Infrastructure as a Service (IaaS)

In this lecture, we use Amazon Web Services (AWS) to study IaaS and learn how to develop a cloud application (app) using IaaS resources. We choose AWS because it provides representative IaaS services that can be found in other cloud providers; it is also the first IaaS provider and today the largest cloud provider in the world.

There are three fundamental types of IaaS services that we typically need to develop a cloud app: compute (AWS EC2), which provides computing resources for executing the applications, storage (AWS EBS, S3), which provides storage resources for storing the data that applications needs to access, and messaging (AWS SQS) for the distributed applications and their distributed components to communicate across the network.

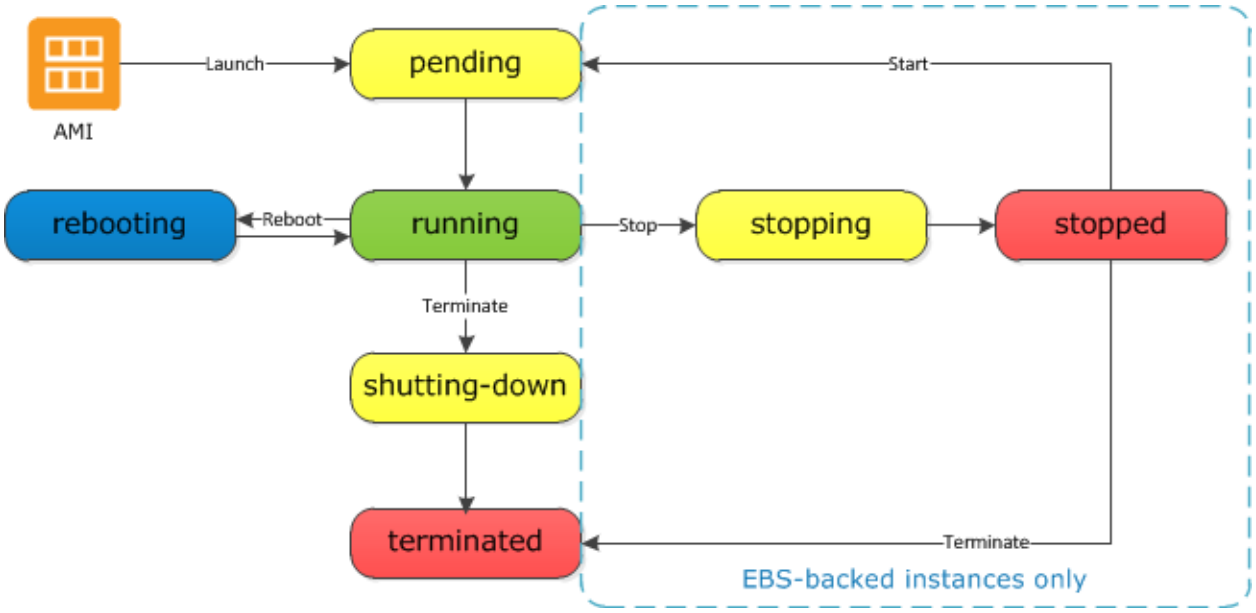
Before we discuss these IaaS services in detail, I want to introduce a success story of AWS. Animoto.com is an SaaS providers on April 18th 2008, it launched a viral ad campaign on Facebook, which led to a huge spike in the demand of its service: its number of users grew from 5,000 to 750,000 in just three days. Fortunately, Animoto.com was hosted on AWS, and it was able to handle this sudden surge of demand by quickly growing its number of EC2 instances from 40 to 4000. Even though it is a really outdated story, it still serves as a good example of the power of cloud computing. Without the cloud, a company like Animoto.com would not be able to deal with the sudden change of demand quickly and cost-effectively.

In the following, we will learn how to use the basic IaaS resources. Although the discussions here are based on AWS, we can also find similar IaaS resources from the other providers.

EC2 provides virtual environments for computing using virtual machine instances. Instances come in different types, each configured with a certain amount of CPUs, memory, storage, and network capacity. As discussed in Lecture 1, users can specify the location of their instances at some higher level of abstraction, regions and availability zones. Each region is a separate geographic area. A region has multiple availability zones. Each zone can fail independently from the other zones.

