[Module 10](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d4780020ca600e6573736a7669d)**[/](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d4780020ca600e6573736a7669d)**Module Introduction

The next resource we will examine, with respect to virtualization, is memory. Memory virtualization should ring a bell; specifically, it is very closely related to the operating systems concept of virtual memory! As such, we will begin our discussion by recalling virtual memory concepts and then discuss memory virtualization as an extension of these techniques. VMWare has pioneered some interesting and clever techniques in the realm of memory reclamation from Guest OSes, which will also be covered in this module.

### [Module 10 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d4780020ca600e6573736a7669d)One-Level Page Mapping

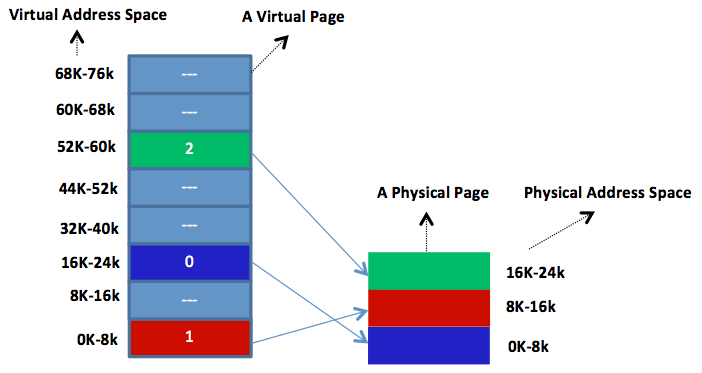
### Learning Objectives

* Identify how most general-purpose operating systems support virtual memory

Discuss the one-level page mapping between virtual and physical addresses

## **Virtual Memory and One-Level Page Mapping**

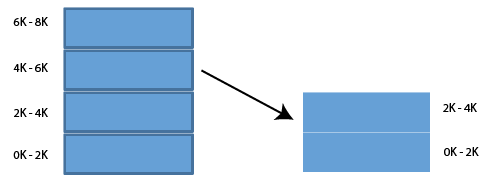
Virtual memory is a well-known virtualization technique supported in most general-purpose OSs. The basic idea of virtual memory is that each process is provided with its own virtual address space, broken up into chunks called virtual pages. A page is a contiguous range of addresses. As shown in Figure 3.24, virtual memory maps virtual pages to physical pages in what is called a page table. We call this **one-level page mapping** between two types of addresses: the virtual and the physical. Each process in the OS has its own page table. A main observation pertaining to page tables is that not all virtual pages of a process need to be mapped to respective physical pages in order for the process to execute. When a process references a page that exists in the physical memory (i.e., there is a mapping between the requested virtual page and the corresponding physical page), a page hit is attained. On a page hit, the hardware obtains the required virtual to physical mapping with no further actions. In contrary, when a process references a page that does not exist in the physical memory, a page miss is incurred. On a page miss, the OS is alerted to handle the miss. Subsequently, the OS fetches the missed page from disk storage and updates the relevant entry in the page table.

Figure 3.24: Mapping a process' virtual address space to physical address space. This is captured in what is called a page table. Each process has its own page table.

**did I get this**

**learn by doing**

Consider the following one-level virtual-to-physical mappings with virtual and physical pages of 2KB. Every page begins on a multiple of 2,048 and ends 2,047 addresses higher. Thus, 6K-8K really means 6,144-8,191.



Assume a program with the following instruction:

**MOV REG, 4095**

That is, the program will attempt to access address 4,095.

Assume a program with the following instruction:

**MOV REG, 5016**

[Module 10 **/**](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d4780020ca600e6573736a7669d)Two-Level Page Mapping

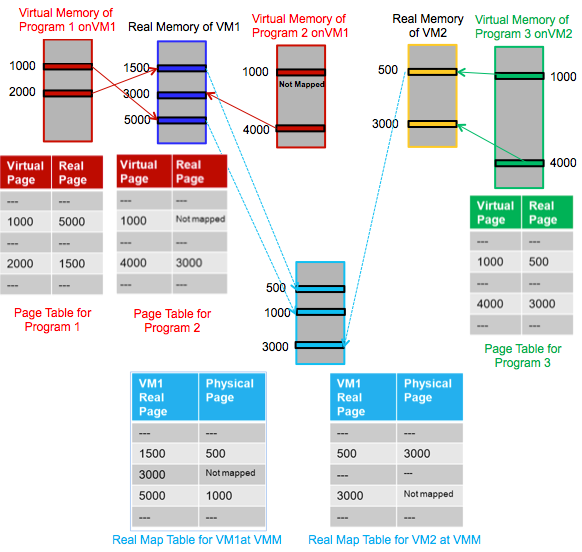
**Learning Objectives**

* Identify the difference between one-level page mapping, as provided in traditional operating systems, and two-level page mapping, as provided in system memory virtualization

Discuss multiple-level page mapping as implied by various virtualized environments, such as native and user-mode hosted virtualized systems

Contrary to OSs in traditional systems, with system virtualization, the hypervisor allocates a contiguous addressable memory space for each created VM (not process). This memory space per a VM is called **real memory**. In return, each guest OS running in a VM allocates a contiguous, addressable memory space for each process in its real memory. This memory space per a process is called virtual memory (same name as in traditional systems). Each guest OS maps the virtual memories of its processes to the real memory of the underlying VM, while the hypervisor maps the real memories of its VMs to the system physical memory. Clearly, in contrast to traditional OSs, this entails two levels of mappings between three types of addresses: virtual, real, and physical. In fact, these virtual-to-real and real-to-physical mappings define system memory virtualization. This basic idea of memory virtualization via two-level page mapping is summarized in Video 3.10:

Video 3.10: Memory Virtualization.

Figure 3.25: Memory virtualization in a native system VM.

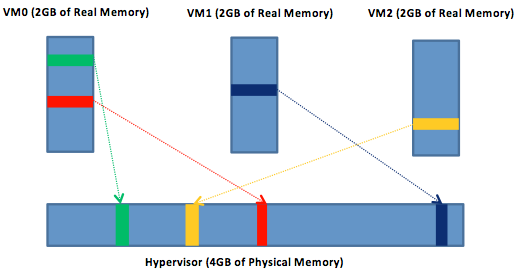
Similar to any general-purpose OS, a guest OS would still own its set of page tables. In addition, the hypervisor would own another set of page tables for mapping real-to-physical addresses. The page tables in the hypervisor are called real map tables. Figure 3.25 demonstrates system memory virtualization in a native system VM. It shows page tables maintained by guest VMs and real map tables maintained by the hypervisor. Each entry in a page table maps a virtual page of a program to a real page in the respective VM. Likewise, each entry in a real map table maps a real page in a VM to a physical page in the physical memory. When a guest OS attempts to establish a valid mapping entry in its page table, it traps to the hypervisor. Subsequently, the hypervisor establishes a corresponding mapping in the relevant VM's real map table.

