Chapter 4

Virtual Machines

Virtual machines (VMs) are an important vehicle to the development of cloud computing. It is no coincidence that cloud emerged (in early 2000s) right after the invention of x86 virtualization technologies including VMware and Xen (in late 90s). In an IaaS cloud, VMs are exactly what provided to users for deploying and executing their applications. In other types of cloud such as PaaS and SaaS, VMs also play an important role behind the scene. In this chapter, we will provide an in-depth discussion on VMs.

4.1 A Taxonomy of VMs

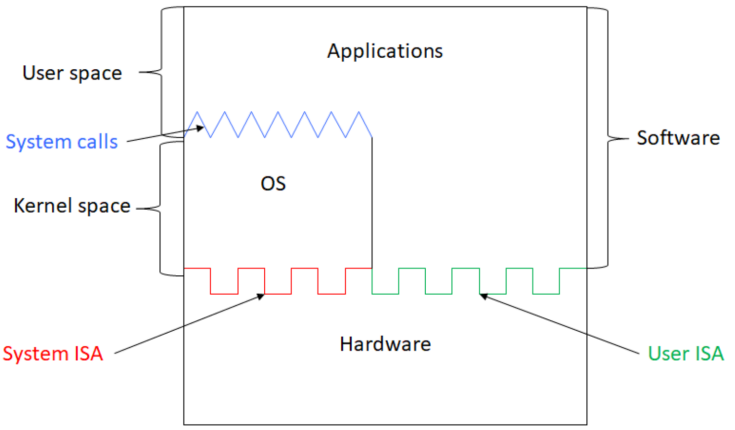

Figure 4.1: Key system interfaces

Recall virtualization is a layer of indirection, inserted into a system stack between two existing layers. Before we discuss virtual machines, let us review

the key system interfaces in a non-virtualized system, shown in Figure 4.1. Depending on which interface the virtualization layer is built upon and what interface that it presents to the upper layers, we have different types of VMs.

Instruction set architecture (ISA) is the interface that hardware, including the CPUs, memory, and I/O devices, provides to the entire software stack, including the applications, libraries/runtimes, and OS. It is what we see from the viewpoint of the whole software environment. It is the interface that the entire software stack uses.

The ISA consists of two parts, with different levels of privileges. System ISA includes privileged instructions such as I/O instructions (in and out). It is available only to the kernel-space software (OS) which runs in the kernel mode of the hardware. User ISA includes unprivileged instructions such as arithmetic instructions (e.g., add, sub), branch instructions (e.g., bz, bnz), and memory access instructions (load and store). It is available to both kernel-space software which runs in the kernel mode of the hardware and user-space software (applications) which runs in the user mode of the hardware.

Application binary interface (ABI) is the interface that the hardware and kernel-space software (OS) provide to the user-space software (applications). It includes the user ISA, which allows applications to directly perform unprivileged operations on the hardware, and the system call interface, which allows applications to request privileged operations through OS. ABI is what we see from the viewpoint of the applications. It is the interface that the application binary code uses, which is also whuy it is called the ABI.

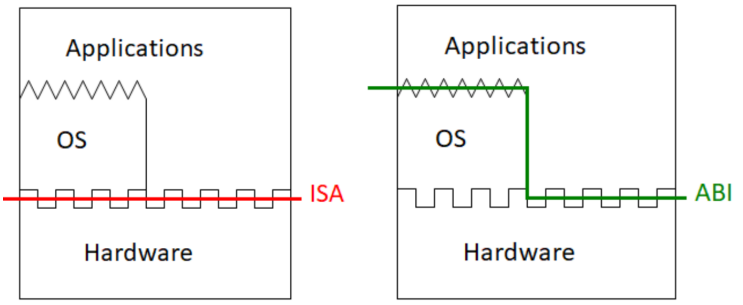

Figure 4.2: ISA and ABI

Figure 4.2 compares ISA to ABI. Based on these two key interfaces, we have two major types of VMs.

ISA virtualization is a layer of indirection inserted in between the software stack and the hardware, as illustrated in Figure 4.3. It virtualizes the underlying physical ISA and presents a virtual ISA to the software stack in the VMs. ISA VMs are the kind of VMs that we are familiar with, including desktop VMs such as VMware Workstation and VirtualBox, and server VMs such as VMware ESX and Xen.