Instances are instantiated from images. The relationship between instances and images is analogous to the relationship between processes and programs. Multiple instances of different types can be started from the same image. AWS provides templates for the images, called AMI. Users can also create their own custom AMIs.

During the lifecycle of an instance, it goes through various state as shown in the state diagram.



Instances that run a distributed application need to communicate with one another. Some of the instances also need to communicate with programs outside of the cloud, e.g., servicing requests received from the clients of the application. An instance always gets a private IP and an internal hostname. These addresses are visible within the cloud, and are useful for communications among the instances. An instance also gets a public IP and an external hostname. These are visible outside of the cloud, but they are not persistent---they are automatically released when the instance is stopped or terminated.

Having a persistent external address is important to an instance that needs to communicate with the outside world. AWS provides these persistent IP addresses as a service: EIP. A user can rent EIP addresses and allocate them to the instances that need persistent external addresses.

There are three basic types of storage services in AWS. Both the instance store and EBS can provide storage for instance volumes. S3 can store the snapshots of instance volumes.

An instance can have multiple volumes, including the volume that stores the file system and volumes that store user data. Volumes are typically stored on EBS, which provides reliable network storage by replicating the volumes within the availability zone. Volumes can be attached to any instance in the same zone. Volumes can persist independently

from the instances---they can be created and deleted independently from the lifecycles of the instances.

The instance store provides temporary storage for volumes. It is local to the instances, therefore providing better performance than EBS-backed volumes. But instance store backed volumes persist only when their associated instances are running; they are automatically deleted when their instances are stopped or terminated.

A user can back up an EBS volume by taking a snapshot of the volume, which is a point-in-time copy of the data on the volume. Snapshots are stored on S3 and replicated across availability zones for high reliability. Later the user can recover EBS volumes or create new volumes from the snapshots.

Unlike EBS and instance store which provide only volume storage, S3 is a general-purpose storage service which can store all kinds of data. It provides an object store interface, which organizes objects into buckets. Objects can be of any formats and any sizes, and they are all accessed through the same Get and Put interface. S3 is also highly reliable and cost-effective. But its performance is inferior to EBS and instance store, and cannot be used to provide volume storage to running instances.

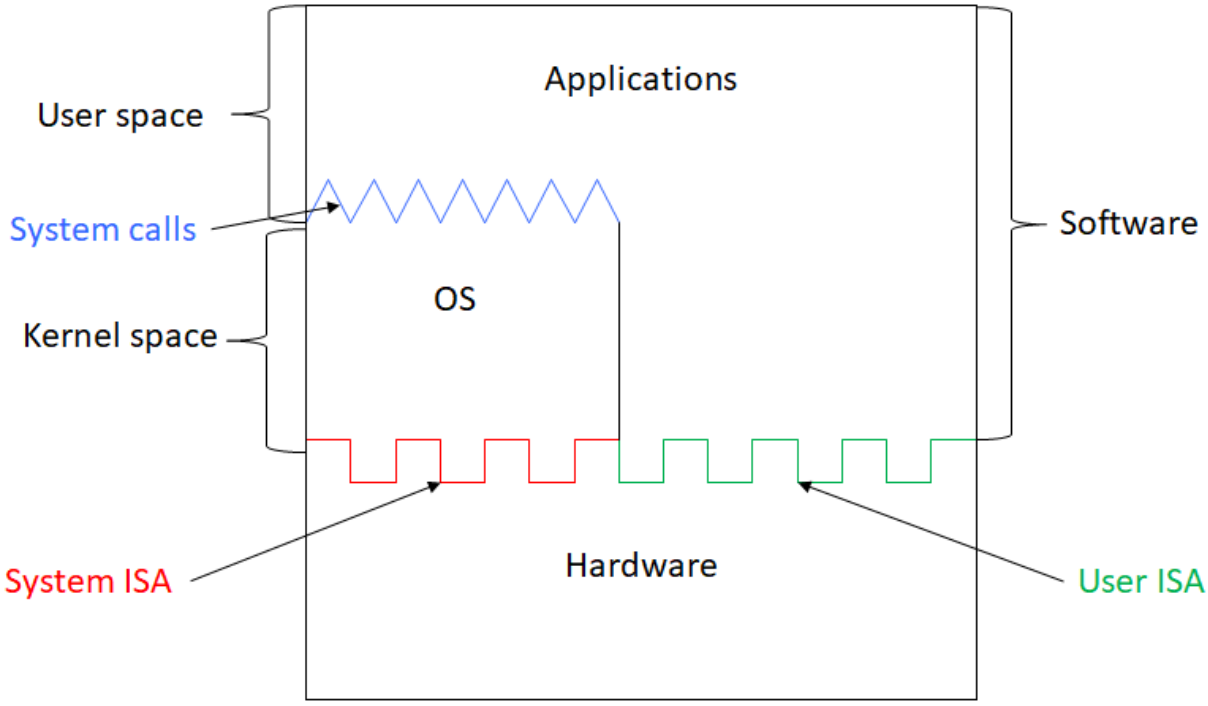
SQS provides reliable and scalable message queuing service. Queues are a fundamental data structure for messaging. Senders deposit messages in a queue, and receivers retrieve the message from the queue. SQS is highly reliable: it employs replication to ensure that every message is always delivered; but occasionally, duplicates of the same message may be delivered. SQS is also highly scalable: it delivers the same performance to message delivery regardless of the number of queued messages and the number of concurrent senders and receivers working on the same queue.