[Module 10 **/**](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d4780020ca600e6573736a7669d)Memory Overcommitment

**Learning Objectives**

* Define memory overcommitment, and discuss the benefits it provides to virtualized systems

In memory virtualization, the combined total size of real memories can grow beyond the actual size of physical memory. This concept is typically called memory overcommitment. Memory overcommitment ensures that physical memory is highly utilized by active, real memories (assuming multiple VMs running simultaneously). Indeed, without memory overcommitment, the hypervisor can only run VMs with a total size of real memories less than that of the physical memory. For instance, Figure 3.26 shows a hypervisor with 4GB of physical memory and three VMs, each with 2GB of real memory. Without memory overcommitment, the hypervisor can only run one VM because of not having enough physical memory to assign to two VMs at once. Although each VM would require only 2GB of memory, wherein the hypervisor has 4GB of physical memory, this memory cannot be afforded because hypervisors generally require overhead memories (e.g., to maintain various virtualization data structures).

Figure 3.26: A hypervisor with 4GB of physical memory, enabling three VMs at once with a total of 6GB of real memory.

To this end, in practical situations, some VMs might be lightly loaded, while others might be heavily loaded. Lightly loaded VMs can cause some pages to sit idle, while heavily loaded VMs can result in memory page thrashing. To deal with such a situation, the hypervisor can take (or steal) the inactive physical memory pages away from idle VMs and provide them to heavily loaded VMs. As a side effect, hypervisors usually write zeros to the stolen/reclaimed, inactive physical memory pages in order to avert information leaking among VMs.

### [Module 10 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d4780020ca600e6573736a7669d)Reclamation Techniques and VMware Memory Ballooning

### Learning Objectives

* Describe what reclamation techniques are and why they are needed

Explain memory ballooning in VMware ESX as an example of a reclamation technique

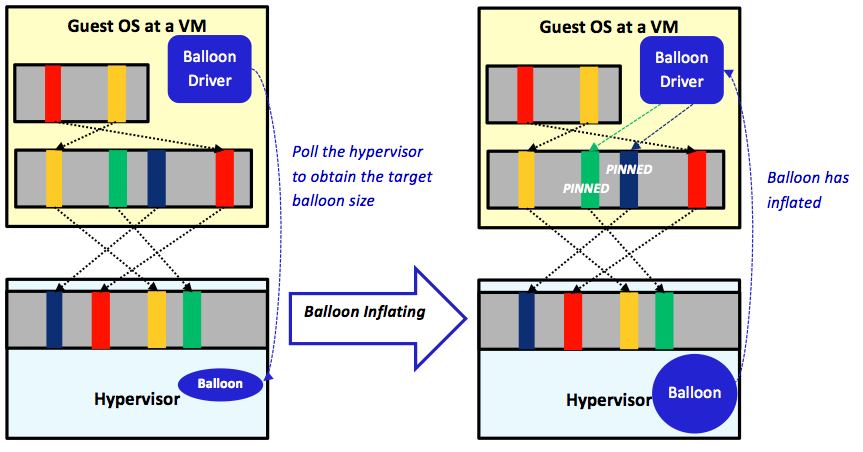
## **Memory Reclamation**

To maintain full isolation, guest OSs are kept unaware that they are running inside VMs. VMs are also kept unaware of the states of other VMs running on the same physical host. Furthermore, with multiple levels of page mapping, VMs remain oblivious of any physical memory shortage. Therefore, when the hypervisor runs multiple VMs at a physical host, and the physical memory turns stressed, none of the VMs can automatically help in freeing up memory.

The hypervisor deals with the situation by applying a **reclamation technique**. As its name suggests, a reclamation technique attempts to reclaim inactive real memory pages at VMs and make them available for the hypervisor when experiencing a memory shortage. Video 3.11 describes a couple of techniques which can be used to reclaim memory from guest operating systems:

Video 3.11: Advanced Memory Management.

One of the popular reclamation techniques is the **ballooning process** introduced in VMware ESX, which has been the basis for similar techniques in other hypervisors.

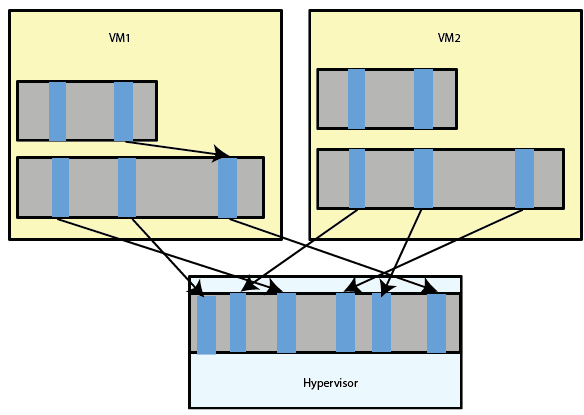
Figure 3.27: The ballooning process in VMware ESX.

In VMware ESX, a balloon driver must be installed and enabled in each guest OS as a pseudo-device driver. The balloon driver regularly polls the hypervisor through a private channel to obtain a target balloon size. As illustrated in Figure 3.27, when the hypervisor experiences memory shortage, it inflates the balloon by setting a proper target balloon size. Figure 3.27(a) shows four real memory pages mapped to four physical pages, of which only two pages are actually active (the red and the yellow ones). Without involving the ballooning process, the hypervisor is unaware of the other two inactive pages (the green and the dark-blue ones) because they are still mapped to physical pages. Consequently, the hypervisor will not be able to reclaim inactive pages unless getting informed. With memory ballooning, however, the hypervisor can set the balloon target size to an integer number (say 2 or 3). When recognized by the balloon driver at the guest OS, the driver checks out the pages, locates the two inactive pages, and pins them (see Figure 3.27(b)). The pinning process is carried out by the guest OS via ensuring that the pinned pages cannot be read/written by any process during memory reclamation. After pinning the inactive pages, the balloon driver transmits to the hypervisor the addresses of the pinned pages. Subsequently, the hypervisor proceeds safely with reclaiming the respective physical pages and allocating them to needy VMs. Last, to unpin pinned pages, the hypervisor deflates the balloon by setting a smaller target balloon size and communicates that to the balloon driver. When received by the balloon driver, it unpins the pinned pages so the guest OS can utilize them. More information about the ballooning process can be found in VMware ESX's documentation.

