes, but each of them doesn't have an individual copy even though the ls command shows the container's /bin directory is full of programs. Instead the containers share a template, a design feature in which all the apps that come with the OS, and many of the

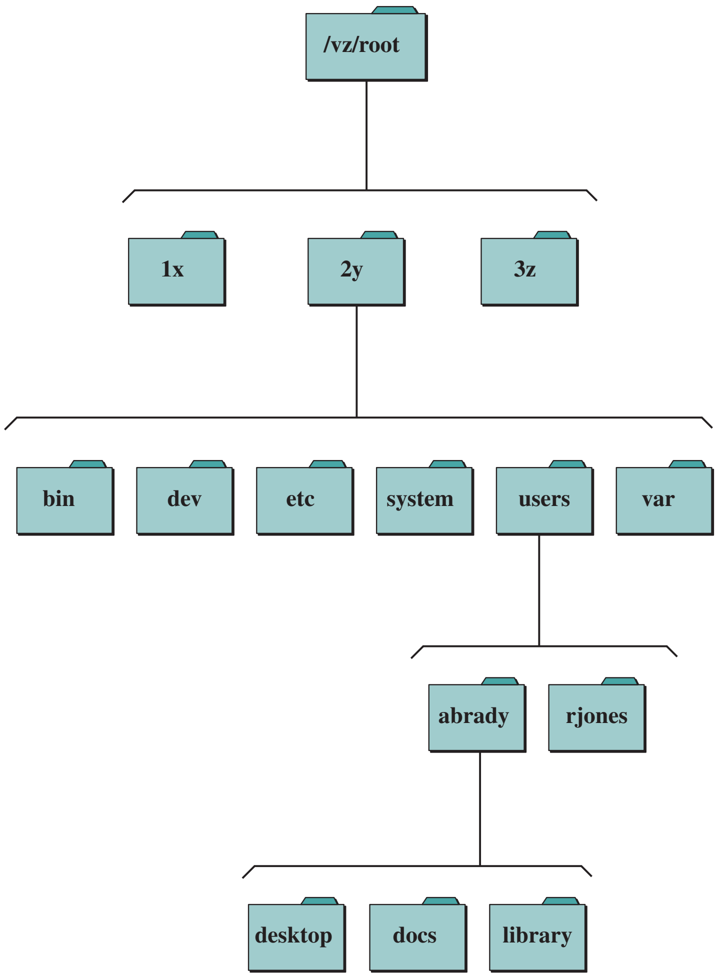

Figure 14.6 OpenVZ File scheme

most common applications, are packaged together as groups of files hosted by the platform's OS and symbolically linked into each container. This includes configuration files as well, unless the container modifies them; when that happens, the OS copies the template file (called copy on write), removes the virtual sym link and puts the modified file in the container's file system. By using this virtual file sharing scheme, a considerable space saving is achieved, with only locally created files actually existing in the container's file system.

At a disk level, a container is a file, and can easily be scaled up or down. From a virus checking point of view, the container's file system is mounted under a special mount point on the hardware node so system tools at the hardware node level can safely and securely check every file if needed.

Microservices

A concept related to containers is that of microservice. NIST SP 800-180 (*NIST Definition of Microservices, Application Containers and System Virtual Machines*, February 2016) defines a microservice as a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol 219 and a set of well-defined APIs, independent of any vendor, product, or technology.

The basic idea behind microservices is, instead of having a monolithic application stack, each specific service in an application delivery chain is broken out into individual parts. When using containers, people are making a conscious effort to break their infrastructure down into more understandable units. This opens an opportunity for networking technologies to make decisions on behalf of the user that they couldn't make before in a machine-focused world.

Two key advantages of microservices are the following:

- Microservices implement much smaller deployable units, which then enables the user to push out updates or do features and capabilities much more quickly. This coincides with continuous delivery practices, where the goal is to push out small units without having to create a monolithic system.
- Microservices also support precise scalability. Because a microservice is section of a much larger application, it can easily be replicated to create multiple instances, and spread the load for just that one small piece of the application instead of having to do so for the entire application.

Docker

Historically, containers emerged as a way of running applications in a more flexible and agile way. Linux containers enabled running lightweight applications, within Linux OS directly. Without a need for the hypervisor and virtual machines, applications can run in isolation in the same operating system. Google has been using Linux containers in its data centers since 2006. But the container approach became more popular with the arrival of Docker containers in 2013. Docker provides a simpler and more standardized way to run containers compared to earlier version of containers. The Docker container also runs in Linux. But Docker is not the only way to run containers. Linux Containers (LXC) is another way to run containers. Both LXC and Docker have roots in Linux. One of the reasons the Docker container is more popular compared to competing containers such as LXC is its ability to load a container image on a host operating system in a simple and quick manner. Docker containers are stored in the cloud as images and called upon for execution by users when needed in a simple way.

Docker consists of the following principal components:

- **Docker image:** Docker images are read-only templates from which Docker containers are instantiated.
- **Docker client:** A Docker client request that an image be used to create a new container. The client can be on the same platform as a Docker host or a Docker machine.

- **Docker host:** A platform with it own host OS that executes containerized applications.
- **Docker engine:** This is the lightweight runtime package that builds and runs the Docker containers on a host system.
- **Docker machine:** The Docker machine can run on a separate system from the Docker hosts, used to set up Docker engines. The Docker machine installs the Docker engine on a host and configures the Docker client to talk to the Docker engine. The Docker machine can also be used locally to set up a Docker image on the same host as is running Docker machine.
- **Docker registry:** A Docker registry stores Docker images. After you build a Docker image, you can push it to a public registry such as Docker hub or to a private registry running behind your firewall. You can also search for existing images and pull them from the registry to a host.
- **Docker hub:** This is the collaboration platform, a public repository of Docker container images. Users can use images stored in a hub that are contributed by others and contribute their own custom images.