To use SQS service, the user first creates an SQS queue; then the senders can send messages to the queue and the receivers can receive messages from the queue. To ensure the delivery of a message, the message stays in the queue until the receiver explicitly deletes the message when it determines it is safe to do so. To avoid multiple receivers getting the same message, SQS provides a visibility timeout clock on the queue. When a receiver retrieves a message from the queue, the timeout period for this message starts. During this period, this message is not visible to the other receivers, and therefore cannot be retrieved. The receiver that has retrieved the message needs to delete the message after it has successfully processed it.

□

Virtual Machines

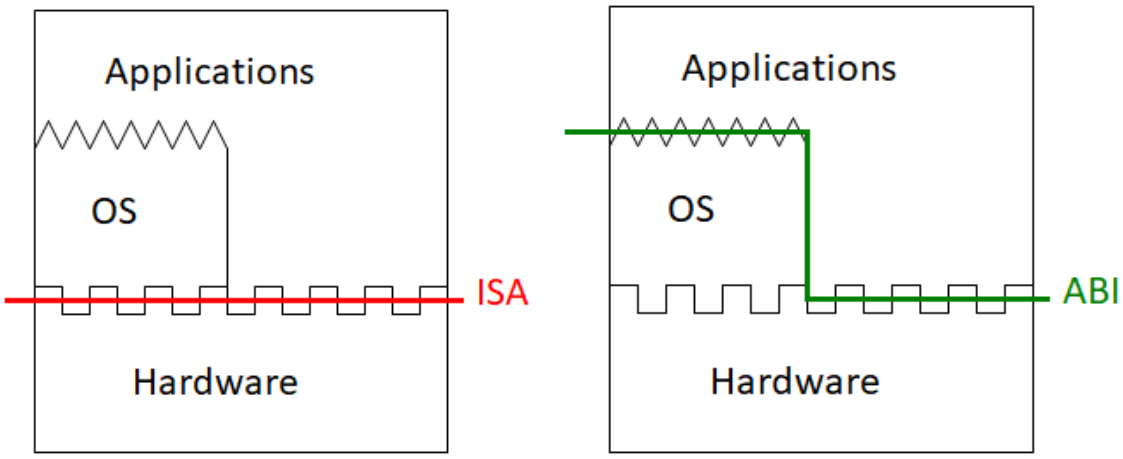
Before we discuss virtual machines, let's review the key system interfaces.



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.