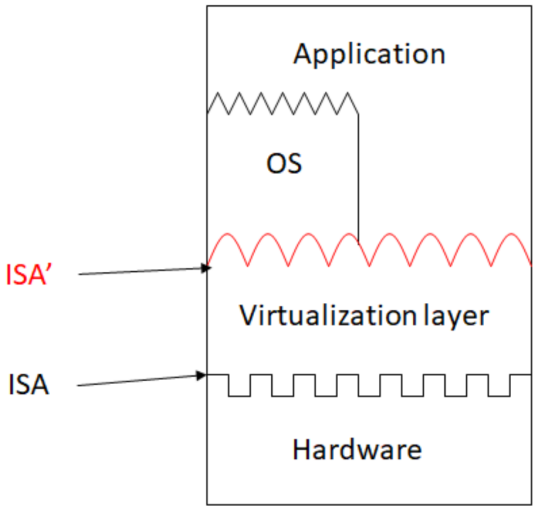

Figure 4.3: Architecture of ISA virtualization

ABI virtualization is a layer of indirection inserted in between the applications and the OS and hardware, as illustrated in Figure 4.4. It virtualizes the underlying physical ABI and presents a virtual ABI to the applications in the VMs. ABI VMs are also widely used such as Java VMs (JVM). With JVM, the bytecode virtualizes the underlying user ISA; it presents the JVM's "hardware" interface to Java applications to use for unprivileged operations. The Java core classes virtualize the underlying system calls; it presents an interface to Java applications to use for requesting privileged operations. Together, the virtual ABI provided by JVMs allow Java applications to run everywhere despite the differences in the underlying OS and hardware.

The virtual ISA (ISA') can be the same as or different from the physical ISA. Emulators such as Android emulators and game emulators provide an ISA' for a hardware architecture (ARM, game consoles) that is entirely different from the underlying hardware, and allow applications (Android apps, games) developed for ISA' to run on the hardware that provides a different ISA. However, because every instruction that the software issues using ISA' has to be emulated by the emulator using instructions from ISA, the speed of emulated applications is typically poor.

In contrast, system VMs such as VMware, Xen, and VirtualBox VMs provide a virtual ISA (ISA') that is identical to the underlying ISA. This allows the majority of the instructions, i.e., the instructions that belong to the user ISA, from the VMs to run directly on the underlying hardware without requiring any emulation. The privileged instructions from the VMs still need to be emulated, but they typically account for only a small portion of all the instructions executed by the software in the VMs. System VMs are the enabling technology

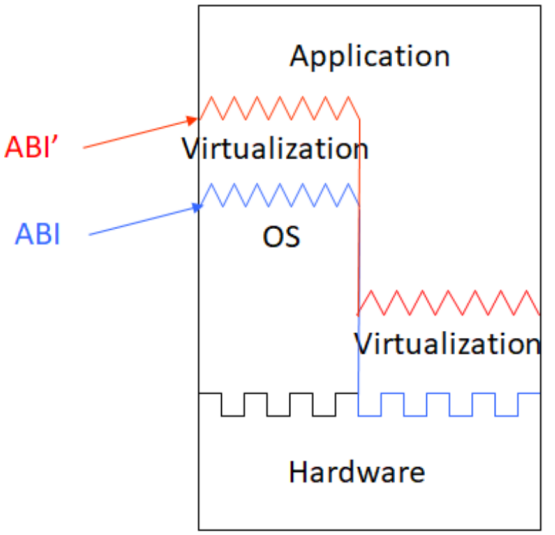

Figure 4.4: Architecture of ABI virtualization

of cloud computing, and by providing an ISA that is identical to the underlying hardware, they ensure that cloud applications can achieve good performance when running on the VMs. In the rest of this chapter, we will focus on system VMs.

Depending on whether there is a host OS on the system, there are two types of system VMs. With classic VMs, the virtualization software, which is called VM monitor (VMM) (a.k.a. hypervisor), is the only kernel-space software running in privileged mode and managing hardware resources at all times. These are called "classic" VMs because they follow the same architecture of the first VMs invented by IBM back in the 70s which we discussed in our history of cloud computing lecture. Modern examples of classic VMs include VMware ESX and Xen.