14.4 PROCESSOR ISSUES

In a virtual environment, there are two main strategies for providing processor resources. The first is to emulate a chip as software and provide access to that resource. Examples of this method are QEMU and the Android Emulator in the Android SDK. They have the benefit of being easily transportable since they are not platform dependent, but they are not very efficient from a performance standpoint, as the emulation process is resource intensive. The second model doesn't actually virtualize processors but provides segments of processing time on the physical processors (pCPUs) of the virtualization host to the virtual processors of the virtual machines hosted on the physical server. This is how most of the virtualization hypervisors offer processor resources to their guests. When the operating system in a virtual machine passes instructions to the processor, the hypervisor intercepts the request. It then schedules time on the host's physical processors, sends the request for execution, and returns the results to the VM's operating system. This ensures the most efficient use of the available processor resources on the physical server. To add some complexity, when multiple VMs are contending for processor, the hypervisor acts as the traffic controller, scheduling processor time for each VM's request as well as directing the requests and data to and from the virtual machines.

Along with memory, the number of processors a server has is one of the more important metrics when sizing a server. This is especially true, and in some way more critical, in a virtual environment than a physical one. In a physical server, typically the application has exclusive use of all the compute resources configured in the system. For example, in a server with four quad-core processors, the application can utilize sixteen cores of processor. Usually, the application's requirements are far less than that. This is because the physical server has been sized for some possible future state of the application that includes growth over three to five years and also incorporates some degree of high-water performance spikes. In reality, from a processor standpoint, most servers are vastly underutilized, which is a strong driver for consolidation through virtualization as discussed earlier.

When applications are migrated to virtual environments, one of the larger topics of discussion is how many virtual processors should be allocated to their virtual machines. Since the physical server they are vacating had sixteen cores, often the request from the application team is to duplicate that in the virtual environment, regardless of what their actual usage was. In addition to ignoring the usage on the physical server, another overlooked item is the improved capabilities of the processors on the newer virtualization server. If the application was migrated at the low end of when its server's life/lease ended, it would be three to five years. Even at three years, Moore's law provides processors that would be four times faster than those on the original physical server. In order to help "right-size" the virtual machine configurations, there are tools available that will monitor resource (processor, memory, network, and storage I/O) usage on the physical servers then make recommendations for the optimum VM sizing. If that consolidation estimate utility cannot be run, there are a number of good practices in place. One basic rule during VM creation is to begin with one vCPU and monitor the application's performance. Adding additional vCPUs in a VM is simple, requiring an adjustment in the VM settings. Most modern operating systems do not even require a reboot before being able to recognize and utilize the additional vCPU. Another good practice is not to overallocate the number of vCPUs in a VM. A matching number of pCPUs need to be scheduled for the vCPUs in a VM. If you have four vCPUs in your VM, the hypervisor needs to simultaneously schedule four pCPUs on the virtualization host on behalf of the VM. On a very busy virtualization host, having too many vCPUs configured for a VM can actually negatively impact the performance of the VM's application since it is faster to schedule a single pCPU. This doesn't mean there are not applications that require multiple vCPUs. There are, and they should be configured appropriately, but most do not.

Native operating systems manage hardware by acting as the intermediary between application code requests and the hardware. As requests for data or processing are made, the operating system passes these to the correct device drivers, through the physical controllers, to the storage or I/O devices, and back again. The operating system is the central router of information and controls access to all of the physical resources of the hardware. One key function of the operating system is to help prevent malicious or accidental system calls from disrupting the applications or the operating system itself. Protection rings describe level of access or privilege inside of a computer system, and many operating systems and processor architectures take advantage of this security model. The most trusted layer is often called Ring 0 (zero) and is where the operating system kernel works and can interact directly with hardware. Rings 1 and 2 are where device drivers execute while user applications run in the least trusted area, Ring 3. In practice, though, Rings 1 and 2 are not often used, simplifying the model to trusted and untrusted execution spaces. Application code cannot directly interact with hardware since it runs in Ring 3 and needs the operating system to execute the code on its behalf in Ring 0. This separation prevents unprivileged code from causing untrusted actions such as a system shutdown or an unauthorized access of data from a disk or network connection.

Hypervisors run in Ring 0 controlling hardware access for the virtual machines they host. The operating systems in those virtual machines also believe they run in Ring 0, and in a way they do, but only on the virtual hardware that is created as part of the virtual machine. In the case of a system shutdown, the operating system on the guest would request a shutdown command in Ring 0. The hypervisor intercepts the request; otherwise, the physical server would be shutdown, causing havoc for the hypervisor and any other virtual machines being hosted. Instead, the hypervisor replies to the guest operating system that the shutdown is proceeding as requested, which allows the guest operating system to complete the necessary software shutdown processes.