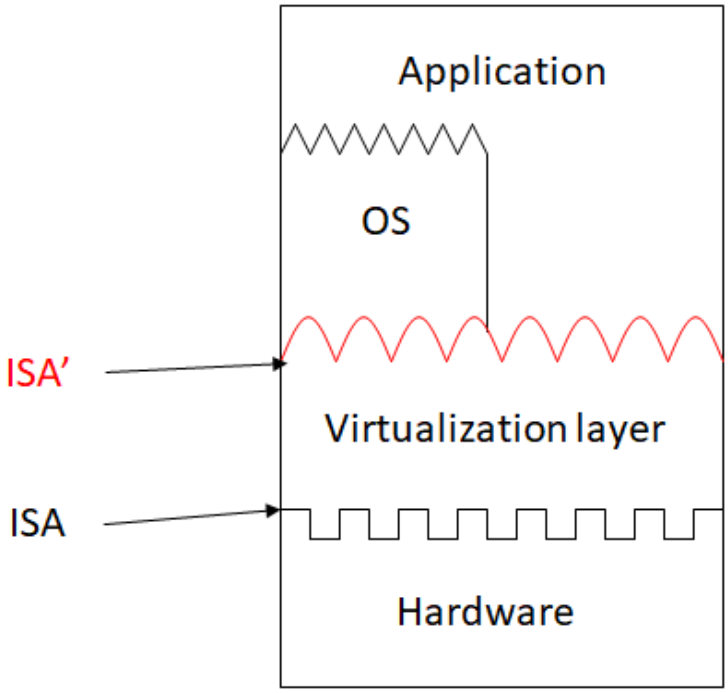
The ISA consists of two parts, with different levels of privileges. System ISA includes privileged instructions such as I/O instructions (in and out). It is available only to the kernel-space software (OS) which runs in the kernel model of the hardware. User ISA includes unprivileged instructions such 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 model 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 the interface that the applications use.

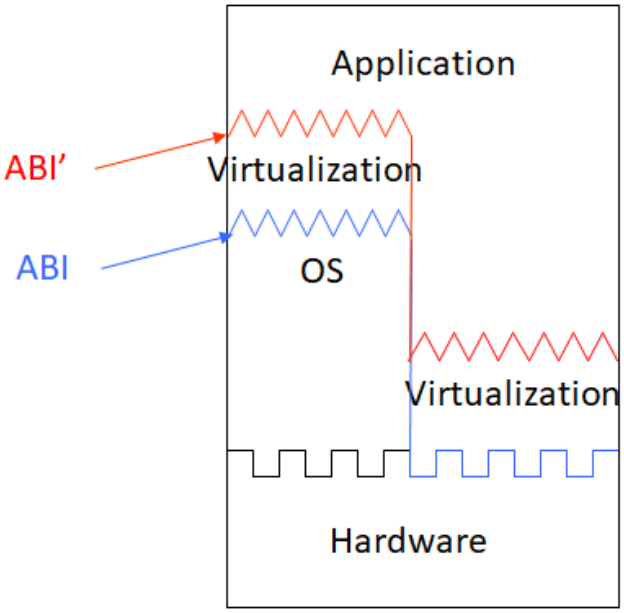


Based on ISA and ABI, we have two major types of VMs.

ISA virtualization is a layer of indirection inserted in between the software stack and the hardware. 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.



ABI virtualization is a layer of indirection inserted in between the applications and the OS and hardware. 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, and Java core classes virtualize the underlying system calls. Together, the virtual ABI provided by JVMs allow Java applications to run everywhere despite the differences in the underlying OS and hardware.

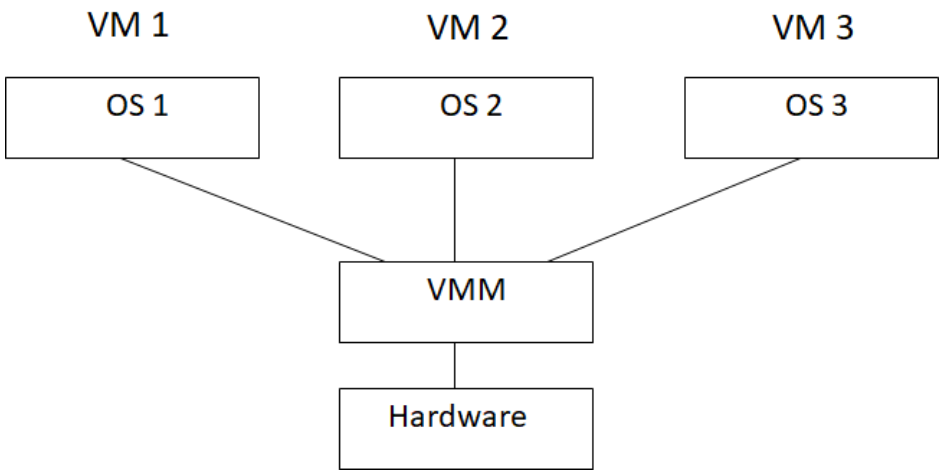


The virtual ISA (ISA') can be the same as or different from the physical ISA. Emulators such as Android emulators and game emulators provide an ISA' for a hardware architecture (ARM, game consoles) that is entirely different from the underlying hardware, and allow applications (Android apps, games) developed 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.