**did I get this**

**did I get this**

Consider the following memory state as imposed by two VMs running on a single physical host:



Assume that the shown hypervisor employs a ballooning process.

[**MY RESPONSE...**](https://oli.cmu.edu/jcourse/webui/resolver/link/resource.do?src=33882d4d80020ca601d76a087ce53e5b&dst=_u03_m04_5_feedback)

## **Resource Virtualization: Memory Summary**

* In the earliest days, either a process fit a memory or it could not be run.
* Virtual memory changed the status quo by allowing a process that cannot fit a physical memory to run as if it essentially fits the memory.
* An indirect inference of virtual memory is that multiple processes that cannot collectively fit a certain physical memory can now run altogether on this same physical memory.
* The basic idea of virtual memory is that each process is provided with its own virtual address space.
* The virtual address space of each process is translated to the physical address space that the physical memory uses.
* The translation of virtual addresses to physical addresses is maintained in a software data structure called the page table.
* In traditional systems (i.e., nonvirtualized environments), the virtual-to-physical translation is calledone-level page mapping.
* In virtualized environments (i.e., when a hypervisor is involved), the virtual-to-physical translation is extended at least one more level and called two-level page mapping.
* The two-level page mapping entails two consecutive translations, virtual-to-real then real-to-physicaltranslations. In this case, the real address space refers to the memory space of a VM, while the virtual and physical address spaces relate to the traditional memory spaces of processes and the physical memory.
* As a result, memory virtualization in virtualized environments typically is perceived as an extension to the classic virtual memory concept supported in most general-purpose OSs.
* When the combined total size of real memories grows beyond the actual size of the underlying physical memory, memory overcommitment is attained.
* Memory overcommitment improves memory utilization via allowing VMs with aggregate real memories larger than the physical memory to run simultaneously.
* Memory overcommitment, however, necessitates reclaiming inactive real memory pages at VMs and relocating them to the hypervisor when experiencing a physical memory shortage. This is calledreclamation technique.
* One of the popular reclamation techniques is the ballooning process incorporated in VMware ESX.

[Module 11 **/**](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d6080020ca600c528dfa729ac65)Module Introduction

Our final segment in resource virtualization is that of IO devices. In this respect, we can consider the VMM or hypervisor the arbiter of communication between multiple guests and the physical hardware, multiplexing the usage (in time/space or both), depending on the actual device being shared.

The virtualization strategy for a given I/O device type consists of:

1. Constructing a virtual version of that device, and
2. Virtualizing the I/O activity routed to the device.

Typical I/O devices include disks, network cards, displays, and keyboards. As discussed previously, the hypervisor might create a virtual display as a window on a physical display. In addition, a virtual disk can be created by assigning to it a fraction of the physical disk's storage capacity. After constructing virtual devices, the hypervisor ensures that each I/O operation is carried out within the bounds of the requested virtual device. For instance, if a virtual disk is allocated 100 cylinders from among 1,000 cylinders provided by a physical disk, the hypervisor guarantees that no I/O request intended for that virtual disk can access any cylinder other than the 100 assigned to it. More precisely, the disk location in the issued I/O request will be mapped by the hypervisor to only the area where the virtual disk has been allocated on the physical disk. Next, we cover some I/O basics and then move on to the details of I/O virtualization. After covering the basics, we will examine the case of Xen, and how it handles IO virtualization.

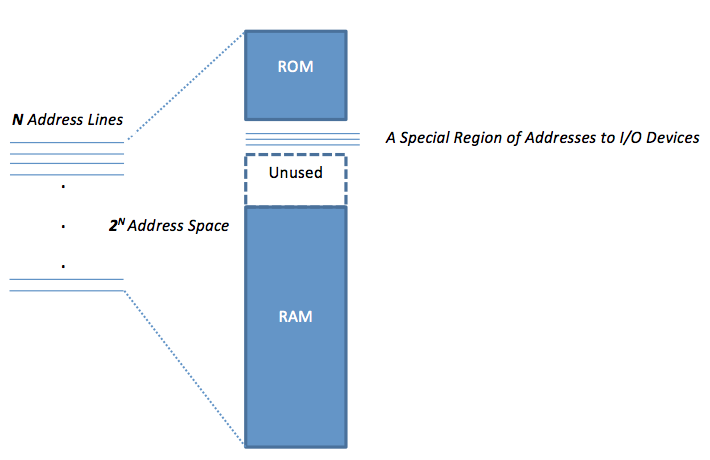
### [Module 11 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d6080020ca600c528dfa729ac65)I/O Basics

### Learning Objectives

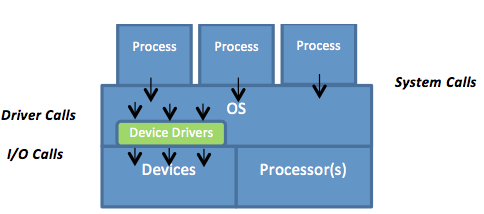
* Recognize how CPU and I/O devices communicate in traditional systems

## **Basics of I/O**

To begin, each I/O device has a device controller. A device controller can typically be signaled either by a**privileged I/O instruction** or **memory-mapped I/O**. I/O instructions are provided by ISAs. Intel-32 is an example of a processor that provides I/O instructions in its ISA. Many recent processors, however, allow performing I/O between the CPU and the device controllers through memory-mapped I/O (e.g., RISC processors). As shown in Figure 3.28, with memory-mapped I/O, a specific region of the physical memory address space is reserved for accessing I/O devices. These addresses are recognized by the memory controller as commands to I/O devices and do not correspond to memory physical locations. Different memory-mapped addresses are used for different I/O devices. Finally, in order to protect I/O devices, both I/O instructions and memory-mapped addresses are handled in system mode, thus becoming privileged.

Figure 3.28: Memory mapped I/O with a specific region in the RAM address space for accessing I/O devices.

Because I/O operations are executed in system mode, user programs can only invoke them through OS system calls (assuming traditional systems). The OS abstracts most of the details of I/O devices and makes them accessible through only well-defined interfaces. Figure 3.29 shows the three major interfaces that come into play when a user program places an I/O request. These are the **system call interface**, the **device driver interface**, and the **operation-level interface**. Starting an I/O operation, a user I/O request causes an OS system call that transfers control to the OS. Next, the OS calls device drivers (a set of software routines) via the device driver interface. A relevant device driver routine converts the I/O request to an operation specific to the requested physical device. The converted operation is subsequently carried through the operation-level interface to the corresponding physical device.

Figure 3.29: The three major interfaces involved in I/O operations: system call, device driver, and operation-level interfaces.