14.5 MEMORY MANAGEMENT

Like the number of vCPUs, the amount of memory allocated to a virtual machine is one of the more crucial configuration choices; in fact, memory resources are usually the first bottleneck that virtual infrastructures reach as they grow. Also, like the virtualization of processors, memory usage in virtual environments is more about the management of the physical resource rather than the creation of a virtual entity. As with a physical server, a virtual machine needs to be configured with enough memory to function efficiently by providing space for the operating system and applications. Again, the virtual machine is configured with fewer resources than the physical host contains. A simple example would be a physical server with 8GB of RAM. A virtual machine provisioned with 1GB of memory would only see 1GB of memory, even though the physical server on which it is hosted has more. When the virtual machine uses memory resources, the hypervisor manages the memory requests through the use of translation tables so the guest (VM) operating system addresses the memory space at the addresses that they expect. This is a good first step, but problems remain. Similar to processor, application owners ask for memory allocations that mirror the physical infrastructures they migrated from, regardless of whether the size of the allocation is warranted or not. This leads to overprovisioned virtual machines and wasted memory resources. In the case of our 8GB server, only seven 1GB VMs could be hosted, with the remaining 1GB needed for the hypervisor itself. Aside from "right-sizing" the virtual machines based on their actual performance characteristics, there are features built into hypervisors that help optimize memory usage. One of these is **page sharing** (see Figure 14.7). Page sharing is similar to data de-duplication, a storage technique that reduces the number of storage blocks being used. When a VM is instantiated, operating system and application pages are loaded into memory. If multiple VMs are loading the same version of the OS, or running the same applications, many of these memory blocks are duplicates. The hypervisor is already managing the virtual to physical memory transfers and can determine if a page is already loaded into memory. Rather than loading a duplicate page into physical memory, the hypervisor provides a link to the shared page in the virtual machine's translation table. On hosts where the guests are running the same operating system and the same applications, between 10 and 40% of the actual physical memory can be reclaimed. At 25%, an 8-GB server could host two additional 1-GB virtual machines.

Since the hypervisor manages page sharing, the virtual machine operating systems are unaware of what is happening in the physical system. Another strategy for efficient memory use is akin to thin provisioning in storage management. This allows

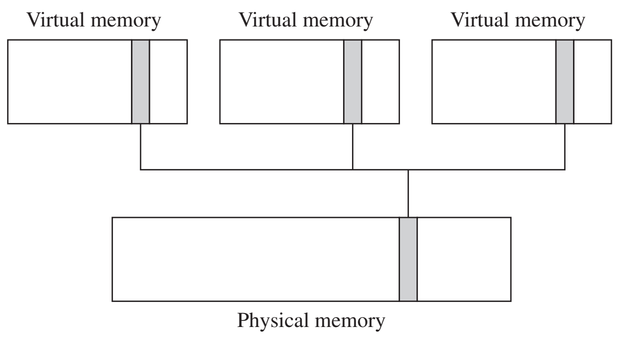

Figure 14.7 Page Sharing

an administrator to allocate more storage to a user than is actually present in the system. The reason is to provide a high water mark that often is never approached. The same can be done with virtual machine memory. We allocate 1GB of memory but that is what is seen by the VM operating system. The hypervisor can use some portion of that allocated memory for another VM by reclaiming older pages that are not being used. The reclamation process is done through **ballooning**. The hypervisor activates a balloon driver that (virtually) inflates and presses the guest operating system to flush pages to disk. Once the pages are cleared, the balloon driver deflates and the hypervisor can use the physical memory for other VMs. This process happens during times of memory contention. If our 1GB VMs used half of their memory on average, nine VMs would require only 4.5GB with the remainder as a shared pool managed by the hypervisor and some for the hypervisor overhead. Even if we host an additional three 1GB VMs, there is still a shared reserve. This capability to allocate more memory than physically exists on a host is called **memory overcommit**. It is not uncommon for virtualized environments to have between 1.2 and 1.5 times the memory allocated, and in extreme cases, many times more.

There are additional memory management techniques that provide better resource utilization. In all cases, the operating systems in the virtual machines see and have access to the amount of memory that has been allocated to them. The hypervisor manages that access to the physical memory to Ensure vs. Insure all requests are serviced in a timely manner without impacting the virtual machines. In cases where more physical memory is required than is available, the hypervisor will be forced to resort to paging to disk. In multiple host cluster environments, virtual machines can be automatically live migrated to other hosts when certain resources become scarce.

14.6 I/O MANAGEMENT

Application performance is often directly linked to the bandwidth that a server has been allocated. Whether it is storage access that has been bottlenecked, or constrained traffic to the network, either case will cause an application to be perceived as underperforming. In this way, during the virtualization of workloads, I/O virtualization is a critical item. The architecture of how I/O is managed in a virtual environment

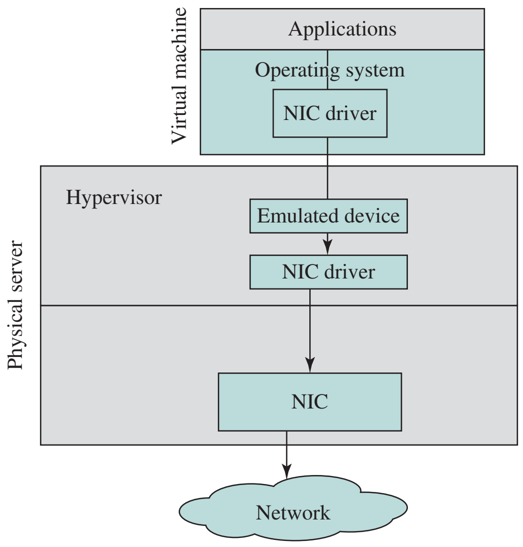

Figure 14.8 I/O in a Virtual Environment

is straightforward (see Figure 14.8). In the virtual machine, the operating system makes a call to the device driver as it would in a physical server. The device driver then connects with the device; though in the case of the virtual server, the device is an emulated device that is staged and managed by the hypervisor. These emulated devices are usually a common actual device, such as an Intel e1000 network interface card or simple generic SGVA or IDE controllers. This virtual device plugs into the hypervisor's I/O stack that communicates with the device driver that is mapped to a physical device in the host server, translating guest I/O addresses to the physical host I/O addresses. The hypervisor controls and monitors the requests from the virtual machine's device driver, through the I/O stack, out the physical device, and back again, routing the I/O calls to the correct devices on the correct virtual machines. There are some architectural differences between vendors, but the basic model is similar.