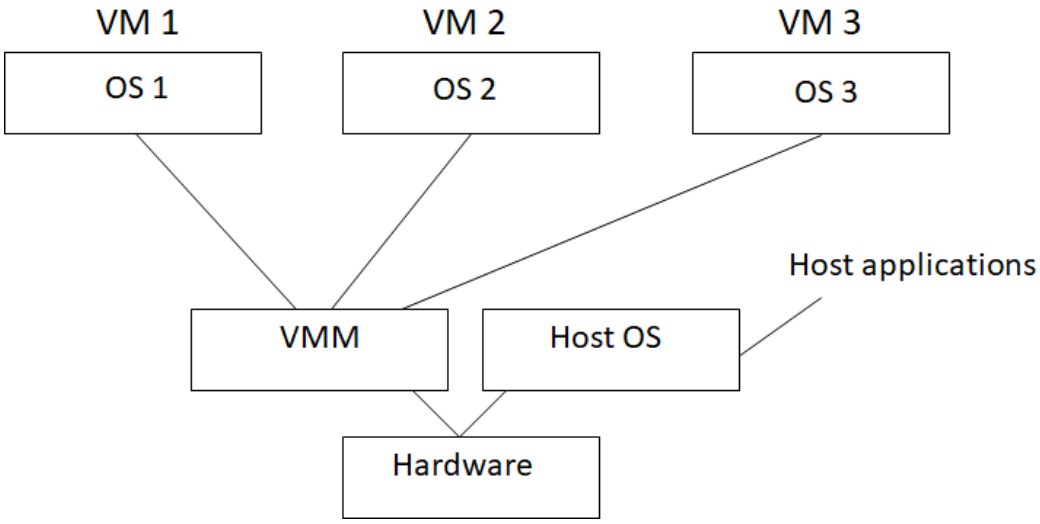
In contrast, system VMs such as VMware, Xen, and VirtualBox VMs provide a virtual ISA (ISA') that is identical to the underlying ISA. This allows the majority of the instructions, i.e., the instructions that belong to the user ISA, from the VMs to run directly on the underlying hardware without requiring 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 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 lecture, 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.



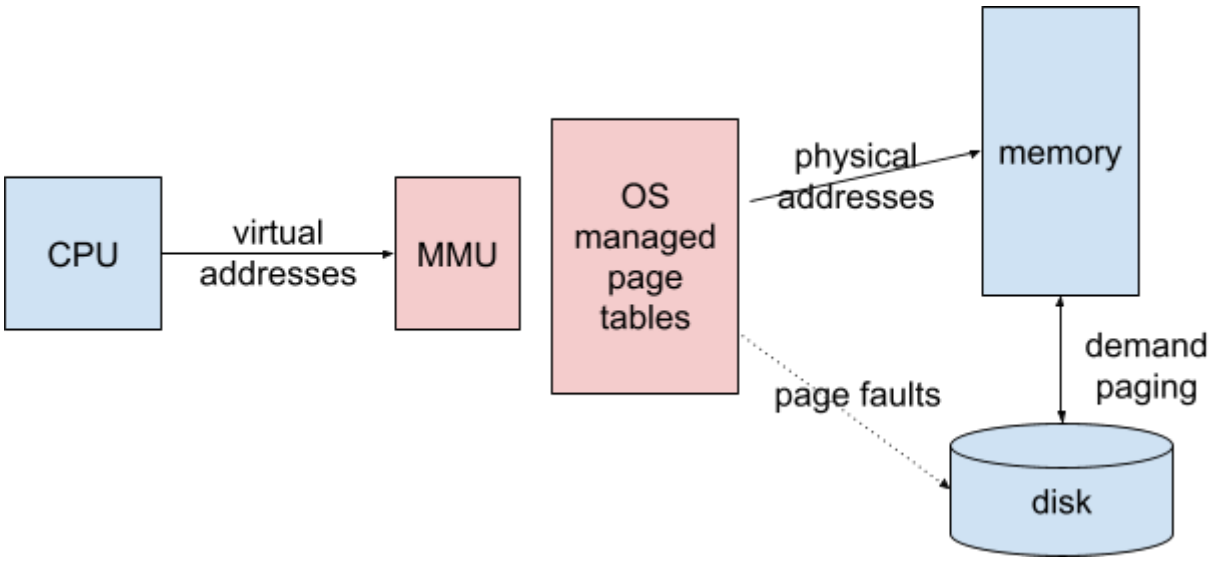
Now look at the details of system virtualization. We will study how VMM virtualizes the three major hardware subsystems, processors, memory, and I/O devices. On a non-virtualized system, the OS manages these three subsystems and in essence also virtualizes them for the processes sharing these resources. We will use what we know about OS to help us understand how VMM works.