### [Module 11 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d6080020ca600c528dfa729ac65)Virtualizing I/O Devices

### Learning Objectives

* Identify how many I/O device drivers can/should be supported per each physical device for different virtualized systems, such as native and dual-mode hosted virtualized systems

Recognize the need for and ease of intercepting I/O requests by the hypervisor

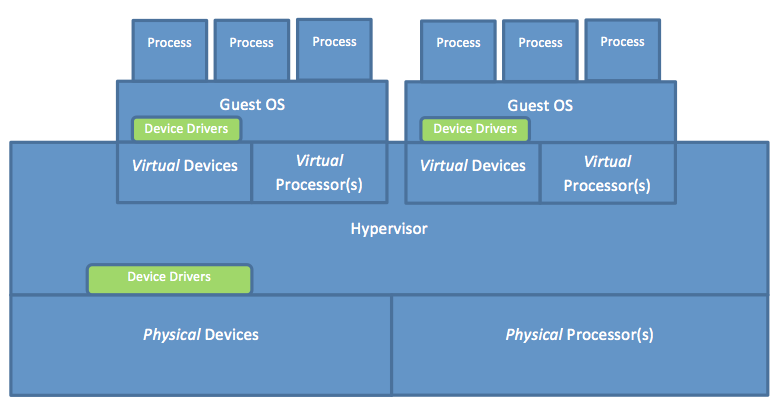
Identify at which system interfaces the hypervisor can intercept I/O requests

* Explain the pros and cons of intercepting I/O requests at different system interfaces

Describe the overall I/O virtualization process as applied to a network interface card

## **I/O Virtualization**

I/O virtualization allows a single physical I/O device to be shared by more than one guest OS. Figure 3.30 demonstrates multiple guest OSs in native system VMs sharing a single hardware machine. As shown, the hypervisor constructs virtual devices from physical devices. A main observation is that both the guest OSs and the hypervisor must have device drivers encapsulating the interfaces to the devices. This means that with virtualization, two different device drivers must be supported per each device versus only one without virtualization. In reality, this is a problem because vendors of devices usually supply drivers for only the major OSs but not for hypervisors (though this could change in the near future). One way to circumvent such a problem is to collocate the hypervisor with a major OS (e.g., Linux) on the same machine. This way, I/O requests can be handled by the OS which holds all requisite I/O drivers. This is the approach adopted by Xen and discussed on the next page.

Figure 3.30: Logical locations of device drivers in multiple guest OSs in native system VMs sharing a single hardware machine.

Moreover, with I/O virtualization, every I/O request issued by a user program at a guest VM should be intercepted by the hypervisor because I/O requests are all privileged and thus need to be controlled by the hypervisor. Clearly, this would entail a trap to the guest OS for every I/O request. All I/O requests are privileged, whether issued using I/O instructions or memory-mapped I/O; hence, they are not critical instructions, and they all trap to the hypervisor. As such, the hypervisor can easily intercept every I/O request simply when trapping. In principle, the hypervisor can intercept I/O requests at any of the three interfaces: the system call interface, the device driver interface, or the operation-level interface.

If the hypervisor intercepts an I/O request at the operation-level interface, some essential information about the I/O action might be lost. The hypervisor needs that information to handle I/O requests correctly. When an I/O request arrives at the device driver interface, it might get transformed into a sequence of instructions. When the sequence of instructions is received at the operation-level interface, it becomes difficult for the hypervisor to identify them as instructions for a single I/O request. For example, a disk write becomes multiple store instructions in case of memory-mapped I/O or multiple ISA I/O instructions. Hence, intercepting I/O requests at the operation-level interface typically is avoided. In contrast, intercepting an I/O request at the device driver interface allows the hypervisor to efficiently map the request to the respective physical device and transmit it through the operation-level interface. Clearly, this process is a natural point for I/O virtualization; yet it would oblige hypervisor developers to learn about the different device driver interfaces of various guest OSs in order to be able to intercept I/O requests. Last, intercepting I/O requests at the system call interface (i.e., the application binary interface [ABI]) might theoretically make the I/O virtualization process easier, whereby the entire I/O operation could be handled for each request by the hypervisor (the solo controller in this case). To achieve that goal, however, the hypervisor has to emulate the ABI routines of every guest OS (different OSs have different ABI routines). Consequently, hypervisor developers need also to learn about the internals of every potential guest OS. Furthermore, emulating ABI routines can degrade system performance due to the overhead imposed by the emulation process. In practice, intercepting I/O requests at the device driver interface can be more efficient. In the next video, we discuss the overall network virtualization process as applied to a physical network adapter.

Video 3.12: Shared Devices in Virtualization. ([Alternative version](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m05_resource_virtualization_IO/webcontent/video_4_shared_devices.pdf))

As explained in Video 3.12, one physical adapter card can appear as multiple virtual network interface cards (vNICs), each with a separate MAC address and on the same network as the physical one. To network infrastructures, such as LANs and SANs, vNICs appear as regular physical cards.

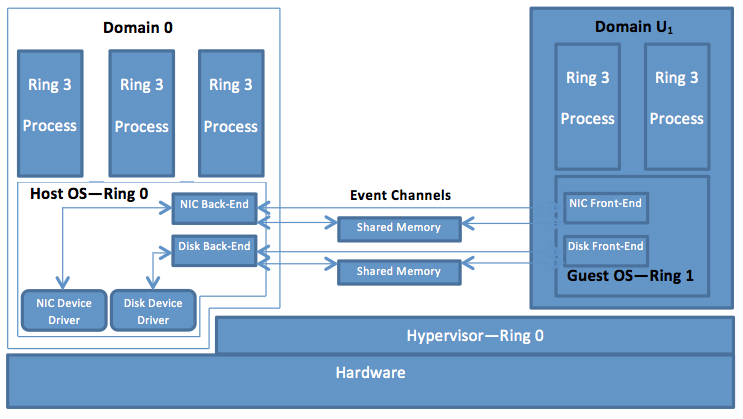
### [Module 11 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d6080020ca600c528dfa729ac65)Xen's Approach to I/O Virtualization

### Learning Objectives

* Discuss Xen's approach to I/O virtualization

## **I/O Virtualization in Xen**

As a concrete example, we discuss Xen's approach to I/O virtualization. As we pointed out earlier, to get around the problem of having device drivers for the hypervisor as well as the guest OSs, Xen collocates its hypervisor with a traditional general-purpose OS. Figure 3.31 shows a host OS and the Xen hypervisor executing in full privileges at ring 0. Guest OSs run unprivileged at ring 1, while all processes at all domains (i.e., virtual machines) run unprivileged at ring 3. Clearly, the figure assumes a system with four rings (e.g., Intel-32). On systems with only two levels of privileges, the hypervisor and the host OS can execute in system mode, while domains and processes can execute in user mode. As illustrated in the figure, Xen eliminates the device drivers entirely from guest OSs and provides a direct communication between guest OSs at domain U and the host OS at domain 0. More precisely, every domain Ui in Xen will not hold any virtual I/O devices or relevant drivers. Rather, every I/O request is now transferred directly to domain 0, which by default hosts all the required device drivers necessary to satisfy all I/O requests. For instance, rather than using a device driver to control a virtual network card interface (vNIC), with Xen network, frames/packets are transferred through event channels directly to and from domain 0. This is done using NIC frontend and backend interfaces at domain Uj (in which j > 0) and U0, respectively. Likewise, no virtual disk is exposed to any guest OS, and all disk data blocks imposed by file reads and writes are delegated by Xen to domain 0.