With hosted VMs, there is a host OS that runs alongside the VMM. The host OS and the VMM both run in kernel mode, and they share the management of the hardware resources. To differentiate from the OS that runs in a VM, we call the OS that runs directly on the VM host the host OS and the OS that runs inside a VM guest the guest OS. In this environment, the host OS supports the execution of the host applications, i..e, the applications that do not run in VMs, on the hardware, and the VMM supports the VMs on the hardware. Examples of hosted VMs include the desktop VMs that we are familiar with, such as VMware Workstation and Virtualbox.

Next we will look at the details of system virtualization. We will study how VMM virtualizes the three major hardware subsystems, processors, memory, and I/O devices. On a non-virtualized system, the OS manages these three

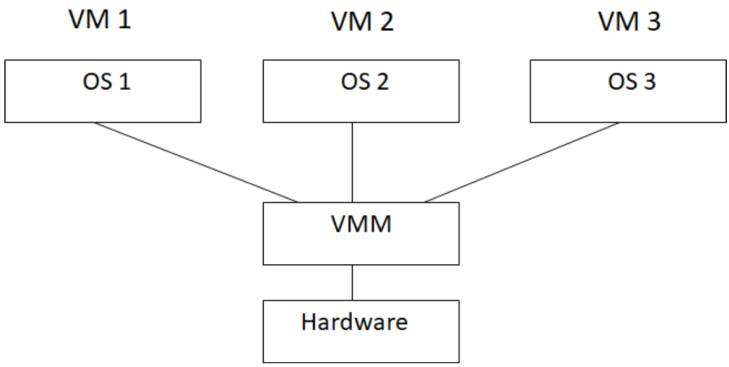

Figure 4.5: Architecture of classic VMs

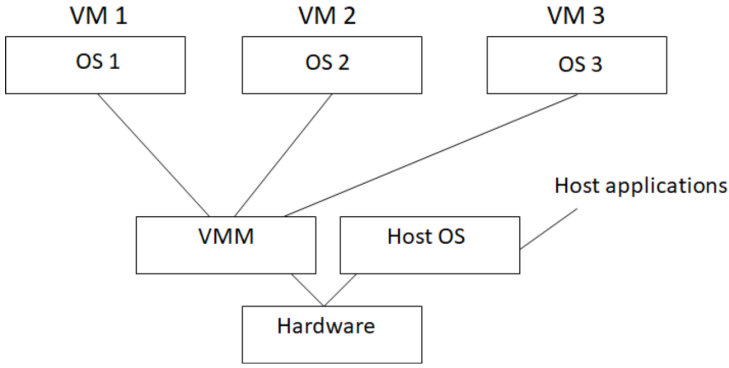

Figure 4.6: Architecture of hosted VMs

subsystems and in essence also virtualizes them for the processes sharing these resources. We will use what we know about OS to help us understand how VMM works.

4.2 Processor Virtualization

On a non-virtualized system managed by the OS, most application instructions (the part that belongs to the user ISA) execute directly on the processor, without involving the OS, which ensures good performance of the applications. Applications' accesses to shared resources such as I/O devices are handled indirectly (because they cannot use privileged instructions) by calling the OS via system calls, and the OS performs these accesses using instructions from the system ISA.

In a virtualized system managed by the VMM, most application instructions

(the part that belongs to the user ISA) still execute directly on the processor, without involving the VMM, which ensures good performance of the virtualized applications. When applications need access to shared resources, they will still make system calls to the guest OS, but the OS cannot run privileged instructions either since it also runs in user mode (and only the VMM runs in kernel mode). When the virtualization is entirely transparent to the guests, the guest OS is unaware that it is running in a VM, and it cannot and will not call the VMM for help. Instead, it will do what it is supposed to do, using privileged instructions to perform access to shared resources. When a privileged instruction is executed in user mode, it will trigger an exception and the system software will be brought in to handle the exception, which is called a trap. In a virtualized system, the system software is the VMM, and the trap gives the VMM the opportunity to intercept the guest's attempt to access shared resources and emulate it with controlled access to the resources that the guest has access to.