Processor virtualization. In 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 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.

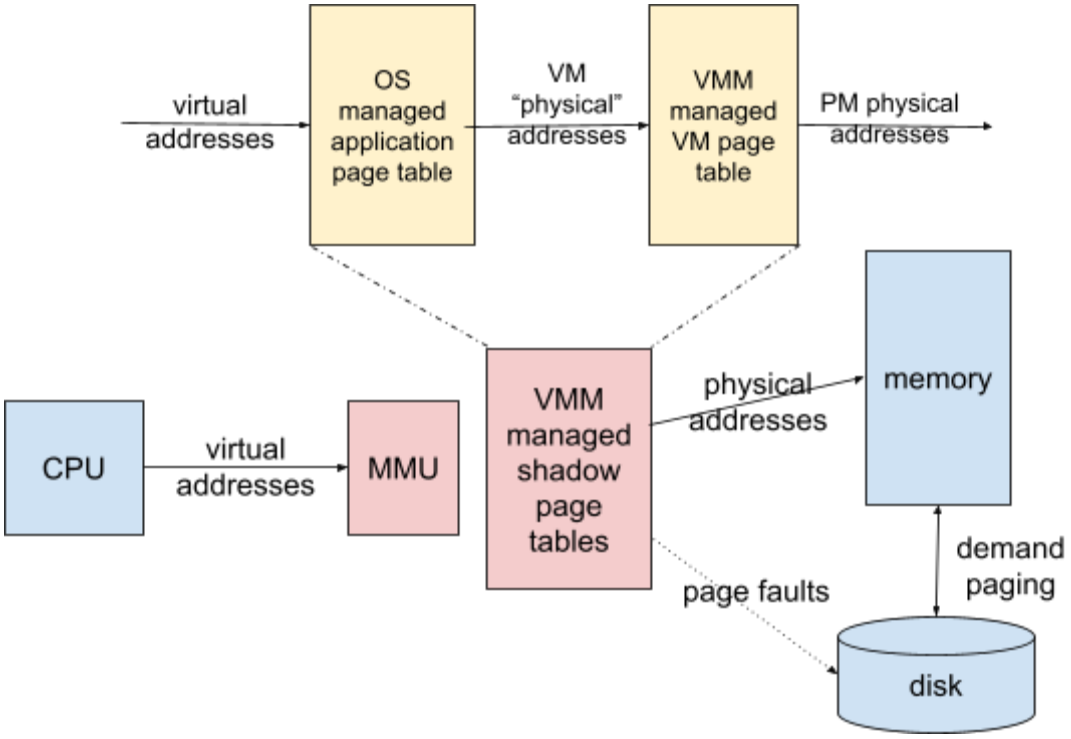
In a virtualized system managed by the VMM, most application instructions (the part that belongs to the user ISA) still execute directly on the processor, without involving the VMM, which ensures good performance of the applications. When applications need access to shared resources, they will still 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 privileged instructions are executed in user mode, they will be trapped by the hardware and handled by the system software which in this case is the VMM. This gives the VMM the opportunity to intercept the

guest's attempt to access shared resources and emulate it using privileged instructions that the VMM can use.

Memory virtualization. In a non-virtualized system managed by the OS, an application running on the CP 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.



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.