Figure 3.31: Xen's approach to I/O virtualization assuming a system with four rings (e.g., Intel-32). Xen collocates an OS, at a VM0 called domain 0, with the hypervisor on the physical platform to "borrow" its device drivers and avoid coding them in the hypervisor. This makes the hypervisor "thinner" and accordingly more reliable. Also, it makes it easier on the hypervisor developers.

## **Resource Virtualization: I/O Summary**

* To virtualize an I/O device, we ought to follow two main steps: (1) construct a virtual version of the device and (2) virtualize the I/O activity routed to the device.
* Constructing a virtual version of an I/O device entails sharing the device across multiple guest OSs.
* Virtualizing the I/O activity to an I/O device passes through the device's controller (each I/O device has a device controller).
* A device controller can be signaled via either a privileged I/O instruction (defined in ISA) or memory-mapped I/O.
* I/O instructions and memory-mapped addresses are handled in system mode to protect the called I/O devices.
* I/O instructions and memory-mapped addresses are not critical, thus can be easily handled by the hypervisor after naturally trapping to it (i.e., because they are privileged and not critical, they will naturally trap to the hypervisor when run in user mode).
* General-purpose OSs abstract most of the details of I/O devices and make them accessible only through well-defined interfaces, such as the system call interface, the device driver interface, and the operation-level interface.
* In the presence of a hypervisor, two different device drivers must be supported per each I/O device versus only one in traditional nonvirtualized systems.
* The redundancy of device drivers in the presence of a hypervisor is usually circumvented by collocating the hypervisor with a major OS (e.g., Linux) on the same machine. Subsequently, the hypervisor leverages the device drivers of the major OS without requiring special device drivers (Xen applies this approach).
* Because all I/O instructions are privileged, they need to be intercepted by the hypervisor.
* In principle, the hypervisor can intercept I/O requests at any of the three interfaces: the system call interface, the device driver interface, and the operation-level interface.
* Intercepting I/O requests at the operation-level interface might lead to the loss of some essential information about I/O actions.
* Intercepting I/O requests at the system call interface (i.e., the ABI) entails emulating the ABI routines of every guest OS (different OSs have different ABI routines).
* In practice, intercepting I/O requests at the device driver interface is typically the most efficient approach because it avoids emulating the ABI routines of every guest OS and losing some necessary information about I/O actions.

### [Module 12 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d7380020ca6002d4754dcc0f7ab)Module Introduction

Now that we have covered some of the theory behind virtualization, we can look at a few case studies. Specifically, we start out with an overview of current virtualization suites in the market, which includes a feature-wise comparison of VMWare, XenServer, HyperV and RHEV. Next, we will take a closer look at Amazon EC2 as a public IaaS provider and some of the design decisions made with respect to providing a general, broad-based public IaaS cloud service.

### [Module 12 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d7380020ca6002d4754dcc0f7ab)A Taxonomy of Virtualization Suites

### Learning Objectives

* Differentiate popular virtualization suites by architecture and features

## **Virtualization Suites**

We briefly survey some of the current and common virtualization software suites and distinguish between virtualization suites and hypervisors. Many vendors often use hypervisor and virtualization suiteinterchangeably. As discussed throughout this chapter, a hypervisor is primarily responsible for running multiple virtual machines (VMs) on a single physical host. A virtualization suite comprises various software components and individual hypervisors that enable the management of many physical hosts and VMs. A management component typically issues commands to the hypervisor to create, destroy, manage, and migrate VMs across multiple physical hosts.

The table below shows our taxonomy of four virtualization suites, [vSphere 5.1](http://www.vmware.com/), [Hyper-V](http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx) , [XenServer 6](http://support.citrix.com/product/xens/v6.0/" \t "_blank), and[RHEV 3](http://www.redhat.com/promo/rhev3/). We compare the suites in terms of multiple features, including the involved hypervisor, the virtualization type, the allowable maximum number of vCPUs per VM, the allowable maximum memory size per VM, and whether memory overcommitment, page sharing, and live migration are supported. In addition, we indicate whether the involved hypervisors contain device drivers, and we list some of the popular cloud vendors that utilize such hypervisors.

To elaborate on some of the features, live migration allows running VMs to be seamlessly shifted from one physical machine to another. It enables many management features, such as maintenance, power-efficient dynamic server consolidation, and workload balancing, among others. Page Sharing refers to sharing identical memory pages across VMs. This renders effective when VMs use similar OS instances. Finally, some hypervisors eliminate device drivers entirely at guest OSs and provide direct communications between guest OSs and host OSs collocated with hypervisors (similar to what we discussed in the Section "Xen's Approach to I/O Virtualization").

| **Feature** | **vSphere 5.1** | **Hyper-V 2012** | **XenServer 6** | **RHEV 6** |
| --- | --- | --- | --- | --- |
| **Hypervisor Name** | ESXi | Hyper-V | Xen | KVM |
| **CPU virtualization support** | Full virtualization | Paravirtualization | Paravirtualization | Full virtualization |
| **Maximum vCPUs per VM** | 160 | 320 | 64 | 160 |
| **Maximum Memory per VM** | 1TB | 1TB | 128GB | 2TB |
| **Memory Overcommitment Support** | Yes | Yes | Yes | Yes |
| **Page Sharing Support** | Yes | No | No | No |
| **Live Migration Support** | Yes | Yes | Yes | Yes |
| **Contains Device Drivers** | Yes | No | No | Yes |
| **Common Cloud Vendors** | [vCloud Hybrid Service](http://www.vmware.com/products/vcloud-hybrid-service) | [Microsoft Azure](http://www.windowsazure.com/en-us/) | [Amazon EC2](http://aws.amazon.com/ec2) and[Rackspace](http://www.rackspace.com/) | [IBM SmartCloud](http://www.ibm.com/cloud-computing/us/en/) |