The advantages of virtualizing the workload's I/O path are many. It enables hardware independence by abstracting vendor-specific drivers to more generalized versions that run on the hypervisor. A virtual machine running on an IBM server as a host can be live migrated to an HP blade server host without worrying about hardware incompatibilities or versioning mismatches. This abstraction enables of one of virtualization's greatest availability strengths: live migration. Sharing of aggregate resources, network paths, for example, is also due to this abstraction. In more mature solutions, capabilities exist to granularly control the types of network traffic and the bandwidth afforded to individual VMs or groups of virtual machines to insure adequate performance in a shared environment to guarantee a chosen Quality of Service level. In addition to these, there are other features that enhance security and availability. The trade-off is that the hypervisor is managing all the traffic, for which it is designed, but it requires processor overhead. In the early days of virtualization this was an issue that could be a limiting factor, but faster multicore processors and sophisticated hypervisors have all but removed this concern.

A faster processor enables the hypervisor to perform its I/O management functions more quickly, and also speeds the rate at which the guest processor processing is done. Explicit hardware changes for virtualization support also improve performance. Intel offers I/O Acceleration Technology (I/OAT), a physical subsystem that moves memory copies via direct memory access (DMA) from the main processor to this specialized portion of the motherboard. Though designed for improving network performance, remote DMA also improves live migration speeds. Offloading work from the processor to intelligent devices is another path to improved performance. Intelligent network interface cards support a number of technologies in this space. TCP Offload Engine (TOE) removes the TCP/IP processing from the server processor entirely to the NIC. Other variations on this theme are Large Receive Offload (LRO), which aggregates incoming packets into bundles for more efficient processing, and its inverse Large Segment Offload (LSO), which allows the hypervisor to aggregate multiple outgoing TCP/IP packets and has the NIC hardware segment them into separate packets.

In addition to the model described earlier, some applications or users will demand a dedicated path. In this case, there are options to bypass the hypervisor's I/O stack and oversight, and directly connect from the virtual machine's device driver to physical device on the virtualization host. This provides the virtue of having a dedicated resource without any overhead delivering the greatest throughput possible. In addition to better throughput, since the hypervisor is minimally involved, there is less impact on the host server's processor. The disadvantage to a directly connected I/O device is that the virtual machine is tied to the physical server it is running on. Without the device abstraction, live migration is not easily possible, which can potentially reduce availability. Features provided by the hypervisor, like memory overcommit or I/O control, are not available, which could waste underutilized resources and mitigate the need for virtualization. Though a dedicated device model provides better performance, today it is rarely used, as datacenters opt for the flexibility that virtualized I/O provides.

14.7 VMware ESXi

ESXi is a commercially available hypervisor from VMware that provides users a Type 1, or bare-metal, hypervisor to host virtual machines on their servers. VMware developed their initial x86-based solutions in the late 1990s and were the first to deliver a commercial product to the marketplace. This first-to-market timing, coupled with continuous innovations, has kept VMware firmly on top of the heap in market share, but more importantly, in the lead from a breadth of feature and maturity of solution standpoint. The growth of the virtualization market and the changes in the VMware solutions have been outlined elsewhere, but there are certain fundamental differences in the ESXi architecture compared to the other available solutions.

The virtualization kernel (VMkernel) is the core of the hypervisor and performs all of the virtualization functions. In earlier releases of ESX (see Figure 14.9a), the hypervisor was deployed alongside a Linux installation that served as a management layer. Certain management functions like logging, name services, and often