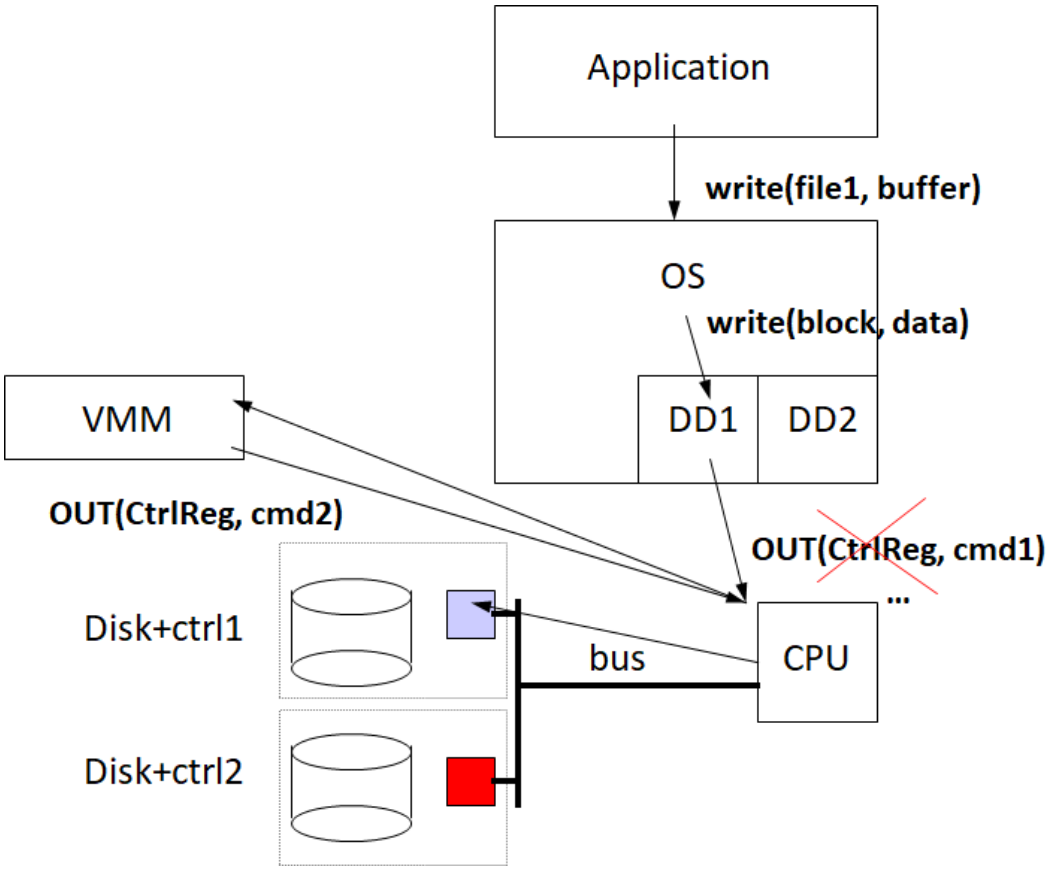


In a virtualized system, there are two levels of demand paging. A guest OS still does demand paging for its applications: it swaps out pages from the VM's "physical" memory to the VM's "disk" (which is often stored as a file on the host's file system). The VMM does the second demand paging for its VMs: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

I/O virtualization. When it comes to the virtualization of I/O devices, there is a major difference between classic VMs and hosted VMs. With classic VMs, the VMM needs to implement all the device drivers to support the diverse I/O devices that the physical machine may have. With hosted VMs, the host OS already has all the device drivers, to

allow its applications to use the devices, and the VMM can exploit these drivers instead of implementing them on its own. Moreover, in both cases, the virtual I/O devices provided by the VMM to the VMs can be very much different from the underlying physical I/O devices, since the applications typically don't really care about what specific devices they are using.