### [Module 12 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d7380020ca6002d4754dcc0f7ab)Amazon's Elastic Compute Cloud

### Learning Objectives

* Describe Amazon's EC2 Service

## **Amazon EC2**

Amazon Elastic Compute Cloud (Amazon EC2) is a vital part of Amazon's cloud computing platform, Amazon Web Services (AWS). On August 25, 2006, Amazon launched EC2, which together with Amazon Simple Storage Service (Amazon S3) marked a change in the way IT was done. Amazon EC2 is a highly reliable and scalable infrastructure as a service (IaaS) with a utility payment model. It allows users to rent virtual machines (VMs) and pay for the resources that they actually consume. Users can set up and configure everything in their VMs, ranging from the operating system to any application. Specifically, a user can boot an Amazon Machine Image (AMI) to create a VM, referred to in Amazon's parlance as an **instance**. AMI is a virtual appliance (or a VM image) that contains the user's operating system, applications, libraries, data, and associated configuration settings.

Users can create EC2 instances either by using default AMIs prepackaged by Amazon or by developing their own AMIs using Amazon's bundling tools. Default AMIs are preconfigured with an ever-growing list of operating systems, including Red Hat Enterprise Linux, Windows Server, and Ubuntu. A wide selection of free software provided by Amazon can also be directly incorporated into AMIs and executed over EC2 instances. For example, Amazon provides software for databases (e.g., Microsoft SQL), application servers (e.g., Tomcat Java Web Application), content management (e.g., MediaWiki), and business intelligence (e.g., Jasper Reports). Added to the wide assortment of free software, Amazon services (e.g., Amazon Relational Database Service, which supports MySQL, Oracle, and Microsoft SQL databases) can be further employed in conjunction with EC2 instances. Finally, users can always configure, install and run at any time any compatible software on EC2 instances, exactly as is the case with regular physical machines.

## **Amazon EC2 Virtualization Technology**

Amazon EC2 demonstrates the power of cloud computing. Under the hood, it is a marvel of technology. As of March 2012, it was hosting around 7,100 server racks with a total of 454,400 blade servers, assuming 64 blade servers per rack.[[1](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_20)]Above its data centers, Amazon EC2 presents a true virtualization environment using the Xen hypervisor. Xen is a leading example of system virtualization, initially developed as part of the Xenoserver project at the Computer Laboratory, Cambridge University.[[2](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_9)]Currently, Xen is maintained by an open source community.[[3](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_34)]The goal of Xen is to provide IaaS with full isolation and minimal performance overhead on conventional hardware. As discussed previously in this unit, Xen is a native hypervisor that runs on bare metal at the most privileged CPU state. Amazon EC2 uses a highly customized version of Xen[[4](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_2)]to provision and isolate user instances rapidly, consolidate instances to improve system utilization, tolerate software and hardware failures by saving and migrating instances, apply system load balancing through live and seamless migration of instances, and more.

Amazon EC2 instances can be created, launched, suspended, resumed, and terminated as needed. The instances are system VMs composed of virtualized (or paravirtualized) sets of physical resources, including CPU, memory, and I/O components. To create instances, Xen starts a highly privileged instance (domain 0) at a host OS out of which other user instances (domain U) can be instantiated with guest OSs (see Fig. 1.25). As host OSs, Novell's SUSE Linux Enterprise Server, Solaris and Open-Solaris, NetBSD, Debian, and Ubuntu, among others, can be used. It is not known, however, which among these host OSs Amazon EC2 supports, but Linux, Solaris and OpenSolaris, FreeBSD, NetBSD, and others can be employed as guest OSs. Among the guest OSs that Amazon EC2 supports are Linux, OpenSolaris, and Windows Server 2003,[[5](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_37)]Amazon EC2’s guest OSs are run at a lesser privileged ring than the host OS and the hypervisor. Clearly, this helps isolate the hypervisor from guest OSs and guest OSs from each other, a key requirement on Amazon AWS’s cloud platform. Nonetheless, running guest OSs in unprivileged mode violates the usual assumption that OSs must run in system mode. To circumvent consequent ramifications, Xen applies a paravirtualized approach whereby guest OSs are modified to run at a downgraded privileged level. As a result, sensitive instructions are enforced to trap to the hypervisor for verification and execution. Linux instances (and most likely OpenSolaris) on Amazon EC2 use Xen’s paravirtualized mode, and it is conjectured also that Windows instances do so. [[5](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_37)] [[6](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_6)]Upon provisioning instances, Xen provides each instance with its own vCPU(s) and associated ISA. The complexity of this step depends entirely on the architecture of the underlying pCPU(s). To this end, it is not clear what vCPU scheduler Amazon EC2 applies, but Xen’s default scheduler is the Credit Scheduler (as discussed in the **Resource Virtualization: CPU** module). Amazon EC2 is thought to have a modified version of the Credit Scheduler.[[5](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_37)]

Memory and I/O resources are virtualized in Xen in a way similar to that described in the **Resource Virtualization: Memory** module. First, Xen uses a two-level page mapping method. Second, the hardware page tables are allocated and managed by guest OSs, with a minimal involvement from the hypervisor.[[5](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_1_ec2.xml#_37)]Speciﬁcally, guest OSs can read directly from hardware page tables, but writes are intercepted and validated by the Xen hypervisor to ensure safety and isolation. For performance reasons, however, the guest OSs can batch write requests to amortize the overhead of passing by the hypervisor per every write request. Finally, with Xen, existing hardware I/O devices are not emulated as is typically done in fully virtualized environments. In contrast, I/O requests are always transferred from user instances to domain 0, and vice versa, using a shared memory communication paradigm as demonstrated in Fig. 1.25. At domain 0, device drivers of the host OS are borrowed to handle the I/O requests.

[**MY RESPONSE...**](https://oli.cmu.edu/jcourse/webui/resolver/link/resource.do?src=33882d7680020ca60065944360203ba2&dst=u03_m06_feedback1)

### [Amazon EC2 Properties: Feedback, Comments, and B](https://oli.cmu.edu/jcourse/webui/resolver/link/resource.do?src=33882d7680020ca60065944360203ba2&dst=u03_m06_feedback1)

### [Module 12 ****/****](https://oli.cmu.edu/jcourse/webui/syllabus/module.do?context=33882d7380020ca6002d4754dcc0f7ab)Amazon EC2 Properties

### Learning Objectives

* Identify the various techniques used by Amazon to offer typical cloud properties to users

Amazon EC2 is designed using virutalization technology in order to meet certain performance, scalability, ﬂexibility, security, and reliability criteria.