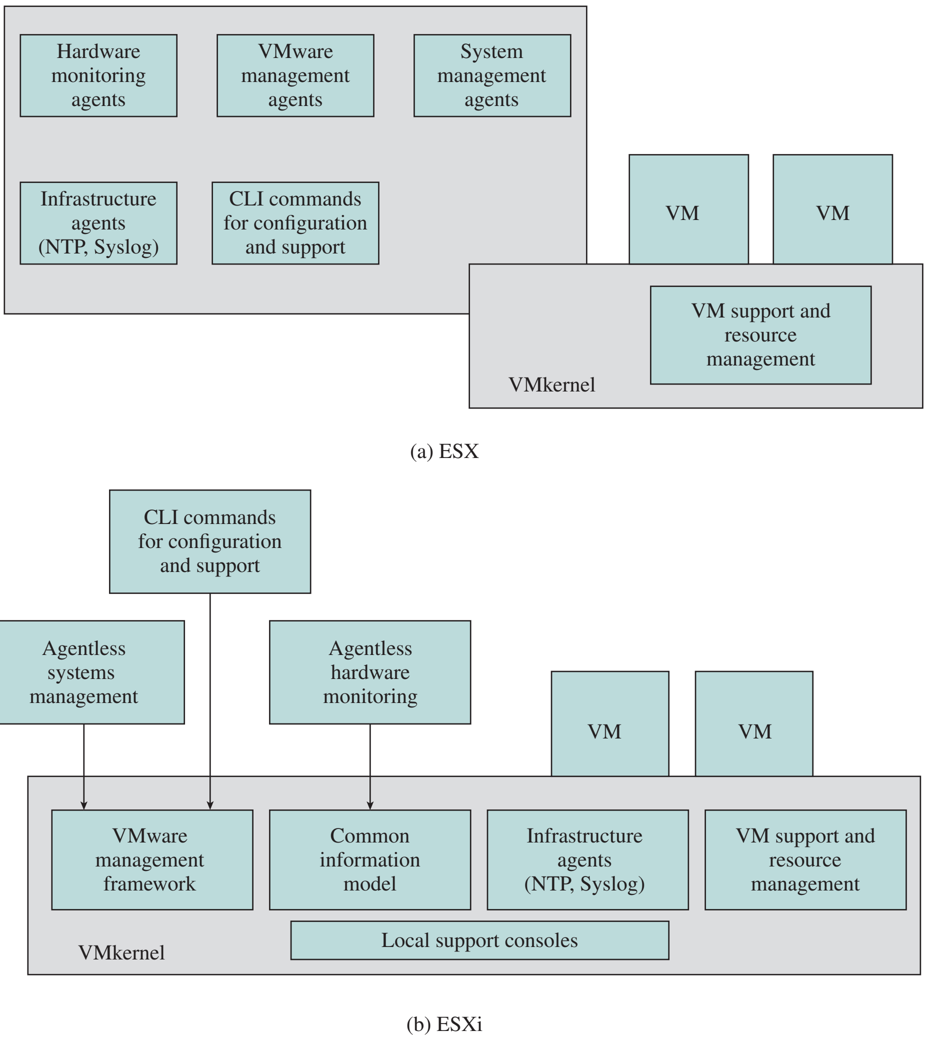
Hardware monitoring agents
VMware management agents
System management agents
Infrastructure agents (NTP, Syslog)
CLI commands for configuration and support
VM
VM
VM support and resource management
VMkernel
(a) ESX
CLI commands for configuration and support
Agentless systems management
Agentless hardware monitoring
VM
VM
VMware management framework
Common information model
Infrastructure agents (NTP, Syslog)
VM support and resource management
Local support consoles
VMkernel
(b) ESXi

Figure 14.9 ESX and ESXi

third-party agents for backup or hardware monitoring were installed on this service console. It also made a great place for administrators to run other scripts and programs. The service console had two issues. The first was that it was considerably larger than the hypervisor; a typical install required about 32MB for the hypervisor and about 900MB for the service console. The second was that the Linux-based service console was a well-understood interface and system, and was vulnerable to attack by malware or people. VMware then re-architected ESX to be installed and managed without the service console.

This new architecture, dubbed ESXi (the "i" for integrated) has all of the management services as part of the VMkernel (see Figure 14.9b). This provides a smaller and much more secure package than before. Current versions are in the neighborhood of about 100MB. This small size allows server vendors to deliver hardware with ESXi already available on flash memory in the server. Configuration management, monitoring, and scripting are now all available through command line interface utilities. Third-party agents are also run in the VMkernel after being certified and digitally signed. This allows, for example, a server vendor who provides hardware monitoring, to include an agent in the VMkernel that can seamlessly return hardware metrics such as internal temperature or component statuses to either VMware management tools or other management tools.

Virtual machines are hosted via the infrastructure services in the VMkernel. When resources are requested by the virtual machines, the hypervisor fulfills those requests, working through the appropriate device drivers. As described earlier, the hypervisor coordinates all of the transactions between the multiple virtual machines and the hardware resources on the physical server.

Though the examples discussed so far are very basic, VMware ESXi provides advanced and sophisticated features for availability, scalability, security, manageability, and performance. Additional capabilities are introduced with each release, improving the capabilities of the platform. Some examples are as follows:

- **Storage VMotion:** Permits the relocation of the data files that compose a virtual machine, while that virtual machine is in use.
- **Fault Tolerance:** Creates a lockstep copy of a virtual machine on a different host. If the original host suffers a failure, the virtual machine's connections get shifted to the copy, without interrupting users or the application they are using. This differs from High Availability, which would require a virtual machine restart on another server.
- **Site Recovery Manager:** Uses various replication technologies to copy selected virtual machines to a secondary site in the case of a data center disaster. The secondary site can be stood up in a matter of minutes; virtual machines power-on in a selected and tiered manner automatically to insure a smooth and accurate transition.
- **Storage and Network I/O Control:** Allows an administrator to allocate network bandwidth in a virtual network in a very granular manner. These policies are activated when there is contention on the network and can guarantee that specific virtual machines, groups of virtual machines that comprise a particular application, or classes of data or storage traffic have the required priority and bandwidth to operate as desired.
- **Distributed Resource Scheduler (DRS):** Intelligently places virtual machines on hosts for startup and can automatically balance the workloads via VMotion based on business policies and resource usage. An aspect of this, Distributed Power Management (DPM), can power-off (and on) physical hosts as they are needed. Storage DRS can actively migrate virtual machine files based on storage capacity and I/O latency, again based on the business rules and resource utilization.

These are just a few of the features that extend VMware's ESXi solution past being merely a hypervisor that can support virtual machines, into a platform for the new data center and the foundation for cloud computing.