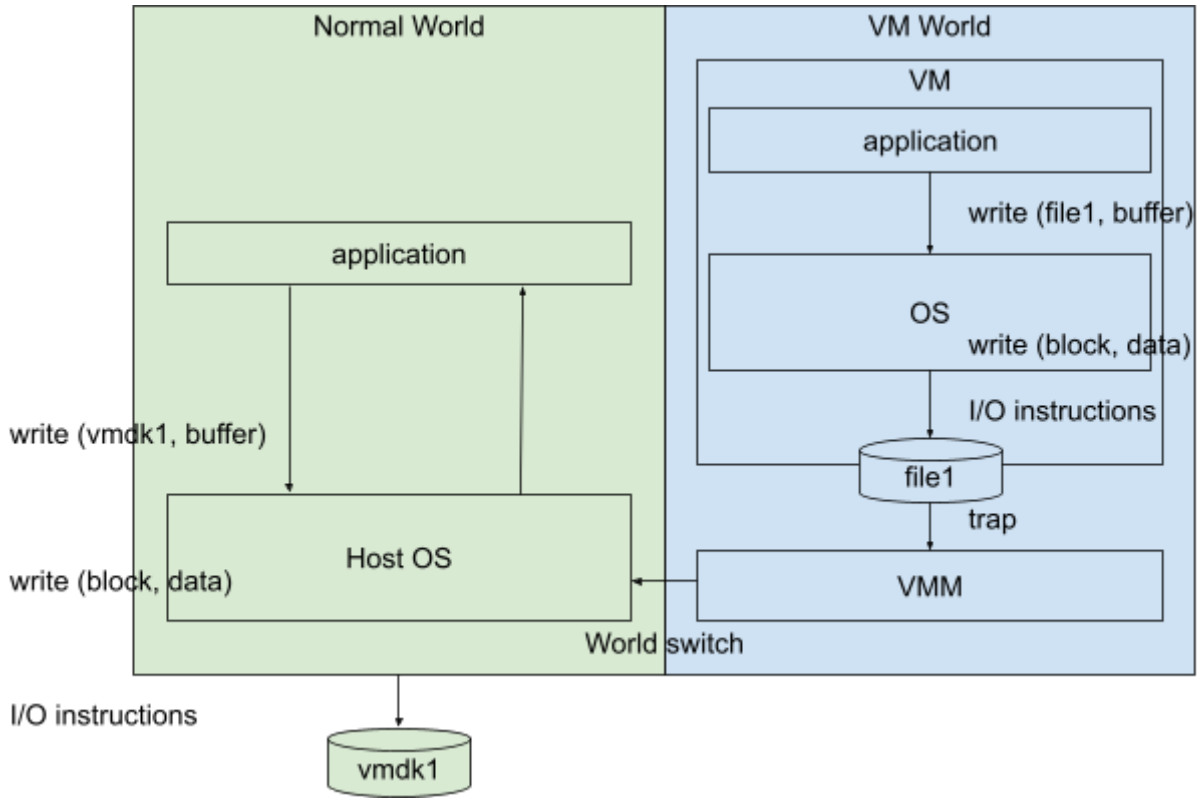


Let's first walk through the steps involved in I/O. The application's I/O request, such as writing to a file (stored on VM's disk), it is handled by the I/O stack in the VM, including API call to the libraries/runtimes, the system call to the guest OS, the file system request to the file system, the block request to the block device layer, and finally I/O instructions to the device. As we discussed in processor virtualization, when the guest OS uses these I/O instructions, which are privileged, they will be trapped into the VMM, which allows them to be intercepted and emulated.

On a classic VM system, the VMM will use its I/O stack to emulate the VM's I/O request. Assuming the VM's disk is stored as a file on the file system of the physical disk, the application's write request should be emulated by a write request to the VM disk file (and other relevant I/Os). The VMM will use its own file system implementation and

block device driver to perform this request, which eventually will generate a series of I/O instructions to the physical disk (more precisely, the disk controller). When the write completes, the disk controller generates an interrupt, which is handled by the VMM's interrupt handler. This result traverses back across all the VMM and guest layers we mentioned above, and eventually returns to the application which is waiting for its write to complete. Note that in order for guest OS to be notified of the write completion, a virtual interrupt needs to be injected into the VM, much like how a virtual page fault is injected as discussed earlier in memory virtualization.

On a hosted VM system, the VMM does not have the necessary I/O stack for handling the VM's I/O request. Instead, it passes it on to the host OS and leverages the host OS's I/O stack to emulate the I/O. This involves a switch from the VM world, managed by the VMM, to the normal world, managed by the host OS. Again, to simplify the implementation, the request is further passed on to a user-space application in the normal world, which implements the emulation of the request using the APIs and system calls available to the user-space applications. The host OS handles the system calls and performs the I/Os using its own I/O stack, including its device driver provided for the physical disk.



□