4.3 Memory Virtualization

In a non-virtualized system managed by the OS, an application running on the CPU loads and stores virtual addresses; the MMU maps the virtual addresses to physical addresses of the physical memory, using page tables managed by the OS. To support many concurrent applications on the limited physical memory, the OS uses the secondary storage (disks) to provide additional space: it swaps out pages that are currently not needed by the applications from the memory and temporarily stores them on disk. When the application makes reference to a swapped out page, the MMU cannot find its mapping from the page table (since it is currently not in the physical memory), which triggers a page fault. The OS handles the page fault by bringing the faulting page back to physical memory on demand, which is called demand paging. Figure 4.7 illustrates memory virtualization in a non-virtualized system.

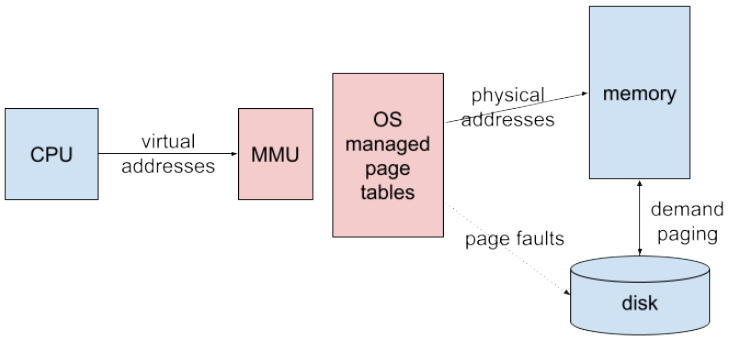

Figure 4.7: Memory virtualization in a non-virtualized system

In a virtualized system managed by the VMM, there is one more level of

memory virtualization. An application still loads and stores virtual addresses; the guest OS maps virtual addresses to the "physical" addresses of the VM using the application's page tables; and the VMM maps the "physical" addresses of the VM to the physical addresses of the physical machine using the VM's page tables. However, traditional MMU can work with only one level of page table for address translation; it cannot use the two levels of page tables needed for translating a virtual address to a physical address. To solve this problem, the VMM maintains a shadow page table on the side for each application, which maps directly from the application's virtual addresses to the physical memory's addresses. The MMU then uses only this single-level page table for address translation. Figure 4.7 illustrates memory virtualization in a virtualized system.

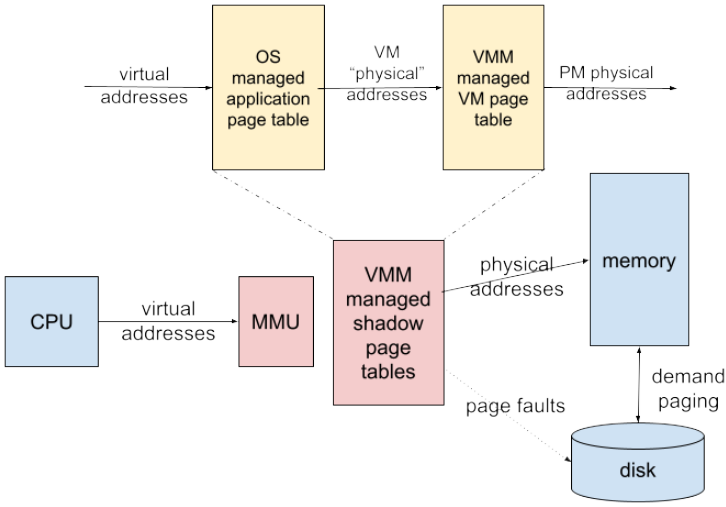

Figure 4.8: Memory virtualization in a virtualized system

In a virtualized system, there are two levels of demand paging. A guest OS still does demand paging for its applications: it swaps out pages from the VM's "physical" memory to the VM's "disk" (which is often stored as a file on the host's file system). The VMM does the second demand paging for its VMs: it swaps out pages from the physical machine's physical memory to the physical disk.

When a page fault happens in a virtualized system, the VMM is the kernel-space software to handle it. The faulting page might have been swapped out by either the guest OS or the VMM. If it was swapped out by the guest OS, the guest OS knows how to handle it and is expecting a page fault when the page was referenced. In this case, the VMM injects a virtual page fault into the VM, and the guest OS runs its page fault handler by bringing the page back from its VM's disk to its "physical" memory. If the faulting page was swapped out by the VMM, the VMM knows how to handle it, and the guest OS shouldn't be

made aware since it expects the page to be present in its “physical” memory. The VMM brings the page back from the physical disk to the physical memory, masking the page fault and its handling both from the guest OS.