14.8 Microsoft Hyper-V and Xen Variants

In the early 2000s, an effort based in Cambridge University led to the development of the Xen, an open-source hypervisor. Over time, and as the need for virtualization increased, many hypervisor variants have come out of the main Xen branch. Today, in addition to the open-source hypervisor, there are a number of Xen-based commercial hypervisor offerings from Citrix, Oracle, and others. Architected differently than the VMware model, Xen requires a dedicated operating system or domain to work with the hypervisor, similar to the VMware service console (see Figure 14.10). This initial domain is known as domain zero (Dom0), runs the Xen tool stack, and as the privileged area, has direct access to the hardware. Many versions of Linux contain a Xen hypervisor that is capable of creating a virtual environment. Some of these are CentOS, Debian, Fedora, Ubuntu, OracleVM, Red Hat (RHEL), SUSE, and XenServer. Companies that use Xen-based virtualization solutions do so due to the lower (or no) cost of the software, or due to their own in-house Linux expertise.

Guests on Xen are unprivileged domains, or sometimes user domains, referred to as DomU. Dom0 provides access to network and storage resources to the guests via BackEnd drivers that communicate with the FrontEnd drivers in DomU. Unless there are pass-through devices configured (usually USB), all of the network and storage I/O is handled through Dom0. Since Dom0 is itself an instance of Linux, if something unexpected happens to it, all of the virtual machines it supports will be affected. Standard operating system maintenance like patching also can potentially affect the overall availability.

Like most open-source offerings, Xen does not contain many of the advanced capabilities offered by VMware ESXi, though with each release, additional features appear and existing features are enhanced.

Microsoft has had a number of virtualization technologies, including Virtual Server, a Type 2 hypervisor offering that was acquired in 2005 and is still available today at no cost. Microsoft Hyper-V, a Type 1 hypervisor, was first released in 2008 as part of the Windows Server 2008 Operating System release. Similar to the Xen architecture, Hyper-V has a parent partition that serves as an administrative adjunct to the Type 1 hypervisor (see Figure 14.11). Guest virtual machines are designated as child partitions. The parent partition runs the Windows Server operating system in addition to its functions, such as managing the hypervisor, the guest partitions, and

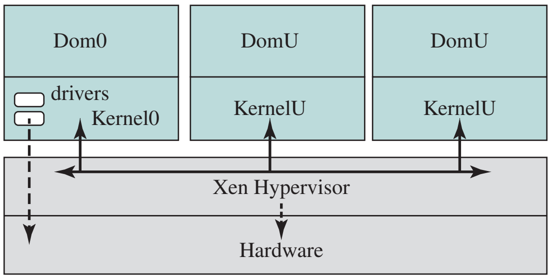

Figure 14.10 Xen

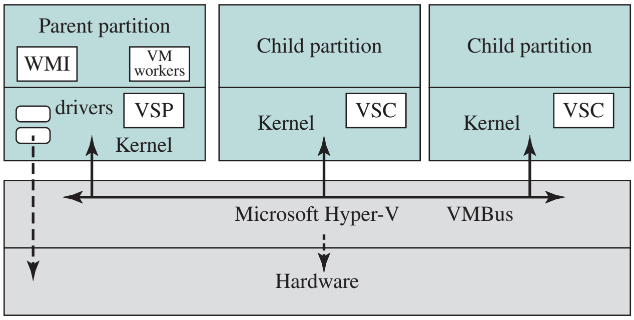

Figure 14.11 Hyper-V

the devices drivers. Similar to the FrontEnd and BackEnd drivers in Xen, the parent partition in Hyper-V uses a Virtualization Service Provider (VSP) to provide device services to the child partitions. The child partitions communicate with the VSPs using a Virtualization Service Client (or Consumer) (VSC) for their I/O needs.

Microsoft Hyper-V has similar availability challenges to Xen due to the operating system needs in the parent partition, the resource contention an extra copy of Windows requires on the server, and the single I/O conduit. From a feature standpoint, Hyper-V is very robust, though not as widely used as ESXi since it is still relatively new to the marketplace. As time passes and new functionality appears, adoption will probably increase.

14.9 Java VM

Though the Java Virtual Machine (JVM) has the term *virtual machine* as part of its name, its implementation and uses are different from the models we have covered. Hypervisors support one or more virtual machines on a host. These virtual machines are self-contained workloads, each supporting an operating system and applications, and from their perspective, have access to a set of hardware devices that provide compute, storage, and I/O resources. The goal of a Java Virtual Machine is to provide a runtime space for a set of Java code to run on any operating system staged on any hardware platform, without needing to make code changes to accommodate the different operating systems or hardware. Both models are aimed at being platform independent through the use of some degree of abstraction.

The JVM is described as being an abstract computing machine, consisting of an **instruction set**, a pc (program counter) **register**, a **stack** to hold variables and results, a **heap** for runtime data and garbage collection, and a **method** area for code and constants. The JVM can support multiple threads and each thread has its own register and stack areas, though the heap and method areas are shared among all of the threads. When the JVM is instantiated, the runtime environment is started, the memory structures are allocated and populated with the selected method (code) and variables, and the program begins. The code run in the JVM is interpreted in real time from the Java language into the appropriate binary code. If that code is valid, and adheres to the expected standards, it will begin processing. If it is invalid, and the process fails, an error condition is raised and returned to the JVM and the user.