## **Elasticity**

In leveraging a major beneﬁt offered by virtualization, Amazon EC2 allows users to statically and dynamically scale up and down their EC2 clusters. In particular, users can always provision and deprovision virtual EC2 instances by manually starting and stopping any number of them using the AWS management console, the Amazon command line tools and/or the Amazon EC2 API. In addition, users can employ Amazon's CloudWatch to monitor EC2 instances in realtime and automatically respond to changes in computing requirements. CloudWatch is an Amazon service that allows users to collect statistics about their cluster resource utilization, operational performance, and overall resource demand patterns. Metrics, such as CPU utilization, disk operations, and network trafﬁc, can be aggregated and fed to the Amazon's Auto Scaling process enabled by CloudWatch. The Auto Scaling process can subsequently add or remove instances so that performance is maintained and costs are saved. In essence, Auto Scaling allows users to closely follow the demand curve of their applications and synergistically alter their EC2 clusters according to conditions they deﬁne (e.g., add three more instances to the cluster when the average CPU utilization exceeds 80%).

## **Scalability and Performance**

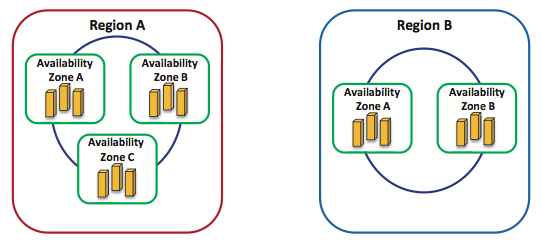
|  |
| --- |
| Table 1.2: Amazon EC2 Instance Types as of March 4, 2013. |
| | **Instance Type** | **Instance Name** | **CPU Capacity** | **Memory Size** | **Storage Size and Type** | **Platform** | | --- | --- | --- | --- | --- | --- | | Standard | M1 Small | 1 vCPU with 1 ECU | 1.7GB | 160GB local storage | 32-bit or 64-bit | | M1 Medium | 1 vCPU with 2 ECUs | 3.75GB | 410GB local storage | 32-bit or 64-bit | | M1 Large | 2 vCPUs, each with 2 ECUs | 7.5GB | 850GB local storage | 64-bit | | M1 Extra Large | 4 vCPUs, each with 2 ECUs | 15GB | 1690 GB local storage | 64-bit | | M3 Extra Large | 4 vCPUs, each with 3.25 ECUs | 15GB | EBS storage only | 64-bit | | M3 Double Extra Large | 8 vCPUs, each with 3.25 ECUs | 30GB | EBS storage only | 64-bit | | Micro | Micro | Up to 2 ECUs | 613MB | EBS storage only | 32-bit or 64-bit | | High Memory | Extra Large | 2 vCPUs, each with 3.25 ECUs | 17.1GB | 420GB local storage | 64-bit | | Double Extra Large | 4 vCPUs, each with 3.25 ECUs | 34.2GB | 850GB local storage | 64-bit | | Quadruple Extra Large | 8 vCPUs, each with 3.25 ECUs | 68.4GB | 1690GB local storage | 64-bit | | High CPU | Medium | 2 vCPUs, each with 2.5 ECUs | 1.7GB | 350GB local storage | 32-bit or 64-bit | | Extra Large | 8 vCPUs, each with 2.5 ECUs | 7GB | 1690GB local storage | 64-bit | | Cluster Compute | Eight Extra Large | 88 ECUs | 60.5GB | 3370GB local storage | 64-bit and 10 GbE | | High-Memory Cluster | Eight Extra Large | 88 ECUs | 244GB | 240GB local storage | 64-bit and 10 GbE | | Cluster GPU | Quadruple Extra Large | 33.5 ECUs 2 × NVIDIA Tesla Fermi M2050 GPUs | 22GB | 1690GB local storage | 64-bit and 10 GbE | | High I/O | Quadruple Extra Large | 35 ECUs | 60.5GB | 2 × 1024GB SSD-based local storage | 64-bit and 10 GbE | | High Storage | Eight Extra Large | 35 ECUs | 117GB | 24 × 2TB hard disk drive local storage | 64-bit and 10 GbE | |

Amazon EC2 instances can scale to more than 255 pCPUs per host,[[1](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_36)]128 vCPUs per guest, 1TB of RAM per host, up to 1TB of RAM per unmodified guest, and 512GB of RAM per paravirtualized guest.[[2](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_35)]In addition, Amazon EC2 reduces the time needed to boot a fresh instance to seconds, thus expediting scalability as the needs for computing varies. To optimize performance, Amazon EC2 instances are provided in various resource capacities that can suit different application types, including CPU-intensive, memory-intensive, and I/O-intensive applications (see Table 1.2). The vCPU capacity of an instance is expressed in terms of elastic compute units (ECU). Amazon EC2 uses ECU as an abstraction of vCPU capacity, whereby one ECU provides the equivalence of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.[[3](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_1)]Different instances with different ECUs provide different application runtimes. The performances of instances with identical type and ECUs may also vary as a result of what is called performance variation[[4](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_13)] [[8](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_25)]in cloud computing.

## **Flexibility**

Amazon EC2 users are provided with complete control over EC2 instances, with a root access to each instance. They can create AMIs with software of their choice and apply many of Amazon services, including Amazon Simple Storage Service (Amazon S3), Amazon Relational Database Service (Amazon RDS), Amazon SimpleDB, and Amazon Simple Queue Service (Amazon SQS). These services and the various available Amazon EC2 instance types can jointly deliver effective solutions for computing, query processing, and provide storage across a wide range of applications. For example, users running I/O-intensive applications, such as data warehousing and Hadoop MapReduce, can exploit high-storage instances. On the other hand, for tightly coupled, network-intensive applications, users can utilize high-performance computing (HPC) clusters.

In addition, users have the flexibility to choose among multiple storage types that can be associated with their EC2 instances. First, users can rent EC2 instances with local instance-store disks as root devices. Instance-store volumes are volatile storage and cannot survive stops and terminations. Second, elastic block storage (EBS) volumes can be attached to EC2 instances, which provide the instances with raw block devices. The block devices can then be formatted and mounted with any file system at EC2 instances. EBS volumes are persistent storage and can survive any EC2 instance state, including stops and terminations. EBS volumes of sizes from 1GB to 1TB can be defined, and RAID arrays can be created by combining two or more volumes. EBS volumes can even be attached or detached from instances while they are running. They can also be moved from one instance to another, thus remaining independent of any instance. Finally, applications running on EC2 instances can access Amazon S3 through a defined API. Amazon S3 is a storage that makes Web-scale computing easier for developers, whereby any amount of data can be stored and retrieved at any time and from anywhere on the Web.[[5](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_14)] [[6](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_16)]

Figure 3.33: Regions and availability zones in AWS cloud platform. Regions are geographically dispersed to avoid disasters. Availability zones are engineered as autonomous failure zones within regions.