Java and JVMs are used in a very wide variety of areas including Web applications, mobile devices, and smart devices from television set-top boxes to gaming devices to Blu-ray players and other items that use smart cards. Java's promise of "Write Once, Run Anywhere" provides an agile and simple deployment model, allowing applications to be developed independent of the execution platform.

14.10 LINUX VSERVER VIRTUAL MACHINE ARCHITECTURE

Linux VServer is an open-source, fast, virtualized container approach to implementing virtual machines on a Linux server [SOLT07, LIGN05]. Only a single copy of the Linux kernel is involved. VServer consists of a relatively modest modification to the kernel plus a small set of OS userland[1] tools. The VServer Linux kernel supports a number of separate *virtual servers*. The kernel manages all system resources and tasks, including process scheduling, memory, disk space, and processor time.

Architecture

Each virtual server is isolated from the others using Linux kernel capabilities. This provides security and makes it easy to set up multiple virtual machines on a single platform. The isolation involves four elements: chroot, chcontext, chbind, and capabilities.

The **chroot** command is a UNIX or Linux command to make the root directory (/) become something other than its default for the lifetime of the current process. It can only be run by privileged users and is used to give a process (commonly a network server such as FTP or HTTP) access to a restricted portion of the file system. This command provides **file system isolation**. All commands executed by the virtual server can only affect files that start with the defined root for that server.

The **chcontext** Linux utility allocates a new security context and executes commands in that context. The usual or *hosted* security context is the context 0. This context has the same privileges as the root user (UID 0): This context can see and kill other tasks in the other contexts. Context number 1 is used to view other contexts but cannot affect them. All other contexts provide complete isolation: Processes from one context can neither see nor interact with processes from another context. This provides the ability to run similar contexts on the same computer without any interaction possible at the application level. Thus, each virtual server has its own execution context that provides **process isolation**.

The **chbind** utility executes a command and locks the resulting process and its children into using a specific IP address. Once called, all packets sent out by this virtual server through the system's network interface are assigned the sending IP address derived from the argument given to chbind. This system call provides **network**

[1] The term *userland* refers to all application software that runs in user space rather than kernel space. *OS userland* usually refers to the various programs and libraries the operating system uses to interact with the kernel: software that performs input/output, manipulates file system objects, etc.

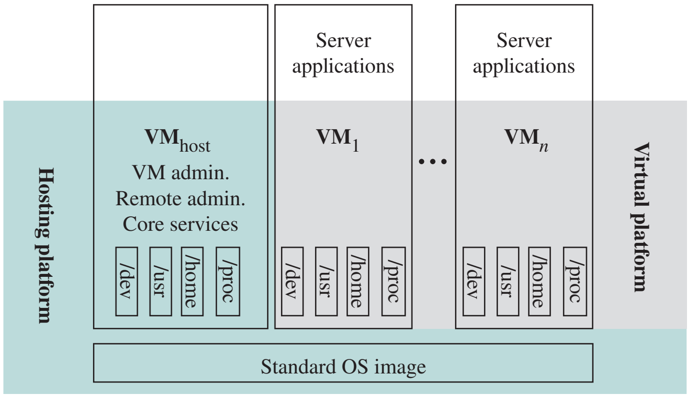

Figure 14.12 **Linux VServer Architecture**

isolation: Each virtual server uses a separate and distinct IP address. Incoming traffic intended for one virtual server cannot be accessed by other virtual servers.

Finally, each virtual server is assigned a set of **capabilities**. The concept of capabilities, as used in Linux, refers to a partitioning of the privileges available to a root user, such as the ability to read files or to trace processes owned by another user. Thus, each virtual server can be assigned a limited subset of the root user's privileges. This provides **root isolation**. VServer can also set resource limits, such as limits to the amount of virtual memory a process may use.

Figure 14.12 shows the general architecture of Linux VServer. VServer provides a shared, virtualized OS image, consisting of a root file system, and a shared set of system libraries and kernel services. Each VM can be booted, shut down, and rebooted independently. Figure 14.12 shows three groupings of software running on the computer system. The **hosting platform** includes the shared OS image and a privileged host VM, whose function is to monitor and manage the other VMs. The **virtual platform** creates virtual machines and is the view of the system seen by the **applications** running on the individual VMs.

Process Scheduling

The Linux VServer virtual machine facility provides a way of controlling VM use of processor time. VServer overlays a token bucket filter (TBF) on top of the standard Linux schedule. The purpose of the TBF is to determine how much of the processor execution time (single processor, multiprocessor, or multicore) is allocated to each VM. If only the underlying Linux scheduler is used to globally schedule processes across all VMs, then resource hunger processes in one VM crowd out processes in other VMs.

Figure 14.13 illustrates the TBF concept. For each VM, a bucket is defined with a capacity of S tokens. Tokens are added to the bucket at a rate of R tokens during every time interval of length T. When the bucket is full, additional incoming tokens are simply discarded. When a process is executing on this VM, it consumes one token for each timer clock tick. If the bucket empties, the process is put in a hold and cannot be restarted until the bucket is refilled to a minimum threshold value of M tokens. At that point, the process is rescheduled. A significant consequence of the TBF approach