To this end, Amazon EC2 users do not only have the flexibility of choosing among many instance and storage types but have the capability of mapping elastic IP addresses to EC2 instances without a network administrator's help or the need to wait for DNS to propagate new bindings. Elastic IP addresses are static IP addresses but tailored for the dynamicity of the cloud. For example, unlike a traditional static IP address, an elastic IP address enables tolerating an instance failure by programmatically remapping the address to any other healthy instance under the same user account. Thus, elastic IP addresses are associated with user accounts and not EC2 instances. Elastic IP addresses exist until explicitly removed and persist even when accounts have no current running instances.

## **Fault Tolerance**

Amazon EC2 users are capable of placing instances and storing data at multiple locations represented as regions and availability zones. As shown in Figure 1.27, a region can consist of one or many availability zones, and an availability zone can consist of many blade servers. Regions are independent collections of AWS resources that are dispersed geographically to avoid catastrophic disasters. An availability zone is a distinct location in a region designed to act as an autonomous failure zone. Specifically, an availability zone does not share a physical infrastructure with other availability zones, thus limiting failures from transcending its own boundaries. Furthermore, when a failure occurs, automated AWS processes start moving customer traffic away from the affected zone. Consequently, applications that run in more than one availability zone across regions can inherently achieve higher availability and minimize downtime. Amazon EC2 guarantees 99.95% availability per each Region.[[3](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_1)]

Last, EC2 instances that are attached to Amazon EBS volumes can attain improved durability over EC2 instances with local stores (or the so-called "ephemeral storage"). Amazon EBS volumes are replicated automatically in the backend of a single availability zone. Moreover, with Amazon EBS, point-in-time consistent snapshots of EBS volumes can be created and reserved in Amazon S3. Amazon S3 storage is automatically replicated across multiple availability zones, not only in a single availability zone. Amazon S3 helps maintain the durability of users' data by quickly detecting and repairing losses. Amazon S3 is designed to provide 99.999999999% durability and 99.99% availability of data over a given year.[[3](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_1)]A snapshot of an EBS volume can also serve as the starting point for a new EBS volume in case the current one fails. Therefore, with the availability of regions and availability zones, the virtualized environment provided by Xen, and Amazon's EBS and S3 services, Amazon EC2 users can achieve long-term protection, failure isolation, and reliability.

## **Security**

Security in Amazon EC2 is provided at multiple levels. First, as pointed out earlier, EC2 instances are controlled completely by users. Users have full root access, or administrative control, over their instances, accounts, services, and applications. AWS does not have any access rights to user instances and cannot log into their guest OSs.[[9](https://oli.cmu.edu/repository/webcontent/30f6332d80020ca6007987a040d06fbc/_u03_virtualizing_resources_for_cloud/_u03_m06_case_study/x-oli-workbook_page/_u03_m06_2_ec2_properties.xml#_2)]Second, Amazon EC2 provides a complete firewall solution, whereby the default state is to deny all incoming traffic to any user instance. Users must explicitly open ports for specific inbound traffic. Third, API calls to start/stop/terminate instances, alter firewall configurations, and perform other related functions are all signed by the user's Amazon Secret Access Key. Without the Amazon Secret Access Key, API calls on Amazon EC2 instances cannot be made. Fourth, the virtualized environment provided by Xen provides a clear security separation between EC2 instances and the hypervisor as they run at different privileged modes. Fifth, the AWS firewall is placed in the hypervisor between the physical network interface and the virtual interfaces of instances. Hence, because packet requests are all privileged, they must trap to the hypervisor and accordingly pass through the AWS firewall. Consequently, any two communicating instances are treated as separate virtual machines on the Internet, even if they are placed on the same physical machine. Finally, because Amazon EBS volumes can be associated with EC2 instances, their accesses are restricted to the AWS accounts that created the volumes. This indirectly denies all other AWS accounts (and corresponding users) from viewing and accessing the volumes. We note, however, that this does not impact the flexibility of sharing data on the AWS cloud platform. In particular, users can still create Amazon S3 snapshots of their Amazon EBS volumes and share them with other AWS accounts/users. Nevertheless, only the users who own the volumes are allowed to delete or alter EBS snapshots.

[**MY RESPONSE...**](https://oli.cmu.edu/jcourse/webui/resolver/link/resource.do?src=33882d7880020ca600da285eb7cefcbc&dst=u03_m06_feedback2)

### [Summary: Feedback, Comments, and Bugs](https://oli.cmu.edu/jcourse/webui/resolver/link/resource.do?src=33882d7880020ca600da285eb7cefcbc&dst=u03_m06_feedback2)

## **Case Study Summary**

* Many virtualization suites are available from multiple cloud vendors. They have differing architectures and features. Most suites support memory over commitment (which enhances server consolidation) as well as live migration (which allows VMs to be seamlessly moved across physical machines).
* **Amazon's Elastic Compute Cloud (EC2)** is Amazon Web Services' primary IaaS offering.
* **Amazon Machine Images (AMIs)** can be used to create a VM (or **instance**) in EC2. Dozens of AMIs are available for various environments and prebuilt software stacks.
* EC2 instances can be created, launched, stopped, resumed, and terminated as needed. EC2 is believed to be using Xen as the hypervisor. Amazon's software completely automates the process of creating instances, configuring their network and storage, and making them accessible to the user.
* Amazon EC2 offers over 17 different instance types, each of which are configured with differing CPU, memory, local storage, and I/O specifications. EC2 instances typically boot in a minute to a few seconds, depending on the instance type, allowing for elastic scaling of compute resources on demand.
* EC2 instances are flexible to be configured in any way the user requires because the user has full administrator privileges to the OS running in the instance. Amazon's **Elastic Block Store (EBS)**service allows for raw block devices to be provisioned and attached or detached to any instance. Amazon also allows for **elastic IPs** (fixed IP addresses) that can be attached to any instance. All of Amazon's services can be programmatically accessed using APIs, which allows for automated resource control and management.
* EC2 instances can be launched in one of many **regions** and **availability zones,** allowing users to improve the fault tolerance and redundancy of their applications, if they wish to do so.
* EC2 instances are secured through Amazon's own configurable firewall, called **security groups.** The OS running in an instance is exclusively under the user's domain and cannot be accessed by Amazon. Furthermore, Amazon employs strict security practices with public-key authentication to provide access to instances as well as API interfaces to various AWS services.