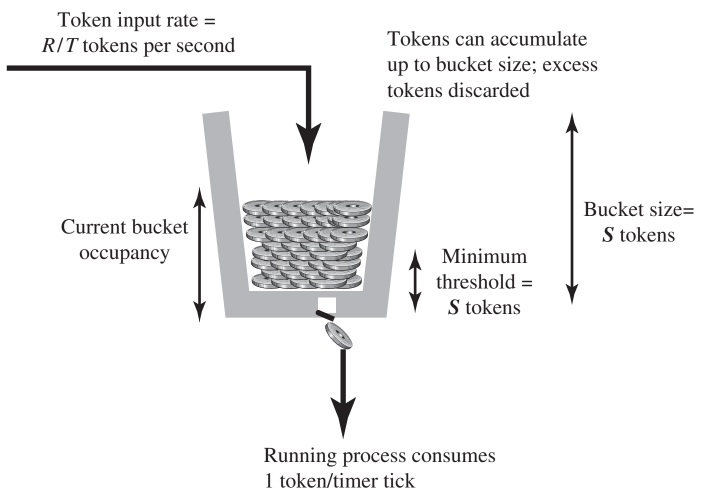
Token input rate =
R/T tokens per second
Tokens can accumulate up to bucket size; excess tokens discarded
Current bucket occupancy
Minimum threshold = S tokens
Bucket size= S tokens
Running process consumes 1 token/timer tick

Figure 14.13 Linux VServer Token Bucket Scheme

is that a VM may accumulate tokens during a period of quiescence, then later use the tokens in a burst when required.

Adjusting the values of R and T allows for regulating the percentage of capacity that a VM can claim. For a single processor, we can define capacity allocation as follows:

$$\frac{R}{T} = \text{fraction of processor allocation}$$

This equation denotes the fraction of a single processor in a system. Thus, for example, if a system is multicore with four cores and we wish to provide one VM an average of one dedicated processor, then we set $R = 1$ and $T = 4$. The overall system is limited as follows. If there are N VMs, then:

$$\sum_{i=1}^{N} \frac{R_i}{T_i} \leq 1$$

The parameters S and M are set so as to penalize a VM after a certain amount of burst time. The following parameters must be configured or allocated for a VM: Following a burst time of B, the VM suffers a hold time of H. With these parameters, it is possible to calculate the desired values of S and M as follows:

$$M = W \times H \times \frac{R}{T}$$

$$S = W \times B \times \left(1 - \frac{R}{T}\right)$$

where W is the rate at which the schedule runs (makes decisions). For example, consider a VM with a limit of 1/2 of processor time, and we wish to say that after using the processor for 30 seconds, there will be a hold time of 5 seconds. The scheduler runs at 1000 Hz. This requirement is met with the following values: $M = 1{,}000 \times 5 \times 0.5 = 2500$ tokens; $S = 1000 \times 30 \times (1 - 0.5) = 15{,}000$ tokens.

14.11 SUMMARY

Virtualization technology enables a single PC or server to simultaneously run multiple operating systems or multiple sessions of a single OS. In essence, the host operating system can support a number of virtual machines (VM), each of which has the characteristics of a particular OS and, in some versions of virtualization, the characteristics of a particular hardware platform.

A common virtual machine technology makes use of a virtual machine monitor (VMM), or hypervisor, which is at a lower level than the VM and supports VMs. There are two types of hypervisors, distinguished by whether there is another operating system between the hypervisor and the host. A Type 1 hypervisor executes directly on the machine hardware, and a Type 2 hypervisor operates on top of the host operating system.

A very different approach to implementing a VM environment is exemplified by the Java VM. The goal of a Java VM is to provide a runtime space for a set of Java code to run on any operating system staged on any hardware platform, without needing to make code changes to accommodate the different operating systems or hardware.

14.12 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

container	hypervisor	type-1 hypervisor
container virtualization	Java Virtual Machine (JVM)	type-2 hypervisor
consolidation ratio	kernel control group	virtual appliance
Docker	memory ballooning	virtualization container
guest OS	memory overcommit	virtualization
hardware virtualization	microservice	virtual machine (VM)
hardware-assisted virtualization	page sharing	virtual machine monitor (VMM)
host OS	paravirtualization	

Review Questions

14.1. Briefly describe Type 1 and Type 2 virtualization.
14.2. Briefly describe container virtualization.
14.3. Explain the concept of ballooning.
14.4. Give a brief description of Java VM.

Problems

14.1. Techniques like memory overcommit and page sharing permit virtual machines to be allocated more resources than are physically in a single virtualization host. Does this allow the aggregate of the virtual machines to perform more real work than a physical workload would be capable of on the same hardware?

14.2. Type 1 hypervisors operate directly on physical hardware without any intervening operating system. Type 2 hypervisors run as an application installed on an existing operating system. Type 1 hypervisors perform much better than Type 2 hypervisors since there is no intervening layer to negotiate between themselves and the system hardware, nor do they need to contend for resources with another controlling layer of software. Why then are Type 2 hypervisors widely used? What are some of the use cases?

14.3. When virtualization first appeared in the x86 marketplace, many server vendors were skeptical of the technology and were concerned that consolidation would impact the sales of servers. Instead, server vendors found that they were selling larger, costlier servers. Why did this happen?

14.4. Providing additional bandwidth for virtualization servers initially involved additional network interface cards (NICs) for more network connections. With the advent of increasingly greater network backbone bandwidths (10Gbit/s, 40Gbit/s, and 100Gbit/s), fewer NICs are necessary. What issues might result from these converged network connections and how might they be resolved?

14.5. Virtual machines are presented with storage in manners similar to physical machines via TCP/IP, Fibre-Channel, or iSCSI connections. There are features in virtualization that optimize memory and processor usage, and advanced features that can provide more efficient use of I/O resources. What do you think might be available to provide better use of storage resources in a virtualized environment?

CHAPTER 15

Operating System Security